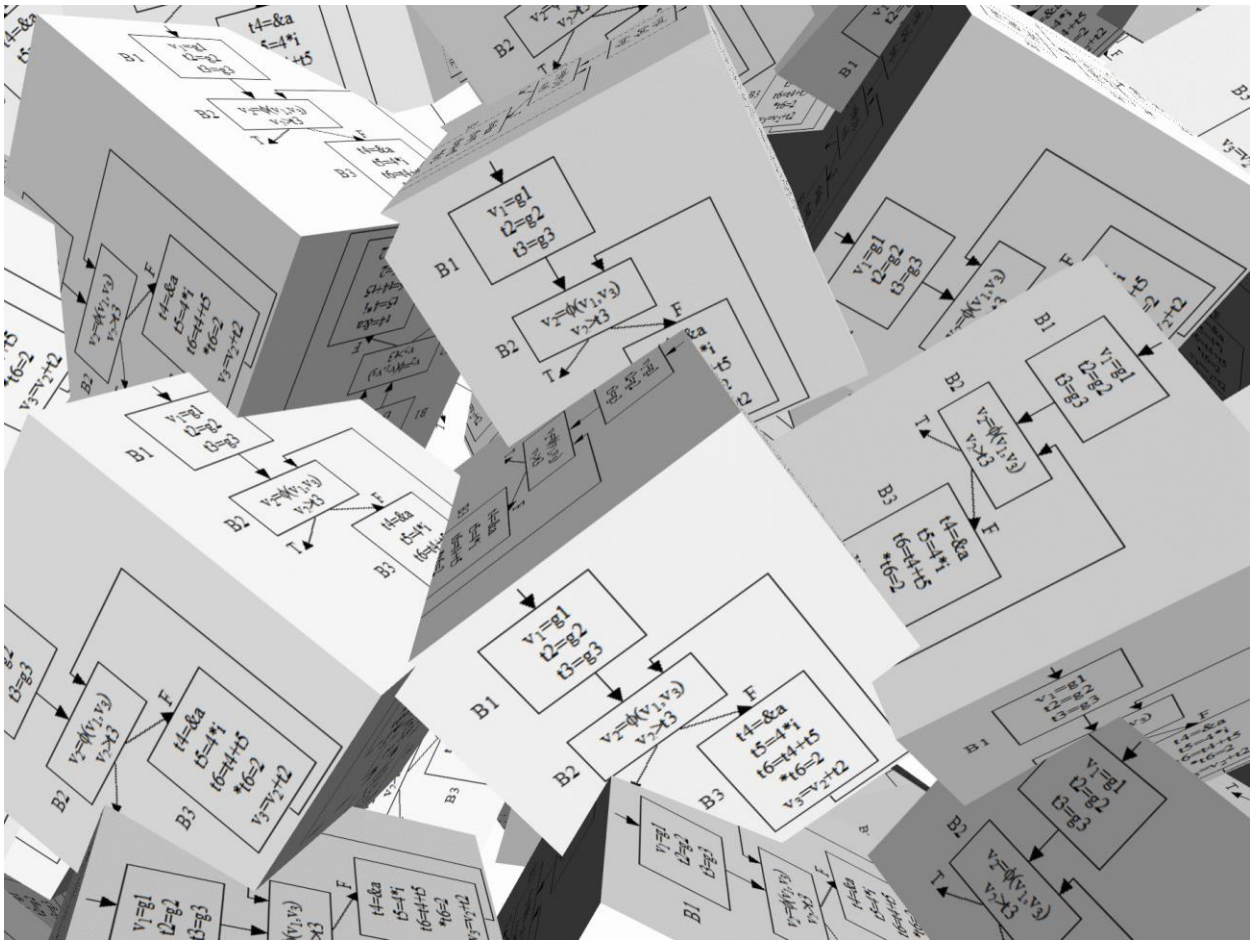
Dragan Bojić, Maja Vukasović

Programski prevodioci 1

verzija 0.9



Beograd, 2022.

Predgovor

Ovaj udžbenik nastao je kao rezultat višegodišnjeg nastavnog rada autora na predmetu Programski prevodioci, na osnovnim akademskim studijama na Elektrotehničkom fakultetu Univerziteta u Beogradu. Udžbenik treba da posluži bržem i potpunijem savladavanju teorijskih i praktičnih osnova iz oblasti konstrukcije prevodilaca za programske jezike i srodnih tehnologija.

Uvodno poglavlje bavi se motivisanjem čitaoca za proučavanje oblasti, uvodi osnovnu terminologiju i daje pregled opšte strukture jednog programskog prevodioca. Takođe se uvodi mikrojava – edukativni programski jezik koji služi kao studija slučaja za ilustraciju izloženih teorijskih koncepata.

Drugo poglavlje bavi se leksičkom analizom, opisuje formalizme konačnih automata, regularnih izraza i opisuje tehnike specifikacije, konverzije iz jednog formalnog modela u drugi, minimizacije i programske realizacije pomenutih formalnih modela.

Poglavlje o sintaksnoj analizi uvodi pojam bezkonteksnih gramatika i opisuje osnovne termine na kojima je zasnovano procesiranje strukture programa na nivou sintakse. Sledi poglavlje o parsiranju od dna ka vrhu koje se bavi konstrukcijom različitih kategorija parsera na principu potiskivanja i svođenja od LR(0), preko SLR(1) do LALR(1) u vidu potisnih automata i razmatra različite praktične aspekte kao što je razrešavanje shift-reduce konflikata i tehnike oporavka od grešaka u parsiranju.

Poglavlje o parsiranju od vrha ka dnu bavi se komplementarnom klasom parsera koji grade sintaksno stablo od korena ka listovima, opisuje tehniku rekurzivnog spusta za realizaciju LL(1) parsera i takođe se bavi praktičnim problemima transformacija gramatika za dovođenje u LL(1) oblik, kao i tehnikom oporavka od grešaka u parsiranju.

Sledeće poglavlje definiše translacione i atributivno-translacione gramatike kao osnovne formalizme na kojem je zasnovano sintaksno-upravljano prevođenje. Kroz pojmove S-atributivnih, L-atributivnih i poljskih translacionih gramatika, formalno se definišu uslovi koje gramatika mora da zadovolji da bi se mogla procesirati odgovarajućim kategorijama parsera, od dna ka vrhu i od vrha ka dnu, Definišu se gramatičke transofrmacije koje pomažu da se gramatika dovede u željeni oblik. Takođe se opisuju tehnike za proširivanje parsera pomenutih kategorija parsera da obavljaju sintaksno-upravljano prevođenje.

Poglavlje o apstraktnim sintaksnim stablima dalje razrađuje koncept semantičkog procesiranja koji je uveden u prethodnom poglavlju. Na konkretnom primeru, ilustrovana je upotreba generatora kompajlera ast-cup. Radi se o originalno razvijenom proširenju standarnog alata cup za generisanje parsera sa semantičkom obradom zasnovanom na apstraktnim sintaksnim stablima. Polazi se od primera standardnog cup generisanog parsera bez semantičke obrade, zatim sa semantičkom obradom u vidu standardnih cup akcija. Zatim se definiše pojam i uloga apstraktnog sintaksnog stabla i opisuje projektni šablon visitor za realizaciju semantičke obrade. Daje se finalni primer sa

semantičkom obradom realizovanom na osnovu apstraktnog sintaksnog stabla automatski generisanog iz gramatičke specifikacije.

Sledeće poglavlje opisuje dizajn i implementaciju tabele simbola u kompajlerima. Definišu se statička i dinamička pravila vidljivosti imena u programu, definiše orgranizacija tabele simbola po pitanjima predstavljana informacija o objektima, opsezima vidljivosti imena, tipovima podataka od ugrađenih skalarnih od kompleksnih objektno-orijentisanih klasa sa nasleđivanjem. Opisuje se jedna moguća realizacija tabele simbola u mikrojavi.

Poglavlje o izvršnom okruženju opisuje izvršnu ogranizaciju memorije, načine pristupa promenljivim, realizaciju pozivanja potprograma i procesorsku podršku višim programskim jezicima na primeru x86 arhitekture.

Sledi poglavlje o arhitekturi mikrojava virtuelne mašine. Ova virtuelna mašina realizuje izvršno okruženje za mikrojava programe i omogućava njihovo interpretiranje. Koncipirana je po ugledu na Java virtuelnu mašinu, ali svedena na najbitnije delove. Daje se pregled memorijskih oblasti, skupa instrukcija bajtkoda sa primerima korišćenja, mehanizma pozivanja statičkih i dinamičkih metoda i strukture bajtkod fajla.

U uvodu je istaknut značaj međukoda kao veze između prednjeg i zadnjeg kraja kompajlera i osnove za sprovođenje optimzacija. Poglavlje o načinima predstavljanja međukoda opisuje različite moguće načine njegove realizacije – sintaksno stablo, aciklički graf, postfiksna notacija, troadresni kod, statička forma jedinstvene dodele. Takođe se definiše pojam osnovnog bloka i grafa kontrole toka programa na nivou osnovnih blokova.

Poglavlje o generisanju koda opisuje ulogu i osnovne probleme generisanja koda, a zatim opisuje primer jednostavnog generatora x86 koda iz troadresnog koda. Na ovo poglavlje nadovezuje se poglavlje o generisanju koda u mikrojavi. Ovo poglavlje opisuje jedan generator koda za mikrojava vrituelnu mašinu kao ciljnu platformu. Detaljno se opisuje prevođenje različitih konstrukcija u mikrojavi, generisanje koda za izraze, pristupanje izvorišnim i odredišnim operandima, generisanje koda za kontrolne konstrukcije, za pozivanje potprograma i obradu deklaracija potprograma.

U prilogu A opisana je sintaksa i semantika jezika mikrojava. U prilogu B opisana je mikrojava virtualna mašina i njen skup instrukcija bajtkoda.

Nadamo se da će ovakva knjiga, pored toga što će biti zvaničan udžbenik na predmetu Programski prevodioci 1, poslužiti i profesionalcima u oblasti razvoja softvera da se upoznaju sa osnovnim konceptima i tehnikama na kojima je zasnovano prevođenje i izvršavanje programa na višim programskim jezicima. Posebnu zahvalnost dugujemo kolegama Jovanu Popoviću, Milošu Gligoriću, Marku Bojoviću i Nemanji Kojiću, nekadašnjim saradnicima u nastavi na kursu iz programskih prevodilaca, za uspešnu saradnju i doprinose nastavnim materijalima kursa koji su predstavljali polaznu osnovu za pisanje ove knjige, kao i prateće softverske infrastrukture.

U Beogradu, decembra 2022. godine *Autori*

Sadržaj

Predgovor 3

Sadržaj 5

Uvod 9

Specifikacija programskog jezika 10

Struktura prevodioca 14

Leksička analiza 23

Deterministički konačni automati 23

Nedeterministički konačni automati 27

Regularni izrazi 38

Konačni procesori 49

Sintaksna analiza 65

Bezkontekstne gramatike 66

Višeznačnost gramatika 71

Opšta klasifikacija gramatika 75

Parsiranje od dna ka vrhu 78

Parsiranje na principu potiskivanja i svođenja 78

Konstrukcija LR(0) parsera 84

Konflikti u SHIFT-REDUCE parsiranju 91

Složenije klase shift-reduce parsera 93

Konstrukcija SLR(1) parsera 96

Konstrukcija LALR(1) parsera 98

Oporavak od grešaka kod parsera od dna ka vrhu 106

Parsiranje od vrha ka dnu 113

Metod rekurzivnog spusta 115

Oporavak od grešaka u metodu rekurzivnog spusta 119

Selekcioni skupovi 120

Transformacije gramatika 122

Sintaksno-upravljano prevođenje 128
Translacione gramatike 128
Atributivno-translacione gramatike 130
Sintaksno-upravljano prevođenje od dna ka vrhu 137
Sintaksno-upravljano prevođenje od vrha ka dnu 142
Apstraktna sintaksna stabla 149
Parser generisan alatom CUP 149
Parser sa semantičkom obradom 153
Konkretno i apstraktno sintaksno stablo 156
Primer kalkulatora sa apstraktnim sintaksnim stablom 158
Tabela simbola 168
Tekući opseg važenja imena 168
Statička pravila vidljivosti imena 169
Dinamička pravila vidljivosti imena 169
Organizacija tabele simbola 170
Pojedinačne tabele za svaki opseg važenja 170
Jedinstvena tabela za sve opsege 171
Objektno-orijentisani jezici 173
Tabela simbola u mikrojavi (objekti, opsezi) 184
Tipovi u mikrojava tabeli simbola 196
Izvršno okruženje 211
Uvod 211
Izvršna organizacija memorije 211
Memorijske klase promenljivih 212
Memorijske klase promenljivih u nekim programskim jezicima 212
Način pristupa skalarnoj (na primer, celobrojnoj) promenljivoj 213
Studija slučaja: Arhitektura procesora Intel 80x86 213
Načini adresiranja kod 8086 214
Primer 215
Pozivanje i povratak iz potprograma 216

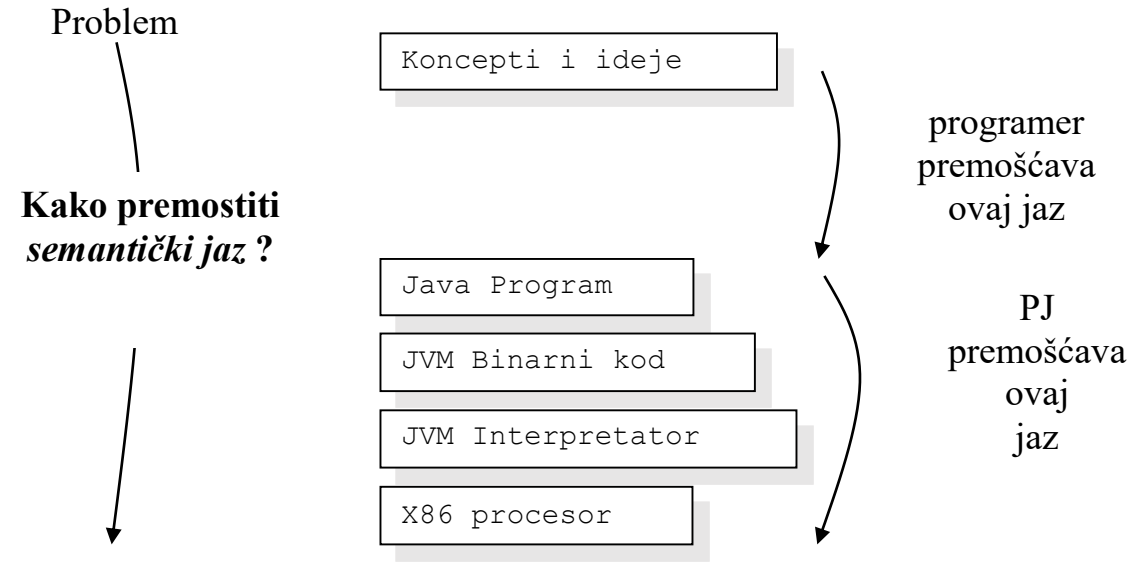
Statičko okruženje za nelokalne promenljive realizovano pristupnim vezama 222
Statičko okruženje realizovano displejima 229
Procesorska podrška višim programskim jezicima 231
Arhitektura mikrojava virtuelne mašine 235
Memorijske oblasti MJVM 236
Pregled skupa instrukcija MJVM 239
Načini predstavljanja međukoda 254
Predstavljanje međukoda putem sintaksnog stabla ili acikličkog grafa 254
Primer 254
Implementacija sintaksnog stabla (grafa) 256
Predstavljanje međukoda putem postfiksne notacije 256
Primer 256
Predstavljanje međukoda putem troadresnog koda 257
Implementacija troadresnih naredbi 260
Predstavljanje međukoda grafom toka kontrole programa na nivou osnovnih blokova 262
Predstavljanje međukoda statičkom formom jedinstvene dodele (engl. Static Single Assignment form – SSA) 263
"Vraćanje" iz SSA forme nazad u običan međukod 265
Primer sa osnovnim blokovima 265
Generisanje koda u kompajleru 267
Ciljna mašina – podskup x86 asemblera 268
Jednostavan algoritam generisanja koda za jedan bazični blok 269
Generisanje mikrojava koda 275
Pristup operandima 276
Prevođenje izraza 277
Generisanje koda za iskaz dodele 281
Skokovi u bajtkodu 282
Generisanje koda za iskaz while 284
Generisanje koda za iskaz if 286
Poziv metoda 288

Obrada deklaracije metoda 289
Iskaz return 292
Prilog A: Specifikacija mikrojave 293
A.1 Opšte osobine jezika 293
A.2 Sintaksa 296
A.3 Semantika 297
A.4 Kontekstni uslovi 298
A.5 Implementaciona ograničenja 301
Dodatak B: Mikrojava virtuelna mašina 302
B.1 Organizacija memorije 302
B.2 Skup instrukcija 303
B.3 Format predmetnog fajla 307
B.4 Runtime greške 307
Literatura 308

Uvod

Od kada postoji računarstvo, postoji i jaz u razumevanju između čoveka i mašine. Način na koji ljudi razmišljaju je jedan, dok s druge strane, kompjuteri rade na sasvim drugačiji način. U krajnjoj liniji kompjuter direktno sledi binarni mašinski kod koji se sastoji od nula i jedinica i to je jedan, da tako kažemo, limitiran skup konkretnih instrukcija. Svaka od njih radi vrlo malo posla, ali veliki niz tih instrukcija kada se na određen način prezentuje kompjuteru, svo moguće ponašanje kompjutera se na taj način postiže.

Kako da programer, koji ima neku ideju šta želi da mu kompjuter radi, na pogodan način isprogramira kompjuter? Ono što karakteriše sadašnje stanje tehnologije to je da deo tog problema mora i dalje čovek, bez nekog formalizma, sam svojom pameću da reši i to je jaz između onoga što čovek ima kao koncepte i ideje do izvornog programa u višem programskom jeziku. Ljudi se školuju i bivaju dobro plaćeni u industriji, da pišu programe na višim programskim jezicima, a onda sledi deo kojim se ovladalo i koji se automatski obavlja od strane računara, da se taj program na višem programskom jeziku preko niza slojeva u softveru spusti na nivo mašine.



Tipično se program na višem programskom jeziku prevodi u jedan oblik koji je opet programski kod ali na nižem nivou, na primer, u slučaju Java programa, to je binarni odnosno bajtkod za Java virtuelnu mašinu. Radi se o jednom zamišljenom softverskom procesoru. Da bi se takav kod izvršavao na fizičkom procesoru potreban je dodatan sloj softvera, interpretator tog Java bajtkoda pod nazivom Java Runtime Environment koji mora biti prisutan. Taj interpretator saopštava fizičkom procesoru, to tipično može da bude neki Intelov procesor, u vidu skupa instrukcija koje Intelov procesor razume. Pošto je Java runtime environment preveden na nivo tog Intelovog mašinskog koda, znači da Intelov procesor izvršava taj runtime environment koji interpretira bajtkod koji je u stvari prevod Java programa koji izražava koncepte i ideje koje programer želi da ostvari na kompjuteru.

Ovaj kurs se najviše bavi prelazom od izvornog programa na višem programskom jeziku (Java) na kod na nižem programskom jeziku koji ili može biti Java bajtkod, ili u nekim drugim varijantama, ne postoje ova dva međusloja nego kompajler direktno prevodi program na višem programskom jeziku na mašinski kod realne mašine kao što je Intelov procesor. Samo taj deo mi proučavamo. Postoje još mnogi drugi delovi sistemskog softvera koji su neophodni da bi se programi izvršavali na savremenim računarima, ali ovaj deo je dosta kompleksan sam po sebi. Veliki kompajleri su jedna od najkompleksnijih komponenata sistemskog softvera tako da zaslužuju poseban kurs. Neke druge teme kao što su formati objektih programa, interpretatori i just-in-time prevodioci, obrađuju se na kursu sistemskog softvera.

Specifikacija programskog jezika

Da bismo konstruisali kompajler potrebno je neki način specificirati programski jezik koji će da, s jedne strane, programer koristi da bi pisao na njemu programe, a s druge strane, za taj isti programski jezik konstruktor kompajlera mora da konstruiše programske alate uključujući i kompajler. Upravo zbog toga što tu postoji više kategorija korisnika koji moraju dobro da razumeju za određeni programski jezik od čega se sastoji i kako funkcioniše, važan pojam je specifikacija programskog jezika. Ona služi tome da odredi u od čega se određeni programski jezik sastoji, kakvo je tačno značenje programa na tom jeziku. Specifikacija jezika, formalno gledano, obuhvata dva aspekta: *sintaksu* (sastav) i *semantiku* (značenje).

Specifikacija sintakse

Sintaksa, odnosno sastav, određuje šta čini jedan program ispravnim odnosno dobro formiranim u matematičkom smislu. Matematičke formule mogu biti ispravne ili neispravne. Ako je formula neispravno napisana, onda je besmislena i dalje se ne ulazi u njenu interpretaciju. Analogno tome, ako jezik zadovolji sintaksna pravila onda se može govoriti o semantici jezika. Ona u suštini znači efekat izvršavanja programa. Semantika je značenje programa odnosno šta se dobija njegovim izvršavanjem. Ljudi su se problemima specifikacije sintakse i semantike bavili otkad postoje viši programski jezici. Više su napredovali u domenu sintakse i tu postoji dobra formalna osnova koja je nastala matematičkom primenom u formi *bezkontekstnih gramatika*. U matematici generalno, formalni jezici su nekakvi skupovi. Jezik je skup sekvenci simbola definisanih na određeni način. Gramatika je pojam koji treba da pomogne da se odredi, u skupu da svih zamislili sekvenci nad nekim simbolima, koje sekvence smatramo ispravnim, a koje ne.

Evo da krenemo od formalne definicije bezkontekstne gramatike: prvo nam je potreban konačan skup takozvanih terminalnih simbola, zatim konačan skup takozvanih neterminalnih simbola od kojih se izdvaja jedan simbol kao takozvani početni simbol. Od tih terminalnih i neterminalnih simbola formiraju se gramatička pravila ili smene. Njih isto ima konačan broj.

Obično se notacija u kojoj se gramatika piše zove BNF, bakus-naurova forma ili bakusova normalna forma. Postoji nekoliko varijacija kako se BNF notacija u raznim knjigama uvodi, ali osnova je ta da svaka smena ima svoju levu i desnu stranu. U jednoj bezkontekstnoj gramatici leva strana je uvek tačno jedan neterminalni simbol, onda postoji nekakav separator leve i desne strane i onda desna strana može biti niz koji je sastavljen od terminalnih i neterminalnih simbola. Smena u BNF notaciji piše se kao:

N ::= α gde je N *neterminal*, α je sekvenca *terminala* i neterminala

N ::= $\alpha \mid \beta \mid ...$ je skraćenica za nekoliko pravila sa levom stranom N.

<N> $\rightarrow \alpha$ Alternativna notacija:

Separator u primeru koji sam ovde naveo je dve dvotačke i znak jednakosti, ali dole je napisano da, alternativno, separator može biti strelica na desno. Pored toga, ako imamo više smena sa istog levom stranom, one naravno mogu da se zapišu svaka na svojoj liniji. Recimo, N se menja sa α, pa N se menja sa β, znači to je druga linija i tako dalje. Ali, moguće je napisati skraćeni oblik gde se samo jednom piše leva strana, evo znači, N se menja sa α, pa se onda napiše uspravna crta, pa se napiše β. Uspravna crta znači da je u pitanju druga smena se isto levom stranom N. Obično postoji neka konvencija po kojoj se razlikuju terminali i neterminali. Konvencija koju u kursu koristimo je da se neterminalni simboli uokvire znacima <>. Znakovi α, β itd. ne odnose se na pojedinačne simbole nego su u stvari oznake da tu dolazi niz simbola. Ove oznake ćemo povremeno koristiti, ali najčešće ćemo pisati po nekoliko konkretnih simbola. Da bi sve bilo jasnije, jedan primer bezkontekstne gramatike.

Primer:

Start ::= Letter

| Start Letter

| Start Digit

Letter ::= **a** | **b** | **c** | **d** | ... | **z**

Digit ::= **0** | **1** | **2** | ... | **9**

Zapis ima pet linija, ali smena ima više. Imamo tri smene koje imaju Start na levoj strani. Znači, prva smena je Start se menja sa Letter. Druga smena je druga linija, samo nije ponovljena leva strana, nego je stavljena uspravna crta. Start se menja sa Start Letter. Treća smena je Start se menja sa Start Digit. Zatim ide niz smena koji imaju Letter na levoj strani. Takvih smena ima (iako je staviljeno na jednom mestu tri tačke, znači to je samo skraćeni zapis), vidimo da u suštini ima 26 smena, koliko ima malih slova engleske abecede. Zatim slede smene sa Digit na levoj strani, odnosno smene za neterminal Digit. Njih ima 10 za svaku cifru. Digit se menja sa nula, Digit se menja sa jedan i tako dalje. Znači ukupno u ovoj gramatici Iako smo iskoristili samo pet linija ima 3 + 26 + 10 znači 39 različitih smena.

Što se tiče gramatičkih simbola, imamo neterminale i terminale. Neterminali su napisani sa velikim slovom, a terminalni simboli su napisani podebljano. Neterminali ove gramatike su Start, Letter i Digit, a terminalni simboli su mala slova i pojedinačne cifre. Ovde nije korišćenje ona konvencija da se neterminali uokviruju sa <>, mada postoje i drugi načini. Kako u opštem slučaju da znamo da li je nešto neterminal ili terminal? Svaki neterminalni simbol mora da se pojavi na levoj strani bar jedne smene. Znači ono što se pojavi na levoj strani smena je neterminal, oni drugi gramatički simboli koji se ne pojave su terminalni simboli.

Koji je početni gramatički simbol (to je uvek neterminal)? Ako ništa nije rečeno, onda postoji konvencija da se uzima uvek leva strana prve smene. Znači to je ovde Start, on je tako i nazvan, tako da se vidi da je to početni neterminal.

Koji jezik definiše ova gramatika? Za sada smo samo definisali pojam gramatike i kako sklopiti gramatička pravila, međutim ništa nismo rekli o tome kako su u stvari gramatike upotrebljavaju da bismo definisali nekakve jezike. O tome ćemo imati posebno lekciju kad pređemo na sintaksu analizu u kompajlerima ima pošto se sintaksa analiza u potpunosti oslanja na ovaj matematički koncert i tamo ćemo vrlo detaljno pojmove raščlaniti. Ovde ćemo se zadovoljiti manje formalnom interpretacijom. U ovom konkretnom slučaju, ako krenemo od Starta, najmanja sekvenca kojom je Start opisan je jedno slovo Letter, to je prva smena (pojedinačna mala slova). Onda šta kaže sledeća smena: duže varijante starta dobijaju se od kraćih tako što na kraju dodamo slovo. A šta kaže treća smena: da na kraće varijante starta može da se doda na kraju cifra. Te smene naravno mogu da se proizvoljan broj puta ponavljaju, što će reći da dužina sekvence opisana startom nije ograničena. Recimo ab1c, na primer, je ispravan Start. Možete primeniti drugu i treću smenu proizvoljan broj puta i međusobno menjati da li ćete primeniti drugu pa treću i tako dalje. To su takozvane rekurzivne smene, pošto vidimo da se Start pojavljuje i na njihovoj levoj i na njihovoj desnoj strani. Takve rekurzivne smene generišu nizove gramatičkih simbola neograničene dužine. Poenta je da ispravan niz simbola mora da se sastoji isključivo iz gramatičkih terminala, odnosno u ovom konkretnom primeru od malih slova i od cifara, znači moraju svi neterminali ako se pojavljuje u nizu biti zamenjeni terminalima.

Sada kada smo izvršili analizu gramatike, možemo zaključiti da Start u stvari opisuje nizove koji su sastoje od malih slova i cifara i koji počinju slovom, znači ne mogu početi cifrom, jer one duže nizove smo dobijali tako što na kraju dodajemo slovo ili cifru a uvek mora početi slovom. U suštini, ova gramatika bi mogla biti primer kako se opisuje sintaksa nekog imena identifikatora, recimo imena promenljive u nekom programskom jeziku. To je samo jedan mali primer, ali smo videli da jedan mali deo sintakse programskog jezika, konkretno način kako se imenuju promenljive može da se izrazi gramatikom.

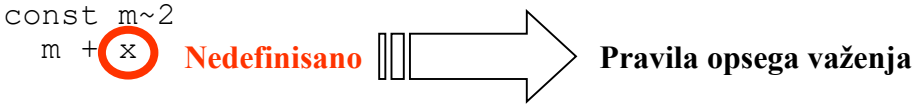
Specifikacija semantike

Za drugi deo specifikacije jezika rekli smo da opisuje značenje programa pri njegovom izvršavanju. Preciznije gledano, značenje programa ne mora da se ustanovljava isključivo izvršavanjem programa, nego deo značenja programa može da se ustanovi i bez njegovog izvršavanja. Evo sada ćemo videti konkretan primer: da bi ste napisali pravilan program koji kad ga propustite kroz kompajler neće prijaviti nikakvu grešku, deo pravila može biti proveren sintaksom jezika znači videli smo da postoji formalan način preko gramatika da se opiše koji su programi sintaksno ispravni. Znači, ako programer napiše sintaksno neispravan program, kompajler će mu prijaviti sintaksnu grešku. Međutim, neki deo značenja programa koji kompajler može da proveri ide izvan okvira bezkontektsnih gramatika.

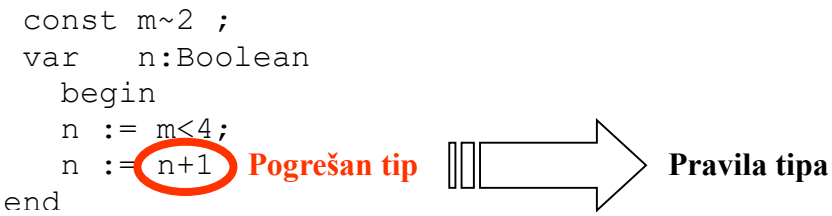
Na primer, mnogi programski jezici zahtevaju da se tipovi promenljivih deklarišu pre upotrebe. Neki programski jezici zahtevaju da deklarišete x znači da u deklaraciji navedete kog je tipa. Ako samo upotrebite x a prethodno nema deklaracije za x, onda će kompajler da se pobuni da x nije definisan -- znači da su narušena takozvana pravila opsega važenja.

U drugom primeru, imamo program koji ima deklarisanu neku konstantu m, ima neku promenljivu n logičkog tipa (boolean). U izvršnom delu programa imamo neku dodelu, znači nekakav logički izraz m < 4 koji, pošto je u pitanju relacioni operator, može da proizvede vrednost koja je logičkog tipa tačno ili netačno. Posle toga imamo dodelu n = n + 1. Tipično je da se definiše da je to operacija sabiranja, ali mora da se definiše koje su tipovi operanada i koji je tip rezultata. Znači, neki programski jezik bi mogao da ima celobrojno sabiranje, koje zahteva da operandi budu celi brojevi i rezultat će onda biti zbir tih operanada kao ceo broj. Ovde promenljiva n je logičkog tipa i to nije celobrojan tip i u principu operacija sabiranja ne mora biti definisana za logički tip podataka. Znači, narušeno je pravilo tipa.

Primer 1:



Primer 2:



Ova pravila opsega važenja, odnosno da li je promenljiva definisana pre upotrebe, i pravila tipa, odnosno da li smo ispravno upotrebili operator, spadaju u semantička pravila (značenje) pojedinih elemenata programa i ova vrsta pravila može se formalno definisati i proveriti bez izvršavanja programa. Kompajler koji ne izvršava program nego samo procesira njegov zapis, može u potpunosti proveriti ovu vrstu semantike i zato se ona zove *statička*.

Izvršna semantika

Takozvana izvršna (runtime) semantika je deo značenja programa koji može da se proveri jedino izvršavanjem programa. Postavlja se pitanje kako dati specifikacije izvršne semantike programa? Za računarske primena i dalje se ograničavamo na opis semantike programa neformalno da kažem na običnom govornog jeziku. Na primer, iskazi programa -- to su obično razne izvršne naredbe -- se izvršavaju i proizvode različite bočne efekte, kao što su promena vrednosti promenljivih ili obavljanje ulazno/izlaznih operacija. Izrazi se izračunavaju I daju vrednost i mogu, ali ne moraju imati bočne efekte. Takođe i delovi programa za koje biste očekivali da kompajler završi sa njima i da u principu u run time-u ne igraju neku ulogu ipak imaju neku svoju izvršnu semantiku. Recimo, različite deklaracije se elaboriraju da proizvedu vezivanje imena za promenljivu u memoriji.

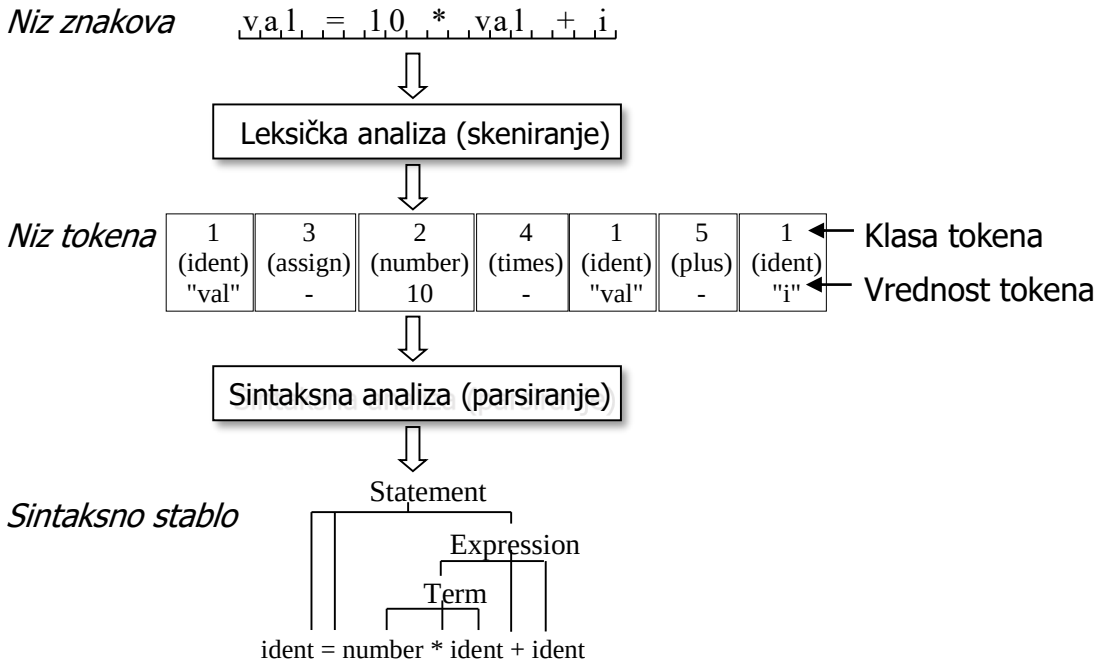
Konkretan primer vezan za iskaz dodele: V := E izvršava se na sledeći način: prvo se izračuna izraz E da dobije neku vrednost v, potom se vrednost v dodeljuje promenljivoj V.

Ili, recimo, iskaz sekvence C;C2, pri čemu su C1 i C2 neki iskazi, izvršava sa na sledeći način: prvo se izvrši iskaz C1, potom se izvrši iskaz C2.

Rekli smo da se deklaracija elaborira da proizvede vezivanje takođe može imati sporedni efekt dodele memorije promenljivoj. Na primer, neka konkretna deklaracije var I:T se elaborira vezujući i za novo alociranu promenljivu čija vrednost nije definisana. Promenljiva će biti dealocirana na izlazu iz bloka koji sadrži pomenutu deklaraciju. To je tipična semantika lokalne promenljive.

Struktura prevodioca

Završili smo deo predavanja koji se odnosio na objašnjenje specifikacije programskog jezika, a sad sledi deo davanja koji treba da nam objasni, na visokom nivou, strukturu jednog tipičnog kompajlera, što u krajnjoj liniji, određuje i strukturu našeg kursa, pošto ćemo se mi pojedinim funkcionalnim celinama kompajlera baviti čitav ostatak kursa. Funkcionalno gledano, prevodilac ima niz faza u svom funkcionisanju koje slede, logički gledano, jedna drugu u smislu da izlaz prethodne faze predstavlja ulaz sledeće faze.



Na samom početku rada prevodioca prva obrada koja se radi je takozvana leksička analiza ili skeniranje programa ili treći naziv je tokenizacija programa, nazvana po rezultatu te prve faze. Na ulaz kompajera tipično dolazi izvorni programski kod u vidu tekstualnog fajla (niza znakova) i uloga skenera je da pretvori taj niz znakova u niz binarnih zapisa koji se najčešće zovu tokeni. Ti binarni zapisi treba da odgovaraju pojedinim podnizovima ulaznog niza znakova, koji čine jednu logičku celinu.

Prva logička celina u ovom primeru koji je dat na slajdu je ime promenljive val. Znači ta tri slova kada se logički objedine čine ime promenljive. Imenu promenljive može da odgovara jedan token. Token možete definisati kao binarni zapis od dve komponente (dva polja): klasa i vrednost. Klasa tokena daje opšti tip tokena, a vrednost tokena onda može dodatno da odredi konkretan token u toj klasi, da ga razlikuje od svih drugih instanci te iste klase. Recimo, imena promenljiva, funkcija i tako dalje u programu, to je tipično klasa identifikator. Drugo polje iskorišćeno je da zapamti sam

ovaj string val tako da se zna o kojem konkretnom identifikatoru je reč. Sledeći token koji se prepoznaje u ovom konkretnom ulazu je operator dodele vrednosti =. Znači specijalni znak = može da ima svoj token klase *assign* i klasa potpuno određuje taj token. Nije potrebno da se koristi vrednost pa je stavljena samo crtica. Znači kod nekih vrsta tokena koristi se samo klasa, a kod neki drugih vrsta tokena i klasa i vrednost. Onda, sledeći token u ovom primeru je token celobrojne konstante koje je klase *number*. Znači znakovi 1 i 0 se interpretiraju kao konstanta 10 i znači pamti se klasa tokena *number* i pamti se sama znači vrednost konstante 10. I tako dalje vidimo isprocesira se znači ulazni niz znakova u niz tokena i u stvari ostatak kompajlera više nikad ne vidi originalni ulazni niz znakova nego on barata isključivo sa ovim nizom tokena.

Izlaz iz leksičke analize tipično ide u sintaksni analizator, odnosno parser. To je pojam vezan specifično za kompajlere i sintaksu analizu. Znači ulaz parsera je ovaj niz tokena, a izraz parsera je sintaksno stablo programa. Parsiranje je zasnovanu upravo na pojmu bezkontekstnih gramatika i u sledećoj lekciji ćemo videti vrlo jasnu vezu između gramatika, tokena i sintaksnog stabla. Neformalno, pojedinačni tokeni odgovaraju pojedinačnim terminalnim simbolima u bezkontekstnoj gramatici, a zadatak parsera je da iz niza terminalih simbola u stvari rekonstruiše kako, na bazi gramatike, počevši od startnog simbola, dobiti upravo taj niz tokena. Znači parser na neki način radi suprotno od ranijeg primera. Mi smo u ranijem primeru krenuli od startnog neterminala da bismo došli do nekog niza tokena. ovde nam je u stvari zadat u početku konkretan niz tokena za koji smatramo da je sintaksno ispravan i moramo rekonstruisati kako dobiti njega iz gramatike,

Sintaksno stablo je jedna grafička reprezentacija upravo tog postupka kako iz ulaznog neterminala gramatike dobiti niz tokena. Ovo što vidimo u korenu sintaksnog stabla *statement* u stvari znači da u gramatici postoji *statement* kao neterminal i ako je to koren stabla ovom primeru, to bi onda bio i početni neterminal gramatike. Statement ima u ovim primeru tri naslednika: ident, jednako i expression to u suštini znači da u gramatici postoji smena koja ima na levoj strani statement a na desnoj strani ident, jednako i expression. Ident je već terminal, dok je expression neterminal, znači on dalje mora da se rasčlanjuje i tako dalje. O svemu tome pričaćemo u poglavlju o sintaksnoj analizi, ali znači krajnji efekat sintaksne analize je da za naš program dobijamo kompletno sintaksno stablo programa gde je u listovima stabla u stvari početni niz ovih tokena odnosno terminalinih simbola.

Nad sintaksnim stablom sprovodi se semantička analiza. Semantička analiza može da se shvati kao provera one statičke semantike, znači raznih pravila koja sintaksi analizator nije mogao. U suštini, semantičku analizu treba shvatiti kao neku vrstu anotacije stabla dodatnim informacijama: na primer, promenljivoj kompajler može već dodeliti neku memorijsku lokaciju. Semantička analiza sem ovoga sintaksnog stabla može da proizvede i dodatne strukture podataka.

Najvažnija struktura podataka je tabela simbola kompajlera. Tabela simbola je jedna kolekcija podataka, kao neka interna baza podataka kompajlera o identifikatorima i drugim celinama od interesa u programu. Na primer, za svako korisničko ime (promenljive u funkciji i tako dalje), evidentira se to ime i evidentiraju se različiti drugi atributi, koji je tip tog imena znači da li je recimo promenljiva, kog tipa, zatim naknade faze kompajlera dodeljuje joj memorijsku adresu i tako dalje.

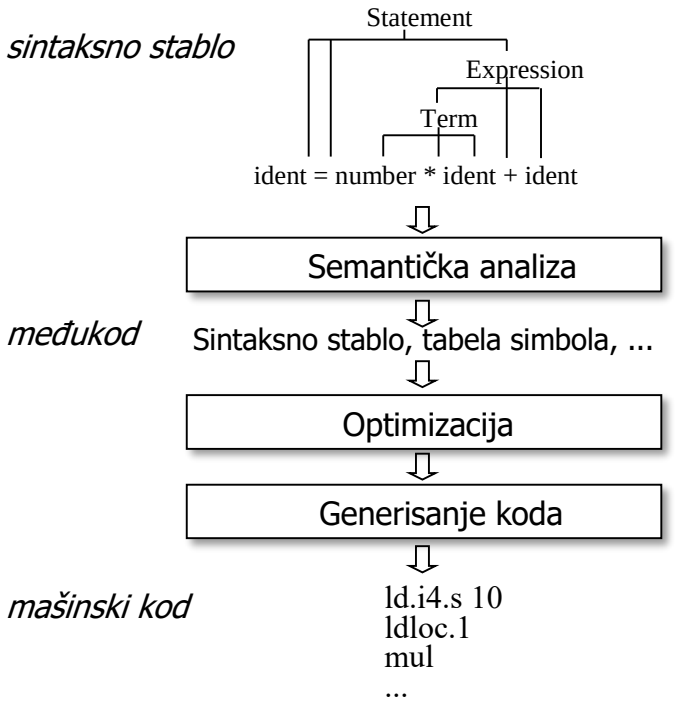


Tabela simbole je tipično jedna asocijativna struktura. Prvi put kada naiđemo na neko ime, to je tipično u nekoj deklaraciji, ubeležimi se novi zapis u tabeli simbola, a zatim kasnije kad na to ime ponovo kompajler naiđe, znači kada se ime upotrebljava, onda se izvrši pretraga po ključu u tabeli simbola da bi se na bazi imena našli ti drugi atributi. Naravno, to nije baš sasvim tako jednostavno, znači nisu to ključevi u nekom linearnom prostoru ključeva ali videćemo kasnije po kojem pravilu se vrši pretraga. Tabeli simbola biće posvećena posebna lekcija u kursu i neke karakteristike jezika određuju koje strukture podataka izabrati za tabelu simbola i uopšte interfejs tabele simbola prema ostatku kompajlera. Znači videćemo kako funkcionišu tabele simbola u modernim kompajlerima.

Izlaz iz semantičke analize je to anotirano sintaksno stablo i te druge strukture podataka kao što je tabela simbola. Ovo je jedna važna tačka u funkcionisanju kompajlera koja odvaja takozvanu prednju stranu kompajlera. Znači sve ove funkcije o kojima smo do sada pričali čine takozvani front end ili prednji deo kompajlera, a njihov rezultati ide u back-end to jest zadnji deo kompajlera. Zadnji deo kompajlera tipično čine optimizator i generator koda. Zadnji deo kompajlera radi nad rezultatima koje ostavi prednji kraj kompajlera. U ovom konkretnom primeru kažem ja sam stavio da je u pitanju struktura tipa stabla plus dodatne informacije (tabela simbola).

Međutim, opšti pojam koji se koristi da opiše rezultate je takozvani međukod, odnosno Intermediate representation, IR u kompajlerskoj terminologiji. Taj međukod u ovom konkretnom primeru podrazumevano je oblika ovoga stabla. U principu to ne mora biti potpuno identično stablo kao što je proizveo parser ili semantički analizator nego može doći do prepisivanja: recimo parser može da formira takozvano konkretno stablo izvođenja (konkretno sintaksno stablo) sa svim mogućim neterminalima, što znači sa gomilom čvorova, a da se onda to stablo prepiše u jedno jednostavnije stablo koje samo ostavlja elemente ili čvorove koji su važni za semantiku u programu

i koje se zove apstraktno sintaksno stablo koje mora posebno da se definiše za programski jezik tako da to čak ne mora biti potpuno identično stablu izvođenja po strukturi.

Drugo, međukod uopšte ne mora biti oblika stabla. Forme međukoda u savremenim kompajlerima su vrlo raznorodne. Postoje još mnogi drugi oblici međukoda. Oblik međukoda koji se zove troadresni kod i koji liči na nekakav apstraktni mašinski kod sličan recimo Intelovom asemlerskom kodu ali neke stvari sa apstrahovane, ili, na primer, takođe može da liči na mašinski kod za stek mašinu (takozvana postfiksna notacija). Postoje neke posebne forme međukoda kao što je statička forma jedinstvene dodele i tako dalje. U posebnoj lekciji ćemo pobrojati i ukratko objasniti različite forme međukoda kod različitih kompajlera. Iz jedne forme međukoda kompajler po potrebi može da transformiše program u neku drugu formu samo je naravno važno da se semantika programa pri tome se ne promeni.

Savremeni kompajleri mogu više puta da rade te transformacije iz jednog oblika međukoda u drugi recimo tokom optimizacija. Šta je motiv tih transformacija? To što sledeća obrada na nekoj posebnoj formi međukoda može lakše da se sprovede nego na onoj prethodnoj formi međukoda. Taj deo je u suštini kompleksniji nego što je prikazano na ovoj slici, ali poenta je da, bez obzira koja je forma tog međukoda, sve te informacije sada idu u taj zadnji kraj kompajlera.

Optimizacija je u suštini proces transformacije tog međukoda. Ciljevi optimizatora mogu biti raznorodni. Jedan cilj može biti dobijanje programa sa što boljim performansama, znači koji se što brže izvršava. Drugi cilj optimizatora može biti dobijanje programa koji troši što manje memorije -- minimizacija korišćenja operativne memorije. U savremenim uslovima cilj optimizacije može biti i da se postigne da ukupno trošenje električne energije pri izvršavanju programa bude što manje. Ovo je od značaja u različitim ugrađenim sistemima, mobilnim sistema koji rade na baterije i tako dalje. Znači uvek postoji nekakav cilj optimizacije.

Optimizacija može na različitim nivoima i u različitim fazama kompajlera u najopštijem slučaju da se sprovede. U nekoj užoj definiciji ona je na ovom mestu gde je nactano, posle semantičke analize, a pre generisanja koda. Ulaz optimizatora je međukod i njegov izlaz je međukod. Znači on ne menja nivo apstrakcije izlaza u odnosu na ulaz, ali vrši transformaciju. Šta znači transformaciju međukoda? Može da se promeni recimo redosled naredbi u međukodu ili može da promeni međukod, znači da izabere neke ekvivalentne naredbe koje su optimalnije i tako dalje. Taj transformisani međukod konačno ide u generator koda.

Uloga generatora koda je da transformiše međukod u konkretan mašinski kod. Pri čemu generator koda tu rešava neke specifične probleme vezane za tu transformaciju, na primer jedan od bitnih problema u generatoru koda je izbor mašinskih instrukcija. Ista obrada može da se izvrši na različite načine i u zavisnosti od izbora kod konkretnog procesora neke instrukcije su efikasnije neke su manje efikasne. Recimo, za uvećavanje neke vrednosti za jedan kod Intelovih procesora postoji posebna naredba inkrementa koja se izvršava određen broj procesorskih ciklusa, ili može da se sprovede direktno sabiranje sa konstantnom jedan, ali to je kompleksnija naredba koja zahteva dva operanda da se dovedu -- vrednost koja se uvećava i konstanta jedan i da se sve provuče kroz ALU, znači to je manje efikasan način. To je jedna stvar koju rešava generator koda -- izbor instrukcija.

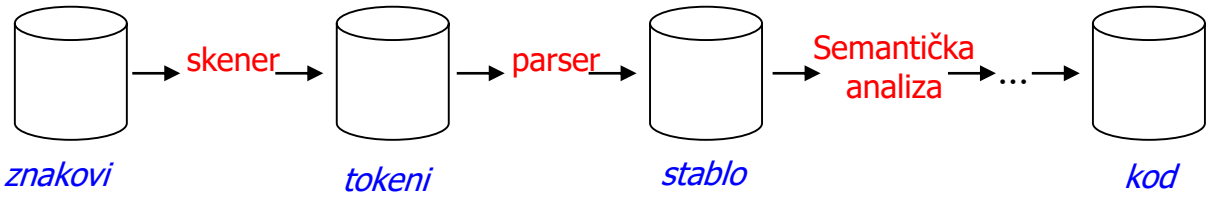
Drugi jako važan problem je dodela procesorskih registara promenljivama. Procesorski registri su tipično oskudan resurs, nema ih dovoljno, ima puno više promenljivih u programu i vrednosti nego procesorskih registara. Kompajler mora na jedan inteligentan način da obezbedi da u određenom delu programa, vrednosti određenih promenljivih, znači onih sa kojima se najviše barata, se čuvaju u registrima koliko je potrebno. Kada ta vrednost više nije potrebna, onda se ona izbaci iz registra, znači sačuva u memoriji, a u registru se čuva nešto drugo.

Kada se razreše svi ti problemi generisanja koda, konačno na izlazu iz generatora kada dobija se mašinski kod. Taj mašinski kod može biti ili za konkretan fizički procesor, na primer neki Intelov ili neki Arm ili neki PowerPC znači realan procesor koji postoji u hardveru, ili to može biti mašinski kod nekog zamišljenog softverskog procesora koji se najčešće zove virtuelna mašina. Poznati primeri virtuelnih mašina su Java virtuelna mašina, koja ima arhitekturu stek mašine. Ona koristi Java bajtkod čiji su primeri naredni dati na slici.

Taj mašinski kod koji kompajler generiše je njegov glavni izlaz, ali on generiše u suštini izlazni format koji je jedan binarni fajl koji se sastoji od više vrsta zapisa i on je obično nekog od onih objektnih formata kao što smo pominjali u sistemskom softveru znači to mogu biti recimo ELF za Linux sisteme, ili Portable Executable (PE) za Windows sisteme i tako dalje. Znači pored čistog binarnog mašinskog koda, tu postoje neke dodatne informacije, obično tabela simbola i neke relokacione informacije i tako dalje. O tome znači podsetite se iz gradiva sistemskog softvera, znači taj deo neće nam biti u fokusu ovog kursa jer pretpostavljamo da smo to obradili u posebnom kursu.

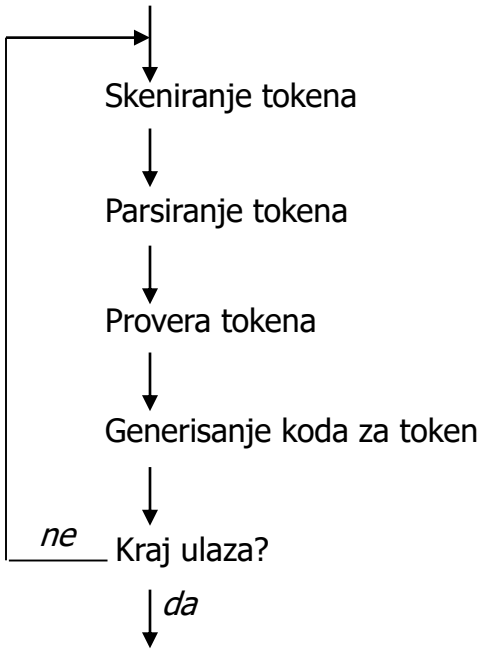
Jednoprolazni i višeprolazni prevodioci

Sada kada smo sagledali koje funkcije sve imaju kompajleri, mi smo rekli u principu niz tih funkcija je sekvencijalno povezan, znači izlaz iz jedne funkcije je ujedno ulaz za drugu funkciju. Međutim, zanimljivo je da u konkretnoj realizaciji kompajlera mogu da se sagledaju nekoliko načina kako povezati te faze. Jedan od načina je ovaj kod višeprolaznih prevodilaca. Intuitivno najjednostavniji za shvatanje je način da se faze prosto izvršavaju sekvencijalno. Znači obrada nad programom u jednoj fazi se potpuno sprovede, pa se tek onda prelazi na sledeću fazu, koja se opet potpuno sprovede, pa onda njen rezultat ide u sledeću fazu i tako dalje.



Kompajler koji je tako napravljen, na slici je njegova konceptualna šema, često koristi zapisivanje rezultata pojedinih faza u nekim privremenim radnim fajlovima na disku pa sledeća faza čita taj prethodni fajl i tako dalje. To je nekad recimo bilo važno zato što je operativna memorija u jednom delu istorije računarstva bila redak resurs, tako da su dobri kompajleri tako bili napravljeni da mogu jako velik program da isprocesiraju sa malo memorije, tako da nisu smeli kompletan rezultat neke faze programa da drže potpuno u memoriji, da ne bi ograničili veličinu programa koje mogu da obrade. Međutim, to je sad manje ovaj bitno, ali i dalje može nekad biti zahtevano da se cela

faza obavi sa rezultatom u operativnoj memoriji pre nego što se pređe na sledeću fazu, ako je na primer jezik kompleksan ili ako postoje neke druge potrebe.



Pored ovakvog načina organizacije faza postoji (to je ovaj prethodni slajd) takozvani način isprepletenog funkcionisanja faza, odnosno nekakvog softverskog pajplajna. Takvi prevodioci se zovu jednoprolazni. Pojam prolaza se odnosi na kompletno izvršavanje neke faze, pa smo u onom prethodnom slučaju imali višeprolazne prevodioce, jer smo imali kompletno izvršavanje jedne faze pre prelaska na sledeću. Međutim, jednoprolazni prevodioci ne završavaju nijednu fazu potpuno pre nego što pređu na sledeću fazu.

Kako kod njih to u stvari funkcioniše? Na primer, kod jednoprolaznog provodioca može da se desi skeniranje jednog ili malog broja tokena. Zatim se ti tokeni od strane parsera parsiraju, recimo formira se za jedan iskaz X = 1, znači, samo se taj deo stabla prepozna. Zatim se uradi semantička provera i generisanje koda za taj jedan iskaz X = 1 i onda se ide na skeniranje sledećeg tokena, parsiranje sledećeg iskaza i tako dalje. U suštini, reč je o jednom softverskom pajplajnu, gde, malo po malo, program na ulazu biva skeniran token po token, a na izlazu se već generiše, naredba po naredba, mašinski kod. Mada ovaj način deluje manje intuitivno za objašnjavanje, ovo je u stvari lakši način kako napraviti kompajler, uključujući kompajler koji se pravi kao deo projekta na kursu.

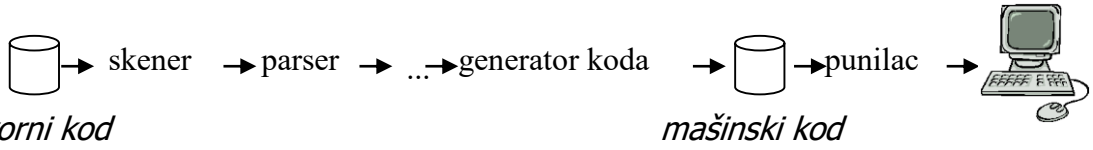
Iz praktičnih razloga, zgodno je napraviti dvodelni prevodilac, da prednji kraj (engl. Front-end) bude jedna celina, to jest skeniranje, parsiranje, semantička analiza, znači da prednji kraj izprocesira kompletno program u kompletan međukod, a onda da se procesiranje prepusti zadnjem kraju (back end-u) kompajlera.

Šta u praksi time može da se postigne? Zamislimo da neka softverska firma koja treba da podržava razne programske jezike za različite mašinske arhitekture. Ako bi ta firma razvijala kompajlere

kao monolitne celine, znači prednji i zadnji kraj zajedno, onda za tri programska jezika i tri neke procesorske arhitekture treba napraviti ukupno devet različitih kompajlera. Međutim, ako se posao podeli posmatrajući posebno prednje krajeve kompajlera, posebno zadnje krajeve kompajlera, a da se definiše standardni međukod, standardizovan u tom smislu da ne zavisi ni od ulaznog višeg programskog jezika, niti od procesorske arhitekture onda je moguće napisati za bilo koji od potrebnih programskih jezika prednji kraj koji generiše taj standardni međukod, a odvojeno od toga pisati zadnje krajeve koji taj standardni međukod prevode na neku konkretnu procesorsku arhitekturu. Kasnijim spajanjem bilo kojeg prednjeg kraja sa bilo kojim realizovanim zadnjim krajem dobijaju se konkretni kompajleri. Time se smanjuje posao, znači izvršena je višestruka upotreba koda.

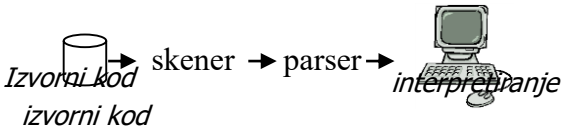
Prevodioci i interpretatori

Pojam klasičnog kompajlera odnosno prevodioca odgovara "unmanaged" C ili C++ kompajleru u Microsoft terminologiji (starijim kompajlerima je to bila jedina varijanta), a to je da se program na višem programskom jeziku prevede direktno na konkretan mašinski kod ciljne fizičke mašine, kao na sledećoj slici.



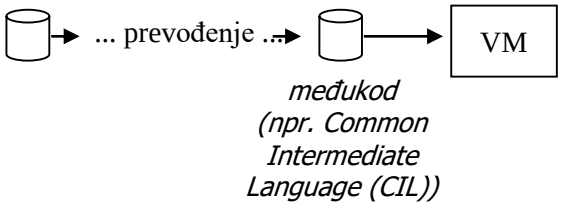
U novije vreme pojavila se potreba, opet iz praktičnih razloga, da se ne prave ti klasični kompajleri, nego da se programski jezici definišu na način da im se prvo obezbedi interpretacija, znači svi ovi moderniji programski jezici su svoj osnovi skript jezici odnosno jezici sa interpretatorom. Šta znači interpretator? Ti pojmovi se detaljno objašnjavaju na kursu sistemskog softvera. Interpretator je u suštini sistemski softver koji prihvata izvorni kod na višem programskom jeziku i onda izvršava njegovu delimičnu obradu koja obuhvata recimo skeniranje i parsiranje koda, ali zatim programu je potrebna podrška u vreme izvršavanja, kao što je to Java runtime environment. Taj kod za podršku izvorni program tretira kao podatak I, prateći instrukcije u izvornom program, radi neka interna ažuriranja programskog stanja, znači menja memoriju, ažurira promenljive i na taj način izvršava program. Znači za interpreter je izvorni program u suštini samo jedan podatak.

Direktna interpretacija izvornog koda



- Iskazi u petlji se parsiraju nanovo i nanovo

Druga varijanta: interpretacija međukoda



- Izvorni kod prevodi se u kod *virtuelne mašine* (VM)
- VM interpretira kod simulirajući fizičku mašinu

Interpretatore je često mnogo lakše napraviti, samim tim što imaju manje faza i tako dalje, nego prevodioce. Glavna mana interpretiranja je što je više redova, recimo tipično dva reda veličine, sporije izvršavanje takvog programa pod interpretatorom, nego istog programa koji je preveden u mašinski kod ciljne mašine. Zašto je onda interpretacija privlačna i zašto za sve više modernih jezika kada se definiše njihova semantika, prvo se pravi interpretator? Prosto zato što to omogućava samim jezicima neke mehanizme koje nije lako realizovati kod klasičnih kompajlera. Na primer, tipični skript jezici su slabo tipizirani. Možete promenljivu da iskoristite i direktnom dodelom vrednosti joj odredite i tip. To se prenosi i na objektno orijentisano programiranje. Koncert duck typing znači da je tip neke promenljive određen upravo tom vrednošću koja je dodeljena. Takvi jezici su tipično, ako ne zahtevaju deklaracije, koncizniji za pisanje. Međutim njihova mana je u tome što se većina grešaka onda ne detektuje pre izvršavanja programa, nego u toku izvršavanja programa. U svakom slučaju, osobine samog jezika su određene time da li je primarno izmišljen za interpretaciju ili klasično prevođenje.

Koncept interpretacije ima svojih privlačnih strana za savremenu upotrebu, na primer kod mobilnih uređaja. Kod jednog da kažemo provajdera tehnologije kao što je Google, mobilni uređaji mogu biti sa različitim procesorima i različitim fizičkim arhitekturama, ali ako se definiše koncept kao kod jave write once run anywhere, znači ako se definiše standardno izvršno okruženje koje ne zavisi od fizičke arhitekture, onda program ne mora svaki put da bi se izvršavao da se prevodi potpuno na konkretnu fizičku arhitekturu.

Mana interpretacije -- sporost -- pokušava se zaobići ili rešiti time ti što su delimično neke faze ipak odrađuju pre izvršavanja programa, tako da recimo, program, umesto da se direktno interpretira na nivou sintaksnog stabla, može da se prvo prevede u neku vrstu međukoda u vidu apstraktnog mašinskog koda, pa da se taj međukod interpretira. Ili, što još češći slučaj u savremenim uslovima, da se nad međukodom umesto čiste interpretacije, vrši just-in-time kompilacija, ili ahead of time kompilacija. To su dve slične varijante. Na primer, koncept just-in-time kompajlera koji rade na nivou metoda je sledeći. U trenutku izvršavanja programa kada treba pozvati neki metod, on je u tom trenutku prisutan u obliku međukoda (recimo Java bajtkoda) i onda se pusti jedan mini kompajler koji taj bajtkod recimo prevede na Intelov mašinski kod, zapamti taj prevod i onda tek pozove taj metod. Naravno time je prvo izvršavanje tog metoda još sporije jer mora da se prvo prevede, pa onda tek može da se izvršava. Ali, ako se recimo takav metod 1.000 ili 10.000 puta repetitivno u nekoj petlji poziva, onda će sva ta druga izvršavanja biti mnogo brža. Znači ako se metod prevede na nivo fizičkog koda ciljne mašine, onda u principu to just-in-time kompajliranje ne može da dostigne brzinu klasičnog kompajliranog koda, ali može vrlo blizu da se približi. Ima dosta pozitivnih momenata kod tog pristupa, tako da je to vrlo popularan pristup. To je ova treća varijanta na slici. Praktično cela priča koja je do sada ispričana o strukturi kompajlera, važi u ovom delu cele slike od izvornog koda do međukoda. Znači to je manje-više taj klasičan kompajler, samo on ne generiše kod za fizičku mašinu nego kod za virtuelnu mašinu.

Programski jezik mikrojava

Knjiga sadrži studiju slučaja konstrukcije kompletnog kompajlera za jedan programski jezik koji se zove mikrojava i koji je posebno osmišljen za potrebe ovog kursa kao jedan mali podskup jave.

Mikrojavu i propratno softversko okruženje je inicijalno osmislio profesor Mössenböck [1] u sklopu svog kursa o konstrukciji kompajlera, ali je u tokom vremena na Elektrotehničkom fakultetu mikrojava semantički bitno proširena u pravi objektno-orijentisani jezik sa nasleđivanjem i polimorfizmom.

Programi na mikrojavi sintaksno liče na programe na javi i ima dosta stvari koje ima i prava java uključujući klase, nasleđivanje klasa i polimorfizam, znači sve što je zanimljivo sa stanovišta da se vidi kako to obrađuje kompajler i kako se u runtime-u ti mehanizmi, recimo objektno orijentisano programiranje, realizuju. Ali, s druge strane, mnogo drugih stvari je izostavljeno da bi kompajler mogao biti što manji. Svejedno, konstrukcija kompletnog kompajlera za mikrojavu spade u velike studentske projekte. Da bi se to lakše realizovalo, u kursu je napravljen niz olakšica: kao prvo student ne pišu bukvalno svaku liniju koda nego neke funkcionalne celine, na primer tabelu simbola, generator koda dobijaju praktično 90% ili 100% završene. Drugo, na raspolaganju su različiti primeri kako se određene stvari realizuju tako da je zadatak studenta da iz svih raspoloćivih materijala koje dobija, prilagođavanjem delova koda, sastavi sopstveni kompajler. To je suština praktičnog projekta na kursu. Na sajtu kursa iz Programskih prevodilaca 1 na Elektretehničkom fakultetu u Beogradu http://ir4pp1.etf.rs/ moguće je pronaći online razvojno okruženje za prevođenje i izvršavanje mikrojava programa, kao i sve potrebne biblioteke i alate za razvoj mikrojava kompajlera.

Leksička analiza

Ova lekcija objasniće leksičku analizu kod programskih prevodilaca. U uvodnom delu, podsetićemo se uloge leksičkog analizatora -- skenera kao funkcionalne celine kompajlera. Njegova glavna uloga je da ostatku kompajlera, konkretno sintaksom analizatoru, isporučuje tokenizovani ulazni program, znači ulazni niz znakova pretvara u niz binarnih zapisa od kojih svaki ima klasu i vrednost. Druga funkcija leksičkog analizatora je da preskoči, to jest, ignoriše, (proguta) neke delove ulaznog teksta. To se odnosi na razne blanko znakove, tabulatore, takođe u većini programskih jezika, kraj linije se isto tretira kao blanko znak i komentare koje programer ubacuje u programski kod. Uz neke izuzetke, kompajler te komentare u potpunosti ignoriše i njih izfiltrira leksički analizator.

Leksički analizatori u kompajlerima zasnovani su na matematičkom pojmu konačnog automata, odnosno, state mašina, mašina sa konačnim brojem stanja.

Deterministički konačni automati

Pojam konačnog automata je dosta korišćen u računarskim naukama. Vi ste se već sreli u okviru osnova računarske tehnike i logičkog projektovanja digitalnih sistema sa tim pojmom -- u okviru sekvencijalnih mreža pominjani su Murovi i Milijevi automati kao primeri konačnih automata. Mi ćemo ponovo definisati taj koncept i podsetiti se na način kako nam odgovara za potrebe ovoga kursa.

Na slici je primer jednog konačnog automata prikazan u formi tabele. Osnovni elementi (S, U, δ, S_t, P) koji čine jedan konačni automat su:

- Konačan skup S stanja automata, kojima su obeležene vrste tabele prelaza; ovde su stanja označena velikim slovima S = {A, B, C, D, E, F},

- Skup U ulaznih simbola, koji obeležavaju kolone tabele prelaza. U konkretnom primeru to su nule i jedinice U = {0, 1},

- Funkcija prelaza automata, δ: S x U → S , koja za tekuće stanje automata i tekući ulazni simbol daje koje je sledeće stanje automata. Ona je data ovom tabelom -- vrsta tabele određuje tekuće stanje, a kolona tekući ulazni simbol.

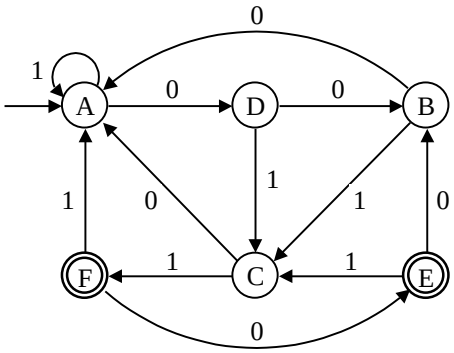
		0	1	
→	A	D	A	0
	B	A	C	0
	C	A	F	0
	D	B	C	0
	E	B	C	1
	F	E	A	1

- Sledeći element definicije konačnog automata je karakterizacija svakog od stanja. Stanja se dele na dva podskupa: stanja prihvatanja $P \subseteq S$ i stanja odbijanja S-P. U konkretnom slučaju P = {E, F}. Ovde je to naznačeno na kraju svake vrste, ako stoji nula, odgovarajuće stanje je stanje odbijanja, a ako na kraju stoji jedinica onda je u pitanju stanje prihvatanja.

- Poslednji element definicije automata je startno ili početno stanje automata $S_t \in S$. Ono može biti ili eksplicitno naznačeno strelicom, ili ako nema nikakve naznake, onda po konvenciji se uzima stanje iz prve vrste ove tabele prelaza. U konkretnom slučaju S_t = A.

Ako funkcija prelaza na jednoznačan način određuje sledeće stanje za određeno tekuće stanje i ulazni simbol, onda je reč o determinističkom konačnom automatu, kao što je na ovom primeru i dato.

Alternativan način prezentovanja konačnog automata koji je inače i češće korišćen u praksi, je putem grafa prelaza. U ovoj varijanti stanja su predstavljena čvorovima grafa, a funkcija prelaza predstavljena je orijentisanim granama koje povezuju pojedine čvorove. Stanja prihvatanja su dvostruko zaokružena, a startno stanje ima ulaznu strelicu. Ako na primer iz stanja B imamo izlaznu granu označenu sa nula u stanje A, to znači da, ako je B tekuće stanje automata i dolazi ulaz 0, sledeće stanje automata je stanje A.



Kako radi automat? U nastavku je naveden algoritam rada automata. Rad automata obavlja se u diskretnim vremenskim trenucima. U svakom trenutku, algoritam vodi evidenciju o tekućem stanju automata (inicijalno je to startno stanje) i tekućem ulaznom simbol. Na ulazu algoritma je sekvenca sastavljena od ulaznih simbola i automat obrađuje tu sekvencu tako što u svakom koraku rada obradi jedan simbol ulazne sekvence i potom se pomeri na sledeći simbol ulazne sekvence, znači obrađuje s leva na desno znak po znak. inicijalno, tekući ulazni simbol je prvi simbol ulazne sekvence.

```
tekuće_stanje := St;
tekući_ulaz := prvi simbol ulazne sekvence;
while not (kraj ulazne sekvence)
        tekuće_stanje := δ( tekuće_stanje, tekući ulaz );
        tekući_ulaz := sledeći znak ulazne sekvence;
end while;
if ( tekuće_stanje ∈ P )
        then ulazna sekvenca se prihvata;
        else ulazna sekvenca se ne prihvata;
```

```
    end if.
```

Glavni deo algoritma odvija se u ovoj while petlji: dok se ne dođe do kraja ulazne sekvence, konsultuje se funkcija prelaza. Za tekuće stanje i tekući ulaz ona daje novo stanje koje postoje tekuće, a tekući ulaz se konzumira, u smislu da sad sledeći znak ulazne sekvence postaje tekući ulazni simbol za sledeću iteraciju petlje. Kada se u ovoj petlji dođe do kraja ulazne sekvence, kada više nema ulaznih simbola, bitno je u kom je stanju automat završio rad. Ako je u tom trenutku, kada se došlo do kraja ulazne sekvence, tekuće stanje jedno od stanja prihvatanja, onda se kaže da automat sekvencu prihvata. U suprotnom automat sekvencu ne prihvata.

Osnovni cilj ovoga algoritma je da za sekvencu sastavljenu od ulaznih simbola da binarni izlaz, znači prihvatljiva je ili nije prihvatljiva. U tom smislu, definiše se jezik konačnog automata kao skup svih sekvenci ulaznih simbola koje automat prihvata. To je u suštini podskup skupa svih zamislivih sekvenci.

Za ovaj naš konkretan primer, probanjem možemo utvrditi da, recimo, sekvenca 0110 predstavlja jednu prihvatljivu ili legalnu sekvencu.

$$A \xrightarrow{0} D \xrightarrow{1} C \xrightarrow{1} F \xrightarrow{0} E$$

Kako pratimo rad automata? Automat kreće iz stanja A, za nulu na ulazu ide se u stanje D prema grafu. Zatim iz stanja D, kada dolazi sledeći simbol ulazne sekvence, jedinica na ulazu, ide se u stanje C. Za sledeću jedinicu na ulazu ide se u F. Zanimljivo je da je F stanje prihvatanja, međutim još nije kraj ulaza. Imamo još nulu na ulazu tako da se iz F za nulu ide u stanje E i sada tek automat završava rad pošto više nema ulaznih simbola. Završno stanje E vidimo da je poduplano, znači radi se o stanju prihvatanja, tako da konstatujemo da je ovo jedna prihvatljiva sekvenca, odnosno sekvenca koja pripada jeziku L(K) automatika K.

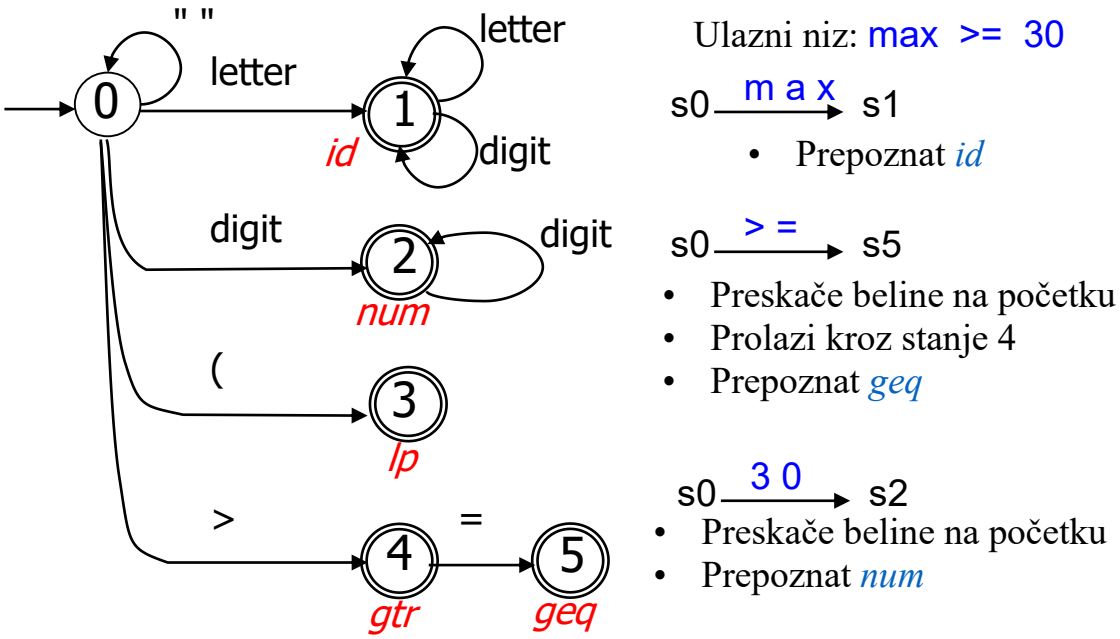
Takođe, probanjem možemo videti da ovaj konkretan automat prihvata neograničen skup sekvenci sastavljenih od nula i jedinica, ali takođe postoji i beskonačno puno sekvenci koje automat ne prihvata. Na primer, dobili smo jednu prihvatljivu sekvencu 0110. Kako možemo konstruisati još proizvoljan broj dužih sekvenci? Tako što uočavamo da iz stanja E možemo stići u stanje F tako što na sekvencu dodamo 11, znači iz E se ide u C, iz C se ide u F, to je onda prihvatljiva ulazna sekvenca. Takođe, opet možemo da se vratimo u stanje E za 0, znači sad sve sekvence koje završavaju u stanju F i još dodamo 011 one će ponovo doći u stanje F, tako da tu podsekvencu 011 možemo proizvoljan broj puta dodati na neku sekvencu koja već završava u stanju F (koja je sekvenca prihvatanja) i dobićemo na taj način još proizvoljno mnogo novih dužih sekvenci prihvatanja, $\{1011, 0011, 011011, 011011011,...\} \in L(K)$.

Šta možemo konstatovati za sekvence koje nisu prihvatljive ovom automatu? Vidimo, recimo, da u startnom stanju A postoji povratna grana označena sa 1 u stanje A, znači ako na ulazu dolazi jedinica automat ostaje u stanju A i sve sekvence koje su sastavljene isključivo od jedinica proizvoljne dužine, ovaj automat će odbijati zato što će uvek završavati rad u stanju A koje je stanje odbijanja. $\{0, 00, 00111, 1, 11, 111, 1111,...\} \notin L(K)$. Vidimo da su i skupovi prihvatljivih i skupovi sekvenci koje nisu prihvatljive neograničeni u opštem slučaju.

Konačni automat kao skener

Kako se koncept konačnog automata koristi unutar kompajlera? Sledi jedan uprošćeni primer kako se kompajlerski skener realizuje u vidu konačnog automata.

Ovde vidimo jedan konačni automat koji ima šest stanja, označena brojevima od 0 do 5. Na bazi oznaka grana možemo videti da su ulazni simboli blanko znak, letter, digit, kao i specijalni znaci leva zagrada, veće i jednako, Znači to sve čini ulaznu azbuku. Takođe vidimo da je 0 startno stanje. Vidimo neke prelaze, recimo ako dolazi blanko na ulazu ostajemo u startnom stanju, ako na ulazu dolazi letter, znači jedno slovo, idemo iz stanja 0 u stanje 1. Vidimo da startno stanje nije stanje prihvatanja, ali su ostala stanja od 1 do 5, stanja prihvatanja. Ono što jedino odudara od dosadašnje definicije konačnog automata su crveno označeni simboli. Završnim stanjima su dodate crvene oznake koje će u nastavku biti detaljno objašnjene, koje u suštini predstavljaju klasne delove tokena koje ovaj automat prepoznaje.



Evo kako može da se odvija rad kompajlerskog skenera odnosno ovog konačnog automata: pretpostavimo nekakav ulazni niz, to je neki mali deo nekog programa, znači neki izraz max >= 30. Ovde može da bude blanko između x i znaka > i između znaka = i 3.

Skener počinje rad tako što ostatak kompajlera poziva skener recimo kao nekakav potprogram, sa idejom da u svakom pozivu bude vraćen jedan token koji je prepoznat u ulaznoj sekvenci. Pri svakom pozivu skenera, njegov rad kreće od startnog stanja 0. Ovde je upisano s0 da se kreće od stanja 0.

Prvo dolazi slovo m na ulazu. Slovo m je slovo. Ovaj automat je sažeto predstavljen, što nije nacrtana posebna grana između stanja nula i jedinica za svako pojedinačno slovo, nego je uveden opšti simbol letter za bilo koje slovo. Vidimo da ima prelaz iz 0 ka 1 kada dođe jedno slovo, znači m, idemo u stanje 1. Sada kada dolazi sledeće slovo a vidimo da postoji povratna grana, znači ostajemo u stanju 1, takođe i kad dođe slovo x ostaje se u stanju 1. Posle slova x dolazi na ulazu blanko. Blanko znak kao ulazni simbol vidimo da se prepoznaje stanju 0. Praktično samo stanje 0 ima definisan prelaz za blanko znak i to je ova povratna grana. Međutim, druga stanja nemaju

definisan prelaz za blanko znak. To znači da to u suštini nije legalan prelaz automata i u principu bi to predstavljalo grešku za dalje prepoznavanje, znači teorijski gledano, ova ulazna sekvenca kao celina, za ovaj automat bila bi iz tog razloga neprihvatljiva. Međutim, u kontekstu korišćenja konačnog automata kao skenera, skener završava rad na poslednjem prihvatljivom delu ulaza.

Pri svakom pozivu skenera od strane kompajlera, skener nastavlja procesiranje ulaza od mesta gde je stao i pokušava da upari na ulazu prihvatljiv deo maksimalne dužine. Znači u prvom pozivu on je uspeo ova tri slova m, a, x da prepozna i poslednje stanje koje bilo prihvatljivo je stanje 1. To poslednje stanje 1 određuje token koji će da skener vrati ostatku kompajlera.

Znači, mi smo proširili osnovnu definiciju automata, da osim prepoznavanja da li je recimo ovaj deo sekvence legalan (ovaj početak tri slova je legalan), on ne vraća samo tu informaciju da je u pitanju ispravan početak sekvence nego i dodatnu informaciju da se radi o tokenu klase ID zato što smo stanje 1 tako označili. Znači, ostatku komajlera vraća se prepoznat token id.

U sledećoj iteraciji, ostatak kompajlera, kada bude potreban sledeći token poziva ponovo skener. Skener ponovo počinje rad od stanja 0 i nastavlja rad od mesta gde je na ulazu stao, odnosno od poslednjeg simbola koji nije mogao da procesira. Znači onaj blanko znak je praktično sad prvi znak koji treba da se na procesira. Takvo ponašanje skenera nije teško realizovati u programiranju pošto se u principu koriste standardni ulazno-izlazni metodi i po potrebi, ako se pročita jedan znak viška, može da se sa nekim unget vrati na ulaz tako da se ponovo kreće od tog istog znaka.

Konkretno. krećemo od blanka. Blanko u stanju 0 može da se procesira i ostajemo u stanju 0, znači prosto skener proguta taj blanko. Posle blanka dolazi znak >. Znak > znači tranzicija ka stanju 4 i to je stanje prihvatanja, međutim, iza znaka > odmah dolazi znak = i konačni automat može dalje da procesira ulaz, ima legalan prelaz iz 4 u 5. Nastavljamo znači do god je moguće procesiranje i konzumira se i znak =. Posle toga ide drugi blanko koji ne može u stanju 5 da se prihvati tako da skener sad završava ovaj poziv u stanju 5 i vraća ostatku kompajlera da je prepoznat token veće do jednako GEQ. Znači, na taj način, kad su ova dva znaka spojena, upravo zbog onog principa da se uvek prepoznaje najduža moguća prihvatljiva sekvenca, neće recimo biti posebno vraćeni dva tokena, strogo veće i token jednakosti, nego će ova dva simbola biti tretirana kao poseban specijalni znak.

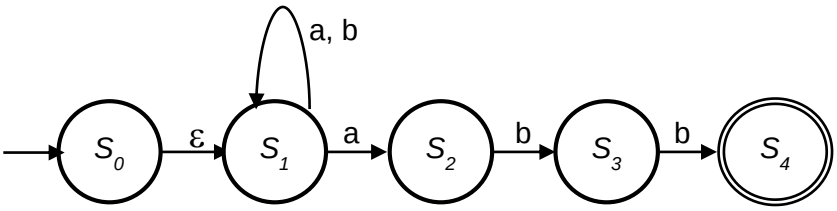
Sledeći poziv skenera kreće od ovog drugog blanka i od stanja 0, znači taj blanko se ovaj proguta, ostaje se stanju 0. Zatim dolazi cifra (digit) 3 i to je prelazi iz stanja 0 u stanje 2. Sledeća cifra 0 izaziva naravno ostajanje u stanju 2. Iza cifre 0 u našem primeru nema više ulaznih zakova. Implementaciono, često se uvodi poseban simbol koji označava kraj ulaznog niza, recimo to može biti EOF (end of file). U svakom slučaju, skener se zaustavlja na poslednjem prepoznatom znaku, to je ova druga cifra, završava rad u stanju 2 i vraća token NUM. Sledeći poziv skenera od strane ostatka kompajlera bi samo konstatovao da više nema simbola na ulazu. U takvoj situaciji skener tipično vraća neki poseban token, recimo EOF, tako da onda i ostatak kompajlera sazna da se došlo do kraja programa.

Nedeterministički konačni automati

Do sada smo definisali determinističke konačne automate, međutim konačni atomati mogu se pojaviti u obliku nedeterminističkhi konačnih automata. Taj oblik je manje korišćen od

deterministčkih, ali nama, za neke algoritme konstrukcije skenera, je zgodan kao koncert, pa ćemo uvesti i definisati taj koncert.

Na sledećoj slici prikazan je primer jednog nedeterminističkog konačnog automata u obliku grafa prelaza i, na prvi pogled, vidimo da je u pitanju automat koji ima pet stanja s0 do s4. Takođe vidimo da postoje prelazi označini raznim simbolima; a, b i posebnim znakom epsilon. Šta ovaj automat izdvaja i čini nedeterminističkim, to jest, po čemu odudara od definicije determinističkih automata?



Prvo što uočavamo je da recimo u stanju s1 postoje dve izlazne grane; jedna povratna označena sa a, b, a druga grana od s1 do s2 je označena sa a. Šta znači kada se napiše a, b? To je samo skraćeno obeležavanje da se ne crtaju dve različite grana, jedna označena samo sa a, a druga označena samo sa b. To može da se radi i kod determiničkih automata, znači to nije ništa posebno što bi odudaralo od definicije determinističkog automata. Međutim, ono što odudara od definicije determinističkog automata, to je činjenica da iz stanja s1 imamo dva prelaza za isti ulazni simbol a. Znači, ovako kako je nacrtano, ovaj automat ako se nađe u stanju s1, a na ulazu mu dolazi a, može ili da ostane u stanju s1, ili da pređe u stanje s2. Znači tu vidimo da funkcija prelaza ne daje jednoznačan rezultat za ovu situaciju i to čini automat nedeterminističkim.

Još jedan aspekt, koji možda ne mora odmah da se uoči, ali kad znamo određenu konvenciju takođe čini automat nedeterminističkim -- slovo epsilon je jedna konvencija koja se često koristi u kompajlerskim knjigama. Ono ima specijalno značenje, znači to nije običan ulazni simbol kao ovi drugi ulazni simboli a,b, nego je to oznaka za, u suštini, nedostatak ulaznog simbola, takozvanu praznu sekvencu, odnosno sekvencu dužine nula. Šta to onda znači ako je ova grana između s0 i s1 označena sa epsilon? To znači da ovaj automat može da odluči da pređe iz stanja s0 u stanje s1 pri čemu na ulazu ne menja ništa. Znači ako je recimo početak ulazne sekvence (prvi simbol) tekući simbol, znači posle ovog prelaza iz s0 u s1 i dalje smo na prvom simbolu ulazne sekvence, znači nije konzumiran ni jedan ulazni simbol, a automat može da pređe iz s0 u s1. To takođe čini automat nedeterminističkim.

Još jedan aspekt koji nije dat u ovom primeru može da čini automat nedeterminističkim, a to je da, za razliku od determinističkog automata koji ima jedinstveno startno stanje, znači samo jedno od stanja može biti početno, kod nedeterminističkih automata u opštem slučaju može biti označen veći broj stanja kao početna. To bi se označilo tako što bismo nacrtali dodatnu strelicu i za neko drugo stanje na primer s2 – da je ovde još jedna strelica, to znači da bi automat mogao da bira da li će početi procesiranje iz stanja s0 ili iz stanja s2. Znači sv iti elementi čine jedan automat nedeterminističkim ako su prisutni.

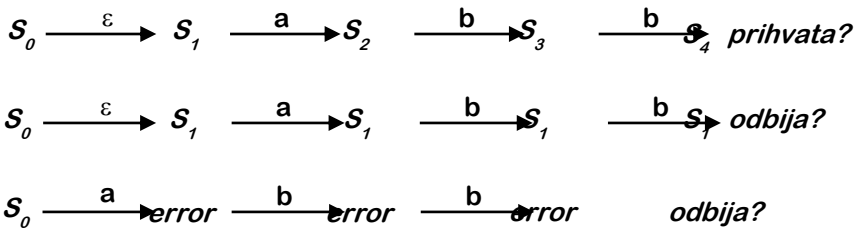
Formalna definicija nedeterminističkog automata je da njega određuje pet elemenata (S, U, δ, S_t, P) gde:

- S predstavlja skup stanja automata, u konkretnom slučaju S = {S_0, S_1, S_2, S_3, S_4}; stanjima su obeležene čvorovi grafa prelaza automata prikazane na prethodnoj slici;
- U predstavlja skup ulaznih simbola (azbuku automata), u konkretnom slučaju U = {a, b}; ulazni simboli obeležavaju grane grafa.
- δ: S x (U $\cup$ {ε}) $\rightarrow$ P(S) predstavlja funkciju prelaza (P - partitivni skup); za svako stanje i svaki ulazni simbol definiše **skup** $S_N \subseteq$ S mogućih novih stanja u koje automat može preći iz stanja s $\in$ S za ulazni simbol u $\in$ U **ili za simbol prazne sekvence** ε; u konkretnom slučaju, funkcija δ zadata je grafom prelaza prikazanim na prethodnoj slici; svaka izlazna grana ćvora X označena sa Y pokazuju na stanja iz skupa δ(X,Y);
- $S_t \subseteq$ S je **skup** startnih stanja automata (stanja iz koga može početi rad). U konkretnom slučaju S_t = {S_0}; startna stanja automata se po konvenciji označavaju strelicama.
- P $\subseteq$ S predstavlja skup stanja prihvatanja, u konkretnom slučaju P = {S_4}. Stanja iz skupa S-P nazivaju se stanjima odbijanja. Na prethodnoj slici, stanje prihvatanja nactano je duplom linijom.

U suštini, ova definicija je slična definiciji deterministiičkog automata i razlike su ovde istaknute bold fontom. Jedna razlika u odnosu na deterministički automat je kod funkcije prelaza. Funkcija prelaza definiše skup mogućih novih stanja, a ne jedinstveno novo stanje. Takođe, domen funkcije prelaza je dekartov proizvod skupa stanja i skupa ulaznih simbola proširenih simbolom prazne sekvence. Razlika je u tome što je definicija funkcije prelaza sada generalizovana na skupove novih stanja i može biti ne jedno startno stanje nego skup startnih stanja. Ostali elementi definicije su isti kao kod determinističkog automata.

Direktna posledica ovog nedeterminizma je da imamo konfuziju oko toga kada je određena sekvenca za automat prihvatljiva ili nije prihvatljiva. Kod determinističkog automata mi smo mogli da, za zadati automat i neku konkretnu ulaznu sekvencu, sprovedemo niz prelazaka, od startnog stanja do završnog stanja. znači da se konzumira cela ulazna sekvenca i da na bazi toga šta je završno stanje -- da li je stanje prihvatanja ili nije stanje prihvatanja -- da kažemo da li se sekvenca prihvata ili ne. Znači, postojao je algoritam koji jasno to određuje.

Probajmo da primenimo taj algoritam determinističkog automata recimo za ovaj primer nedeterminističkog automata koji smo već uzeli i na primer uzmimo sekvencu abb, znači tri ulaza slova, da vidimo znači da li tu sekvencu ovaj automat prihvata ili ne.



Krećemo od stanja s0 i sad, pošto u s0 nemamo direktan prelaz za a, po jednom od scenarija, mi možemo, a u suštini i moramo ići u s1. Pošto je to epsilon prelaz, mi nismo ništa konzumirali sa ulaza. U stanju s1 možemo da biramo, da li ćemo za a ostati u s1 ili ići u s2. Ako se, u ovom prvom

scenariju, odlučimo za stanje s2, znači posle procesiranja a sa ulaza smo u stanju s2, posle toga dolazi na ulazu b koje nas vodi u stanje s3 i dolazi na ulazu drugo b, koje automat vodi u stanje s4. Stanje s4 je označeno kao jedino stanje prihvatanja ovoga automata. Posmatrajući ovaj prvi scenario, deluje da je ova sekvenca abb za ovaj automat prihvatljiva.

Međutim, s obzirom da je ovaj automat nedeterministički i da u nekim koracima možemo proizvoljno da biramo dalji tok rada, postoji alternativni scenario za isti automat i istu ulaznu sekvencu. Po alternativnom scenariju, iz s0 po epsilon prelazu idemo u s1 i sad biramo povratnu granu označenu sa a, znači ostajemo u s1. Kada dođe b na ulazu, opet se ostaoje u s1 i kada dođe drugo b na ulazu, opet po istoimenoj povratnoj grani moramo ostati u stanju s1. Stanje s1 je stanje odbijanja. Ako gledamo ovaj drugi scenario, završavamo u stanju odbijanja i izgleda, kada bismo se držali naravno definicije determinističkog automata, da se ova ulaza sekvenca odbija.

Znači imamo jedan scenario po kome se ta sekvenca prihvata i neki drugi scenario po kome se sekvenca odbija. Da li tu postoje za ovu sekvencu i ovaj automat još neki scenariji rada ovoga automata?

Postoje još neki scenariji koji zahtevaju malo još dodatnih objašnjenja. Ovde vidimo treći scenario gde mi krećemo iz stanja s0 i na ulazu dolazi a. Obzirom da u stanju s0 nemamo nijednu izlaznu granu označenu sa a, postavlja se pitanje, šta to znači sa stanovišta funkcije prelaze? Ova predstava automata u formi grafa prelaza je zgodna za sagledavanje strukture automata, ali ona u stvari skriva jednu činjenicu. Ako uzmemo definiciju funkcije prelaza, ona mora biti definisana za za bilo koje stanje i bilo koji ulazni simbol iz skupa ulaznih simbola. Ovaj konkretan automat ima ulazne simbole a i b. Rekli smo epsilon je posebno i nije ulazni simbol, ali a i b su ulazni simboli. Postavlja se pitanje, a koja je onda vrednost funkcije prelaza za stanje s0 i simbol a, recimo, kada to ni na koji način na ovom grafu nije naznačeno eksplicitno.

Tu se koristi još jedna konvencija, a to je da takav prelaz naravno mora da bude definisan i znači mora postojati stanje u koje se u takvim situacijama prelazi, a to je takozvano stanje greške. Ovakav automati i većina sličnih automata koji su predstavljeni ovakvim grafovima gde fale neki prelazi na prvi pogled, u stvari podrazumeva da postoji još jedno dodatno stanje automata koje se najčešće naziva i obeležava kao greška. To stanje po konvenciji se ne navodi u grafu. Zašto? Kada bismo crtali to stanje i sve prelaze ka njemu jako bismo opteretili ove grafove, u suštini većina prelaza bi bila ka tom stanju greške i, u tom smislu, po konvenciji se to stanje podrazumeva ali se ne crta.

Gde god ne postoji eksplicitno nacrtan prelaz, to je prelaz u to podrazumevano stanje greške. U ovom primeru, iz s0, ako dolazi a na ulazu, to je znači prelaz u stanje greške. Zašto se to stanje zove stanje greške? Zato što je to stanje odbijanja i jednom kada se dođe u stanje greške, kakav god dalji ulaz da dolazi, bilo a bilo b. ostaje se u stanju greške. Znači, kada jednom automat dođe u stanje greške, onda smo sigurni da sekvenca više ne može da bude prihvaćena.

Ako uzmemo ovu konvenciju u obzir, za naš primer ulaza abb, moguće je da već u stanju s0 kada dolazi a, odemo u stanje greške i tu se zadržavamo do kraja ulazne sekvence. I po tome ispada da se ova sekvenca odbija Ovo nije jedini dodatni scenario, možda postoje još neki. Znači konstruisali smo više scenarija u kojima se sekvenca odbija, nego onih u kojima se sekvenca prihvata.

Sada postoji dilema oko toga da li je ova konkretna sekvenca abb ovom automatu prihvatljiva ili nije? Kod nedeterminističkih automata definiše se da je sekvenca prihvatljiva, ako nađemo bar jedan scenario promene stanje od nekog početnog do nekog završnog stanja koji završava u stanju

prihvatanja. U suprotnom, ako svi scenariji vode u završna stanja odbijanja onda sekvenca nije prihvatljiva. Tako da smo razrešili dilemu i iza ovaj konkretan primer ova sekvenca abb je prihvatljiva.

Algoritam rada netederminističkog automata

Kako do definišemo rad ovog nedeterminističkog konačnog automata? Ako definišemo algoritam rada automata u dve odvojene celine, možemo izraziti algoritam rada nedeterminističkog automata da bude vrlo sličan algoritmu rada determinističkog automata uz razlike definisane samo na nekim mestima. Prvo definišemo takozvano epsilon zatvaranje (engl. epsilon closure).

```
Ulaz: skup stanja T, izlaz: skup stanja ε–closure(T);
for (∀ stanje t ∈ T )
        staviti t na stek;
end for;
ε–closure(T) := T;
while (stek nije prazan)
        Skinuti vršni element t sa steka;
        for (∀ stanje u za koje postoji ε–prelaz od t ka u)
                ε–closure(T) := ε–closure(T) ∪ {u};
                staviti u na stek;
        end for;
end while
```

To je jedna funkcija koja se definiše nad svakim stanjem nedeterminističkog automata i kao rezultat ta funkcija daje skup stanja dostupnih iz posmatranog stanja prateći isključivo prazne (espilon) prelaze. Za neko stanje s, kako računamo epsilon zatvaranje? Tako što prvo u rezultat dodamo samo stanje s, a zatim konsultujući funkciju prelaza, uzimamo svako od tih novo dodatnih stanja i posmatramo po funkciji prelazak, iz tog stanja, da li postoji neki epsilon prelaz. Uzimamo sva stanja koja su naznačena kao odredišta tih epsilon prelaza i dodamo ih u rezultujući skup. Celu stvar iterativno ponavljamo, kad god ažuriramo rezultujuci skup dodavanjem nekog novog stanja, konsultujemo funkciju prelaza za to stanje i gledamo da li postoje prazni prelazi. Znači do god postoji promena u tom skupu, vrtimo se u ovim koracima 2 do 6 i kad jednom ne postoji promena znači da smo završili konstrukciju zatvaranja za određeno ulazno stanje.

Sada kada znamo šta je funkcija Epsilon zatvaranja za neko stanje, možemo formulisati algoritam rada celog nedeterminističkog konačnog automata u jednom formatu koji je jako sličan algoritmu rada deterministickog konačnog automata. U nastavku je dat taj algoritam rada nedeterminističkog automata i plavim su naznačene razlike u odnosu na algoritam rada determinističkog konačnog automata.

```
tekući_skup_stanja := ε-closure(S_t) ;
tekući ulaz := prvi simbol ulazne sekvence;
while not (kraj ulazne sekvence)
    novi_skup_stanja := ∅;
```

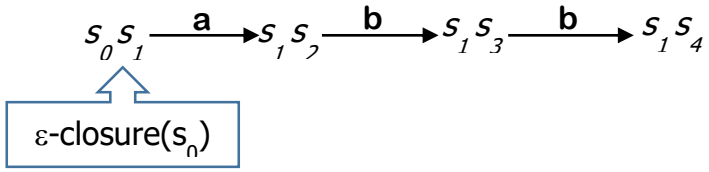
```
    for (∀ s ∈ tekući_skup_stanja)
        novi_skup_stanja:=novi_skup_stanja ∪ δ(s,tekući_ulaz);
    end for;
    tekući_skup_stanja := ε-closure(novi_skup_stanja);
    tekući_ulaz := novi simbol ulazne sekvence;
end while;
if (tekući_skup_stanja ∩ P ≠ ∅)
    then ulazna sekvenca se prihvata;
    else ulazna sekvenca se ne prihvata;
end if.
```

Algoritam ima istu strukturu kao i algoritam determinističkog automata. Koje su konkretne razlike? Isti je ulaz algoritma, isti je izlaz. Ulaz algoritma je ulazna sekvenca. Izlaz algoritma je binarni: da li se ulazna sekvenca prihvata ili ne. Najveća i glavna razlika je što deterministički automat uvek, u svakom koraku, ima jedinstveno tekuće stanje, dok u ovom nedetrminističkom, vodimo evidenciju o skupu stanja koja su moguća tekuća stanja. Mi u stvari u paraleli razmatramo razne sekvence prelaska između stanja i vodimo evidenciju o mogućem skupu tekućih stanja po bilo kojoj od tih dozvoljenih sekvenci prelaza -- to je tekući skup stanja.

Inicijalno uzimamo skup startnih stanja i vršimo njegovo Epsilon zatvaranje i to je inicijalni tekući skup stanja, a tekući ulaz je prvi simbol ulazne sekvence. Onda u while petlji obrađujemo ulaznu sekvencu znak po znak. U svakoj iteraciji napredujemo i procesiramo po jedan simbol ulazne sekvence zato što se ovi Epsilon prelazi obuhvataju funkcijom Epsilon zatvaranja. Oni ne mogu da se pojave u ulazu. U ulazu se pojavljuju konkretni ulazni simboli, u svakoj iteraciji po jedan novi.

Kako formiramo u jednoj iteraciji novi skup stanja? Tako što konsultujemo funkciju prelaza za svako od stanja iz tekućeg skupa stanja i za tekući ulazni simbol i onda uradimo uniju svega što nam vrate te funkcije prelaza. Tako formirano novi tekući skup stanja, međutim to još nije kraj, nego još nad tako formiranim novim skupom stanja uradimo Epsilon zatvaranje i onda dobijamo tekući skup stanja za novu iteraciju. Konačno, tekući ulaz za novu iteraciju je novi simbol ulazne sekvence koji je iza posmatranog, znači posle prvog drugi, posle drugog treći i tako dalje.

U while petlji obrada se vrti sve dok se ne stigne do kraja ulazne sekvence. Kada automat dođe do kraja ulazne sekvence imamo tekući skup stanja u tom trenutku i potrebno je izvršiti proveru da li možda u tom tekućem skupu imamo bar jedno stanje prihvatanja. Znači uradimo presek sa skupom prihvatljivih stanja i ako je taj presek različit od praznog skupa (u tekućem skupu stanja imamo bar jedno prihvatljivo stanje), onda se sekvenca prihvata, u suprotnom se sekvenca ne prihvata.



Ako primenimo ovaj algoritam na prethodni primer automata i sekvence abb, prvo treba da sprovedemo Epsilon zatvaranje nad startnim stanjem s0 i time dobijemo s0s1 i to nam je inicijalni tekući skup stanja. Sada na ulazu dolazi malo a, moramo znači videti prvo za stanje s0 kada dolazi a na ulazu, ovde nema takvog prelaza eksplicitno naznačenog, to je u suštini onaj implicitan prelaz

u stanje greške i mi bi ovde mogli znači da napišemo i error stanje, ali pošto ga nismo naznačili u grafu, mi ga i u ovoj sekvenci koju konstruišemo izostavljamo i nećemo ništa pogrešiti pošto error stanje neće dovesti do prihvatanja. Znači za a kao tekući ulazni simbol, s0 ne daje ni jedno novo stanje, ali stanje s1 za a dodaje dva stanja u novi skup stanja. To je s1 zbog ove povratne grane i s2. Znači oba stanja s1,s2 su posledica prelaza iz s1 pod ulazom a.

Da li je ovo finalni skup? Nije -- kad god se dobije jedan skup stanja na njega treba primeniti Epsilon zatvaranje da bismo formirali finalni novi skup tekućih stanja. Epsilon zatvaranjem s1,s2 ne dobija se ništa novo zato što je jedini Epsilon prelaz bio iz stanja s0 Tako da ovaj znači Epsilon zatvaranje će dati opet ovaj isti skup s1,s2 i to je novi tekući skup stanja.

Sada kada dolazi b na ulazu, Iz s1 možemo ostati u s1, a iz stanja s2 se ide u s3, znači ovde pišemo s1s3. Na to primenjujemo Epsilon zatvaranje ali ne dodajemo nikakvo novo stanje zato što nema epsilon prelaza iz s2 i s3. Konačno kada dođe finalno b na ulazu, iz s1 ostajemo opet u s1, a iz s3 pod b idemo u s4. Znači finalni skup stanja je s1s4, epsilon zatvaranje ne dodaje ništa novo. U tom finalnom skupu stanja postoji stanje prihvatanja to je s4, konstatujemo da se ova cela sekvenca prihvata. Znači ovo je bila primena algoritma rada nedeterminističkog automata na konkretnom primeru.

Odnos nedeterminističkih i determinističkih automata

Ako posmatramo međusoban odnos nedeterminističkih i determinističkih automata, možemo videti da je deterministički automat jedan specijalan slučaj nedeterminističkog automata, odnosno da je definicija nedeterminističkog automata generalnija (sa manje ograničenja).

U odnosu na nedeterministički automat, deterministički automat ima tačno jedno starno stanje, nema praznih prelaza i funkcija prelaza za svako stanje vraća tačno jedno novo stanje. Važan teorijski rezultat je da se za svaki zadati nedeterministički konačni automat može uvek konstruisati ekvivalentan deterministički konačni automat.

Kada govorimo o pojmu ekvivalencije dva automata, ona znači da automati prihvataju tačno isti skup sekvenci, odnosno da je jezik oba automata isti. Za primenu je zanimljivo, ako nam je zadat nedeterministički automat, kako konstruisati njegov deterministički ekvivalent i taj algoritam ćemo proučiti u nastavku.

Algoritam konverzije nedeterminističkog automata u deterministički je izložen je u nastavku. Osnovna ideja je da sistematski popunjavamo tabelu prelaza determinističkog automata. Tabela ima vrste označene stanjima i kolone označene ulaznim simbolima i moramo popuniti svaku ćeliju tabele, čime dobijamo nova stanja determinističkog automata kao skupove stanja nedeterminističkog. Ovo je slično posmatranju kako radi nedeterministički automat za jednu ulazu sekvencu, samo što ovde ne uzimamo jednu konkretnu ulaznu sekvencu, nego sistematski određujemo te prelaze.

Postupak dobijanja tabele prelaza determinističkog ekvivalenta za dobijeni nedeterministički automat glasi:

1. Startno stanje determinističkog automata određuje se kao ε–zatvaranje skupa startnih stanja nedeterminističkog automata. Obeležiti prvu vrstu tabele ovim stanjem. Operacija ε–zatvaranja definisana je u prethodnom odeljku.
2. Izabrati nepopunjenu vrstu tabele označenu stanjem S.

3. S je stanje prihvatanja ako sadrži bar jedno stanje prihvatanja nedeterminističkog automata, u suprotnom radi se o stanju odbijanja.
4. Svaki ulaz u izabranoj vrsti tabele popunjava se na sledeći način: za svaki ulazni simbol u odrediti skup $S' = \bigcup_{s \in S} \delta(s, u)$, gde je δ funkcija prelaza nedeterminističkog automata. Ulaz tabele u vrsti S i koloni u popuniti sa skupom koji se dobija ε–zatvaranjem skupa S'. Ako nijedna vrsta tabele prelaza nije označena sa S', označiti novu vrstu tabele prelaza sa S'.
5. Ako postoji nepopunjena vrsta tabele, preći na korak 2. U suprotnom, konstrukcija automata je završena.

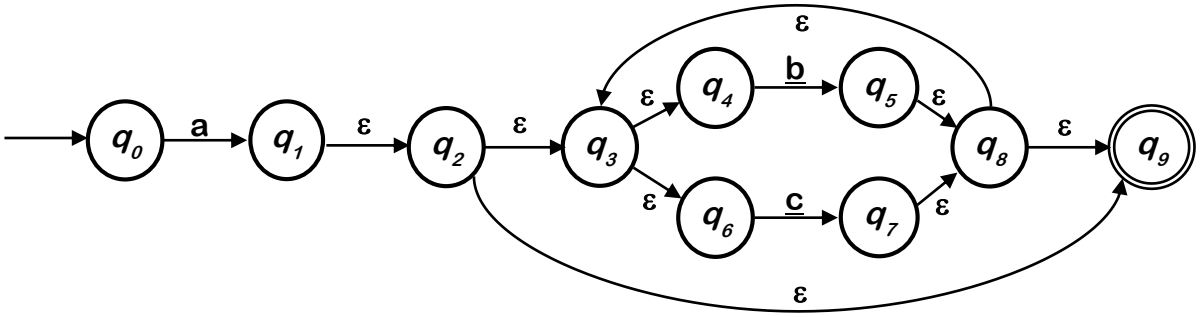
Krećemo od skupa startnih stanja nedeterminističkog automata. Epsilon zatvaranjem dobijamo startno stanje determinističkog automata. Tim stanjem, nazovimo ga So, obeležavamo prvu vrstu tabele prelaza determinističkog automata.

Sada je potrebno popuniti svaku ćeliju te prve vrste. Da bismo popunili svaku ćeliju prve vrste, razmatramo startno stanje So determinističkog automata za svaki od ulaznih simbola, u koje novo stanje taj deterministički automat može da pređe. Kako se konstruiše to novo stanje determinističkog automata? Tako što za neki konkretan ulazni simbol u uzmemo skup stanja nedeterminističkog automata koji odgovara stanju So. Potom razmatramo za svako s iz tog skupa, za određeni ulazni simbol u, šta nam daje funkcija prelaza nedeterminističkog automata -- to je skup stanja s' -- i onda uzmemo Epsilon zatvaranje skupa s'. Pošto tu ima znači niz s u početnom stanju So determinističkog automata, uzimamo uniju svega što nam vrate sva ta Epsilon zatvaranja, to je novo stanje S' determinističkog automata i time popunjavamo određenu ćeliju u vrsti So i koloni u. I tako dalje, za svaki ulazni simbol, dok se ne popuni cela vrsta So.

Kada se popuni prva vrsta So. mi ćemo dobiti niz nekih novih stanja, kojima osim što popunjavamo ćelije prve vrste, označavamo i posebne vrste determinističkog automata. Tako smo dobili nova stanja determinističkog automata i sada je potrebno sve ovo iterativno raditi dok god ne popunimo i te novodobijene vrste tabele prelaza determinističkog automata, sve dok se ne kompletira tabela determinističkog automata.

Mana izloženog algoritma je što, teorijski, može doći do eksponencijalnog povećanja broja stanja. Znači, ako nedeterministički automat ima nekih N stanja, deterministički automat u najgorem slučaju može da ima 2^N stanja. Zašto je to tako? Stanja determinističkog automata odgovaraju nekim skupovima stanja nedeterminističkog automata, a to znači nekom podskupu skupa svih mogućih stanja nedeterminističkog automata. Jedan od matematičkih rezultata je da partitativni skup nekog skupa od N elemenata ima 2^N elemenata, to je skup svih podskupova nekog skupa. U tom smislu i deterministički automat može da ima u najgorem slučaju 2^N stanja ako nedeterministički automat ima N stanja. Međutim, u praksi ova mana algoritma koji ćemo izložiti ne dolazi do izražaja zato što automati sa kojima mi radimo generalno nemaju jako puno starija, drugo konverzijom se uvek dobija automat koji nema bitno više stanja, znači da kod determinističkih automata koji se javljaju u kontekstu konstrukcije kompajlera nikada se ne pojavljuje taj efekat da dolazi do eksponencijalnog povećanja broja stanja.

Primer konverzije nedeterminističkog automata u deterministički



Cela stvar je jasnija kada se sagledava na konkretnom primeru. Ovde gore je zadat u formi grafa prelaza nedeterministički automat. Zašto je ovaj automat nedeterministički? Vidimo da on ima jedno startno stanje q0, nema veze što su ovde neka stanja obeležena drugim bojama, vidimo da je jedino stanje prihvatanja duplo podebljano q9. Takođe vidimo da postoji veći broj Epsilon prelaza recimo između q1 i q2, pa između q2 i q3 i tako dalje. Znači zbog svega toga je automat nedeterministički. Kako ćemo konstruisati ekvivalentan deterministički automat?

Primenićemo opisanu proceduru na dati automat. Krećemo od startnog stanja nedeterminističkog automata q0 i pošto nema Epsilon prelaza, znači Epsilon zatvaranje samo daje q0. Donja tabela odgovara determinističkom automatu. Znači prvo unosimo q0 u prvu vrstu i to je startno stanje determinističkog automata. Inicijalno, prva vrsta ove tabele i cela tabela je prazna. Tabela ima tri kolone: vidimo iz gornjeg grafa da a, b i c predstavljaju ulazne simbole i zato nam trebaju te tri kolone. Naravno ne treba unositi Epsilon kao posebnu kolonu jer mi konstruišemo deterministički automat, znači kad bismo dodali kolonu Epsilon to bi onda bio nedeterministički automat, to bi bila greška.

Kako se sada popunjavaju kolone prve vrste ove tabele? Posmatramo prvo kolonu a u prvoj vrsti. Da bismo popunili kolonu a, uzimamo tekući skup nedeterminističkih stanja to je samo jedno stanje q0. q0 ima prelaz a i taj prelaz je u stanje q1. Znači inicijalno unosimo stanje q1 u prvu vrstu i prvu kolonu tabele. Međutim to nije kraj, nego uvek moramo primeniti Epsilon zatvaranje da bismo dobili finalni skup stanja. Šta se sve dodaje Epsilon zatvaranjem q1?

Stanja DKA	Stanja NKA	a	b	c	
→s_0	q_0	q_1, q_2, q_3 q_4, q_6, q_9			0
s_1	q_1, q_2, q_3 q_4, q_6, q_9		q_5, q_8, q_9 q_3, q_4, q_6	q_7, q_8, q_9 q_3, q_4, q_6	1
s_2	q_5, q_8, q_9 q_3, q_4, q_6		s_2	s_3	1
s_3	q_7, q_8, q_9 q_3, q_4, q_6		s_2	s_3	1

Imamo prelaz po Epsilon u q2, znači njega dodajemo. Pa onda iz q2 imamo izlazne Epsilon grane u q3 i q9, znači moramo i njih dodati. Zatim iz q3 imamo Epsilon grane u q4 i q6, znači moramo i njih dodati. Sada iz tih pomenutih q4, q6 i q9 više nema Epsilon grana i tu smo onda završili. Tranzitivno se posmatra lanac sastavljen isključivo od Epsilon prelaza. Svih ovih šest stanja u stvari čine novo stanje u koje se prelazi za ulaz a. Sada, osim što popunjavamo prvu ćeliju u prvoj

vrsti, mi takođe odmah unosimo to novo dobijeno stanje u drugu vrstu tabele. Ovde su uvedene i skraćene oznake za stanja deterministčkog automata s1, s2, s3, ali ta stanja su reprezentovana ovim skupovima stanja nedeterminističkog automata. Znači s1 je ovih šest stanja nedeterminističkog automata.

Sada smo tek završili prvu ćeliju prve vrste. Sada razmatramo drugu ćeliju prve vrste, znači prelaz za b iz q0. Iz q0, pošto nemamo eksplicitno naznačenu granu za b, znači da je to prelaz ka stanju greške za koji smo rekli po konvenciji da nećemo eksplicitno pisati tako da ovde ostaje prazan skup, kao skup novih stanja. Pošto je prazan skup, ni Epsilon zatvaranje ne može ništa tu da doda, ta ćelija (prva vrsta kolona b), kao što vidimo, ostaje prazna. To je u stvari stanje greške ovog deterministčkog automata.

Stanje greške deterministčkog automata odgovara praznom skupu stanja nedeterminističkog automata. To bi moglo u ovoj ovoj tabeli eksplicitno da se naznači, znači da se doda još jedna vrsta koja bi bila označena sa error ili praznim skupom stanja. Međutim, kao što možemo u grafu prelaza da izostavimo stanje greške, tako i u tabelama. Gde god ostane prazan ulaz, to po konvenciji znači prelaz u to posebno stanje greške koje su isto ne dodaje u tabelu (ono nema svoju vrstu). Sve se to ovde podrazumeva, znači kada bismo hteli da dodamo još to stanje eksplicitno mi bismo morali da dodamo novu vrstu i u toj vrsti svi prelazi bili bi ostajanje u stanju greške i to stanje bi bilo stanje odbijanja. Ali, po konvenciji, mi to stanje nikad ne dodajemo u tabelu, a svi prelazi u njega ostaju ovi naizgled nepopunjeni odnosno prazni ulazi.

Znači završili smo sa drugim ulazim u prvoj vrsti i na sličan način i treći ulaz u prvoj vrsti (za c) ostaje u stvari taj prelaz u stanje greške, znači nepopunjen. Time smo potpuno završili sa prvom vrstom. Ostaje još samo da kažemo da li je ovo početno stanje deterministčkog automata stanje prihvatanja ili ne. Pogledamo skup ekvivalentnih stanja nedeterminističkog automata, to je ovo q0 i pošto q0 nije stanje prihvatanja, znači beležimo da i startno stanje s0 deterministčkog automata nije stanje prihvatanja.

Stanje s1 uneto je u drugu vrstu tabele prelaza. Nastavljamo konstrukciju tabele prelaza tako što sada razmatramo drugu vrstu redom za kolone a, b i c i iste rezone kao maločas ponovimo. Vidimo, recimo, da iz bilo kog od ovih stanja označenih u stanju s1, znači q1, q2, q3, q4, q6 i q9, nema prelaza ni iz jednog od tih stanja označenih simbolom a, jer vidimo da se simbol a pojavljuje u onom nedeterminističkom automatu samo na jednom mestu, iz q0 u q1. Tako da u suštini to ostaje prelaz u stanje greške znači nepopunjen.

Međutim, vidimo da iz stanja s1 deterministiičkog automata imamo prelaz označen sa b. Konkretno, b je na jednom mestu iz q4 u q5. Pošto u s1 imamo q4 kao jedno od stanja u tom tekućem skupu stanja, u koloni b prvo pišemo q5 kao novo stanje, a onda na q5 primenjujemo Epsilon zatvaranje čime dodajemo q8, zatim, iz q8, q9 i q3, zatim, iz q3, q4 i q6 i tako formiramo kompletan novi skup stanja koji postaje novo stanje deterministčkog automata s2. Njemu dodeljujemo posebnu vrstu u ovoj tabeli prelaza. Time smo završili sa drugom ćelijom druge vrste, ali smo dodali još jedno stanje u deterministički automat.

I tako dalje, sada moramo nastaviti ceo postupak iterativno do god ne popunimo celu tabelu. Za ovaj konkretan primer, postupak se nastavlja i dodaje se još samo jedno novo stanje (to je stanje s3) u deterministički automat, a zatim se samo nastavi popunjavanje i ostatak tabele se popuni ne dodajući nijedno novo stanje. Stanje greške sa implicitno podrazumeva. Na kraju dobijamo deterministički automat koji ima samo četiri stanja, dok je recimo onaj polazni nedeterministički automat imao 10 stanja.

Ako predstavimo rezultujući automat koristeći ove skraćena oznake stanja i u formi tabele prelaza, dobijamo sledeću tabelu.

	a		b	c	
s_0	s_1				0
s_1			s_2	s_3	1
s_2			s_2	s_3	1
s_3			s_2	s_3	1

Tu sada možemo primetiti, kad pogledamo ovu uprošćenu tabelu zanimljivo svojstvo te tabele da imamo tri vrste koje su identične: s1, s2, s3 su na potpuno isti način popunjene i s1,s2 i s3 su stanja prihvatanja. Praktično to znači da u bilo koje od tih stanja unutar tog skupa s1,s2,s3, ako ovaj konačni automat tokom rada dođe, on će se na potpuno isti način ponašati u nastavku rada kao da je došao u bilo koje od drugih stanja. Znači ako dođe u stanje s2 potpuno je isto kao da je došao u stanje s1 ili da je došao stanje s3.

U suštini to znači da nama ne trebaju tri različita stanja u ovom automatu, nego da umesto ta tri stanja može da se ostavi samo jedno od tih stanja kao reprezent sva ta tri stanja. Recimo, izaberemo stanje s1 i, gde god je u polaznoj tabeli ili grafu bio prelaz u s2, mi sad to tretiramo kao prelaz u s1. Takođe, gde god je bio prelaz u s3, to sada tretiramo kao prelaz u s1. Na taj način možemo dobiti mnogo jednostavniji graf prelaza koji je prikazan na samom dnu slajda, gde imamo samo dva stanja s0 i s1. Iz s0 za a idemo u s1. s1 je stanje prihvatanja. Ako dolaze u nastavku b ili c ostaje se u stanju s1 i sve te sekvence su prihvatljive. Tim postupkom smo dobili deterministički automat koji ima samo dva stanja od onog polaznog nedeterminističkog automata koji je imao 10 različitih stanja.

	a	b	c	
s_0	s_1			0
s_1		s_1	s_1	1

minimalni automat

Ovaj postupak da se neka stanja ponašaju na isti način tokom rada automata i da onda mogu da budu redukovana, zamenjena znači, samo jednim svojim predstavnikom, zove se *minimizacija automata*.

Formalno se može definisati kada su dva stanja nekog automata *ekvivalentna*, a to je ako prihvataju tačno iste sekvence u nastavku rada automata počev od svakog od tih stanja. Ako se utvrdi da su dva stanja evivalentna, onda se, pošto je u pitanju relacija ekvivalencije znači to može da se odnosi na više stanja, znači ako je s1 ekvivalentno sa s2 a s2 ekvivalentno sa s3, prema tranzicionononoj osobini relacije ekvivalencije onda je s1 ekvivalentno sa s3. Generalno može biti N međusobno ekvivalentnih stanja i ceo taj skup ekvivalentnih stanja može se zameniti jednim predstavnikom.

Čitalac se na ovom mestu upućuje na Zbirku zadataka gde je pojam ekvivalencije preciznije definisan, kao i algoritmi koji proveravaju za automat da li su stanja ekvivalentna. Opisan je I postupak redukcije automata, odnosno određivanja minimalnog ekvivalentnog automata.

Za deterministički automat može se definisati pojam minimalnog ekvivalentnog automata. To je u suštini automat koji više nema ekvivalentnih stanja, znači sva preostala stanja međusobno nisu

ekvivalentna, takođe sva stanja su dostižna. U opštem slučaju može da se konstruiše neki automat koji može da ima nedostižno stanje iz početnog. Minimalni automati nemaju nedostižnih stanja.

Da li treba težiti minimalnim automatima? Oni imaju manje memorijsko zauzeće kad se programski realizuju, ali to ne utiče na brzinu rada automata jer je algoritam rada automata linearne složenosti u odnosu na dužinu ulaza i to se ne menja. Znači potpuno je isti broj koraka, bez obzira da li automat minimalan ili ne, samo se na reprezentaciju funkcije prelaza troši manje memorije. Za kompajlerske primene automati nisu preterano veliki, to je par desetina, ili par stotina stanja. U skladu sa tim, minimizacija nije neka bitna osobina sama po sebi, ali pošto se formalno može sprovesti za bilo koju automat, obično postoje alati koji automatski sintetizuju ovakve automate i oni pri tome mogu bez problema da sprovedu i minimizaciju.

Regularni izrazi

Regularni izrazi su sledeći koncept koji ćemo definisati, povezan sa leksičkom analizom u kompajlerima. Zašto ih sada pominjemo i proučavamo? Zato što su regularni izrazi alternativni formalizam za opis leksike programskih jezika i opisuju tačno isti skup jezika kao konačni automati. Formalno se može, za svaki regularni izraz, naći konačni automat koji opisuje isti jezik kao taj izraz. Postoji algoritam za to, takođe za svaki konačni automat može se naći regularni izraz koji opisuje isti jezik kao taj automat. Oba algoritma su definisana, ali mi ćemo proučiti samo ovaj prvi scenario, znači kada je zadat regularni izraz kako dobiti konačni automat, jer je on nama interesantan za primenu.

Šta su regularni izrazi i zašto ih sad uvodimo kad već imamo konačne automate i oni deluju dosta zgodno, zašto sad uvesti neki novi formalizam? Regularni izrazi u suštini su još zgodniji formalizam od konačnog automata i sa njima čovek barata još jednostavnije.

Formalno ćemo definisati regularne izraze, a kada budemo videli primere onda će nam stvar postati još jasnija. Formalna definicija je da je regularni izraz način da opišemo nekakav skup nizova ili sekvenci znakova. Ako je izraz S, onda se jezik regularnog izraza, to jest skup sekvenci znakova izraza S, označava kao L(S). Definicija polazi od jednostavnih izraza i onda definišemo kako od jednostavnijih izraza primenom regularnih operatora da dobijemo kompleksnije regularne izraze.

Najjednostavniji regularni izrazi su:

- prazan skup sekvenci $L(\varnothing)=\varnothing$ (koristi se oznaka praznog skupa, $\varnothing$), zatim
- regularan izraz koji opisuje praznu sekvencu ε, odnosno sekvencu dužine 0 znakova: $L(\varepsilon)=\{\varepsilon\}$. Zanimljivo je da su $\varnothing$ i ε dva različita regularna izraza: nije svejedno da li uopšte nemamo sekvencu ili imamo jednu sekvencu dužine nula.
- Sledeći regularni izraz je prost niz znakova n, $L(n)=\{n\}$. Znači, ako napišemo ABC to je regularni izraz koji označava jednu sekvencu ABC sastavljenu od tri znaka.

Složeniji regularni izrazi dobijaju se od jednostavnijih regularnih izraza A i B primenom različitih regularnih operatora:

- Operator opcije ? je unarni operator, $L(A?) = L(A) \cup \{\varepsilon\}$. A? je oznaka da je A opciono, odnosno jezik ovog regularnog izraza obuhvata jezik samog izraza A sa dodatom praznom sekvencom. Drugim rečima, ako izostavimo A, to je u stvari sekvenca dužine nula.

- Operator unije, A | B, označava se kao uspravna crta, predstavlja uniju skupova sekvenci koje opisuju posebno izraz A i posebno izraz B, formalno $L(A|B) = L(A) \cup L(B) = \{x \mid x \in A \text{ ili } x \in B\}$.
- Operator nadovezivanja, piše se AB ili A.B, je dekartov proizvod L(A) x L(B), odnosno to je skup svih sekvenci x i y, gde komponenta x dolazi iz skupa A, a komponenta y dolazi iz skupa B, formalno $L(AB) = L(A) \times L(B) = \{xy \mid x \in A, y \in B\}$.
- Potom uvodimo takozvano stepenovanje, kao unarni operator za neki regularni izraz A, pa onda napišemo neki faktor N, na primer A^0, A^1, A^2 i tako dalje. Po konvenciji, to je u stvari nadovezivanje A na samog sebe N puta, pri čemu A^0 je po definiciji prazna sekvenca.

$$A^n \equiv \underbrace{A \cdot A \cdot \ldots \cdot A}_{n}, \qquad A^0 \equiv \{\varepsilon\}$$

Sada kada znamo šta je to A^N, možemo još dva regularna operatora da definišemo:

- Zvezdasto zatvaranje, ili ponavljanja određene sekvence nula ili više puta A*, po definiciji je $A^*=A^0|A^1|A^2|A^3|\ldots$ i tako dalje, znači proizvoljno veliki broj članova.
- Pozitivno zatvaranje $A^+ = A^1|A^2|A^3|\ldots$ je vrlo slično zvezdastom, ali izostavljen je onaj prvi član A^0. To je dakle ponavljanje sekvenci jedan ili više puta.

Time bismo završili formalnu definiciju regularnog izraza, a sada da vidimo kroz primere šta to u stvari znači. Ako napišemo samo ABC to u stvari predstavlja tu jednu sekvencu ABC dužine tri, L(abc)={abc}

Ako napišemo A | B | C, znači operator unije, onda L(a | b | c)={a, b, c}, to su u stvari tri sekvence dužine jedan: sekvenca samo A, druga sekvenca samo B i treća sekvenca samo C.

Izraz (A | B) (C | D) u suštini znači da treba da kombinujemo iz prve zagrade bilo koji član sa bilo kojim članom iz druge zagrada tako da dobijemo četiri različite sekvence znakova AC, AD, BC, BD. Dakle L[(a | b)(c | d)]={ac, ad, bc, bd}. U principu mogu da se definišu prioriteti ovih operatora pa unija je niži prioritet nego nadovezivanje i zato je stavljena u zagrade. Vidimo da se u regularnim izrazima mogu koristiti zagrade da bi bio jasan prioritet.

Izraz A* predstavlja simbol A i njegovo ponavljanje nula ili više puta, $L(A^*)=\{\varepsilon, A, AA, AAA, \ldots\}$ To je niz sekvenci od kojih je najkraća prazna sekvenca (dužine 0), zatim jedno A, pa dva A, pa tri A i tako dalje znači proizvoljan broj simbola A, bilo koja takva sekvenca.

Izraz $(A|B)^+$ označava A|B, ponovljeno jedan ili više puta, to je znači prvo skup sekvenci dužine jedan {A, B}, pa onda uzimamo uniju sa dekartovim proizvodom {A, B} x {A, B}, znači sekvence dužine dva, pa onda unija sa svim sekvencama dužine 3 i tako dalje. U svakoj od tih sekvenci mogu se koristiti A i B u proizvoljnim kombinacijama. U krajnjoj liniji, to su znači svi mogući nizovi sastavljeni od slova A i B, dužine jedan ili više.

$L[(a \mid b)^+]=\{a, b\} \cup \{a, b\} \times \{a, b\} \cup \{a, b\} \times \{a, b\} \times \{a, b\} \cup \ldots=$
$=\{a,b\} \cup \{aa, ab, ba, bb\} \cup \{aaa, aab, aba, abb, baa, bab, bba, bbb\} \cup \ldots=$
$=\{a, b, aa, ab, ba, bb, \ldots\}$

U praksi mi često pišemo veći broj regularnih izraza i možemo određenim regularnim izrazima dodeliti neko ime. Na primer:

Slovo =A | B |…| Z | a | b |…| z
Cifra = 0 | 1 |…|9
Identifikator = Slovo (Slovo | Cifra)*

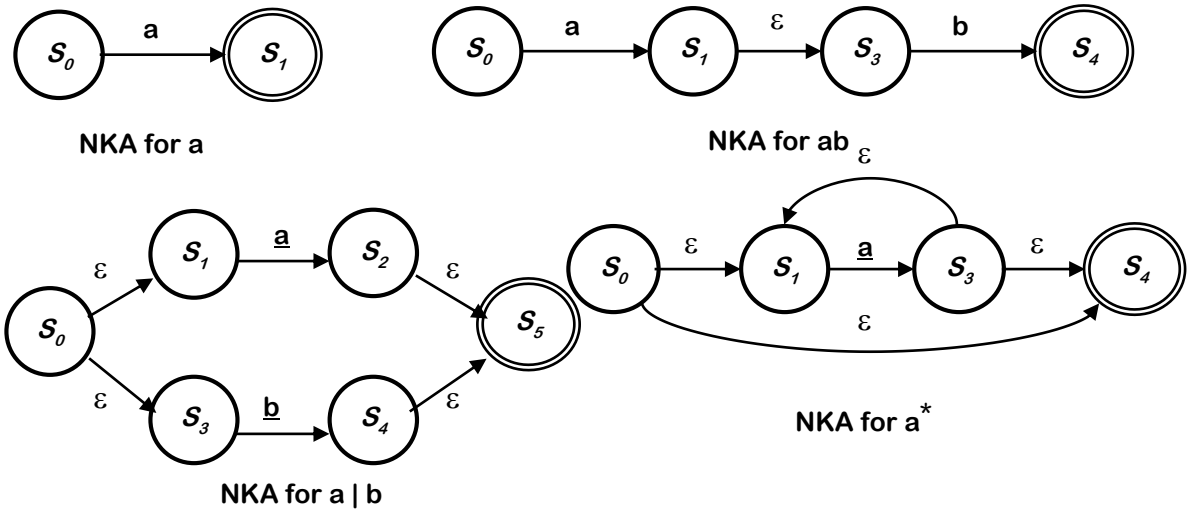
Regularni izraz Slovo je unija svih velikih i svih malih slova, regularni izraz Cifra je unija svake pojedinačne cifre, to je skup koji se sastoji od deset raznih sekvenci dužine jedan, a to su cifre 0, 1 do 9. Onda možemo reći da je regularni izraz Identifikator u stvari Slovo nadovezano na Slovo ili Cifru što se ponavlja nula ili više puta. Drugim rečima, leksiku jednog jezika, znači svaki token, možemo opisati regularnim izrazima upravo na ovaj način. Ovaj identifikator može biti klasa jednog tokena u nekom programskom jeziku.

Regularni izlazi su vrlo pogodan mehanizam da se na koncizan način opiše opiše kako se formiraju pojedinačni tokeni programskog jezika. Ideja je znači da programer opiše svaki token nekog programskog jezika regularnim izrazom, pa da onda postoji alat koji će tu ulaznu specifikaciju, znači niz takvih regularnih izraza, prevesti u jedan konačni automat koji se onda može lako programski implementirati. Zato je nama od interesa da proučimo algoritam konstrukcije konačnog automata iz regularnog izraza.

Konstrukcija konačnog automata iz regularnog izraza

Postoji ne jedan, nego više algoritama koji mogu da reše taj problem. Mi ćemo proučiti dva različita algoritma. Prvi algoritam je takozvani Tomsonov algoritam. On koristi kao međurezultat nedeterminističke konačne automate i to je bio primarni razlog zašto smo mi definisali pojam nedeterminističkog konačnog automata, upravo zato što nam on treba da bismo formulisali ovaj algoritam. Tomsonov algoritam je jako zgodan kada čovek treba da na papiru znači ručno, odredi za regularne izraz automat.

Po Tomsonovom algoritmu ceo postupak se obavlja nad grafovima konačnih automata. Kreće se od najjednostavnijih elemenata nekog regularnog izraza. Prvo dobijamo nedeterminističke konačne automate za najjednostavnije regularne izraze, a zatim, da bi smo dobili konačni automat za složeniji regularni izraz, koji predstavlja primenu nekog operatora na jednostavniji regularni izraz, primenjujemo Tomsonova pravila koja su ovako grafički predstavljena.



Šta kaže prvo Tomsonovo pravilo? Ako je regularni izraz samo nekakav ulazni simbol a, onda odgovarajući nedeterministički konačni automat treba nacrtati tako što uvedemo dva stanja, znači početno stanje S0, završno stanje (stanje prihvatanja) S1 i prelazi od S0 za a u S1.

Zatim, ako imamo dva nadovezana regularna izraza, jedan je predstavljen simbolom a drugi je predstavljen simbolom b, onda prvo konstruišemo posebno automat za a, posebno automat za b. Automat za a je predstavljen stanjima S0 i S1, automat za b stanjima S3 i S4. Da bismo dobili za konkatenaciju ta dva izraza rezultujući automat, povežemo završno stanje automata za a (stanje S1), Epsilon granom ka startnom stanju automata za b (stanju S3). Sada u tom rezultujućem automatu, startno stanje automata za a ostaje jedino startno stanje, a završno stanje automata za b ostaje jedino završno stanje, znači S1 više nije završno stanje.

Svi ovi Tomsonovi automati jesu nedeterministički zato što imaju Epsilon prelaze, međutim uvek imaju jedno početno i jedno završno stanje, što u stvari omogućuje ova pravila. Iako je pravilo ovde izraženo tako što su a i b neki elementarni regularni izrazi, znači pojedinačni ulazni nizovi, isto pravilo funkcioniše i za složenije izraze koji bi bili međusobno nadovezani, jer i automat nekog složenijeg izraza će imati jedno početno i jedno završno stanje. Bez obzira koliko još ima međustanja između, čitav podgraf može da bude vrlo kompleksan, uvek će biti jedno startno i jedno završno stanje i uvek može da se primeni ovo pravilo, bez obzira koliko su složeni ovi regularni izrazi a i b i odgovarajući automati.

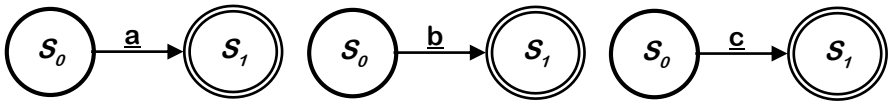
Pravilo konstrukcije automata za uniju dva izraza a i b je takvo da, ako imamo automat za a (on je reprezentovan startnim stanjem S1, završnim stanjem S2), zatim imamo poseban automat za b (koji je reprezentovan startnim stanjem S3, završnim stanjem S4), onda ih objedinjujemo u zajednički automat koji predstavlja ceo izraz, tako što uvodimo novo stanje S0 koje će postati startno stanje rezultujućeg automata i onda uvodimo Epsilon prelaze od tog novog stanja ka svakom od originalnih početnih stanja, znači ka S1 i ka S3. Takođe uvodimo novo stanje S5 koje postaje rezultujuće završno stanje i onda imamo prazne prelaze od svakog od ovih inicijalnih završnih stanja (to su S2 i S4) do S5. Ovo pravilo može da se generalizuje u slučaju da imamo a|b|c, znači za veći broj podizraza. Opisali smo dakle rezultujući automat sa istom napomenom da ovi podizrazi a i b mogu biti složeni i samim tim njihovi automati mogu biti složeni, ali oni uvek imaju jedno početno i jedno završno stanje, tako da se uvek može primeniti ovo pravilo.

Konačno, automat za a*, znači zvezdasto zatvaranje odnosno ponavljanje nula ili više puta ima najsloženije pravilo. To je unarni operator, znači imamo prvo automat za a, on je reprezentovan startnim stanjem S1, završnim stanjem S3 i prelazom za a (ali tu može biti i složeniji automat). Šta sve treba uvesti da bi smo za ceo izraz a* dobili rezultujući automat? Moramo uvesti novo startno stanje S0 i povezati ga Epsilon prelazom ka starom startnom stanju S1. Takođe, moramo uvesti novo završno stanje S4 i od starog završnog stanja povući Epsilon prelaz ka novom S4. Takođe povući Epsilon prelaz od novog startnog stanja do novog završnog stanja i još dodati jedan Epsilon prelaz od starog završnog stanja ka starom početnom stanju. U rezultujućem automatu samo novo S0 je startno stanje, a samo S4 je završno stanje. Nije dato na slici, ali malom modifikacijom ovog pravila se dobija pravilo za a+, odnosno ponavljanje a jedan ili više puta. U tom pravilu za a+ treba samo donji Epsilon prelaz od S0 ka S4 izostaviti, znači isti je automat, samo nema ove grane od S0 ka S4 za a+ pravilo.

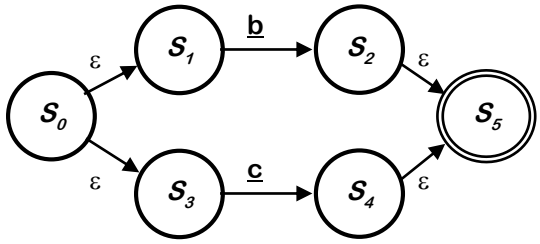
Primer

Sada da vidimo na konkretnom primeru kako se primenjuju ova Tomsonova pravila. Recimo, želimo da konstruišemo nedeterminističkii automat za ovaj regularni izraz a (b | c)*, znači to su sve sekvence koje počinju sa a, a onda imaju nula ili više simbola b ili c u proizvoljnom redosledu.

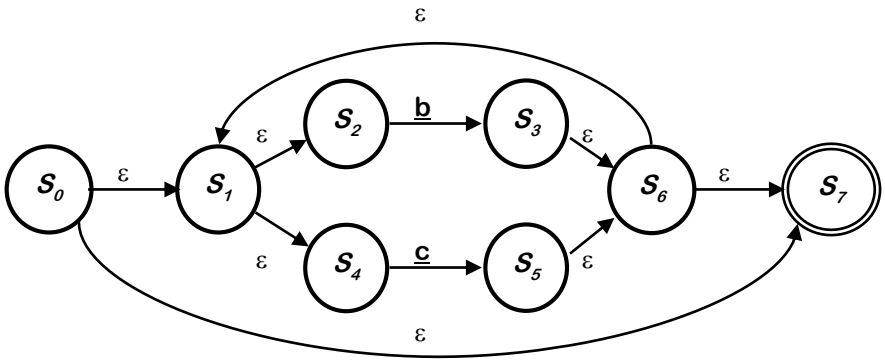
Krećemo od elementarnih izraza, to su samo a i samo bi samo c, i konstruišemo automate za njih.



Zatim kombinujemo b ili c, pošto su unutar zagrada, to je prioritetno, znači pravilom za uniju dobijamo ovaj rezultujući automat.

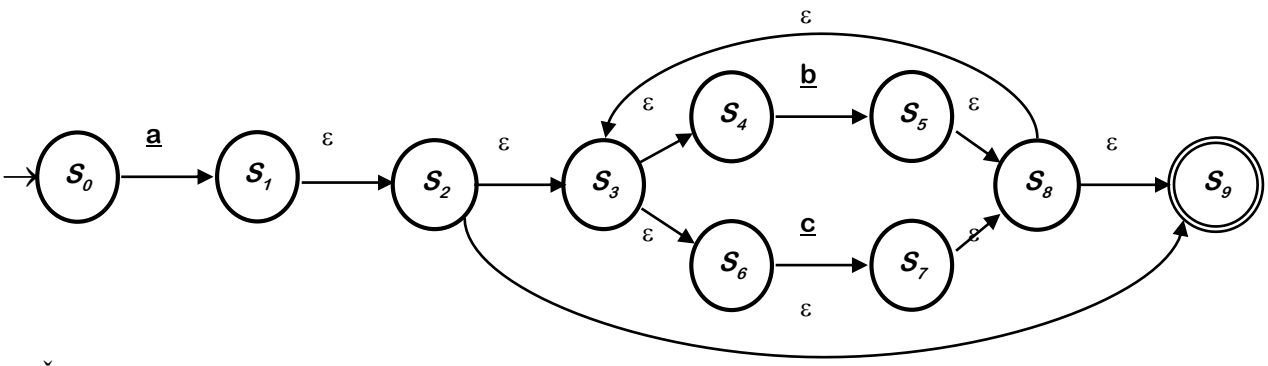


Zatim kao sledeće na zagradu (b|c) treba primeniti operator zvezdastog zatvaranja, znači uzimamo ceo ovaj automat koji smo dobili (S0 je njegovo početno stanje, S5 je njegovo završno stanje). Primenom prethodnog pravila treba da dodamo novo početno, novo završno stanje i četiri epsilon grane. Ovde su malo drugačije oznake, ponovo su numerisana u rezultujućem automatu stanja da redom idu njihove oznake, zato se malo razlikuju u odnosu na prethodni korak, ali struktura je ista znači ovo je rezultujući automat za (b|c)*.



Sada još treba na a nadovezati ceo ovaj podizraz, znači na ovaj automat za a nadovezati ceo ovaj prethodni automat tako što od završnog stanja za a, ka startnom stanju ovog drugog automata povučemo Epsilon granu.

Dobili smo jedan nedeterminički automat od deset stanja, od S0 do S9 i zanimljivo je da je to upravo automat koji smo koristili u prethodnom primeru gde smo ilustrovali konverziju nedeterminističkog automata u deterministički automat i na kraju smo kao rezultat u tom primeru dobili jedan mali deterministički automat od dva stanja koji u stvari reprezentuje ovaj izraz (da se počinje sekvenca sa a, a onda se dodaju b ili c proizvoljan broj puta).



Šta nam ta dva algoritma objedinjeno daju, znači Tompsonov i algoritam za konverziju nedeterminističkog automata u deterministički?

Daju nam u suštini način kako automatski konstruisati skener. Ono što konstruktor kompajlera treba da uradi je da napiše regularne izraze za svaki token programskog jezika za koji pravi kompajler, a zatim dalji postupak može potpuno da se automatizuje alatom. Znači da se za regularne izraze prvo konstruiše nedeterministički automat. pa da se zatim za dobijeni nedeterministički automat konstruiše ekvivalentni deterministički automat, eventualno minimizuje taj deterministički automati i zatim implementira programski deterministički automat u samoj implementaciji kompajlera.

Za ovakav postupak već su napravljeni alati i ne mora programer sam od nule da implementira sve ove algortime. Mi smo izložili sve te algoritme da bi nam bilo jasno kako rade ovi alati koji postoje za različite programske jezike na kojima je zamislivo implementirati kompajler. Recimo, za C i C++ jezike tu su alati lex i flex, dva dosta slična alata, a za onog ko koristiti javu tu su alati jlex i jflex, koji su u suštini nastali prevodjenjem ovih starijih alata lex i flex koji su prvi postojali, da mogu Java programeri da koriste slične alate. Jedan primer korišćenja alata jflex dat je u poglavlju o apstraktnim sintaksnim stablima. Ulazna specifikacija praktično predstavlja skup regularnih izraza. Za svaki regularni izraz može da se napiše neka akcija, a to je u stvari fragment java koda koji se izvršava kada se prepozna taj regularni izraz i akcija se obično svodi na vraćanje kompajleru nekog određenog tokena.

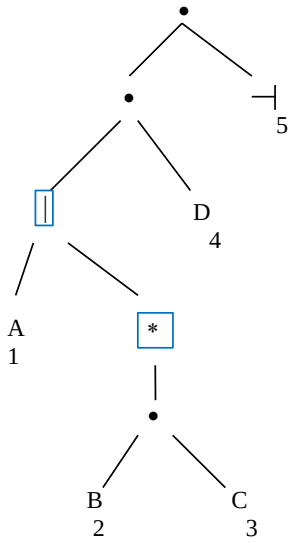
Metod pozicija

Izložićemo alternativni algoritam za dobijanje konačnog automata na osnovu regularnog izraza. Ovaj algoritam, koji neformalno nazivamo algoritam zasnovan na pozicijama, polazi od konstrukcije sintaksnog stabla za dati regularni izraz. Izložićemo osnove ovog algoritma na primeru:

(A | (BC)*)D⊣

Regularni izrazi je A ili B konkatenirano na C što se ponavlja nula ili više puta, sve nadovezano na D. Na kraju izraza, da bi primena algoritma bila jednostavnija, obavezno se dodaje jedan poseban znak, to je ovo položeno slovo T koje ima značenje i koristiićemo ga i nadalje u kursu kao marker kraja ulazne sekvence, nešto slično end-of-file znaku u programima. Uvek dodajemo to na kraj regularnog izraza i konstruišemo sintaksno stablo.

Čvorovi sintaksnog stabla su pojedini podizrazi. Listovi sintaksnog stabla su elementarni regularni izrazi: ulazni simboli i marker kraja ulazne sekvence, a unutrašnji čvorovi su regularni operatori. Poštujući prioritete konstruišemo za ovaj izraz sintaksno stablo koje je prikazano na slici.



Sledeći korak je da numerišemo redom listove sintaksnog stabla i tu numeraciju zovemo pozicijama. Prolazi se stablom redom s leva u desno (u in order redosledu) i numerišu se svi listovi. Ako eventualno list sadrži Epsilon, to jest praznu sekvencu, onda se on preskače. Znači Epsilon se ne numeriše, svi drugi listovi se tretiraju kao ulazi simboli, tako da se numerišu.

Za čvorove stabla definišemo osobinu *poništivosti*: čvor sintaksnog stabla po definiciji je poništiv, ako podizraz sa korenom u datom čvoru može generisati praznu sekvencu. Da bismo izračunali poništivost, krećemo od listova:

1. Listovi nisu poništivi, osim ako su označeni sa ε.
2. Čvor konkatenacije • je poništiv ako su oba njegova naslednika poništiva.
3. Čvor unije | je poništiv ako je bar jedan od njegovih naslednika poništiv.
4. Čvor zvezdastog zatvaranja * je uvek poništiv.

Na primeru koji razmatramo, listovi nisu poništivi, a jedini poništivi čvorovi su ovi koji su uokvireni kvadratom. Zvezdasto zatvaranje je uvek poništiv čvor i čvor ili iznad njega je poništiv zato što je jedan od sinova, odnosno taj čvor zvezdastog zatvaranja, poništiv.

Za svaki čvor stabla definišemo funkciju prva pozicija kao skup pozicija listova koji se mogu pojaviti na prvom mestu sekvence koje generiše podizraz sa korenom u datom čvoru. Da bismo izračunali prve pozicije, moramo redom obići unutrašnje čvorove stabla.

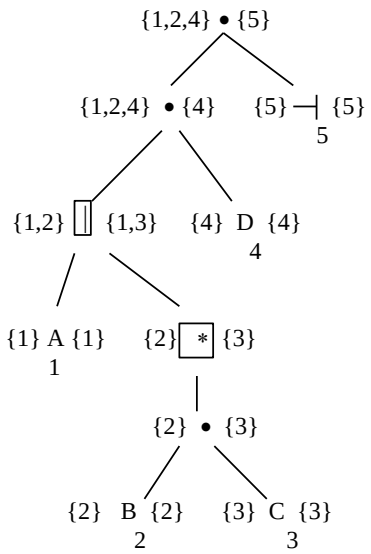
Ako je u pitanju čvor konkatenacije i ako imamo već izračunate pozicije sinova (opet se kreće od listova pa na gore) tada je prva pozicija čvora konkatenacije jednaka prvoj poziciji njegovog levog naslednika, ako levi naslednik nije poništiv, odnosno jednaka uniji prvih pozicija levog i desnog naslednika, ako je levi naslednik poništiv.

Prva pozicija čvora alternacije, odnosno unije, je unija prvih pozicija njegova oba naslednika, odnosno svih naslednika ako ih ima više.

Konačno, kod čvora zvezdastog zatvaranja, koje je unarna operacija, znači imamo jednog naslednika, prosto prekopiramo prvu poziciju naslednika.

Rezimirana pravila računanja prvih pozicija:

1. $prva_pozicija(a \bullet b) = \begin{cases} prva_pozicija(a), & a \text{ nije poništiv} \\ prva_pozicija(a) \cup prva_pozicija(b), & a \text{ je poništiv} \end{cases}$
2. $prva_pozicija(a|b) = prva_pozicija(a) \cup prva_pozicija(b)$
3. $prva_pozicija(a^*)$ = prva_pozicija(a).



Na primeru našeg stabla, krećemo od listova i po definiciji listovi imaju numerisane pozicije. To je ujedno i njihova prva pozicija. Oni predstavljaju same sebe. Sada redom idemo, prvo recimo ovaj čvor konkatenacije B i C, pošto njegov levi sledbenik nije poništiv, onda samo kopiramo prvu poziciju levog sledbenika, zatim se penjemo na čvor zvezdastog zatvaranja i kopiramo prvu poziciju jedinog sledbenika. Prve pozicije su označene levo od svakog čvora.

Zatim, prva pozicija ili čvora, pošto je u pitanju čvor unije, jednaka je uniji prvih pozicija oba naslednika -- pozicije 1 i 2 čine skup prvih pozicija. Zatim idemo na čvor konkatenacije iznad unije, gledamo da li je levi sledbenik poništiv. Pošto u ovom slučaju jeste poništiv, uzimamo prvu poziciju levog naslednika i pravimo uniju sa prvom pozicijom desnog naslednika, znači dobijamo skup {1,2,4}.

Konačno, kada pređemo na koreni čvor, pošto levi naslednik nije poništiv u ovom slučaju samo kopiramo prve pozicije levog naslednika i tako smo znači izračunali prve pozicije svakog čvora u stablu.

Sledeće što definišemo je funkcija *poslednja pozicija* za svaki čvor stabla. Ona predstavlja skup pozicija listova koji se mogu pojaviti na poslednjem mestu sekvence koji generiše podizraz sa korenom u datom čvoru. Znači ima neke analogije sa prvim pozicijama, samo sada se posmatra sekvenca od pozadi.

Računanje poslednjih pozicija; Ako je u pitanju čvor konkatenacije, onda gledamo da li je poništiv njegov desni sledbenik i, ako je on poništiv, onda uzimamo uniju poslednjih pozicija oba sledbenika, a ako desni sledbenik nije poništiv, onda uzimamo samo njegovu poslednju poziciju.

Poslednja pozicija čvora unije (ili alternative A ili B) je unija poslednjih pozicija oba naslednika.

Konačno, poslednja pozicija čvora zvezdastog zatvaranja je jednaka poslednjoj poziciji njegovog jedinog sledbenika.

Rezimirana pravila računanja poslednjih pozicija:

1. $poslednja_pozicija(a \bullet b) = \begin{cases} poslednja_poz.(b), & \text{b nije poništiv} \\ poslednja_poz.(b) \cup poslednja_poz.(a), & \text{b je poništiv} \end{cases}$
2. $poslednja_pozicija(a|b) = poslednja_pozicija(a) \cup poslednja_pozicija(b)$
3. $poslednja_pozicija(a^*) = \text{poslednja_pozicija(a)}.$

U našem primeru, poslednje pozicije označene su desno od svakog čvora. Isto se kreće od listova. Prvo razmatramo čvor konkatenacije, njegovi sledbenici su B i C, njihove poslednje pozicije 2 i 3 i pošto ovaj desni sledbenik C nije poništiv, onda uzimamo samo njegovu poslednju poziciju 3 i beležimo desno od čvora. Kada pređemo na čvor zvezdastog zatvaranja, samo prekopiramo tu poslednju poziciju 3. Kada pređemo na čvor iznad, čvor unije uzima uniju poslednjih pozicija sledbenika (1 i 3). Za sledeći čvor konkatenacije, pošto desni sledbrenik nije poništiv, uzimamo samo njegovu poslednju poziciju. Isto tako, za koreni čvor uzimamo samo poslednju poziciju desnog sledbenika (5). Tako da smo sada odredili poslednje pozicije svih čvorova stabla.

Sledeće što definišemo je funkcija *sledeća pozicija*. Za razliku od prethodnih funkcija, ova funkcija se ne definiše za čvorove stabla, nego za pozicije. Znači svaka pozicija ima sopstveni skup sledećih pozicija. Po definiciji, pozicija lista P stabla ima kao sledeće pozicije skup pozicija koje mogu slediti posmatranu poziciju P u proizvoljnoj sekvenci znakova izvedenoj iz datog regularnog izraza.

Pri računanju funkcije sledeća pozicija u sintaksnom stablu razmatraju se čvorovi tipa konkatenacija i zvezdastog zatvaranja, znači ignorišu se čvorovi unije, to jest alternative. Evo kako procesiramo ova dva tipa ćvorova:

- Čvor •: prva pozicija desnog naslednika je sledeća pozicija za sve iz poslednje pozicije levog naslednika.
- Čvor *: prva pozicija naslednika je sledeća pozicija za sve iz poslednje pozicije naslednika.

Sada ćemo to videti na tekućem primeru. Uzimamo već dobijeno anotirano stablo, gde smo naznačili prve i poslednje pozicije svih čvorova. Konstruišemo tabelu koja ima onoliko vrsta koliko ima pozicija u stablu. Imamo pet numerisanih listova, znači pet pozicija, imamo pet vrsta, 1, 2, 3, 4, 5. U ovoj drugoj koloni (sledeća pozicija) inicijalno navodimo prazne skupove. Postepeno konstruišemo ovu kolonu, tako što redom procesiramo čvorove stabla koji su ili konkatenacija ili zvezdasto zatvaranje. Redosled kojim ćemo te čvorove razmatrati nije bitan. Možemo krenuti ili od korena ka listovima ili obrnuto, ali je važno da odradimo svaki takav čvor.

poz.	sled. pozicija
1	{4}
2	{3}
3	{2,4}
4	{5}
5	/

Recimo, ako krenemo opet od dole pa na gore, ako uzmemo da procesiramo čvor konkatenacije B i C, možemo reći da njegov levi sin ima u poslednjoj poziciji 2, a njegov desni sin ima u prvoj poziciji 3. To znači da će 3 biti sledeća pozicije od 2 i to unosimo u tabelu, znači u vrstu 2 tabele unosimo 3 kao sledeću poziciju. Time smo znači završili procesiranje ovog čvora.

Kada dođemo na čvor iznad prethodno posmatranog, pošto je u pitanju čvor zvezdastog zatvaranja, ono što je prva pozicija tog čvora, biće sledeća pozicija za sve pozicije iz poslednje pozicije tvog čvora. Zašto je to tako? Ovaj čvor zvezdastog zatvaranja u stvari opisuje izraz koji stalno ponavlja (nula ili više puta) ovu sekvencu BC, koja počinje pozicijom 2, a završava pozicijom 3. To znači, kada se ponovi BCBC iza trojke će doći dvojka. Znači evo ovde kao sledeću poziciju od 3 upisujemo 2 i time smo završili sa ovim čvorom zvezdastog zatvaranja.

Čvor iznad njega ignorišemo pošto je to čvor alternative. Idemo na čvor iznad, znači treba upisati 4 kao sledeću poziciju pozicije 1 i kao sledeću poziciju pozicije 3. U trećoj vrsti imamo skup {2,4}, od ranije smo imali 2 a sada smo dodali 4. Time smo završili procesiranje ovog čvora konkatenacije i ostaje koreni čvor, gde beležimo poziciju 5 kao sledeću poziciju od 4. Pošto smo procesirali ceo stablo, završili smo formiranje kompletne tabele i imamo sve sledeće pozicije.

Vidimo da je jedino za poziciju 5 koja je završna pozicija, ostao prazan skup pozicija. Znači ona nema sledeći poziciju jer je to end marker koji ima tu osobinu da se pojavljuje na kraju svake ulazne sekvence kao poslednji simbol tačno jedanput, znači iza njega ne može ništa da se pojavi, tako da je logično da njegova sledeća pozicija bude prazan skup.

Sada kada imamo sve ove informacije, one su nam dovoljne da konstruišemo deterministički automat za ovaj regularni izraz. Od svih ovih informacija koje smo dosad izračunali, većina toga su neki međurezultati koji nam u sada više ne trebaju. Ono što nam treba za konstrukciju automata, to je tabela funkcije sledeća pozicija i treba nam prva pozicija korenog čvora, to je u ovim primeru skup {1,2,4}. Znači to su svi elementi koji su nam potrebni, iz kojih možemo konstruisati deterministički konačni automat.

Na koji način se obavlja ta konstrukcija? Stanjima automata će odgovarati skupovi pozicija. Znači svako stanje će biti neki skup pozicija i pri tome startno stanje će odgovarati onom skupu koji je prva pozicija korena stabla. U našem primeru, to je skup pozicija {1,2,4}.

Potrebno je da postepeno konstruišemo tabelu prelaza determinističkog automata. Ta tabela prelaza treba da ima onoliko kolona koliko ima pozicija, to jest ulaznih simbola u regularnom izrazu. Ovde može da se primeti jedan detalj da nismo u automatu dodali i end marker kao ulazni simbol, znači njega možemo dodati ili ne. Ako želimo onu striktnu definiciju automata, pošto smo mi na neki način veštački dodavali taj end marker regularnom izradu znači sad možemo opet da ga izostavimo Tako da za naš primer ostaju samo A,B,C i D se pojavljuju kao ulazni simboli u regularnom izrazu.

Međutim, ovaj end marker ipak ima uticaja na definiciju automata prema tački donjeg algoritma gde se kaže: stanja prihvatanja su ona koja sadrže poziciju end markera. Kad god unesemo neko stanje -- inicijalno mi ćemo samo prvu vrstu definisati, ostale će još biti nedefinisane -- ta prva vrsta će biti ona prva pozicije korena {1, 2, 4}. Sada proveravamo za ove pozicije 1, 2, 4 da li neki od tih brojeva odgovara markeru kraja. Ako se vratimo na primer našeg stabla, marker kraja označen je brojem 5. U skupu {1, 2, 4} nemamo 5 i zbog toga ovde na kraju pišemo nulu, znači startno stanje je stanje odbijanja.

Kada jednom unesemo neku vrstu i definišemo neko stanje kao skup pozicija, potrebno je da popunimo ovu tabelu prelaza u sledeće stanje. Kako se popunjava ova tabela? Stanja se formiraju i evidentiraju u skupu States a pravi se i tabela prelaza δ po sledećem algoritmu:

```
startno stanje S := prva_pozicija(koren sintaksnog stabla);
inicijalizovati States sa S koje nije markirano;
while (∃ T ∈ States ∧ T nije markirano)
        markiraj T;
        if (pozicija markera kraja ∈ T )
                then označiti T kao stanje prihvatanja;
                else označiti T kao stanje odbijanja;
        end if;
        for (∀a ∈ U)
                N := ∅;
                for (∀p ∈ T)
                        if (simbol na poziciji p = a)
                                then N := N ∪ sledeća_pozicija(p);
                        end if;
                end for;
                δ(T, a) := N;
                if (N ≠ ∅ ∧ N ∉ States)
                        then dodati U kao nemarkirano stanje u States;
                end if;
        end for;
end while
```

Vrsta tabele prelaza, koja je prikazana na sledećoj slici pod (a), za određeno stanje popunjava se razmatrajući pojedine pozicije iz skupa pridruženog tom stanju. Na primer, za stanje {1, 2, 4}:

- razmatramo ulaz u koloni A: određujemo pozicije iz skupa {1, 2, 4} koje odgovaraju listovima označenim sa A. Radi se o poziciji 1 i pri tome je *sledeća_pozicija*(1) = {4}. U tabeli se u vrsti {1, 2, 4} i koloni A upisuje prelaz u stanje {4}. U slučaju da je u skupu {1, 2, 4} bilo više pozicija koje odgovaraju znaku A, skup pozicija novog stanja bio bi unija odgovarajućih skupova sledećih pozicija.
- razmatramo ulaz u koloni B: pozicija 2 iz skupa {1, 2, 4} odgovara slovu B, sledeća_pozicija(2) = {3}, znači da se iz stanja {1, 2, 4} za ulaz B prelazi u stanje {3}.
- razmatramo ulaz u koloni C: nema pozicija u skupu {1,2,4} koje odgovaraju slovu C, znači da se iz stanja {1, 2, 4} za ulaz C prelazi u stanje {}.

- razmatramo ulaz u koloni D: pozicija 4 iz skupa {1,2,4} odgovara slovu D, *sledeća_pozicija*(4)={5}, znači da se iz stanja {1, 2, 4} za ulaz D prelazi u stanje {5}.

Popunjavanje tabele nastavlja se za novodobijena stanja, sve dok se sva stanja ne pronađu i sve vrste tabele ne popune. U nekom trenutku će biti završeno popunjavanje cele tabele bez unošenja novih stanja i time smo znači formirali kompletan konačni automat. Možemo primetiti da stanje {}, koje predstavlja prazan skup pozicija, uvek mora biti stanje odbijanja. Jednom kada automat dođe u njega on uvek ostaje tu, znači svi prelazi iz tog stanja su u to isto stanje, tako da je ovo stanje u stvari stanje greške. Znači ono je ovde eksplicitno naznačeno a mogli bismo ovu vrstu kompletno izostaviti i uopšte i ne pisati ove prazne skupove, nego ako dobijemo prazan skup, ostaviti ulaz prazan, čisto da rasteretimo tabelu prelaza.

Automat sa izostavljenim stanjem greške i preimenovanim stanjima, M = {1, 2, 4}, N = {4}, O = {3}, P = {5} i Q = {2, 4} prikazan je na slici (b). Na taj način smo dobili iz regularnog izraza deterministički konačni automat i kompletirali primer primene algoritma pozicija.

	A	B	C	D	
→ {1, 2, 4}	{4}	{3}	{}	{5}	0
{4}	{}	{}	{}	{5}	0
{3}	{}	{}	{2, 4}	{}	0
{}	{}	{}	{}	{}	0
{5}	{}	{}	{}	{}	1
{2, 4}	{}	{3}	{}	{5}	0

	A	B	C	D	
→ M	N	O		P	0
N				P	0
O			Q		0
P					1
Q		O		P	0

Konačni procesori

U lekciji o konačnim procesorima razmotrićemo kako formalizam konačnog automata proširiti da obavlja nekakvo korisno procesiranje. Pored toga, pomenućemo kako implementirati konačni automat u programskom jeziku Java u nekoliko mogućih varijanti. Dosadašnja definicija konačnog automata definisala ga je kao prepoznavač, odnosno izlaz algoritma je bio binarni -- da li se ulazna sekvenca prihvata ili ne. Procesor može da ima i nekakav bočni efekat, odnosno da uradi neko procesiranje ulazne sekvence sem toga osnovnog izlaza, da li je ulazna sekvenca ispravna ili nije. Neki primeri u kontekstu kompajlera bi bili; Šta može leksički analizator da radi da recimo izračuna binarnu vrednost konstante, ili da recimo neki identifikator smesti u tabelu simbola.

Da bismo pretvorili jedan konačni automat koji je prepoznavač u procesor, možemo svakom ulazu u tabeli prelaza da pridružimo neku akciju. Akcija u opštem slučaju može biti proizvoljna sekvenca programskog koda. Takođe, iz praktičnih razloga, zgodno je u ulaznu azbuku automata dodati marker kraja ulazne sekvence, odnosno end marker, koji se javlja samo jedanput u ulaznoj sekvenci, tačno na njenom kraju, tako da možemo lako posle u implementaciji programa detektovati kada je došao kraj ulazne sekvence. Što se tiče praznih ulaza tabele prelaza oni u praktičnim primenama mogu poslužiti za detekciju i oporavak od leksičkih grešaka.

Uvodni primer

Na jednom primeru videćemo kako konačni prepoznavač proširiti u konačni procesor. Želimo da konstruišemo procesor za izračunavanje vrednosti neoznačenih realnih konstanti. Sintaksa realnih konstanti, znači šta je dozvoljeno a šta je nije, data je nizom primera ispravnih konstanti i primera

neispravnih konstanti. Primeri dozvoljenih konstanti su: 1257,.0392, 1E + 534, 1., 1.E5, 2.0E10 a nedozvoljenih -21, .E, E, E10, 12E.

Na bazi primera može se zaključiti da konstante imaju svoj deo mantise i eksponenta. Mantisa je deo konstante ispred E, ovakva sintaksa ima naziv naučna notacija, znači vrednost mantise, pa E, pa vrednost eksponenta. E je u stvari oznaka za stepenovanje osnove 10 na vrednost eksponenta.

Sama mantisa može da ima svoj celobrojni deo i decimalni deo razdvojeni tačkom. Šta je sve dozvoljeno izostaviti? Mantisa može biti striknto celobrojna, ne mora da ima tačku i decimale. Takođe, mantisa može da ima samo decimale, pri čemu mora da se navede tačka na početku, ali je moguće izostaviti vodeću nulu, na primer .0392. Takođe, ako mantisa ima celobrojni deo moguće ju je završiti tačkom i ne mora se navesti ni jedna decimala. Ali, ako je tačka prisutna u mantisi, onda ili sa leve strane tačke, ili sa desne strane tačke, mora biti bar jedna cifra. U postavci je da primer mantise koja ima samo tačku i to nije dozvoljeno.

Deo koji se odnosi na eksponent, znači E i vrednost eksponenta moguće je izostaviti, ovaj deo je opcioni. Vidimo u postaci primere konstanti koje su bez tog eksponencijalnog dela. Ako se navodi E, iza E mora doći vrednost eksponenta, znači bar jedna cifra eksponenta. Nije dozvoljeno konstantu završiti samo sa E. Takođe ako se navodi E, mora biti nečega ispred E, znači nije dozvoljeno potpuno izostaviti mantisu. Dodatno, konstante su neoznačene kao celina, znači nije dozvoljeno ispred konstante da stoji plus ili minus. Ovde je primer nedozvoljene konstante. Međutim, eksponent je označen iza E, a pre vrednosti eksponenta može stajati plus ili minus. Plus se podrazumeva, tako da ne mora da se navodi.

Prema ovoj definiciji, ukupan broj cifara u mantisi kao i u eksponentu nije ograničen. Očigledno da u realnim primenama, zbog veličine konstante, tu mora biti nekog ograničenja, ali na ne nivou konačnog automata. Mi ćemo automat napraviti tako da na nivou stanja automata ne ograničava eksplicitno koliki je taj broj, pa ovde to nije ni navedeno.

Ovo je samo deo postavke kako izgledaju ispravne konstante. Sledi nastavak postavke. Naš procesor treba da izračuna binarnu vrednost konstante tako što postavlja sadržaje sledeće 4 registra:

- NR (number register) sadrži binarnu vrednost celog broja koji se dobija uklanjanjem decimalne tačke iz mantise;
- ER (exponent register) sadrži binarnu vrednost eksponenta koji je naveden u konstanti (ne vrši se normalizacija mantise);
- CR (count register) sadrži broj cifara iza decimalne tačke;
- SR (sign register) pamti predznak eksponenta (+1 ili -1)

Na primer, za konstantu 12.3E4 registri će imati sledeće vrednosti:

NR	ER	CR	SR
123	4	1	+1

Ovaj binarni oblik nije po onom IEEE standardu za predstavljanje realnih brojeva u pokretnom zarezu nego je osmišljen za potrebe ovog primera, da nam bude lako da prikažemo akcije računanja ovih vrednosti. NR treba da sadrži kompletnu matisu broja kad se iz nje ukloni decimalna tačka, znači za ovu konstantu 12.3 treba da sadrži 123. Registar eksponenta treba da sadrži vrednost eksponenta onako kako je data, nema nikakve normalizacije mantise, znači tačka ostaje tamo gde

jeste. CR govori u okviru ovog NR registra gde se nalazi decimalna tačka, odnosno koliko ima decimala. U ovom konkretnom primeru ima jedna decimala, tako da je CR jednako jedan. Konačno, predznak se odnosi na predznak eksponenta, znači ovde je eksponent podrazumevano pozitivan. Znači generalno SR može biti ili plus jedan ili minus jedan.

Da bismo konstruisali konačni procesor za opisane konstante, prva faza je da konstruišemo konačni automat koji samo prepoznaje ispravne konstante. To naravno možemo uraditi na primer tako što bismo napisali regularni izraz na bazi onih primera koji opisuju ispravne konstante i zatim upotrebili Tomsonov algoritam ili algoritam zasnovan na pozicijama da konstruišemo konačni automat iz regularnog izraza.

Međutim, ja ću ovde pokazati treći način, koji je ad hoc, znači manje formalan od ovih prethodnih algoritama i pogodan je za ručno rešavanje kada nemamo nikakav alat na raspolaganju i za neke jednostavnije primere, ako su suviše komplikovani može se napraviti previd i u toj situaciji bolji su ranije pomenuti formalin pristupi. Ovaj pristup, koji inače može da se neformalno nazove metod enumeracije stanja, polazi od toga da napišemo jedan ili nekoliko karakterističnih primera ispravnog ulaza

3 8 . 7 E -3 i .9 E 2 1.

Primetimo da su ovde malo razmaknuti ulazni znakovi, dakle, ostavili smo malo prostora između svakog znaka. Potom krećemo na postepenu konstrukciju tabele prelaza automata. Dobićemo odmah deterministički automat.

Ono što u početku znamo, to je ulazna azbuka {DIGIT, •, E, SIGN}. Bilo koja cifra predstavljena je ulaznim znakom digit. Pošto manje više sve cifre se isto tretiraju, nije ih potrebno razdvajati, onda bismo samo dobili automat sa više jednakih kolona. Zatim decimalna tačka, zatim slovo E i plus ili minus koji mogu slediti pre eksponenta objedinjeno ćemo predstaviti sa ovim SIGN.

Ovaj postupak da se pojedini ulazni simboli grupišu u klase ulaznih simbola (nekad se to zovu ulazni tokeni da bi se razlikovali od izlaznih tokena koji izlaze iz ovog leksičkog analizatora), zove se *transliteracija* i programski je to moguće lako realizovati tako što su uvede jedan vektor kojim se recimo indeksira, na primer, ASCII kodovima ulaznih simbola tako da na indeksu koji odgovara ASCII kodu cifre 0, upiše se vrednost digit i takođe na indeksima koji odgovaraju ASCII kodovima drugih cifara, svuda se upiše digit. Tako da, kada pročitamo neki znak sa ulaza, mi prvo pristupimo tom nizu i pročitamo vrednost elementa i tako dobijemo taj ulazni token. Na taj način možemo realizovati transliteraciju i dobitak je u tome što nam je automat pregledniji i manji.

Konstanta 38.7E -3 daje sekvencu ulaznih tokena:

DIGIT DIGIT **.** DIGIT E SIGN DIGIT

Pokušavamo za ovaj konkretan ulaz da rekonstruišemo sekvencu promene stanja našeg budućeg automata. To počinjemo tako što uvodimo startno stanje. Ovde su stanje označena brojevima od nula na dalje, tako da krećemo od startnog stanja 0. Znači to napišemo ispod prve ulaznog DIGIT, ali malo ulevo. Sada razmatramo prelaze između stanja traženog automata koje izazivaju ove ulazne sekvence:

DIGIT DIGIT . DIGIT E SIGN DIGIT
0 1 1 2 3 4 5 6

Iz startnog stanja 1 pod ulazom DIGIT prelazi se u novo stanje 1, itd. Pri odluci da li za određeni prelaz uvesti novo stanje ili upotrebiti neko od postojećih, razmatramo dokle je automat došao s nekim prepoznavanjem i šta možemo očekivati u nastavku. Ovde kada dolazi prvi DIGIT možemo

znači ili da ostanemo u stanju 0, li da uvedemo neko novo stanje 1. Šta je razlog da uvedemo novo stanje? Kada uvedemo novo stanje 1 ono će se od 0 razlikovati po tome što je u stanju jedan prihvatljivo da nam dođe E kao sledeći simbol na ulazu, dok u stanju 0 to nije prihvatljivo. Znači nije prihvatljivo da konstanta počinje sa E, ali kada dođe jedan DIGIT onda može da dođe je i E. Zbog toga znači stanje 1 je različito u suštini od stanja 0 i mi ovde uvodimo stanje 1.

Sada kada dolazi sledeća cifra na ulazu, vidimo da je ovde pala odluka da sledeća trazicija bude opet ostajanje u stanju 1, jer je što se tiče automata svejedno da li će u celobrojnom delu mantise doći jedna ili mnogo više cifara. Za sledeće cifre ostajemo u istom stanju. Na taj način nema potrebe da naš automat odbrojava kolika je dužina celobrojnog dela mantise. To se rešava kroz akcije koje ćemo naknadno ubaciti. U samoj definiciji sintakse ove konstante nema ništa o tome koliki je broj cifara mantise, on može biti proizvoljno velik. Znači ostaje se u stanju 1.

Sada kada dolazi na ulazu decimalna tačka, treba doneti odluku da li iskoristiti postojeće stanje (nula naravno ne dolazi u obzir, ali recimo stanje jedan dolazi u obzir) ili uvesti novo stanje. Tu je jedino ispravno da se uvede novo stanje. Zašto? Jer kada bismo ostali u stanju 1, to bi automatski značilo da bi naš automat prihvatao i veći broj tačaka, znači da iza prve tačke dođe još tačaka. Ovako kada odemo u stanje 2, tačka više nije prihvatljiva, kada je jedna tačka već došla. U suštini, stanje dva označava situaciju da je počeo decimalni deo mantise. I tako dalje, do kraja ulazne sekvence koju smo izabrali.

Ako smatrate da ova jedna konstanta nije pokrila sve zanimljive situacije, sasvim ste slobodni da uvedete još neke primere ispravnih konstanti i sprovedete isto ovo procesiranje. Druga izabrana konstanta, .9E21 daje ulaznu sekvencu:

. DIGIT E DIGIT DIGIT

Recimo drugi primer se razlikuje od prvoga po tome što je ovde izostavljen celobrojni deo mantise, znači cela konstanta počinje tačkom. Takođe vidimo da nema predznaka eksponenta. Znači opet imamo E, ali nemamo predznak eksponenta.

Krećemo uvek od stanja 0. Kada dolazi tačka na ulazu, ovde je prva dilema, da li iskoristiti stanje dva u koje smo išli u prvom primeru kada dođe tačka, ili treba uvesti novo stanje. Interesantno je da ovde nije iskorišćeno stanje dva. Zašto nije? Na to možemo odgovoriti time što ćemo pogledati razlike između vrste dva i vrste 7 u dole priloženom automatu. U vrsti 2, prihvatljivo je znači da u stanju 2 posle dođe E odmah, međutim u stanju 7 to nije prihvatljivo. Kada bi u stanju 7 došlo E, od cele mantise ostala bi tačka što smo rekli da ne može, nego kad je tačka mora ili levo od nje ili desno da bude bar jedna cifra. U stanju dva mi znamo da je već prošla jedna cifra pre tačke. a u stanju 7 to nije slučaj, znači stanja 2 i 7 nisu ekvivalentna. I tako dalje, manje-više ovde su iskorišćena već postojeća stanja za ove naredne tranzicije.

.	DIGIT	E	DIGIT	DIGIT
0	7		3	4
			6	
			6	

Na osnovu prethodnog razmatranja popunjavamo tabelu prelaza automata. Svaki put kad definišemo neki prelaz to treba uneti u tabelu prelaza. U prvom primeru sekvence definisali smo prelaz iz stanja 0 kada dolazi digit u stanje jedan, znači definisali smo ćeliju tabele u vrsti 0 i koloni DIGIT, da treba da ima vrednost 1. Zatim, u stanju 1, kada dolazi digit, ostajemo u stanju 1, to je ćelija tabele u vrsti 1 i koloni DIGIT popunenja sa 1 i tako dalje, znači unosimo u tabelu sve ove prelaze.

Pored brojeva stanja navedeni su neformalni opisi stanja. Ovo nije neophodno za konstrukciju automata, ali je korisno radi provere. Stanja prihvatanja su ona stanja u kojima se očekuje kraj konstante, odnosno stanja 1, 2, 3 i 6. Ostala stanja su stanja odbijanja.

		DIGIT	E	•	SIGN	
inicijalno stanje	0	1		7		0
cifra pre opcione tačke	1	1	4	2		1
opciona decimalna tačka	2	3	4			1
cifra nakon decimalne tačke	3	3	4			1
slovo E	4	6			5	0
predznak eksponenta	5	6				0
cifra eksponenta	6	6				1
tačka iza koje mora ići cifra	7	3				0

Nepopunjene ulaze tabele smatramo prelazima u posebno stanje E, takozvano stanje greške (radi se o stanju odbijanja i kada automat u toku rada jednom dođe u to stanje on ostaje u njemu do kraja ulazne sekvence).

Postavlja se pitanje, da li smo završili sa konstrukcijom tabele prelaza? Odgovor je da nismo, to jest, ostala je još jedna stvar da se proveri, a to, je kada prođete kroz sve primere koje ste odabrali i unesete sve ove prelaze koje ste definisali, potrebno je razmotriti svaki ulaz koji je u tabeli prelaza ostao prazan. Pri tom razmatranju treba odlučiti da li je ulaz prazan zato što on zaista predstavlja grešku (znači nelegalnu situaciju) ili je ulaz prazan, a reprezentuje neku ispravnu situaciju, samo mi nismo imali dovoljno karakteristične primere da pokrijemo tu situaciju.

Recimo, ako smo u inicijalnom stanju 0 i dolazi E to je u stvari pogrešna situacija, pošto to odgovara onom neispravnom primeru gde odmah dolazi E na ulazu. Ovde je to zaista prelaz u stanje greške i zato će zaista ostati prazan. Zatim, ako smo u inicijalnom stanju, a dolazi plus ili minus (predznak broja) to smo isto rekli, pošto su kompletne konstantne neoznačene po postavci, znači to isto ostaje prelaz u stanje greške. Imamo i drugačiji primer. prelaz iz jedan direktno u 4. Ovo je naravno finalni automat u kome je sve popunjeno, ali kada samo prođemo kroz ova dva primera ulaza, ove četvorke ne bi bilo, znači ovaj ulaz bi bio prazan. U stanju jedan kada dolazi E to odgovara situaciji kada imamo isključivo celobrojnu mantisu, znači nemamo tačku i decimale i to naravno nismo pokrili ni jednim od primera. Znači ovo bi naknadno uneli direktnim razmatranjem, u prvoj vrsti i koloni E ovog ulaza. I tako dalje, znači kada prođemo kroz sve te prazne ulaze i uverimo se da li su zaista greška ili ih prosto nismo pokrili, mi smo konačno dobili prepoznavač.

Automat dobijen metodom enumeracije simbola ne mora nužno biti sa najmanjim mogućim brojem stanja. Kada pogledamo ovo finalno rešenje, onda vidimo sledeće: vrste dva i vrsta tri su potpuno identične, znači svi prelazi su isti i oba stanja su stanja prihvatanja, što će reći da su stanja 2 i 3 ekvivalentna i to u suštini znači da smo mi ovde mogli iskoristiti stanje dva, znači nije bilo neophodno uvoditi novo stanje.

Da li je bila greška uvesti novo stanje? U suštini, nije. Kad god ste znači u dilemi da li iskoristiti postojeće stanje ili uvesti novo, bolje je i sigurnije uvesti novo stanje, makar se ono naknadno pokazalo ekvivalentnim nekom postojećem, jer u krajnjoj liniji, to će dovesti do samo malo većeg automata, znači njegov rad će biti ispravan i postupku minimizacije to stanje lako može naknadno da su elimiriše. Međutim, mnogo opasnija greška je da se iskoristi isto stanje u nekoj situaciji gde

u stvari ponašanja nisu ista. Greška toga tipa da se iskoristi postojeće stanje tamo gde bi trebalo novo stanje da se uvede, rezultuje u tome da se promeni jezik automata koji se prihvata. Znači kad god ste u nekoj dilemi da ne možete da razrešite, uvek uvedite novo stanje i to je znatno sigurnija varijanta. Kriterijum da li treba da bude isto stanje ili različito, je prema tome kakvo je ponašanje automata u nastavku. Prvi kriterijum je da li, kada biste sekvencu presekli na tom mestu, recimo baš u stanju 2, da li je ona prihvatljiva? Vidimo da ovde jeste prihvatljiva. Znači to je prvi kriterijum, a onda naravno, ako nije kraj, šta može u nastavku, da li je u jednom i u drugom stanju mogu iste stvari da dođu a da to bude ispravno.

Sada je potrebno preći na drugu fazu, odnosno od prepoznavača napraviti procesor. Procesor se pravi tako što se u svaku od ovih ćelija tabele prelaza doda neka akcija. Da bismo dodali akciju, razmotrimo, na slici u nastavku, donekle modifikovanu tabelu prelaza u odnosu na prethodnu sliku.

Prvo, objedinjena su stanja 2 i 3 pošto smo videli da su ekvivalentna, u jedno stanje koje se sada zove stanje 23.

Druga izmena je u eksplicitnom dodavanju kolone za marker kraja ulaza. Automat nema više kao kod prepoznavača one nule i jedinice, koje je stanje prihvatanja, a koje stanje odbijanja, nego sa to eksplicitno pretvara u akcije -- gde god je bilo stanje prihvatanja, mi sad imamo akciju YES kao završnu akciju, gde konstatujemo da je konstanta ispravna, ali videćemo da postoji i neko procesiranje da bismo dobili vrednost konstante. Znači gde god su bile jedinice u prvobitnom automatu ovde su stavljene akcije YES.

Takođe, drugi prelazi su malo promenjeni. U prvoj ćeliji tabele nalazilo se samo 1, a sada piše 1a. To i dalje znači da, kada smo u stanju 0 i dolazi digit na ulazu idemo u stanje 1, ali ovo a služi samo tome da jedinstveno označi ovu ćeliju i da je razlikujemo, recimo, od ove druge ćelije gde je isto prelaz u stanje jedan, ali tu smo sada dodali 1b, da te dve ćelije razlikujemo. Znači ove dodatne oznake služe samo za identifikaciju ćelija.

Mi možemo napisati akcije koje svakoj od tih ćelija pripadaju. Automat, osim što će raditi onaj svoj osnovni posao, to je znači procesirati ulaz znak po znak, konsultovati tabelu prelaza i menjati stanja svaki put kada pročita neki od ulaza ove tabele prelaza, on će sprovesti odgovarajuću akciju.

	DIGIT	E	•	SIGN	⊣
0	1a		7		
1	1b	4a	23c		YES 1
23	23a	4b			YES 2
4	6a			5	
5	6b				
6	6c				YES 3
7	23b				

U ovom našem konkretnom primeru akcije se svode na ažuriranje vrednosti nekog od onih četiri regstra po potrebi. Akcije su navedene u nastavku. Operator val(cifra) daje binarnu vrednost cifre. Ovo rešenje, kao, na kraju krajeva, sva programerska rešenja, nije jedinstveno, moguće su neke male varijacija da se dobije isti krajnji efekat.

```
1a:  NR := val(DIGIT)
1b:  NR := NR * 10 + val(DIGIT)
23a: CR := CR + 1
     NR := NR * 10 + val(DIGIT)
23b:    CR := 1
     NR := val(DIGIT)
23c: CR := 0
7:   ———
YES1: ER := 0
      CR := 0
```

```
4a:  CR := 0
4b:  ———
5:   SR := val(SIGN)
6a:  SR := +1
     ER := val(DIGIT)
6b:  ER := val(DIGIT)
6c:  ER := ER * 10 + val(DIGIT)
YES2: ER := 0
YES3: ———
```

Da vidimo, na primer, šta bi trebalo da bude akcija kada dolazi prva cifra mantise na ulazu. Rekli smo da je NR registar koji pamti vrednost mantise i pošto smo u startnom stanju, dolazi prva cifra mantise. Znači treba tu cifru upisati u registar. Pošto je digit možda nekakav kod na primer ASCII, u svakom slučaju digit ima neku svoju vrednost, mi uzimamo da val(digit) u stvari znači klasni deo ulaznog tokena, odnosno vrednost konkretne cifre (0, 1 do 9). i to direktno upisujemo u registar NR. To je ta prva cifra mantise koja dolazi.

Sada, na primer, možemo nastaviti razmatranje ovom akcijom 1b. Akcija 1b nastupa kada dolaze sledeće cifre celobrojnog dela mantise pre decimalne tačke. Šta onda treba raditi? Tada već imamo neku vrednost, recimo onu prvu ili prethodne cifre su zabeležene u registru NR. Kako sada da ažuriramo vrednost registra NR. Podrazumeva se da je mantisa u decimalnom sistemu. Znači treba pomnožiti prethodnu cifru težinskim faktorom koji odgovara osnovi brojnog sistema, znači pomnožimo prethodnu vrednost NR sa 10 i dodamo vrednost cifre koja je upravo došla.

Ako je mantisa 12, prvo je došla 1, onda pomnožimo jedan puta 10 je 10, plus 2 je 12., posle dolaska te druge cifre. Ako dođe treća cifra 3, množimo ovo 12 sa 10 to je 120, plus 3 to je 123, tako formiramo NR. I tako dalje, čitaocu se ostavlja da prouči i sve ostale akcije, u zavisnosti od stanja obrade biće ažurirani neki od registara.

Za neke prelaze nije potrebna nikakva akcije, jer su sve odgovarajuće akcije već urađene pre, ili će biti urađene posle. Očekujemo da kada dođemo do akcija YES, da smo finalno ažurirali vrednosti sva četiri ova registra koja su navedena u postavci. Time bi bio završen taj konceptualni primer kako prepoznavač proširiti u procesor.

Moguće su i druge varijante proširivanja prepoznavača u procesor. Primer koji razmatrali odgovara definiciji Milijevog automata. Nasuprot tome, kod Murovih automata, akcija bi bila pridružena stanju, a ne svakom prelazu, znači kada automat dođe u određeno stanje preduzima određenu akciju.

Alat jflex za automatsko generisanje konačnog procesora iz ulazne specifikacije (regularni izrazi+akcije), proširuje konačni automat na način drugačiji od svih prethodno opisanih. Kod njega se dešava kompletno prepoznavanje nekog regularnog izraza, a tek onda, na kraju kad je čitav regularni izraz prepoznat, praktično bi to bilo tek u ovim YES akcijama, dozvoljeno je da se izvrši programski kod koji je definisan kao akcija pridružena izrazu, a toj akciji se dostavlja kompletan prepoznati deo ulazne sekvence kao ulazni string.

Programska realizacija konačnog automata

Sledi izlaganje o različitim varijantama kako programski implementirati konačni automat. Konačni automati su popularni u računarskoj praksi i u otvorenom kodu može se pronaći veliki broj različitih implementacija konačnih automata. Ponašanje različitog softvera može da se opiše konačnim automatima, na primer sistemi pokretani događajima kao što su grafički korisnički interfejsi, u telekomunikacijama razni komunikacioni protokoli, zatim u upravljačkim (embedded) sistemima često može da se izrazi znači funkcija upravljanja nekakvom state mašinom i tako dalje. U ovom poglavlju razmatraće se neke karakteristične varijante implementacije koje se često primenjuju. Svaka ima svoju svrhu i mogućnosti. Dali smo im određena imena.

Implementacija sa *implicitnom predstavom* funkcije prelaza podrazumeva da je stanje određeno mestom u programu koji se izvršava. U principu možete u nekom programu da uopšte nemate promenljivu koje označava tekuće stanje, nego isključivo deo koda u kome se program trenutno nalazi implicitno određuje tekuće stanje.

E*ksplicitna predstava* stanja je najintuitivnija za implementaciju automata. Kada funkciju prelaza predstavimo tabelarno, pravolinijski je uvesti strukturu podataka dvodimenzionalna matrica koja pamti elemente funkcije prelaza, uvesti promenljive koje pamte tekuće stanje i tekući ulaz i potom iskoristiti vrednosti te dve promenljive za indeksiranje u toj matrici i na taj način čitati menjati stanje tokom rada automata.

Objektno-orijentisana implementacija sledi projektni uzorak (engl. design pattern) State prema onoj poznatoj knjizi Gang of four Design patterns elements of reusable object-oriented software, koji je pominjan u okviru predmeta projektovanje softvera.

U nastavku ćemo proučiti na primerima Java implementacije za implicitnu i eksplicitnu predstavu funkcije prelaza i, takođe, jednu moguću implantaciju koja je objektno-orijentisana i prati projektni uzorak State.

Prvo da razmotrimo proceduralne realizacije gde ne moramo koristiti objektno-orijentisani jezik. Na slici vidimo jedan mali konačni automat koji ima samo dva stanje, implicitno ima i treće, stanje greške, pošto vidimo da je druga ćelija prve vrste prazna. Vidimo da je u pitanju prepoznavač, lako bismo ga proširili u procesor, ako bi bilo potrebno.

	a	b	
→A	B		1
B	A	B	0

Eksplicitna predstava funkcije prelaza

Prvo ćemo pogledati realizaciju sa eksplicitnom predstavom funkcije prelaza, pošto je ona najbliža onom algoritmu kako smo definisali rad konačnog automata.

```
class FSM_TT
{

enum State {
    A( true ),
    B( false ),
```

```
    ERROR( false );

    final boolean accepting;

    State( boolean accepting ) {
        this.accepting = accepting;
    }
}

static State state;
static State transition[][] =
{
    { State.B, State.ERROR},
    { State.A, State.B},
    { State.ERROR, State.ERROR}
};
  static void handleInput(char ch)
  {
    int index;
    switch( ch )
    {
      case 'a':
        index=0;
       break;
      case 'b':
        index=1;
       break;
      default:
        state=State.ERROR;
        return;
    }
    state = transition[state.ordinal()][index];
  }

  public static void main(String[] args) throws java.io.IOException
  {
    int ch;
    state = State.A;
    while ((ch = System.in.read ()) != '\n')
      handleInput((char)ch);

    if (state.accepting)
      System.out.println("Prihvata");
    else
      System.out.println("Odbija");
  }
}
```

Pošto je u pitanju Java, pakujemo sve u jednu klasu nazvanu FSM_TT (od transition table) da nas asocira da postoji tabelarna predstava funkcije prelaza. Stanja programski uvek možete realizovati tako da za identitet stanja uvedete cele brojeve koji predstavljaju različita stanja, ali da bismo održali simbolička imena stanja koristićemo nabrojivi tip podataka. Tako da imamo simboličke konstant A, B i Error. Kod Jave, enum tip podataka pomalo liči na klasu. Mi smo ovde instancirali, od ovog nabrojivog tipa State, prvu konstantu A i ovo true u zagradi označava jedan dodatni atribut konstante. Slično kao kod klase, možete znači uvesti jedno ili više polja po potrebi. Ovde smo

uveli jedno polje accepting logičkog tipa i iskoristili da svako stanje ima atribut *accepting*, da li je stanje prihvatanja ili ne. Tako da ovom enum tipu moramo napisati i jedan konstruktor. Svaki put kad instanciramo jedno stanje moramo, moramo navesti, za ovaj atribut accepting, da li ima vrednost True ili False. To se radi koristeći parameter u konstruktoru.

Prema ovom kako je ovde napisano, A je stanje prihvatanja, a B i Error su stanja odbijanja. Ostatak ove klase FSM_TT predstavlja elemente našeg konačnog automata. Potrebno je da imamo jednu promenljivu koja reprezentuje tekuće stanje, to je ova promenljiva state, onog enum tipa State.

Zatim moramo imati dvodimenzionalnu matricu koja predstavlja tabelu prelaza i odmah je ovde inicijalizujemo. Elementi te matrice imaju vrednosti onog nabrojivog tipa, znači pamte pojedina stanja. Ovde inicijalizujemo matricu po vrstama. Prva vrsta, prva kolona je prelaz u stanje B, a druga kolona je prelaz u stanje greške što naravno odgovara prvoj vrsti ove tabele prelaza, za stanje A. Druga vrsta je prelaz u A, prelaz u B, to je za stanje B. I konačno, treća vrsta je za stanje greške, znači to je ostajanje u stanju greške.

Na taj način smo definisali funkciju prelaza i, zatim, imamo još dve funkcije, pomoćnu funkciju handleInput i konačno funkciju main koja realizuje algoritam rada automata. Prvo ćemo objasniti funkciju main, pa ćemo se onda osvrnuti na ovu pomoćnu funkciju handleInput.

Funkcija main implementira osnovni algoritam rada determinističkog konačnog automata. Automat počinje rad od početnog stanja. Ovde je navedeno da je početno stanje A tako da inicijalizujemo promenljivu tekućeg stanja na stanje A. U while petlji učitavamo znak po znak sa ulaza. Za to koristimo Java funkciju za čitanje sa standardnog ulaza jednog znaka. Kraj reda nam izigrava marker kraja ulaza. Kada se pročita jedan znak, jedna iteracija while, obrada jednog znaka, izdvojena je u pomoćnu funkciju handleInput kojoj se prosleđuje ovaj tekući ulazni simbol. Šta u stvari radi funkcija handleInput? Ona radi promenu tekućeg stanja, na bazi sadašnjeg stanja i tekućeg ulaznog simbola.

U while petlji funkcije main obrada se ponavlja dok se ne dođe do kraja ulaza, to jest do novog reda, Jednom kada se dođe do kraja ulaza, potrebno je ispitati da li ono stanje, u kom se u tom trenutku automat zatekao, da li je stanje prihvatanja ili ne. Znači ispitujemo vrednost atributa acccepting i ako je u pitanju stanje prihvatanje, onda ovde ispisuje program da je finalni rezultat da se sekvenca prihvata, inače se ispisuje da se sekvenca odbija.

Sada samo još treba videti kako izgleda procesiranje jednog ulaznog simbola, što je izdvojeno u ovu pomoćnu funkciju handleInput. Tu treba pročitati transition tabelu, to jest matricu, za tekuće stanje i tekući ulaz. Pošto tekući ulazni znak ima svoj kod, a u matrici imamo kolone koje redom odgovaraju pojedinim ulazim simbolima, ova switch petlja služi da mapira kod znaka u odgovarajući indeks kolone tranzicione matrice.

Znači, ako na ulazu dođe znak A, to je u stvari nulta kolona, ako dođe znak B, to je prva kolona, a ako dođe bilo koji drugi znak na ulazu -- u suštini ne očekujemo nijedan znak osim A, B i na kraju novi red -- onda mi znamo da je u pitanju neispravna ulazna sekvenca i onda automat postavljamo u Error stanje pošto nismo predvideli u onoj matrici nijednu kolonu osim 0 ili 1. Ovako se znači štitimo od pogrešnog ulaza i naravno pri tome odmah se vraćamo iz ove handleInput funkcije, znači izvršili smo odmah prelaz u stanje greške. Ako nije u pitanju greška, odnosno ako je na ulazu došlo A ili B, sada imamo vrednost ovog indeksa (nula ili jedan) i onda konsultujemo matricu transition. Vrstu određujemo na bazi tekućeg stanja, kod enum tipa moramo da uzmemo ordinal da bi smo dobili celobrojnu vrednost stanja. Znači, ako je stanje A, pošto smo ga prvog naveli u

definiciji, njegov ordinal je nula, znači biće pročitana nulta vrsta, a Index određuje kolonu (0 ili 1). Matrica nam daje stanje, koje će postati tekuće stanje za sledeću iteraciju.

Implicitna predstava funkcije prelaza

```
class FSM
{

enum State {
    A( true ),
    B( false ),
    ERROR( false );

    final boolean accepting;

    State( boolean accepting ) {
        this.accepting = accepting;
    }
}

static State state;

  static void handleInput(char ch)
  {
    switch( state )
    {
      case A:
        if ( ch=='a' )
          state = State.B;
        else
          state = State.ERROR;
        break;

      case B:
        if ( ch=='a' )
          state = State.A;
        else if ( ch=='b' )
          state = State.B;
        else
          state = State.ERROR;
        break;

      case ERROR:
          //state = State.ERROR;
        break;
    }
  }

  public static void main(String[] args) throws java.io.IOException
  {
    int ch;
    state = State.A;
    while ((ch = System.in.read ()) != '\n')
      handleInput((char)ch);
```

```
    if (state.accepting)
      System.out.println("Prihvata");
    else
      System.out.println("Odbija");
  }
}
```

Po čemu se razlikuje varijanta sa implicitnom predstavom funkcije prelaze? Reč je o vrlo sličnom programu kao ovaj prethodni. On ima istu strukturu, sve je sada upakovano u klasu koja je nazvana FSM. Takođe smo za stanja, da bismo imali simboličke oznake, opet uveli enum tip koji ima svoj atribut accepting, da bismo znali da li je stanje prihvatanja ili ne

Iako je implicitna predstava stanja, ovde je ostala promenljiva koja reprezentuje tekuće stanje. Zašto je to tako? To je isključivo posledica ograničenja programskog jezika Java. Recimo kada biste implementirali ovaj automat u programskom jeziku C to vam ne bi trebalo. U kom detalju je razlika između Ca i jave, pa da Cu ne treba, a Javi treba? Problem je u tome što Java nema goto naredbu. U programskom jeziku C imate goto naredbu i možete navesti neku oznaku u programu (labelu) na koju želite da skočite. U principu, kada imate taj eksplicitan goto, vama ova promenljiva ne treba. Međutim, Java to nema i onda moramo na neki način simulirati taj goto. To radimo jednom switch petljom. Kada objasnimo ostatak programa, vratićemo se na ovaj detalj.

Ceo ostatak programa je praktično isti. Main program je potpuno identičan onom prethodnom. Isto imamo inicijalizaciju početnog stanja, while petlju koja čita sa ulaza znak po znak, dok se ne dođe do novog reda. Ta jedna iteracija realizovana je u ovoj pomoćnoj funkciji handleinput kojoj se prosleđuje tekući ulazni znak, znači sve potpuno isto i na kraju se na samo ispituje, kao i ranije, da li u pitanju stanje prihvatanje ili odbijanja.

Jedina razlika u ovom kodu u odnosu na prethodnu implementaciju je u ovoj handle input funkciji. Kako možemo u Javi realizovati tu implicitnu predstavu? Nema više matrice koja reprezentuje funkciju prelaza automata, nego to je sve ukodirano u sam program. Prva vrsta tabele prelaza automata sada je reprezentovana programskim segmentom switch. Znači unutar ove switch petlje po vrednosti stanja, promenljiva State se koristi kao uslov switch (odnosno case, razgranatog goto). Slučaj (case) A odgovara prvoj vrsti tabele prelaza. Ako je u pitanju stanje A, onda ispitujemo da li je ulazni simbol a, i ako jeste, idemo u stanje B kao što nam ova slika govori. Ako ulazni simbol nije a, znači bilo šta drugo, idemo u stanje greške. Ovde bi to trebalo da bude eksplicitno za simbol B, ali u stanje greške se ide i ako je bilo šta drugo, ako dođe neki nelegalni znak sa ulaza, zato je ovde stavljeno else.

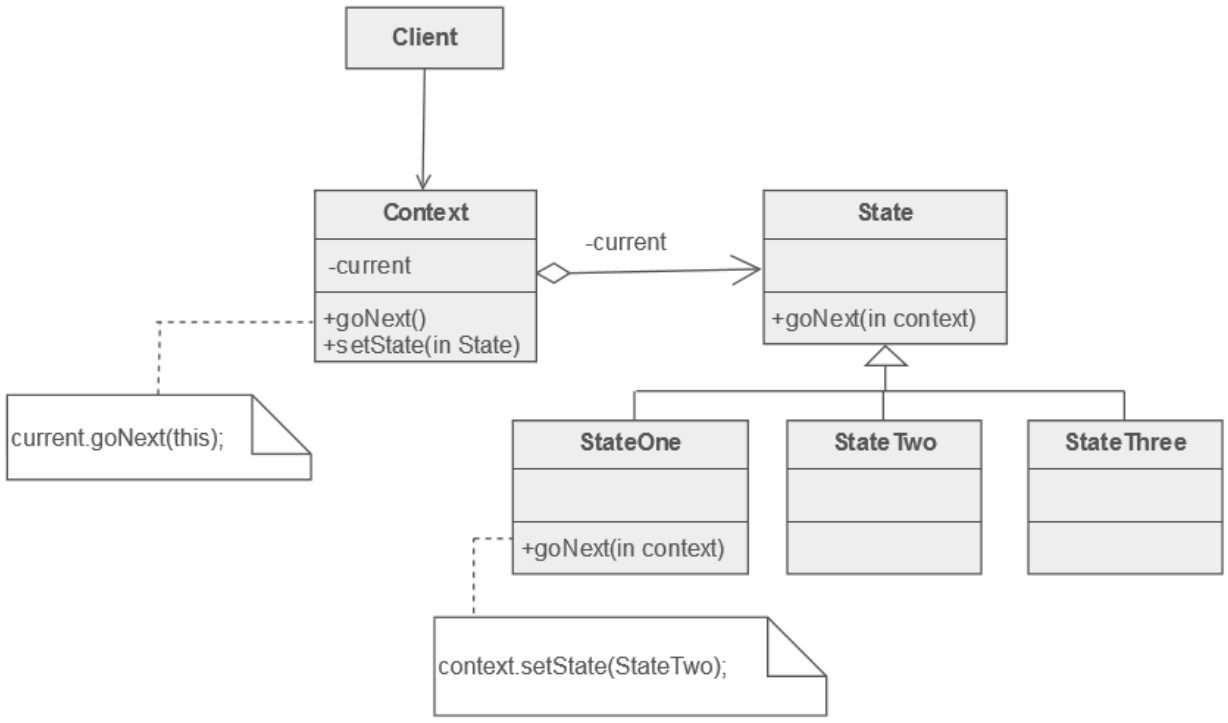
Slično tome, slučaj B odgovara drugoj vrsti tabele prelaza automata. Ako sa ulaza dođe a ide se u stanje A, znači moramo eksplicitno ažurirati promenljivu state. Inače, ako na ulazu dolazi b,i idemo u stanje B, znači dodeljujemo promenljivoj state konstantu State.B. Inače (za nelegalna ulaze) idemo u stanje greške. Konačno imamo i za stanje greške njegov case. Pošto tu nema nikakve tranzicije, znači jednom kada dođemo u stanje greške, to je došlo iz switch-a znači promenljiva state je već imala vrednost error, praktično ostajemo u stanju greške, tako da je ovo zakomentarisano, znači nije ni potrebno vršiti eksplicitnu dodelu, jer promenljiva state već ima tu vrednost.

Kompletan handleinput, odnosno određivanje sledećeg stanja, radi se unutar switch-a. Vidimo da mesto u kodu odgovara stanju u kome se automat u datom trenutku nalazi. Koja bi razlika bila u Cu? U Cu mogli bismo isto ovako raditi ali u Cu postoji i varijanta da mi uopšte ne koristimo

switch, nego da se umesto dodele gde je odredište, na primer, State.B, da se tu eksplicitno napiše goto B. Tamo gde u Javi stoji slučaj B, u Cu bi pisala labela B: kao odredište skoka. Na isti način bi se rešila i sva ostala stanja i prelazi, tako da switch ne bi ni bio potreban. U principu tada ceo ovaj deo ne bi bio išao u handleInput, nego bi išao direktno u glavni program. Stanje A, to jest startno stanje, bi bilo navedeno prvo, tako da bi program u prvoj iteraciji naišao prvo na A, a onda bi dalji tok programa određivale ove goto naredbe. Zašto Java nema goto naredbu, kao ni većina modernih programskih jezika? U nekom trenutku goto je proteran iz modernih programskih jezika jer je dovodio do suviše nečitljivih i nestrukturiranih programa, znači smatran je štetnom programskom naredbom. Međutim, ovde vidimo da, u nekim kontekstima, ako znamo šta radimo, ako je cela stvar jasna i šablonske strukture, on može biti potencijalno koristan. Naravno moguće ga je putem switch-a simulirati ako ga nemamo.

Objektno-orijentisana implementacija

Do sada je bilo reči o proceduralnim načinima realizacije, koje možete upotrebiti u jezicima koji nemaju objekte i objektnu orijentaciju kao što je C. Međutim, ako želimo objektno-orijentisani pristup realizaciji konačnog automata, jedan od načina, koji je da kažemo standardni, je projektni uzorak State, iz ove poznate knjige, u stvari nešto što je ciljano za implementaciju state mašine. Projektni uzorak Stanje reprezentovan je ovim klasnim dijagramom.



Na dijagramu vidimo nekoliko klasa. Stanja automata reprezentovana su tako što postoji jedna apstraktna klasa State (koja inače može da bude interfejs, u realizaciji koji ću ja dati State će biti jedan interfejs) i potom za konkretan automat koji ima određen broj stanja, treba za svako konkretno stanje ili naslediti apstraktnu klasu ili implementirati interfejs. U primeru koji sledi za svako konkretno stanje imamo implementaciju interfejsa State.

Šta je zajedničko tim stanjima? Svako stanje je odgovorno za svoju vrstu tabele prelaza, znači ima nekakav metod goNext koji na bazi ulaznog konteksta, to je u stvari onaj tekući ulazni znak, govori

koje stanje je sledeće. Na dijagramu imamo klasu Context. Ona reprezentuje ostatak automata van onoga što je pokriveno jednim stanjem, ili grubo gledano, to je onaj deo ponašanja koji smo u ranijim proceduralnim primerima imali u glavnom (main) metodu.

Kontekst -- koje su njegove odgovornosti -- da pamti koje stanje u automatu je tekuće stanje. Znači kontekst ima jedno polje current koje je referenca na neko od ovih stanja koje smo instancirali. Svako stanje mora da ima jednu instancu i referenca na odgovarajući objekat označava koje je stanje tekuće. Druga odgovornost Konteksta je da procesira ulazni niz, znači kontekst zna koji je kompletan ulazni niz i iz njega izdvaja redom znak po znak, da bi to prosledio tekućem stanju. Procesiranje se obavlja tako što se kontekst obrati svom tekućem stanju (ovom servisu goNext), prosledi sebe ili tekući ulazni znak, a onda tekuće stanje (to se prosledi preko polimorfnog mehanizma nekoji od ovih metoda) u okviru svog goNext metoda odredi koje stanje treba da bude sledeće.

Potom stanje mora da se obrati kontekstu, pošto je kontekst jedini odgovaran za promenu stanja. Kontekst mora da ima setState metod, na taj način se kontekstu saopšti koje stranje treba da bude tekuće stanje u sledećoj iteraciji. Na taj način se vrši obrada u goNext metodu. Posle toga se kontrola opet vrati Kontekstu, koji sada za sledeću iteraciju, za sledeći ulazni znak, ponovo poziva goNext. Naravno, moguće da je promenjeno tekuće stanje, pa će se obratiti nekom od ovih drugih objekata i tako dalje. Znači, vidimo da su raspodeljene odgovornosti između konteksta i stanja. Svako stanje odgovorno je samo za svoju vrstu tabele prelaza, a kontekst je odgovoran da pamti koje je tekuće stanje i da pamti ulazni niz i stanje obrade ulaza. Klijent je neko ko spolja aktivira ceo automat.

Kako je ovaj šablon moguće implementirati? Ovo je jedan pristup implementaciji šablona Stanje.

```
class FSM_OO
{
  State A = new StateA();
  State B = new StateB();
  State ERROR = new StateERROR();

  interface State {
    boolean accepting();
    void handleInput(FSM_OO fsm, char ch);
  };


  class StateA implements State
  {
    public boolean accepting() { return true;}
    public void handleInput(FSM_OO fsm, char ch)
    {
        if ( ch=='a' )
          fsm.setState(B);
        else
          fsm.setState(ERROR);
    }

  };

  class StateB implements State
  {
    public boolean accepting() { return false;}
```

```
    public void handleInput(FSM_OO fsm, char ch)
    {
        if ( ch=='a' )
          fsm.setState(A);
        else if ( ch=='b' )
          fsm.setState(B);
        else
          fsm.setState(ERROR);
    }

  };

    class StateERROR implements State
  {
    public boolean accepting() { return false;}
    public  void handleInput(FSM_OO fsm, char ch)
    {
       fsm.setState(ERROR);
    }

  };

  private State state;
    public void setState(State state) {
        this.state = state;
    }

    public State getState() {
        return state;
    }

    void handleInput(char ch)
    {
       state.handleInput(this,ch);
    }

  public static void main(String[] args) throws java.io.IOException
  {
    int ch;
    FSM_OO fsm = new FSM_OO();
    fsm.setState(fsm.A);
    while ((ch = System.in.read ()) != '\n')
      fsm.handleInput((char)ch);

    if (fsm.getState().accepting())
      System.out.println("Prihvata");
    else
      System.out.println("Odbija");
  }
}
```

Imamo Java klasu FSM_OO u koju je sve upakovano. Uvedeno je apstraktno stanje reprezentovano interfejsom State i taj interfejs ima goNext funkciju koja je ovde nazvana handleInput da bi asocirala na ranija proceduralna rešenja. Dodatno, svako stanje zna da kaže za

sebe da li je stanje prihvatanja ili ne. To je isto odgovornost stanja da se izjasni da li je stanje prihvatanja ili nije. To će naravno kontekst da proverava na kraju procesiranja.

Naš automat, to je onaj primer od ranije, implementira tri stanja, A, B i Error. Znači moramo imati tri klase. Klasa StateA implementira interfejs State, mora da implementira njegov accepting() metod. Stanje A vraća True, pošto spada u stanje prihvatanja. U handleInput funkciji treba da se dostavi kontekst i mala izmena u odnosu na ono što smo videli u dijagramu šablona je što dostavljamo i tekući ulazni znak eksplicitno kao parametar. U šablonu u goNext funkciji (koja odgovarala ovoj handleInput funkciji) kontekst je dostavljao referencu na sebe. U posmatranoj implementaciji nije uvođena posebna klasa Kontekst, nego ova klasa FSM_OO u koju je sve upakovano, izigrava kontekst tako da se ona dostavlja i dodatno se dostavlja tekući ulazni znak.

Kako se implementira handleInput (odnosno goNext) funkcija? Stanje A odgovara za svoju, prvu vrstu tabele, a to je da, ako dođe a na ulazu treba promeniti stanje u B, a ako dođe b na ulazu, ili bilo šta drugo, treba promeniti stanje u Error. Tako da ispitujemo za ovaj ulazni znak da li je a i ako jeste onda se vrši promena stanja: obraćamo se kontekstu koji je prosleđen kao parameter. Kontekst ima svoj metod setState i prosleđujemo mu objekat klase B jer ovde, na samom početku, mi smo za svako stanje instancirali po jedan objekat. U ovom slučaju prosleđujemo objekat B, inače prosleđujemo objekat Error. Ovako kako je implementirano stanje A implementirana su i sva ostala stanja. To je stanje B, ono odgovara za svoju vrstu tabele prelaza i stanje greške (znači samo se ostaje u stanju greške).

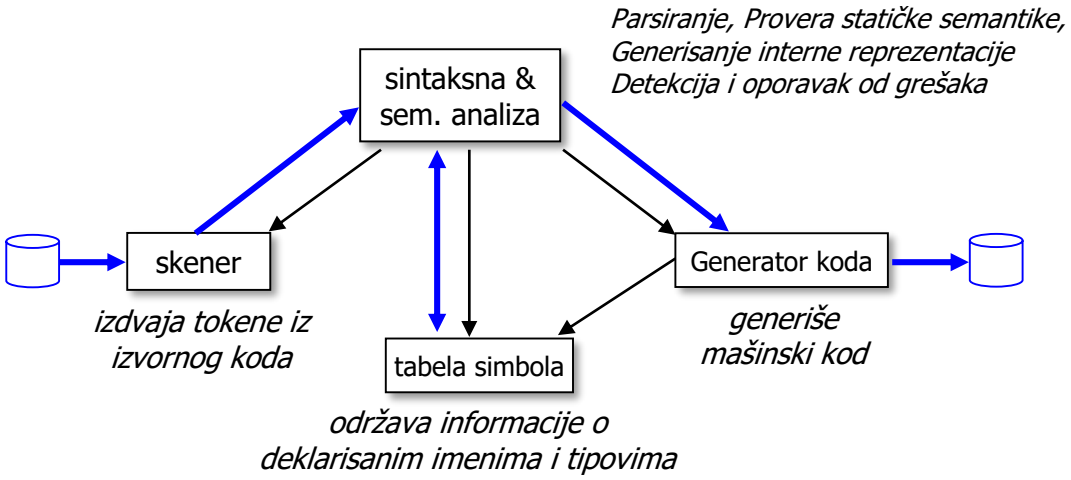
Ostatak koda je kontekst, to jest, klasa FSM_OO i u kontekstu imamo tekuće stanje, znači to je promenljiva state koja ima vidljivost private. Da bismo promenili stanje moramo pozvati metod setState i on naravno ažurira koje je stanje tekuće. Takođee sa getState klijent može, ako želi, da utvrdi koje je tekuće stanje automata, na primer na kraju rada. Servis getState nije bio naznačen na onom dijagramu, ali ga je moguće u implementaciji po potrebi dodati.

Metod handleInput() klase FSM_OO je u stvari ona goNext funkcija u kontekstu. Rekli smo da ona u stvari samo delegira tekućem stanju da obradi tekući ulaz, tako je to definisano i u šablonu. Konačno imamo main funkciju. U ovom primeru nemamo posebnog klijenta van ove klase nego u suštini klijent je ta main funkcija. Ona treba da instancira kontekst, odnosno ovu celu klasu FSM_OO, zatim da postavi da je inicijalno stanje A i onda da se unutar while petlje obrađuje znam po znak dok se ne dođe do kraja ulaza. Zamoli se kontekst da obradi taj znak, a kontekst će to onda da delegira tekućem stanju. Kada se dođe do kraja ulazne sekvence onda može da se ispita za stanje, koje je tekuće u tom trenutku, da li je stanje prihvatanja i da se ispiše odgovarajuća poruka.

Sintaksna analiza

Uvodno poglavlje sintaksne analize objasniće ulogu sintaksno-semantičke analize u okviru prevodioca. Razmotrićemo neke osnovne pojmove i formalizme na kojima se zasnivaju algoritmi sintaksne analize kao što su bezkontekstne gramatike. Videćemo neke osobine gramatika, dobre i loše i daćemo jednu opštu klasifikaciju gramatika, da vidimo u okviru koje klase spadaju gramatike koje su nama od interesa za programske jezike.

Uloga sintaksno-semantičkog analizatora u kompajleru prikazana je šematski na ovoj slici na kojoj su prikazane glavne funkcionalne celine kompajlera. Sintaksno-semantički analizator zadužen je za parsiranje, znači određivanje sintaksne strukture programa, najčešće u obliku sintaksnog stabla programa, zatim proveru statičke semantike (proveru raznih pravila tipa: da li je promenljiva deklarisana pre upotrebe, tipova u izrazima i tako dalje), generisanje interne reprezentacije programa i takođe detekciju i oporavak od grešaka koje je programer uneo u program.



Sintaksno-semantički analizator komunicira sa ostalim celinama kompajlera: od skenera dobija potrebna ulazne tokene, u tabelu simbola upisuje i iz nje čita informacije o deklarisanim korisničkim identifikatorima tipovima promenljivih, klasa, metoda i tako dalje i konačno rezultate svoje analize isporučuje generatoru i optimizatoru koda.

Ovaj dijagram daje dve hijerarhije. S jedne strane vidimo da plavim strelicama je označen tok podataka, recimo sa tabelom simbola postoji dvosmeran protokol pošto se u nju upisuju i iz nje čitaju podaci, a s druge strane, crne strelice označavaju tok kontrole. Prema tom toku kontrole, vidimo da sintaksno-semantički analizator predstavlja u suštini master komponentu kompletne kompajlerske arhitekture, u smislu da sintaksno-semantička analiza pokreće sve druge celine. Možemo reći da je u glavni program u okviru sintaksno-semantičke analize, a da su svi ovi drugi delovi kompajlera u suštini servisi ili potprogrami koji bivaju aktivirani da urade potrebne obrade. Takav koncept kontrole se zove sintaksno-upravljano prevođenje. Znači razne sintaksne

konstrukcije u programu u zavisnosti od toga što je programer upotrebio, utiču na to šta će biti aktivirano u okviru procesa prevođenja.

Bezkontekstne gramatike

Radi se o jednom matematički formalizmu. Bezkontekstna gramatika G definisana je uređenom četvorkom (V_N, V_T, P, <S>) gde:

- V_N predstavlja skup neterminalnih simbola, u primeru koji sledi V_N = {<Expr>, <Sign>, <Addop>, <Term>, <Mulop>, <Factor>}.
- V_T predstavlja skup terminalnih simbola, u primeru koji sledi V_T = {+, -, *, /, ident, number, (,)}. Terminalni i neterminalni simboli se jednim imenom nazivaju gramatički simboli. $V = V_T \cup V_N$ predstavlja skup gramatičkih simbola.
- P predstavlja skup smena (produkcija) oblika <X> $\rightarrow \alpha$, gde <X> na levoj strani smene predstavlja proizvoljan neterminalni simbol, a α na desnoj strani smene predstavlja proizvoljan niz od nula ili više gramatičkih simbola. U primeru koji sledi numerisane su i ima ih 14.
- Jedan neterminal, u primeru koji sledi <Expr> $\in V_N$ predstavlja startni neterminal.

Gramatike se mogu pisati u različitim notacijama. Mi ćemo najviše koristiti ovu BNF notaciju i ona ima svoje podvarijante, znači neke stvari mogu donekle različito da se obeležavaju. Recimo, jedna varijanta, koju u kursu najviše koristimo, je da se neterminali uokviruju ovim znakovima <>, terminali se pišu bez zagrada, bilo da predstavljaju neki identifikator ili neki specijalan znak, zatim leva i desna strana smene razdvajaju se ili sa ::= ili strelicom na desno, što češće ovde. mi koristimo kao obeležavanje.

1. <Expr> $\rightarrow$ <Sign> <Term>
2. <Expr> $\rightarrow$ <Expr> <Addop> <Term>
3. <Sign> $\rightarrow$ +
4. <Sign> $\rightarrow$ -
5. <Sign> $\rightarrow \varepsilon$
6. <Addop> $\rightarrow$ +
7. <Addop> $\rightarrow$ -
8. <Term> $\rightarrow$ <Factor>
9. <Term> $\rightarrow$ <Term> <Mulop> <Factor>
10. <Mulop> $\rightarrow$ *
11. <Mulop> $\rightarrow$ /
12. <Factor> $\rightarrow$ ident
13. <Factor> $\rightarrow$ number
14. <Factor> $\rightarrow$ (<Expr>)

Ovde imamo primer jedne gramatike koje opisuje aritmetičke izraze koji imaju različite operatore sabiranja, oduzimanja, množenja, deljenja, a može da se upotrebi ime promenljive, može da se upotrebi celobrojna konstanta (to su različiti terminali) i tako dalje.

Takođe izraz može da se stavi u zagradu. Gramatika ima jedan specifičan oblik, znači to nije jedini način, niti je najjednostavniji način da se izrazi sintaksa nekog aritmetičkog izraza, ali prosto je jedan primer gramatike u BNF notaciji.

U BNF notaciji ne koristimo neke dodatne simbole koje koristimo u proširenoj (engl. Extended) BNF notaciji. S te strane, čitljivije su smene, ali ih ima više nego u EBNF notaciji. EBNF notacija, ili proširena BNF notacija, gde BNF može biti skraćenica od Bekus Naurove forme, neko kaže i Bekusova normalna forma, je notacija koja isto mora da predstavi terminalne i neterminalne simbole. Stringovi su terminalni simboli koji predstavljaju nekakav ulazni niz simbola. S druge strane, mogu se koristiti identifikatori koji, ako se pišu malim slovima, onda su obično po konvenciji terminali, ako se pišu velikim slovima, onda znamo da su neterminali. Smenama mogu da se razdvajaju leva i desna strana znakom jednakosti. Imamo tačku koja završava svako smenu, kao prosto jedan sintaksni element.

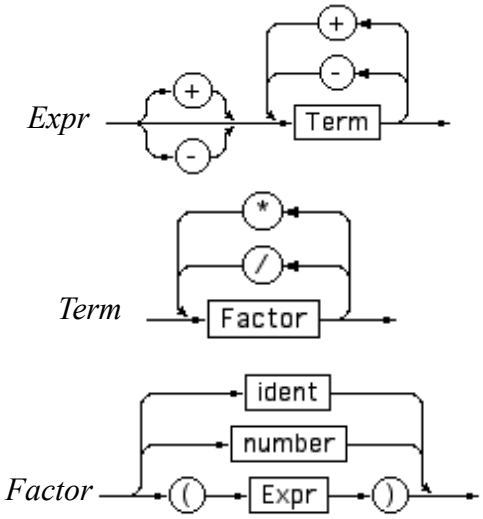
simbol	*značenje*	*primeri*
string	Predstavlja “sam sebe”	"=", "while"
ime	predstavlja T ili NT simbol	ident, Statement
=	razdvaja strane smene	A = b c d .
.	završava smenu	
|	Razdvaja alternative	a | b | c značenje a or b or c
(...)	grupiše alternative	a (b | c) značenje ab | ac
[...]	opcioni deo	[a] b značenje ab | b
{...}	ponavljajući deo	{ a } b značenje b | ab | aab | aaab | ...

Osim tih sitnih da kažemo kozmetičkih detalja u pisanju terminala, neterminala i razdvajanja leve i desne strane, najvažnija razlika između BNF i proširene BNF notacije je što je u proširenoj BNF notaciji dozvoljeno na desnoj strani pisati, ne samo proste nizove gramatičkih simbola (znači da ih ređamo jedan iza drugog) kao u BNF, nego je na desnoj strani dozvoljeno koristiti regularne operatore, tako da desna strana smene može da bude regularni izraz. Znači može da se koristi operator alternative, zatim može da se koriste zagrade da bi se grupisali podizrazi, može da se koristi opcija (znači ako se stavi A u uglaste zagrade to predstavlja ili sekvencu samo B, ako je A izostavljeno, ili sekvencu AB ako se A navede) i konačno može da se reprezentuje ponavljanje nula ili više puta, to je kao onaj operator zvezdastog zatvaranja, samo malo drugačije napisan. Ovakav regularni izraz predstavlja sekvence koje su ovde navedene, znači A može biti nula ili više puta navedeno i mora da se sekvenca završi sa B.

Recimo, za istu onu gramatiku za aritmetičke izraze koju smo videli u BNF notaciji, u EBNF notaciji dovoljno je napisati sledeća tri reda uz korišćenje ovih regularnih operatora. Kao što vidimo, ima mnogo manje smena, ali smene su malo kompleksnije. mora se uložiti više napora u raščlanjivanje šta tačno definiše desna strana smene.

```
Expr   = [ "+" | "-" ] Term { ( "+" | "-" ) Term }.
Term   = Factor { ( "*" | "/" ) Factor }.
Factor = ident | number | "(" Expr ")".
```

Treći način izražavanja bezkontekstnih gramatika su takozvani sintaksni dijagrami. Oni su vrlo bliski proširenoj BNF notaciji, samo što se određenim grafičkim simbolima reprezentuju regularni izrazi koji se inače pišu sa regularnim operatorima. Sintaksni dijagram za aritmetičke izraze dat je na sledećoj slici.



Ovde vidimo da je leva strana svake smene navedena na levoj strani glavne horizontalne linije i ta linija reprezentuje kako se konstruiše desna strana smene. Vidimo da se alternative označavaju veritkalnim deljenjem (razgranavanjem) linije, naravno pošto u prvom pravilu ima i opcija znači imamo i pravu liniju da mogu da se zaobidju i plus i minus. Ponavljanje može da se realizuje cikličnim granama. Vidimo da se neterminalni simboli, kad se pojavljuju na desnoj strain, pišu u okviru pravougaonika, a terminalni simboli se pišu u okviru kruga ili elipse. Znači ovo je možda malo čitljiviji način, ali izomorfno može da se preslika na EBNF notaciju.

Ove dve notacije, EBNF i sintaksni dijagrami, pominjemo sada informativno. U ostatku kursa ćemo praktično isključivo koristiti BNF notaciju, što ne znači da u praksi ne može da se koristi EBNF. Često u jezičkim priručnicima data je EBNF sintaksa, čak i neki alati za konstrukciju kompajlera mogu da koriste tu proširenu BNF notaciju. Stvar je više preference koja će se koristiti ali kompajlerske knjige većinom koriste striktnu BNF notaciju, pa i mi u kursu koristimo uglavnom samo BNF.

Vidimo neke sintaksne detalje, da se terminalni simboli, ako su u pitanju pojedinačni specijalni znaci, stavljaju pod znake navoda, a ako želimo da reprezentujemo neku klasu, recimo ime promenjive, gde tih imena može biti različit broj, pa je potreban i onaj vrednosni deo tokena da bi se dodatno definisalo o kom je tačno simbolu reč, onda može da se uvede neko ime koje počinje malim slovom. Neterminalni simboli po konvenciji mogu da se pišu velikim slovima.

U nastavku ćemo dati neku osnovnu terminologiju vezanu za bezkontekstne gramatike koju kasnije u izlaganju raznih parserskih algoritama koristimo. Skup terminalnih simbola drugim imenom se naziva *alfabet* posmatrane gramatike.

Sentencijalna forma je proizvoljan niz gramatičkih simbola. Mada je ovde napisano proizvoljan, u suštini te sentencijalne forme se pojavljuju u pojmu koji se zove izvođenje nekog niza iz startnog neterminala koji ćemo naknadno definisati.

Sentenca je sekvenca sastavljena isključivo od terminalnih simbola opisanih gramatikom. Za razliku od sentencijalne forme u kojoj se mogu pojavljivati i neterminali i terminali, u sentenci se pojavljuju isključivo terminali. Niz terminalnih simbola treba da bude prihvatljiv za gramatiku, znači legalan, da bi bio nazvan sentencom.

Gramatičko izvođenje je još jedan važan pojam za gramatike i ono predstavlja primenu gramatičkih smena u nekoj sentencijalonoj formi. Formalno, gramatičko izvođenje, u oznaci

$$\beta <X> \gamma \Rightarrow \beta \alpha \gamma$$

predstavlja primenu neke gramatičke smene $<X> \rightarrow \alpha$ da bi se u nekom nizu gramatičkih simbola (takozvanoj sentencijalnoj formi) $\beta <X> \gamma$ zamenilo pojavljivanje leve strane smene odgovarajućom desnom stranom.

Drugim rečima, postupak dobijanja ove poslednje sentencijalne forme iz početne zove se gramatičko izvođenje. Gramatičko izvođenje ima svoj oznaku $\Rightarrow$. Izvođenje se može primeniti višekratno, tako što se različiti neterminali menjaju u sentencijalnoj formi i time se onda dobija niz sukcesivnih sentencijalnih formi. Da bi skraćeno napisali ako ima jedan ili više koraka izvođenja, onda pišemo $\overset{+}{\Rightarrow}$. Ako ima nula ili više koraka izvođenja (originalna sentencijalna forma ili neka izvedena iz nje) onda pišemo $\overset{*}{\Rightarrow}$.

Redukcija je pojam obrnut od izvođenja. Ako imamo dve sentencijalne forme i ako od jedne dobijamo drugu tako što identifikujemo desnu stranu neke smene u početnoj sentencijalnoj formi i onda tu desnu stranu uklonimo, a umesto nje stavimo levu stranu neke smene i tako dobijemo novu sentencijalnu formu, to se zove redukcija.

Formalno definisano, šta je *jezik* gramatike G? Jezik opisan gramatikom G, u oznaci L(G) predstavlja skup svih sentenci (nizova terminalnih simbola) koje se mogu izvesti iz startnog neterminala <S> u nula ili više koraka izvođenja:

$$L(G) = \{x \mid <S> \overset{*}{\Rightarrow} x, x \in V_T^*\}$$

Drugim rečima radi se o skupu svih takvih x gde se x može dobiti izvođenjem u nula ili više koraka iz startnog terminala i pri tome x je element skupa VT* znači ponavljanje terminala nula ili više puta, to su u suštini nizovi terminalnih simbola dužine nula ili više. Prema tome, ako je neki niz terminala moguće izvesti iz startnog neterminala, onda je on legalan prema toj gramatici. Skup svih takvih ispravnih nizova zove se jezik gramatike G.

Ilustrativni primer

Sledeći primer ilustruje pojam izvođenja i legalnih sekvenci. Data je jedna gramatika koja imao osam smena, vidimo da neterminalni simboli su <S>, <B>,<A>. Imamo sledeće smene: prva i druga smena su za <S> (imaju <S> na levoj strani). Vidimo da <S>, prema prvoj smeni, može da se zameni sekvencom simbola a (znači to je terminal), <A> (to je neterminal), c, <B>. Terminalni simboli gramatike su a, b, c i d. Znači ova mala slova su terminalni simboli, a velikim slovima su označeni neterminali simboli.

1. $<S> \rightarrow$ a <A> c <B>
2. $<S> \rightarrow$ <B> d <S>
3. $<B> \rightarrow$ a <S> c <A>
4. $<B> \rightarrow$ c <A> <B>
5. $<A> \rightarrow$ <B> a <B>
6. $<A> \rightarrow$ a <B> c
7. $<A> \rightarrow$ a
8. $<B> \rightarrow$ b

Sada želimo da dokažemo da je sekvenca aacb ispravna sa stanovišta ove gramatike, odnosno da je u skupu svih takvih ispravnih sekvenci, to jest u jeziku gramatike G. Sekvenca aacb pripada jeziku date gramatike ako i samo ako je moguće izvođenje sekvence aacb iz startnog neterminala <S>. U prvom koraku izvođenja, u obzir za zamenu <S> dolaze:

- smena 1, čija desna strana počinje terminalom a, kao i
- smena 2. čija desna strana počinje neterminalom <B>, koji se može kasnije zameniti primenom smene 3. čija desna strana takođe počinje terminalom a.

Uočimo međutim da desna strana smene 2. sadrži terminal d koji se ne pojavljuje u datoj gramatici, pa prema tome otpada iz daljeg razmatranja. U izvođenju, prema tome, primenjujemo smenu 1:

<S> ⇒ a <A> c <B>
↑
1

Sada se vidi da neterminal <A> treba zameniti primenom smene 7, a neterminal <B> primenom smene 8 (ostale smene eliminišu se iz razmatranja jer sadrže i druge terminale osim potrebnih):

<S> ⇒ a <A> c <B> ⇒ a a c <B> ⇒ a a c b
↑ ↑ ↑
1 7 8

Time smo izbacili sve neterminale iz niza, dobijamo sentencu, sastavljenu isključivo od terminalnih simbola aacb i time smo dokazali da je ovaj niz ispravan to jest, možemo ga izvesti iz startnog neterminala, odnosno nalazi se u jeziku ove gramatike.

Pojam *levog izvođenja* ili engleski leftmost derivation: pošto je u svakom koraku izvođenja zamenjivan prvi neterminal s leve strane u svakoj sentencijalnoj formi, dobijeno izvođenje naziva se levim izvođenjem (engl. leftmost derivation), što se može posebno naznačiti na sledeći način:

$$\langle S\rangle \Rightarrow_{lm} a\ \langle A\rangle\ c\ \langle B\rangle \Rightarrow_{lm} a\ a\ c\ \langle B\rangle \Rightarrow_{lm} a\ a\ c\ b$$

Ukoliko se u svakom koraku izvođenja zamenjuje krajnje desni neterminal u sentencijalnoj formi, dobija se *desno izvođenje* (engl. rightmost derivation):

$$\langle S\rangle \Rightarrow_{rm} a\ \langle A\rangle\ c\ \langle B\rangle \Rightarrow_{rm} a\ \langle A\rangle\ c\ b \Rightarrow_{rm} a\ a\ c\ b$$

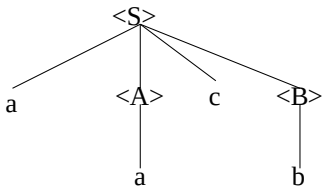
Zašto su nam bitni ti pojmovi videćemo malo kasnije, kada budemo razmatrali različite metode parsiranja, pošto neki metodi prate levo izvođenje, a neki drugi metodi prate desno izvođenje.

Šta je *parser*? To je dosta važan pojam kompajler, Centralni deo kompajlera je nekakav parser. Parser je procedura, koja na osnovu date gramatike, utvrđuje, za zadatu ulaznu sekvencu, pripadnost sekvence jeziku opisanom datom gramatikom i njeno izvođenje iz startnog neterminala. Parseru se na ulazu zadaje nekakav niz terminala, a parser mora da rekonstruiše ovo izvođenje, ako ono postoji. Time on dokazuje da je ulazni niz ispravan. Što se kompajlera tiče ovaj ulazni niz terminala je u stvari tokenizovani program koji programer piše, a na bazi zadatog programa mora da se rekonstruiše stablo izvođenja programa.

Stablo izvođenja je grafička predstava izvođenja. Sledeća slika predstavlja stablo izvođenja koje odgovara gramatičkom izvođenju iz ranijeg primera:

<S> ⇒ a <A> c <B> ⇒ a a c <B> ⇒ a a c b
↑ ↑ ↑
1 7 8

U stablu izvođenja startnom neterminalu odgovara koreni čvor. Primena smene <S> → a <A> c <B>, zamena neterminala <X> desnom stranom, naznačena je tako što je neterminal <S> roditeljski čvor, a simboli desne strane a, <A>, c i <B> su njegovi neposredni sledbenici u stablu, napisani onim redom kako su napisani na desnoj strani smene. Takođe videli smo da se u nastavku izvođenja <A> menjalo sa a, to je isto ovde naznačeno i <B> se menjalo sa b. Kompletirano stablo izvođenja u listovima ima isključivo terminalne simbole ili oznaku za praznu sekvencu Epsilon, a u unutrašnjim čvorovima ima neterminale.



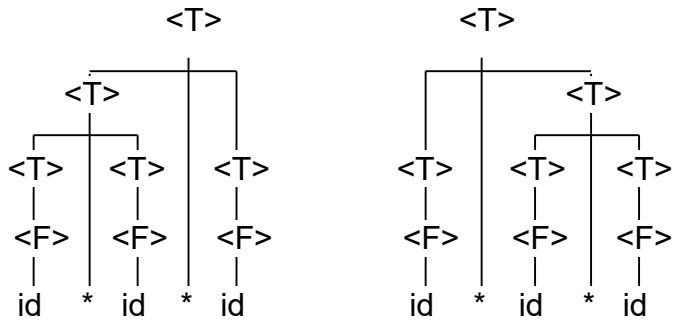
Višeznačnost gramatika

Gramatike imaju različite svoje osobine koje su i dobre i loše. Mi ćemo u ovom trenutku pomenuti samo po neku osobinu. Jedno od dosta važnih svojstava gramatike je njena višeznačnost. Ako je gramatika *višeznačna*, za jednu ulaznu sentencu moguće je odrediti više različitih stabala izvođenja. Imamo recimo primer gramatike koja ima u suštini tri smene, mada ima dve linije.

<T> → <F> | <T> * <T>
<F> → id

Prva smena je <T> se menja sa <F>, znači <T> i <F> su neterminali, a zvezdica i id su terminalni simboli. Ima ukupno tri smene: <T> se menja sa <F>, <T> se menja sa <T> * <T>, ova vertikalna linija nije terminalni simbol, nego ona u stvari predstavlja skraćenicu da je to sledeća smena se istom ovom levom stranom <T>. Znači, umesto da smo napisali poseban red, <T> se menja sa <T> * <T>, mi smo samo stavili ovaj skraćeni oblik. I konačno treća smena <F> se menja sa id.

Sada, ako razmotrimo sekvencu id * id * id, možemo tu sekvencu izvesti na dva načina iz startnog neterminala <T>. Jedan način je da prvo upotrebimo ovu drugu smenu <T> → <T> * <T>. Tu se sada razlikuju podvarijante: da li ćemo ovim prvim <T> upariti id * id a ovim drugim <T> samo ovo treće id, ili ćemo prvim <T> upariti samo prvo id, a drugim <T> upariti ovaj ostatak id * id. Znači u tome se razlikuju ova dva stabla izvođenja.



Za istu gramatiku i za istu ulaznu sekvencu dobijamo dva po strukturi različita sintaksna stabla. Višeznačnost gramatike predstavlja nepoželjno svojstvo pri prevođenju Zašto je nepoželjna, videćemo na još jednom primeru koji će biti direktnije vezan za kompajler i programske jezike.

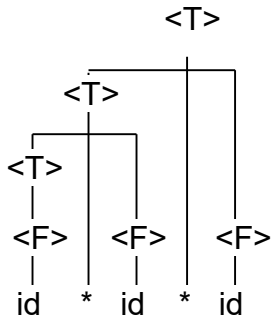
Ali evo prvo da vidimo kako ovo nepoželjno gramatičko svojstvo na neki način ukloniti ili preduzeti neke mene da izbegnemo loše posledice. Jedan od načina uklanjanja višeznačnosti gramatike je da transformišemo gramatiku u ekvivalentnu gramatiku koja opisuje tačno isti jezik (isti skup sekvenci), ali koja nema tu višeznačnost.

Recimo, ovaj konkretan primer koji nije mnogo komplikovan, reč je o relativno maloj gramatici, može lako da se koriguje. Smena <T> → <T> * <T> je problematična sa stanovišta višeznačnosti, jer imamo ova dva <T> i videli smo da oni u suštini mogu da uparuju na različite načine one izraze koji imaju više id-ova.

Da bismo otklonili višeznačnost, na neki način moramo transformisati ovu smenu, da otklonimo tu veliku fleksibilnost. Jedan od načina je da ograničimo kontekst ovog drugog podizraza, znači da više ne bude <T> -- najgeneralniji izraz koji je opisan celom gramatikom -- nego da desni podizraz ograničimo samo na <F>, odnosno samo na id. Time u stvari forsiramo, u onoj situaciji kad imamo id * id * id, u ovoj smeni, da levi <T> mora da uparuje ovaj veći podizraz, a desni <F> uparuje samo ovaj zadnji id, pošto ne može više da upari kompleksniji izraz.

<T> → <F> | <T> * <F>
<F> → id

Od ona dva stabla koja su bila prisutna za istu ulaznu sekvencu, ostaje samo ovo jedno stablo gde smo kompleksniji podizraz uparivali ovim levim <T>.



Postoji još načina, recimo u gramatikama za izraze, kako može da se eliminiše dvosmislenost. Recimo kod aritmetičkih izraza, asocijativnost operatora znači sledeće: ako imate a + b + c a plus je levo asocijativni operator, to znači da je propisano da prvo treba sabrati a + b pa onda dodati c. Prioriteti operatora znači: ako imate a + b * c, a znamo da je prioritet množenja veći od prioriteta sabiranja, onda treba prvo pomnožiti b * c, pa onda dodavati a. Znači, asocijativnost i prioriteti operatera mogu da pomognu eliminaciji dvosmislenosti kod opisivanja izraza gramatikama. Za detaljniji primer kako svojstva asocijativnosti i prioriteta operatora iskoristiti u uklanjaju višeznačnosti, čitalac se upućuje na Zbirku zadataka.

Matematički, sabiranje je komutativna operacija i, matematički gledano, a + b isto što i b + a. Međutim, u računarskim naukama, operacija sabiranja se izvodi u konačnoj tačnosti, znači različito je definisana nego u matematici i ona sama po sebi nije komutativna, U računarskom sabiranju, a + b i b + a mogu dati različite rezultate, pogotovo ako ima tri sabirka a + b + c, u zavisnosti od toga da li se prvo sabiraju a i b, pa dodaje c, ili se prvo sabiraju b plus c, pa dodaje a, mogu se dobiti različiti rezultati zbog konačne tačnosti. Recimo, ako je neki međurezultat jako velik, a dodaje se jako mali faktor, mogu da zbog predstavljanja vrednosti u pokretnom zarezu otpadnu decimale.

Problem visećeg else

Još jedno očiglednije negativno svojstvo dvosmislenosti je problem takozvanog visećeg else, koji se javlja u jezicima koji imaju definisano sintaksu if naredbe, tako da ona nema jasan terminator. Recimo, mamo if, ali nemamo endif. U nekom jeziku, sintaksa if bi mogla biti opisana sledećim fragmentom gramatike.

<Stmt> → if <Expr> then <Stmt>
| if <Expr> then <Stmt> else <Stmt>
| other Stmts

Statement, odnosno iskaz, je if izraz then iskaz. Znači to je if sa then delom. Pa zatim, sledeća smena: if izraz then iskaz else iskaz znači uveli smo i else. I treća smena odnosi se na druge vrste iskaza, znači tu sad mogu da se opišu iskazi dodale, iskazi kontrole toka while, for i tako ko dalje. Prve dve smene opisuju if samo sa then delom i if sa else delom.

Ako imamo ulazni program **if e1 then if e2 then s1 else s2** i ako primenimo ovu opisanu gramatiku možemo tumačiti ovaj fragment programa na dva načina, odnosno možemo napraviti dva različita stabla izvođenja za ovu situaciju (koja ovde nisu prikazana).

Za ovo else, postavlja se pitanje kojem if-u se pridružuje. U jednoj varijanti, else može da se pridruži ovom prvom spoljnom if e1, tako da znači u then delu toga if-a imamo if e2 then s2 bez else grane, znači da unutrašnji if bude bez else, a da ovo else pripada spojljnom if-u, to je jedno tumačenje. Drugo tumačenje da else grana pripada unutrašnjem if-u, tako da spoljni if E1 ima samo then granu, a u toj then grani je ceo ovaj unutraašnji if, ko ima i end i else granu. Očigledno da značenje programa može biti sasvim različito, znači kontrola toka, u zavisnosti od toga da li se usvoji prvo ili drugo tumačenje.

To jeste problem se jezicima koji imaju ovako definisan if i takav problem se rešava dodatnim pravilom da se u ovakvim situacijama else pridružuje ugneždenom if-u, Ovde je, na neki način, ovim što je napisano else S2, već sugerisano rešenje da to else S2 pripada ovom if E2.

Ovo je u suštini pravilo statističke semantike, možemo tako da ga shvatimo jer gramatika ga ne uslovljava nego tek dodatnom kontrolom možemo to da razrešimo. Neka druga pravila statičke semantike ne mogu se rešiti na nivou sintakse, međutim ovaj problem može da se reši na nivou sintakse, odnosno može da se napiše alternativna gramatika koja bi zabranila ono drugo tumačenje da else pridružuje spoljnom if-u, nego da forsiramo znači da se u ovakvim situacijama else pridružuje unutrašnjem if-u. Evo kako izgleda takva jedna gramatika kod koje je eliminisana ova dvosmislenost.

<Stmt> → <Matched>
| <Unmatched>
<Matched> → if <Expr> then <Matched> else <Matched>
| other Stmts
<Unmatched> → if <Expr> then <Stmt>
| if <Expr> then <Matched> else <Unmatched>

Kao što se može videti, gramatika je dosta složenija, ima više smena i malo je teža za razumevanje. Uvedeni su novi neterminali <Matched> i <Unmatched>. Neterminal <Matched> odnosi se na if koji obavezno ima else. Znači u "matched" vrsti iskaza, to mogu biti i neki drugi iskazi koji nisu if, ako je u pitanju if, on mora da ima else. A "unmatched" vrsta iskaza je onaj if koji ima opciono else, znači može da ima else, ali ne mora. Ovo raščlanjivanje nam omogućava da ako postoji else

u if-u, onda iza then dozvoljavamo samo "matched" iskaze. Time praktično forsiramo to da bilo koji drugi else uvek uđe prvo u taj "matched" iskaz, znači ne može nikako da ga otme spoljni if, jer unutrašnji if uvek mora da ima sebi pridruženo else. Tako da ovaj <Unmatched>, kao što vidimo ovde, dozvoljavamo recimo u if-u koju uopšte nema else i takođe dozvoljavamo u else delu if-a, ali znači ako if ima else, onda u then delu mora biti isključivo <Matched>. Ovo nije formalan dokaz, ali daje objašnenje zašto gramatika više nije dvosmislena.

Druge vrste dvosmislenih jezičkih konstrukcija

Postoje druge vrste dvosmislenosti koje takođe usložnjavaju konstrukciju prevodilaca, to jest zahtevaju neke posebne mehanizme da se razreši situacija. Neki jezici imaju određene konstrukcije koji su po svojoj prirodi dvosmislene. U programskom jeziju ANSI C se ne može jednostavno iz pozicije imena razrešiti o kojoj vrsti simbola se radi, jer to zavisi od konteksta upotrebe, tj. od drugih deklaracija. Posmatrajmo dva primera gde imamo konstrukciju ime(x) u glavnom programu.

Prvi primer:

```
typedef int ime;
main()
{
   ime(x); // deklaracija promenljive x tipa ime
}
```

Drugi primer:

```
void ime(int y){}
int x;
main()
{
   ime(x); // poziv funkcije ime sa parametrom x
}
```

U ANSI Cu je dozvoljeno da složeni iskaz <compound_statement>, koji čini telo main funkcije, počne opcionom listom deklaracija iza koje sledi opciona lista iskaza:

<compound_statement> → { }
→ { <statement_list> }
→ { <declaration_list> }
→ { <declaration_list> <statement_list> }

<declaration_list> → <declaration>
→ <declaration_list> <declaration>

<statement_list> → <statement>
→ <statement_list> <statement>

tako da se isto ime u prvom primeru interpretira kao deklaracija, a u drugom nema deklaracija, nego je u pitanju iskaz, poziv funkcije. Da bismo stvarno znali šta je od te dve stvari, mi moramo znati tip idenfikatora ime. Zašto je to problem? Ako u gramatici za C za ime samo uvedemo IDENT kao terminal, za korisnički tip ili funkciju, onda će postojati sledeće dve smene[1], gde smena za

[1] Razmatranje je uprošćeno. U ppravoj ANSI C gramatici postoji veći broj smena, ali sa istim krajnim efektom.

<declaration> opisuje deklarisanje promenljivih korisničkog tipa, druga smena za <statement> opisuje poziv funkcije:

<declaration> → IDENT (<declarator_list>)

<declarator_list> → IDENT
→ <declarator_list> , IDENT

<statement> → IDENT (<argument_expression_list>)

<argument_expression_list> → IDENT
→ <argument_expression_list> , IDENT

Te dve smene će napraviti dvosmislenost, znači za navedeni izraz ime(x) mogla bi se napraviti oba stabla izvođenja, gde je prepoznata jedna smena, ili gde je prepoznata druga smena. Kako razrešiti takvu situaciju? Jedan od načina rešavanja je da se uvede posebna vrsta terminalnog simbola za jednu od tih stvari, recimo za korisničko ime tipa uvedeno deklaracijom typedef:

U toj varijanti, morali bismo realizovati leksički analizator koji vraća terminalne simbole, odnosno tokene, sintaksnom analizatoru na taj način da leksički analizator pokupi identifikator, prema onim leksičkim pravilima, na primer, identifikator počinje slovom, a iza toga slede slova, cifre i eventualno underscore (_). Kada identifikuje korišćenje identifikatora leksički analizator onda mora da pogleda u tabelu simbola:

```
if ( Tab.lookup(yytext).Kind == "Type" )
  return TYPEDEF_name;
else
  return IDENT;
```

Ako je u tabeli simbola zapisano da se određeni identifikator ime odnosi na tip (koji je u ranijem delu programa bio deklarisan sa typedef), onda leksički analizator vraća posebnu klasu tokena, na primer, TYPEDEF_name. U suprotnom se vraća običan token IDENT koji se odnosi na sve druge upotrebe tog imena. Ako potom u gramatici u smeni za <declaration> zamenimo terminal IDENT terminalom TYPEDEF_name, a smenu za <statement> ostavimo neizmenjenu, onda gramatika više neće biti dvosmislena, jer je leksički analizator uvidom u tabelu simbola razrešio o čemu se radi.

<declaration> → TYPEDEF_name (<declarator_list>)

<statement> → IDENT (<argument_expression_list>)

Šta je negativna strana ovoga rešenja? Ovime dodatno sprežemo leksičku i sintaksnu analizu jer sada leksički analizator mora da proverava tipove imena. Tip imena je nešto što postavlja sintaksni analizator, kad prepozna celu typedef deklaraciju, on ažurira tabelu simbola. Obično je znači sintaksni analizator samo korisnik servisa leksičke analize, ovako leksička analizator ima tu povratnu zavisnost prema sintaksnoj analizi indirektno preko tabele simbola. Ali znači i to je moguće, znači prosto svaki taj problem može u praksi da ima neko rešenje koje ponekad komplikuje realizaciju.

Opšta klasifikacija gramatika

Na kraju ove priče, informativno ćemo predstaviti klasifikaciju svih gramatika u matematičkom smislu. Autor ove klasifikacije je poznati matematičar Čomski, koji je inače i politički aktivan, pa

štampa i dan danas citira neke neke njegove izjave, ali njegov glavni stručni doprinos je u teoriji formalnih jezika, vezan za ove bezkontekstne gramatike. Cela ta teorija razvijana je dosta davno, 50-60-tih godina 20. veka. Sledi klasifikacija gramatika, pri čemu su gramatike skupovi smena opšteg oblika $\alpha \rightarrow \beta$, pri čemu su α i β nizovi gramatičkih simbola.

klasa 0 **Neograničene gramatike** (α i β proizvoljni)
npr: <A> → a <A> b | <B> c <B>
a<B>c → d
d<B> → bb
<A> ⇒ a<A>b ⇒ a<B>c<B>b ⇒ d<B>b ⇒ bbb
Prepoznaju se Tjuringovim mašinama

klasa 1 **Kontekstne gramatike** ($|\alpha| \leq |\beta|$)
npr: a <A> → a b c
Prepoznaju se linearno ograničenim automatima

klasa 2 **Bezkontekstne gramatike** (α = NT)
npr: <A> → a b c.
Prepoznaju se potisnim automatima

klasa 3 **Regularne gramatike** (α = NT, $\beta = \varepsilon \mid$ T NT)
npr: <A> → ε | b <B>.
Prepoznaju se konačnim automatima

} Samo ove dve klase su relevantne za konstrukciju prevodilaca

Najgeneralija klasa 0 gramatika su takozvane *neograničene gramatike*, gde ne uslovljavamo α i β nikakvim posebnim pravilima. Jedna gramatika klase 0 je napisana sa nekoliko smena. Ovde leva strana smene znači ne mora biti jedan neterminal, nego može biti čitav niz (sentencijalna forma). Ovde vidimo neke četiri smene i isto važi pojam izvođenja iz startnog neterminale evo izvodimo sekvencu bbb. Takve gramatike prepoznaju se Tjuringovim mašinama, to je jedan poznat koncept u računarskim naukama i bitan je po tome da u suštini bilo kakav računarski algoritam može da se izrazi programom za Tjuringovu mašinu. Vidimo da su u suštini i neograničene gramatike isto tako generalan mehanizam koji zahteva Tjuringovu mašinu kao prepoznavač.

Klasa 1 je malo restriktivnija i to su takozvane *kontekstne gramatike*. Tu su uvodi (u odnosu na one neograničene gramatike) dodatno ograničenja da je dužina leve strane smene manja do jednaka dužini desne strane u broju simbola i takve gramatike se prepoznaju takozvanim linearno ograničenim automatima, u šta nećemo ulaziti, pošto za našu primenu to nije bitno.

Klasa 2 su bezkontekstne gramatike i tu dolazimo do klase gramatika koje su od interesuja za programske jezike za izražavanje njihove sintakse i za kompajlersku obradu. Kod bezkontekstnih gramatika, ograničenje je da leva strana smene mora biti tačno jedan neterminal. Takve gramatike prepoznaju se takozvanim *potisnim automatima*. Potisni automat detaljno ćemo proučavati u narednom poglavlju. Za sada mogu samo da kažem to je to jedan koncept sličan konačnom automatu, ali proširen u smislu da, pored stanja i kontrolne tabele ima i strukturu podataka tipa steka koju koristi u svom radu i koja mu onda omogućuje da bude u suštini jači, primenljiviji na mnogo više jezika nego konačni automat.

Konačno, klasa 3 su takozvane *regularne gramatike,* to je najrestriktivnija klasa gramatika. Restrikcija se sastoji u tome da smene moraju biti u tačno jednom od dva oblika: prvo leva strana smene mora biti tačno jedan neterminal, a desna strana smene ili mora biti prazna, to je ova epsilon

sekvenca (znači nema ništa na desnoj strani) ili desna strana smena mora biti tačno jedan terminal iza koga sledi tačno jedan neterminal. Ako imamo ovoliko ograničene gramatike, onda u te regularne gramatike tačno odgovaraju regularnim jezicima odnosno konačnim automatima, pojmu koji je nama poznat i koji smo obradili. U suštini kada se uvede ta najveća restrikcija, praktično ekvivalentiramo takve vrste gramatika sa konačnim automatima. Ali, za sintaksu jezika koristimo gramatike klase 2. Klase 2 i 3 za leksičku i sintaksnu analizu su nama od interesa, dok ove klase 0 i 1 su od interesa samo u matematičkoj teoriji.

Parsiranje od dna ka vrhu

U lekciji o parsiranju od dna ka vrhu, podsetićemo se pojma parsera i proučiti jedan konkretan metod parsiranja koji se zove shift-reduce, odnosno parsiranje na principu potiskivanja i svođenja. Videćemo kako se konstruišu određene kategorije shift-reduce parsera i, takođe, videćemo moguću pojavu konflikata u shift-reduce parsiranju i kako se izboriti sa tom negativnom pojavom. Da se podsetimo od ranije, parser je algoritam koji utvrđuje, za zadatu ulaznu sekvencu, da li pripada jeziku zadate gramatike. Ako sekvenca pripada jeziku, parser određuje izvođenje posmatrane ulazne sekvence iz startnog neterminala gramatike.

Opšte strategije parsiranja razlikuju se u tome u kom smeru parser razotkriva ili rekonstruiše stablo izvođenja ulazne sekvence. Parseri od vrha ka dnu, (engl. top-down) parseri rekonstruišu stablo izvođenja počev od korena ka listovima, a parseri od dna ka vrhu (engl. bottom-up) rade suprotno, znači rekonstruišu stablo izvođenja ulazne sekvence od listova ka korenu.

Parseri su zasnovani na konceptu potisnog automata i neformalno ćemo ga definisati kao neku vrstu proširenja konačnog automata (automata sa konačnim brojem stanja), gde je dodata struktura steka, to jest, first in, first out struktura. To čini potisni automat u suštini moćnijim formalizmom nego konačni automat. Potisni automati su jednako jak formalizam za opis jezika kao bezkontekstne gramatike, to jest, za svaki jezik za koji može da se opiše beskontekstnom gramatikom, može da se konstruiše i potisni automat. Važi i obrnuto, za svaki jezik za koji postoji potisni automat, postojaće i bezkontekstna gramatika koja ga opisuje.

Parsiranje na principu potiskivanja i svođenja

Sada ćemo definisati jednu vrstu parsiranja od dna ka vrhu, parsiranje po principu potiskivanja i svođenja, to je prevod engleskog termina shift-reduce parsing. Često se ova vrsta parsiranja označava terminom LR parsiranje, gde L označava da se ulazna sekvenca procesira sa leva na desno znak po znak, a R da se otkriva desno (engl. rightmost) izvođenje ulazne sekvence iz startnog neterminala, što je pojam definisan u prethodnom poglavlju.

Reč je o jednom potisnom automatu, koji je definisan skupom ulaznih simbola, skupom simbola steka, početnim stekom i potisnom i kontrolnom tabelom. Na slici koja sledi imamo primer jednog shift-reduce parsera za gramatiku od četiri smene koja je ovde data.

0. <S′>→<S>–|
1. <S>→ <A><A><S>
2. <S>→ ε
3. <S>→ a
4. <A>→ a

Često proširujemo formalno ove gramatike dodatnom nultom smenom, koja ima dodatni simbol (novi neterminal) kao startni simbol, recimo <S'>, a na desnoj strani ima originalni startni simbol <S> iza čega sledi jedan novi simbol, a to je marker kraja ulazne sekvence –|. On je kao položeno slovo T. Znači on se smatra jednim dodatnim ulaznim simbolom, koji se pojavljuje u ulaznoj sekvenci tačno jedanput i to na samom kraju ulaza, slično kao u programiranju end-of-file.

Jedan shift-reduce parser definisan je sledećim elementima: skupovima simbola steka i ulaznih simbola, sa dve tabele, takozvanom potisnom tabelu, to je na slici leva tabela i kontrolnom tabelom, to je desna tabela na slici. Svaka od njih ima svoju specifičnu ulogu. Poslednji element definicije shift-reduce parsera je početna konfiguracija steka.

Važan element parsera je skup simbola steka, znači vrste ovih tabela označene su simbolima steka parsera. Iz ovog primera već možemo videti da simboli steka odgovaraju gramatičkim simbolima, kako terminalima tako i neterminalima. Međutim, vidimo i ove dodatne oznake u vidu indeksa i vidimo da u opštem slučaju jednog gramatičkom simbolu može odgovarati veći broj simbola steka. Na primer, ovde imamo da $<A>_{11}$ i $<A>_{12}$ odgovaraju gramatičkom simbolu <A>.

	<S>	<A>	a
∇	$<S>_0$	$<A>_{11}$	a_x
$<S>_0$			
$<A>_{11}$		$<A>_{12}$	a_4
a_x			
$<A>_{12}$	$<S>_1$	$<A>_{11}$	a_x
a_4			
$<S>_1$			

(a) potisna tabela

	a	⊣
∇	SHIFT	REDUCE(2)
$<S>_0$	REJECT	ACCEPT
$<A>_{11}$	SHIFT	REJECT
a_x	REDUCE(4)	REDUCE(3)
$<A>_{12}$	SHIFT	REDUCE(2)
a_4	REDUCE(4)	REDUCE(4)
$<S>_1$	REJECT	REDUCE(1)

(b) kontrolna tabela

Simboli steka odgovaraju pojedinačnim pojavama, odnosno dešavanjima gramatičkih simbola na desnim stranama smena. Tako da recimo $<A>_{11}$ odgovara prvom simbolu <A> na desnoj strani prve smene, a $<A>_{12}$ odgovara drugom dešavanju gramatičkog simbola <A> na desnoj strani prve smene. Simbol steka $<S>_1$ odgovara dešavanju gramatičkog simbola <S> na desnoj strani prve smene. a_4 odgovara dešavanju simbola a na desnoj strani četvrte smene.

Zanimljivo je razmotriti simbol steka a_x. Može se desiti da jedan stek simbol u stvari odgovara, ne jednom određenom dešavanju gramatičkog simbola na desnoj strani neke smene, nego, u opštem slučaju, skupu dešavanja, to jest pojava gramatičkih simbola na desnim stranama smena, koje (pojave) odgovaraju istom gramatičkom simbolu. Značu ovo a_x može da predstavlja ili simbol a na desnoj strani treće smene ili simbol a na desnoj strani 4. smene. Videćemo kasnije da parser u toku rada, kada mu se simbol a_x pojavi na vrhu steka, još ne zna da li je prepoznao treću ili četvrtu smenu. Potrebna su mu dodatne informacije da jednoznačno odredi svoju sledeću akciju.

Postoje još dva simbola steka koja nisu pomenuta, jedan je $<S>_0$. Ovo nula deluje na prvi pogled zbunjujuće, ali je posledica toga što se gramatika formalno često proširuje nultom smenom, koja ima ovaj početni simbol <S> na desnoj strani, a iza toga dolazi marker kraja ulaza. Tako da se ovaj $<S>_0$ može shvatiti kao pojavu tog startnog simbola na desnoj strani nultr smene.

Konačno, postoji jedan stek simbol koji nema svoj pandan u gramatičkim simbolima. To je oznaka ∇, obrnuti trougao ili navla, specijalni znak koji označava dno steka. Značenje ovog simbola je da se, u toku rada parsera, na steku nalazi jedna instanca tog simbola i ona je na samom dnu steka. On se niti stavlja na stek, niti skida sa steka. Kada se on pojavi na vrhu steka, to ima značenje da je stek ispražnjen od svih drugih simbola steka. Svi shift-reduce parseri počinju sa početnim stekom praznim, odnosno jedini simbol na steku je dno steka ∇.

Naredni element ovoga parsera je skup ulaznih simbola. Ulazni simboli vide se u desnoj, takozvanoj kontrolnoj tabeli, označavaju njene kolone. Oni se mapiraju na gramatičke terminalne simbole. Ova gramatika vidimo ima terminal a (naravno prazna sekvenca nije terminal nego oznaka da tu nema na desnoj strani ništa) i konačno dodatni terminalni simbol end marker –| je posledica proširivanja gramatike nultom smenom. Znači terminalni simboli su a i –|.

Leva tabela, potisna tabela u svojim kolonama ima gramatičke simbole, kako terminale tako i neterminale. Potisna tabela određuje koji simbol steka se u određenom trenutku potiskuje na stek. Pošto u opštem slučaju veći broj stek simbola odgovara određenom gramatičkom simbolu, kada postoji potreba da se na stek stavi neki od gramatičkih simbola (koji određuju kolone potisne tabele) ulazi potisne tabele određuju koji konkretni stek simbol treba da se stavi na stek. Recimo, ako treba da uradimo operaciju PUSH(<A>), odnosno potisnemo neterminal <A> na stek, onda konsultujemo potisnu tabelu za kolonu <A>, a vrsta je određena trenutnim vrhom parserskog steka, znači u zavisnosti od toga šta je na vrhu steka biće konsultovala odgovarajuća vrsta potisne tabele i onda se iz ulaza potisne tabele čita simbol steka. Recimo, ako je, na primer, stek bio prazan, a treba da stavimo <A> na vrh steka, onda potisna tabela kaže da je to stek simbol $<A>_{11}$. Neki ulazi potisne tabele su prazni. Oni odgovaraju grešci u toku rada parsera. Ako u toku rada parsera dođe do situacije da se konsultuje prazan potisne tabele, to automatski znači da ulazna sekvenca nije ispravna i da parser završava rad sa tim negativnim zaključkom da ulaz nije ispravan.

Kontrolna tabela (desna tabela) parsera određuje akciju parsera na osnovu tekućeg simbola na vrhu steka i tekućeg ulaznog simbola. U kontrolnoj tabeli može se naći četiri tipa parserskih akcija. Prve dve su finalne akcije, znači završavaju rad parsera. Akcija ACCEPT znači uspešan završetak parsiranja (ulazna sekvenca je ispravna, opisana gramatikom), a REJECT je neuspešan završetak parsiranja, ulazna sekvenca nije ispravna. Preostale dve vrste parserskih akcija su SHIFT i REDUCE. Radi se o složenim parserskim akcijama koje se sastoje od komponenata ažuriranja parserskog steka (PUSH, POP) i ažuriranja preostalog ulaza parsera (ADVANCE; RETAIN).

Jedan od preostala dva tipa parserskih akcija je SHIFT. To je jedna složena akcija, koja opisuje šta se dešava sa stekom i šta se dešava sa preostalim ulazom. SHIFT se svodi na akciju nad stekom potiskivanja (PUSH) tekućeg ulaznog simbola na vrh steka uz konsultaciju potisne tabele, a nad ulazom se radi ADVANCE, odnosno konzumira se tekući ulazni simbol i sledeći simbol posle tekućeg, postaje za sledeći korak parsera tekući ulaz znači (prelazi se na ulazu jedan simbol u desno, pri čemo se ulaz čita sa leva na desno znak po znak).

Konačno postoji i parserska akcija REDUCE koja je takođe složena akcija ažuriranja steka i ulaza. Ona ima parametar p koji predstavlja redni broj smene i ima onoliko različitih REDUCE akcija koliko ima smena u gramatici. Redukcija smene p, odnosno njeno sažimanje ili svođenje, znači da treba sa parserskog steka skinuti onoliko simbola steka koliko iznosi dužina desne strane smene p. Znači ako na desnoj strani smene p ima tri simbola, onda treba uraditi tri puta POP i posle toga na vrh steka treba potisnuti simbol leve strane smene p uz konsultaciju potisne tabele. I to je bila komponenta ažuriranja steka. Komponenta ažuriranja ulaza je ova akcija RETAIN. Akcija RETAIN označava da ulaz ostaje neizmenjen, odnosno ono što je bio tekući ulazni simbol u posmatranog koraku, isti taj simbol biće tekući i u sledećem koraku. Znači, na ulazu se ne konzumira ulazni simbol. To je malo različito od konačnog automata koji je bezuslovno u svakom koraku konzumirao tekući ulazni simbol.

Znači ovo su operacije nad ulaznom sekvencom: ADVANCE za prelazak na sledeći ulazni simbol uz konzumiranje tekućeg. RETAIN je akcija koja označava da se ulaz ne dira.

Kontrolna tabela diktira rad automata. Za prethodni primer parsera, ako na tekućem ulazu dolazi ulazni simbol a, vrh steka je na primer prazan stek, onda kontrolna tabela u tekućem koraku rada parsera sugeriše na primer akciju SHIFT.

Sledeća slika ilustruje rad potisnog automata koji smo uzeli za tekući primer, pri čemu nismo objasnili -- to sledi u nastavku -- kako doći do jednog ovakvog parsera nego smo krenuli od primera gotovog parsera.

	sadržaj steka	preostali ulaz	akcija parsera
1.	∇	aa–|	SHIFT
2.	∇a_x	a–|	REDUCE(4)
3.	$\nabla <A>_{11}$	a–|	SHIFT
4.	$\nabla <A>_{11} a_4$	–|	REDUCE(4)
5.	$\nabla <A>_{11} <A>_{12}$	–|	REDUCE(2)
6.	$\nabla <A>_{11} <A>_{12} <S>_1$	–|	REDUCE(1)
7.	$\nabla <S>_0$	–|	ACCEPT

Trejs rada parsera prikazuje se u nekoliko kolona. U prvoj koloni pišemo redni broj koraka, rad parsera obavlja se u diskretnim koracima. Sledeća kolona je sadržaj parserskog streka. Stek se obeležava tako što se uvek piše sa leva na desno u smeru rasta steka, znači vrh neka je uvek krajnje desno u svakom koraku. Sledeća kolona je preostali ulaz. Prvi od simbola je uvek tekući ulazni simbol u posmatranom koraku. U poslednjoj koloni se evidentira akcija koju će parser u datom koraku preduzeti.

Inicijalno, u prvom koraku rekli smo da shift-reduce parser uvek kreće od praznog steka. To znači da se ceo sadržaj svodi na dno steka ∇. Preostali ulaz je kompletan ulaz. U ovom primeru smo uzeli aa i end marker –| kao ulaznu sekvencu, znači tekući ulazni simbol je prvo a.

Sada konsultujemo kontrolnu tabelu, to je desna tabela parsera sa slike na strani 79. Vrh steka određuje vrstu kontrolne tabele, a tekući ulazni simbol određuje kolonu. Znači čitamo ulaz u vrsti prazan stek ∇ i koloni a, evidentiramo da je u pitanju akcija SHIFT. Kake efekte ima akcija SHIFT? Sa ulaza se potiskuje a na vrh steka, pri čemu se konsultuje potisna tabela. Potisna tabela se konsultuje za trenutni vrh steka pre izvršavanja te akcije SHIFT (određuje vrstu) i za onaj simbol koji potiskujemo na stek, znači a.(određuje kolonu). Vidimo da u ovoj prvoj vrsti za prazan stek i koloni a piše a_x i zbog toga je u koraku 2. na stek stavljen simbol a_x, dok je a uklonjeno su ulaza, tako da ovo drugo a postaje tekući ulazni simbol.

U drugom koraku, konsultujemo kontrolnu tabelu za vrstu a_x i za kolonu a, to je ulaz REDUCE(4), znači parser preduzima svođenje četvrte smene $<A> \rightarrow a$. Treba prvo skinuti jedan simbol (to jest, desnu stranu smene) sa parserskog steka, znači skida se a_x (ostaje u tom trenutku prazan stek), a zatim potisnuti <A>, levu stranu smene, na vrh steka. Znači konsultujemo potisnu tabelu za prvu vrstu (prazan stek) i <A>, piše $<A>_{11}$ i zbog toga je a_x zamenjeno u trećem koraku sa $<A>_{11}$. Ulaz ostaje nepromenjen, znači ovo a ostaje kao tekući ulaz i za sledeći korak.

U trećem koraku, vrh steka $<A>_{11}$ određuje vrstu kontrolne tabele, dok a određuje kolonu. Reč je o akciji SHIFT, koja znači da se ovo drugo a sada sa ulaza seli na vrh steka uz konsultaciju u vrsti $<A>_{11}$ i koloni a potisne tabele. Tamo piše a_4, znači potiskuje se na stek simbol a_4. Potom se a uklanja sa ulaza, znači na ulazu ostaje marker kraja ulazne sekvence kao tekući ulazni simbol.

U 4. koraku konsultujemo kontrolnu tabelu za a_4 i marker kraja ulaza, piše REDUCE(4), znači prepoznavanje četvrte smene. Slično kao ranije skida se jedan simbol, pošto je tolika dužina desne strane. Ostaje na steku $<A>_{11}$. U tom trenutku treba potisnuti <A>, levu stranu četvrte smene, znači u vrsti potisne tabele $<A>_{11}$ i koloni <A> piše da $<A>_{12}$ treba da se stavi na stek.

U 5. koraku konsutujemo kontrolnu tabelu u vrsti $<A>_{12}$ i koloni markera kraja ulaza. Piše REDUCE(2), znači sada se prepoznaje druga smena. Druga smena je prazna smena, dužina njene desne strane je nula, znači sa steka neće biti ništa skinuto, nego će se samo potisnuti leva strana veliko <S>. Prema tome, konsultujemo potisnu tabelu za vrstu $<A>_{12}$ i kolonu <S>, tu piše $<S>_1$. Ništa nije skinuto sa steka, samo se $<S>_1$ potiskuje na stek.

U 6. koraku gleda se kontrolna tabela u vrsti $<S>_1$ i koloni markera kraja ulaza, piše REDUCE(1), znači prepoznavanje odnosno sažimanje prve smene. Prva smena ima tri simbola na desnoj strani Znači treba tri stek simbola skinuti, time se dolazi do praznog steka i posle toga potisnuti na prazan stek <S> na levoj strani smene. Za prazan stek ∇ i <S> piše da treba $<S>_0$ da se stavi na stek.

U poslednjem, 7. koraku kontrolna tabela se konsultuje za vrstu $<S>_0$ i za kolonu $\dashv$ i to je akcija ACCEPT znači završetak rada parsera uz prihvatanje ulazne sekvence. Na ovaj način vidimo da je ova ulazna sekvenca sastavljena od dva a i markera kraja ulaza ispravna i da ju je parser prihvatio.

Ovaj parser je predstavnik parsera od dna ka vrhu i takvi parseri otkrivaju strukturu stabla izvođenja počev od listova stabla na gore, određujući čvorove roditelja prepoznavanjem leve strane smena na osnovu poznate desne strane. Zanimljivo je da je redosled smena koji parser prepoznaje tokom rada obrnut od redosleda pri krajnje desnom izvođenju date sekvence. Krajnje desno izvođenje je pojam koji smo definisali u uvodnoj lekciju sintaksne analize.

Dodatno, za rad potisnog parsera od dna ka vrhu važi:

- u svakom koraku rada, ako se na sadržaj steka nadoveže preostali ulaz, dobija se sentencijalna forma u krajnjem desnom izvođenju ulaza iz <S>.

Videćemo da u svakom koraku rada parser akcijom SHIFT potiskuje simbole sa ulaza na stek, dok se ne kompletira takozvana ručka, a onda se ručka zamenjuje levom stranom smene. Ručka je u suštini kompletna desna strana smene koja se u tom trenutku prepoznaje, a onda se akcijom REDUCE ta ručka zamenjuje levom stranom smene. Sve ovo što je rečeno biće ilustrovano na primeru.

Sledeća slika ponovo prikazuje trejs rada parsera koji smo analizirali, a gore je odgovarajuće izvođenje iz posmatrane gramatike sekvence aa iz startnog simbola <S>. Razmatra se originalna gramatika bez nulte smene, pa u izvođenju nema markera kraja ulaza. Da smo uzeli gramatiku koja je proširena nultom smenom, onda bismo imali u izvođenju još jedan korak pošto bismo kretali od <S'>. To ćemo kasnije u proceduri konstrukcije parsera detaljno definisati.

Prikazano je da se u krajnje desnom izvođenju iz startnog simbola <S>, zamenom korišćenjem prve smene, dobija sentencijalna forma <A><A><S>, zatim po pravilima krajnje desnog izvođenja menja se završno <S> primenom druge smene praznom smenom. Ostaje aa i zatim se prvo drugo <A>, a zatim i prvo <A>, primenom četvrte smene zamene sa dva a i tako dobijamo rezultujuću sekvencu aa.

<S> ⇒ rm <A><A><S> ⇒ rm <A><A> ⇒ rm <A> a ⇒ rm a a

↑ 1 ↑ **2** ↑ 4 ↑ 4

1.<S>→<A><A><S>

2.<S>→ ε

3.<S>→a

4.<A>→a

	sadržaj steka	preost. ulaz	akcija parsera
1.	∇	aa-|	SHIFT
2.	∇a_x	a-|	REDUCE(4)
3.	$\nabla <A>_{11}$	a-|	SHIFT
4.	$\nabla <A>_{11} a_4$	-|	REDUCE(4)
5.	$\nabla <A>_{11} <A>_{12}$	-|	REDUCE(2)
6.	$\nabla <A>_{11} <A>_{12} <S>_1$	-|	REDUCE(1)
7.	$\nabla <S>_0$	-|	ACCEPT

Uporedimo ovo izvođenje sa trejsom rada parsera. Ono što možemo uočiti je da kad pogledamo ovaj niz primena smena 1244, isti niz se pojavljuje i u trejsu rada parsera, ali kada se taj trejs posmatra u suprotnom pravcu od poslednjeg koraka ka prvom. To je jedan od tih opštih principa parsiranja od dna ka vrhu, da redosled prepoznavanja smena u toku parsiranja od dna ka vrhu odgovara obrnutom redosledu krajnje desnog izvođenja, što je i u ovom primeru potvrđeno.

Dodatno, rekli smo da se pojam ručke sentencijalne forme odnosi na pojavu, u sentencijalnoj formi, desne strane smene koja je posljednja primenjena u krajnje desnom izvođenju. Znači ručka može da se identifikuje u bilo kom koraku izvođenja, odnosno rada parsera, a produkcija ručke je poslednja produkcija (ili smena) primenjena u krajnje desnom izvođenju sentencijalne forme. U prethodnom primeru, ručka za finalnu sentencu je početno slovo a, znači produkcija ručke je 4. smena. Kada gledamo trejs u prirodnom smeru, videli smo da smo <A> zamenili sa a. Međutim, za ručku i za rad parsera gleda se suprotno, odnosno gleda se ručka u stvari na vrhu steka, znači pojava kompletne desne strane četvrte smene i to je ovo ovde malo a, produkcija smene je 4.

Šta je još dodatno rečeno od svojstava parsiranja? Kada se nadovežu stek i ulaz u bilo kom koraku rada parsera od dna vrhu, dobijamo neku sentencijalnu formu u krajnje desnom izvođenju.

To možemo videti na ovom primeru. Uzmimo bilo koji korak rada parsera, na primer 5. korak. Sadržaj steka je dno steka ∇ i dva <A> i na to nadovezujemo preostali ulaz koji je samo marker kraja. To treba da mapiramo na neku sentencijalnu formu u krajnje desnom izvođenju. Pošto ovde nije određeno izvođenje za proširenu gramatiku nego samo za originalnu gramatiku, onda ćemo zanemariti marker kraja, tako da će ostati ova dva <A>, koja odgovaraju 3. koraku izvođenja. Znači mapirali smo stek plus preostali ulaz na neku sentencijalnu formu.

Isto razmatranje možemo ponoviti recimo za 6. korak trejsa rada parsera. Stek je <A><A><S> i ualz je marker kraja (koji zanemarimo), dobijamo drugi korak izvođenja. Ili recimo u 2. koraku trejsa, na steku je a, na ulazu su a i marker kraja. Spajanjem dobijamo dva a, naravno odbacuju se dno steka i marker kraja ulaza. To odgovara poslenjem koraku izvođenja.

Generalno važi da, gde god su u trejsu parsiranja REDUCE akcije, na vrhu steka se formira desna strana smene koja će biti redukovana i ta desna strana se u radu parsera pojavljuje na vrhu steka, a u sentencijalnoj formi čini ručku. Na primer, u 6. koraku trejsa <A><A><S> čine ručku i u pitanju

je prva smena, znači to je produkcija ručke. Ovime završavamo izlaganje o povezanosti gramatičkog (krajnje desnog) izvođenja i rada parsera od dna ka vrhu.

Zašto proučavamo shift/reduce metode parsiranja? yacc familija alata. takozvanih generatora parsera je zasnovana upravo na shift/reduce parsiranju. Oni ne samo što generišu parser kao potisni automat, nego generišu i program na nekom od jezika kao što su C, C++, java i tako dalje, Znači za svaki jezik može da se nađe takav alat, recimo alat cup je varijanta ovog yacc parsera za javu i koristimo ga za praktičan projekat u okviru kursa.

Konstrukcija LR(0) parsera

Do sada smo videli rad i primer jednog gotovog shift-reduce parsera, ali ono što još ne znamo, to je kako za zadatu gramatiku dobiti takav potisni automat. Mi ćemo proceduru konstrukcije prikazati za nekoliko klasa gramatika. Najjednostavnija klasa su takozvane LR(0) gramatike, a kasnije ćemo videti da proširivanjem te osnovne procedure konstrukcije možemo dobiti i shift-reduce parsere za neke generalnije klase gramatika, sve do one klase koja nam za praktičnu primenu treba. Kod shift-reduce parsiranja na LR(0) konceptu, taj parser je karakterističan po tome što se strategija odlučivanja zasnivaju isključivo na sadržaju steka. Ulazni simboli se ne uzimaju u obzir pri odlučivanju, što za direktnu posledicu ima to da se kontrolna tabela parsera svodi na jednu kolonu. Znači pri odabiru sledećeg koraka gleda se isključivo vrsta znači vrh steka, a ne gleda se ulaz. Zašto je baš oznaka LR(0)? Ovo prvo L znači da parser čita ulaznu sekvencu s leva na desno, da u njoj nalazi krajnje desno odnosno rightmost izvođenje i da na ulazu, pri odlučivanju o sledećem koraku rada, gleda nula simbola.

Konstrukcija LR(0) parsera zasniva se na pojmu LR(0) konfiguracije. Engleski termin je još i item. Radi se o tome da se u gramatičku smenu, na određeno mesto na desnoj strani smene između dva gramatička simbola, ili sasvim napred ili sasvim nazad, doda tačka i takva smena plus tačka na desnoj strani zove se item ili LR(0) konfiguracija.

1. <S> → a
2. <S> →(<S> <R>
3. <R> →, <S> <R>
4. <R> →)

Ilustrovaćemo konstrukciju parsera na primeru ove jednostavne gramatike koje ima četiri smene i dva neterminala <S> i <R>. Vidimo da su terminalni simboli a, otvorena zagrada, zarez i zatvorena zagrada. Ako posmatramo drugu smenu koja ima na levoj strani <S>, a na desnoj strani otvorenu zagradu, <S> i <R>, praktično imamo četiri razna itema ili LR(0) konfiguracije.

<S> → • (<S> <R>
<S> → (• <S> <R>
<S> → (<S> • <R>
<S> → (<S> <R> •

Znači tačku možemo staviti na sam početak desne strane, zatim iza otvorene zagrade a ispred <S>, između <S> i <R> i iza <R>. Konfiguracije koja imaju tačku na samom početku se zovu konfiguracije zatvaranja ili closure, a ove ostale konfiguracije koje nemaju tačku na početku, znači tačka može biti i na kraju, ali treba da bude neki gramatički simbol ispred tačke, zovu se bazične konfiguracije. Mi ćemo uvesti i alternativne, konciznije oznake za konfiguracije, tako što određena konfiguracije gde je tačka neposredno iza nekog pojave nekog gramatičkog simbola na desnoj

strani smene, biće označena tim gramatičkim simbolom i, ako je on jedini na desnoj strani smene, indeksom smene.

Ovde u <S> → (<S> <R> • je tačka neposredno iza <R> i zato ćemo konfiguraciju označavati sa $<R>_3$, jer je reč je o trećoj smeni. Da je tu bilo dva <R> na desnoj strani, onda bismo uveli dvostruki Index $<R>_{31}$ $<R>_{32}$.

Što se tiče praznih smena, one imaju samo jednu konfiguraciju i možemo je označiti ili tako što ostavimo epsilon na desnoj strani, pa tačku stavimo bilo pre njega, bilo posle njega, ili prosto stavimo samo tačku. Pošto epsilon ionako znači odsustvo bilo kakvog simbola znači u smeni postoji samo jedna konfiguracija zatvaranja.

<X> → ε • ili <X> → •

Za potrebe konstrukcije ovih parsera, o tome je već bilo reči, proširićemo svaku gramatiku dodatnom smenom, uvešćemo novi startni simbol (ovde u oznaci <S'>) a na desnoj strani te nove proširujuće smene rednog broja nula biće originalni startni simbol, a iza njega marker kraja ulazne sekvence.

0. <S'> → <S> ⊣

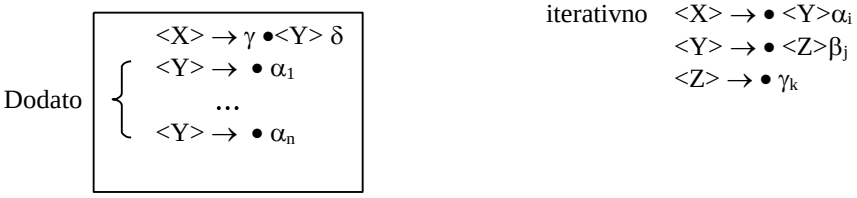
Na taj način u skup ulaznih simbola dodajemo marker kraja ulaza i konstrukcija parsera je onda malo jednostavnija za objašnjavanje. Konstrukcija počinje konstrukcijom takozvanog karakterističnog LR(0) automata, detektora ručki. Videćemo zašto se on zove tako, ali prvo da vidimo o čemu se radi. To je jedan konačni automat, čija stanja odgovaraju skupovima LR(0) konfiguracija ili itema i konstruisqćemo ga u obliku grafa prelaza koristeći operacije koje ćemo definisati: closure0 i goto0.

```
function Closure0(ulaz: skup bazičnih konfiguracija S) rezultat: skup konfiguracija;
    Inicijalno S' := S;
    for (∀K ∈ S')
        if (u konfiguraciji K tačka se nalazi ispred neterminala <X>)
            thenfor (∀ smenu P)
                        if (P je oblika <X>→ α)
                            Napomena: α je proizvoljan niz znakova)
                            then S' := S' ∪ { <X> → • α};
                        end if;
                end for;
        end if
    end for;
    return S';
end function Closure0;
```

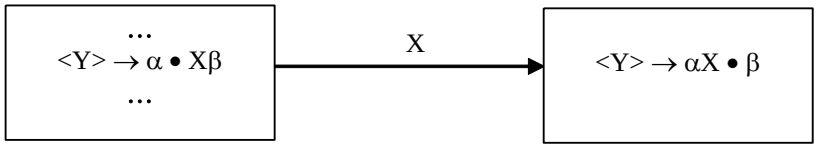
Operacija closure(0) se obavlja nad nekim skupom konfiguracija S i dodaje nove konfiguracije ili iteme u taj isti skup S. Formalno je opisujemo matematičkim izrazom da je to skup S svih takvih konfiguracija zatvaranja (onih koje imaju tačku na početku desne strane) smena za neki neterminal <B>, koji se pojavljuje u drugim konfiguracijama iz skupa S, takvim da se tačka nalazi ispred <B>.

$$CLOSURE(s) = s \cup \{<B> \rightarrow \bullet\gamma \mid <A> \rightarrow \alpha\bullet<B>\beta \in CLOSURE(s)\}$$

Operacija closure0 mnogo jasnije može da se objasni grafički, kao na slici koja sledi. Uokvireno je neko stanje karakterističnog automata, u njemu znači već postoji nekakva konfiguracija sa proizvoljnim početkom α, može biti nula ili više gramatičkih simbola, zatim ide tačka i bitno je da neposredno iza tačke ide neki naterminal. Tada closure0 operacija nalaže da dodamo sve smene koje imaju na levoj strani <B> i da dodamo sve iteme zatvaranja tih smena (odnosno tačku na početku desne strane svake od tih smena). Znači ako <B> ima tri smene, dodaćemo tri Nova itema. Cela stvar treba iterativno da se ponavlja dok god ima promene. Ako zbog ovakvog itema sa tačkom ispred <B> dodamo novi item gde je <B> na levoj strani, a tačka recimo ispred nekog <C>, onda treba dodati i smene za <C> koje imaju tačku na početku desne strane i tako dalje, do god postoji promena.



Druga operacija Goto0 nam definiše prelaze među stanjima karakterističnog LR(0) automata. Da pogledamo grafičko objašnjenje na sledećoj slici. Znači ako imamo neko stanje automata, u kome imamo item gde je tačka ispred nekog gramatičkog simbola X i samo je bitno da tačka nije na samom kraju nego da ima nešto ispred tačke, onda će takvo stanje obavezno imati izlaznu granu označeno tim gramatičkim simbolom X (može biti i terminali neterminal). I biće prelaz u novo stanje, koje će imati item iste ove smene, ali s tom razlikom što je sada tačka prebačena iza X. Ovo sve nalaže operacija goto0.



Algoritamska formulacija Goto0 operacije:

```
function GoTo0(ulaz: skup konfiguracija S; simbol X) rezultat: skup konfiguracija;
    Inicijalno S' := ∅;
    for (∀K ∈ S)
        if (K je oblika <Y> → α • X β)
                Napomena: α i β proizvoljne sekvence
                then S' := S' ∪ {<Y> → α X • β}
        end if;
    end for;
    return Closure0(S');
end function GoTo0;
```

Matematička (skupovna) formulacija Goto0 operacije:

$$GOTO(s,X) = CLOSURE(\{<A>\rightarrow\alpha X\bullet\beta \mid <A>\rightarrow\alpha\bullet X\beta \in s\})$$

Ono što možda ovde treba napomenem, znači ovde nije prikazano u primeru, ali u opštem slučaju može se desiti, da u ovom početnom stanju imamo dva različita itema od dve različite smene, a da u oba od njih je tačka ispred istog X. Zamislimo situaciju kao na prethodnoj slici, ali da je u levom

pravougaoniku dopisan još neki novi item, ali koji isto ima tačku ispred X. Onda bi bilo pogrešno crtati iz tog stanja još jednu novu granu označenu istim X. Nije u pitanju nedeterministički automat, nego deterministički. Znači jedinstvena je grana sa X i treba desnom stanju na slici dodati i taj drugi item i prebaciti tačku iza X.

Sada kada znamo ove dve operacije, evo kako se konstruiše karakteristični LR(0) automat. Njegovo startno stanje dobija se počevši od konfiguracije zatvaranja proširujuće smene (to je ova konfiguracija prikazana u nastavku) tako što se na nju primeni closure0 operacija i tako se dobije startno stanje.

Closure0({<S'> →• <S> ⊣ })

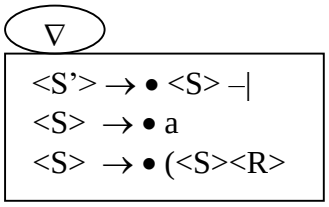
Potom, kada god ubacimo u automat neko novo stanje, na njega primenimo proceduru goto0. Redom tražimo za razne ulazne simbole gde goto0 generiše nove izlazne grame i stanja i sva ta nova stanja dodajemo u automat. Kada god dodamo novo stanje, na njega primenimo closure0 operaciju. I tako dalje, posle ponovo sa goto0 obrađujemo ta novodobijena stanja i sve se ponavlja dok god postoji promena u grafu.

Primer

1. <S> → a
2. <S> →(<S> <R>
3. <R> →, <S> <R>
4. <R> →)

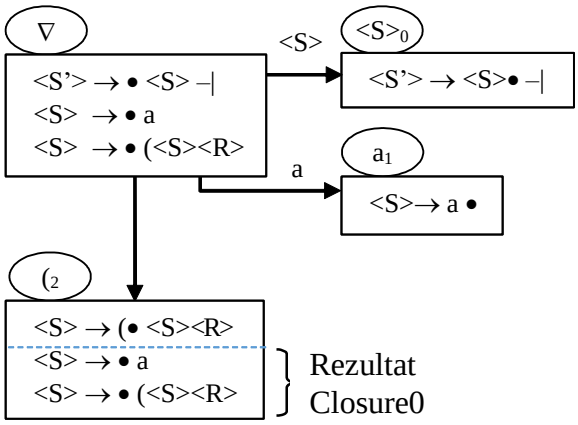
Ilustrovaćemo na ovoj gramatici od četiri smene celu ovu proceduru. Prvo se konstruiše startno stanje automata. U grafu ćemo čvorove prikazivati pravougaonicima. Unutra ćemo pisati konfiguracije. Gramatici dodajemo proširujuću smenu <S'>→ <S>–|.

Dodajemo tačku na početak desne strane i sad treba primeniti closure0 operaciju. Operacijom closure0 dodaju se još druge dve konfiguracije u startno stanje. Pošto je tačka ispred <S>, potrebno je za ove dve smene koje imaju <S> na levoj strani, dodati njihove konfiguracije zatvaranja. Time smo finalizovali startno stanje automata, prikazano na sledećoj slici. Interesantno je da ćemo za svako stanje uvesti i koncizniju oznaku stanja koju ću ja ovde ovako zaokruženo uvek pisati. To je prosto alternativna oznaka ovog istog stanja. Za startno stanje usvojićemo da je oznaka ista kao za dno steka.



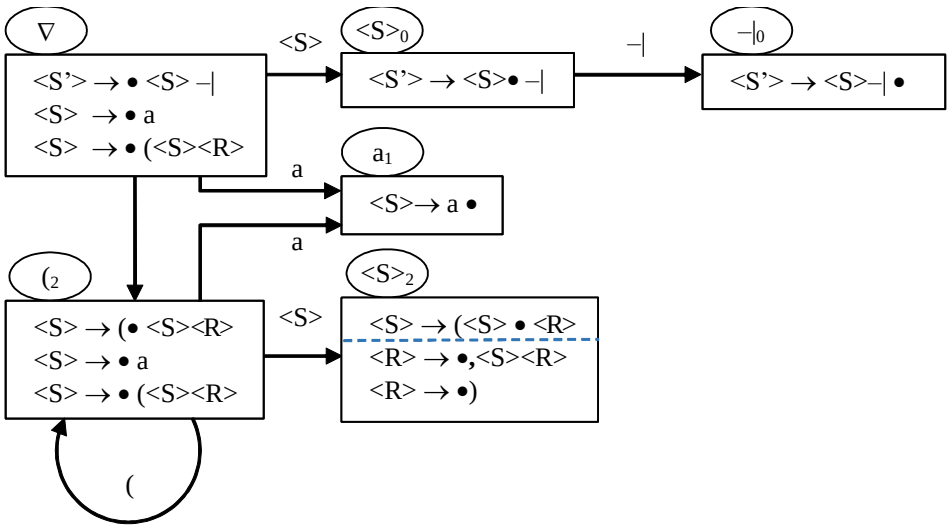
Sada na startno stanje primenujemo goto0 operaciju, za sve gramatičke simbole koji se u ovom startnom stanju nalaze neposredno iza tačke u nekoj od konfiguracija. To su <S>, a i). Unosimo izlaznu granu označenu sa <S> i ona ide u novo stanje, gde prebacujemo iz konfiguracije <S'> → • <S> –| tačku iza <S>. Zatim imamo izlaznu granu označenu sa a u novo stanje prebacujemo item <S> → • a**,** ali prebacujemo tačku iza a. Konačno, imamo izlaznu granu označenu otvorenom zagradom i prebacujemo item <S> → • (<S><R> sa tačkom iza otvorene zagrade u novom stanju.

Dobili smo tri nova stanja, kao na slici koja sledi. Odmah uvodimo prema onome objašnjenju od ranije i konciznije oznake stanja. Te oznake se uvek formiraju prema bazičnim konfiguracijama u stanjima i najčešće, ako je u pitanju jedna konfiguracija, rekli smo da je oznaka onda gramatički simbol koji je neposredno levo od tačke i redni broj smene. Naravno nekad može da se desi da postoji više itema iz više smena, pa bi onda umesto konkretnog rednog broja smene mogli da stavimo one simbole xyz i tako dalje.



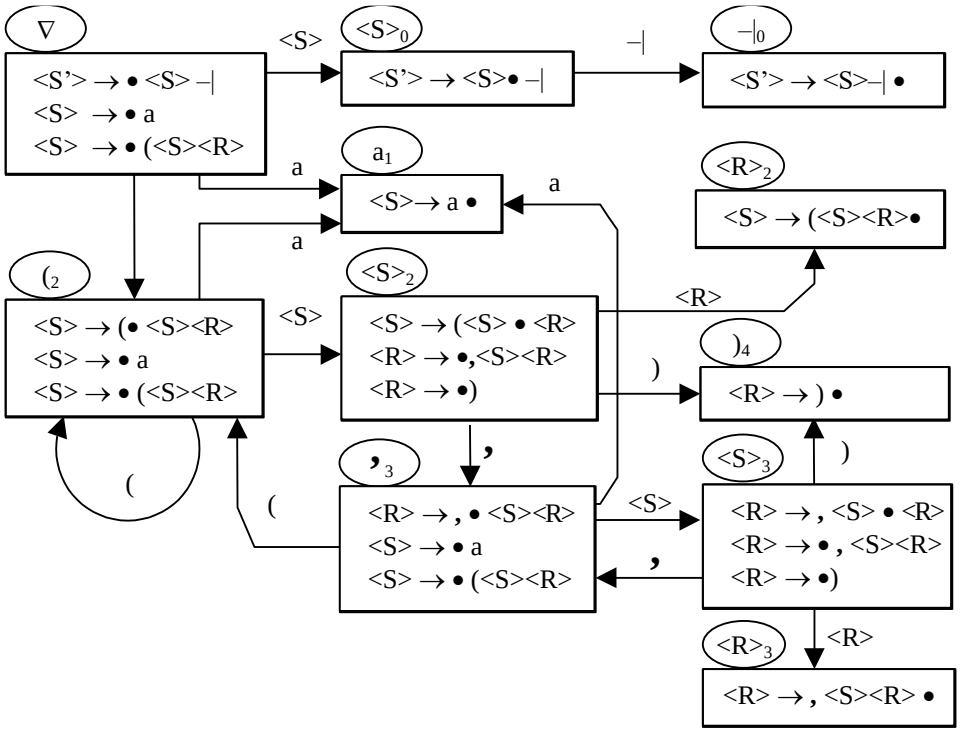
Mora se na svako od ovih novih stanja, da bismo ih finalizovali, primeniti operacija zatvaranja. Zatvaranjem se ne dodaju nove konfiguracije u ova stanja <S>$_0$ i a_1 pošto tačka nije ispred neterminala. Međutim, stanju ($_3$ tačka je bila ispred velikog <S> i zbog toga moramo dve nove konfiguracije da dodamo zatvaranjem kao što smo to uradili u ovom inicijalnom stanju i tek sad smo finalizovali ova tri nova stanja.

Konstrukcija se sada nastavlja tako što gledamo, iz ovih novih stanja, gde sve možemo dospeti primenom operacije goto0. Stanje <S>$_0$ treba da ima izlaznu granu označenu markerom kraja ulaza. Stanje a_1 nema više izlaznih grana, pošto je tačka na samom kraju. Stanje ($_2$ će imate izlazne grane označene sa a, <S> i otvorenom zagradom. Slika prikazuje primenu operacije goto0 i pratećeg closure0 na nova stanja.



Vidimo da je stanje ($_2$ zanimljivo, zato što izlazna grana označena otvorenom zagradom vodi u isto ovo stanje. Skup konfiguracija stanja u stvari predstavlja identitet stanja i ne treba crtati dva

različita stanja sa identičnim skupovima konfiguracija, nego je reč o jednom istom stanju i zato imamo ovu povratnu granu. Iz stanja ($_2$ prateći granu a idemo u postojeće stanje a_1. Konačno, kada razmatramo prelaz po <S>, onda se u itemu <S> → (• <S><R> prebacuje tačka iza <S>, a ispred <R>, to je sad novi item koji do sada nismo imali, pa to odgovara ovom novom stanju. Na slici već vidimo da je u ovom stanju primenjena closure0 operacija, tako da smo dobili finalni izgled stanja. Cela procedura se nastavlja u još nekoliko iteracija dok se konačno ne finalizuje ceo automat.



Gornja slika prikazuje finalni karakteristični automat za ovu gramatiku. Može da se kaže da ovaj automat ujedno predstavlja potisnu tabelu parsera. On može da se napiše u drugom obliku, da ga umesto grafa prikažemo tabelarno, umesto ovih itema da iskoristimo ove konciznije oznake i praktično dobijamo potisnu tabelu koja je prikazana na slici koja sledi levo od duple uspravne linije. Karakteristični automat ujedno poseduje sve informacije da se konstruiše i kontrolna tabela, što ćemo razmotriti u nastavku.

Popunjavanje LR(0) kontrolne tabele

Rekli smo da se LR(0) kontrolna tabela sastoji se samo od jedne kolone, znači u svakoj vrsti treba popuniti samo jedan ulaz. Svaka vrsta kontrolne tabele odgovara nekom stanju karakterističnog automata, to jest, odgovarajućoj vrsti potisne tabele, znači možemo u automatu videti koji je skup konfiguracija unutar stanja.

- Ako stanje sadrži konfiguraciju oblika <X> → α • x β, ulaz treba popuniti akcijom SHIFT.
- Ako stanje sadrži konfiguraciju oblika <Y> → γ •, odnosno tačka se nalazi na kraju desne strane, ulaz treba popuniti akcijom REDUCE(<Y> → γ). Izuzetno, ako se radi o nultoj smeni, akcija je ACCEPT.
- Ukoliko gornja pravila na jednoznačan način određuju akciju za svaki ulaz kontrolne tabele, gramatika pripada klasi LR(0).

U kontrolnoj tabeli LR(0) parsera praktično ne može da se nađe operacija REJECT koju smo pomenuli definišući shift-reduce parsere generalno. LR(0) parser može da odbaci ulaznu sekvencu, samo ako u toku rada naiđe na prazan ulaz potisne tabele. Za određeni ulaz kontrolne tabele može da se desi da je prema gornjim pravilima moguće staviti više od jedne akcije, ali tu situaciju ćemo kasnije da razmotrimo.

Za naš tekući primer, može čak i pregledom ove konciznije potisne tabele da se popunjava ova kontrolna tabela. Ako je neka vrsta potisne tabele neprazna, znači bilo gde, u bilo kom ulazu ima nešto što nije prazan ulaz, onda se u odgovarajuću vrstu kontrolne tabele stavlja SHIFT.

Potisna i kontrolna tabela imaju paralelne vrste, tako da na slici nisu ponavljane oznake za stanja, nego su spojene potisna i kontrolna tabela, razdvojene su samo uspravnom duplom linijom. Vrste kontrolne tabele koje nismo popunili sa SHIFT, biće popunjene REDUCE akcijama. Obično nam indeks ovog simbola steka govori o kojoj smeni je reč. Simbol a_1 je pojava a na kraju prve smene, onda će biti u vrsti a_1 biti redukcija prve smene. Kolona $<R>_2$, odgovara konfiguraciji koja ima tačku na samom kraju smene i u pitanju je druga smena, znači zbog toga je u toj vrsti navedena akcija REDUCE(2). Slično prethodnom, kolona $<R>_3$ popunjena je akcijom REDUCE(3). a kolona $)_4$ akcijom REDUCE(4). Pošto je u koloni $\dashv_0$ indeks nula, rekli smo da je REDUCE(0) u stvari ACCEPT, znači kraj rada parsera uz prihvatanje ulazne sekvence.

	<S>	<R>	(	,	)	a	⊣	
∇	$<S>_0$		$(_2$			a_1		SHIFT
$<S>_0$							$\dashv_0$	SHIFT
$\dashv_0$								ACCEPT
a_1								REDUCE(1)
$(_2$	$<S>_2$		$(_2$			a_1		SHIFT
$<S>_2$		$<R>_2$		$,_3$	$)_4$			SHIFT
$<R>_2$								REDUCE(2)
$,_3$	$<S>_3$		$(_2$			a_1		SHIFT
$<S>_3$		$<R>_3$		$,_3$	$)_4$			SHIFT
$<R>_3$								REDUCE(3)
$)_4$								REDUCE(4)

potisna tabela — kontrolna tabela

Možemo primetiti da ovako dobijen parser ima razlika u odnosu na parser u primeru koji je ilustrovao generalni koncept shift-reduce parsiranja. U čemu se sastoje razlike?

Prva razlika je u tome što je ovaj ovde parser jednostavniji, njegova kontrolna tabela ima samo jednu kolonu. Prethodni parser je imao više kolona u kontrolnoj tabeli, onda je na bazi ulaznog simbola bila birana akcija. U suštini to znači da onaj prvi primer nije predstavljao LR(0), nego neku od generalnijih kategorija shift-reduce parsera.

Druga razlika u odnosu na onaj prethodni primer: u vrsti $<S>_0$ se ranije odmah radilo prihvatanje ulazne sekvence, ovde je stajao ACCEPT. U potisnoj tabeli nije ni bilo kolone za marker kraja ualza i nije bilo stek simbola $\dashv_0$. Šta možemo reći o odnosu rada parsera iz ranijeg primera i ovog sadašnjeg? Ovaj parser ima jedan korak više. On kada dođe do te situacije da na vrhu steka ima $<S>_0$ i na ulazu mu je marker kraja, on još jednom uradi SHIFT, potisne taj marker kraja na stek i onda tek uradi ACCEPT. U ranijem primeru, onog generalnijeg shift-reduce parsera, u suštini je

urađena optimizacija za taj jedan korak, znači u toj situaciji se već ovde radilo ACCEPT i nije se potiskivao marker kraja na stek. Na taj način je malo optimizovan rad parsera za jedan korak i zbog toga takođe nije uvodjen simbol $-|_0$ kao simbol steka. U tekućem primeru nije primenjena ta sitna optimizacija.

Konflikti u SHIFT-REDUCE parsiranju

Ostalo je još samo da kažemo šta se dešava u slučaju kada pravila popunjavanja kontrolne tabele ne dovode do jednoznačnog popunjavanja kontrolne tabele. To su takozvane situacije konflikata u shift-reduce parsiranju. Oni mogu da se pojave i kod LR(0) parsera čiju smo konstrukciju upravo proučili, ali mogu da se pojave i kod drugih kategorija shift-reduce parsera koje su generalnije i koje ćemo tek proučavati. Kako se manifestuju te konfliktne situacije? Postoje dve vrste konflikata: takozvani shift/reduce i reduce/reduce konflikti.

Konflikt tipa shift/reduce se pojavljuje u situaciji kada se u stanju karakterističnog automata pojave dva različita itema, od kojih jedan ima tačku na samom kraju, a drugi item ima tačku negde u sredini desne strane (ne na samom kraju).

<X> → a •
<Y>→ a • b

Zanimljivo je iz pravila konstrukcije karakterističnog automata sledi da gramatički simbol neposredno levo od tačke uvek mora biti identičan, znači nije slučajno napisano isto a levo od tačke u oba itema. Ovakva situacija realno može da se pojavi vezano za dve različite smene.

Prema pravilima za kontrolnu tabelu, item koja ima tačku na samo kraju bi zahtevao da u kontrolnoj tabeli u toj vrsti stoji operacija redukovanja smene <X> → a. Drugi item bi zahtevao akciju SHIFT, da se dalje potiskuje b sa ulaza na vrh steka. Znači to je takozvani shift/reduce konflikt jer u nekim situacijama tokom rada, parser bi na tom mestu trebalo da preduzima SHIFT, a u nekim drugim situacijama da preduzima REDUCE, ali parser nema dodatnih informacija da zna kada se odluči za SHIFT, a kada za REDUCE.

Druga vrsta konflikata je takozvani reduce/reduce konflikt, gde u nekom stanu karakterističnog automata mogu da se pojave dva različita itema koji oba imaju tačke na krajevima, ali da je reč o različitim smenama. Tada bi prvi item zahtevao da se redukuje smena <X> → d, a drugi da se redukuje smena <Y> → d.

<X> → d •
<Y>→ d •

Teorija kaže da, ako se pojave konflikti, onda gramatika nije u kategoriji LR(0) i ne može da se konstruiše LR(0) parser za nju. LR(0) je dosta ograničena klasa parsera i gramatika. Dovoljno je da se u gramatici pojavi prazna smena i ona će uvek izazvati konflikt. Evo primera sa vrlo malom gramatikom:

0. <S'> → <S> –|
1. <S> → a
2. <S> → ε

Sledeća slika prikazuje početno stanje ∇ karakterističnog automata. Zatvaranjem smo osim prvog itema dodali još dva i odmah uočavamo konflikt u vrsti ∇ kontrolne tabele. Item

<S> → •a bi tražio da u kontrolnoj tabeli stoji SHIFT, a item <S> → •, pošto je tačka na kraju smene, bi zahtevao redukciju ove prazne smene.

<S'> → • <S> –|
<S> → • a
<S> → •

Rekli smo da teorija kaže da u situaciji pojave konflikata mora da se pređe na neki drugi metod, odnosno ne možemo da konstruišemo odgovarajući parser. Kako rešiti taj problem? Pošto za praksu nama treba naravno da dobijemo funcionalan parser za neku gramatiku koje nam je data, postoji nekoliko rešenja tog problema.

Jedno od rešenja je da se sa LR(0) kategorije pređe na neku generalniju kategoriju parsiranja. To mogu biti i druge, generalnije kategorije shift-reduce parsera. Takođe postoje još generalniji algoritmi parsiranja koji su primenljivi praktično na sve bezkontekstne gramatike, ali se ti najgeneralniji algoritmi ne koriste u konstrukciji kompajlera, zato što oni ili nisu deterministički, ili zahtevaju nepraktične transformacije originalne gramatike, i što je najvažnije, imaju veću složenost (lošije performanse).

Red funkcije složenosti shift-reduce parsera, koji spade u determinističke parsere pošto u svakom koraku preduzima jednoznačnu akciju, je linearan u odnosu na dužinu ulaza, znači oni rade brzo, dok najgeneralniji algoritmi koji mogu da parsiraju bilo koju bezkontekstnu gramatiku imaju složenost reda n^3 u odnosu na dužinu n ulaza (to su, na primer, Early-ev algoritam koji je nedeterministički i CYK algoritam koji zahteva transformaciju gramatike u normalnu formu Čomskog). U novije vreme neki generatori parsera (na primer byacc) imaju ugrađen GLR algoritam parsiranja (generalizovani LR algoritam). Radi se o nedeterminističkom algoritmu koji pri pojavi konflikata razgranava stablo parsiranja u šumu stabala i u paraleli sledi sve moguće alternative, da bi se na kraju na određeni način razrešile semantičke akcije, tako što se ili izabere jedna alternativa po definisanom prioritetu, ili se rade sve akcije za sve varijante i finalno razrešavanje prepušta korisniku. Ovaj algoritam ima "skoro linearne" performanse za gramatike koje nemaju mnogo konflikata, ali je složeniji za upotrebu, pa nema zabeležene primene u realnim komercijalnim kompajlerima. Znači moguće je upotrebiti neki generalniji algoritam, to je jedno od rešenja ako će on dati zadovoljavajući rezultat.

Takođe je moguće napraviti ekvivalentnu transformaciju gramatike, kao što su već izloženi primeri eliminacije višeznačnosti. U narednim poglavljima navešćemo još nekoliko primera tih transformacija, koje ne garantuju da ćemo bilo kakvu gramatiku u opštem slučaju moći parsirati. Bilo kakva bezkontekstna gramatika može da opisuje jezik koji nije uopšte moguće deterministički pasirati, ali u praksi gramatike za programske jezike često mogu nekim relativno jednostavnim transformacijama da se upodobe i da se dobije gramatika koja moguće parsirati.

Sledeći pristup rešenju problema je da zadržimo i gramatiku i metod parsiranja koji daje konflikte, ali da onda, na određeni način, pomognemo parseru da u toku rada, uz dodatne informacije koje mu obezbedimo, zna da li da uzima jednu akciju ili drugu. u tim konfliktnim situacijama.

Generatori parsera familije yacc koji primenjuju ovaj algoritam i automatski konstruišu parser, detektuju ako je nastao konflikt i imaju ugrađeno podrazumevano razrešavanje konflikata. Znači ako korisnik ništa ne pomogne dodatnim informacijama, podrazumevano će se shift-reduce konflikt uvek razrešavati u korist shift-ovanja, znači alat će prosto staviti shift, a zanemariti sve

druge akcije. Reduce/reduce konflikt će se tako razrešiti da veći prioritet ima smena koja je prva navedena, znači prosto će se staviti bezuslovno redukcija te smene. Kod shift/reduce konflikata ovaj podrazumevani način razrešavanje često je i adekvatan. Naravno uvek mora da se izvrši analiza svakog konflikta zašto se dešava, ali često u nekim aritmetičkim izrazima sa levo asocijativnim operatorima adekvatno rešenje je da se konflikt razreši u korist akcije shift. Ali reduce/reduce konflikti su ozbiljniji problem i obično zahtevaju da se gramatika transformiše da bi se konflikt izbegao.

Postoji način da se alatu pomogne po pitanju shift-reduce konflikata bez transformisanja gramatike. U pitanju su precendence (%prec) direktive kojima se uređuju po prioritetu terminalni simboli, kao i %left, %right i %nonassoc koji određuju asocijativnost terminala (svih, ili samo onih koji učestvuju u konfliktnim situacijama). Određivanje prioriteta i asocijativnosti je relativno lako na primer za simbole koji predstavljaju operatore aritmetičkih izraza, pošto operatori prirodno imaju sopstvene prioritete, na primer, množenje je prioritetnije od sabiranja. Osnovna ideja je da se u konfliktnoj situaciji uporede poslednji terminal u smeni koja je kandidat za reduce i tekući ulazni simbol po prioritetu i asocijativnosti (naravno ako su ove osobine definisane za oba tokena, inače se primenjuje podrazumevano razrešavanje konflikata).

Ako je u pitanju shift-reduce konflikt, on će se razrešiti akcijom shift ili reduce u korist prioritetnijeg terminala. Ako je prioritet oba terminalna simbola isti, koristi se asocijativnost. Leva asocijativnost razrešava se u korist REDUCE, desna u korist SHIFT, a ne-asocijativnost izaziva prijavu greške. Čitalac se upućuje da pogleda primer u Zbirci zadataka koji ilustruje ove načine razrešavanja konflikata, takođe postoji i detaljna diskusija u uputstvima za yacc ili bison alate.

Složenije klase shift-reduce parsera

U prethodnom odeljku videli smo konstrukciju najjednostavnijih shift-reduce parsera klase LR(0). U nastavku ćemo razmotriti složenije klase shift-reduce parsera. Prva od njih je takozvana SLR(1) klasa, Skraćenica dolazi od Simple LR(1). Kod ove tehnike parser će konsultovati tekući ulaz kada određuje korake rada. Dobitak je što se smanjuje broj potencijalnih konflikata u kontrolnoj tabli koja sada ima oblik pune matrice, vrste su određene stek simbolima, a kolone su određene ulaznim simbolima, to jest, terminalnim simbolima gramatike. Osnovna ideja je da se redukcija neke smene $<X> \rightarrow \gamma$ izvršava samo ako je tekući ulazni simbol iz skupa follow(x) pri čemu je to jedan novi pojam koji ćemo sada definisati.

Za svaki neterminal definiše se follow skup tog posmatranog neterminala <X> kao skup terminalnih simbola koji se mogu pojaviti neposredno iza <X> u bilo kojoj sentencijalnoj formi izvedenoj iz <S> –| gde je <S> startni simbol gramatike. To je prikazano i donjom formulom, gde je t neki terminal iz skupa terminala V_t, <X> je posmatrani neterminal, dok su α i γ proizvoljne sekvence dužine nula ili više. Znači ako se u bilo kojem izvođenju neposredno iza <X> može pojaviti terminal t, on se ubacuje u njegov follow skup.

$$\text{FOLLOW}(<X>) = \{t \mid t \in V_t \cup \{\dashv\} \wedge <S> \dashv \stackrel{*}{\Rightarrow} \alpha <X> t\, \gamma\}$$

Definicija nam ništa ne kaže od toga kako ćemo računati follow skupove. Da bismo izračunali follow skupove, moramo uvesti pojmove poništivosti neterminala, takođe i pojam takozvanih first skupova neterminala.

Neterminal <X> je poništiv ako i samo ako se iz njega može izvesti prazna sekvenca:

$$<X> \stackrel{*}{\Rightarrow} \varepsilon$$

Sekvenca gramatičkih simbola je poništiva ako i samo ako se sastoji od nula ili više poništivih neterminala. Skup poništivih neterminala P formiramo na sledeći način:

1. U skup P staviti sve neterminale koji se pojavljuju na levoj strani neke prazne smene.
2. Razmatrati redom smene čije se desne strane sastoje isključivo od neterminala, a leva strana se ne nalazi u skupu P. Ako se svi neterminali desne strane pojavljuju u skupu P, dodati levu stranu smene u P.
3. Sve dok postoji promena u skupu P ponavljati korak 2, u suprotnom završiti postupak.

Skup FIRST(<X>) za proizvoljan neterminal <X> predstavlja skup svih terminalnih simbola kojima može početi sentencijalna forma izvedena iz <X>:

$$FIRST(<X>) = \{ t \mid t \in V_t \wedge \alpha \in V^* \wedge <X> \Rightarrow^* t\,\alpha\}$$

U gornjoj formuli V_t označava skup svih terminala gramatike, t je terminal, a α proizvoljna sekvenca dužine nula ili više. Moguće je definisati FIRST skupove sentencijalnih formi: Neka je $\alpha = A_1A_2...A_iA_{i+1}...A_n$ proizvoljna sentencijalna forma u kojoj prvih i simbola predstavlja poništive neterminale. Tada skup FIRST(α) predstavlja uniju FIRST skupova svih simbola sa levog kraja sekvence zaključno sa prvim koji nije poništiv:

$$FIRST(\alpha) = FIRST(A_1) \cup FIRST(A_2) \cup ... \cup FIRST(A_i) \cup FIRST(A_{i+1})$$

Ukoliko je simbol A terminal, onda je FIRST(A) = {A}.

Znači to su sve bile definicije, sada da vidimo kako se operativno računaju first skupovi za neku gramatiku:

1. Inicijalno, FIRST skupovi svih neterminala su prazni.
2. Razmatramo redom svaki neterminal <X>
3. Razmatramo redom sve smene sa levom stranom <X>.
4. Neka je jedna takva smena $<X> \rightarrow \alpha$. Tada računamo:
   a. $FIRST(<X>) = FIRST(<X>) \cup FIRST(\alpha)$

   Drugim rečima, ažuriramo skup first (<X>), tako što u njega dodajemo first skup njegove desne strane.
5. Tačke 2-4 algoritma ponavljamo sve dok ima promene u FIRST skupovima.

Sada kada smo definisali poništivost i first skupove neterminala, konačno možemo objasniti i definisati računanje follow skupova gramatičkih neterminala:

1. Skup FOLLOW startnog neterminala sadrži marker kraja ulaza ⊣. U suštini, ovo pravilo, ako dodajemo onu proširujuću nultu smenu, ne mora posebno da se istakne, ali to je nekakav specijalni slučaj.
2. Za određivanje FOLLOW skupa neterminala <X> posmatramo sve gramatičke smene na čijoj se desnoj strani nalazi <X>. Neka je $<Y> \rightarrow \alpha <X> \beta$ jedna od tih smena. Tada:

   $FOLLOW(<X>) \supset FIRST(\beta)$, ako β nije poništivo, ili
   $FOLLOW(<X>) \supset FIRST(\beta) \cup FOLLOW(<Y>)$, ako je β poništivo.

Drugim rečima, za neki proizvoljan neterminal <X> posmatramo sve gramatičke smene na čijoj se desnoj strani nalazi <X> (za razliku od računanja FIRST skupova gde smo <X> posmatrali na levim stranama smena). Recimo da je neka smena koji ima na levoj strani neko <Y>, a na desnoj strani <X> negde u sredini, znači α ispred <X> i β iza njega mogu biti proizvoljni nizovi od nula ili više gramatičkih simbola. U zavisnosti od toga da li je β (kraj desne strane iza <X>) poništivo ili nije, imamo dve varijante formule. Po prvoj varijanti, u skup follow(<X>) uključujemo skup first(β), ukoliko β nije poništivo. Ako je β poništivo, potrebno je "gledati" i iza β, tako da u follow(<X>) uključujemo first(β), ali dodajemo i elemente iz follow skupa leve strane smene. Zašto je to tako? Ako je β poništilvo (ili ne postoji uopšte, recimo da smena završava sa <X>), ako bismo utvrdili da, u nekom izvođenju iz startnog neterminala, možemo iza <Y> dobiti neki terminal t odmah iza leve strane smene <Y>, prostom zamenom umesto <Y> stavimo desnu stranu (α<X> ako nema β), onda bi se znači t našlo odmah iza <X>. Tako da znači sve što može da se nađe iza <Y>, automatski može da se nađe i iza <X> ako je β poništivo. Zbog toga sve što je u skupu follow(<Y>) uključujemo i u skup follow(<X>).

Uz prethodna pravila računanja follow skupova, nekad je potrebno proći, kao i ovim prethodnim algoritmima, više puta kroz gramatiku iterativno, dok se follow skupovi ne stabilizuju.

1. Inicijalno se usvoji da su svi FOLLOW skupovi prazni.
2. Usvojene vrednosti se uvrste u formule, čime se dobijaju nove vrednosti FOLLOW skupova.

Korak 2. ponavlja se sve dok postoji promena bar u jednom skupu. Krajnje vrednosti predstavljaju konačno rešenje.

Primer

Na primeru ćemo ilustrovati sve ovo što smo do sada objašnjavali. Zadata je gramatika sa startnim simbolom <A>, koja ima sedam smena.

1. <A> → a <B> <C>
2. <A> → b <B>
3. <A> → ε
4. <B> → a <B> b
5. <B> → ε
6. <C> → b <C>
7. <C> → c

Vidimo da ima i prazne smene, treću i petu. Ima neterminale <A>, <B> i <C> i terminalne simbole malo a,b i c. Želimo da odredimo follow skupove za svaki neterminal ove gramatike.

Odredimo prvo poništive neterminale u datoj gramatici. Inicijalno,

P = { <A>, <B>}

na osnovu 3. i 5. smene. Ovo je ujedno i konačan skup, jer neterminal <C> ne poseduje smene sa poništivom desnom stranom.

Na osnovu 1. i 2. smene je:

FIRST(<A>)={a, b}

Na osnovu 4. smene je:

FIRST(<B>)={a}

Konačno, na osnovu 6. i 7. smene:

FIRST(<C>)={b, c}

Odredimo sada FOLLOW skupove neterminala:

- Startni neterminal ne pojavljuje se na desnim stranama smena, pa je na osnovu pravila 1. :

  FOLLOW(<A> = {⊣ }
- Razmotrimo neterminal <B> koji se pojavljuje na desnim stranama 1. , 2. i 4. smene. Na osnovu prve smene i pravila 2. gde je α = a, β = <C>:

  FOLLOW(<B>) $\supset$ FIRST(<C>)

  Na osnovu 2. smene i istog pravila, pri čemu je α = a, β = ε, a leva strana smene je <A>:

  FOLLOW(<B>) $\supset$ FIRST(ε) $\cup$ FOLLOW(<A>), odnosno

  FOLLOW(<B>) $\supset$ FOLLOW(<A>)

  pošto je FIRST(ε) = $\varnothing$.

  Na osnovu 4. smene i pravila 2. , pri čemu je α = a, β = b:

  FOLLOW(<B>) $\supset$ {b}

  jer je FIRST(b) = {b}. S obzirom da nema više pojava neterminala <B> u drugim smenama, važi da je:

  FOLLOW(<B>) = FIRST(<C>) $\cup$ FOLLOW(<A>) $\cup$ {b}

  S obzirom da su svi članovi na desnoj strani poznati možemo odrediti konačnu vrednost:

  FOLLOW(<B>) = {b, c, ⊣ }
- Razmotrimo sada neterminal <C> pojavljuje se u 1. i 6. smeni, pa na osnovu pravila 2. za obe smene:

  1. FOLLOW(<C>) $\supset$ FIRST(ε) $\cup$ FOLLOW(<C>)
  2. FOLLOW(<C>) $\supset$ FIRST(ε) $\cup$ FOLLOW(<A>)

  Izraz 1. ne doprinosi konačnom rešenju pa ga možemo ukloniti, čime se dobija:

  FOLLOW(<C>) = $\varnothing$ $\cup$ FOLLOW(<A>) = {⊣ }

Konačno rešenje je, prema tome:

FOLLOW(<A> = {⊣ }
FOLLOW(<B>) = {b, c, ⊣ }
FOLLOW(<C>) = {⊣ }

Konstrukcija SLR(1) parsera

Sada kada znamo da izračunamo follow skupove gramatičkih neterminala, konstrukcija SLR(1) parsera je jedan prost dodatak na konstrukciju LR(0) parsera. Karakteristični automat SLR(1) parsera, odnosno njegova potisna tabela, identična je kao kod LR(0) parsera. Onim postupkom od ranije konstruišemo potisnu tabelu. Jedina razlika u algoritmima konstrukcije LR(0) i SLR(1) parsera, svodi se na konstrukciju kontrolne SLR(1) tabele, potisne su identične.

Vrste kontrolne tabele SLR(1) parsera odgovaraju pojedinim stanjima karakterističnog automata, a kolone pojedinim ulaznim simbolima gramatike proširene 0. smenom. Pravila popunjavanja SLR(1) kontrolne tabele na osnovu karakterističnog LR(0) automata i FOLLOW skupova gramatičkih neterminala su sledeća (razmatra se ulaz u vrsti V, kojoj odgovara skup konfiguracija S i koloni x):

1. Ako S sadrži konfiguraciju oblika <X> $\rightarrow$ α • x β, odnosno tačka se nalazi ispred terminalnog simbola x, ulaz treba popuniti akcijom SHIFT.

2. Ako S sadrži konfiguraciju oblika <Y> → γ •, odnosno tačka se nalazi na kraju desne strane, a važi da je x ∈ FOLLOW(<Y>), tada ulaz treba popuniti akcijom REDUCE(<Y> → γ). Izuzetno, ako se radi o nultoj smeni, akcija je ACCEPT.
3. Vrste koje nisu pokrivene ni 1. ni 2. pravilom treba da sadrže akciju REJECT.

Ukoliko gornja pravila na jednoznačan način određuju akciju za svaki ulaz kontrolne tabele, gramatika pripada klasi SLR(1). Naravno, mogu se desiti oni shift/reduce i reduce/reduce konflikti, na na isti način kao kod LR(0) parsera, što bi značilo da se gramatika ne može deterministički parsirati SLR(1) parserom.

Primer konstrukcije SLR(1) parsera

Evo primera konstrukcije. Ranije smo za ovu gramatiku sa četiri smene, kojoj smo dodali ovu proširujuću nultu smenu, konstruisali LR(0) parser.

0. <S'> → <S> –|
1. <S> → a
2. <S> → (<S> <R>
3. <R> → , <S> <R>
4. <R> →)

Sada određujemo follow skupove gramatike, znači za neterminal <S'>, pošto je on startni naterminal, follow skup je {–|}. Ovde nema poništivih neterminala. Što se tiče follow(<S>), vidimo da se iza <S> pojavljuje –|, u ovoj nultoj smeni i iza <S> se pojavljuje neterminal <R> u drugoj i trećoj smeni. Čime može <R> početi? <R> može da počne zarezom i zatvorenom zagradom na osnovu smena 3. i 4. Tako da follow(<S>) čine marker kraja i elementi first(<R>). Vidimo da je follow(<R>) isto što i follow(<S>), što opet sledi iz druge ili treće smene. Pošto je ovde <R> na samom kraju druge smene, to je onda ono pravilo da follow(<R>) u sebe uključuje follow leve strane smene, odnosno follow(<S>). Konačni skupovi su:

FOLLOW (<S'>) = { –| }
FOLLOW (<S>) = { ,) –| }
FOLLOW (<R>) = { ,) –| }

U ranijem primeru konstrukcije LR(0) parsera za ovu gramatiku smo dobili karakteristični LR(0) automat, prikazan na slici na strani 89 iterativnom konstrukcijom sa closure0 i goto0 operacijama. Kada se u tabelarnom obliku, koristeći skraćene oznake stanja prikaže ovaj automat dobija se potisna tabela na strani 90.

Kontrolna tabela SLR(1) parsera treba da ima vrste označene svim stanjima karakterističnog automata. Kolone treba da odgovaraju gramatičkim terminalnim simbolima. Vidimo da su u ovoj gramatici u pitanju marker kraja ulazne sekvence, a, otvorena zagrada, zarez, zatvorena zagrada. Znači to su kolone.

Sada popunjavamo tabelu kako gornja pravila diktiraju, gledajući karakteristični automat i dodatno konsultujući follow skupove. Akcija shift se stavlja gde god je tačka ispred odgovarajućeg ulaznog simbola. Recimo u startnom stanju, shift će biti u kolonama a i otvorena zagrada. Takođe smo rekli da se akcije redukcije neke smene stavljaju kada je u odgovarajućem stanju item te smene sa tačkom na samom kraju.

	(	,	)	a	⊣
∇	SHIFT			SHIFT	
<S>$_0$					SHIFT
⊣$_0$					ACCEPT
a$_1$		REDUCE(1)	REDUCE(1)		REDUCE(1)
($_2$	SHIFT			SHIFT	
<S>$_2$		SHIFT	SHIFT		
<R>$_2$		REDUCE(2)	REDUCE(2)		REDUCE(2)
,$_3$	SHIFT			SHIFT	
<S>$_3$		SHIFT	SHIFT		
<R>$_3$		REDUCE(3)	REDUCE(3)		REDUCE(3)
)$_4$		REDUCE(4)	REDUCE(4)		REDUCE(4)

U startnom stanju ∇ i stanju <S>$_0$ nema itema sa tačkom na samom kraju smene, tako da nema ni reduce akcija. Međutim u stanju –|$_0$, vidimo da postoji tačka na samom kraju, to je ona nulta proširujuća smena. Tako da se u toj vrsti radi reduce(0), odnosno accept. Nadalje, za stanje a$_1$ i smenu <S> → a, to je smena koja ima <S> na levoj strani, a videli smo da follow(<S>) čine zarez, zatvorena zagrada i marker kraja ulaza. Znači u stanju a$_1$ potrebno je u kolonama zarez, zatvorena zagrada, marker kraja staviti reduce(1). Slično je i za ostale smene.

Da biste na kraju konstrukcije proverili da li vaš parser ima smisla neke elementarne provere su: da li parser sadrži akciju accept i da li imate bar po jednu reduce akciju za svaku od smena. Pri tome po smenama još dodatno na bazi levih strana proverite kolone gde ste stavili reduce akcije, jer one ne smeju biti u kolonama koje nisu u follow skupovima odgovarajućih neterminala. Ovde vidimo da su neki ulazi kontrolne tabele ostali prazni To su oni ulazi koji potpadaju pod pravilo za sneštanje reject akcija. Po konvenciji ne pišemo eksplicitno reject, nego ostavljamo prazan ulaz da bi bila preglednija tabela. jer je najveći broj ulaza često upravo reject.

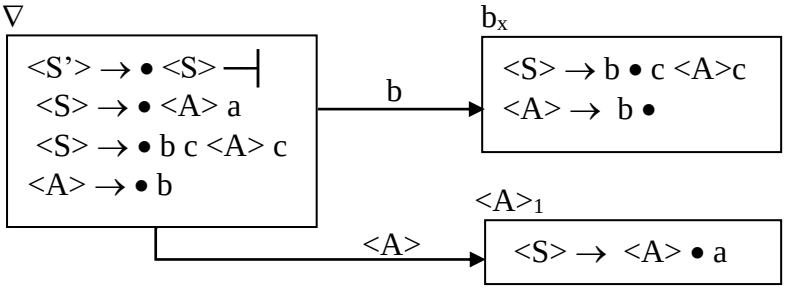
Konstrukcija LALR(1) parsera

Time smo izložili SLR(1) metod parsiranja. Sada ćemo objasniti jedan generalniji metod parsiranja LALR(1). Počećemo sa uvodnim motivacionim primerom da vidimo šta su nedostaci SLR(1) metoda, koji je inače jači od LR(0), na primer, možemo imati gramatiku sa praznom smenom što u LR(0) metodu nije moguće. Međutim i SLR(1) metod ima svoja ograničenja.

0. <S'>→ <S> ⊣
1. <S> → <A> a
2. <S> → b c <A> c
3. <A> → b

Posmatrajmo ovu relativno jednostavnu gramatiku, koja osim ove proširujuće nulte smene ima dva neterminala <S>, <A> i tri smene. Ovde je gramatika tako konstruisana da se <A> pojavljuje na desnim stranama smena za <S> i to tako da je donekle kontekst različit, u smislu da se u prvoj smeni iza <A> pojavljuje terminal a, a u drugoj smeni se iza <A> pojavljuje terminal c. Možemo da zaključimo da follow skup neterminala <A>, upravo na bazi tih smena gde se pojavio na desnoj strani, sadrži a i c.

Sada posmatrajmo samo jedan manji deo karakterističnog automata, koji odgovara ovoj gramatici. Startno stanje ∇ sadrži item sa tačkom na početku nulte smene i primenom closure0 operacije, potrebno je dodati smene za <S> sa tačkom na početku. Zbog itema prve smene, gde je tačka odmah ispred <A> closure0 dodaje item smene za <A>. Znači ovo je finalni skup itema u startnom stanju.



Sada posmatrajmo izlaznu granu b, ona je rezultat goto0 akcije. Ovde je zanimljiva situacija, imamo dva itema gde je tačka neposredno ispred b. Ispravan način je da se oba ova itema prebace u novo stanje b_x i da se tačka samo premesti iza b.

Stanje b_x je konfliktno stanje, zato što po pravilu za shift, pošto se tačka u itemu <S> → b • c <A>c nalazi neposredno ispred c, u vrsti b_x i koloni c treba da se stavi shift. Međutim, pravilo za redukciju smene kaže, pošto imamo item <A> → b • gde je tačka na samom kraju, da treba redukovati ovu treću smenu, a to znači treba staviti reduce(3) u sve kolone koje su u skupu follow(<A>), a to su a i c. Znači za ulaz tabele u vrsti b_x i koloni c, dolazi do konflikta jer istovremeno bi trebalo staviti i shift akciju i reduce(3) akciju, Reč je o shift/reduce konfliktu. Zaključujemo da je teorijski nemoguće konstruisati SLR/1) parser i da gramatika ne pripada klasi SLR(1).

Postavlja se pitanje da li je neuspeh ove procedure konstrukcije suštinski za ovu gramatiku, ili je to više posledica nekih nedostataka SLR(1) metoda. O čemu se radi? Ako posmatramo šta se dešava kada parser radi i ako on zaista treba da preduzme redukciju, znači reduce(3) akciju, šta to znači? Reduce akcija je skidanje sa steka desne strane smene, to jest onoliko simbola koliko ima desna strana smene, u konkretnom slučaju jedan simbol. Pri radu parsera na vrhu steka bi se našlo stanje b_x. Kada parser skine to stanje, on bi dobio ovo stanje dna steka, jer je odatle sa shift došao do b_x u nekom prethodnom koraku, potiskujući b sa ulaza na stek. Da bi se završila reduce akcija, znači prvo je skinuto b_x sa steka, došlo se do startnog stanja ∇ i sad je potrebno na stek potisnuti stek simbol koji odgovara levoj strani smene <A>.

Potiskivanje <A> u stanju ∇ bi parser vodilo u stanje $<A>_1$ karakterističnog automata u koje se ide iz stanja ∇ po izlaznoj grani <A> i koje ima item sa tačkom iza <A>. Šta možemo uočiti? Redukcijom treće smene mi u stvari nastavljamo prepoznavanje prve smene gramatike <S> → <A> a, jer u startnom stanju to je jedini item koji ima tačku neposredno ispred <A>. Znači to je sigurno, jednoznačno, ovo <A> iz prve smene. Međutim, vidimo da iza njega u smeni stoji a i onda se postavlja pitanje zašto je onda uopšte ovom stanju bilo potrebno da se akcija reduce radi, ako na ulazu dolazi malo c? Odgovori upravo taj da ovde suštinskog konflikta nema i parser bi sasvim korektno radio da se u vrsti b_x i koloni c stavlja isključivo shift.

Akcija reduce(3) nikada ne treba da se preduzima jer u suštini ta akcija je došla u kontrolnu tabelu zato što se ne uzima pravi kontekst parsiranja ovim follow skupovima. Follow skup određuje šta

se sve može naći iza <A> u bilo kom kontekstu, u prepoznavanju bilo koje sekvence, znači prosta unija svega što se u bilo kom kontekstu može naći.

Potrebno je pažljivije sagledati kontekst parsiranja, a za to je potrebno da uđemo unutar ovih stanja parsera, odnosno karakterističnog automata, da skup preciznije definišemo kao skup svih terminala koji se mogu naći na tekućem ulazu u trenutku kada treba raditi redukciju smene u određenom stanju. Znači ako te skupove preciznije definišemo povezujući ih za stanja karakterstičnog automata, onda neke konflikte možemo eliminisati.

Pojam LR(1) konfiguracije

Upravo je to ideja ovih LALR(1) gramatika. Kod njih se follow skupovi "uvlače" u sam automat, u njegova stanja i pridodaju item-ima ili LR(0) konfiguracijama. LR(1) konfiguracija ili LR(1) item predstavlja LR(0) konfiguraciju sa dodatim *predikcionim (engl. lookahead) simbolom*

$$A \rightarrow X_1...X_i \bullet X_{i+1}....X_n, p \qquad p \in V_T \cup \{\varepsilon\}$$

(ε se koristi kao predikcija samo kod konfiguracija smene 0.)

Predikcioni simbol p može biti bilo koji terminalni simboli i, za neki specijalni slučaj, p može biti prazna sekvenca, odnosno nedostatak bilo kog terminalnog simbola i to se samo koristi kod konfiguracija nulte smene. U opštem slučaju p je terminalni simbol.

Predikciona komponenta x konfiguracije $<A> \rightarrow \alpha \bullet \beta$, x označava jedan od mogućih tekućih ulaznih simbola u trenutku rada parsera kada se na vrhu steka kompletira ručka smene $<A> \rightarrow \alpha\beta$.

Gramatika koju ćemo posmatrati u daljem izlaganju je sledeća:

1. $<S> \rightarrow <A> <S>$
2. $<S> \rightarrow \varepsilon$
3. $<A> \rightarrow <B>\ b\ <B>\ c$
4. $<A> \rightarrow <C>\ c\ <B>$
5. $<B> \rightarrow a\ <D>$
6. $<C> \rightarrow a\ <D>$
7. $<D> \rightarrow \varepsilon$

Navedimo za primer četiri LR(1) konfiguracije za datu gramatiku sa istom prvom komponentom:

$<S> \rightarrow \bullet <A> <S>, a$ $\qquad$ $<S> \rightarrow \bullet <A> <S>, b$
$<S> \rightarrow \bullet <A> <S>, c$ $\qquad$ $<S> \rightarrow \bullet <A> <S>, \dashv$

Skup koji sadrži navedene četiri konfiguracije može se kraće obeležiti sa:

$$<S> \rightarrow \bullet <A> <S>, \{a, b, c, \dashv\}$$

pri čemu {a, b, c, ⊣} nazivamo predikcioniom skupom.

Pravilne LR(1) konfiguracije

Sada kada smo uveli pojam LR(1) konfiguracije, nije baš svaka kombinacija smene sa tačkom i predikcione komponente smislena. Zato se uvodi pojam takozvanih pravilnih LR(1) konfiguracija. To su u suštini sve kombinacije koje imaju smisla -- u okviru simbola steka LALR(1) parsera može se pojaviti samo podskup LR(1) konfiguracija koje nazivamo *pravilnim.*

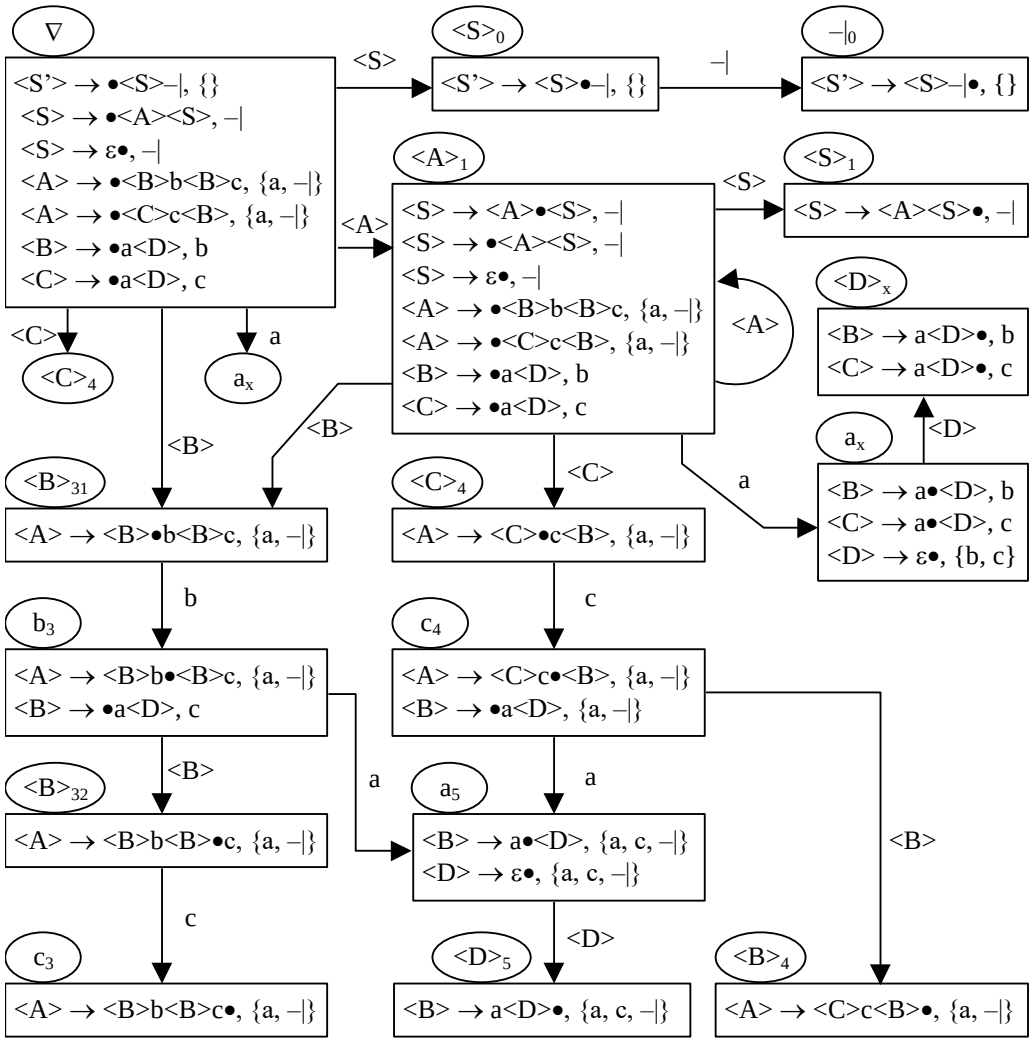
LR(1) konfiguraciju $<X> \rightarrow \alpha\bullet\beta$, t nazivamo pravilnom LR(1) konfiguracijom ako i samo ako je:

1. $t \in$ FOLLOW(<X>), ili
2. reč o smeni 0 i $t=\varepsilon$.

Koje konfiguracije nazivamo pravilnim? Znači samo ako su ispunjeni uslovi da je ova predikciona komponenta iz follow skupa leve strane smene koja je predstavljena ovom prvom komponentom ili, u specijalnom slučaju, ako je reč o nultoj smeni, a predikciona komponenta je Epsilon. Druge moguće kombinacije ove prve komponente i druge komponente ne smatramo pravilnim, odnosno ne smatramo smislenim, odnosno neće se ni pojavljivati ni obrađivati.

Karakteristični LALR(1) automat

Sada ćemo objasniti konstrukciju LALR(1) parsera. Međutim, pre prelaska na sve detalje bolje je otići malo u napred i pogledati šta je nama cilj, šta želimo da dobijemo. Kao jedan među korak iz koga dobijamo finalne tabele parsera (potisnu i kontrolnu) sada dobijamo takozvani karakteristični LALR(1) automat. Na slici je prikazan primer karakterističnog LALR(1) automata za posmatranu gramatiku sa strane 100:



To je pandan karakterističnog LR(0) automata kod ovih prethodnih metoda konstrukcije. Odnos LR(0) i LALR(1) automata za istu gramatiku je takav da su ta dva automata, kada posmatramo njihove grafove, izomorfni, odnosno potpuno je isti broj stanja i veze su potpuno iste. Jedina razlika između LR(0) i LALR(1) automata je u ovim unutrašnjim sadržajima stanja. Sadržaji stanja su takođe povezani. Kada bismo iz LALR(1) izbrisali ove predikcione komponente koje se pojavljuju iza LR(0) itema koji predstavljaju smene sa tačkom, dobili bismo LR(0) karakteristični

automat. Znači LALR(1) automat se od LR(0) razlikuje samo po tome što su dodate ove predikcione komponente u stanja. I upravo te predikcione komponente se kasnije koriste umesto follow skupova za pravila popunjavanje kontrolne tabele, odnosno gde treba smestiti reduce akcije.

Sada ćemo videti kako se konstruiše karakteristični LALR(1) automat, pa ćemo potom navesti pravila kako se na bazi njega konstruiše finalni parser. Da bismo izračunali karakteristični LALR(1) automat:

- Prvo ćemo konstruisati karakteristični LR(0) automat već opisanim metodom preko closure0 i goto0 operacija.
- Potom je potrebno odrediti predikcione (lookahead) skupove. To su oni skupovi koji se u finalnom karakterističnom LALR(1) automatu pojavljuju iza LR(0) item-a.

Postoji više metoda za izračunavanje predikcionih skupova. Mi ćemo objasniti metod prosleđivanja predikcionih simbola, odnosno lookahead propagation metod. U prvoj fazi potrebno je odrediti takozvane spontane predikcione simbole, odnosno spontaneous lookahead skupove (SLA). Oni se određuju za sve konfiguracije zatvaranja u svakom od stanja LR(0) karakterističnog automata Šta su konfiguracije zatvaranja? To su oni item-i koji imaju tačku na samom početku. Prvo za njih izračunavamo predikcije u vidu SLA skupova.

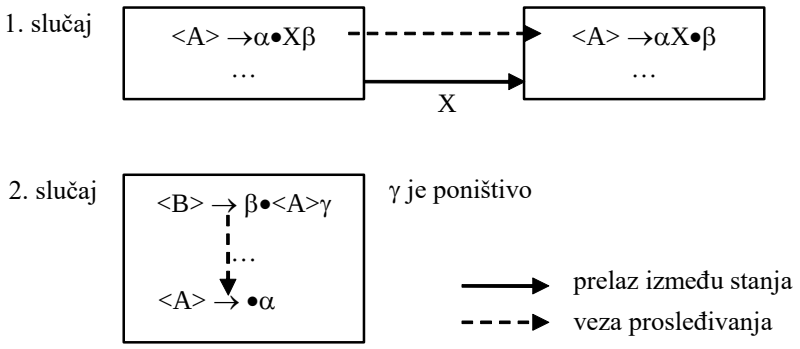
U cilju računanja predikcionih skupova prvo se odrede takozvani skupovi spontanih predikcionih simbola (skr. SLA) za sve konfiguracije zatvaranja u svakom od stanja LR(0) detektora ručki.

SLA skup za proizvoljnu LR(0) konfiguraciju zatvaranja $<A> \rightarrow \bullet\ \alpha$ u stanju s računa se tako što se nađu sve konfiguracije oblika $<B> \rightarrow \beta \bullet <A>\ \gamma$ u stanju s i u skup SLA($<A> \rightarrow \bullet\alpha$) uključe svi terminali $t \in FIRST(\gamma)$.

Između LR(0) konfiguracija u stanjima LR(0) automata definišu se *veze prosleđivanja* prema sledećim pravilima:

1. Za svaku konfiguraciju oblika $<A> \rightarrow\alpha\bullet X\beta$ u posmatranom stanju S, gde su α i β proizvoljne sekvence od nula ili više gramatičkih simbola, a X proizvoljan gramatički simbol, uvodi se veza prosleđivanja ka konfiguraciji oblika $<A> \rightarrow\alpha X\bullet\beta$ u stanju u koje se prelazi iz S po X.
2. Za svaku konfiguraciju oblika $<B> \rightarrow\beta\bullet<A>\gamma$ u posmatranom stanju S, gde je β proizvoljna sekvenca od nula ili više gramatičkih simbola, γ je poništiva sekvenca od nula ili više gramatičkih simbola, a <B> je proizvoljan neterminal, uvode se veze prosleđivanja ka svim konfiguracijama oblika $<A> \rightarrow\bullet\alpha$ u posmatranom stanju S.

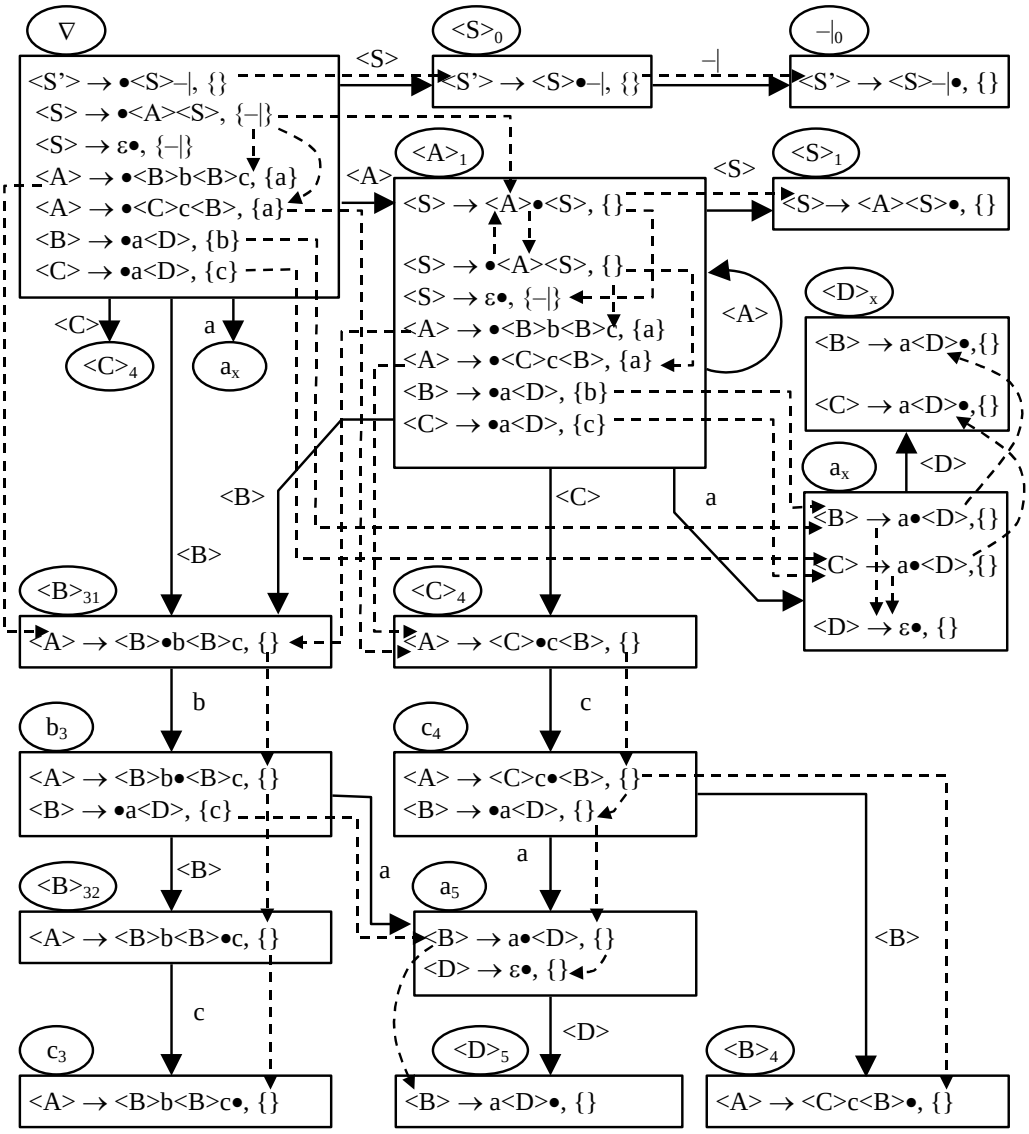
Pravila određivanja veza prosleđivanja ilustrovana su na sledećoj slici.



Kada odredimo sve spontane predikcione skupove i kada definišemo sve veze prosledjivanja, finalni predikcioni (LA) skupovi računaju se sledećim iterativnim algoritmom:

- Inicijalno su LA skupovi konfiguracija zatvaranja u svakom od stanja jednaki SLA skupovima odgovarajućih konfiguracija, dok su LA skupovi ostalih konfiguracija prazni.
- Predikcioni simboli prosleđuju se među LA skupovima konfiguracija koje su povezane vezama prosleđivanja sve dok postoji promena u LA skupovima.

Evo na jednom konkretnom primeru da vidimo kako to sve izgleda. Za gramatiku sa strane 100 koju posmatramo, koja je proširena smenom 0., prikazaćemo stanje karakterističnog automata kada je konstruisan LR(0) automat, kada su određeni SLA skupovi za konfiguracije zatvaranja (sa tačkom na samom početku) i kada su ustanovljene isprekidanim linijama veze prosleđivanje.



Veze prosleđivanja prikazane su isprekidanom linijim, a prelazi među stranjima punom. Svaki item koji ima tačku na samom početku, treba da ima definisan skup SLA. Jedini izuzetak je baš ovaj item nulte smene, čiji je SLA skup prazan. Za iteme drugih smena koji imaju tačku na samom

početku, to su takozvani item-i zatvaranja, njihovi SLA skupovi se određuju tako što moramo naći item u istom stanju, koji je uzrokovao operacijom zatvaranja da se doda posmatrani item.

Jedan konkretan primer u stanju ∇ je item nulte smene <S'> → •<S>–|, koja je imala tačku ispred <S> i zatvaranjem je dodat item <S> → •<A><S>. FIRST skup onoga što se nalazi iza <S> u nultoj smeni se ubacuje kao SLA skup za item <S> → •<A><S>, znači to je marker kraja ulaza.

Drugi primer, u stanju ∇ za item <B> → •a<D>, pošto je leva strana <B> on je upravo ubačen kao posledica zatvaranja item-a <A> → •<B>b<B>c koji ima tačku ispred <B>. Sada treba u SLA skup itema <B> → •a<D> ubaciti sve iz skupa FIRST(b<B>c), to jest svega što sledi <B> u onom drugim itemu. To se svodi na b.

Za sve druge item-e gde tačka nije na samom početku smene, inicijalni lookahead skupovi su prazni. Što se tiče veza prosleđivanja, one su dodate prema pravilima 1 i 2 na strani 102.

Primer primene pravila 1 je veza prosleđivanja od itema <S'> → •<S>–| u stanju ∇ do itema <S'> → <S>•–| u stanju $<S>_0$. Dakle ovo pravilo povezuje iteme koji su logički povezani i u goto0 operaciji konstrukcije automata. Primer primene pravila 2 su veze od itema <S> → •<A><S> u stanju ∇ ka itemima <A> → •<B>b<B>c i <A> → •<C>c<B> u istom stanju. Primetiti da je neterminal <S> poništiv, što je uslov da bi se ove veze uspostavile (da ostatak iza •<A> u izvorišnom itemu veze bude poništiv). Upravo iz tog razloga ne postoji veza prosleđivanja u stanju ∇ između itema <S'> → •<S>–| kao izvorišnog i itema <S> → •<A><S> i <S> → ε• kao odredišnih, jer nastavak iza •<S> nije poništiv, reč je o terminalnom simbolu.

Sada kada imamo SLA skupove i veze prosleđivanja, ono što je potrebno to je da terminale iz skupova, koristeći veze prosledjivanja, prosleđujemo (ubacujemo njihove kopije) u odredišne skupove. Recimo marker kraja iz SLA skupa itema <S> → •<A><S> u stanju ide ∇ u predikcioni skup itema <A> → •<C>c<B>, a zatim sledeći izlaznu vezu prosleđivanja ide i u skup itema <A> → <C>•c<B> u stanju $<C>_4$, a odatle u još šest drugih skupova sledeći redom ostale veze.

Prosleđivnje se sprovodi sve dok ne dobijemo stabilnu situaciju. Kada nema više promena, dobijamo finalne predikcione skupove i karakteristični LALR(1) automat.

Konstrukcija tabela LALR(1) parsera

Ostaje da vidimo kako se konstruišu potisna i kontrola tabela parsera. Ovaj automat ima sve informacije za obe tabele. Potisne tabela LALR(1) parsera je identična potisnoj tabeli LR(0), odnosno to je ovaj isti karaktestični automat predstavljen u tabelarnom obliku, pošto smo rekli da je da je reč o potpuno identičnom grafu kada se samo zanemari unutrašnji sadržaj stanja, dobijaju se iste potisne tabele i kod LR(0) i kod LALR(1) parsera.

Glavna razlika je u kontrolnoj tabeli. Popunjavanje kontrolne LALR(1) tabele je vrlo slično popunjavanju SLR(1) kontrolne tabele, sa jedinom razlikom što se kod SLR(1) uzimaju follow skupovi leve strane smena, a ovde se uzimaju predikcioni skupovi u samim stanjima.

Vrste kontrolne tabele LALR(1) parsera odgovaraju pojedinim stanjima karakterističnog automata, a kolone pojedinim ulaznim simbolima gramatike proširene 0. smenom. Pravila popunjavanja LALR(1) kontrolne tabele na osnovu karakterističnog LALR(1) automata su sledeća (razmatra se ulaz u vrsti V, kojoj odgovara skup konfiguracija S i koloni x):

1. Ako S sadrži konfiguraciju oblika <X> → α • x β, t odnosno tačka se nalazi ispred terminalnog simbola x, ulaz treba popuniti akcijom SHIFT.

2. Ako S sadrži konfiguraciju oblika <Y> → γ •, x odnosno tačka se nalazi na kraju desne strane, a predikciona komponenta je x, tada ulaz treba popuniti akcijom REDUCE(<Y> → γ). Izuzetno, ako se radi o nultoj smeni, akcija je ACCEPT.
3. Vrste koje nisu pokrivene ni 1. ni 2. pravilom treba da sadrže akciju REJECT.

Za naš tekući primer, finalni parser u tabelarnom obliku prikazan je na slici koja sledi. U finalnom parseru urađena je optimizacija da se u finalnom koraku rada end marker ne potiskuje na stek, već se akcija ACCEPT radi kada je na vrhu steka $<S>_0$, a na ulazu end marker, dakle jedan korak ranije i izostavljeno je stanje $-|_0$ koje se u optimizovanom parseru ne koristi .

	<S>	<A>	<B>	<C>	<D>	a	b	c	a	b	c	–|
∇	$<S>_0$	$<A>_1$	$<B>_{31}$	$<C>_4$		a_x			SHIFT			RED(2)
$<S>_0$												ACC
$<A>_1$	$<S>_1$	$<A>_1$	$<B>_{31}$	$<C>_4$		a_x			SHIFT			RED(2)
$<B>_{31}$							b_3			SHIFT		
$<C>_4$								c_4			SHIFT	
a_x					$<D>_x$					RED(7)	RED(7)	
$<S>_1$												RED(1)
b_3			$<B>_{32}$			a_5			SHIFT			
c_4			$<B>_4$			a_5			SHIFT			
$<D>_x$										RED(5)	RED(6)	
$<B>_{33}$								c_3			SHIFT	
a_5					$<D>_5$				RED(7)		RED(7)	RED(7)
$<B>_4$									RED(4)			RED(4)
c_3									RED(3)			RED(3)
$<D>_5$									RED(5)		RED(5)	RED(5)

potisna tabela kontrolna tabela

Ako svaki uloz u kontrolnoj tabeli, popunjenoj navedenim pravilima, sadrži jedinstvenu akciju, onda zadata gramatika pripada klase LALR(1) gramatika i konstrukcija parsera uspešno je okončana. U suprotnom slučaju, teorijski ne možemo konstruisati LALR(1) parser. Znači i ovde mogu da se dese i kod drugih LR parsera konflikti tipa shift/reduce i reduce/reduce.

Diskusija

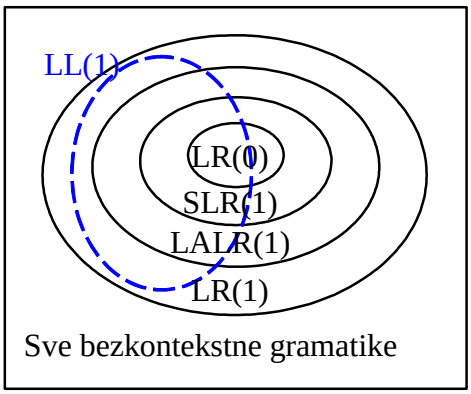
Da završimo ovo izlaganje o konstrukciji shift-reduce parsera jedinim informativnim odeljkom koji rekapitulira ono što smo do sada radili i upoređuje pojedine kategorije parsera. Ovo je Venov dijagram i kvadrat predstavlja skup svih bezkontekstnih gramatika. Nije moguće za proizvoljnu bezkontekstnu gramatiku konstruisati parser ovih kategorija koje smo mi objasnili, dakle svaka od kategorija parsera koje proučavamo je pravilan podskup unutar skupa svih gramatika.

Najveći podskup pripada LR(1) klasi gramatika. LR(1) klasa gramatika odgovara parsirima čiju konstrukciju nismo objašnjavali, a koji su vrlo slični LALR(1) parserima. Razlika je u karakterističnom automatu čija stanja takođe sadrže LR(1) iteme kao i kod LALR(1), ali identitet

stanja određuju kompletni LR(1) itemi sa predikcinim komponentama, ne samo LR(0) itemi. Zato LR(1) parser za istu gramatiku ima mnogo više stanja nego LALR(1), a kada bi se iz stanja uklonile predikcione komponente i zatim spojila ista stanja, dobio bi se automat identičan LR(0). Na primer, jedna gramatika za Javu ima 273 terminala i neterminala i 516 smena. LALR(1) parser za nju ima manje od 800 stanja, a LR(1) parser za istu gramatiku više od 6000 stanja.

LR(1) tehniku nismo objašnjavali zato što konkretni alati koji se konstruišu u konstrukciji kompajlera ne konstruišu LR(1) parsere nego konstruišu LALR(1) parsere. LR(1) parser jeste još generalniji, može da obradi još veći skup gramatika od LALR(1) i to je najgeneralniji deterministički parser sa jednim predikcionim simbolom koji se može napraviti. Međutim u praksi razlike između LALR(1) i LR(1) nisu toliko izražene, jer problemi sa gramatikama koje LALR(1) ne može da obradi ili su vezani za višeznačnost, ili za činjenicu da je potreban lookahead veći od 1. Takođe, analiza gramatičkih konflikata pregledom karakterističnog automata ("debagovanje gramatike") znatno je otežano kada je automat veći.

Vidimo da su LALR(1) parseri pravilan podskup LR(1) parsera, SLR(1) parseri su pravilan poskup LALR(1) parsera. LR(0) kao najuža klasa, su pravilan podskup SLR(1) parsera. LALR(1) imaju praktičan značaj.



U nastavku ćemo obraditi još jednu klasu deterministčkih parsera koja je na ovom dijagramu naznačena isprekidanom linijom i koja kao što vidimo preseca skoro sve ove metode koje smo mi dosad objašnjaali, znači ima neki presek s njima, međutim pravilan je podskup ovih LR(1) parsera. To je top-down klasa parsera LL(1), poseban metod koji ćemo u nastavku obrađivati u sklopu parsiranja od vrha ka dnu. I taj metod ima praktičan značaj u nekim situacijama.

Oporavak od grešaka kod parsera od dna ka vrhu

Prevođenje sa obično ne zaustavlja na prvoj grešci u ulaznom programu, nego prevodilac pokušava da, pri nailasku na grešku u programu, detektuje grešku odnosno izda odgovarajuću izlaznu poruku koja će poslužiti programeru da ispravi program i zatim da izvrši oporavak od greške, odnosno da određenim akcijama učini nastavak sintaksno-semantičke analize programa mogućim, sa idejom da se prođe kroz ceo programski kod i da se detektuju i druge potencijalne greške.

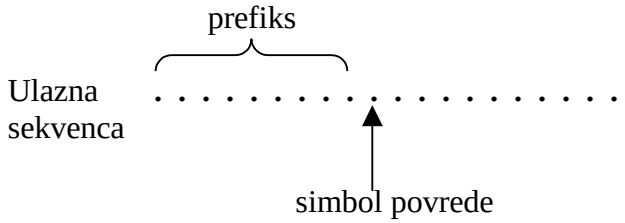
Pri tome se tipično ne vrši generisanje izlaznog programa sem ako je u pitanju upozorenje tipa warning. Obično ima nekoliko nivoa poruka: informaciona poruka, upozorenje, greška i fatalna greška. Na informacione poruke i upozorenja u principu može da se izvrši kompletna sintaksno-semantička analiza programa, međutim na greške i fatalne greške obično se ne generiše izlazni program, već samo izveštaj o greškama. Obične greške i fatalne se razlikuju po tome što parser

pokušava da izvrši oporavak i završi procesiranje programa kod obične greške. Kod fatalne greške obično se rad kompajlera prekida istog trenutka.

Videli smo da su parseri zasnovani na potisnim automatima koji imaju stek. Prema tome, akcije za oporavak moraju da obuhvate, u opštem slučaju, modifikaciju parserskog steka i preostalog ulaznog niza. Malo ambicioznija varijanta je da kompajler korisnikov ulaz sa greškom popravi u validan program, semantički ispravan uz generisanje prevedenog koda. To je više teorijski cilj. Praktični kompajleri retko imaju taj nivo odziva na greške programera, zato što kompajler ne može garantovano da pogodi šta je programer hteo da iskaže, odnosno da je semantika programa garantovano dobra.

Da pomenemo i šta ne valja da kompajler uradi, odnosno šta nije prihvatljivo ponašanje za jedan profesionalno razvijen kompajler. Pri nailasku na grešku u korisničkom programu ne bi smeo da pređe preko postojeće greške bez ikakve detekcije. Zatim, u principu, kompajler ne bi smeo da prekine rad sa greškom u izvršavanju (da izgleda kao da je to neki bug u kompajleru). Takođe, s obzirom da se radi o potisnim automatima i steku, loš oporavak može da izazove zaglavljivanje kompajlerskog procesiranja u mrtvoj petlji gde se stalno na ekranu pojavljuje ista greška. Bilo je primera nekih komercijalnih programa u vreme kada je C++ bio novi jezik i kada su se prvi C++ kompajleri pravili tako što su se adaptirali i proširivali postojeći C kompajleri pa su se firme trudile da brzo izađu na tržište, pojavili su se C++ kompajleri koji su imali jako nizak kvalitet oporavka od greške.

Neke osobine parserskog algoritma mogu da olakšaju ili otežaju oporavak od greške. Definisaćemo svojstvo ispravnog prefiksa kod parsera, ali pre toga moramo da definišemo šta je to simbol povrede ulazne sekvence. Simbol povrede je onaj simbol ulaznog niza na kome parser otkriva grešku. To znači da je parser već isprocesirao neki deo ulazne sekvence, ovaj prefiks bez ikakvih problema i onda u trenutku kada je tekući ulazni simbol ovaj simbol povrede, parser u svojoj kontrolnoj ili potisnoj tabeli otkriva grešku. Ako parser ima svojstvo ispravnog prefiksa, onda za sve neispravne ulazne sekvence, niz simbola pre simbola povrede predstavlja prefiks nekog prihvatljivog ulaznog niza.



Šta je neposredna posledica? Kod takvih parsera možemo modifikacijom preostalog ulaza učiniti ceo ulazni niz prihvatljivim. Svojstvo ispravnog prefiksa je dosta zgodno svojstvo koje nam govori da pri nailasku na grešku parser ne mora da radi nikakav backtracking (vraćanje u ulazu unazad) nego samo modifikacijom preostalog ulaza i svog stanja (steka) da načini ceo ulazni niz prihvatljivim, što je u stvari cilj oporavka od greške. Ono što je dobro to je da, za sve metode kojima mi u kursu konstruišemo parsere, rezultujući parseri imaju to svojstvo.

Što se tiče izveštavanja o grešci, od koristi su trenutni vrh steka i tekući ulazni simbol u trenutku greške i naravno obično se vodi evidencija o poziciji u ulaznom programu na kojoj je nastupila greška. Pošto leksički analizator vrši tokenizaciju programa, on mora da vodi evidenciju i o tome. Poruka o grešci treba da se isprocesira od strane pisca kompajlera, da se ne svede samo na to koji

su tekući token i stanje, nego da se na bazi tokena i stanja proba formulisati takva poruka da programera uputi na mesto i uzrok zašto je greška nastala.

Posle prijave greške vrši se oporavak parsera od greške. Ono što je ponekad neizbežno prilikom oporavka od greške, to je da može da se desi da prva greška indukuje još dodatne prijave grešaka koje u stvari u programu ne postoje, nego su posledica prve greške. U principu, kvalitet oporavka ceni se i po tome koliko je tih lažnih prijava, da ih bude što manje. Jedan najprostiji primer: ako bi recimo programer zaboravio da deklariše promenljivu X, pa je posle upotrebi 10 puta u nekoj funkciji, loše bi bilo da se 10 puta prijavljuje ista prijava za to X. Rešenje je da kompajler prijavi prvo korišćenje Xa bez definicije, a zatim u sklopu oporavka ubaci privremenu definiciju X u tabelu simbola, tako da, kada sledeći put kad naiđe na X, ne radi istu prijavu.

Oporavak od greške kod SHIFT-REDUCE parsera

U uvodu smo izneli opšta razmatranja, nevezana za parserske tehnike. Sada ćemo konkretizovati priču tako što ćemo razmotriti shift-reduce parsere od dna ka vrhu i kako kod njih može konkretno da se ugradi oporavak od greške. Izložićemo osnovne i najčešće primenjivane varijante takozvanog paničnog oporavaka (engleski termin je panic mode recovery). a potom ćemo razmotriti kako je oporavak koncipiran kod yacc/cup familije alata. Postoje i dosta kompleksniji metodi oporavka i popravka greške. Na primer knjiga autora Fisher, Leblanc dosta razmatra razne kompleksnije metode oporavaka.

Kod "paničnog oporavka" osnovna ideja je da, pri pojavi greške, parser pokušava da izađe iz jezičkog konstrukta u kome je nastala greška tražeći siguran ulazni simbol kao što je terminator naredbe ili počtna ključna reč sledeće naredbe i tome slično, od koda može da nastavi normalno pasiranje.

Za najjednostavniji algoritam paničnog oporavka od greške, bitno je uz zadatu gramatiku definisati skup takozvanih sigurnih simbola, a u njih obično ulaze terminalni simboli koji završavaju razne programske konstrukcije, kao što su na primer tačka zarez završava naredbu, end kod nekih jezika, zatim marker kraja ulaznog niza i tome slično.

Osnovni algoritam paničnog oporavka je formulisan kroz ova tri koraka:

1. U prvom koraku kada parser detektuje grešku, sa ulaza se redom uklanjaju simboli dok se ne naiđe na neki od sigurnih simbola.
2. U drugom koraku, u tom trenutku se sa steka parsera skidaju simboli, sve dok se ne naiđe na neki koji može da potisne sigurni simbol na ulazu na stek. Kako se to može videti? Iz odgovarajuće vrste potisne tabele.
3. U trećem koraku, parsiranje se posle toga nastavlja na uobičajeni način.

Prednost ovog osnovnog algoritma je što ne zahteva neki poseban tretman, predstavlja automatizovan i generalan odziv parsera pri pojavi greške. Jedino što se od konstruktora kompajlera traži, to je da definiše ovaj skup sigurnih simbola, znači vrlo malo posla u odnosu na osnovni posao konstrukcije parsera. Koje su mane osnovnog algoritma? To nije neki jako kvalitetan način oporavka. Ovaj algoritam dobro radi kod nestrukturiranih jezika kao što su recimo asemblerski jezici, međutim kod jezika koji imaju ugneždene strukture, recimo dovoljno je da imate izraz sa ugneždenim zagradama ili blok strukturirane jezike gde, na primer, vitičaste zagrade otvaraju funkciju, pa onda unutar nje opet imamo neke vitičaste zagrade koji neki blok naredbi ograđuju. Mi tada možemo definisati, na primer, da je terminator zatvorena vitičasta zagrada,

međutim šta se dešava u toku oporavka? Nailazi se na ugneždenu strukturu koja ima otvorenu i zatvorenu vitičastu zagradu. Oporavak bi trebalo celu tu ugneždenu strukturu da proguta da bi došao do one primarne vitičaste zagrade na koju treba se sinhronizuje, međutim on će pogrešno progutati samo početak te ugneždene strukture, znači otvorenu vitičastu zagradu. Onda će se pogrešno sinhronizovati na prvu zatvorenu vitičastu zagradu. Trebalo je u stvari da i nju preskoči i da onda ide na spoljnu zatvorenu vitičastu zagradu. Znači ovaj algoritam nema tu mogućnost da vodi evidenciju ugneždenih struktura koje imaju sigurne simbole.

Zbog toga je smišljena poboljšana varijanta osnovnog algoritma parničnog oporavka, gde se pored skupa sigurnih simbola definiše takozvani skup simbola zaglavlja, odnosno header simbola. Simboli zaglavlja su komplementarni sigurnim simbolima, znači ako je siguran simbol zatvorena vitičasta zagrada, simbol zaglavlja je otvorena vitičasta zagrada; ako je siguran simbol end onda je simbol zaglavlja begin, i tako dalje.

Poboljšani algoritam paničnog oporavka glasi:

(1) Kada parser detektuje grešku, sa ulaza se redom uklanjaju simboli dok se ne naiđe na neki od sigurnih simbola ili simbola zaglavlja.

(2a) Ako je naišao siguran simbol, tada se sa steka parsera skidaju simboli sve dok se ne naiđe na neki koji može da potisne sigurni simbol na ulazu na stek.

(2b) Ako je naišao simbol zaglavlja, tada se on potiskuje na stek i posebno markira.

(3) Parsiranje se posle toga nastavlja na uobičajen način. Kada se redukcijom neke smene ukloni markirani simbol sa steka, prelazi se ponovo na korak (1).

Koraci ovog algoritma su slični originalnim koracima 1,2 i 3, uz male modifikacije. U prvom koraku, kada parser detektuje grešku, sa ulaza se redom uklanjaju simboli, dok se ne naiđe na neke od sigurnih simbola ili simbola zaglavlja. Ne preskače se nijedna od te dve kategorije simbola. U koraku 2a, ako je naišao siguran simbol, tada se sa steka parsera skidaju simboli, sve dok se ne naiđe na neki koji može da potisne taj sigurni simbol na ulazu na stek, znači taj korak je isti kao onaj originalni korak 2. Međutim, korak 2b, ako je naišao simbol zaglavlja, tada se on potiskuje na stek i posebno markira. Drugim rečima, on na taj način čini nekakvo lažno privremeno dno steka i to je povezano sa korakom tri. Znači parsiranje se posle toga nastavlja na uobičajen način. Kada se redukcijom neke smene ukloni markirani simbol sa steka, prelazi se ponovo na korak jedan. Korak 2b i korak 3 služe da pri oporavku detektuju i uparuju ugneždene kontrolne strukture, koje počinju simbolom zaglavlja, a završavaju nekim sigurnim simbolom.

U pratećoj Zbirci zadataka možete pronaći detaljne primere sa trejsom rada parsera pri oporavku od greške po ovim algoritmima koji su ovde izloženi.

Oporavak od greške kod yacc/cup alata

Izloženi algoritmi su bili sasvim generalni i mogu se uvek primeniti kod shift-reduce parsera uz vrlo mali dodatan napor da se definišu sigurni i simboli zaglavlja. Međutim, generatori parsera iz familije yacc, odnosno cup, imaju malo drugačije realizovan oporavak od greške koji zahteva za nijansu više truda od konstruktora kompajlera, ali sam oporavak može da se kvalitetnije uradi.

Kako se parseri generisani alatima yacc odnosno cup, koji su inače shift-reduce parseri, ponašaju pri pojavi greške u ulazu parsera? Što se tiče prijave greške, pri nastanku sintaksne greške iz generisanog parsera poziva se funkcije yyerror() kojoj se prosleđuje string sa opisom greške. Po

povratku iz te funkcije, parser može izvršiti oporavak od greške ako u gramatici postoje odgovarajuća pravila. U suprotnom, parserska funkcija završava rad i vraća nenultu vrednost. Ova varijanta koju sam opisao važi za C-ovski yacc parser. Međutim, kod cup generisanog Java parsera, u slučaju greške baca se izuzetak, znači ta parserska funkcija su uvek stavlja u try-catch blok, da pokrije situaciju kada se završava rad parsera bacanjem izuzetka, ako nema ugrađenog nikakvog oporavka od greške, ili ako oporavak od greške nije uspeo. Međutim ideja jeste da se ugradi kvalitetan oporavak od greške koji bi sprečio ovaj neuspešan završetak parsiranja.

Da bismo kod yacc/cup parsera ugradili oporavak od greške, moramo modifikovati gramatiku. Za to postoji poseban gramatički simbol error. error je specijalan gramatički simbol koji se pojavljuje samo na desnim stranama smena i te smene u kojima se pojavi simbol error potpuno se ignorišu kad je ulaz sintaksno ispravan. Kada nema nikakve greške u gramatici, onda kao da te smene sa error ne postoje. Međutim, one se mogu aktivirati ako se pojavi greška u ulazu. Osnovna ideja je da error upari neki pogrešan deo ulaza i da omogući da se pogrešan deo ulazne sekvence proguta i da parser dalje sinhronizovano nastavi parsiranje ostatka ulazne sekvence.

```
stmt : expr ';'
                | while_stmt ';'
                | if_stmt ';'
                | ...
                |  error ';'
          ;
if_stmt : if '(' expr ')' stmt
```

Na ovom primeru ćemo sada detaljnije videti šta sve to znači. Ova mala gramatika je napisana u sintaksnom obliku kakav zahteva yacc odnosno cup alat, a to je da se leve i desne strane smena razdvoje sa dve tačke. Stmt, expr, while_stmt, if_stmt su neterminalni simboli. Prvih nekoliko smena se odnosi na smene za iskaz stmt. Jedna vrsta iskaza je izraz (expr) i iskaz se završava terminalnim simbolom tačka zarez. Po konvenciji se uvek grupišu sve smene se istom levom stranom i piše se samo uspravna crta umesto leve strane.

Druga vrsta iskaza stmt je neki while iskaz predstavljen neterminalom while_stmt koji treba da se dalje elaborira u gramatici nije dato šta je to, isto se završava tačkom zarezom. Treća vrsta iskaza je if_stmt, završava se tačkom zarezom. Za if_stms ovde imamo posebnu smenu znači terminal if pa u zagradama izraz expr pa onda iskaz stmt. Tri tačke označavaju da tu može biti još nekoliko smena za druge vrste statement-a odnosno iskaza.

Konačno, ova zanimljiva smena koja je označena sa **bold**, iskaz je error tačka zarez. To je ta dodatna smena koja, ako je ulaz sintaksno ispravan kao da ne postoji, znači ne menja opis sintaksno ispravnih ulaza. Međutim, ako u bilo kojoj vrsti iskaza dođe do neke greške, ova smena se aktivira da se parser oporavi do momenta kada na ulazu ima tačku zarez, kada će prepoznati ovu smenu sa error i posle toga da parser izađe iz režima oporavka i nastavi normalno pasiranje.

Evo sada detaljno kako funkcioniše algoritam oporavka od greške kod ovih yacc/cup parsera:

- Po detektovanju greške, parser skida sa steka stanja dok ne dođe do stanja koje ima neposredan oporavak od greške (konfiguraciju $X \rightarrow \ldots \bullet error \ldots$).
- Tada parser počinje “preskakanje” ulaznih tokena do trenutka kada dođe neki koji očekuje.

- Tada nastavlja parsiranje ali bez izvršavanja semantičkih akcija. Ako uspešno procesira bar tri tokena sa ulaza, ulaz se resetuje na stanje kada je prestalo preskakanje tokena i parsiranje se ponavlja normalno, ovaj put izvršavajući sve akcije.
- Ako parser nije uspeo uspešno da isprocesira tri tokena, parser odbacuje sledeći token sa ulaza i pokušava da nastavi parsiranje.

Ako se u pokušajima oporavka dođe do kraja ulaza ili isprazni parserski stek (na primer nije bilo odgovarajućeg stanja koje ima oporavak) parser neuspešno završava rad.

Algoritam dosta intuitivno radi i zadatak je programera odnosno konstruktora kompajlera isključivo da se fokusira na to gde će ubaciti ove error smene. Da bi se potpuno razumeo algoritam preporuka je da se praktično isproba na nekoj gramatici. Znači, krenete od originalne gramatike, ubacite error smene, namerno postavite test primere sa neispravnim ulazima i posmatrate šta će u stvari parser da uradi. To je jedini način da se u potpunosti shvati ponašanje parsera. Međutim, ovde postoje neke preporuke u propratnom uputstvu ovih alata, to su prosto neka iskustvena pravila gde i kako dodavati ove error smene. Ono što na neki način emulira onaj panic mode recovery je da se iskoriste simboli koji terminišu neke konstrukcije, pa da se napiše za tu konstrukciju neka error smena koja na kraju ima simbol za koji recimo znamo da završava sve statement-e i tako dalje.

```
stmnt: error ';'  /* on error, skip until ';' is read */
```

Takođe, ovo odgovara opet onom naprednijem algoritmu, kod konstrukcija koje imaju neki početak i neki kraj i u principu mogu i da se ugneždavaju, može da se recimo doda ovakva smena:

```
primary:  '(' expr ')'
        | '(' error ')'
         . . .
        ;
```

Ako imate izraz u zagradama, onda dodate posebnu smenu koja umesto izraza ima error unutar zagrada, što znači da ako unutar ovog izraza nastane greška, ideja je da ćemo se na ovim zagradama sinhronizovati. Ovi primeri su relativno jednostavni i ovde smo dodavali po jednom neterminalu samo jednu error smenu. Ništa nas ne ograničava da na jednu ispravnu gramatičku smenu dodamo i više ovih error smera. U ovom drugom primeru to donekle zavisi od toga šta očekujete da korisnik može da pogreši i šta smatrate da bi bio relativno kvalitetan oporavak u nekim relativno čestim situacijama greške. Recimo, ako smatrate da korisnik često može da pogrešno izostavi zatvorenu zagradu, može da se doda ovde kao treća smena, smena koja na desnoj strani ima otvorenu zagradu, expression i onda umesto zatvorene zagrade piše error. I tome slično, u suštini na jednu ispravnu smenu možete imati nekoliko smena sa ovim error simbolom.

Da li postoji gornji limit dodavanja tih error smena? On nominalno ne postoji, znači neko bi mogao na 10 ispravnih gramatičkih smena da doda 100 ovih error smena. Međutim, postoji drugi faktor koji limitira da se ne pretera sa ovim dodavanjem error smena. Prilično je recimo opasno da imate smenu koja ima samo error na desnoj strani i da error nije ograničen nekim drugim simbolom. Zašto? Ako stavite na neko nezgodno mesto taj error može vam se desiti da za gramatiku koja bez error smena nije imala nikakav parserski konflikt, tipa shift/reduce ili reduce/reduce, može se desiti da se odjednom pojave zbog tog dodavanja error smena konflikti. To u stvari limitira mesta gde

možete staviti i koliko možete staviti tih error smena. Naravno, tu uglavnom radi neka intuicija dosta dobro i prosto treba malo probanjem steći osećaj gde i kako dodavati te smene.

Ovaj algoritam ima dodatnu logiku da se ne prijavljuje sintaksna greška neposredno posle prve prijave, nego tek posle tri ispravno procesirana ulazna tokena ponovo se omogućava prijavljivanje greške, odnosno pozivanje one yyerror() funkcije. To je opet jedan detalj vezan za način rada tih parsera. Ova poslednja stavka je još jedan detalj koji je prisutan samo u yacc c/c++ verziji alata, a nije u cup parserima. Moguće je na neki način resetovati režim oporavka od greške, ako u error smeni stavimo specifičnu semantičku akciju (ovaj kod koji je stavljen na kraj ove smene i uokviren vitičastim zagradama).

```
primary: '(' error ')' { yyerrok; }
```

To je inače sintaksa za ubacivanje semantičkih akcija c-ovskog yacc alata, a kod java cup alata stavljali bi se delimiteri {:: i ::}. Kada parser prepozna smenu preduzima neku akciju. Kada u c-ovskoj varijanti parsera stavite posebnu reč yyerrok u akciju error smene, to znači da želite da resetujete taj režim oporavka od grešaka, kažete parseru da smatra kao da je već prošao tri uspešna tokena i da odmah nastavi sa normalnim radom. To može u nekim specifičnim situacijama da malo promeni ponašanje parsera, ali nije od nekog suštinskog značaja. Ono što je najvažnije zapamtiti, to je da i ove dodatne error smene mogu da normalno sadrže akcije, koje prilikom oopravka treba da učinte konzistentnim stanje semantičke obrade (koriguju po potrebi strukture kao što je sintaksno stablo ili tabela simbola).

Parsiranje od vrha ka dnu

U ovom poglavlju ćemo prvo obraditi osnovne koncepte parsera od vrha ka dnu (engl. top down), zatim ćemo videti konkretno kako se konstruišu parseri od vrha ka dnu, definisaćemo odgovarajuću klasu gramatika za koje ta konstrukcija uspeva, to su LL(1) gramatike, videćemo i bitan pojam selekcionog skupa smene, koji pomaže parseru u svakom koraku rada, da odluči koju smenu će sledeću izabrati. Takođe ćemo razmotriti šta ako gramatika nije LL(1), znači moguće ekvivalentne transformacije gramatika, da se dobije pogodna gramatika za parsiranje od vrha ka dnu i, konačno, videćemo kako se top down prepoznavači proširuju u procesore da obavljaju sintaksno-upravljamo prevođenje od vrha ka dnu, na osnovu atributivno-translacione gramatike. Takođe ćemo videti kako se vrši oporavak od greške kod ove vrste parsera.

Koncept parsiranja od vrha ka dnu je da parser na ulazu ima kompletnu ulaznu sekvencu i da je zatim procesira sa leva na desno, znak po znak, i pri tome parser rekonstruiše celokupno levo izvođenje ulazne sekvence počev od startnog neterminala. Ako se posmatra stablo izvođenja otkriva ga od korena ka listovima.

Rad parsera se obavlja u diskretnim koracima i u svakom koraku rada, parser pokušava da upari tekući simbol iz sentencijalne forme koju trenutno razmatra, sa delom preostalog ulaza. Ako je tekući simbol u stanju parsera terminal, znači isti taj terminal mora da bude i tekući ulazni simbol, inače dolazi do greške i prekida se parsiranje uz odbijanje ulazne sekvence. Ako je sve u redu, tada se u tekućem stanju pređe na sledeći gramatički simbol i takođe na ulazu se konzumira tekući ulazni simbol i prelazi na sledeći.

Ako je tekući simbol u stanju parsera neterminal, onda je potrebno izabrati jednu od smena kojom se taj neterminal zamenjuje i na taj način dobija nova sentencija forma u stanju parsera za razmatranje, a tekući gramatički simbol postaje prvi simbol te izabrane smene. Videćemo da nam pojam selekcionog skupa smene daje rešenje koju smenu izabrati, jer tekući ulazni simbol mora da se nalazi u selekcionom skupu izabrane smene. Opisana procedura se ponavlja sve dok se ne rekonstruiše celokupno levo izvođenje, ako je naravno ulaz ispravan.

Sada ćemo videti, na primeru sledeće gramatike od četiri smene, gde još dodatnu pomoć imamo da svaka smena počinje drugim terminalom, tako da je vrlo lako videti na osnovu početka smene šta se očekuje na ulazu. Krećemo od inicijalnog stanja parsera, to je startni neterminal, a sa druge strane, na ulazu, na primer, imamo ulaznu sekvencu a c d d. Potrebno je rekonstruisati korake između u izvođenju iz <S> do finalnog ulaznog niza.

1. <S> → a <A> <B>
2. <S> → b
3. <A> → c <B>
4. <B> → d

Parser počinje rad razmatranjem startnog neterminala. To mu je, u njegovom stanju, tekući neterminalni simbol i naravno prvi simbol ulazne sekvence a je tekući ulaz.

```
<S> => ? => acdd
 ↑          ↑ tekući ulaz
tekući neterminal
```

U obzir dolaze, kada gledamo gramatiku, smene za <S>. To su prva i druga, i na bazi tekućeg ulaza, parser se odlučuje za primenu prve smene. U opštem slučaju, tekući ulazni simbol mora da se nalazi u selekcionom skupu izabrane smene. Pojam selekcionog skupa ćemo malo kasnije definisati. Kada jednom izaberemo smenu, u stanju parsera <S> se menja desnom stranom izabrane smene, znači dobijamo a<A><B> i to je sad novo stanje parsera. Ujedno smo dobili i ovaj jedan međukorak izvođenja.

<S> =>$_{lm}$ a <A><B>=> ? => acdd

tekući simbol tekući ulaz

Znači sada smo umesto polaznog <S> dobili sentencijalnu formu a<A><B>, to je novo stanje parsera. S druge strane, ulaz je nepromenjen, tekući ulazni simbol je prvi simbol a. Sada važi pravilo da, ako je tekući simbol u stanju terminal, on mora da se poklopi sa ulaznim simbolom i u tom slučaju parser pomera pointer sa tekućeg na sledeći simbol u stanju parsera, znači ovo <A>, a u tekućem ulazu takođe prelazimo na sledeći ulazni simbol, to je c.

<S> => $_{lm}$ a <A><B>=> ? => acdd

tekući simbol tekući ulaz

Sada je potrebno na bazi tih informacija da je tekući simbol stanja <A> i tekući ulaz c, izabrati smenu. Jedino smena za <A> je treća smena, znači tu čak i nema nekog izbora, menjamo <A> desnom stranom treće smene, čime rekonstruišemo sledeći korak ovog krajnje levog izvođenja, pošto se u stanju uvek bira za zamenu prvi neterminal sa leve strane.

<S> => $_{lm}$ a <A><B>=> $_{lm}$ a c <B> <B> => ? => acdd

tekući simbol tekući ulaz

Neterminal <A> smo zamenili sa c<B> u ovoj sentencijalnoj formi i dobili smo sada ovakvo tekuće stanje parsera gde je početak treće smene c tekući simbol u stanju parsera, a s druge strane tekući ulaz nije promenjen, ostalo je c. Sada uparujemo naravno c u stanju i u tekućem ulazu i pomeramo pointere za po jedan simbol na desno.

<S> => $_{lm}$ a <A><B>=> $_{lm}$ a c <B> <B> => ? => acdd

tekući simbol tekući ulaz

Sada je tekući simbol u stanju <B>, tekući ulaz d. Treba primeniti smenu za <B> i to je četvrta smena. Primenom četvrte smene menjamo prvo <B> sa d i posle toga znači opet još pomeramo pointere, znači uparujemo d u stanju i na ulazu. Pomeramo pointere na drugo <B> u stanju i na poslednje d na ulazu.

<S> => $_{lm}$ a <A><B>=> $_{lm}$ a c <B> <B> => $_{lm}$ a c d <B> => ? => acdd

tekući simbol tekući ulaz

Opet se vrši zamena <B> primenom četvrta smene sa d u stanju. S druge strane, tekući ulaz ostaje nepromenjen. Konačno sada kada uparimo terminal u stanju i terminal na ulazu i pomerimo pointere na desno, došli smo do kraja ulazne sekvence. Vidimo da smo u tekućem stanju dobili upravo ulaznu sekvencu. Znači potpuno smo rekonstruisali izvođenje te ulazne sekvence od startnog neterminala. Parser je na uspešan način završio rad.

$$<S> \Rightarrow_{lm} a <A><B> \Rightarrow_{lm} a\ c <B> <B> \Rightarrow_{lm} a\ c\ d <B> \Rightarrow_{lm} acdd$$

U nastavku ćemo razmotriti kako konstruisati parser koji radi po ovom konceptu. Postoje različite metode konstrukcije top-down parsera, prvi je tabelarno u vidu potisnog automata kojim se nećemo baviti, nego ćemo pokazati metod rekurzivnog spusta. To je jedan zanimljiv metod koji samo zahteva da parser bude implementiran u višem programskom jeziku koji ima rekurziju. Kod potisnog automata znamo da imamo stek koji određuje rad automata, a u metodu rekurzivnog spusta run-time stek programskog jezika na neki način preuzima tu ulogu parserskog steka.

Metod rekurzivnog spusta

Metod ćemo objasniti na konkretnom primeru. Ovo je jedna gramatika od osam smena za opis aritmetičkih izraza, koja ima malo specifičan oblik, ali to kako interpretiramo ovo gramatiku naravno nije važno za sam metod konstrukcije.

1. <E> → <T> <E_list>
2. <E_List> → + <T> <E_List>
3. <E_list> → ε
4. <T> → <P> <T_list>
5. <T_List> → * <P> <T_List>
6. <T_list> → ε
7. <P> → (<E>)
8. <P> → I

Ipak ćemo napomenuti, da ova gramatika opisuje infkisne aritmetičke izraze koji imaju operacije sabiranja i množenja, a operandi su reprezentovani terminalnim simbolom I. Preko ovih uvedenih neterminala, množenje ima već prioritet od sabiranja. Obe operacije su levo asocijativne, obavljaju se s leva na desno. Dodatno, prioritet može da se menja zagradama. Ova gramatika deluje neuporedivo komplikovanije od gramatika kojima smo mogli da opišemo aritmetički izraz, koja bi recimo glasila:

<E> → <E> + <E> | <E> * <E> | (<E>) | I

Ili, ako eliminišemo dvosmislenost i ispoštujemo prioritete i asocijativnost operacija:

<E> → <E> + <T> | <T>

<T> → <T> * <P> | <P>

<P> → (<E>) | I

Zašto je u polaznoj gramatici to sve razdvojeno u više smena? To je upravo posledica transformacija gramatika o kojima ćemo kasnije pričati, znači da smo napisali tako prostiju gramatiku za izraze sa kojima smo se sretali kod bottom-up parsera i koje su često imale i neku direktnu levu rekurziju. da na levoj i na početku desne strane imamo isti neteminal, znači top-down parseri ne mogu da obrade takve gramatike i zbog toga je potrebno izvršiti transformaciju takve gramatike i kao rezultat te transformacije može da se dobije ova malo manje pregledna gramatika, ali koje se lepo ponaša u top-down parsiranju.

Bez obzira na to, videćemo neke formalne uslove koje gramatika mora da zadovolji da bi mogla da se parsira, ali za sada da se vratimo na primer i da pretpostavimo da su ti uslovi zadovoljeni za ovu gramatiku i da vidimo kako se može konstruisati parser na bazi rekurzivnog spusta.

Programsko rešenje čini jedan glavni program i nekoliko potprograma (procedura). Po jedan potprogram imamo za svaki gramatički neterminal. U ovom konkretnom slučaju, pošto u gramatici imamo <E>, <E_list>, <T>, <T_list> i <P>, isto toliko treba da imamo potprograma i oni će imati samo prefiks proc: PROCE, PROCE_list i tako dalje. Potprogram za određeni neterminal ima ulogu da prepozna primenu neke od smena sa datim neterminalom na levoj strani i upari neterminal sa ulazom u skladu sa prepoznatom smenom. Recimo za sedmu i osmu smena, koje se odnose na neterminal <P> u datoj gramatici, one su jednostavnije pa da krenemo objašnjenje od njih:

7. <P> → (<E>)
8. <P> → I

Vidimo da 7. smena ima na desnoj strani <E> u zagradama, a 8. smena ima terminal I. Zagrade su naravno terminalni simboli, <E> je neterminal. Procedura P treba da pokrije deo procesiranja parsera za prepoznavanje 7. ili 8. smene na osnovu tekućeg ulaza.

```
procedure PROCP:
        case IN of
                'I'   : goto P8;
                '('   : goto P7;
                '+', '*', ')', '┤': REJECT;
        end case;
  P8:   kod za prepoznavanje        8. smene
  P7:   kod za prepoznavanje        7. smene
end procedure;
```

Pretpostavićemo da u programskom rešenju postoji globalna promenljiva IN koja predstavlja tekući ulazni simbol, znači ono što je u tom trenutku na ulazu parsera. Smenu biramo upravo na osnovu tekućeg ulaznog simbola. Videli smo da 8. smena ima samo I na desnoj strani, znači ako je I tekući ulazni simbol, onda prepoznajemo 8. smenu, ako je otvorena zagrada tekući ulazni simbol, onda prepoznajemo 7. smenu, koja ima otvorenu zagradu, <E> i zatvorenu zagradu. U svim ostalim slučajevima ulaznog simbola, znači ovo je default grana, ali ovde je eksplicitno navedeno šta još mogu biti ulazni simboli, to su ostali terminali u gramatici plus, puta, zatvorena zagrada i end marker, parser završava rad uz odbijanje ulaza.

Kod unutar svake case grane nam služi da skočimo na deo koda procedure za prepoznavanje određene smene. Ovde je eksplicitno dato Covsko rešenje sa goto da bi se naglasila sa ovim labelama koja smena se prepoznaje. Taj kod može da se ugradi i direktno u svaku granu, ne mora se koristi goto.

Sada da vidimo kako se pojedine smene prepoznaju. Nama je u ovom primeru bilo lako da odredimo, kada je tekući ulaz određeni znak, koju smenu prepoznajemo, pošto tačno svaka smena počinje određenim terminalom. Međutim, u opštem slučaju, smena može da počne neterminalom, ili da bude prazna smena i tada je malo komplikovanija situacija, ali mi ćemo nju razrešiti kroz selekcione skupove smena o kojima ćemo kasnije pričati.

Ovde je opis dela koda koji odgovara prepoznavanju određene smene. Kod treba tako da formiramo što razmatramo desnu stranu smene, redom znak po znak, s leva na desno. Ako je određeni znak neterminal, onda treba pozvati odgovarajući potprogram. Ako je određeni znak terminal, onda je potrebno ispitati da li je tekući ulaz u tom trenutku jednak terminalu. Ako jeste, pozvati funkciju koja napreduje ulaz za jedan znak, odnosno prelazi na sledeći znak na ulazu, nazovimo je nextchar(). A ako ne dođe do poklapanja terminala koga očekujemo u tom trenutku u smeni i tekućeg ulaznog terminala, onda se preduzima akcija reject kojim se završava rad, uz odbijanje ulazne sekvence.

```
P7:         // kod za prepoznavanje smene <P> → ( <E> )
  IN = NEXTCHAR();
        call PROCE;
        if ( IN = ')' )
                then    IN = NEXTCHAR();
                                return;
                else    REJECT;
        end if
```

Analiza gornjeg koda nije teška kada poznajemo navedena pravila. 7. smena je na desnoj strani imala (<E>). Prvo je potrebno upariti ovu otvorenu zagradu sa tekućim ulazom, međutim to smo već uradili u case komandi koja nas je dovela do ovog slučaja za 7. smenu, tako da ne moramo da ponovo ispitujemo da li je tekući ulaz jednak otvorenoj zagradi, nego samo stavljamo prelazak na sledeći ulazni znak i to opet ažurira globalnu promenljivu IN koju vide svi ovi potprogrami. Time smo znači obradili otvorenu zagradu. Potom na desnoj strani ide <E>. Tu je potrebno ugraditi poziv odgovarajuće procedure PROCE i zatim potrebno je upariti zatvorenu zagradu u 7. smeni sa tekućim ulazom. Znači, da li je tekući ulaz u tom trenutku kada se vratimo iz procedure PROCE zatvorena zagrada, ako jeste, onda prelazimo na sledeći znak na ulazu i završavamo procesiranje, vraćamo se iz procedure PROCP jer smo potpuno uparili ovu smenu. U suprotnom slučaju izvršava se reject, koje treba da implementira završetak parsiranja uz odbijanje sekvence.

Videćemo kasnije i druge delove programa za druge smene. Šta treba da uradi glavni program? On mora da inicijalizuje promenljivu IN da ukaže na prvi znak na ulazu, da pozove potprogram za startni neterminal, čime inicira uparivanje cele ulazne sekvence. U suštini, kada pozivamo potprogram za neki neterminal, on će upariti deo preostalog ulaza sa onim delom ulazne sekvence koliko taj neterminal reprezentuje. Što se tiče startnog neterminala, on treba da reprezentuje celu ulaznu sekvencu ako je ona ispravna. Tako da, kada se vratimo iz procedure za startni neterminal, potrebno je ispitati da li je isprocesiran ceo ulaz, tako što ćemo ispitati da li je tekući ulazni simbol jednak markeru kraja ulazne sekvence. Ako jeste, onda je uspešan kraj rada pasiranja, accept. Ako nije, reč je o neuspešnom kraju rada pasiranja, reject.

U nastavku je dato kompletno rešenje za posmatranu gramatiku i za sve druge procedure. Mi smo razmatrali i proceduru PROCP koja je pokrivala 7. i 8. smenu. Videli smo već kod za 7. smenu. Za 8. smenu je kod još jednostavniji, zato što je na desnoj strani bio samo terminal I koji je u case-u već ispitan da se poklapa sa ulaznom, tako da samo prelazimo na ulazu na sledeći znak i vraćamo se iz procedure PROCP. I tako dalje, za sve druge procedure po ovim opštim pravilima je napisan kod. Vidimo da su procedure uzajamno rekurzivne, znači iz jedne procedure se poziva druga, onda se iz druge poziva treća, a treća opet može da pozove prvu.

```
Glavni program:
        IN = NEXTCHAR();
        call PROCE;
        if (IN = '┤')
            then  ACCEPT;
            else  REJECT;
        end if;
end;

procedure PROCE:
        case IN of
            'I', '(': goto P1;
            '+','*',')','┤':REJECT;
        end case;
P1:     call PROCT;
       call PROCE_LIST;
        return;
end procedure;

procedure PROCP:
        case IN of
            'I'   : goto P8;
            '('   : goto P7;
            '+','*',')','┤':REJECT;
        end case;
P8:     IN = NEXTCHAR();
        return;
P7:     IN = NEXTCHAR();
        call PROCE;
        if ( IN = ')')
            then IN=NEXTCHAR();
                    return;
            else  REJECT;
        end if;
end procedure;

procedure PROCT:
    case IN of
        'I', '(':  goto P4;
        '+','*',')','┤': REJECT;
    end case;
P4:call PROCP;
       call PROCT_LIST;
        return;
end procedure;

procedure PROCE_LIST:
    case IN of
        '+' : goto P2;
        ')', '┤'  : goto P3;
        'I', '(', '*'   : REJECT;
    end case;
P2:     IN = NEXTCHAR();
        call PROCT;
        call PROCE_LIST;
        return;;
P3:     return;
end procedure;

procedure PROCT_LIST:
    case IN of
        '*'    : goto P5;
        '+', ')', '┤'   : goto P6;
        'I', '('  : REJECT;
    end case;
P5:     IN = NEXTCHAR();
        call PROCP;
        call PROCT_LIST;
        return;
P6:     return;
end procedure;
```

Accept ili Reject u realnoj implementaciji, šta bi bili? Accept je kraj rada sa prihvatanjem ulaza, to je neki povratak iz parsera, a reject bi možda bio prijava neke greške i isto kraj rada parsera.

Reject u principu ne može biti samo prost potprogram, zašto? Recimo ako bi reject bio prost potprogram, greška u ulazu može se desiti u bilo kom trenutku rada parsera. Kada je recimo parser otišao duboko u rekurzije, jedna procedura pozvala drugu, druga treću i tako dalje, znači napunjen je onaj run-time stek i onda se pozove reject i sada, reject prijavi grešku i uradi return. Taj return bi se samo vratio u tu proceduru u kojoj je nastala greška, ali parser bi onda nastavio rad, što u principu ne želimo, ako nije urađen oporavak od greške. Mi želimo da potpuno završimo rad, a parser bi pogrešno nastavio da dalje procesira sekvencu kao da nije bilo greške.

Mora se dakle obezbediti mehanizam takozvanog odmotavanja steka, da reject odmah iskoči na nivo glavnog programa (procedure main) pri pojavi greške, čak i ako se reject pozove iz ovih rekurzivnih potprograma. Kako se taj mehanizam obezbeđuje? To zavisi od jezika do jezika. Kod jezika koji imaju bacanje izuzetka, ideja je da reject po prijavi greške baci izuzetak i da se taj izuzetak hvata recimo u glavnom programu. Jezici koji nemaju bacanje izuzetaka, kao što je C, imaju obično neki drugi mehanizam koji je adekvatan za ovu situaciju. U Cu to su bibliotečke funkcije setjmp i longjmp. Funkcijom setjmp postavlja se odredište skoka i ona bi bila pozvana negde u glavnom programu pre cele obrade. A longjmp je bi bilo pozivano na kraju reject, i ovaj poziv bi uzrokovao odmotavanje steka i dolazak u glavni program na mesto određeno sa setjmp. Tako da o tom detalju treba voditi računa.

Oporavak od grešaka u metodu rekurzivnog spusta

Dodatno praktično razmatranje odnosi se na oporavak od grešaka. Realan parser mora da uzme u obzir da ulaz može da bude pogrešan, da znači korisnik može da pogreši, kao što smo već pominjali kod parsera od dna ka vrhu. Želimo da parser ne prekine potpuno rad na prvoj sintaksnoj grešci, nego, posle detekcije i prijave greške, da se pokuša oporavak parsera. Oporavak znači da parser nastavi rad da isprocesira celu ulaznu sekvencu kao da nije bilo greške. I ova vrsta parsera konstruisana po principu rekurzivnog spusta, poseduje ranije pomenuto svojstvo ispravnog prefiksa, da u trenutku detekcije greške, parser ne mora da se vraća unazad na ulazu, nego nastavlja rad modifikacijom preostalog ulaza i modifikacijom svoga stanja.

Kako izgleda implementacija osnovnog algoritma paničnog oporavka u varijanti top-down parsera? Dodaćemo svakoj od rekurzivnih procedura za neterminale dodatni parametar pod nazivom termset, koji je u stvari nekakav skup terminalnih simbola. U normalnom radu parsera taj pametar ne igra nikakvu ulogu. Međutim u slučaju pojave greške. koristimo taj termset skup terminala. U termset-u će se naći terminali, odnosno ulazni simboli koji bi mogli biti ispravan tekući ulaz posle povratka iz odgovarajuće procedure. Ako se unutar procedure PROCA(termset) detektuje greška ideja je da na ulazu "gutamo" simbole, sve dok ne naiđe neki simbol iz termset-a. Kada naiđe neki simbol iz termset-a onda prosto izvršimo povratak iz procedure PROCA. Naravno, to se sve radi posle prijave greške.

Evo kako bi na konkretnom primeru izgledao jedan rekurzivni parser sa oporavkom od greške. Uzećemo vrlo jednostavnu gramatiku koja ima dve smene za <S> i jednu smenu za <E>.

1. <S> $\rightarrow$ [E]
2. <S> $\rightarrow$ (E)
3. <E> $\rightarrow$ a

Recimo ako razmatramo proceduru E, ona će prepoznati treću smenu i njen kod će biti vrlo jednostavan, a njen sadržaj njenog termset-a, znači tog ulazni parametra koji je skup terminalnih simbola, zavisiće od mesta poziva E. Ako zovemo proceduru E iz koda procedure S pri prepoznavanju prve smene, u termset ćemo ubaciti ']', a ako zovemo proceduru E iz koda za prepoznavanje druge smene, u termset ćemo ubaciti ')'. Dodatno, potrebno je ubaciti i ono što sledi <S>. Pošto je <S> startni simbol, znači njega sledi uvek u svakom kontekstu end marker. Tako da će u kontekstu prve smene termset će biti {']', end_marker}, a u kontekstu druge smene biće {')', end_marker}. Zašto ubacujemo end marker? Ako je korisnikova greška da je izostavio jednu ili više ili sve zatvorene zagrade, oporavak će na kraju naleteti na end marker i na taj način će biti završen povratak iz svih procedura.

Evo kompletnog primera za ovu malu gramatiku. Ovde je glavni program, iz njega se poziva procedura S. U njenom termset-u je simbolom $ reprezentovan end marker. Imamo proceduru oporavka od greške koja prima dva parametra, jedno je parametar koji određuje poruku o grešci, a drugo je termset koji je prosleđen odgovarajućoj proceduri, jer obradu greške procedura delegira ovoj proceduri recovery. Znači njeno je da prijavi grešku, tako što ispisuje da je bilo očekivano ovo što je prosleđeno kao prvi parametar, ali je nađeno ono što je trenutno tekući ulazni simbol i potom je realizovano gutanje ulaza do god se ne pojavi neki simbol iz termset-a i onda se kontrola vraća iz recovery. Poziv procedure recover je ugrađen na svako od mesta gde bi inače stajalo reject.

```
procedure S(termset)
  case IN of
  '[' : IN = NEXTCHAR();
        call E(termset U ']');
        if IN <> ']'
           call RECOVERY(']', termset);
        endif;
  '(':  IN = NEXTCHAR();
        call E(termset U '}');
        if IN <> ']'
           call RECOVERY('}', termset);
        endif;
   default: call RECOVERY('([',termset);
   end case;
end procedure;

procedure E(termset)
  if IN == 'a'
  then IN=NEXTCHAR();
  else call RECOVERY('a', termset);
  endif
end procedure;
```

```
procedure RECOVERY( exp, termset)
   call PRINT( 'ERROR Expected', exp,
                  'but found', IN);
   while IN not in termset do
     IN=NEXTCHAR();
   end;
end procedure;

Glavni program:
  IN=NEXTCHAR();
  call S('$');
  if IN <> '$'
     call RECOVERY('$', '$');
  endif;
end;
```

Kada u kontekstu prve smene zovemo proceduru E, onda na ovaj termset od procedure S koji je inače $, dodajemo, operacijom unija, pošto je to skup, zatvorenu uglastu zagradu. Tako da E pozivamo sa tim termset-om $ i uglasta zagrada. Sada, ako pri povratku iz E ne bi bila zatvorena zagrada na ulazu, znači to bi bila sada greška u S, mi bismo zvali recovery ali samo sa termset-om od S, odnosno parser bi sve "pojeo" do kraja ulazne sekvence. Međutim, u proceduri E, ako a ne bi bilo na ulazu u trenutku kada pozivamo proceduru E, onda bismo zvali recovery sa njenim termset-om, koji bi, kao što smo rekli, ako je to iz konteksta prve smene, sadržao uglastu zagradu i end marker. Tako da bilo šta da naiđe na ulazu, znači ulaz bi se gutao ili do zatvorene uglaste zagrade ili do markera kraja.

Selekcioni skupovi

Sada je došao trenutak da objasnimo pojam selekcionih skupova i da vidimo šta gramatike moraju da zadovoljavaju da bi mogle da se procesiraju metodom od vrha ka dnu. Selekcioni skup je skup ulaznih simbola koji mogu predstavljati tekući ulaz u trenutku prepoznavanja neke smene $\langle X \rangle \rightarrow \alpha$, gde je α proizvoljna sekvenca gramatičkih simbola. Skup tih terminalnih simbola

naziva se selekcioni skup smene i označava ovim terminom SELECT(<X> → α), znači pridružen je smeni.

Da bi parser u nizu smena za isti neterminal <X> mogao jednoznačno se odluči za jednu od njih, selekcioni skupovi parova smena se istom levom stranom ne smeju sadržati ni jedan zajednički element, odnosno njihov presek mora biti prazan skup, odnosno oni moraju biti disjunktni.

$$SELECT(<X> \rightarrow \alpha) \cap SELECT(<X> \rightarrow \beta) = \emptyset, \forall \alpha,\beta: \alpha \neq \beta$$

Gramatike kod kojih je ovo ispunjeno, za sve moguće parove smena sa istom levom stranom, nazivaju se LL(1) gramatike. To je slična skraćenica kao što smo upotrebljavali kod bottom-up parsera. Prvo L znači da parser gleda ulaznu sekvencu sa leva na desno, drugo L znači da se gleda levo izvođenje ulazne sekvence i da se pri odlučivanju o sledećem koraku gleda 1 simbol sa ulaza.

Kako operativno računati selekcione skupove? Možemo preko već definisanih first i follow skupova. Za neku smenu <A> → α selekcioni skup smene je jednak first skupu njene desne strane α, ako desna strana nije poništiva,

$$SELECT(<A> \rightarrow \alpha) = FIRST(\alpha),$$

odnosno uniji first skupa desne strane i follow skupa leve strane, ako je desna strana poništiva:

$$SELECT(<A> \rightarrow \alpha) = FIRST(\alpha) \cup FOLLOW(<A>).$$

Ranije smo već definisali pojmove first i follow skupova i poništivosti simbola i nizova simbola. Tako da možemo samo videti kroz jedan primer izračunavanje selekcionih skupova.

Primer

Razmotrićemo primer gramatike od sedam smena koja ima i prazne smene, one malo komplikuju računanje select i follow skupova. Treba da odredimo selekcioni skup svake smene i posle toga da ispitamo da li se radi o LL(1) gramatici.

1. <A> → a <B> <C>
2. <A> → b <B>
3. <A> → ε
4. <B> → a <B> b
5. <B> → ε
6. <C> → b <C>
7. <C> → c

Ovaj primer smo već razmatrali na strani 95, gde smo dobili sledeće rezultate:

Poništivi = { <A>, <B>}

FIRST(<A>)={a, b}	FOLLOW(<A>)= {⊣ }
FIRST(<B>)={a}	FOLLOW(<B>)= {b,c, ⊣ }
FIRST(<C>)={b, c}	FOLLOW(<C>) = {⊣ }

Na osnovu ovih rezultata, sada je potrebno odrediti selekcione skupove. Select prve smene vidimo odmah da je to {a}. Select druge smene je {b}. Zanimljiviji slučaj je select treće smene. Tu moramo primeniti ovu drugu varijantu formule, za desnu starnu FIRST od prazne sekvence je prazan skup, desna strana ne daje nam ništa. Moramo uzeti follow skup leve strane smene, to je FOLLOW(<A>), znači selekcioni skup treće smene je {⊣ }. Što se tiče četvrte smene, selekcioni

skup je {a}. Što se tiče pete smene, moramo uzeti follow(<B>), a to je {b, c, ⊣}. Što se tiče šeste smene, selekcioni skup je {b}. Što se tiče sedme smene, selekcioni skup je {c}.

Dobili smo selecione skupove svih sedam smena. Šta je sada potrebno dodatno ispitati? Kada se gleda prva, druga i treća smena, selekcioni skupovi ne smeju sadržati ni jedan zajednički element, što je ispunjeno. Zatim kada se gledaju četvrta i peta smena koje imaju <B> na levoj strani, njihovi selekcioni skupovi ne smeju imati zajedničkih elemenata i to je ispunjeno. Konačno, kod šeste i sedme smene ne sme biti zajedničkih elemenata u selekcionim skupovima, to je takođe ispunjeno.

SELECT(1)={a}
SELECT(2)={b}
SELECT(3)=FOLLOW(<A>)={⊣}
SELECT(4)={a}
SELECT(5)=FOLLOW(<B>)={ b, c, ⊣}
SELECT(6)={b}
SELECT(7)={c}

Provera za smene 1. , 2. i 3. :
SELECT(1) ∩ SELECT(2) = {a} ∩{b} = ∅
SELECT(1) ∩ SELECT(3) = {a} ∩{⊣} = ∅
SELECT(2) ∩ SELECT(3) = {b} ∩{⊣} = ∅
Provera za smene 4. i 5. :
SELECT(4) ∩ SELECT(5) = {a} ∩{b, c, ⊣}= ∅
Provera za smene 6. i 7. :
SELECT(6) ∩ SELECT(7) = {b} ∩{c} = ∅

Pošto je sve to ispunjeno, konstatujemo da su svi selekcioni skupovi disjunktni za smene se istom levom stranom, ovo jeste LL(1) gramatika, to jest, možemo konstruisati top-down parser na bazi rekurzivnog spusta za ovu gramatiku. U okviru procedura u case strukturama, kada se odlučujemo za neku smenu, uzimamo u obzir ove izračunate selekcione skupove.

Transformacije gramatika

Šta kada gramatika nije LL(1), odnosno kada dobijemo negativan rezultat proveravanja disjunktnosti selekcionih skupova? Možemo probati neku od sledećih transformacija da bismo dobili gramatiku koja prepoznaje iste ulazne sekvence, ali koja jeste LL(1). Ove transformacije mogu u praksi da pomognu često, međutim teorijski ne garantuju da ćemo dobiti LL(1) gramatiku. I uopšte, mogu postojati jezici i gramatike za koje ne postoji uopšte ekvivalentna LL(1) gramatika. Videli smo ranije, kada smo pričali o deterministčkim parserima sa jednim ulaznim simbolom, da oni pokrivaju samo podskup skupa svih bezkontekstnih gramatika. Imamo tri transformacije koje će biti objašenje:

1. Uklanjanje direktne leve rekurzije

2. Uklanjanje indirektne leve rekurzije i

3. Leva faktorizacija.

Vrlo često se primenjuju prva i treća, ova druga po potrebi, ali ređe postoji potreba za njom i dosta komplikuje gramatiku koja se dobije, dok ova prva i treća su u suštini zgodne za primenu, zato što nisu preterano komplikovane.

Uklanjanje direktne leve rekurzije

Kod uklanjanja direktne leve rekurzije, tipično imamo u gramatici smenu koja ima na levoj strani neki neterminal <X> i desna strana počinje tim istim <X> i ima neki proizvoljan nastavak, a može za taj <X> postojati jedna ili više smena čije desne strane ne počinju sa <X>.

$\langle X \rangle \rightarrow \langle X \rangle \alpha$

$\langle X \rangle \rightarrow \beta$

Prva smena za <X> je levo rekurzivna smena. Pošto će ta smena imati sigurno u svom selekcionom skupu i simbole za ove druge smene, ona kvari koncept top-down parsiranja. Kako ćemo transformisati, odnosno, odrediti ekvivalentnu gramatiku koja nema ovo negativno svojstvo direktne leve rekurzije?

Analizirajmo šta opisuju ove dve smene. Najkraća sekvenca koja opisuje <X> je β, a duže sekvence se dobijaju od kraćih tako što se dodaje α nula ili više puta na desnoj strani. U suštini, to su sekvence koje počinju sa β, a iza toga imamo proizvoljan broj α. Mi moramo istu takvu sekvencu da opišemo, a da izbegnemo direktnu levu rekurziju.

Evo sada pravila koje je generalizovano, jer u opštem slučaju možemo imati za isto <X> više različitih smena, koje su direktno levo rekurzivne i takođe možemo imati više ovih drugih smena, čije desne strane ne počinju sa <X>. Čitav ovaj skup smena za <X> transformiše se u smene gde imamo niz smena čije desne strane počinju betama: β_1, do β_N, a u nastavku je uveden novi neterminal <Y>. Uloga <Y>-a je da opiše niz od nula ili više simbola α. <Y> mora biti novo uvedeni neterminal koji se do tada nije pojavljivao u gramatici. Opisujemo <Y> nizom α tako što koristimo desnu rekurziju: $<Y> \rightarrow \alpha_1 <Y>$, do $<Y> \rightarrow \alpha_N <Y>$. To je niz, gde svaki element može biti bilo koji ovaj α_1 ili α_2, ..., ili α_N. Konačno pošto to može biti i niz od nula α, moramo imati i praznu smenu za <Y>.

$<X> \rightarrow <X> \alpha_1$		$<X> \rightarrow \beta_1 <Y>$
....		....
$<X> \rightarrow <X> \alpha_n$	transformiše se u:	$<X> \rightarrow \beta_n <Y>$
$<X> \rightarrow \beta_1$		$<Y> \rightarrow \alpha_1 <Y>$
.....		....
$<X> \rightarrow \beta_n$		$<Y> \rightarrow \alpha_n <Y>$
		$<Y> \rightarrow \varepsilon$

Znači umesto originalnog skupa smena, uvodimo novi skup smena i time uklanjamo direktnu levu rekurziju. Ovu transformaciju je ponekad potrebno ponoviti nekoliko puta, pošto jedna primena ove ili neke druge transformacije može ponovo da uvede direktnu levu rekurziju, tako da primenjujemo dovoljan broj puta dok ne eliminišemo sve direktno levo rekurzivne smene.

Uklanjanje indirektne leve rekurzije

Što se tiče indirektne leve rekurzije, to je situacija kada recimo neka smena za <X> počinje nekim neterminalom <Y>, onda imamo smenu za <Y> koja počinje ovim polaznim neterminalom <X>. Naravno mogu postojati i neke smene koje nisu rekurzivne. U suštini, zamenom nekoliko puta <X> koristeći niz smena, opet možemo doći do sentencijalne forme koja počinje sa <X>, to je znači indirektna leva rekurzija. Primer gramatike sa indirektnog levom rekurzijom:

$<X> \rightarrow <Y>$ a
$<Y> \rightarrow <X>$ b
$<Y> \rightarrow$ c

Algoritam za uklanjanje indirektne leve rekurzije uvek može da se primeni na gramatiku kod koje ne postoje ciklusi ($<A>_i \Rightarrow^+ <A>_i$), odnosno kod koje ne postoji neki neterminal, gde bismo mogli izvođenjem u jednom ili više koraka. da dođemo do istog tog neterminala. Ovaj uslov nije preterano ograničavajući, jer takva gramatika bi bila i dvosmislena, tako da, ako gramatika nije dvosmislena, ona sigurno neće imati ni ove cikluse.

Kako uklanjamo indirektnu levu rekurziju iz gramatike? Algoritam deluje mnogo složenije nego ona pravila za uklanjanje direktne leve rekurzije, ali detaljno ćemo ga objasniti i ilustrovati na jednom primeru.

Poređati neterminale u nekom redosledu A1, A2, …, An
for i ← 1 to n
  for j ← 1 to i – 1
    zameniti svaku smenu oblika
      $<A>_i \rightarrow <A>_j\gamma$
    sa
      $<A>_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \ldots \mid \delta_k\gamma$,
      gde su $<A>j \rightarrow \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$ sve aktuelne smene za $<A>_j$
    end_for
  eliminisati direktno levo rekurzivne smene za $<A>_i$
  novodobijene neterminale dodati na kraj
end_for

Kao prvo, potrebno je gramatičke neterminale poređati u neki redosled razmatranja. Taj redosled može biti onaj redosled kako su napisane smene za njih. Sada imamo dvostruku petlju: spoljašnja petlja ide po ovom nizu neterminala redom, od prvog do poslednjeg. Unutrašnja petlja uvek ide po neterminalima koji su pre onog koji je tekući u spoljašnjoj. Recimo ako je u spoljašnjoj A2, u unutrašnjoj petlji razmatraćemo A1.

U unutrašnjoj petlji, ako dođemo do smene koja ima spoljašnji neterminal na levoj strani recimo, to bi bio u ovom primeru što sam naveo, A2, a da desna strana počinje sa ovim A1, onda je potrebno takvu smenu eliminisati i uzeti sve moguće desne strane smena za ovaj Aj i umesto ove originalne smene upisati nove smene gde ćemo Aj zameniti redom njegovim desnim stranama. Znači ako je Aj imao k smena, onda ćemo umesto jedne smene dobiti k novih smena, gde će redom biti desna strana prve smene za Aj, pa onda ovo gama, pa desna strana druge smene za Aj pa gama i tako dalje. Na taj način smo eliminisali iz ove smene Aj.

Kada se izađe iz te unutrašnje petlje, odnosno razmotre redom svi mogući parovi ovih neterminala, onda je još potrebno u spoljašnjoj petlji, pre nego što pređemo na sledeću iteraciju, razmotriti da li postoji direktna leva rekurzija po neterminalu spoljašnje petlje. Ako postoji, treba je prema onom prethodnom pravilu eliminisati i onda se pređe na sledeću iteraciju spoljašnje petlje.

Pri eliminaciji direktne leve rekurzije moguće je, prema onom prethodnom pravilu, da se u gramatici pojave i novi neterminali koji se nisu pojavljivali u inicijalnoj listi. Takve nove neterminale treba dodati na kraju ove liste i mora se iterirati i po njima, znači lista za iteraciju nije fiksna, nego može dinamički da se proširi.

Primer

Evo sada jedan konkretan primer gramatike koja poseduje indirektnu levu rekurziju. Ona poseduje i direktnu levu rekurziju zbog druge smene, ali ima i indirektnu jer, recimo, <E> se menja sa <T>, a <T> onda može početi sa <E>.

$<S> \rightarrow <E>$
$<E> \rightarrow <E> + <T>$
$<E> \rightarrow <T>$

<T> → <E> - <T>
<T> → id

Da ilustrujemo algoritam na ovoj gramatici. Poredjajmo neterminale redom <S>, <E>, <T>. Sada razmtramo onu dvostruku petlju. U prvoj iteraciji, spoljašnji neterminal je <S>, unutrašnjeg neterminala nema pošto je <S> prvo u redosledu neterminala, znači ne ulazimo u unutrašnju petlju. Međutim, potrebno je razmotriti da li imamo direktnu levu rekurziju po <S> -- nemamo. Tako da smo završili sa prvom iteracijom spoljašnje petlje i sa <S>.

U drugoj iteraciji spoljašnje petlje, <E> je tekući spoljašnji neterminal. Ulazimo u unutrašnju petlju i u unutrašnjoj petlji <S> je tekući neterminal. Razmatramo da li postoje smene u gramatici koje imaju na levoj strani <E>, a da desna strana počinje sa <S>. Nema ih (imamo da je leva strana <S>, a da desna počinje sa <E>, ali nama treba obrnuto). Znači da u unutrašnjoj petiji nema transformacija. Izlazi se iz unutašnje petlje. Onda još dodatno treba proveriti da li postoji direktna rekurzija po neterminalu spoljašnje petlje <E>. Postoji, to je ova druga smena. Primenom pravila za uklanjanje direktne leve rekurzije, α je +<T>, a beta je <T>. Pišemo <E> → β <E'>, uvedeni neterminal je <E'> i onda pišemo <E'> → α <E'> i <E'> → ε, znači dve nove smene za novi neterminal. Taj novi neterminal stavljamo na kraj iteracione lste. Time smo završili sa <E> u spoljašnjoj petlji.

STARO	**NOVO**
<S> → <E>	<S> → <E>
<E> → <E> + <T>	**<E> → <T> <E'>**
<E> → <T>	**<E'> → + <T> <E'>**
<T> → <E> - <T>	**<E'>→ ε**
<T> → id	<T> → <E> - <T>
	<T> → id

Prelazimo na treću iteraciju spoljašnje petlje, gde je <T> tekući neterminal spoljašnje petlje, a redom prvo razmatramo <S>, pa onda <E> u unutrašnjoj petlji. Ako gledamo da li imamo smenu sa <T> na levoj strani, <S> na desnoj strani, to nemamo znači u unutrašnjoj petlji u prvoj iteraciji nema promene. zatim, u drugoj iteraciji unutrašnje petlje, na levoj strani imamo <T>, a na početku desne strane <E>, znači tu situaciju imamo. Znači treba izbaciti ovu smenu i zameniti <E> svim njegovim smenama.

STARO	**NOVO**
<S> → <E>	<S> → <E>
<E> → <T> <E'>	<E> → <T> <E'>
<E'> → + <T> <E'>	<E'> → + <T> <E'>
<E'>→ ε	<E'>→ ε
<T> → <E> - <T>	**<T> → <T> <E'> - <T>**
<T> → id	<T> → id

Sva sreća za <E> imamo samo jednu smenu, tako da umesto ove izbačene, dodajemo samo jednu novu i u toj novoj smo <E> zamenili sa <T><E'>. Više nema šta da razmatramo u unutrašnjoj petlji, nego još samo treba u spoljašnjoj petlji razmotriti da li ima direktne leve rekurzije po <T>. Ima -- zbog pretposlednje smene i potrebno ju je eliminisati. Znači α je <E'>-<T>, β je id. Umesto starih smena za <T> pišemo jednu novu i dve smene za novouvredeni neterminal <T'> i dodajemo <T'> na kraj uredjene liste neterminala.

STARO	**NOVO**
<S> → <E>	<S> → <E>
<E> → <T> <E'>	<E> → <T> <E'>
<E'> → + <T> <E'>	<E'> → + <T> <E'>
<E'>→ ε	<E'>→ ε
<T> → <T> <E'> - <T>	**<T> → id <T'>**
<T> → id	**<T'>→ <E'> - <T> <T'>**
	<T'> → ε

Završili smo sa trećom iteracijom i sa <T> u spoljašnjoj petllji. Šta je još ostalo za spoljašnju petlju? Onaj novouvedeni neterminal <E'>, a u unutrašnjoj petlji redom Idemo <S>, <E>, <T>. Gledamo da li imamo <E'> na levoj strani, a neke od ovih pomenutih <S>, <E> i <T> na počecima desnih strana. Nemamo, nema promene u unutrašnjoj petlji i pošto nema direktne leve rekurzije po <E'> završavamo iteraciju po <E'>.

Onda uzimamo <T'> u spoljašnji petlji kao tekući. Sada moramo redom da gledamo <S>, <E>, <T> i <E'> u unutrašnjoj petlji. Za <S>, <E> i <T> nema promene, međutim za <T'> u spoljašnjoj petlji, a za <E'> u unutrašnjoj, ima potrebe da se transformiše ova smena i da se izbaci početno <E'>. Za <E'> imamo dve smene, znači moramo napraviti dve nove smene. Jednom ćemo <E'> zameniti za +<T><E>, a drugi put ćemo <E'> samo izbaciti primenom ove prazne smene. Tako da umesto jedne smene dobijamo ove dve smene. Sada pošto završavamo unutrašnju petlju još razmatramo da li postoji direktna leva rekurzija po <T'>. Pošto te direktne leve rekurzije nema, zaključujemo da je ovo finalna gramatika.

STARO	**NOVO**
<S> → <E>	<S> → <E>
<E> → <T> <E'>	<E> → <T> <E'>
<E'> → + <T> <E'>	<E'> → + <T><E'>
<E'>→ ε	<E'>→ ε
<T> → id <T'>	<T> → id <T'>
<T'>→ <E'> - <T> <T'>	**<T'>→ +<T><E'> - <T> <T'>**
<T'> → ε	**<T'>→ - <T> <T'>**
	<T'> → ε

Leva faktorizacija

Time smo završili ilustraciju ovog komplikovanog pravila uklanjanja indirektne leve rekurzije, a sada sledi jedno jednostavno pravilo vezano za levu faktorizaciju. Ovo pravilo primenjuje se ako u gramatici imamo niz smena za neki ne terminal, ovde je napisan <S>.

<S> → α β_1		<S> → α <S′>
....		<S′> → β_1
<S> → α β_n	→	....
		<S′> → β_n

Te smene mogu imati različite desne strane, međutim te desne strane mogu imati neki zajednički prefiks α, pa onda neke različite nastavke. Ovo α smeta u selekcionim skupovima, uzrokovaće da isti terminali uđu u selekcione skupove ovih smena. Da bismo to eliminisali, uvodimo novi

neterminal <S'> koji će opisati ove različite nastavke β_1 do β_N i umesto ovih originalnih smena, stavljamo smenu za <S> -> α <S'>.

Kao što je već rečeno, teorijski nije garantovano da ćemo ovim transformacijama proizvoljnu gramatiku dovesti u LL(1) oblik, ali u praksi, kod većine programskih jezika, moguće ih je izraziti LL(1) gramatikama.

Primer jezika koji nema LL(1) gramatiku

Izložićemo jedan čisto teorijski primer jezika koji nije uopšte opisiv LL(1) gramatikom:

$$\{ a^n 0 b^n \mid n \geq 1 \} \cup \{ a^n 1 b^{2n} \mid n \geq 1 \}$$

To je jezik koji se sastoji od dve vrste sekvenci. Ulazni simboli su a, 0, 1 i b. Ako imamo sekvencu koja počinje sa jednim ili više a, onda ide 0 i onda ide b onoliko puta koliko je bilo onih a, to je jedna vrsta sekvenci u jeziku. Druga vrsta sekvenci je da počinju sa a, 1 ili više puta, zatim da ide 1, pa posle toga ide b duplo više puta nego što ima a na početku.

Ako je jezik definisan na ovaj način, možemo napisati neku bezkonteksntu gramatiku za ovaj skup sekvenci. Sekvence prvog oblika ćemo opisati sa ovim <S'>. Sekvence drugog oblika ćemo opisati sa <S">. Onda ćemo ih podvesti pod zajednički neterminal <S>.

Kako opisujemo recimo sekvence prvog oblika? Najkraća takva sekvenca je 0ba. Duža sekvenca se dobija od kraće, tako što napred dodamo a, nazad b. Za ovaj drugi oblik sekvence, najkraća sekvenca je a1bb, a duže sekvence se dobijaju tako što na kraću napred dodamo jedno a, nazad dva b. Kao što vidimo, nije nikakav problem da se ovaj jezik opiše bezkontekstnom gramatikom.

<S> $\rightarrow$ <S'>
<S> $\rightarrow$ <S">
<S'> $\rightarrow$ a0b
<S'>$\rightarrow$ a<S'>b
<S"> $\rightarrow$ a1bb
<S">$\rightarrow$ a<S">bb

Međutim, ova gramatika nije LL(1) zato što recimo selekcioni skupovi prve dve smene imaju zajedničko a. Na ulazu mora da se prođe svih ovih N simbola a, da bi se došlo do 0 ili 1, da bi se moglo videti da li je u pitanju prvi ili drugi oblik ulazne sekvence. Simboli 0 i 1 ne mogu biti u istoj smeni, jer se nikada ne pojavljuju zajedno ni u jednoj sekvenci. Kada se uzme u obzir postojanje dve smene sa 0 i 1, ne možemo gledajući samo po jedan simbol na ulazu u jednom trenutku, konstruisati stablo izvođenja od vrha ka dnu, uvek će se znači negde pojaviti konflikt za ulaz a pri izboru smena. To je (neformalno) objašnjenje zašto ne može da se nađe za ovaj jezik LL(1) gramatika.

Sintaksno-upravljano prevođenje

Prvo ćemo definisati pojam translacionih gramatika, zatim ćemo proširiti translacione gramatike atributima da bismo dobili atributivno translaciona gramatike. To je formalna osnova za definisanje sintaksno-upravljanog prevođenja. Zatim ćemo videti kako proširiti parsere od dna ka vrhu da obavljaju sintaksno-upravljano prevođenje u skladu sa specifikacijom koja je zadata jednom atributivno-translacionom gramatikom.

Translacione gramatike

Pojam translacione gramatike razmotrićemo gledajući primer ove gramatike od 8 smena. Vidimo da su smene napisane crnim slovima sem simbola koji su označeni plavim i uokvireni vitičastim zagradama. Kod translacionih gramatika skup terminalnih simbola delimo na ulazne i akcione simbole. Ovi što su posebno izdvojen plavo i koji se pišu u vitičastim zagradama su akcioni simboli. Ostali terminali su ulazni simboli. U ovom konkretnom primeru, neterminali su <E>, <T>, <P>, a ostalo su terminalni simboli. Ulazni simboli su plus, zvezdica, otvorena zatvorena zagrada, a, b, c. Što se tiče akcija, to su sve ove uokvirene u vitičaste zagrade.

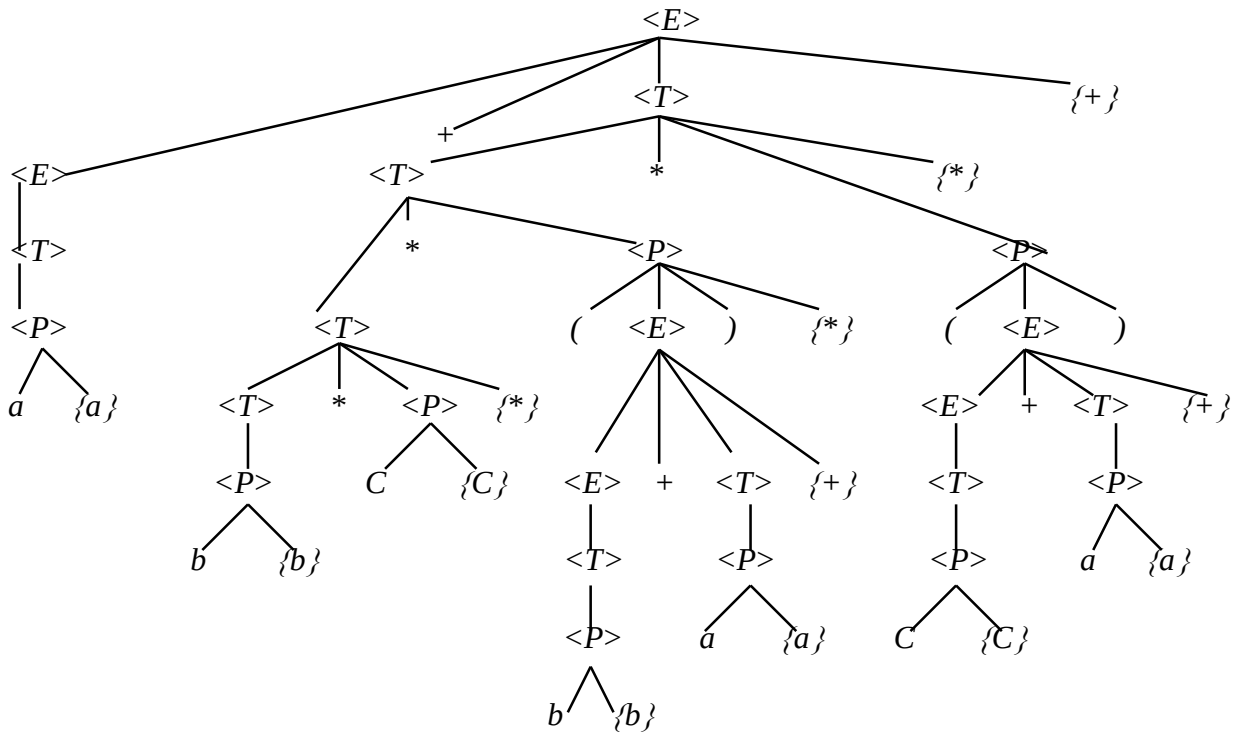
1. <E> → <E> + <T> {+}
2. <E> → <T>
3. <T> → <T> * <P> {*}
4. <T> → <P>
5. <P> → (<E>)
6. <P> → a {a}
7. <P> → b {b}
8. <P> → c {c}

Šta se postiže ovim razdvajanjem terminala na ulazne i akcione simbole? Dobija se specifikacija prevođenje koje matematički odgovara pojmu preslikavanja. Određena ulazna sekvenca, koja se sastoji od ulaznih simbola, preslikava se u odgovarajuću akcionu sekvencu. Gramatika u prethodnom primeru na svom ulazu opisuje infiksne aritmetičke izraze sa operandima a, b, c, sa operacijama plus i puta i upotrebom zagrada. Jedan primer takvog izraza bi bio ovaj a + b u zagradi što se množi sa c. Ovo su infiksni izrazi. Infiksni znači da se operator nalazi između operanada. Uobičajeno kad ljudi pišu aritmetičke izraze pišu u infiksnoj notaciji, a može da se koristi i postfiksna notacija gde se prvo pišu operandi pa onda operatori.

Ekvivalentan izraz ovom infiksnom izrazu (a+b)*c bi bio ovaj izraz {a} {b} {+} {c} {*}. Ovde vidimo da su u pitanju akcioni simbolimi. Ova konkretna gramatika prevodi, kada dođe recimo a na ulazu, na izlaz će biti emitovana odgovarajuća akcija {a}. Kako postižemo da infiksni oblik prevedemo u postfiksni? Tako što akciju za operator pišemo tek kada prođu operandi. Recimo ovde <E> plus <T> pa tek onda se emituje na izlaz operator plus. Da smo ovu akciju {+} stavili odmah Iza ovog ulaznog znaka plus, dobili bismo na izlazu potpuno isti oblik kao na ulazu. Zanimljivo je da u izlaznim akcionim simbolima nemamo zagrade bez obzira kakvi su infiksni izrazi, ako upotrebljavaju zagrade za menjanje prioriteta operacija. U postfiksnim izrazima zagrade

nikad nisu potrebne. Bilo kakav infiksni izraz sa zagradama može se prevesti u postfiksni izraz bez zagrada.

Ovaj primer nam je pomogao da definišemo translacionu gramatiku. Malo formalnije gledano, kako se odvija to preslikavanje ulaza na izlaz? Možemo prikazati preko translacionog stabla izvođenja.



Translaciono stablo izvođenja je vrlo slično klasičnom stablu izvođenja. Njega konstruišemo na osnovu ulazne sekvence, na isti onaj način kako smo konstruisali izvođenja i stablo izvođenja, samo što se u translaciono stablo izvođenja, kada se naznačava primena neke smene, ovde recimo vidimo koren i neposredni sledbenici korena su <E> plus <T> i akcija plus, da je u pitanju ova ovde gramatika i primena prve smene. Znači <E> je bilo u korenu, a njegovi neposredni sinovi ovim redom kako su navedeni na desnoj strani smene, <E> + <T> i uvek kad detektujemo primenu neke smene unose se i odgovarajući akcioni simboli. Jednom kada za zadati ulaz konstruišemo kompletno translaciono stablo izvođenja, što znači da će u njegovom korenu biti startni simbol, a u listovima možemo imati samo terminalne simbole bilo ulazne bilo akcione, sekvencu aktivnosti dobijamo obilaskom ovog translaciono stabla izvođenja s leva na desno i beleženjem simbola u listovima. Ako zabeležimo redom sve listove, dobićemo takozvanu sekvencu aktivnosti:

a {a} + b {b} * c {c} {*} * (b {b} + a {a} {+}) {*} * (c {c} + a {a} {+}) {*} {+}

Ako iz sekvence aktivnosti uklonimo akcione simbole, dobićemo ulaznu sekvencu. znači ono što dolazi na ulazu parsera,

a + b * c * (b + a) * (c + a)

a ukoliko iz sekvence aktivnosti uklonimo ulazne simbole, a preostanu samo akcije, dobićemo takozvanu akcionu sekvencu koja u stvari predstavlja prevod date ulazne sekvence:

{a} {b} {c} {*} {b} {a} {+} {*} {c} {a} {+} {*} {+}

Ovde vidimo da je stablo na strani 129 odgovaralo jednom složenijem infiksnom izrazu i da tom infiksnom izrazu odgovara ovaj ovde postfiksni izraz. Nije bitno za ovu definiciju translacione gramatike ali evo da dam mali komentar. Zašto nam je nekad zgodno prevoditi infiksne u postfiksne izraze? Za razliku od infiksnih izraza koji su ljudima lakši za baratanje, postfiksni izrazi su lakši za kompjutere. Recimo, ako treba da izračunamo vrednost izraza, a on je zadat u ovom postfiksnom obliku, onda je dovoljno uvesti jedan stek vrednosti, gde kada čitamo redom ove elemente postfiksnog izraza s leva na desno, kad god naiđemo na operand odnosno vrednost, stavljamo ga na stek, a kada naiđemo na operator, ako je u pitanju recimo binarni operator, onda skidamo poslednje dve vrednosti s vrha steka, primenimo operaciju i rezultat operacije stavimo na stek. I tako vrlo lako odredimo vrednost izraza što bi dosta teže bilo isprogramirati kada je izraz u ovom infiksnom obliku.

Atributivno-translacione gramatike

Sada ćemo uvesti pojam atributivno-translacionih gramatika. Videli smo da je vrlo zgodno uvesti akcije, jer time definišemo prevođenje, način kako se jedan jezik može prevesti u drugi izlazni jezik, kao skup akcionih sekvenci. Međutim, atributi nam daju i novu fleksibilnost -- mogućnost da se unutar stabla izvođenja prosleđuju dodatne informacije među pojedinim čvorovima putem vrednosti atributa. Atribute pridružujemo pojedinim gramatičkim simbolima. U zapisivanju gramatike videćemo da atribute pišemo uz gramatički simbol kao donji indeks uz simbol. Takođe ćemo videti da, sa stanovišta toka prenosa informacija kroz stablo, atribute delimo na dva tipa atributa; to su takozvani *sintetizovani* atributi koji prenose informacije od listova stabla ka korenu, to jest odozdo na gore i *nasleđeni* atributi koji mogu da prenose informacije od korena stabla ka listovima.

Primer atributivno-translacione gramatike je jedna gramatika od četiri smene koja ima neterminalne simbole <S> i <A>, a ulazne simbole a, b i c u ovoj četvrtoj smeni, a kao akcione simbole sve ove simbole sa vitičastim zagradama: {c}, {d}, {f} i u četvrtoj smeni {e}.

1. $\langle S\rangle_{p4,q1} \rightarrow a_{p1}\ \langle S\rangle_{t1,p2}\ b_{s1}\ \{c\}_{q2,p3}\ \langle A\rangle_{r1,q3}\ \{d_{r2,s2,t2}\}$
   $(p2, p3, p4) \leftarrow p1(\ q2, q3\ ) \leftarrow q1 \quad s2 \leftarrow s1\ \ t2 \leftarrow t1\ \ r2 \leftarrow r1$
2. $\langle S\rangle_{p2,p1} \rightarrow \varepsilon$
   $p2 \leftarrow p1$
3. $\langle A\rangle_{p3,t1} \rightarrow \langle S\rangle_{p1,t2}\ b_{q1}\ \langle A\rangle_{r1,q2}\ \langle A\rangle_{s1,q3}\ \{f_{r2,s2,p2}\ \}$
   $(p2, p3) \leftarrow p1 \quad (q2, q3) \leftarrow q1 \quad r2 \leftarrow r1\, s2 \leftarrow s1 t2 \leftarrow t1$
4. $\langle A\rangle_{p2,q1} \rightarrow \{e_{q2}\}\ c_{p1}$
   $p2 \leftarrow p1 \qquad q2 \leftarrow q1$

Svi ovi gramatički simboli imaju neke atribute. Atributi su navedeni u indeksu uz simbol i imaju svoja imena. Za gramatiku se često navode tipovi pojedinačnih atributa, recimo ovde je navedeno da neterminal <S> ima dva atributa gde je prvi atribut uvek sintetizovan, a drugi nasleđen. Takođe,

gramatički simbol <A> ima dva atributa gde je prvi atribut sintetizovan, a drugi nasleđen. I navedeno je da su atributi akcionih simbola nasleđeni. Primetite da notacija omogućava i da se piše atribut van vitičaste zagrade, ili da se atribut piše unutar vitičaste zagrade, to je u suštini svejedno.

Ako smo rekli da su atributi vrednosti, potrebno je znati kako odrediti te vrednosti. Tome služe atributivna pravila. Atributivna pravila pišu se uz gramatičke smene. Najčešće se pišu odmah ispod određene smene pravila za računanje vrednosti raznih atributa koji se pojavljuju u smeni. Recimo, vrednost atribut p2 je kopija vrednosti atributa p1 i takođe ovo je neki skraćeni oblik, vidimo da istovremeno i p3 i p4 dobijaju tu vrednost atributa p1 i tako dalje. U ovom konkretnom primeru, sva pravila dodele vrednosti atributa su prosta pravila kopiranja, znači direktno se kopira vrednost nekog drugog atributa, ali to u opštem slučaju ne mora biti. Znači nismo ograničeni u tome kojom funkcijom su povezani vrednosti izvorišnih i odredišnih atributa, to može biti proizvoljno složena funkcija, odnosno u implementaciji parsera procesora to može biti proizvoljni fragment programskog koda.

Osnovna pravila konzistencije atributivne gramatike

Ono što je bitno za gramatike, to je da se moramo pridržavati određenih pravila pri pisanju atributa i njihovih pravila, da bi cela gramatika imala smisla. Gledajući ovaj primer navešću neka od tih osnovnih pravila, pa ćemo kroz naredne slajdove sistematski pobrojati sva ta pravila, koja uvek moraju biti zadovoljena i koja se zovu osnovna pravila konzistencije gramatike da bi uopšte imala smisla.

1. Svakoj smeni pridružuju se pravila izračunavanja vrednosti atributa simbola koji se pojavljuju u toj smeni. Ne mogu se koristiti niti računati atributi simbola iz drugih smena.
2. Atributi se moraju slagati po broju i tipu za sve pojave istog simbola X, ne moraju i po imenu.
3. Za svaki *sintetizovani* atribut koji se pojavljuje na *levoj* strani smene, mora postojati pravilo za računanje i on može zavisiti od nasleđenih atributa sa leve strane i od bilo kog atributa sa desne strane smene. Dakle, sintetizovane attribute računamo kada se pojavljuju na levim stranama smena.
4. Komplementarno, za svaki *nasleđeni* koji se pojavljuje na *desnoj* strani smene mora postojati pravilo za njegovo računanje. On može da zavisi od nasleđenih atributa sa leve strane smene i od bilo kog atributa sa desne strane smene.
5. Vrednost nasleđenih atributa startnog neterminala mora biti unapred zadata, zato što prosto nema smene gde možemo te vrednosti navesti. To je nešto što dolazi kao ulazni parametar parsera.
6. Vrednosti atributa ulaznih simbola se ne računaju gramatičkim pravilima, nego moraju biti unapred zadate. Njihove vrednosti obezbeđuje leksički analizator kroz vrednosni deo ulaznog tokena. Rekli smo da je token obično zapis od dva polja, klasa i vrednost. Klasa određuje koji terminal je u pitanju, a vrednost određuje njegov sintetizovani atribut.

Za ilustraciju 2. pravila u gramatici iz primera, recimo <S> se pojavljuje na levim stranama prve i druge smene, takođe se pojavljuje na desnoj strani prve smene i na desnoj strani treća smene. Znači sve pojave određenog gramatičkog simbola moraju imati isti broj atributa, ako smo ovde naveli dva atributa, onda i na svim drugim mestima mora biti dva atributa. Oni ne moraju biti isto imenovani, ali moraju biti po tipu sve pojave iste. Znači, svuda prvi atrubut <S> mora biti sintetizovan, a drugi atribut nasleđen i ako ne moraju svuda imati isto ime.

Za ilustraciju 3. pravila, vidimo recimo da je prvi atribut od leve strane 1. smene <S> sintetizovan, to je p4 i to znači da mora postojati pravilo koje definiše kako ćemo tom p4 dodeliti vrednost. Isto važi i za t1 da je sintetizovan, ali pošto se on pojavljuje na desnoj strani smene, njemu ne dodeljujemo vrednosti. Mi vidimo da on u pravilima figuriše na ovom mestu, ali on je izvor, znači možemo da koristimo njegovu vrednost, ali dodela vrednosti mora postojati samo za sintetizovane atribute na levoj strani smene.

Za ilustraciju 4. pravila, drugi atribut p_2 neterminala <S> sa desne strane 1. Smene, recimo, tretira se kao nasleđen. To recimo znači da p_2 atribut mora imati pravilo kako mu se određuje vrednost. I zaista prvo atributivno pravilo 1, smene određuje da p_2 dobija vrednost od p_1. Znači za svaki nasleđeni atribut, kada se pojavljuje na desnoj strani smene, mora postojati pravilo za dodelu vrednosti. Recimo za q_1 iz prve smene, on jeste nasleđen, ali pošto se pojavljuje na levoj strani smene, njemu ne određujemo vrednost, ali naravno možemo da ga koristimo.

Pravila uvek imaju neke svoje izuzetke, na primer alati za generisanje parsera yacc i cup omogućuju pod određenim okolnostima pristup i atributima iz nekih drugih smena, ali to je jedan mehanizam koji je sa jedne strane ograničen (pod određenim uslovima), a s druge strane, vrlo je nesiguran, zato što ako kasnije menjate gramatiku, može lako da vam se desi da poremetite ta pravila, tako da, čak i ako alat to omogućava, dobra praksa je da se zadržimo na ovom teorijskom ograničenju.

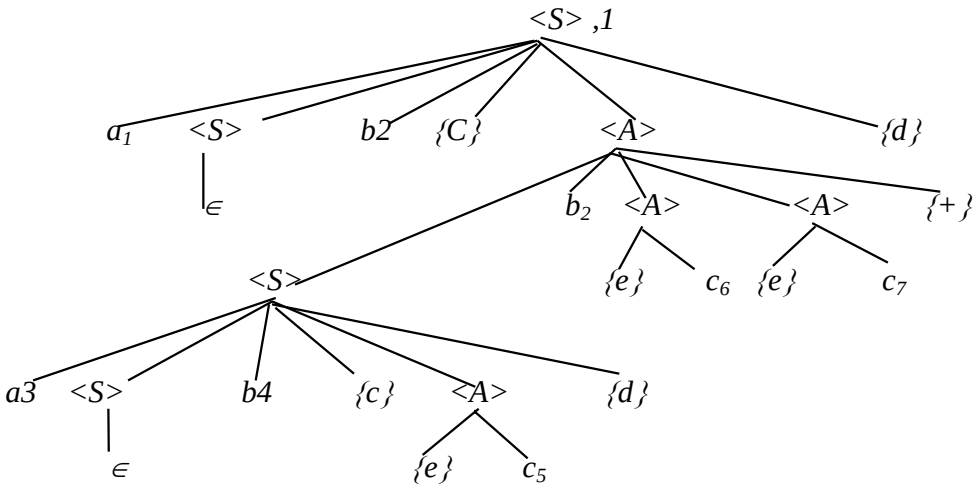
Pošto smo videli primer jedne atributivno-translacione gramatike koja zadovoljava sva ova pravila konzistencije, mi ćemo kako izračunati vrednosti atributa. Objasnićemo upravo koristeći ovaj primer gramatike. Uzećemo i primer ulazne sekvence koja se sastoji od ovih osam ulaznih simbola i uz svaki simbol navedena je i vrednost njegovog atributa:

$a_1\ b_2\ a_3\ b_4\ c_5\ b_2\ c_6\ c_7$.

Dodatno, pošto startni neterminal <S> ima nasleđeni atribut, moramo unapred definisati njegovu vrednost i neka to bude 1.

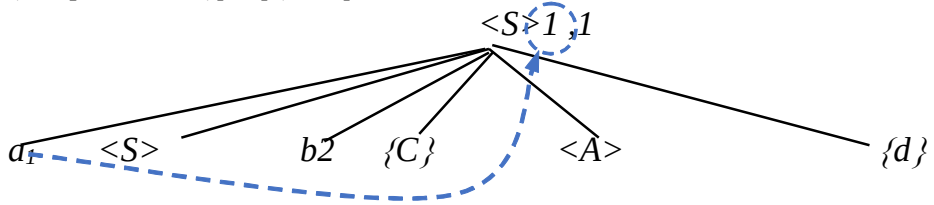
Prvo je potrebno, za zadati ulaz, konstruisati stablo izvođenja, to je ono translaciono stablo izvođenja. Vidimo da se ovde pojavljuju i akcioni simboli, mada samo ulazni simboli definišu strukturu stabla. Stablo odgovara ovoj gramatici od četiri smene. Posle je upotrebljena smena za <A>, posle je opet upotrebljena ista prva smena za <S>, to je za zadatu ulaznu sekvencu stablo izvođenja. Kako možemo da se uverimo da jeste za datu sekvencu? Pa tako što obiđemo ovo stablo i redom s leva na desno popišemo sve listove, znači a, b, pa se spuštamo dole, a, b, c, pa onda ide b, c, c. Znači konstruisali smo stablo izvođenja bez atributa.

U drugom koraku naznačimo sve što inicijalno znamo, sve što je unapred zadato, a to su znači atributi u listovima ovih ulaznih simbola i takođe znamo ovaj drugi, odnosno nasleđeni atribut korena, startnog neterminala.



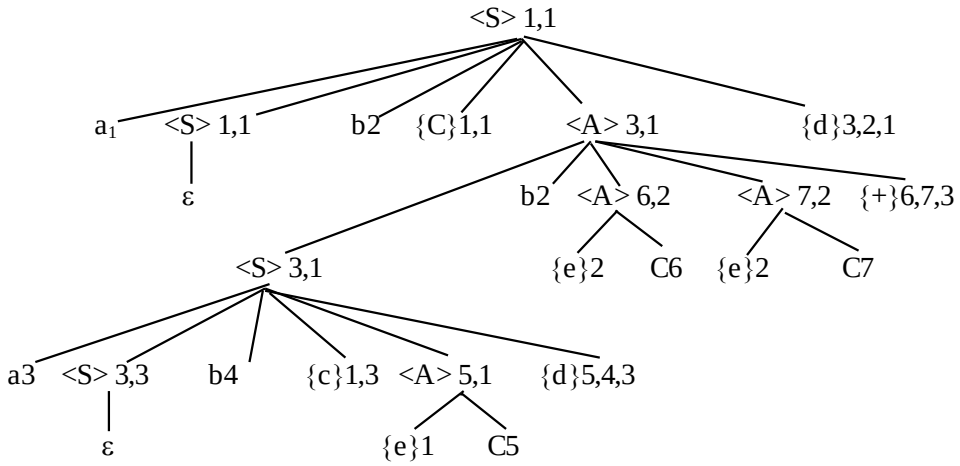
Sada je potrebno izračunati vrednosti preostalih atributa. Koju sistematiku koristimo? Moramo da idemo kroz celo stablo i razmatramo primenu svake smene koja je primenjena u stablu. Redosled kako ćemo ići, da li ćemo krenuti od korena prema listovima ili obrnuto, nije važan. Samo je važno sistematski obići celo stablo ćvor po čvor za te unutrašnje čvorove koji predstavljaju primene smena. Mi možemo da krenemo od korena i to predstavlja primenu prve smene koja je ovde prepisana.

1. $<S>_{p4,q1} \rightarrow a_{p1}\ <S>_{t1,p2}\ b_{s1}\ \{c\}_{q2,p3}\ <A>_{r1,q3}\ \{d_{r2,s2,t2}\}$
(p2, p3, p4) ← p1 (q2, q3) ← q1 s2 ← s1 t2 ← t1 r2 ← r1



Na osnovu pravila p4 ← p1 iz smene 1.

Primenjujući ova atributivna pravila, računamo vrednosti atributa. Mogu biti izračunate sve vrednosti za koje su poznati izvorišni delovi pravila. Recimo, atribut ulaznog simbola a t1 je jednak 1, znači on je poznat. Tako da možemo izračunati p2, p3 i p4. p4 je recimo sintetizovani atribut na levoj strani. Ovde je prikazano da p4 dobija vrednost od p1 i tako dalje. Možemo naznačiti da je i p3 jednako 1, p3 je drugi atribut akcije {c}. Zatim p2 je isto jednako 1, to je drugi nasleđeni atribut <S> na desnoj strani smene, to je ovaj ovde čvor pa njegov drugi atribut. q1 je drugi nasleđeni atribut od <S>, on nam je isto poznat i jednak 1 tako da nam je poznato i q2 i q3. s1 je atribut ulaznog simbola b jednak 2, tako da nam je poznato i s2. Zatim t1 je prvi atribut od ovoga <S> na desnoj strani. t1 nije poznat u ovom trenutku, tako da ovo pravilo ne možemo primeniti, to jest, ne možemo izračunati t2. I konačno, r1 je prvi atribut od <A>, on nam isto tako nije poznat, to je ovo <A> na desnoj strani, znači nije nam poznata vrednost r1 tako da ne možemo izračunati r2.



U opštem slučaju nećemo uspeti već u prvom prolazu izračunati sve vrednosti. Zbog toga, kada jednom prođemo kroz celo stablo čvor po čvor, vraćamo se ponovo, pa ćemo ponovo razmotriti koren u drugoj iteraciji i svaki drugi čvor i tako dalje, znači ponavljaćemo te prolaske kroz stablo iterativno dok postoji bilo kakva promena. To znači, dok ili sve atribute ne izračunamo, Ili dok dalje računanje nije moguće. Konkretno u našem primeru, moguće je bilo izračunati u konačnom broju iteracija sve atribute i ovde je navedeno kompletno stablo. To je sada atributivno-translaciono stablo izvođenja.

Ako popišemo sistematski sve listove stabla, dobićemo sekvencu aktivnosti:

$a_1\ b_2\ \{c\}_{1,1}\ a_3\ b_4\ \{c\}_{1,3}\ \{e\}_1\ c_5\ \{d\}_{5,4,3}\ b_2\ \{e\}_2\ c_6\ \{e\}_2\ c_7\ \{d\}_{6,7,3}\ \{d\}_{3,2,1}$

Ako iz sekvence aktivnost izostavimo sve akcione simbole dobićemo ulaznu sekvencu $a_1\ b_2\ a_3\ b_4\ c_5\ b_2\ c_6\ c_7$, a ako iz sekvence aktivnosti izostavimo ulazne simbole a ostavimo akcione simbole dobijamo izlaznu sekvencu koja u stvari predstavlja prevod i vidimo da ovde i akcioni simboli imaju konkretne vrednosti svojih atributa:

$\{c\}_{1,1}\ \{c\}_{1,3}\ \{e\}_1\ \{d\}_{5,4,3}\ \{e\}_2\ \{e\}_2\ \{d\}_{6,7,3}\ \{d\}_{3,2,1}$

Sinteza atributivno-translacione gramatike

Prethodni primer atributivno-translacione gramatike je bio čisto formalan, nismo mu davali nikakvo značenje, samo nam je poslužio da objasnimo osnovne pojmove. Sada možemo videti jedan primer tipa sinteze atributivno-translacione gramatike. Zadata je jedna ulazna gramatika koja nema atribute niti akcije, koja ima tri smene:

1. <integer> → digit <more digits>
2. <more digits> → digit <more digits>
3. <more digits> → ε

<integer> se opisuje kao digit iza čega sledi <more_digits>. Smene 2 i 3 opisuju <more_digits> kao niz od nula ili više digit terminala. U suštini <integer> je opisan kao sekvenca od jednog ili više digit terminala.

Dalje se nastavlja postavka: treba dodeliti atribute ovoj gramatici, tako da neterminalni simbol <integer> ima sintetizovani atribut jednak vrednosti celog broja koji ovaj simbol generiše. Pri tome treba pretpostaviti da ulazni simbol digit ima atribut jednak vrednosti odgovarajuće cifre.

Uradićemo dva različita rešenja, da se ilustruje da je moguće koristiti različite vrste atributa. U prvom rešenju, koje je inače lakše za shvatanje, pa se samim tim češće i primenjuje, koriste se samo sintetizovani atributi.

1. $\langle integer \rangle_{value} \rightarrow digit_{val} \langle more\ digits \rangle_{v,\ b}$
   value ← val * 10^b + v

2. $\langle more\ digits \rangle_{v,\ b} \rightarrow digit_{val} \langle more\ digits \rangle_{v1,\ b1}$
   v ← val * 10^{b1} + v1 b ← b1 + 1

3. $\langle more\ digits \rangle_{v,\ b} \rightarrow \varepsilon$
   v←0 b←0

Rečeno je, prema postavci, da svaki terminal digit ima atribut koji predstavlja vrednost odgovarajuće cifre. Mi ćemo taj atribut nazvati val i dodaćemo indeks val svakoj pojavi terminala digit.

S druge strane, rečeno je da startni neterminal treba da ima atribut koji će na kraju da sadrži vrednost celog broja. Treba da vidimo kako na bazi tih podataka izračunati vrednost broja. Ako gledamo samo prvu smenu, mi imamo prvu cifru i ostatak broja koji je reprezentovan ovim neterminalom <more_digits>.

Da bismo izračunali kompletnu vrednost broja, moramo poznavati vrednost ostatka broja, koji čine sve cifre mimo ove prve cifre i moramo poznavati koliko tih preostalih cifara ima. Tada možemo izračunati vrednost kompletnog broja po formuli vrednost prve cifre puta njen težinski koeficijent i plus vrednost ostatka broja. Njen težinski koeficijent je osnova brojnog sistema na mesto te cifre, odnosno broj preostalih cifara, znači 10^b, gde je b broj preostalih cifara. Podrazumevan je broj u decimalnom brojnom sistemu. Ova vrednost dodeljuje se atributu na levoj strani, a to jeste sintetizovani atribut. Pri tome smo dodelili neterminalu <more_digits> dva nova atributa: jedan predstavlja vrednost dela broja koji je njime predstavljen, a drugi predstavlja broj cifara. To znači da i sve druge pojave ovog neterminala moraju imati po dva atributa. Nazivamo ih onako kako nam odgovara.

Završili smo sa pravilima prve smene, jer su i ovo sintetizovani atributi, ne moramo ih računati pošto je <more_digits> na desnoj strani, već ćemo imati njihove vrednosti u prvoj smeni, računa se samo value.

Ostale su druga i treća smena i u njima smo uveli atribute za <more_digits>. U drugoj smeni, situacija sa ovom vrednošću v koju treba izračunati, znači vrednost tog dela broja koji je reprezentovan celom desnom stranom, je vrlo slična situaciji kako smo računali value u prvoj smeni. Uzećemo vrednost cifre val, pomnožiti sa težinskim koeficijentom koji je deset na broj preostalih cifara b1 i dodati ostatak broja v1. Šta još moramo računati? Moramo računati b pošto je to sintetizovani atribut i nalazi se na levoj strani. b je broj preostalih cifara na desnoj strani (b1) plus jedan, za ovaj digit. Time smo izračunali sve sintetizovane atribute na levoj strani, tako da smo završili sa drugom smenom.

Treća smena je još jednostavnija. Treba isto izračunati vrednosti v i b. Pošto je ovo smena koja završava izvođenje, opisuje broj koji se sastoji od nula cifara, znači broj cifara b je nula i vrednost

samog broja v je nula. Čitaocu se preporučuje da nacrta za neki konkretan ulaz, recimo 123 atributivno stablo izvođenja i odredi vrednosti svih atributa, da bi se videlo kako ovde, pošto su u pitanju sintetizovani atributi, tok vrednosti ide od listova ka korenu stabla.

Moguće je i drugačije rešenje koje pored nasleđenih koristi i sintetizovane atribute i na malo drugačiji način računa vrednost broja. Prvo rešenje računa tu vrednost cifru po cifru s desna na levo, na primer za 123: prvo smo imali 0 kad nema cifara, pa prva cifra 3, pa kada se doda cifra 2 s leve strane, to je 2*10+3=23, pa kada se doda 1 s leve strane, to je 1*100+23=123.

Međutim, moguće je takvu jednu konstantu računati s leva na desno: prvo imamo cifru najveće težine 1, pa sada kada dodajemo sledeću cifru 2, prvu cifru treba pomnožiti sa osnovom brojnog sistema i dodati sledeću cifru: 1*10+2=12, Kada dodajemo svaku sledeću cifru, prethodna vrednost se množi sa osnovom brojnog sistema 10 i dodaje sledeća cifra, znači 12*10+3=123, Kod računanja sa leva na desno nema stepenovanja. Kako se realizuje ovaj način računanja u gramatici?

1. $\langle integer \rangle_{value} \rightarrow digit_{val} \langle more\ digits \rangle_{i,s}$
   value ← s i ← val
   Napomena: i nasleđen, s sintetizovan

2. $\langle more\ digits \rangle_{i,s} \rightarrow digit_{val} \langle more\ digits \rangle_{i1,s1}$
   i1 ← 10i + val s ← s1

3. $\langle more\ digits \rangle_{i,s} \rightarrow \varepsilon$ s ← i

U postavci je zadato da postoji val atribut terminala digit i atribut value neterminala <integer>. Ono što je u ovom rešenju drugačije, to su atributi neterminala <more_digits>. Ideja je sledeća: u prvoj smeni nećemo još ništa računati, jer tu imamo samo jednu, najvišu cifru. Računanje treba da prenesemo do druge smene. To je u suštini prenos na niže kroz stablo izvođenja, pošto je <integer> u korenu, a <more_digits> je niže u stablu. Da bismo preneli vrednost prve cifre na niže, uvodimo jedan atribut i koji je nasleđen. Izabran je naziv i zbog engleskog termina inherited. Atributu i dodeljujemo vrednost prve cifre i na taj način to spuštamo u drugu smenu. Kasnije ćemo prokomentarisati drugi atribut s neterminala <more_digits>.

Šta imamo u drugoj smeni? Imamo preko i sa leve strane prenetu vrednost iz prve smene prve cifre i imamo val, to je vrednost sledeće cifre s desna. Treba izračunati vrednost koju predstavljaju te prve dve cifre tako što pomnožimo prenetu vrednost sa 10 i dodamo vrednost tekuće cifre val. Time dobijamo vrednost koju dalje prenosimo niže niz stablo. Ako broj ima više cifara, u stablu će biti više puta primenjena druga smena. U treću smenu preko i1 prenosimo vrednost koju predstavljaju prve dve cifre. Uvek se ta preneta vrednost množi sa 10 i dodaje se tekuća cifra i dalje se prenosi dobijena vrednost preko nasleđenog atributa. To prenošenje ide do kraja samog broja, kada se primenjuje treća, prazna smena.

U trećoj smeni imamo, pošto smo isprocesirali ceo broj da ovaj nasleđeni atribut i ima zahtevanu vrednost kompletnog broja, finalni rezultat. Međutim, od nas se traži da finalni rezultat bude pridružen startnom neterminalu, to jest korenu stabla, a epsilon je jedan od listova stabla. Imamo finalnu vrednost, ali ona je na potpuno drugoj strani stabla (u listu a ne u korenu). Šta onda možemo da uradimo? Možemo koristeći sintetizovane atribute tu vrednost samo da iskopiramo uz stablo, od lista ka korenu.

Kako to radimo? Tako što uvodimo drugi, sintetizovani atribut neterminalu <more_digits> i svim njegovim pojavama. Prvo u trećoj smeni prosto iskopiramo vrednost i u s. To isto ponovimo u drugoj smeni, ono što dolazi kroz s1 dalje će ka vrhu stabla biti kopirano kroz sintetizovani atribut s (računamo atribut na levoj strani, a koristimo na desnoj). Isto to se radi i u prvoj smeni, vrednost s koja dolazi od dole iz stabla dodeljujemo atributu value i tako smo konačno doveli finalnu vrednost do korena stabla. Opet je preporuka da se za jednu konkretnu konstantu 123 za vežbu nacrta kompletno stablo izvođenja i odrede vrednosti svih atributa, da bi se jasno videlo kako se prenose vrednosti.

Sintaksno-upravljano prevođenje od dna ka vrhu

Sledeći deo bavi se sintaksno-upravljanim prevođenjem od dna ka vrhu, to jest, načinom kako iskoristiti ove atributivno-translacione gramatike u parsiranju od dna ka vrhu i konkretno kako proširiti shift-reduce parser, koji je bio prepoznavač ulazne sekvence i davao je samo logičku vrednost ispravna/neispravna, da pored tog izlaza daje i nekakve bočne efekte u smislu izvršavanja akcija koje mi zadamo u okviru atributivno-translacione gramatike.

Da bismo mogli atributivno-translacionu gramatiku primeniti u bottom-up parsiranju, moramo uvesti neka dodatna ograničenja u samu gramatiku, van onih osnovnih pravila konzistencije:

1. Jedno dodatno ograničenje odnosi se na smeštaj akcionih simbola -- to ograničenje ima naziv poljska translaciona gramatika.
2. Drugo ograničenje je vezano za atribute gramatike. To drugo ograničenje se zove S-atributivna gramatika.

Poljska translaciona gramatika

Poljska translaciona gramatika zahteva da se akcioni simboli pojavljuju isključivo na samom kraju desne strane smene. Ako imate akcione simbole u smeni, onda desno od njih ne sme postojati nikakav drugi simbol koji nije akcioni.

Primeri

<X> → a {T} {V}	OK
<Y> → {R}	OK
<Z> → {T} y	NIJE OK

Dozvoljeno je recimo da imate dva akciona simbola na samom kraju desne strane, ili je dozvoljeno da ne bude drugih simbola, nego samo akcionih -- kada bi se uklonili ti akcioni simboli, to bi bila prazna smena. Međutim, ono što ne sme to je da neka druga vrsta simbola, terminalnih ili neterminalnih, bude iza akcionog.

Šta ako naša gramatika ne zadovoljava to ograničenje? Nije veliki problem, moguće je lako naći gramatiku koja opisuje iste nizove, a zadovoljava ograničenje. Uzmimo na primer da imamo neku smenu gde akcija nije na samom kraju, nego imamo nešto desno od akcije

<X> → A {y} B.

Tada postoje dve varijante transformacije.

U prvoj varijanti, ceo ovaj deo smene, prefiks desne strane zaključno sa akcijom odvajamo u novi neterminalni simbol koji do tada nije postojao u gramatici. U originalnoj smeni brišemo a{y} i uvodimo taj novi neterminal. Tako smo od ove jedne inicijalne smene koju smo uklonili, dobili dve u kojima vidimo da je zadovoljeno da se akcija pojavljuje na samom kraju desne strane.

<X> → <N> B
<N> → A {y}

U drugoj varijanti, možemo isto uvesti novi neterminal <M> koji se do tada nije pojavljivao u gramatici, ali njime zamenjujemo samo ovu akciju, na taj način je izvlačimo iz originalne smene. Originalnu smenu modifikujemo da umesto akcije pišemo taj novi neterminal <M> i onda dodajemo smenu za <M> koja sadrži na desnoj strani samo tu akciju {y}.

<X> → A <M> B
<M> → {y}

Ova druga varijanta transformacije koristi se automatski u cup alatu za generisanje parsera. To omogućava pri korišćenju alata da se gramatičke akcije stavljaju u sredinu smene, ali treba biti svestan toga da alat automatski radi ovu transformaciju.

Zašto je to bitno? Možete dobiti takav efekat, da recimo, krenete od gramatike koja nema nikakvih parserskih konflikata. Kada realizujete kompajler naknadno dodajete akcije. Ako dodajete akcije na same krajeve smena nećete nikako pokvariti gramatiku. Međutim, ako neoprezno u nekim situacijama dodate akciju u sredinu smene, može da se desi odjednom da se prijave parserski konflikti. To je zbog toga što se gramatika promenila, jer kada ovom transformacijom uklonite akciju iz sredine smene, dobija se jedan novi neterminal i nova prazna smena, tako da to može, zavisno od pozicije akcije i ostatka gramatike, da dovede do konflikata.

Kada se u stvari dešava konflikt? Kada ste ubacili akciju negde u sredinu desne strane neke smene, na poziciju koja još nije jednoznačno prepoznata od strane parsera. Ako još uvek parser ima nekoliko smena u opciji, onda ubacivanje akcije u jednu od tih smena može da izazove konflikt. Preporuka u takvim situacijama je ili da vi sami primenite ovu drugu vrstu transformacije koja daje malo drugačiji ishod, ili da prosto razmotrite da malo pomerite na desno akciju. Recimo ako je sakrijete iza nekog terminalnog simbola obično može da prođe bez konflikata.

S-atributivna gramatika

Drugo ograničenje koje se odnosi na atribute gramatike je, kao što smo već pomenuli. da gramatika koja treba da se procesira bottom-up parserom mora biti s-atributivna. Gramatika je s-atributivna ako su:

- svi atributi neterminalnih simbola sintetizovani. Zatim,
- nijedan sintetizovani atribut ne može da zavisi od drugog sintetizovanog atributa iste pojave simbola u gramatici i konačno,
- nasleđeni atributi akcionih simbola smeju da zavise samo od atributa onih simbola smene koji su levo od tog akcionog simbola.

Od svih ovih ograničenja najvažnije je ovo prvo, da kod neterminala ne možete uopšte koristiti nasleđene atribute. U praksi, sve situacije mogu da se reše isključivo sa sintetizovanim atributima, uz uvođenje dodatnih struktura podataka (kolekcija) mogu da se razreše situacije kada bi vam normalno trebali nasleđeni atributi. Ovo drugo ograničenje potpuno je sporednog značaja. Ako

recimo imate na levoj strani neke smene simbol koji ima dva atributa a i b, to drugo ograničenje kaže da a ne sme zavisiti od b niti b od a, što se u normalnom slučaju i ne koristi -- a i b će zavisiti od atributa sa desne strane te smene. Nasleđeni atributi jedino mogu u akcionim simbolima da se pojave i za njih onda važi ranije navedeno ograničenje.

Primer

Jedan konkretan primer gramatike koja zadovoljava ova ograničenja dat je u nastavku. Osim što možemo proveriti da su ograničenja zadovoljena, iskoristićemo ovaj primer da objasnimo princip proširenja jednog parsera od dna ka vrhu da obavlja atributivnu translaciju specificiranu ovakvom gramatikom.

Ova gramatika ima tri smene, ima samo neterminal <E>. Osim što je zadata gramatika, ostatak postavke je da data gramatika prevodi izraz u niz sastavljen od atoma ADDp,q,r i MULTp,q,r gde su vrednosti atributa p,q i r respektivno ukazatelji na levi operand, desni operand i rezultat operacije u tabeli simbola, što je nekakav ekvivalent asemblerskom kodu. Novi ulaz u tabelu simbola generišemo pozivom funkcije NEWT. Atribut tokena I je ukazatelj na tabelu simbola na simbol koji odgovara tokenu I.

1. $\langle E\rangle_p \rightarrow (\langle E\rangle_q + \langle E\rangle_r)$ $\{ADD\}_{s1,s2,s3}$
   $s1 \leftarrow q,$ $s2 \leftarrow r,$ $p,s3 \leftarrow NEWT$
2. $\langle E\rangle_p \rightarrow (\langle E\rangle_q * \langle E\rangle_r)$ $\{MULT\}_{s1,s2,s3}$
   $s2 \leftarrow q,$ $s2 \leftarrow r,$ $p,s3 \leftarrow NEWT$
3. $\langle E\rangle_p \rightarrow I_r$ $p \leftarrow r$

Da vidimo kako treba da funkcioniše parser napisan na bazi ove gramatike. Izrazi koje opisuje ova gramatika imaju specifičnu sintaksu da imamo + i *, ali su zagrade uvek obavezne, čak i tamo gde nisu neophodne. Na ulazu ako dođe ulazni niz

(1+(a*2)).

Pošto sama gramatika što se tiče operanda ima samo I, koje opisuje i konstante i promenljive, tokenizator bi dao ovakav ulaz parseru

(I_1 + (I_2 * I_3)) -|

Ovi 1, 2 i 3 su redni ulazi u tabeli simbola gde su opisani ovi operandi. Recimo, prvi ulaz tabele simbola opisuje konstantu 1. Drugi ulaz opisuje promenljivu a i treći ulaz u tabeli simbola opet opisuje konstantu vrednosti 2.

RB	Naziv	Tip
1	1	C
2	a	V
3	2	C

Ovo je sve ulaz u parser, a šta treba da bude finalni izlaz parsera prema ovoj gramatici? Finalni izlaz možemo videti u nastavku.

Konačni izlaz:
{MULT 2,3, 4}
{ADD 1,4, 5}

Konačna tabela simbola:

RB	Naziv	Tip
1	1	C
2	a	V
3	2	C
4	t_1	T
5	t_2	T

Parser izraz (1+(a*2)) treba da prevede u neku vrstu akcione sevence koja u nekoj meri podseća na asemblerski program koji kaže prvo treba pomnožiti operande koji su opisani drugim i trećim ulazom tabele simbola, a to su promenljiva a i konstanta 2 i međurezultat se stavlja na mesto koje je opisano u četvrtom ulazu tabele simbola.

U četvrtom ulazu tabele simbola vidimo nekakvu promenljivu koja inače nije postojala u originalnom ulazu. Takve vrste promenljivih, koje kompajler interno uvede se zovu privremene promenljive (temporaries) i ima neko svoje ime, t1 u ovom primeru. Vidimo da je parser dodao ova dva nova ulaza tabele simbola označena žutom bojom. Svaki od tih ulaza je dobijen pozivom funkcije NEWT. Poziv funkcie NEWT pravi novu privremenu promenljivu sa jedinstvenim imenom i stavi je negde u tabelu simbola. To znači, rezultat akcije MULT, taj međurezultat se čuva u t1.

Onda ide druga akcija ADD, koja kaže da treba operand opisan u prvom ulazu tabele simbola (to je konstanta 1), sabrati sa operandom opisanom u četvrom ulazu tabele simbola, to je ova privremena promenljiva t1 i rezultat smestiti u odredište koje je opisano petim ulazom tabele simbola. To je opet nova privremena promenljiva t2.

Dakle sve ovo je obrada koju očekujemo da uradi parser na bazi ove gramatike: da emituje ove dve akcije, da alocira ova dva nova ulaza tabele simbola i da zadati izraz isprevodi u niz ovih akcija. Ovo je kao jedan kompajler u malom, jer u suštini posao kompajlera je upravo to -- on ne računa vrednost izraza, nego emituje mašinski program. Izlaz iz primera možete shvatiti kao neku vrstu apstrahovanog mašinskog programa koji opisuje kako će se u run-time-u računati vrednost izraza.

Videli smo šta se očekuje u finalnom izlazu. Sada da vidimo kako proširiti potisni prepoznavač da bi obavljao traženu atributivnu-translaciju:

- u implementaciji treba predvideti da simboli steka sadrže i polja za atribute, znači uvesti za svaki simbol steka nekakav zapis koji ima onoliko dodatnih polja koliko ima vrednosti atributa. Vrednosti svih atributa će biti poznate u trenutku kada parser u toku rada stavlja neki simbol na stek.
- Ostatak parsera treba izmeniti samo utoliko što se proširuju akcije redukcije smene, sve drugo je isto. Akcije reduce za svaku smenu proširujemo računanjem svih atributivnih pravila te smene i emitujemo akcione simbole navedene u toj smeni na izlaz.

Evo za ovaj primer, parser nije veliki, ne prikazujemo potisnu i kontrolnu tabelu. Prikazujemo samo deo parsera koji se menja a to su ove proširene REDUCE akcije i kako sada izgledaju zapisi za gramatičke simbole.

Proširene REDUCE akcije za tekući primer:

REDUCE(1):

BEFORE
)
<E>
vredn. r
+
<E>
vredn. q
(
...

AFTER
<E>
NEWT
....
∇

OUTPUT{$ADD_{q,r,NEWT}$}

REDUCE(3) :

BEFORE
I
vredn. r
...
∇

AFTER
<E>
vredn. r
...
∇

REDUCE(2) je kao i REDUCE(1) samo umesto "+" stoji "*", a akcija je OUTPUT{$MULT_{q,r,NEWT}$}.

U objašnjenu proširenja, prvo čemo da se skoncentrišemo na pojedine gramatičke simbole. U zadatoj gramatici, neterminal <E> ima jedan atribut, a akcioni simboli imaju po tri atributa, dok terminali osim I nemaju atributa. Terminal I koji predstavlja operand ima jedan atribut. U proširenom parseru, <E> je reprezentovano ovim zapisom gde prva komponenta označava identitet stek simbola, a druga komponenta, razdvojena crtkastom linijom, predstavlja vrednost njegovog odgovarajućeg atributa. Takođe, ako se negde pojavljuje I kao stek simbol, imaćemo komponentu koja određuje koji je simbol u pitanju i vrednost njegovog atributa. Drugi stek simboli predstavljaju samo svoj identitet, nemaju atribute.

Proširene REDUCE akcije ovde su prikazane na jedan specifičan način, vizuelno, stanjem parserskog steka pre nego što se sprovede akcija i stanjem steka pošto se sprovede akcija i dodatnim opisom šta ide na izlaz, to je prikazano kao output.

Reduce(1) akcija, da se podsetimo gramatike, prva smena ima na levoj strani <E> na desnoj strani (<E> + <E>) {ADD}. Šta se dešava u okviru redukcije te prve smene? Kao što znamo, redukcija smene očekuje da će na vrhu steka zateći redom simbole steka (<E> + <E>), gde je) na vrhu steka pa ispod redom idu drugi simboli sa desna na levo. Akcioni simboli ne idu na stek.

Šta treba Reduce akcija da uradi? Svih ovih 5 simbola se skida sa steka, zatim potrebno je izračunati tri atributa akcije {ADD}, emitovati sa output akciju na izlaz (u programskoj realizaciji to može da znači zapis u neki izlazni fajl, ili čak može da znači da je ovo add neki poseban metod koji će biti pozvan u tom trenutku sa stvarnim parametrima s1, s2, s3). Takođe biće izračunata vrednost p pozivom NEWT funkcije i posle toga će <E> sa ovom vrednošću biti stavljen na stek. Primetiti da se vrednost koja se dobije pozivom NEWT dodeli istovremeno atributima p i s3. Znači ovde nema dva različita poziva NEWT nego samo jedan. Sve je to vizuelno predstavljeno na slici. Stek je prikazan tako da je vrh steka gore, dno steka dole.

Reduce akcija druge smene skoro je identična reduce(1) akciji, samo što umesto + stoji * na steku i takođe u izlazu umesto {ADD} stoji {MULT}. Treća smena je jednostavnija, na desnoj strani je samo I, pri redukciji skine se I, stavlja se <E> i kopira se vrednost atributa I u p.

Ovde je konstruisan trejs rada ovoga parsera za onaj primer ulaza (I1+(I2*I3)). Parser će obaviti rad u 15 koraka.

	Stek	Tekući ulaz	Akcija	Izlaz
1.	∇	(I_1 + (I_2 * I_3)) -|	SHIFT	
2.	∇ (	I_1 + (I_2 * I_3)) -|	SHIFT	
3.	∇ (I_1	+ (I_2 * I_3)) -|	REDUCE(3)	
4.	∇ (<E>$_1$	+ (I_2 * I_3)) -|	SHIFT	
5.	∇ (<E>$_1$ +	(I_2 * I_3)) -|	SHIFT	
6.	∇ (<E>$_1$ + (	I_2 * I_3)) -|	SHIFT	
7.	∇ (<E>$_1$ + (I_2	* I_3)) -|	REDUCE(3)	
8.	∇ (<E>$_1$ + (<E>$_2$	* I_3)) -|	SHIFT	
9.	∇ (<E>$_1$ + (<E>$_2$ *	I_3)) -|	SHIFT	
10.	∇ (<E>$_1$ + (<E>$_2$ * I_3	)) -|	REDUCE(3)	
11.	∇ (<E>$_1$ + (<E>$_2$ * <E>$_3$	)) -|	SHIFT	
12.	∇ (<E>$_1$ + (<E>$_2$ * <E>$_3$)	) -|	REDUCE(2)	{$MUL_{2,3,4}$}
13.	∇ (<E>$_1$ + <E>$_4$	) -|	SHIFT	
14.	∇ (<E>$_1$ + <E>$_4$)	-|	REDUCE(1)	{$ADD_{1,4,5}$}
15.	∇ <E>$_5$	-|	ACCEPT	

Radiće shift-ovanje simbola sa ulaza sve dok ne formira desnu stranu neke smene. Prvi put će se to desiti kada na vrh steka potisne konstantu I1, onda će prepoznati da može to redukuje primenom 3. smene u <E>$_1$. Nastaviće shift-ovanje sve dok ne dođe do I_2 na vrhu steka, kada će opet primenom treće smene to pretvoriti u <E>$_2$. Slično tome će i I_3 pretvoriti u <E>$_3$. Kada bude prvi put shift-ovao desnu zagradu, prepoznaće i zameniti (<E>$_2$*<E>$_3$) redukcijom druge smene sa jednim <E> i dodeliće pozivom NEWT novi temporary a na izlaz će emitovati prvu akciju {MUL}. Kada dođe ova druga zagrada i kada se formira (<E>$_1$+<E>$_4$), prepoznaće prvu smenu, redukovati je, emitovaće ovu akciju {ADD}, interno će pozvati NEWT i dobiti vrednost 5 i kada na vrhu steka mu se nađe samo startni simbol a na ulazu kraj ulaza, uradiće akciju ACCEPT.

Izlaz će biti sekvenca je ispravna i prihvata se, a kao bočni efekat će biti emitovane ove dve akcije. To odgovara slici konačnog izlaza sa strane 139.

Sintaksno-upravljano prevođenje od vrha ka dnu

Kako proširiti parser od vrha ka dnu koji je prepoznavač, koji samo određuje da li je sekvenca ispravna ili ne, da obavlja atributivnu translaciju specificiranu nekom atributivno-translacionom gramatikom? Evo polaznog primera koji pomaže da se na neki logičan način zapamte pravila koje ćemo posle formalno definisati.

Ako imamo neku smenu <A> → <B><C> koja ima na levoj strani neki neterminal <A>, a desna strana na primer ima neke gramatičke simbole, ovde su zadati <B> i <C>, onda treba zapamtiti pravilo da nasleđeni atributi nekog gramatičkog simbola određuju u njegovoj proceduri rekurzivnog spusta ulazne parametre, a sintetizovani atributi su izlazni parametri procedure. Znači ako se ta logika zapamti, onda sve drugo logično proizilazi iz toga.

Mi znamo da rekurzivni parser, kada obrađuje ovu smenu, prvo u runtime-u ulazi u proceduru A da bi prepoznao njenu desnu stranu, a to znači, pošto su nasleđeni atributi A ulazni parametri, oni su već poznati -- njih je definisao pozivalac procedura A.

Zatim, da bi se pri prepoznavanju isprocesiralo <B>, pozvaće se procedura B, ali pre tog poziva moraju se izračunati nasleđeni atributi <B> kao ulazni parametri za tu proceduru. Znači to je sledeće što se računa. Zatim dolazi do poziva procedure B, pa se unutar toga računaju izlazni parametri B, toj jest sintetizovani atributi <B>, znači to je sledeće što se računa.

Zatim, kada se kontrola vrati iz B, moraju se izračunati ulazni parametri za C to jest nasleđeni atributi neterminala <C>. Posle toga, ide taj poziv procedure C gde se računaju sintetizovani atributi <C>. Konačno kad se kontrola vrati iz C, šta je poslednje što se računa? Izlazni parametri procedure A (to jest sintetizovani atributi neterminala <A>) i posle toga se dešava povratak iz procedure A.

Znači upravo smo definisali redosled računanja svih vrsta atributa pomenutih simbola. Znamo čak, za sve tipove atributa, od kojih drugih atributa mogu da zavise, recimo, sintetizovani atributi <A> pošto se računaju poslednji, mogu da zavise od bilo kojih od ovih prethodno pomenutih atributa: nasleđenih od <A>, nasleđenih od <B>, sintetizovanih od <B>, nasleđenih od <C> i sintetizovanih od <C>.

Dok, recimo, nasleđeni atributi od <B>, pošto se oni računaju odmah pri pozivu A, oni mogu da zavise samo od nasledjenih atributa od <A>. Generalno pravilo je da atributi mogu zavisiti od onih koji su levo od njih u smeni.

Ova pravila, koja su ovde neformalno objašnjena na primeru, formalizuju se kroz pojam L atributivnih gramatika. Uslovi koje mora gramatika da zadovoljava da bi bila L atributivna, je da:

- nasleđeni atributi simbola sa desne strane smene mogu da zavise od nasledjenog atributa sa leve strane i od bilo koga atributa bilo kog simbola, koji je na desnoj strani ispred posmatranog simbola. Zatim,
- sintetizovani atribut simbola sa leve strane smene može da zavisi od nasledjenih atributa tog simbola i od bilo koga atributa bilo kog simbola sa desne strane smene. I konačno,
- sintetizani atribut akcionog simbola sme da zavisi isključivo od nasledjenih atributa istog akcionog simbola.

Prva dva pravila ilustrovana su u prethodnom primeru. Jedino treba dodatno zapamtiti poslednje pravilo koje se tiče akcionih simbola, znači njegov sintetizovani atribut sme da zavisi isključivo od njegovih nasleđenih atributa. To je najlakše zapamtiti po tome da akcioni simbol možete u implementaciji shvatiti kao poseban potprogram, pa tada njegovi nasleđeni atributi odgovaraju ulaznim parametrima tog potprograma, a njegovi sintetizovani atributi izlaznim parametrima. Tako da je logično da izlazni parametri zavise isključivo od ulaznih.

Primer

Razmotrimo sada kratak primer koji dodatno ilustruje uslove L atributivnosti. Sledeće produkcije su deo neke atributivne gramatike. U svakoj od njih, *p*, *q* i *r* su nasleđeni atributi, a *s* i *t* sintetizovani. Za svaku od produkcija pronaći od kojih sve atributa mogu da zavise atributi *p* i *r* da bi produkcija bila L-atributivna.

a) $<S>_{s,q} \rightarrow a_u <S>_{t,p} <A>_r$

b) $<S>_{s,q} \rightarrow \{c_p\}\ b_u <S>_{t,r}$

c) $<S>_{s,q} \rightarrow c_u <A>_r <S>_{t,p}$

Ako razmotrimo ovo prvo pravilo, šta možemo reći? Atribut p je nasleđeni atribut. On se računa naravno kada je na desnoj strani smene, to je u redu i rekli smo da on može da zavisi od nasleđenog atributa simbola na levoj strani, a to je samo q i od atributa simbola na desnoj strani koji je levo od njegovog simbola, to je terminal a i njegov atribut u. Znači p može biti funkcija od q i u.

Što se tiče atributa r, znači to je takođe nasleđeni atribut na desnoj strani i može da zavisi od nasleđenog na levoj strani q i od atributa svih simbola koji su na desnoj strani ispred <A>, a to su znači q, t i p. Konačno, nije se tražilou postavci, ali možemo reći da ovde treba definisati i sintetizovani atribut simbola na levoj strani s, a on može da zavisi od svih drugih atributa jer se on poslednji računa.

a) $<S>_{s,q} \rightarrow a_{t,r}<S>_{t,p}<A>_r$

$p = f_1(q, u), \qquad r = f_2(q, u, t, p) \quad s=f_3(q, u, t, p,r)$

Slično tome, u tački b, vidimo da p može da zavisi od nasledjenog q na levoj strani. Atribut r može zavisiti od q i od ovih koji su znači levo od <S> na desnoj strani, b, i tako dalje.

b) $<S>_{s,q} \rightarrow \{c_p\}\ b_u <S>_{t,r}$

$p = f_1(q), \qquad r = f_2(q, p, u)$

U tački c) rešenje se dobija na sličan način.

c) $<S>_{s,q} \rightarrow c_u <A>_r <S>_{t,p}$

$p = f_1(q, u, r) \qquad r = f_2(q, u)$

Optimizacija pravila preimenovanjem atributa

Kako proširiti prepoznavač da obavlja sintaksno-upravljano prevođenje u slučaju da su zadovoljeni uslovi L atributivnosti i da sama gramatika što se tiče selekcionih skupova smena, zadovoljava LL(1) uslov? Ako je sve to zadovoljeno, onda je moguće konstruisati procesor za takvu gramatiku.

Osnovna ideja je da nasleđene atribute prenosimo po vrednosti kao ulazne parametre potprograma, sintetizovane atribute po adresi kao izlazne parametre, jer kod njih može posle pozivalac da koristi njihove promenjene vrednosti. Svi atributi koji se pominju na desnoj strani neke smene postaće lokalne promenljive. Svi atributi istog simbola sa leve strane moraju imati isto ime u svim

njegovim smenama, jer to sada postaju formalni parametri. I poslednja tačka je jedna mala optimizacija koja može da se sprovede, ali ako se ne sprovede nije strašno, a to je da se neko pravilo prostog kopiranja vrednosti x ← y (da x dobije vrednost od y) može u potpunosti eliminisati preimenovanjem x u y ili obrnuto. Prilikom preimenovanja atributa, ne smeju se menjati imena atributa sa leve strane koje smo prethodnim pravilom doveli na isto.

Primer

Evo jednog primera gde ćemo ilustrovati sve navedeno i pokazati kako izgleda odgovarajući procesor. Vidimo da gramatika ima četiri smene, dve za <S>, dve za <A>. Ima i neke akcione simbole i ima neka atributivna pravila. Prvo je potrebno proveriti da li je u pitanju LL(1) gramatika, proverom da li su selekcioni skupovi smena sa istom levom stranom disjunktni. Vidimo da prva i druga smena imaju različite selekcione skupove, jedna počinje sa a, druga sa b. Isto važi i za treću i četvrtu smenu, tako da gramatika jeste LL(1).

1. $<S>_q \rightarrow a_{x1} <A>_{x,y,z} <S>_{q1} \{b_w\}$
   $w \leftarrow y * z + q1 \quad x \leftarrow x1 \quad q \leftarrow q1$
2. $<S>_{r1} \rightarrow b\ a_y \{c_w\} <A>_{u,r,x} \{b_v\}$
   $w \leftarrow y + 3 \quad r1 \leftarrow r \quad v \leftarrow r * x - w \quad u \leftarrow w + 2$
3. $<A>_{x,y,z1} \rightarrow a_q \{c_v\} <A>_{x1,z,u} <A>_{u,t,y1}$
   $v \leftarrow x - q \quad x1 \leftarrow x \quad y \leftarrow y1 \quad z1 \leftarrow z \quad u \leftarrow 3$
4. $<A>_{t,s,s1} \rightarrow b\ a_q \{b_{t1}\}\ a_{s2}$
   $t1 \leftarrow t \quad (s,s1) \leftarrow s2$

Što se tiče uslova L atributivnosti, trebalo bi ustanoviti koje su vrste atributa i proveriti da li su pravila zadovoljena. Evo mi možemo za prvu smenu pogledati. Vidimo da w se računa, to je znači nasleđeni atribut akcionog simbola i on u principu može da zavisi od bilo kojeg od ovih atributa osim od q koji je sintetizovan na levoj strani. Sme da zavisi od ovih atributa koji su na desnoj strani levo od akcije {b} i to su svi ovi drugi atributi na desnoj strani. tako da je pravilo za w u redu.

Sledeće pravilo je za atribut x, koji je očigledno nasleđeni atribut od <A> na desnoj strani prve smene i on zavisi od x1, a x1 je atribut terminalnog simbola za koji znamo da je u tom trenutku prepoznavanja smene dat i njega daje leksički analizator, tako da je ovo pravilo u redu (x zavisi od simbola levo od sebe kao nasleđeni i to je u redu).

Konačno pravilo dodele u prvoj smeni je za q. q je sintetizovani atribut, pošto se on poslednji računa, on može znači da zavisi bez problema od bilo kojeg od ovih atributa. Znači za prvu smenu zadovoljeno je pravilo L atributivnosti. Isto tako može da se proveri za sve ove druge smene.

Sintaksno-upravljano prevođenje u metodu rekurzivnog spusta

Kako konstruišemo procesor? Prvo ćemo uraditi preimenovanje atributa. Ono što svakako moramo da uradimo, to je da dovedemo na ista imena atribute ovih simbola na levim stranama smena. Znači za atribute <S> u prvoj i drugoj smeni, to je urađeno tako što je r1 preimenovano u q. Takođe i u trećoj i četvrtoj smeni atributi od <A> moraju imati ista imena. vidimo da su preuzeta imena iz treće smena i za četvrtu.

Dodatno možemo eliminisati pravila prostog kopiranja vrednosti. Vidimo da recimo u prvoj smeni imamo dva takva, x ← x1 i q ← q1. Preimenovanjem x1 u x i q1 u q, prosto su ta pravila uklonjena i ostalo je samo jedno pravilo prve smene koje nije prosto kopiranje. Znači ovo je jedna mala optimizacija da u kodu ne moramo ni da imamo eksplicitnu dodelu.

1. $<S>_q \rightarrow a_x <A>_{x,y,z} <S>_q \{b_w\}$ (x1 postalo x, q1 postalo q)
   $w \leftarrow y * z + q$
2. $<S>_q \rightarrow b\ a_y \{c_w\} <A>_{u,r,x} \{b_v\}$ (r1 postalo q, r postalo q)
   $w \leftarrow y + 3 \quad v \leftarrow q * x - w \quad u \leftarrow w + 2$
3. $<A>_{x,y,z1} \rightarrow a_q \{c_v\} <A>_{x,z1,u} <A>_{u,t,y}$ (x1 postalo x, z postalo z1, y1 postalo y)
   $v \leftarrow x - q \quad u \leftarrow 3$
4. $<A>_{x,y,z1} \rightarrow b\ a_q \{b_x\}\ a_{s2}$ (t i t1 postali x, s postalo y, s1 postalo z1)
   $(y,z1) \leftarrow s2$

Sve te optimizacije su urađene u sve četiri smene i sad imamo ovu varijantu gramatike, na bazi koje direktno pišemo parser po principu rekurzivnog spusta (primer je napisan u pseudokodu sa sintaksom nalik Paskalu). Ovde se promenljiva koja predstavlja tekući ulazni simbol zove U i ona sad nije celobrojna nego, pošto ulazni token ima i klasu i vrednost, to je jedan zapis, u onom paskalskom stilu. Jedna komponenta zapisa je klasni deo ime, a druga komponenta je njegov atribut neka celobrojna vrednost i postoji procedura advance koja mora da, svaki put kad se pozove sledeći ulazni simbol, zapiše njegovu klasu i da odredi vrednost njegovog atributa.

```
Program Parser;
  type ulaz =  record of
                                ime:char;
                                atrib: integer;
               end;
  var q: integer;
      u: ulaz;

procedure ADVANCE;
          u.ime = sledeći ulazni simbol;
          u.atrib = njegov atribut;
    end;

procedure ACCEPT;
    begin
    kraj rada sa prihvatanjem ulaza
    end;

procedure REJECT;
    begin
    kraj rada sa neprihvatanjem ulaza
    end;
```

```
procedure S (var q :integer)
    var x,y,z,w,u,v :integer;
    begin
      case u.ime of
            'a':    x := u.atrib;
                              ADVANCE;
                              A(x,y,z);
                              S(q);
                              w := y * z + q;
                              out (b,w);
            'b':    ADVANCE;
                              if u.ime <> 'a'
                              then REJECT;
                              y := u.atrib;
                              ADVANCE;
                              w := y + 3;
                              out ('c', w);
                              u := w + 2;
                              A(u,q,x);
                              v := q * x - w;
                              out ('b', v);
           default:REJECT;
      end; {od "case"}
    end; {od "begin"}

procedure A(x:integer,var y,z1:integer)
    var q,v,u,s2, t:integer;
    begin
      case u.ime of
            'a':    q:=u.atrib;
                              ADVANCE;
                              v:=x-q;
                              u:=3;
                              out ('c', v);
                              A(x,z1,u);
                              A(u,t,y);
            'b':    ADVANCE;
                              if u.ime <> 'a' then
                                   REJECT;
                              q:=u.atrib;
                              ADVANCE;
                              out ('b', x);
                              if u.ime <>'a' then
                                   REJECT;
                              s2:=u.atrib;
                              ADVANCE;
                              y:=s2;
```

```
                                z1:=s2;
            default:REJECT;
      end;
    end;

begin {main}
  ADVANCE;
  S(q);
  if u.ime=ENDMARKER then ACCEPT
  else REJECT;
end.
```

Takođe treba da imamo procedure accept i reject. Sada ćemo objasniti najvažnije delove ovog rekurzivnog parsera. Moramo imati proceduru S, proceduru A i glavni program. Možemo krenuti od glavnog programa. U glavnom programu moramo inicijalizovati promenljivu U pozivom Advance procedure, pozvati proceduru S(q) za startni neterminal, q je izlazni parametar, znači njega posle možemo koristiti, na primer ispisati vrednost. On je na nivou glavnog programa kao globalna promenljiva definisan i posle ovog poziva možemo da ga koristimo. Potrebno je, kada se vratimo iz S(q), ispitati da li je ulaz došao do end markera, onda je accept, inače je reject.

Što se tiče procedure S, ona treba da implementira prepoznavanje ove prve dve smene. S ima sintetizovati atribut koji predstavlja izlazni parametar, to jest prenos po referenci. U paskalu je to parametar za koji piše var, a u nekim drugim programima koristili biste nešto drugo, u javi biste koristili omotače za primitivne tipove kao što je Integer, ili reference, u Cu biste koristili pokazivače i tako dalje. Promenljiva q je znači taj izlazni parametar, a ovi ostali atributi koji se pojavljuju na desnoj strani prve i druge smene, to su x,y, z, w, r i u moraju postati lokalne promenjive u proceduri S. Konačno procedura S ima takvu strukturu da ima case na bazi tekućeg ulaznog simbola, ako je 'a' prepoznaje se prva smena ako je 'b', prepoznaje se druga smena.

Kod prepoznavanja prve smene, pošto smo ovde već uparili 'a' u case, možemo pozvati samo advance i posle toga pozvati proceduru A i proceduru S. Naravno, potrebno je izračunati atribute, ako je bilo šta potrebno prethodno izračunati, znači Imamo taj advance, međutim, pošto smo ovde izvršili preimenovanja, znači nemamo nikakva računanja da ubacujemo nego direktno zove moja zovemo A, zovemo S. Sada treba izračunati w, jer se on koristi u akciji {b}. Akcija {b} ovde je implementirana kao poziv neke procedure out() koja kao stvarne parametre ima identifikator akcionog simbola koji se izdaje ne izlaz i vrednost njegovog atributa.

Uslovi L atributivnosti će vam garantovati da uvek imate mesto gde ćete ubaciti kod za računanje vrednosti atributa. Najkasnije mesto za računanje vrednosti je neposredno pre nego što se ta vrednost upotrebi. Pošto je ovde to bio nasleđeni atribut v koji je ulazni parametar za out, ovo je bilo poslednje mesto gde to mora da se računa, a ujedno i najranije mesto, pošto on koristi q koje se vraća iz S. Za razliku od ove situacije, nekada postoji više mogućih mesta u kodu gde se računanje vrednosti atributa može staviti.

Apstraktna sintaksna stabla

Premda bi se mnogi zadaci kompajlera mogli izvršiti u jednoj fazi putem sintaksno-upravljanog prevođenja zasnovanog na atributivno-translacionim gramatikama, savremene softverske prakse obeshrabruju implementaciju većeg broja različitih funkcionalnosti u jednoj komponenti kao što je parser.

Zadaci kao što su semantička analiza, konstrukcija tabele simbola, optimizacija programa i generisanje koda zaslužuju zaseban tretman u kompajleru. Ako se svi ti zadaci smeste na jedno mesto, u specifikaciju semantičkih akcija gramatike, rezultujući kompajler je teško razumeti, proširiti i održavati. Zato razmatramo dizajn i primenu strukture podataka poznate kao apstraktno sintaksno stablo, koja će služiti kao centralna struktura podataka za sve aktivnosti posle parsiranja. Cilj sintaksno-upravljanog prevođenja se tada svodi samo na izgradnju tog apstraktnog sintaksnog stabla.

Najčešće se u procesiranju apstraktnog sintaksnog stabla koristi projektni šablon posetilac (engl. visitor). Taj šablon obezbeđuje efikasan obilazak stabla i razdvajanje algoritama od strukture nad kojom se operiše. Nove vrste procesiranja, različiti kompajlerski prolazi, mogu se dodavati bez menjanja postojećih klasa koje definišu strukturu dodavanjem novih posetilaca.

Parser generisan alatom CUP

Uzećemo jedan konkretan tekući primer da ilustrujemo tehniku apstraktnih sintaksnih stabala i uporedimo je sa tehnikom sa atributivnom translacijom. Radi se o jednostavnom primeru kalkulatora za osnovne celobrojne aritmetičke izraze sa osnovnim operacijama sabiranja, oduzimanja, množenja i deljenja. Gramatika koju ćemo koristiti prilagođena je parserima od dna ka vrhu. Imamo neterminale <Expr>, <Term> i <Factor>, Ceo izraz opisan je smenama za <Expr>. Prva smena se koristi ako se koriste operacije sabiranja i oduzimanja koje su preko ovoga neterminala <addop> zajednički predstavljene, a ako nema sabiranja/oduzimanja onda se <Expr> menja sa <Term>, a <Term> u izraze ubacuje operacije množenja i deljenja. Naravno, ako nema ni tih operacija, onda se <Term> direktno predstavlja <Factor>-om, a <Factor> je konstanta, odnosno terminalni simbol NUMBER.

<Expr> → <Expr> <addop> <Term>
<Expr> → <Term>
<Term> → <Term> <mulop> <Factor>
<Term> → <Factor>
<Factor> → NUMBER
<addop> → PLUS | MINUS
<mulop> → STAR | DIVIDE

Prvo ćemo videti implementaciju čistog prepoznavača, koji samo prepoznaje sintaksu izraza a ne radi nikakvo procesiranje. Dovoljna su tri fajla. Prvi fajl je specifikacija leksičkog analizatora, ima ekstenziju .lex. Sve se stavlja u ovoj varijanti primera u paket parser. import je potreban da bi se skener razumeo sa parserom. Od parsera dobija klasu Symbol koja reprezentuje tokene. Ovaj skener će da komunicira i generisaće se interfejs da može da prosleđuje tokene cup generisanom parseru. Ovde je niz regularnih izraza i kako se koji prepozna, vraća se odgovarajući simbol, znači

ovo je klasni deo tokena. Vidimo da skener ignoriše razmake. Konačno, ako ne upali nijedno od ovih prethodnih pravila, podrazumevano pravilo za neprepoznati simbol na ulazu je da se na izlazu ispiše odgovarajuća poruka o grešci.

```
package parser;
import java_cup.runtime.Symbol;
%%
%cup
%%
"+" { return new Symbol(sym.PLUS); }
"-" { return new Symbol(sym.MINUS); }
"*" { return new Symbol(sym.STAR); }
"/" { return new Symbol(sym.DIVIDE); }
[0-9]+ { return new Symbol(sym.NUMBER); }
[ \t\r\n\f] { /* ignore white space. */ }
. { System.err.println("Illegal character: "+yytext()); }
```

Fajl sa ekstenzijom .cup je specifikacija sintaksnog analizatora i on će biti procesiran od strane alata cup, generatora parsera. Imamo deklaraciju paketa gde će se smestiti svi fajlovi. Specifikacija je manje-više pravolinijska. Znači deklariše se koji su gramatički terminali, koji su gramatički neterminali i onda ide niz smena. Leva strana smene je od desne razdvojena sa ::=, a ako imamo više smena sa istom levom stranom, da ne bismo stalno pisali levu stranu, možemo kao skraćenicu staviti uspravnu crtu. Ova druga linija je smena <Expr> → <Expr> <addop> <Term>. Niz tih specifikacija smena sa istom levom stranom završava sa znakom ;. I tako vidimo da je ona naša polazna gramatika predstavljena u cup specifikaciji.

```
package parser;
terminal NUMBER;
terminal PLUS, MINUS, STAR, DIVIDE;
nonterminal expr, term, factor;
nonterminal addop, mulop;
expr ::= term
| expr addop term;
term ::= factor
| term mulop factor;
factor ::= NUMBER;
addop ::= PLUS  | MINUS ;
mulop ::= STAR  | DIVIDE ;
```

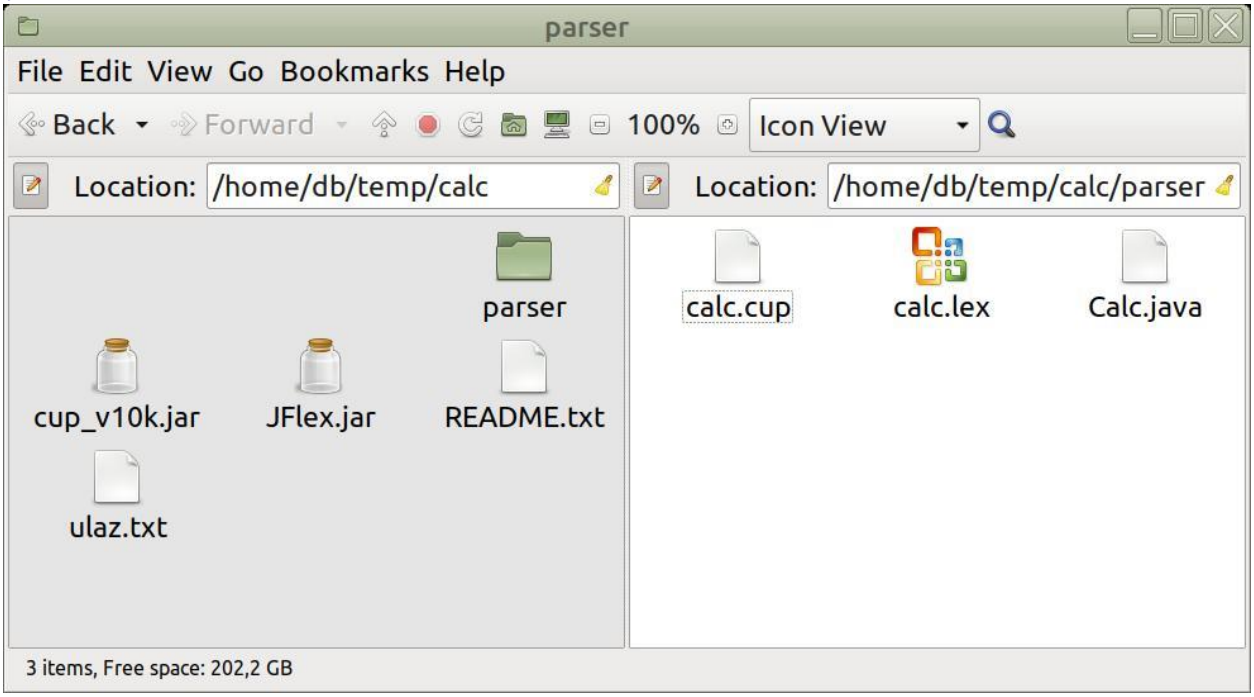
Nadalje imamo java klasu Calc, koja sadrži glavni program. Ona je isto u paketu parser, mora da importuje cup runtime klase, tako da i kada se pokreće prevedeni program mora u classpath putanji postojati odgovarajući cup_v10k.jar fajl. Pošto ćemo nešto ispisivati na izlazu, možda ne u ovom primeru ali kasnije, imamo i ovaj drugi import iz io javinog paketa. Glavni program instancira FileReader nad ulaznim argumentom. Programu ćemo zadavati ulazni fajl iz koga čita izraz. Zatim će da instancira skener koji će jflex da napravi u obliku ove yylex klase i njemu se prosleđuje ovaj instancirani FileReader. Zatim se instancira parser, to je isto tako automatski generisana klasa alatom cup i njemu se prosleđuje instancirani skener. Zatim je potrebno aktivirati parser tako što nad objektom p pozovemo metod parse() ili debug_parse(). U ovom prvom primeru pozvan je debug_parse().

Evo kako izgleda struktura fajlova na disku. Na sajtu kursa, uz ovu lekciju mogu se naći kao prilog svi ovi izvorni fajlovi. Za ovaj prvi, najjednostavniji primer, prvo imamo zajedničke delove za sve primere, to su alat cup u obliku ove jar arhive za generisanje parsera; alat jflex za generisanje skenera; objašnjenje osnovnih komandi koje ćemo mi videti u ovoj lekciji i ovaj ulazni fajl koji ćemo koristiti u primerima. U folderu parser ovog prvog primera su ova tri fajla koja smo upravo obašnjavali.

```
package parser;

import java_cup.runtime.*;
import java.io.*;

class Calc {
	public static void main(String args[]) throws Exception {
		FileReader r = new FileReader(args[0]);
		Yylex skener = new Yylex(r);
		parser p = new parser(skener);
		Symbol s = p.debug_parse();
	}
}
```



Da biste primere preveli i pokretali, dovoljno je da startujete terminal. U lekciji su svi primeri dati u linux sintaksi, gde se koristi / da razdvoji foldere i : kada se navodi lista putanja. Na Windowsu se koristi \ i ;. Da biste pokrenuli sve ove primere, dovoljno je da sa sajta jdk.java.net pokupite, recimo ako radite na Windows-u, referentnu implementaciju java jdk za Windows, to je jedan .zip fajl koji raspakujete u neki folder na disku, recimo c:\jdk15. Potom, da biste mogli da radite sa java kompajlerom i java run time-om, potrebno je samo da podesite putanju, evo primer je za Windows: `set path=c:\jdk15\bin;%PATH%` gde je %PATH% stari sadržaj path promenljive. Posle zadavanja ove komande, možete proveriti komandom `javac -version`. Ako vam se ne prijavi nikakva greška, znači da imate sve spremno za rad.

Za ovaj prvi primer, šta sve treba uraditi? Treba izgenerisati skener, to je ova komanda kojom se poziva jflex.

```
java -jar JFlex.jar -d parser parser/calc.lex
```

Crtica d znači da želimo da rezultate rada jflex-a, odnosno ovaj fajl yylex.java smesti u folder parser, a takođe, pošto se ova cela komanda zadaje iz korena primera, lex specifikaciju će naći isto u folderu parser. Prva komanda generiše ovaj izvorni fajl yylex.java.

Druga komanda je da se izgeneriše izvorni fajl parsera na bazi cup specifikacije. Zadaje se ime .jar-a koji sadrži cup alat, koristi se -destdir da svi izlazni fajlovi idu u folder parser.

```
java -jar cup_v10k.jar -destdir parser parser/calc.cup
```

Ulazni fajl je cup specifikacija koja se isto nalazi u folderu parser. Komanda će u tom folderu kreirati parser.java i sym.java. To je definicija one klase konstanti Sym koju posle koristi skener da vrati različite tokene. Deo izlaza kada se pokrene cup je izveštaj o radu alata. Važno je da nema nikakvih grešaka, ovde se nabraja neka statistika o gramatici koja je urađena i ispisuje se da su odgovarajući izlazni fajlovi napravljeni.

```
------- CUP v0.10k Parser Generation Summary -------
  0 errors and 0 warnings
  7 terminals, 6 non-terminals, and 10 productions declared,
  producing 14 unique parse states.
  0 terminals declared but not used.
  0 non-terminals declared but not used.
  0 productions never reduced.
  0 conflicts detected (0 expected).
  Code written to "parser.java", and "sym.java".
----------------------------------------------------- (v0.10k)
```

Zatim treba java kompajlerom prevesti sve java fajlove koje smo dobili. To su onaj polazni Calc.java. Potrebno je staviti classpath da je tekući folder, to je folder iznad foldera parser, pa ovde stoji :, na Windows-u bi stajalo ; pa naveden ovaj .jar fajl od cup alata. On je neophodan da bi se našle one run time klase od koje parser zavisi.

```
javac -cp .:cup_v10k.jar parser/Calc.java
```

Ono što prevodimo je u folderu parser, fajl calc.java. Znači dovoljna je ova jedna komanda i da se navede samo ovaj jedan glavni java fajl. Java kompajler je dovoljno pametan da nađe i sve ostale povezane java fajlove i isprevodi ih u .class fajlove koje će isto staviti u ovaj folder parser. Posle toga smo spremni da izvršavamo program. Program se izvršava kada se pokrene java runtime.

```
java -cp .:cup_v10k.jar  parser.Calc ulaz.txt
```

Sada nema c nego samo java, ovaj classpath je isti kao što je bio, pošto i u run time-u su potrebne neke pomoćne klase parsera. parser.Calc sadrži main i navodi se ulaz.txt. U fajlu ulaz.txt nalazi se sadržaj 3+5, nema enter, nego direktno završen tekstualni fajl. Može se naravno promeniti primer, u ovoj prezentaciji je korišćen uvek ovaj isti ulaz. Pošto smo u parseru pozvali debug_parse() metod, dobićemo trejs rada parsera čime se uveravamo da parser zaista prepoznaje gramatiku. Vidi se svaki korak. Reč je o shift-reduce lalr(1) parseru, tako da vidimo odgovarajuće akcije koje takav parser radi. Ako je sve u redu, izlaz će biti ovakav. Ako bi došlo do neke sintaksne greške u radu parsera, poziv metoda parse() bi bacio izuzetak, pošto mi to ne hvatamo u glavnom programu, program bi završio sa prijavom izuzetka. Ovaj primer, da bi bio dovoljno jednostavan, nije uzimao u obzir obradu greške.

```
# Initializing parser
# Current Symbol is #2
# Shift under term #2 to state #3
# Current token is #3
# Reduce with prod #5 [NT=3, SZ=1]
# Reduce rule: top state 0, lhs sym 3 -> state 1
# Goto state #1
# Reduce with prod #3 [NT=2, SZ=1]
# Reduce rule: top state 0, lhs sym 2 -> state 2
# Goto state #2
# Reduce with prod #0 [NT=1, SZ=1]
# Reduce rule: top state 0, lhs sym 1 -> state 4
# Goto state #4
# Shift under term #3 to state #6
# Current token is #2
# Reduce with prod #6 [NT=4, SZ=1]
# Reduce rule: top state 4, lhs sym 4 -> state 5
# Goto state #5
# Shift under term #2 to state #3
# Current token is #0
# Reduce with prod #5 [NT=3, SZ=1]
# Reduce rule: top state 5, lhs sym 3 -> state 1
# Goto state #1
# Reduce with prod #3 [NT=2, SZ=1]
# Reduce rule: top state 5, lhs sym 2 -> state 9
# Goto state #9
# Reduce with prod #2 [NT=1, SZ=3]
# Reduce rule: top state 0, lhs sym 1 -> state 4
# Goto state #4
# Shift under term #0 to state #8
# Current token is #0
# Reduce with prod #1 [NT=0, SZ=2]
# Reduce rule: top state 0, lhs sym 0 -> state -1
# Goto state #-1
db@zbook:~/temp/calc$
```

Parser sa semantičkom obradom

Prelazimo na sledeću fazu i sledeći primer, a to je proširenje parsera semantičkom obradom. Želimo da naš kalkulator računa vrednost izraza. Ova obrada se može specificirati sledećom atributivno-translacionom gramatikom.

1. $\langle Expr \rangle_r \rightarrow \langle Expr \rangle_e \langle addop \rangle_{op} \langle Term \rangle_t$
   $r \leftarrow (op == \text{'add'}) \ ? \ e+t : e-t$
2. $\langle Expr \rangle_r \rightarrow \langle Term \rangle_t$
   $r \leftarrow t$
3. $\langle Term \rangle_r \rightarrow \langle Term \rangle_t \langle mulop \rangle_{op} \langle Factor \rangle_f$
   $r \leftarrow (op == \text{'mul'}) \ ? \ t*f : t/f$
4. $\langle Term \rangle_r \rightarrow \langle Factor \rangle_f$
   $r \leftarrow f$
5. $\langle Factor \rangle_r \rightarrow NUMBER_{val}$
   $r \leftarrow val$
6. $\langle addop \rangle_r \rightarrow PLUS \ r \leftarrow$ 'add'
7. $\langle addop \rangle_r \rightarrow MINUS \quad r \leftarrow$ 'sub'
8. $\langle mulop \rangle_r \rightarrow STAR \quad r \leftarrow$ 'mul'
9. $\langle mulop \rangle_r \rightarrow DIVIDE \quad r \leftarrow$ 'div'

U onu polaznu gramatiku, smene su potpuno iste, samo dodajemo atribute i atributivna pravila. Neterminali <Expr>, <Term> i <Factor> dobili su jedan sintetizovani atribut, koji predstavlja vrednost dela izraza opisanog datom smenom i neterminalom.

Možemo krenuti od najjednostavnije pete smene, znači kada je izraz samo nekakva celobrojna konstanta, skener će nam dostaviti vrednost konstante kroz atribut terminala NUMBER. Potrebno je samo atributivnim pravilom to iskopirati u atribut <Factor>-a. Ako se <Term> direktno menja sa <Factor>, reč je samo o kopiranju vrednosti atributa. Takođe i u smeni 2, kada se <Expr> direktno menja sa <Term>, radi se samo o kopiranju vrednosti atributa. Jedino za nijansu komplikovanije procesiranje je kod smena 1 i 3. Kada imamo sabiranje ili oduzimanje, imamo dva podizraza, prvi je predstavljen sa <Expr>, drugi je predstavljen sa <Term>, vrednosti podizraza, pošto je reč o sintetizovanim atributima, biće već smeštene u atribute e i t. Koja operacija je u pitanju, tu informaciju prosleđujemo iz smena 6 i 7. Ako je smena 6, onda će sintetizovani atribut neterminala <addop> dobiti string "add", inače će dobiti string "sub". I onda kroz ovaj atribut op možemo ispitati njegovu vrednost, da li je upotrebljeno sabiranje ili oduzimanje. U zavisnosti od toga, sabraćemo ili oduzeti vrednosti e i t i upisati u atribut r. Analogno je urađeno za smenu 3, samo tu su u pitanju operacije množenja i deljenja.

Ovom atributivno-translacionom gramatikom smo specificirali kompletnu semantičku obradu koju naš kalkulator treba da radi. Sada da vidimo kako se to programski realizuje.

Trebaju nam isti fajlovi kao u prethodnom primeru. To je specifikacija leksičkog analizatora. Ovde je prikazan izlaz programa koji poredi tekstualne fajlove da se tačno vidi kakve smo izmene u specifikaciji skenera načinili. Promenjeno je ime paketa, sada je to paket atributi. Jedina suštinska izmena je da u ovom pravilu koje prepoznaje konstantu i vraća token NUMBER, uz instancu tog simbola (objekat Symbol) taj token sada ima dodatni vrednosni deo tokena, prosleđujemo instancu java klase Integer. yytext nam daje tekstualni oblik konstante, koju konstruktor Integer-a konvertuje u binarnu vrednost. Kroz ovaj java tip Integer prosleđujemo iz skenera u parser vrednost konstante.

```
package atributi;
import java_cup.runtime.Symbol;
%%
%cup
%%
"+" { return new Symbol(sym.PLUS); }
"-" { return new Symbol(sym.MINUS); }
"*" { return new Symbol(sym.STAR); }
"/" { return new Symbol(sym.DIVIDE); }
[0-9]+ { return new Symbol(sym.NUMBER, new Integer(yytext())); }
[ \t\r\n\f] { /* ignore white space. */ }
. { System.err.println("Illegal character: "+yytext()); }
```

Što se tiče specifikacije sintaksnog analizatora, tu su promene najveće. Prvo, moramo uvesti atribut za terminal NUMBER. To se radi u specifikaciji (ne)terminala pre imena terminala, navodi se odgovarajući java tip, ako terminal ili neterminal imaju atribut. To mogu biti ugrađeni java tipovi, kao što NUMBER ima tip int (u programu mora da se prosledi Integer), a atribut neterminala <Expr>, <Term> i <Facto> je java klasa Integer. Na ovom mestu specifikacije tipa atributa, u

trećem primeru kada se koriste apstraktna sintasna stabla, mora biti java klasa izvedena iz standardne java klase Object, ne može biti primitivan tip podataka. Neterminali <addop> i <mulop> imaju java String da se reprezentuje koja operacija je prepoznata.

```
package atributi;
terminal int NUMBER;
terminal PLUS, MINUS, STAR, DIVIDE;
nonterminal Integer expr, term, factor;
nonterminal String addop, mulop;
expr ::= term:t {: RESULT = t; :}
| expr:e addop:op term:t {: RESULT = new Integer ((op.equals("add"))? e.intValue()+t.intValue() : e.intValue()-t.intValue()); :};
term ::= factor:f {: RESULT = f; :}
| term:t mulop:op factor:f {: RESULT = new Integer((op.equals("mul"))? t.intValue()*f.intValue() : t.intValue()/f.intValue()); :};
factor ::= NUMBER:val {: RESULT = new Integer(val); :};
addop ::= PLUS {: RESULT="add"; :} | MINUS {: RESULT="sub"; :};
mulop ::= STAR {: RESULT="mul"; :} | DIVIDE {: RESULT="div"; :};
```

Smene sada imaju jedan novi deo, koji se ranije nije pojavljivao i koji je ovde naznačen kao dodat. Taj deo počinje sa {:: i završava sa ::}. To su takozvane semantičke akcije. Smene koje treba da obave neku semantičku obradu imaju imaju te semantičke akcije. Da se podsetimo, bottom-up parseri prirodno prihvataju te akcije na samim krajevima smena i u ovom jednostavnom primeru to nas je zadovoljavalo da semantičke akcije budu na samim krajevima smena. Alat cup dozvoljava da akcije ne budu na samim krajevima, ali kao što smo već rekli, onda dolazi do one automatske konverzije gramatike, da se dodaje novi neterminal i da se takva akcija interno izvlači u posebnu smenu.

Semantičke akcije koriste i atribute pojedinih simbola sa desne strane smene i onda moramo imenovati te atribute. Recimo u ovoj smeni <Expr> → <Term>, taj <Term> ima atribut koji se zove t, u smeni se piše Term::t da se uvede ime tog atributa. Sećamo se još iz polazne atributivno-translacione gramatike u drugoj smeni treba samo iskopirati vrednost t u r, to je ovde realizovano semantičkom akcijom RESULT=t. Ime atributa na levoj strani se ne zadaje, nego ime atributa leve strane je uvek RESULT.

Jedinu malo složeniju semantičku obradu imaju smene za sabiranje i množenje. To je ova smena koja ima na levoj strani <Expr> na desnoj strani ima <Expr> sa atributom e, <addop> sa atributom op i <Term> sa atributom pod nazivom t. Vrednost atributa na levoj strani RESULT se računa odgovarajućim java izrazom (koji koristi i konverziju između primitivnog java tipa i Integera). Kompletnu semantičku obradu smo ubacili u gramatiku. Ovaj fajl se malo zakomplikovao. Bio je vrlo pregledan dok je sadržao samo gramatičke smene, a sada je to kombinacija smena i nekog java koda za procesiranje semantike. U ovom primeru to je još uvek dovoljno jednostavno, ali zamislite neki veliki kompajler za neki kompleksan programski jezik, kome treba na desetine i stotine hiljada linija koda, zamislite ceo taj kod smešten na jedno mesto i još pomešan sa nekom gramatikom koja ima na hiljade smena, bilo bi vrlo nepregledno.

Kako izgleda Calc klasa u ovom slučaju? Osim što je promenjen paket, čitav početak je isti kao ranije, instanciranje FileReader-a, instanciranje skenera, instanciranje parsera, dostavlja mu se skener i poziv metoda parse(). U prvom primeru je bio debug parse čime smo dobijali trejs rada parsera, uobičajeno se poziva parse() koji radi celo procesiranje samo ne ispisuje trejs rada na standarnom izlazu. Ono što je u ovom primeru dodato, to su poslednje dve linije. Funkcija parse() nam vraća, ako uspešno završi parsiranje, rezultujući objekat kome pristupamo putem ovog interfejsa Symbol. Interfejs Symbol je iz cup runtime-a, predstavlja simbol steka sa njegovom

semantičkom vrednošću. Taj simbol s odgovara startnom neterminalu gramatike po završetku parsiranja i možemo dobiti preko s.value sadržaj atributa koji smo definisali. Naš startni simbol je <Expr> i definisali smo da on ima atribut tipa Integer, sa ovim cast-om možemo iz ovog simbola s dobiti finalnu vrednost njegovog atributa i onda tu vrednost ispisati.

```
package atributi;

import java_cup.runtime.*;
import java.io.*;

class Calc {
    public static void main(String args[]) throws Exception {
        FileReader r = new FileReader(args[0]);
        Yylex skener = new Yylex(r);
        parser p = new parser(skener);
        Symbol s = p.parse();
        Integer result = (Integer)(s.value);
        System.out.println(result.intValue());
    }
}
```

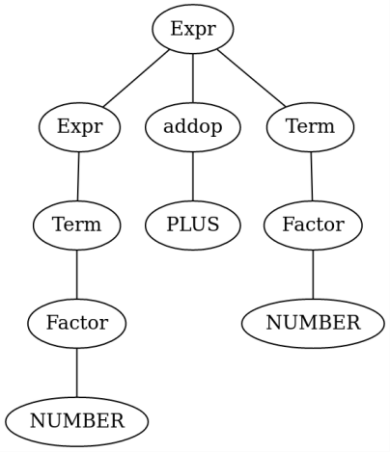
Kompletan niz komandi za generisanje skenera, generisanje parsera, prevođenje svih fajlova i izvršavanje programa, ako je ulaz bio 3+5, na izlazu će se ispisati rezultat 8:

```
java -jar JFlex.jar -d atributi atributi/calc.lex
java -jar cup_v10k.jar -destdir atributi atributi/calc.cup
javac -cp .:cup_v10k.jar atributi/Calc.java
java -cp .:cup_v10k.jar  atributi.Calc ulaz.txt
```

Konkretno i apstraktno sintaksno stablo

Nastavljamo dalje, da uporedimo pojmove konkretnih i apstraktnih sintaksnih stabala. Konkretno sintaksno stablo smo pominjali u kontekstu parsiranja i rekli da ono striktno odgovara gramatici i izvođenju neke konkretne ulazne sekvence.

Za našu gramatiku kalkulatora i ulaznu sekvencu 3 + 5 njoj pri radu parsera odgovara sledeće konkretno sintaksno stablo.



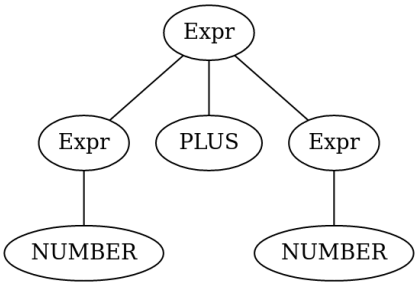
U sintaksnom stablu čvorovi odgovaraju neterminalima, a primena neke smene, recimo ove prve smene, gde je sabiranje, odgovara ovom čvoru stabla i njegovim neposrednim sinovima. I tako dalje, kasnije se <Expr> menjao sa <Term>, <Term> se menja sa <Factor>, a da bismo došli do prvog broja, <Factor> se menja sa NUMBER. Na sličan način i u ovom delu stabla, ali nismo krenuli od <Expr> nego samo <Term>, <Factor>, NUMBER, <addop> se menjao u konkretnom primeru sa terminalom PLUS, tako da, kad povežemo listove stabla, vidimo NUMBER PLUS NUMBER, to je ono što dolazi iz skenera.

Koncept apstraktnog sintaksnog stabla

Izraze bismo mogli opisati znatno jednostavnijom gramatikom, koja opisuje njihovu suštinu, a suština izraza su operandi i operatori. Jedna od takvih gramatika je gramatika koji ima samo jedan jedini tip neterminala <Expr> i ima sve ove smene kojima uvodimo pojedinačne operacije i konačno, <Expr> može biti samo jedna celobrojna konstanta. Ovakvom gramatikom možemo opisati izraz 3 + 5 i dobićemo još jednostavnije stablo sa manje čvorova, nego u prethodnom primeru (slika je data u nastavku).

<Expr> → <Expr> PLUS <Expr>
<Expr> → <Expr> MINUS <Expr>
<Expr> → <Expr> STAR <Expr>
<Expr> → <Expr> DIVIDE <Expr>
<Expr> → NUMBER

Ovako uprošćeno sintaksno stablo, koje samo suštinske elemente reprezentuje koji su važni za kasniju semantičku obradu, nazivamo apstraktnim sintaksnim stablom. Može se primetiti da bi stablo moglo biti još jednostavnije, kada bi se objedinili svaki od čvorova <Expr> sa čvorom koji je neposredno ispod njega što čak možemo zamisliti u nekoj programskoj realizaciji, to ćemo kasnije videti, da je Expr neka bazna klasa, a onda mogu postojati izvedene klase tako da u stvari instanca čvora stabla može biti izvedena klasa iz Expr, znači da imamo samo tri čvora. Dva čvora su vrednosti, a jedan čvor je operacija, što bi onda baš bila najkonciznije izražena semantika izraza. Poenta svega ovoga je da apstraktna sintaksna stabla uprošćavaju konkretna i ostavljaju samo elemente važne za kasniju semantičko obradu, ona su manjeg obima i jednostavnija za shvatanje i, samim tim, kod koji vrši semantičku obradu može biti koncizniji i efikasniji ako se koriste ova apstraktna stabla, nego ako se koriste konkretna stabla sa svim detaljima.



Problem sa uprošćenom gramatikom je dvosmislenost u ovom primeru i činjenica da ne specificira asocijativnost niti prioritete operatora. Ovakvu gramatiku ne bi se mogla provući kroz alat cup, jer bi vam prijavio različite shift/reduce konflikte. To smo kroz ranije primere šire objašnjavali,

recimo da već ova prva smena koja ima na dva mesta <Expr> odmah ubacuje dvosmislenosti. Znači ne možemo direktno koristiti ovakvu gramatiku.

Međutim, ideja je sledeća: ako za parsiranje izraza koristimo početnu gramatiku i parser, a zatim na osnovu konkretnog stabla izvođenja za sekvencu naknadno konstruišemo uprošćeno, to jest, apstraktno sintaksno stablo, rešili smo problem dobijanja apstraktnog stabla. Apstraktno stablo možemo dobiti ako odgovarajuće akcije za konstrukciju i povezivanje čvorova stabla ubacimo kao semantičke akcije u početni parser za konkretnu sintaksu. Međutim, ostaje problem, u opštem slučaju, mapiranja između apstraktne i konkretne sintakse. Ako sada ne razmatramo samo naš jednostavan primer kalkulatora, nego zamislite da imamo zadatak da napravimo kompajler za neki ozbiljan programski jezik, mi moramo odvojeno definisati konkretnu i apstraktnu sintaksu i manuelno ubaciti u parser sve te semantičke akcije da se konstruiše apstraktno sintaksno stablo iz konkretnog stabla. Znači, kada je u pitanju neka složena sintaksa, ako sve to treba ručno raditi, onda to deluje kao veliki posao koji anulira prednosti korišćenja apstraktnog sintaksnog stabla samog po sebi.

Međutim, alternativni pristup bi bio da se iskoristi početna konkretna sintaksa jezika i automatski izgeneriše apstraktno sintaksno stablo koje u tom slučaju nema mnogo apstrakcije, recimo mogu da se odseku delovi konkretnog stabla izvođenja od izabranog čvora naniže ako nemaju semantički značaj, ali prednost ovog pristupa je da može da se skoro potpuno automatizuje, znači uz minimalne intervencije na polaznoj gramatici, automatski mogu da se izgenerišu sve potrebne strukture podataka za dalje kompajlerske obrade. Ovaj pristup primenjen je proširivanjem standardnog cup alata i to je osmišljeno upravo na ovom kursu i realizovano kroz jedan diplomski rad kolege Dušana Stankovića [6], a mi upravo taj proširene alat astcup koristimo i za projekat na kursu. Ovaj alat možete naći na sajtu kursa koji je pomenut u uvodu. U delu sajta domaći zadatak nalazi se dokumentacija za ovaj prošireni alat astcup, a ovo poglavlje predstavlja pokušaj da se objasni suština alata i način njegovog korišćenja.

Primer kalkulatora sa apstraktnim sintaksnim stablom

U nastavku predavanja ćemo primeniti ovaj pristup na primeru kalkulatora da bismo videli suštinu pristupa. Ovo je sada treća varijanta primera, ovaj put sa apstraktnim sintaksnim stablom. Da vidimo kakve smo izmene pravili u onim fajlovima od kojih smo počeli. Imamo specifikaciju skenera i on je potpuno isti kao kod varijante sa atributivnom translacijom, samo je dodato tokenu NUMBER da vraća brojnu vrednost konstante.

```
package ast;
terminal int NUMBER;
terminal PLUS, MINUS, STAR, DIVIDE;
nonterminal Value expr, term, factor;
nonterminal String addop, mulop;
expr ::= (SingleExpr) term
| (Addition) expr addop term;
term ::= (SingleTerm) factor
| (Multiplication) term mulop factor;
factor ::= (Constant) NUMBER:val;
addop ::= PLUS {: RESULT="add"; :} | MINUS {: RESULT="sub"; :};
mulop ::= STAR {: RESULT="mul"; :} | DIVIDE {: RESULT="div"; :};
```

Što se tiče specifikacije parsera, ona naravno mora da pretrpi izmene i zgodnije je gledati u odnosu na polaznu varijantu čistog parsera koji nije imao nikakvu semantičko obradu. U ovoj varijanti specifikacija parsera generalno trpi manje izmene, pogotovo kada je u pitanju neka kompleksna semantika kao u pravom kompajleru, Za ovaj mali primer, deluje da ovde postoje neke znatne izmene, ali što je veća se semantička obrada, videli biste da u stvari su to prilično minorne izmene u samoj specifikaciji parsera i gramatike.

Menjamo opet paket, sad je ast. Kao i ranije, pošto je ovo u odnosu na čist parser, dodali smo da token NUMBER ima atribut ceo broj. <Expr>, <Term> i <Factor> neterminali imaju svoj semantički atribut. Ranije smo ovde koristili direktno Integer, znači onu standardnu Java klasu. Standardne klase mogu da se upotrebe, recimo <addop> i <mulop> vraćaju tip java String. Međutim, ovaj proširen cup ima to ograničenje, da ako bismo upotrebili neki primitivan tip ili tipove kao što su Integer i String, tada alat seče apstraktno stablo, Znači, to možete da uradite na "krajevima" gramatičkog izvođenja. Ovde je ideja da stablo malo uprostimo, tako što nećemo eksplicitno u njemu reprezentovati ove smene kad se <addop> menja sa PLUS, nego ćemo prosto izvršiti jedine semantička akcije koje su ostale, da se ovom <addop>-u dodeli rezultujući atribut. U suštini, videćemo da naše apstraktno stablo i nema <addop> kao poseban čvor, nego će se on samo dodavati kao atribut ovih čvorova u kojima se on kasnije javlja, to su expression i ova smena gde se upotrebljava <addop>.

Da se vratimo na suštinu, znači za ove gramatičke neterminale koji imaju svoje reprezentaciju u apstraktnom sintaksnom stablu, atribut mora biti neka korisnička klasa pa smo mi dodali klasu Value, koja treba da pamti samo jedan ceo broj. Tu postoje dve varijante realizacije. Jedna varijanta, koja nije primenjena, je da ova klasa proširuje Java klasu Integer i onda bismo morali obezbediti i konstruktor, pošto Integer nema default konstruktor. Međutim, ja sam ovde krenuo putem da ovaj Value bude kao neki zapis, znači nema nikakve metode, nego ima samo javno dostupno celobrojno polje v. Znači, tako ćemo pamtiti semantički sadržaj ovih naših neterminala u apstraktnom sintaksnom stablu. Za ove druge neterminale, rekli smo, oni neće ni imati svoj reprezentaciju, znači malo ćemo skratiti u odnosu na konkretno sintaksno stablo.

```
package ast;

class Value {
    public int v;
}
```

Sada ide specifikacija gramatike sa smenama.

```
package ast;
terminal int NUMBER;
terminal PLUS, MINUS, STAR, DIVIDE;
nonterminal Value expr, term, factor;
nonterminal String addop, mulop;
expr ::= (SingleExpr) term
| (Addition) expr addop term;
term ::= (SingleTerm) factor
| (Multiplication) term mulop factor;
factor ::= (Constant) NUMBER:val;
addop ::= PLUS {: RESULT="add"; :} | MINUS {: RESULT="sub"; :};
mulop ::= STAR {: RESULT="mul"; :} | DIVIDE {: RESULT="div"; :};
```

Taj deo specifikacije trpi jednu vrlo malu izmenu. I dalje je moguće koristiti po želji semantičke akcije, ali to ovde predstavlja samo specijalan slučaj kad želite da malo skratite stablo i da u toku parsiranja prosledite odgovarajuće podatke do delova stabla koji ostaju. U apstraktnom sintaksnom stablu, recimo, <addop> više neće imati svoju posebnu prezentaciju, znači moramo nekakve vrednosti iz njegovih smena, ako postoje, još u toku parsiranja proslediti do nekog čvora stabla koji ima semantičku reprezentaciju. To je u prvoj smeni ova pojava <addop>. Međutim, to je specijalni slučaj, ali u normalnom slučaju će smena imati svoju reprezentaciju u apstraktnom sintaksnom stablu i onda ta reprezentacija mora da se imenuje. Znači ovde dodajemo u svaku od tih normalnih smena koje će ostati u stablu njeno ime. Imenujemo ih na taj način što na samom početku desne strane smene u zagradi stavimo ime smene koje mi biramo proizvoljno da nas asocira na neko značenje smene. Recimo, SingleExpr je ako se ceo <Expr> svodi samo na <Term>, a Addition je ako se pojavljuju oni operatori sabiranje ili oduzimanja. I tako dalje, svaka od ovih smena koja će imati svoju reprezentaciju u stablu dobija svoje ime.

Kasnije ćemo detaljno razmotriti kakva je struktura tog sintaksnog stabla, ali prvo da vidimo, u ovim našim fajlovima koje smo pisali u primeru koji nisu automatski generisani, šta još imamo kao sadržaj fajla.

Calc.java, ovo je opet u odnosu na varijantu čist parser, osim što je izmenjen paket, nema izmena do poziva metoda parse(). Sada ide onaj deo koji se u stvari odnosi na semantičku obradu i sve ovo zeleno su ubačene nove linije. Ono što smo već videli u varijanti sa atributima, to je da, ako parser uspešno završi rad, dobijamo kao povratnu vrednost iz metoda parse() stek simbol koji odgovara startnom simbolu gramatike. Njemu pristupamo preko ovoga interfejsa Symbol iz java_cup runtime-a, i rekli smo da s.value onda može da nam da semantički sadržaj.

```
package ast;

import java_cup.runtime.*;
import java.io.*;

class Calc {
        public static void main(String args[]) throws Exception
                FileReader r = new FileReader(args[0]);
                Yylex skener = new Yylex(r);
                parser p = new parser(skener);
                Symbol s = p.parse();
                Expr e = (Expr)(s.value);
                ArithmeticVisitor v = new ArithmeticVisitor();
                e.traverseBottomUp(v);
                System.out.println(e.value.v);
                System.out.println(e.toString());
        }
}
```

U varijanti alata ast-cup, kada se uključe apstraktna sintaksa stabla, uvodi se posebna klasa u apstraktnom sintaksnom stablu, za svaki neterminal koja ima isto ime kao neterminal, sa početnim velikim slovom. Tako da, pošto se neterminal zvao <Expr> nama je alat izgenerisao tu klasu Expr da predstavi startni neterminal u apstraktnom sintaksnom stablu i onda s.value možemo da kastujemo na objekat te klase. To je u stvari koren apstraktnog sintaksnog stabla. Ako želimo sada

da obrađujemo apstraktno sintaksno stablo, tipično je da za svaku obradu obezbeđujemo poseban posetilac (visitor) u koji je ugrađena obrada čvorova stabla i onda nad korenim čvorom stabla pozovemo odgovarajuću traverse funkciju. Alat automatski generiše dva moguća iteratora, da se obiđe stablo od dna ka vrhu, znači od listova ka korenu, ili alternativno top-down. Ovi obilasci odgovaraju onim postorder i preorder obilascima poznatim iz struktura podataka.

Mi želimo ovde da izračunamo vrednost izraza, što odgovara obilasku stabla od dna vrhu. Tako da tom iteratoru, ovoj traverse funkciji, prosleđujemo odgovarajući visitor koji specificira šta ćemo raditi u svakom čvoru kada dođemo pri obilasku u odgovarajući čvor. Naša semantička obrada će se svoditi na računanje vrednosti izraza. U ovom Čvoru Expr koji reprezentuje startni neterminal, taj semantički dodatak je preko ove klase Value tako da će taj čvor u automatski generisanom kodu imati polje value koje je public vidljivosti. Onda ovde u instanci klase Value moramo pristupiti ovom v. Ovo jeste za nijansu neelegantno (moglo je naravno da se to sve oklopi metodama, ali je zbog jednostavnosti ovako odabrano). Znači preko te instance klase Value i njenog polja v u ovom objektu koji u apstraktnom sintaksnom stablu reprezentuje koreni čvor, mi ispisujemo finalnu vrednost izraza. Takođe, ovde ima još jedan ispis, a to je da se ispiše celo stablo u svojoj tekstualnoj reprezentaciji. U sklopu automatski generisanog koda za svaki čvor stabla automatski se generiše i ovaj toString() method i kad se on pozove nad korenim čvorom, ispiše se izgled kompletnog stabla. Taj ispis ćemo videti u nastavku.

Već sam pomenuo da moramo te visitore da specificiramo i zato, u odnosu na sve ove prethodne primere, ovaj primer ima još jedan dodatan četvri fajl, koji mi ručno pišemo. To je atrithmeticVisitor.java ali njega ćemo proučiti malo kasnije.

Kako u ovoj varijanti primera prevodimo i pokrećemo program? Prvo, što se tiče generisanja skenera i parsera sa podrškom za apstraktno sintaksno stablo, generisanje skenira ide na potpuno isti način kao ranije, osim što je sad folder ast. Generisanje parsera ima malo više parametara komande linije. Kao i ranije, poziva se `java -jar <ime ast kup alata>.` destdir imamo kao i ranije, znači da svi generisani fajlovi idu u ast folder. Ovo su dva nova parametara komande linije: -ast ast definiše ime paketa za sve generisane klase apstraktnog sintaksnog stabla (to su automatski generisane Java klase); -buildtree je važna opcija koja koja inicira generisanje svih potrebnih klasa za reprezentaciju apstraktnog sintaksnog stabla. Na kraju komande se nalazi ime fajla sa ulaznom ast-cup specifikacijom.

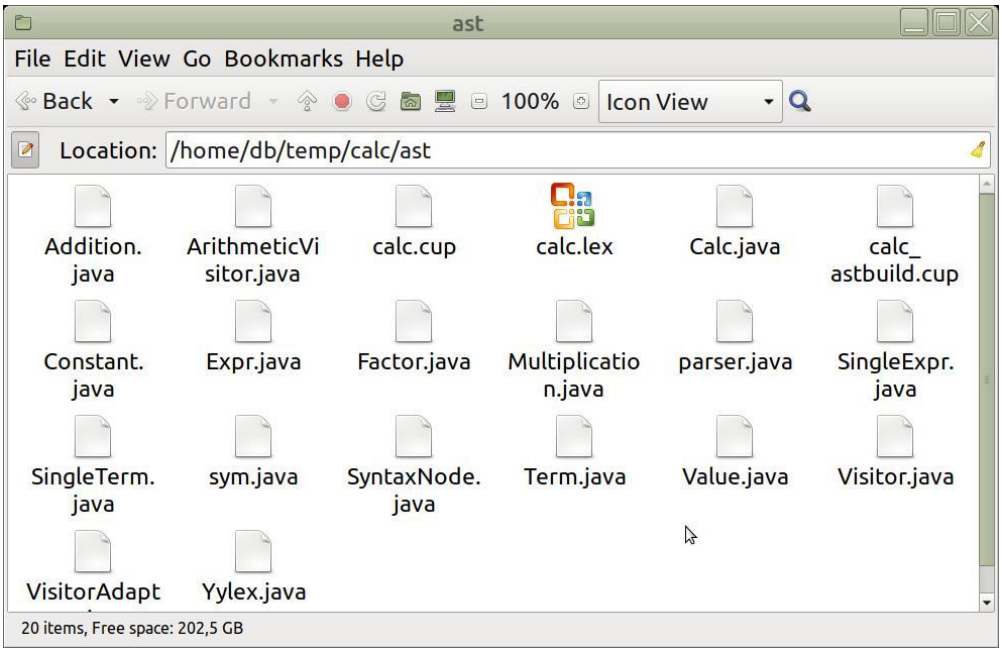
```
java -jar JFlex.jar -d ast ast/calc.lex
java -jar cup_v10k.jar -destdir ast -ast ast -buildtree ast/calc.cup
```

U nastavku je slika sadržaja foldera. Kao rezultat ovih komandi se pojavljuje mnogo više fajlova nego što ih je na početku bilo, znači oni fajlovi koji su ručno pisani i od kojih smo krenuli, to je ovaj .lex (specifikacije skenera), zatim .cup (specifikacija parsera), Calc.java sadrži glavni program i arithmeticVisitor.java Znači samo ova četiri fajla su ručno pisana, sve drugo je automatski izgenerisano:

`yylex.java` je skener koji je jFlex alatom izgenerisan i njega smo i ranije pominjali.

`Sym.java` je fajl koji generiše ast-cup alat za sinhronizaciju između skenera i parsera.

`parser.java` je parser koji generiše ast-cup.



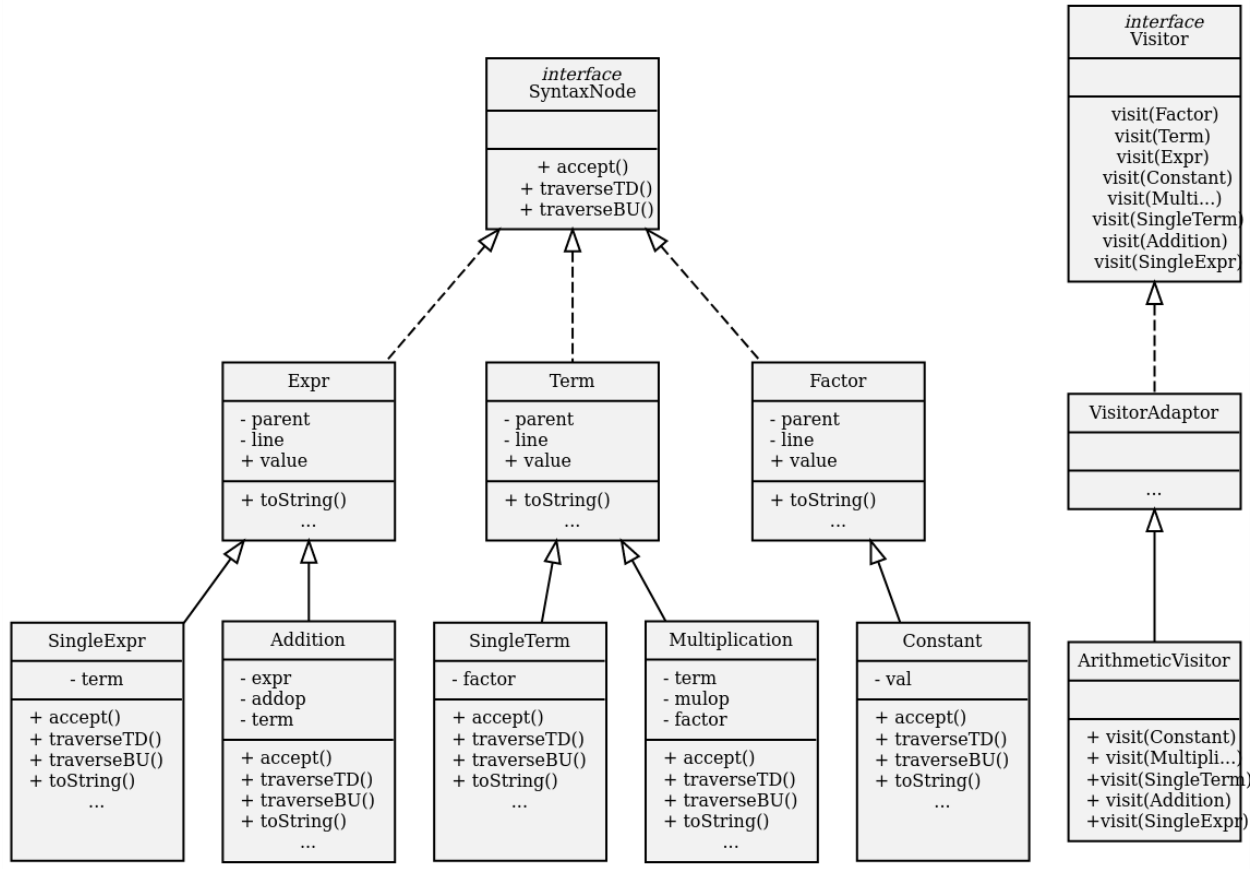
Pored ovih fajlova koji su bili generisani i u ranijim primerima, sada u ovom primer sa ast-cup imamo još veliki broj drugih fajlova:

`CalcAstBuild.cup` može da izazove interes, zato što ima istu ekstenziju .cup. To je radni fajl, znači njega treba potpuno zanemariti. Možete da naravno pogledati, ali ne treba ga ni na koji način editovati, jer se on automatski generiše na osnovu glavnog .cup fajla. U pitanju je prilagođeni .cup fajl u koji su bačene semantičke akcije za izgradnju apstraktnog sintaksnog stabla tokom parsiranja. Znači, tu nas je alat ast-cup oslobodio posla da mi smišljamo kako ćemo izgraditi sintaksno stablo.

Svi ostali fajlovi sa slike, koje ja još nisam pomenuo, su fajlovi koji reprezentuju apstraktno sintaksno stablo i osnovne obrade, prema projektnom šablonu visitor.

Znači to su fajlovi `SyntaxNode.java`, `Visitor,java`, `VisitorAdapter.java`, to su razne klase i interfejsi: `Expr.java`, `Term.java`, `Factor.java`, to prepoznajemo kao imena gramatičkih neterminala, zatim `Addition.java`, `Multiplication.java`, `SingleExpr.java`, `SingleTerm.java`, `Constant.java` -- to su imena smena, znači svaka smena je zaslužila sopstvenu klasu.

Sve to predstavlja podršku za apstraktno sintaksno stablo sa obilaskom čvorova po visitor šablonu. Svi ovi fajlovi su automatski generisani i ne sme se u njima ništa ručno menjati, jer će te izmene biti uklonjene pri ponovnom pokretanju ast-cup alata. Ako bilo šta menjate u onom polaznom .cup fajlu, uvek mora da se ponovo pokrene ast-cup alat i ovi automatski generisani fajlovi će biti prvo obrisani, pa posle izgenerisane nove varijante. Bolje nego gledanje samog koda generisanih fajlova je gledanje klasnog dijagrama, koji prikazuje statičku strukturu ovoga apstraktnog sintaksnog stabla. Klase Expr, Term i Factor odgovaraju neterminalima u gramatici. One implementiraju interfejs SyntaxNode i time obezbeđuju obilazak stabla. Ali ove tri klase su apstraktne klase i one se ne instanciraju direktno tokom rada parsera.

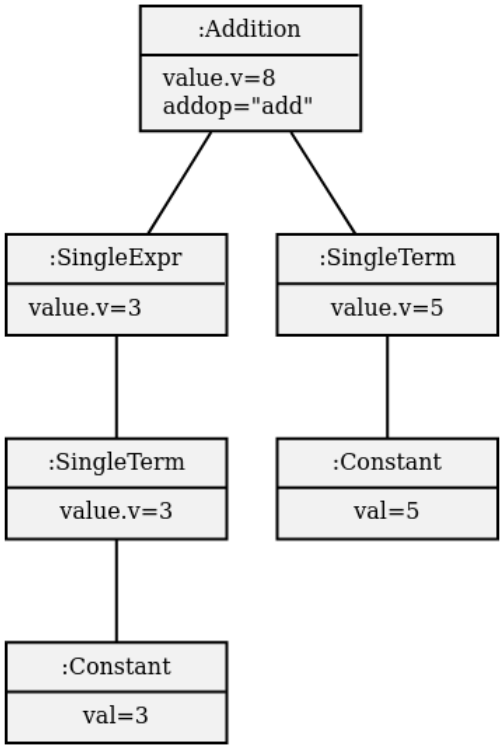


Ove klase na donjem nivou dijagrama, to su Addition, Multiplication, SingleExpr, SingleTerm i Constant odgovaraju imenovanim gramatičkim smenama. Svaka od njih proširuje klasu neterminala kojem smena pripada, koji je znači na levoj strani smene. Takođe primetite da neterminali <addop> i <mulop> nemaju svoje svoje klase, niti smena za njih, jer nijedna nije imenovana. Znači ovi delovi parserskog stabla neće se pojaviti u rezultujućem apstraktnom sintaksnom stablu, ali informacije iz ovih smena su, korišćenjem semantičkih akcija u parseru prosleđene smenama u kojima se <addop> i <mulop> pojavljuju i zapamćene u poljima odgovarajućih klasa.

Recimo, klasa Addtion ima polje addop koje je tipa stringa. Pomenuta klasa Addition obezbeđuje informacije o desnoj strani smene kroz svoja polja, jer na desnoj strani smene bili su neterminali <Expr>, <addop> i <Term>. Tu sad imamo tri polja, za <addop> je tipa stringa, a za <Expr> i <Term> su reference na odgovarajuće objekte izvedenih klasa iz Expr i Term. Osim toga, klase za smene obezbeđuju konkretnu implementaciju accept metoda kao dela šablona visitor i implementaciju iteratora kroz stablo traverse. Znači to je sve automatski generisan kod. U opštem slučaju šablona visitor obično se iteratori ne stavljaju u istu klasu koja je za čvor stabla, ali ovde je to urađeno zbog jednostavnijeg korišćenja i manje generisanih klasa. Takođe vidimo u implementacij automatski generisan metod toString koji može da nam obezbedi ispisivanje stabla.

Apstraktno sintaksno stablo koje nastaje kao rezultat rada parserskog procesiranja konkretnog ulaza biće struktura sastavljena od instanci primenjenih smena, znači od instanci klasa u donjem redu dijagrama. Posebno se generiše interfejs Visitor koji ima po jedan visit() metod za svaku vrstu ćvora apstraktnog sintaksnog stabla, kako za čvorove koji reprezentuju neterminale, tako i za čvorove koji reprezentuju smene. Imamo i automatski generisanu klasu VisitorAdapter koja implementira interfejs Visitor i obezbeđuje prazne visitor metode koji nemaju nikakvu semantičku obradu, ali obezbeđuju nastavak prolaska kroz stablo tako da ne moramo da pišemo bukvalno sve visitore za našu semantičku obradu, nego samo one gde ćemo vršiti neku realnu semantičku obradu. A iteriraćemo svakako kroz celo stablo.

Na ovom dijagramu, jedina klasa koja nije automatski generisana, koju mi moramo da napišemo, je klasa ArithmeticVisitor. Takve klase uvek se nasleđuju iz klase VisitorAdapter i potom samo redefinišemo one metode za one čvorove koji su nama zanimljivi, gde imamo neku semantičku obradu. Pre nego što pogledamo klasu ArithmeticVisitor, da vidimo objektni dijagram apstraktnog sintaksnog stabla za konkretan ulaz 3 + 5.



Po završetku pasiranja, parser vraća referencu na koren ove ast strukture. Ona je sastavljena od instanci klasa koje predstavljaju imenovane smene, sa informacijama o levoj strani smene nasleđenim iz bazne klase. Prikazan je sadržaj stabla pošto se aktivira prolazak kroz stablo od dna ka vrhu. Videli smo da se u main funkciji pozove TraverseBottomUp iterator kome se prosledi instanca ArithmeticVisitor-a. Ovde vidimo da su vrednosti objekta value već izračunate, znači prikazano je stanje po završetku rada tog iteratora.

ArithmeticVisitor je poslednja klasa koje je ručno napisana, a koju još nismo proučili. Ovde se nalazi kompletna semantička obrada, a u ovom primeru to je računanje vrednosti ulaznog izraza

reprezentovanog ovim ast stablom. Taj visitor je niz ovih visit() funkcija, koje su redefinisane pošto nasleđujemo VisitorAdaptor, a on ima rekli smo prazne implementacije raznih vizitora. Mi redefinišemo samo one visitore gde nam treba neka semantička obrada. Znači, kada se poseti odgovarajući čvor, onda se radi semantička obrada.

```
package ast;

public class ArithmeticVisitor extends VisitorAdaptor {

    public void visit(Constant constant) {
        constant.value = new Value();
        constant.value.v = constant.getVal();
    }

    public void visit(Multiplication multiplication) {
        multiplication.value = new Value();
        int op1=multiplication.getTerm().value.v;
        int op2=multiplication.getFactor().value.v;
        if(multiplication.getMulop().equals("mul")) {
            multiplication.value.v = op1*op2;
        } else {
            multiplication.value.v = op1/op2;
        }
    }

    public void visit(SingleTerm SingleTerm) {
        SingleTerm.value = SingleTerm.getFactor().value;
    }


    public void visit(Addition addition) {
        addition.value = new Value();
        int op1=addition.getExpr().value.v;
        int op2=addition.getTerm().value.v;
        if(addition.getAddop().equals("add")) {
            addition.value.v = op1+op2;
        } else {
            addition.value.v = op1-op2;
        }
    }

    public void visit(SingleExpr SingleExpr) {
        SingleExpr.value = SingleExpr.getTerm().value;
    }
}
```

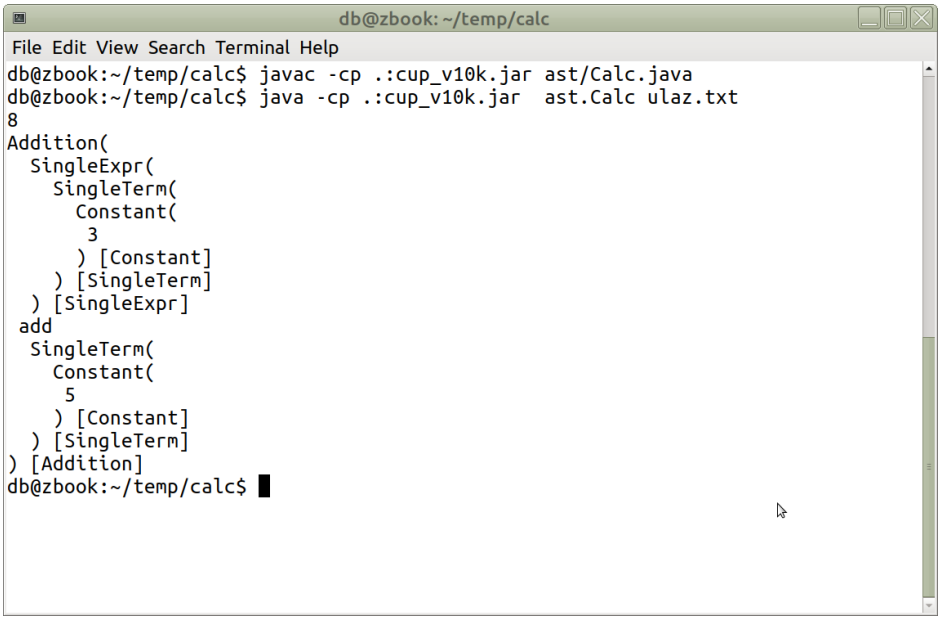
Nećemo objašnjavati detaljno sve visitore, ali recimo smena Addition je dovoljno složena da možda zavređuje posebno objašnjenje. Metod visit() dobija objekat tekućeg čvora kao svoj stvarni argument. Čvor Addition ima polja za svoje sinove u stablu i oni su već obiđeni kod bottom-up prolaska. Do tih polja se dolazi preko ovih getX() izraza. Na desnoj strani smene su <Expr> i <Term>. Oni jesu u generisanom kodu u skladu sa dobrom praksom OO programiranja privatna polja, pa se njima pristupa putem get-era i set-era. Znači ovaj metod izračunava informacioni sadržaj tekućeg čvora, to jest vrednost value objekta na osnovu vrednosti value objekata u sinovima koji su već obiđeni u tom trenutku, pošto je u pitanju bottom-up obilazak. Vrstu operacije

dobijamo iz addop polja sa getAddop(), njegova vrednost je String i onda ispitujemo da li je u pitanju sabiranje ili oduzimanje. Na osnovu vrste operacije računamo rezultujuću vrednost.

Treba primetiti da, prva stvar koja se radi u ovom visitor-u je da za ovo polje value koje je sa public pravima pristupa, znači direktno pristupamo polju, moramo instancirati novi Value objekat. To je zato što u generisanom Expr odakle je to polje nasleđeno u Addition, tamo se taj value objekat uvodi tako što postoji samo jedna automatski generisana linija public Value value = null; Kada instanciratirate klasu Addition ovo polje je null, tako da mi, da bismo zaista pamtili objekat Value, uvek moramo pri dolasku u čvor prvo instancirati objekat Value.

Zanimljivo je recimo u ovom drugom vizitoru uočiti, pošto je to smena koja na levoj strani ima <Expr>, na desnoj strani je <Term> i mi tu ne moramo instancirati poseban objekat Value, nego samo preuzimamo referencu, pošto smo već u okviru Term instancirali takav objekat. Znači ponekad možemo preuzeti postojeću referencu. Ovde pošto nam treba novi međurezultat na bazi postojeća dva, od ova dva postojeća objekta Value njihovim procesiranjem pravimo novi objekat Value.

U ovom primeru nas sličan način kao pre, moramo zadati komande za prevođenje i pokretanje. Prevodimo samo onu Calc.java glavnu klasu Java kompajlerom, ali to će prevesti i sve ostale generisane Java fajlove. Konačno, pokretanje je na potpuno isti način, samo sada je ime paketa ast. Ulaz je 3 + 5 i ono što će ispisati ova varijanta rešenja je prvo finalni izlazni rezultat 8, a zatim kao posledica poziva toString() metoda nad korenom apstraktnog sintaksnog stabla, linearizovanu tekstualnu predstavu stabla u obliku nekakve hijerarhijske liste, gde vidimo da je koren stabla Addition, unutar toga na prvom nivu imamo SingleExpr add SingleTerm. To odgovara upravo ovom objektnom dijagramu. Add nije poseban čvor, nego ispisuje vrednost ovoga membera. Taj automatski generisan toString() metod je dovoljno pametan da azna da ispiše ovo "add" kao vrednost sadržaja čvora. Jedino što ne zna da ispiše je vrednost Value objekta, pošto toString() metodi ne pozivaju value.toString, to je neko implementaciono ograničenje. Ona osmica što ispisana je posledica toga što smo mi eksplicitno ispisivali za koreni čvor, međutim ponegde na nivou ovih konstanti, to je, pošto nema onog nižeg dela parserskog stabla, to je isto kao i ovaj addop direktno member u generisanom kodu, pa će ispisati konstante 3 i 5.



U zaključku, postavlja se pitanje da li je varijanta sa apstraktnim sintaksnim stablom, koja deluje na prvi pogled, znatno složenije nego sa atributima i akcijama, bar u ovom primeru, vredna truda? Za ovaj prost primer deluje da je ovo dosta komplikovanije, ali kod ozbiljnih primena, svaki semantički prolaz odvojio bi se pisanjem posebnog visitora i onaj .cup fajl ne bi se ni na koji način opterećivao. Vredi još napomenuti da je ovo bio samo najelementarniji primer korišćenja alata ast-cup, a znatno složeniji primer, to je praktično kompletan kompajler sa sintaksno-semantičkom analizom i generisanjem koda za jedan podskup jezika mikroJava, nalazi se u okviru vežbi na sajtu kursa.

Tabela simbola

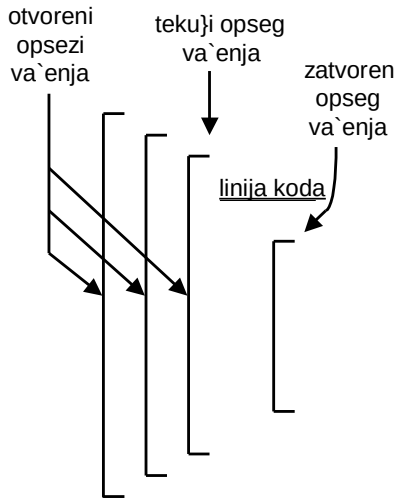
U ovoj lekciji upoznaćemo se sa ulogom, funkcijama i strukturom tabele simbola u kompajleru. Tabela simbola je mehanizam koji omogućava pamćenje različitih informacija o identifikatorima i deljenje tih informacija među fazama kompajlera. Svaki put kada se identifikator deklariše ili koristi, tabela simbola omogućava pristup prikupljenim informacijama o tom identifikatoru. Tabela simbola se najviše koriste tokom provere tipova (semantičke analize), ali ih takođe koriste i druge faze kompajlera za pamćenje i kasnije preuzimanje informacija o tipovima, promenljivima, procedurama i drugim programskim konstruktima.

Tekući opseg važenja imena

Kada se razmatraju identifikatori to jest imena u programu značajni pojmovi su opseg važenja imena i pravila vidljivosti imena u programu.

Opseg važenja imena je deo listinga programa u kojoj je neko ime vezano za određenu deklaraciju (a u vreme izvršavanja za određenu memorijsku lokaciju)**.** Bilo koja linija koda sadrži se unutar jedne programske jedinice ili više programskih jedinica koje definišu opseg važenja imena. Većina savremenih jezika je *blokovski strukturirana* - dozvoljava da opseg važenja imena bude ugnežđen.

Tekući opseg važenja je opseg važenja imena definisan trenutnim najdubljim ugnježđenjem. *Otvoreni opsezi* su tekući opseg važenja i svi opsezi koji okružuju tekuću liniju koda. *Zatvoreni opsezi* su opsezi važenja imena definisani bilo kojom programskom jedinicom koja ne okružuje posmatranu liniju izvornog koda. Tekući opseg važenja nije fiksan već se menja dinamički zavisno od tačke posmatranja.



U odnosu na tekuću liniju koda, svaki opseg važenja čiji početak je pre tekuće linije, a završetak posle tekuće linije je otvoren. Najugneždeniji od tih opsega, koji neposredno okružuje tekuću liniju, zove se tekući opseg. Opsezi kod kojih se i početak i kraj nalaze ili oba pre, ili oba posle tekuće linije koda su zatvoreni opsezi.

Statička pravila vidljivosti imena

Pravila vidljivosti imena definišu način na koje se neka određena upotreba imena povezuje sa nekom od definicija toga imena. Većina savremenih programskih jezika koristi statička pravila vidljivosti imena, koja su poznata svakom programeru.

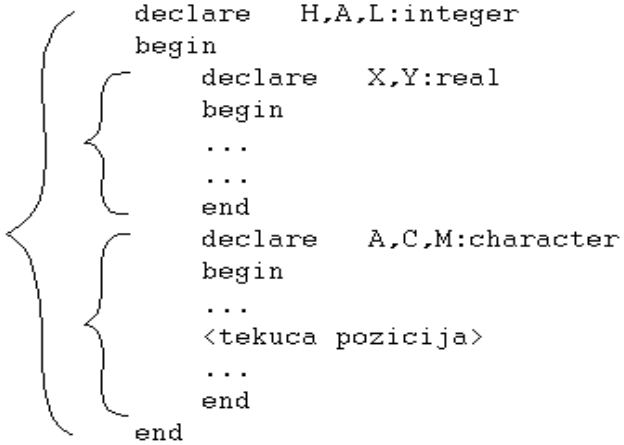
Ako su na snazi statička pravila vidljivosti imena, u bilo kojoj tački izvršnog programa pristupačna su samo imena deklarisana u tekućem opsegu važenja i u otvorenim opsezima važenja koji sadrže tekući opseg važenja.

Ako je ime deklarisano u više od jednog opsega važenja, za interpretaciju tog imena koristi se deklaracija čija je dubina najveća i najbliža tački obraćanja. Nove deklaracije se dozvoljavaju samo u tekućem opsegu važenja.

Očigledna posledica ovih pravila je da pri zatvaranju nekog opsega, sve deklaracije zadate u njemu postaju nepristupačne.

Primer

Prikazani primer pseudokoda sadrži tri opsega sa deklaracijama. U odnosu na tekuću poziciju otvoren je spoljni opseg i drugi unutrašnji opseg.



```
declare   H,A,L:integer
begin
    declare   X,Y:real
    begin
    ...
    ...
    end
    declare   A,C,M:character
    begin
    ...
    <tekuca pozicija>
    ...
    end
end
```

Na mestu tekuća pozicija u programu vidljiva su sledeća imena: A,C,M:character, H,L,:integer. Nisu vidljivi X i Y jer je njihov blok zatvoren, dok se vidi A iz treće deklaracije a ne iz prve zbog drugog pravila vidljivosti.

Dinamička pravila vidljivosti imena

Za razliku od statičkih pravila vidljivosti, gde se uvidom u listing programa može otkriti veza imena sa deklaracijom, kod *dinamičkih pravila* vidljivosti tok izvršavanja programa određuje vezivanje upotrebe promenljive sa deklaracijom. Važi da se ime vezuje za *najsveži ju aktivnu deklaraciju* u vreme izvršavanja programa.

Samo neki dijalekti Lispa, Perl imaju ovakva pravila, svi ostali programski jezici koriste statička pravila.

Primer

Primer koji ilustruje dinamička pravila vidljivosti imena sadrži dve različite deklaracije promenljive a. Jedna je gloalna, a druga je lokalna za proceduru second. Procedura first sadrži upotrebu a i postavlja se pitanje na koju deklaraciju se odnosi ta upotreba a.

Izvršavanje programa daje različit rezutat ukoliko pretpostavimo dinamička pravila vidljivosti, u odnosu na slučaj kada važe statička pravila vidljivosti.

```
var a : integer;
procedure first
    a := 1;
end first;
procedure second
  var a : integer;
    first();
end second;
begin
    a := 2;
    second();
    write_integer(a);
end;
```

Dodela a:=2 u glavnom programu odnosi se na globalno a. Poziv second() aktivira lokalnu deklaraciju a u second. Poziv first() i dodela a:=1 menja lokalno a. Po povratku iz second deaktivira se lokalno a; write ispisuje globalno a, to jest vrednost 2.

U slučaju statičke vidljivosti, a:=1 odnosi se uvek na globalno a, tako da se na kraju ispisuje 1.

Organizacija tabele simbola

Razmotrićemo različite moguće varijante interne organizacije tabele simbola kao strukture podataka prilikom njenog dizajniranja i implementacije. Postoji nekoliko dizajnerskih alternativa u odnosu na opsege važenja imena i u odnosu na vrstu kolekcije koji se koristi za pamćenje imena unutar jednog ili više opsega.

U zavisnosti od opsega važenja postoje dva pristupa realizaciji tabele simbola:

1. pojedinačna tabela simbola za svaki opseg važenja
2. jedinstvena tabela za sve opsege važenja

Pojedinačne tabele za svaki opseg važenja

Kod pojedinačnih tabela simbola za svaki opseg važenja stek mehanizam je pogodan za realizaciju. Pri otvaranju novog opsega stavlja se tabela simbola na stek, a po zatvaranju opsega skida se sa steka. Umesto cele tabele može se koristiti i samo ukazatelj na tabelu. Detaljnije ćemo proučiti ovu varijantu kod tabele simbola za Mikrojavu. Nedostaci ovog pristupa su:

i. Ako želimo da pristupimo nekom nelokalnom imenu moramo pristupati većem broju tabela simbola, pogotovo ako želimo da pristupimo nekom globalnom imenu.

ii. Ako su tabele realizovane kao heš tabele, za svaki opseg važenja se rezerviše memorijski prostor za tabelu simbola. Na duboko ugnježdenim nivoima gde ima malo imena gubitak

prostora je najveći (nije veliki problem danas kada računari imaju puno memorije). Ovaj problem se ne javlja ako su tabele organizovane u obliku binarnog stabla.

Jedinstvena tabela za sve opsege

Sva imena koja se pojave u toku prevodjenja se smeštaju u jedinstvenu tabelu simbola. Pošto se može desiti da u različitim opsezima važenja imamo ista imena, svakom imenu se pridruži opseg važenja za koji je to ime definisano. Posebna globalna promenljiva „broji" opsege, inkrementira se pri otvaranju opsega.

Realizacija jedinstvene tabele u vidu heš tabele

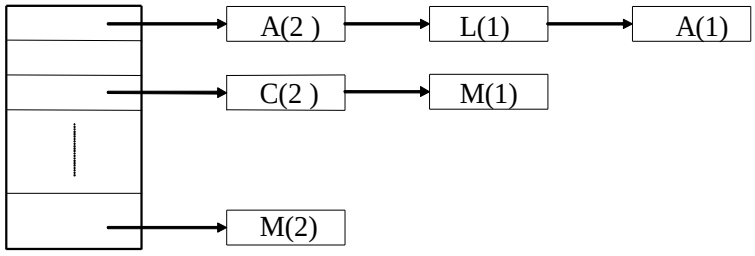
Na slici je prikazan primer gde je za tabelu simbola iskorišćena heš funkcija sa kolizionim lancima. Sa stanovišta tabele simbola važni događaji tokom procesiranja programa su otvaranje opsega, zatvaranje opsega, deklarisanje idenfitikatora i upotreba idenfitikatora.

Prilikom otvaranja opsega, opsegu se dodeljuje jedinstveni celobrojni indeks. Identifikator se prilikom obrade deklaracije, heš funkcijom preslikava u indeks u heš tabeli i vezuje na početak lanca kolizije. Uz identifikator se pamti i indeks programskog opsega u kome je deklarisan. Različite deklaracije istog identifikatora na primer A iz prvog opsega (indeks 1) i A iz drugog ugneždenog opsega (indeks 2) biće zabeležene u okviru jednog istog lanca kolizije i to tako da se na početku lanca kolizije nalazi deklaracija iz najugneždenijeg opsega. Pri korišćenju identifikatora, indeksiranjem heš tabele pristupa se odgovarajućem lancu kolizije. Dovoljno je pretražiti lanac od početka do pojave prvog zapisa koji sadrži traženi identifikator.

Prilikom zatvaranja opsega, potrebno je proći kroz sve lance kolizije u heš tabeli i sa njihovih početaka ukloniti zapise koji se odnose na tekući opseg koji se upravo zatvara.

Primer

```
{ int M, A, L;
      ...
       {float X, Y;
        ...
       }
       {char A, C, M;
        ...
        //trenutna pozicija
       }
}
```



heš tabela

Prednosti heš tabele simbola su:

i. Brži je pristup simbolu jer se pretražuje samo jedna tabela, i obično su ti traženi simboli na početku lanaca.

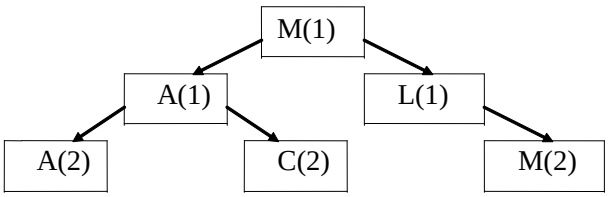
ii. (manje važno) Memorijski prostor je efikasnije iskorišćen jer imamo samo jedno zaglavlje tabele, ali je taj dobitak umanjen zato što se uz svako ime dodaje prostor za opseg važenja.

Realizacija jedinstvene tabele simbola korišćenjem binarnog, leksikografski uređenog, stabla

Leksikografska uređenost stabla znači da kada posmatramo neki čvor stabla, čvorovi u njegovom levom podstablu imaju u abecednom redosledu manja imena, a čvorovi u desnom podstablu abecedno veća. Opet koristimo jednoznačne numeričke identifikatore za svaki opseg koje pamtimo u čvorovima stabla uz imena.

Primer

```
{ int M, A, L;
      ...
       {float X, Y;
        ...
       }
       {char A, C, M;
        ...
        //trenutna pozicija
       }
}
```



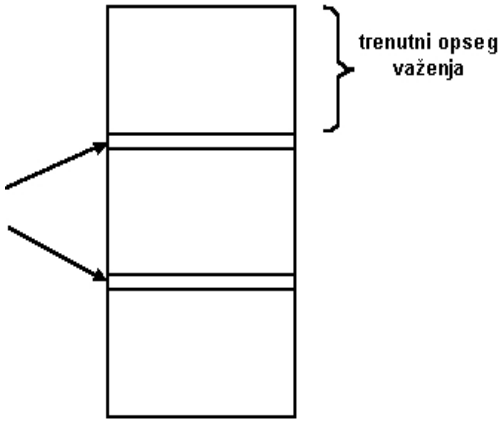
Kod ovog načina realizacije tabele simbola, svi simboli formiraju stablo. Ubacivanje novih simbola pri otvaranju novog opsega važenja i izbacivanje simbola pri zatvaranju opsega važenja zahteva pretraživanje stabla do listova, što nije baš efikasno pa se ne preporučuje ova struktura podataka ukoliko imamo jednu tabelu za sve opsege važenja. Binarno stablo se može koristiti kada imamo pojedinačne tabele za sve opsege važenja.

Razmotrimo situaciju kada ubacujemo novo ime u gornje stablo pri obradi deklaracije. Ako smo tek otvorili drugi ugnežđeni opseg, čvorovi A(2), C(2) i M(2) ne bi još bili prisutni u stablu. Kada treba ubaciti A(2), krećemo od korenog čvora, pošto je A < M u abecedi, prelazimo na levo podstablo. Upoređujemo A sa postojećim čvorom A. Radi se o istom imenu, ali različitoj deklaraciji pa moramo napraviti novi čvor. Možemo po konvenciji odlučiti da idemo u levo podstablo (kao i kada je ime manje). Pošto je A(1) bio list, dodaćemo A(2) kao novi čvor u stablo, levi sin čvora A(1). Dakle identifikatori najugneždenijeg opsega nalaze se u listovima stabla ili blizu njih. Zbog toga, kada pri korišćenju nekog imena vršimo prolaz kroz stablo počev od korena, ne možemo se zaustaviti u prvom čvoru gde dolazi do poklapanja imena iz čvora sa traženim imenom, nego treba nastaviti pretragu kroz stablo sve dok ne dođemo do lista stabla. Kada

zatvaramo opseg, moramo proći kroz celo stablo i ukloniti iz njega sve čvorove (u listovima stabla ili blizu njih) koji odgovaraju imenima iz tog opsega.

Realizacija jedinstvene tabele u vidu steka

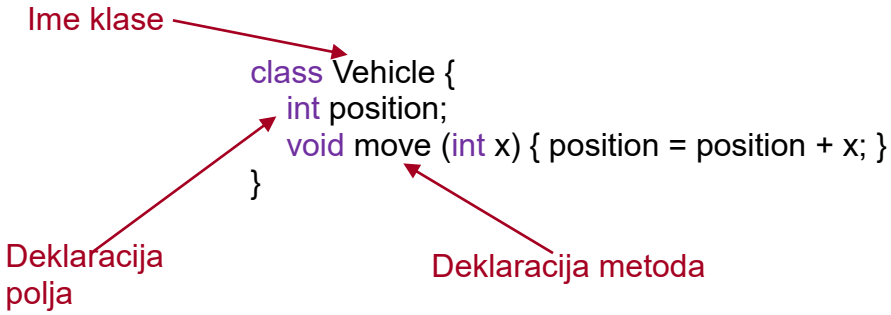
U ovom slučaju na vrhu steka se nalaze simboli iz tekućeg opsega važenja, i razdvojeni su od simbola iz prethodnog opsega važenja sa markerom. Kada se opseg zatvara sve što se nalazi na steku iznad markera se skida sa steka. Pretraživanje imena ide redom počev od vrha steka na niže, zaustavljamo se kod prvog poklapanja. Ovakva tabela je korišćena u ranim verzijama prevodilaca.



Objektno-orijentisani jezici

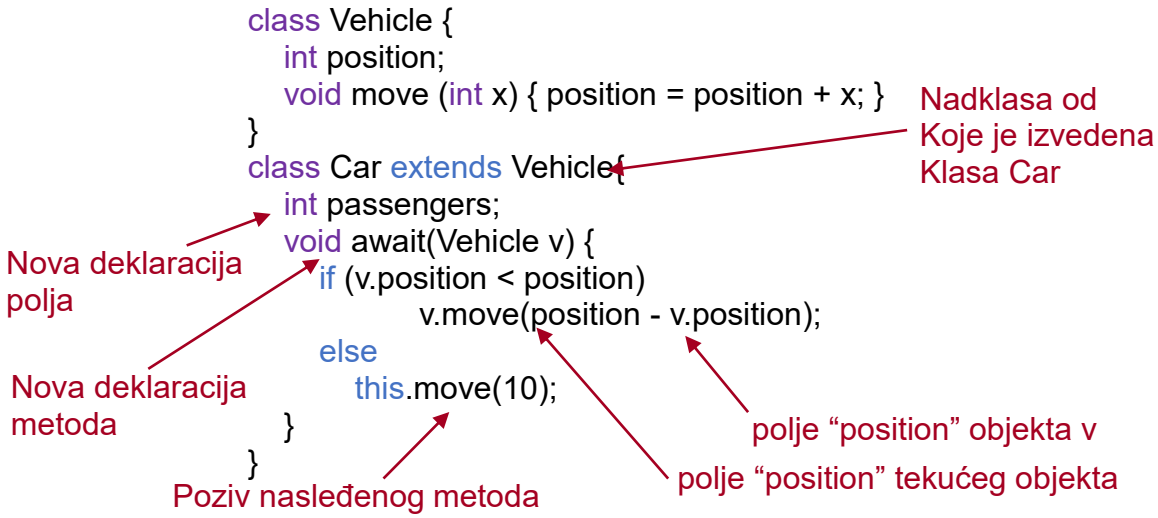
U ovom poglavlju razmotrićemo kako se sa stanovišta kompajlerske obrade realizuju neki koncepti objektno-orijentisanih jezika. Mada ovo izlaganje izlazi izvan okvira organizacije i funkcionisanja tabela simbola u kompajleru, pozicionirano je upravo ovde jer predstavlja potreban uvod da bi se razumeo dizajn tabele simbola mikrojava kompajlera čiji opis ćemo dati u narednim poglavljima.

Kao što ste već naučili, u objektno-orijentisanom programiranju, važni koncert ove vrste programiranja su klase, objekti, apstrakcija, enkapsulacija, nasleđivanje i polimorfizam. Kratko ćemo se podsetiti ovih karakteristika, pa ćemo potom videti kako se ovi koncepti realizuju u programskim prevodiocima.

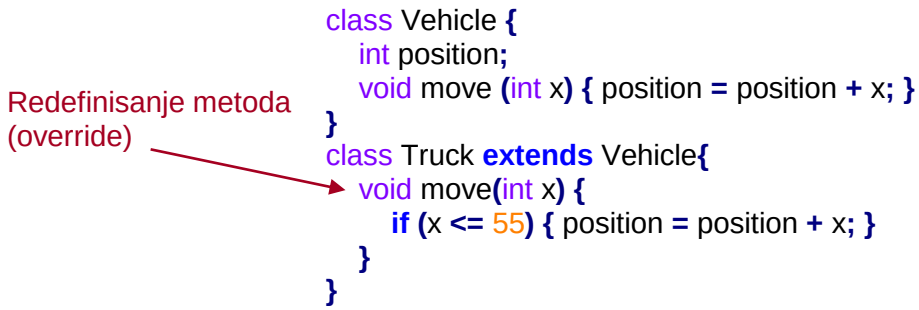


U Java programu možemo definisati klasu, u ovom primeru klasu Vehicle (vozilo). Klasa može da ima polja i metode. Klasa može proširiti, odnosno naslediti drugu klasu. U ovom primeru klasa Car proširuje klasu Vehicle. Car nasleđuje polje position i method move(), a dodaje sopstveno polje passengers i metod await(), koji odgovara nekakvom sačekivanju drugog vozila, ažuriranjem

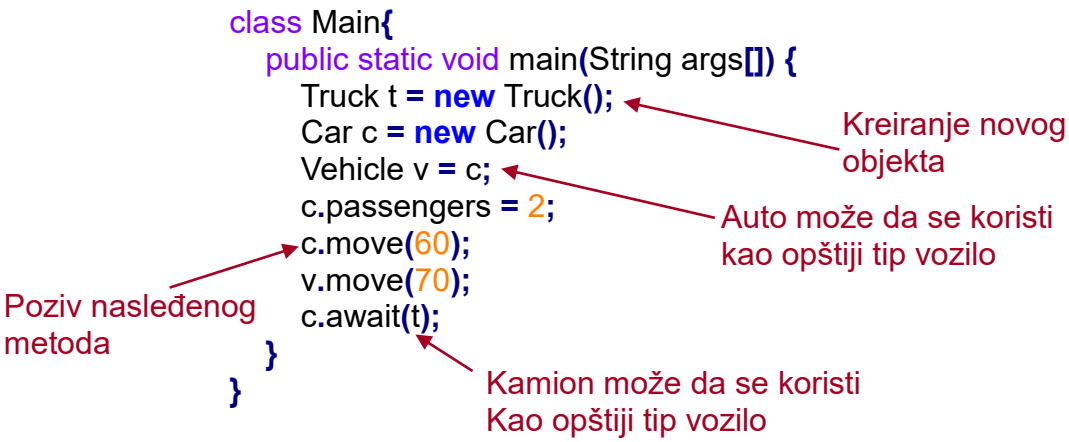
njegove pozicije. U opsegu imena metoda await(), vidljive su promenljive passengers, jer predstavlja polje klase Car, position predstavlja nasleđeno polje klase Car, v kao formalni parametar metoda await() i takođe this kao implicitni formalni parametar metoda await().



Klasa Truck takođe proširuje klasu Vehicle nasleđujući polje position i metod move(). Međutim, Truck pruža sopstvenu implementaciju metoda move(). Na taj način, on redefiniše ili overajduje nasleđeni metod.

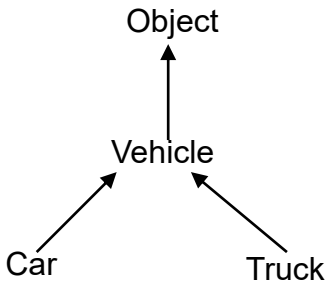


Nastavljajući ovaj primer, u metodu main() imamo upotrebu ovih klasa koje smo u prethodnim slajdovima definisali. Instanciraju se objekti t klase Truck i c klase Car, a ovi objekti mogu da se upotrebe i na mestu gde se očekuje objekat opštije klase Vehicle, na primer u pozivu c.await(t) objekat t se prosleđuje na mestu formalnog parametra v.



Kada se u ovom await() metodu klase Car, pozove v.move(), ovaj move() se odnosi na redefinisani move() iz klase Truck, a ne na originalni move() iz klase Vehicle. To je zato što Java pamti dinamički tip objekta v u vreme izvršenja, koji se razlikuje od statičkog tipa sa kojim je v deklarisaana. Znači v je deklarisana sa statičkim tipom Vehicle, a u ovom pozivu, sam objekat v je tipa Truck. Poziv standardnog, odnosno dinamičkog metoda klase se razrazrešava gledajući dinamički tip objekta.

Ključni problemi koje u kompajlerskoj obradi treba razrešiti su: Kako pristupiti poljima objekta uključujući nasleđena i polja tekućeg objekta. Takođe, kako pristupiti programskom kodu metoda, ako tekuća klasa nema definiciju metoda kako stići do nasleđenog metoda i kako se obrađuje redefinicija metoda, odnosno overajdovanje.



Hijerarhija klasa predstavlja grafički prikaz relacija nasleđivanja među klasama programa. Za onaj naš prethodnih primer, imali smo klasu Vehicle. Ako je u pitanju standardna Java, sve klase u Javi proširuju ugrađenu klasu Object. U mikrojavi toga nema, znači klasa je reprezent same sebe. lmali smo i klasu Car koja nasleđuje Vehicle i klasu Truck koja takođe nasleđuje Vehicle. Ako postoji samo jednostruko nasleđivanje, kao u ovom primeru, graf je oblika stabla, a u jeziku sa višestrukim nasledjivanjem, kao što je na primer C++ ili Python, graf je aciklički, to jest jedna klasa može imati više klasa iz kojih nasleđuje. Višestruko nasleđivanje je nešto složenije za implementaciju od jednostrukog nasleđivanja.

Memorijsko predstavljanje objekata

Što se tiče memorijske reprezentacije objekata, kod jednostrukog nasleđivanja objektima odgovaraju zapisi u memoriji. Svaka promenljiva ima polje u zapisu, a kod pristupa polju zapisa, na primer v.position, glavni problem je što v može znači u runtime-u ukazivati ili na osnovnu klasu Vehicle, ili na neku od izvedenih klasa Car ili Truck. U ovom primeru namerno instanciramo Truck i Car i onda pozivom metoda await() prosleđujemo bilo koji od tih objekata u trenutku kada se izvršava ovaj deo koda. Isti mašinski kod mora funkcionisati kada znamo statički tip objekta v, to je recimo Vehicle, ali ne znamo njegov dinamički tip.

```
class Vehicle { ...  }

class Car extends Vehicle {
    int passengers;

    void await(Vehicle v) {
      if (v.position < position) then ... }}

class Truck extends Vehicle {...}

class Main { …
    public static void main(…) {
     Truck t = new Truck();
     Car c = new Car();
     Vehicle v = c;
     c.await(t);
     c.await(c);
     c.await(v);
    }
}
```

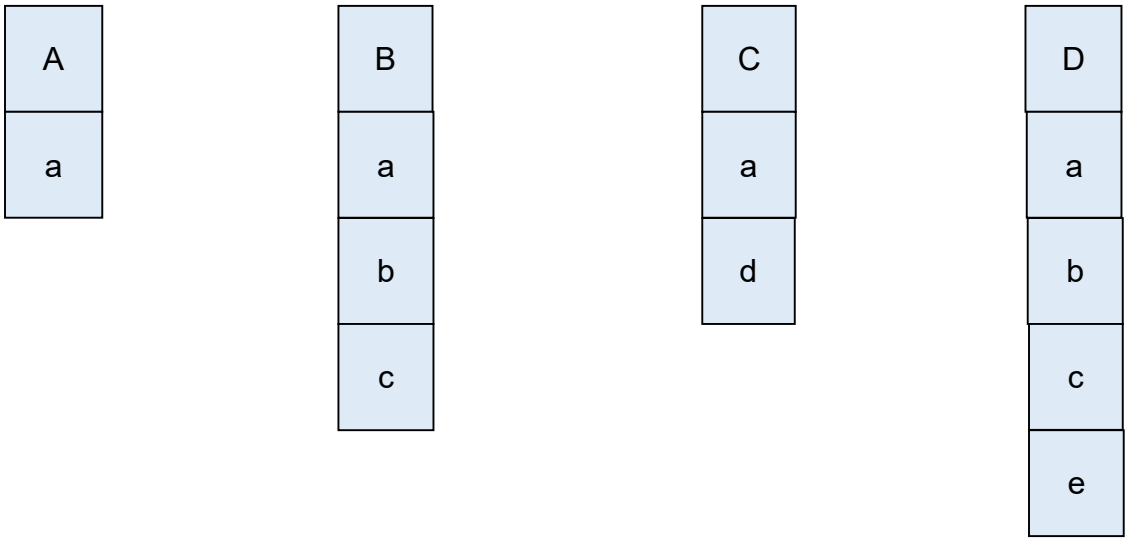
Znači kompajler ne zna šta će u runtime-u na ovom mestu doći kao objekat, mogu biti u ovom primeru Vehicle, Car ili Truck. Rešenje se može postići time da polje position mora biti na istom mestu u zapisima koji reprezentuju objekte klase Vehicle, Car i Truck. Pretpostavimo da je ovaj v u runtime-u realizovan kao pokazivač koji je smešten u neki registar R. Kompajler će prevesti v.position u mašinski kod ovog oblika, nekakva naredba load koja dohvata vrednost svog operanda i smešta, ako je u pitanju mikro java virtuelna mašina, reč je o stek mašini, tako da postoji takozvani stek izraza umesto registara, znači ide na vrh tog steka izraza. Koja vrednost ide iz memorije sa adrese R plus neka konstanta koju kompajler zna u vreme prevođenja programa? To je vrednost polja position objekta, odnosno R je početak zapisa objekta, a konstanta treba da bude pomeraj do polja position. Koju konstantu treba uzeti? Ta konstanta, taj pomeraj mora biti isti, bez obzira da li v ukazuje na Car, Truck ili Vehicle. Znači raspored polja u zapisima koji reprezentuju kola, kamione i vozila mora biti sličan.

Vrednost svakog objekta u vreme izvršavanja predstavljena je zapisom u memoriji sa poljima zapisa za svako polje u klasi. Pomeraji (ofseti) tih polja u odnosu na početak zapisa su konstantni, to jest poznati kompajleru pri prevođenju programa. U slučaju jednostrukog nasleđivanja, jednostavna tehnika proširivanja na desno, to jest ka višim memorijskim lokacijama, rešava problem pristupa poljima objekta. Znači prvo se u zapisu objekta naređaju polja koja se nasleđuju iz natklase u istom redosledu kao i u natklasi, a tek onda sopstvena polja potklase.

Pogledajmo jedan primer koji ilustruje ovaj koncept proširivanja na desno. Imamo četiri klase. Klasa A (za sada ne razmatramo metode samo polja), znači klasa A ima polje a. Klasa B proširuje A i, osim nasleđivanja polja a, dodaje sopstvena polja b i c. Klasa C takođe proširuje klasu A i, pored nasledjenog polja a, dodaje svoje polje d. I klasa D konačno proširuje klasu B, znači nasleđuje sve što B ima od polja i dodaje sopstveno polje e. Kako izgleda memorijski layout objekata ovih klasa?

```
class A                 { int a = 0; }
class B extends A       { int b = 0;
                            int c = 0; }
class C extends A       { int d = 0; }
class D extends B       { int e = 0; }
```

Znači u runtime-u u operativnoj memoriji računara, za objekat klase A imaćemo jedan ovakav zapis. U zapisima objekata, na početku zapisa uvek stoji takozvani deskriptor ili identifikator klase, da se zna kojoj klasi objekat pripada, znači to je ona dinamička, runtime klasa objekta. Ostatak zapisa objekta čine vrednosti polja objekta, znači ovde što piše a, to je samo simbolički označeno da je u pitanju polje a, ali tu u runtime-u stoji naravno neka konkretna celobrojna vrednost, inicijalno 0.



Kako izgleda zapis objekta B? Znači ima svoj identifikator klase kao početnu komponentu zapisa, a zatim redom idu vrednosti polja. Jako je važno da prvo dođe nasleđeno polje a na istom mestu kao i u zapisu objekta a, zatim sopstvena polja b i c. Slično tome, klasa C koja isto nasleđuje A mora imati na prvom mestu a, a zatim sopstvenog polje d. I konačno klasa D koja nasleđuje klasu B, mora imati u svom zapisu ova nasleđena polja u istom redosledu znači na istim offset-ima a, b, c i zatim dodaje sopstveno polje e.

Prevođenje metoda klase

Kako se realizuje prevođenje poziva metoda klase, kako statičkih metoda, tako i dinamičkih? Rezultat prevođenja metoda je mašinski kod na određenoj memorijskoj adresi. Na mestu pozivanja

metoda moramo odrediti na koju adresu treba izvršiti skok. Java ima statičke i dinamičke metode. Za razrešavanje poziva statičkih metoda, dovoljno je odrediti statički tip objekta čiji se metod poziva, a za razrešavanje poziva dinamičkih metoda treba poznavati dinamički tip pozvanog objekta.

Detalje ćemo videti razmatrajući primere. Ako deklaracija metoda sadrži ključnu reč static onda mašinski kod koji realizuje poziv metoda, na primer c.foo(), zavisi od tipa sa kojim je promenljiva c deklarisana, a ne od tipa objekta koji je referisan sa c u vreme izvršenja.

```
  class A {
     static void foo (int x) {…}
     static void bar (int x) {…}
  }
  class B extends A {
     static void foo (int x) {…}
  }

Class Main {
  public static void main(…){
     A a= new A();
     A b = new B();
     B c = new B();
     a.foo(3);     /* calls foo in class A */
     b.foo(3);     /* calls foo in class A */
     c.bar(3);     /* calls bar in class A */
     c.foo(3);     /* calls foo in class B */
  }
}
```

Na primer, vidimo da u klasi A imamo statički metod foo() i statički metod bar(). a klasa B koja proširuje A, takođe ima statički metod foo(). Ne možemo reći u ovom slučaju da ovaj foo() overajduje, redefiniše ovaj gornji foo(), jer ovde pri statičkim metodima ne funkcioniše polimorfni mehanizam, nego pravilno je reći da u kontaktu klase B sakriva ovaj ovde gornji foo() u kontekstu klase A. Znači sve se razrešava statički u vreme prevođenja i vidi iz listinga programa.

U ovom primeru za poziv a.foo(), da bismo znali koje foo() se poziva, dovoljno je pogledati statički tip promenljive a, znači poziva se foo() klase A. Ovde vidimo da promenljiva b ima statički tip A, a dinamički objekat je tipa B. Međutim, poziv b.foo() se uvek razrešava koristeći statički tip, tako da se opet poziva, bez obzira što je objekat klase B, poziva se foo() iz klase A. I konačno, promenljiva c ima i statički i dinamički tip isti, klase B, tako da tek u ovome c.foo() poziva se foo() iz klase B. Genalno, da bi preveo poziv statičkog metoda, tokom semantičke analize prevodilac određuje statički tip objekta čiji se metod poziva i, naravno, listu metoda koji su prisutni u svakoj klasi. Zatim utvrđuje najbliži metod na gore u hijerarhiji klasa sa datim imenom, znači krene od tekuće klase, pa ako nema u tekućoj klasi krene do klase iz koje ona neposredno nasleđuje i tako dalje, i zatim ubacuje neposredan poziv utvrđenog metoda.

Prethodno opisana tehnika nije primenljiva na dinamičke metode. Pogledajmo sličan primer, sada je izostavljena ključna reč static i u Javi podrazumevano metodi koji se definišu u okviru neke klase su dinamički.

```
class A {
    void foo (int x) {…}
    void bar (int x) {…}
}
class B extends A {
    void foo (int x) {…}
}

Class Main {…
  public static void main(…){
     A a = new A();
     A b = new B();
     A c;
       if (long_and_tricky_computation)
         c = a;
        else c = b;
     c.foo(3);
  }
}
```

Znači sada su foo() i bar() iz klase A i takođe ovaj drugi foo() iz klase B dinamički metodi. Ovaj foo() iz B, pošto klasa B proširuje klasu A, slično kao u prethodnom primeru, rerdefiniše odnosno overajduje ovaj foo() iz klase A. Vidimo da u glavnom programu instanciramo jedan objekat klase A, jedan objekat klase B i takođe koristimo referencu c koja je statičkog tipa A, znači osnovne klase. Promenljivoj c može da se dodeli dinamički, to je ovde simbolizovano ovim "if long and tricky computation", da ukazuje na objekat a, ili da ukazuje na objekat b. Kompajiler ni na koji način ne može da zna, na ovom mestu, da li c ukazuje na a ili na b, to se tek u runtime-u određuje. Postavlja se pitanje za poziv c.foo() kako se razrešava? Metod koji se poziva zavisi od dinamičkog zipa objekta i ne mora biti poznat, i uopštem slučaju i nije, tokom semantičke analize u kompajleru. U vreme izvršavanja programa mora se odrediti gde će se skočiti. Zbog toga, u vreme izvršavanja programa treba da postoji podrška za to odlučivanje, koja je realizovana tako što objekat, osim što ima vrednosti svojih polja, ima i pokazivač na tabelu svojih metoda, takozvanu v-tabelu ili virtuelnu tabelu. U vreme prevođenja generiše se kod koji će u runtime-u pretražiti tu v-tabelu objekta, pronaći lokaciju metoda u tabeli i izvršiti skok na početak tela metoda.

Evo da vidimo na konkretnom primeru kako je to realizovano. Ovaj primer odnosi se na jednostruko nasleđivanje i ima definisane četiri klase A, B, C i D. U ovom primeru, pored polja, imamo dodate svakoj klasi i određene metode. Klasa A ima polje a i metod f(). Klasa B proširuje klasu A i dodaje sopstveni metod g(). Klasa C proširuje klasu B i overajduje metod g(). Svi metodi su dinamički. Klasa D proširuje klasu C, znači nasleđuje polje a još iz klase A i takođe metod f() iz klase A nasleđuje, ali ga redefiniše (overajduje) i, takođe, nasleđuje iz klase C njegov metod g(). Kako je to sve u memoriji tokom izvršavanja programa reprezentovano?

```
class A extends Object    { int a = 0; void f () }
class B extends A         { void g () }
class C extends B         { void g () }
class D extends C         { int b = 0 ; void f () }
```

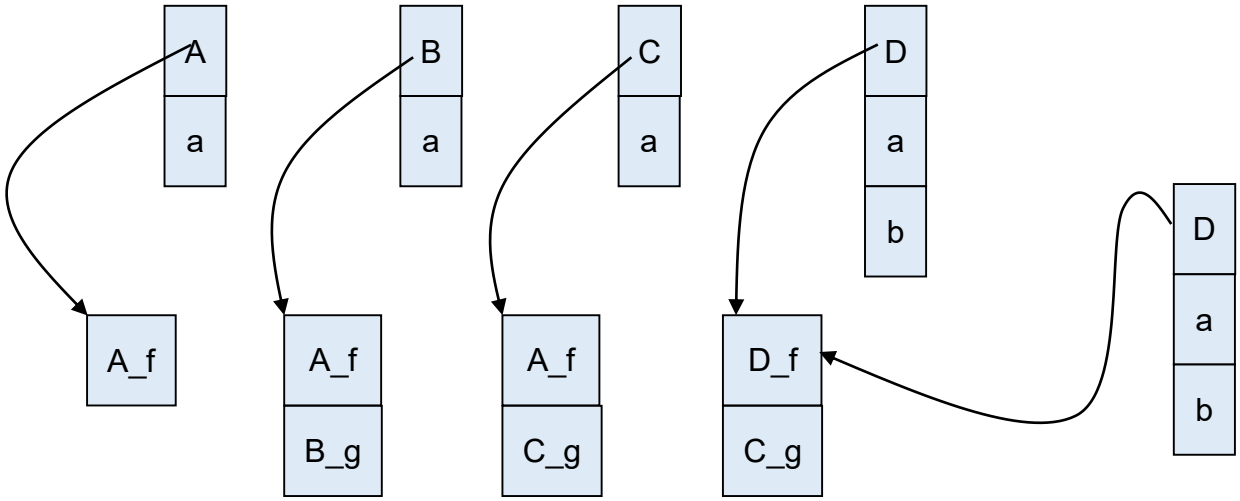
Imamo objekat klase A koji ima svoj identifikator. odnosno deskriptor klase i vrednosti polja, znači ovo je objekat klase A. Dodatno, imamo jednu strukturu (v-tabelu) koja ne mora da se pamti na istom mestu u memoriji. Važno je primetiti da svaka instanca klase A, svaki objekat ima jedan vrednosni zapis i oni se tipično nalaze u dinamičkoj memoriji u onom delu memorije koju u vreme

izvršavanja program može slobodno da se alocira i dealocira, a za v-tabelu određene klase dovoljna je jedna instanca na nivou klase i svi objekti te klase dele tu instancu. Znači iz svakog objekta ukazivač ide na ovo isto mesto i zato ovo može biti u nekoj drugom delu memorije u toku izvršavanja programa, to je tipično statička memorija. Za klasu A. njena virtuelna tabela sadrži pokazivač na početak implementacije ovoga metoda f().

Objekat klase B ima polje za ovo nasledjeno a na istom mestu kao i u osnovnoj klasi i u virtuelnoj tabeli ima dva pokazivača. Prvi pokazivač mora da bude na istom mestu i iste vrednosti kao u osnovnoj klasi, znači ovaj A_f() jer se prvo naređaju svi nasleđeni metodi, a zatim se dodaju sopstveni metodi, znači pokazivač B_g na metod g() definisan u okviru klase B.

Objekat klase C ima nasledjeno polje a i u svojoj virtuelnoj tabeli nasleđuje metod f() iz klase A, a overajduje g() iz klase B, tako da na istom mestu gde je u klasi B bio pokazivač na g(), ovde imamo pokazivač na overajdovani metod C_g.

Konačno, klasa D nasleđuje a, potom dodaje sopstveno d, a u svojoj virtuelnoj tabeli ima sopstvenu, overajdovanu verziju metoda f() na istom indeksu gde je metod f() iz klase A. i nasleđuje direktno, bez overajdovanja, metod g() iz klase C.



Ovde je prikazan još jedan objekat klase D, oni se naravno instanciraju sa new D u runtime-u, tako da vidimo da svaki objekat ima sopstveni zapis sa vrednostima polja, ali svi oni dele jedinstvenu v-tabelu.

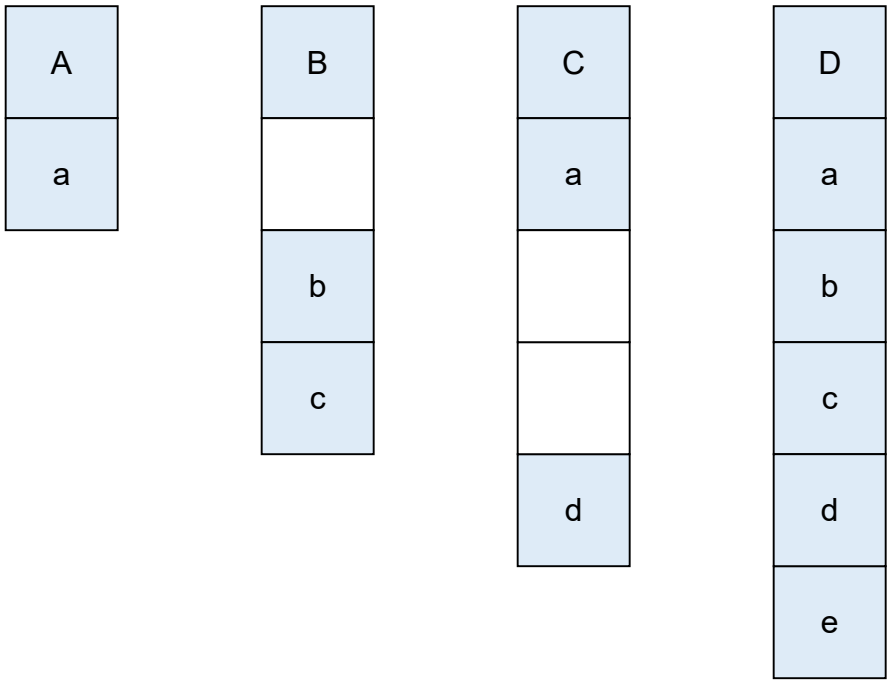
Realizacija višestrukog nasleđivanja

Višestruko nasleđivanje složenije je za implementaciju od jednostrukog, jer za objekte potklasa proširenje nadesno ne funkcioniše. Recimo, ako neka klasa C nasleđuje i od klase A i od klase B, ne možemo na isto mesto na početku zapisa tog objekta staviti i promenljive klase A i promenljive klase B. Za predstavljanje objekata u kontekstu višestrukog nasleđivanja, neophodna je globalna analiza, odnosno moraju se uzeti u obzir sve postojeće klase.

Razmotrićemo to na sledećem primeru. Primer sadrži definicije klasa A, B, C i D i za sada razmatramo samo polja. Klasa A ima polje a. klasa B ima polja b i c. Klasa C proširuje klasu A, znači nasleđuje polje a i dodaje svoje polje d. A klasa D je jedina koja višestruko nasleđuje i iz klase A i iz klase B i iz klase C i dodaje još sopstveno polje e.

```
class A                 { int a = 0; }
class B                 { int b = 0;
                          int c = 0; }
class C extends A       { int d = 0; }
class D extends A,B,C   { int e = 0; }
```

Kako izgledaju memorijski zapisi objekata? Za objekat klase A imamo samo vrednost polja a. Za objekat klase B imamo polja b i c. Za objekat klase C imamo nasleđeno a i dodato sopstveno polje d i konačno, objekat klase D nasleđuje sva polja ovih prethodnih klasa, znači mora da ima naređano a, b, c, d i sopstveno polje e.

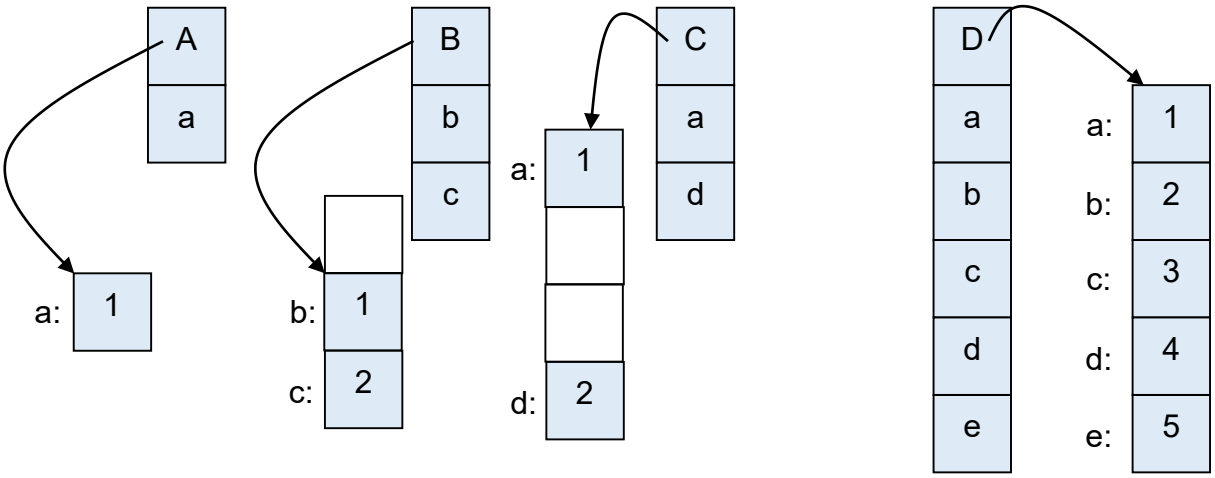


Ovde je naravno zanimljivo, to se uvek razrešava posebno u semantici tih jezika sa višestrukim nasleđivanjem, što na primer klasa A i klasa C imaju polje a i onda se postavlja pitanje u klasi D, da li treba da se pamte odvojeno pa te vrednosti, iili da postoji samo jedna vrednost kao što je ovde naznačeno. Druga stvar, zašto je potrebna globalna analiza? Zato što vidimo da memorijski raspored polja objekta D određuje i rasporede polja ovih drugih objekata. Znači da bi polja u objektu D dobila jednoznačne pozicije, pomeraje, koji su isti kod nasleđenih polja kao i u natklasama, mi moramo u objektima drugih klasa konkretno B i C, ostaviti rupe, znači neiskorišćene slotove, da bi moglo svako polje objekta D dobiti svoj jednoznačan Indeks.

Kojim algoritmom se određuju ovi pomeraji polja u svakom objektu? Memorijsko predstavljanje objekata može se odrediti algoritmom bojenja grafa, poznatin matematičkin algoritmom. Po jedan čvor grafa se formira za svakog posebno ime polja. Zatim čvorove povezujemo granama, za polja koja koegzistiiraju u istoj klasi, putem nasleđivanje ili na neki drugi način. Cilj algoritma je da čvorovi koji su međusobno povezani granama ne budu obojeni istom bojom, odnosno konkretno kada to interpretiramo za naš problem, to znači da polja koja koegzistiraju u istoj klasi, da svako

dobije svoj jedinstveni pomeraj, Znači ti pomeraji u zapisu su rešenje algoritma, to znači oni odgovaraju bojama grafa.

Problem sa ovim pristupom je potencijalno ostavljanje praznog prostora u sred objekata pošto se ne može uvek obezbediti da ofseti N polja uvek pokrivaju prvih N pozicija u zapisu. U konkretnom primeru, zbog klase D imamo rupe u objektima B i C iako te dve klase nemaju nikakvu direktnu povezanost. Znači to je problem interne fragmentacije memorije programa. Jedno od rešenja za taj problem je sledeće: da bi se eliminisala prazna mesta u objektu, možemo spakovati vrednosti polja objekta neposredno jednu za drugom, a delegirati deskriptoru klase da saopštava gde se koje polje nalazi, znači da pamti ofsete polja. Ofseti polja u samom deskriptoru klase određuju se kao i ranije, metodom bojanja grafa. Na ovaj način, prazan prostor preselio se u deskriptor klase, što je poboljšanje jer svaka klasa ima samo jedan deskriptor, a potencijalno veliki broj instanci.



Vidimo evo u primeru, da smo sad u deskriptor klase ubacili indeks polja. Recimo objekat B koji je ranije imao rupu pre b, sad se ta rupa prosleđuje u deskriptor klase koji je jedan na nivou klase, a u samom objektu je iskorišćen direktno prostori i u samom deskriptoru onda pišu ti ofseti kako doći do realnih vrednosti polja. Znači sada neiskorišćen prostor ostaje na nivou klase.

Ovo je naravno memorijski bolje zato što deskriptora klasa ima neki određen broj, onoliko koliko ima klasa u programu, a potencijalno objekata u programu može biti neuporedivo više, znači gubi se mnogo manje memorije. Međutim, cena koju plaćamo je nešto sporiji pristup polju klase. Znači pristup polju klase sada zahteva da prvo pročitamo pokazivač na deskriptor klase iz objekta, znači pratimo ovaj pointer, zatim pročitamo vrednost pomeraja iz polja deskriptora i kao treće, pročitamo ili upišemo podatak na taj pomeraj. U prvoj varijanti dovoljna je bila samo jedna naredba, da se pristupi polju objekta kad znamo adresu objekta.

Problem je što u modernim jezicima, globalna analiza pravi dodatne probleme. Mnoge aplikacije koriste dinamički povezane biblioteke. U toj situaciji kompajler ne zna sve klase u programu, tako da ne može lako da izvrši globalnu analizu. Tako da neki jezici pribegavaju naravno jednostrukom nasleđivanju kao jednostavnijoj varijanti. Ako želimo višestruko nasleđivanje, a da izbegnemo globalnu analizu, poznavanje svih klasa u programu, možemo upotrebiti hešing u pristupu poljima objekta, znači umesto globalnog bojenja grafa u svaki deskriptor klase ide hash tabela koja mapira

imena polja u ofsete i imena metoda (jer i za metode mora da se primeni sličan mehanizam) u adrese instanci metoda i onda se izbegava globalna analiza. Detalji ovog rešenja sa hešingom mogu se naći u knjizi profesora Apela. Takođe, objektno-orijentisani jezici imaju još neke specifičnosti koje mi samo pominjemo mikrojava to nema.

Ostale implementacione specifičnosti OO jezika

Sve ostale specifičnosti mogu da se reše određenim mehanizmima u prevođenju i izvršavanju. Radi enkapsulacije lokalnog stanja objekta, Java ima kvalifikatore private, protected i public. Znamo da privatni metodi polja ne mogu biti korišćeni izvan klase gde su definisani, protected mogu biti korišćeni samo u klasi i njenim nasleđenim klasama i tako dalje. Podrška za ovo je relativno laka i svodi se na semantičku analizu, znači ove informacije treba čuvati u tabeli simbola i proveravati u semantičkoj analizi.

Objektno-orijentisani jezici imaju operatore provere tipa u run-time-u. U Javi imamo instanceof koji ispituje da li je neki objekat x instanca klase C ili neke njene potklase, a u C++, ako je uključen mehanizam čuvanja informacija o tipu u toku runtime-a, takozvani runtime type information (skr. RTTI), postoji sličan typeid operator.

Takođe, objektno-orijentisani jezici imaju mehanizam sigurne konverzije tipa. Recimo, neka u programiranju imamo promenljivu x koja je statičke klase C, a sam objekat u runtime-u je stvari neke klase D koja je izvedena iz C. Obično naravno kada koristimo x, onda koristimo ono što je definisano na nivou C. Međutim, kada mi znamo da je x određene izvedene klase i kada hoćemo da baš pristupimo specifičnom sadržaju te izvedene klase, to inače nije dobar stil u objektno-orijentisanom programiranju, ali nekad zatreba, onda znači treba uraditi konverziju tipa (cast). U Javi napišemo ovako (D)x, tako nešto možemo napisati i u C++, ali onda bi se radilo o nesigurnoj konverziji za koju kompajler bi slepo verovao nama da je x sigurno u runtime-u tipa D što može, ali ne ne mora biti ispunjeno. Međutim ako želimo sigurnu konverziju koja prvo proverava da li je x zaista odgovarajućeg tipa, pa tek onda ovaj radi ono što treba znači imam dynamic_cast i navodi se znači dinamička klasa. Potrebno je poznavanje tipa objekta u runtime-u da bi se ovakvi operatori implementirali.

Možemo razmotriti implementaciju instanceof operatora odnosno provere tipa. Pošto svaki objekat pokazuje na svoj deskriptor klase, adresa deskriptora klase može služiti za identifikaciju tipa. Međutim ako je x instanca klase D, a D proširuje C, onda je x takođe instanca klase C. Kako može da izgleda implementacija instanceof operatora?

```
        T1 ← x.descriptor;
L1:     if t1 = C goto true;
        t1 ← t1.super;
        if t1 = null goto false;
        goto L1;
```

Ovo je kod koji se izvršava, koji kompajler treba da prevede u mašinski kod koji se onda izvršava u runtime-u na mestu gde upotrebljen ovaj instanceof operator. Znači krene se od deskriptora objekta x i sad se ispituje da li je taj deskriptor upravo tipa C. Ako jeste, onda smo završili posao i vraćamo true. Međutim, ako ne dolazi do poklapanja u deskriptoru klase objekta x za koji ne

znamo da kažemo koje je klase (to jest, poznato je tek u runtime-u), mora da postoji polje super koje ukazuje na natklasu kod jednostrukog nasleđivanja. Ovo super će ili biti null vrednosti ilii će postojati jedinstvena natklasa. Ažuriramo t1 da sad ukazuje na natklasu, ako ona ne postoji vraćamo False. Znači da x nije instanca C. Ako t1 nije null, vraćamo se na ovu proveru da nismo sada došli do klase C. Ako jesmo vraćamo true. Ako nismo, gledamo da li postoji još neka natklasa za trenutnu klasu. Ako postoji, opet se vraćamo na ispitivanje da li je to C i tako dalje. U nekom trenutku izlaz će biti ili na liniji 2, ili na liniji 4.

Tabela simbola u mikrojavi (objekti, opsezi)

Konkretizovaćemo priču o konstrukciji tabele simbola na primeru mikrojave i ovo poglavlje odnosi se na programske objekte i opsege. U narednom poglavlju biće izloženo predstavljanje tipova i provere tipa. Struktura ovog poglavlja sastoji se od uvodnog dela, gde se podsećamo na ulogu tabele simbola u kompajleru, zatim ćemo razmotriti način organizacije struktura podataka kojima možemo predstaviti tabelu simbola, zatim ćemo videti kako predstaviti informaciju o programskim objektiva: to su konstante, promenljive, klase, metodi, parametri metoda, itd. Videćemo kako predstaviti informacije o opsezima imena i definisaćemo kako se tačno realizuju osnovne operacije nad tabelom simbola.

Rekli smo da tabela simbola u kompajleru služi za pamćenje informacija o svim deklarisanim objektima. Informacije zavisi od vrste objekata, znači sve vrste objekata imaju ime i tip, zatim konstante recimo imaju vrednost, promenljive i metodi imaju memorijsku adresu na kojoj se pamte, parametri mogu imati tip, broj parametara i tako dalje. Tabela simbola služi za nalaženje informacija o korišćenim objektima, što se svodi na asocijativni pristup, odnosno ime se mapira u ostale atribute objekta, kao što su tip, vrednost, adresa i tako dalje. Pošto postoje programski opsezi to nije jednostavno linearno mapiranje imena, ali o tome ćemo posebno govoriti u nastavku.

Najosnovniji interfejs tabele simbola prema drugim delovima kompajlera mora da ima metode za ubacivanje informacija o objektu u tabelu simbola, to je ovaj metod insert gde se zadaje ime objekta i ove tri tačke simbolizuju još neke atribute objeka, a tabela simbola vraća, u slučaju uspeha operacije insert, zapis o objektu, gde mogu dalje da se postavljaju i čitaju atributi.

```
class Table {
    static Obj insert(String name,…);
    static Obj find(String name,…);
}
```

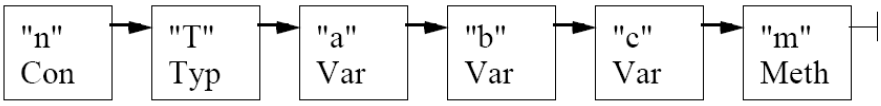
Naknadno, za već postojeći objekte pri njihovoj upotrebi, metod find na osnovu imena vraća postojeći zapis iz koga mogu da se čitaju atributi.

Interna organizacija tabele simbola

Tabela simbola je u svojoj osnovi jedna kolekcija, znači dinamička struktura podataka. Reč je o skupu objektnih, strukturnih zapisa i zapisa o opsezima. Objektni zapisi u kontekstu ove mikrojava tabele simbola biće instance klase Obj, čuvaju informaciju o deklarisanim imenima. Strukturni zapisi, instance klase Struct, čuvaju informaciju o tipovima u programu i na njih ukazuju objektni zapisi. O klasi Struct pričaćemo u posebnoj lekciji. Instance klase Scope čuvaju informacije o aktivnim opsezima imena u programu.

Neke moguće strukture podataka za organizovanje tabele simbola bi bile jednostruko u lančana neuredjena lista, binarni leksički uređeno stablo, ili heš tabela. Ako iskoristimo jednostruko ulančane liste, onda imamo evo primer za niz od nekoliko deklaracija: jedna je konstanta, druga je nekakva klasa, treće su promenljive celobrojnog tipa a, b i c i metod m. U tabeli simbola formirala bi se jedna jednostruko ulančana lista, gde bi svaki novi element bio dodavan na kraj liste.

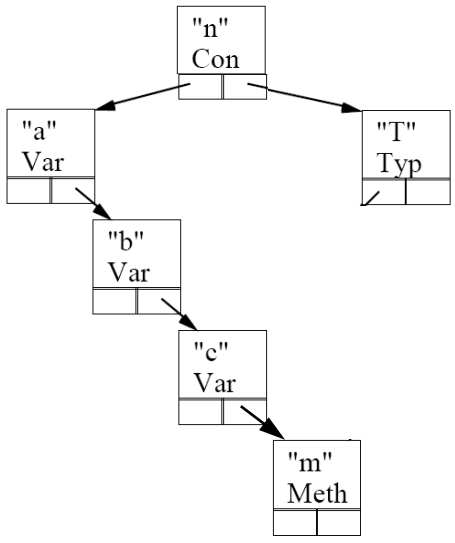
```
const int n=10;
class T {…}
int a, b, c;
void m(){…}
```



Vidimo da imamo niz objektnih zapisa koji imaju kao svoje komponente ime, vrstu objekta i još neke parametre koji nisu tu prikazani. Prednost korišćenja jednostruko ulančane liste kao strukture podataka je u jednostavnosti realizacije. Takođe, zapamćen je redosled deklarisanja imena, što je ponekad potrebno ako specifikacija jezika recimo zahteva alokaciju objekata u memoriji onim redom kojim su deklarisani. Nedostatak ovog načina organizacije je sporost pretrage, znamo da je red složenosti funkcije pretrage u jednostrukoj neuredjenoj listi N/2, gde je N dužina liste, znači ako ima mnogo imena to je relativno sporo. Zbog svoje jednostavnosti mi ćemo u nastavku ovog predavanja podrazumevati ovaj način organizacije tabele simbola.

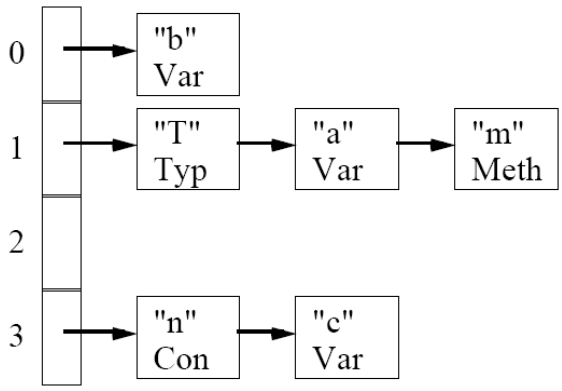
Inače u realnoj tabeli simbola objektno-orijentisanim mehanizmima je potpuno enkapsulirana organizacija tabele simbola, tako da korisniku nije značajno da li je to lista ili neka druga struktura. Iterfejs za korišćenje ni malo ne zavisi od toga. Međutim, kada se ovde reprezentuju primeri implementacije određenih metoda tabele simbola, najlakše je to prikazati, ako se usvoji ovaj organizacija.

Alternativna struktura podataka za realizaciju tabele simbola su binarna leksikografski uređena stabla. Evo imamo isti ovaj primer, sa nekoliko objekata, mogao bi da rezultuje u jednom ovakvom stablu. Znači kada god dolazi neka nova deklaracija, operacije ubacivanja novog objekta u tabelu simbola uvek kreće od korena. Uporedimo ime zapamćeno u korenu se imenom koje ubacujemo.



Ako ime koje ubacujemo leksikografski dolazi Iza imena u čvoru, idemo na desno potstablo, ako leksikografski dolazi pre imena čvora idemo na levo potstablo i tako dalje. Celu stvar ponavljamo sada se sledećim čvorom. Da bismo ubacili novo ime, ono uvek mora doći kao novi list stabla, znači moramo se spustiti skroz do nekog postojećeg lista i onda kao njegovog levog ili desnog sina ubaciti novi list. U opštem slučaju, pretraga je brža od linearne, ona je reda složenosti logaritma broja deklaracija. Podrazumeva se naravno da je raspored imena slučajan, međutim ako su imena baš sortirana na određeni način, onda će se ovo stablo degradirati u linearnu listu. Nešto je veća potrošnja prostora i vremena pri ažuriranju i originalni redosled deklaracija je izgubljen. Ako nam treba redosled, za to moramo uvesti poseban atribut u ovim zapisima koji čuva taj podatak.

Tabela simbola u vidu hash tabele realizovana je tako što definišemo hash funkciju kojom mapiramo ime u neki celobrojni Indeks. Uvodimo jedan hash vektor koji ima niz pokazivača, po jedan za svaki od tih heš indeksa, na kolizione liste, tako da određeno ime se mapira u određeni Indeks i onda se ulančava, recimo na kraj ili na početak lanca kolizije, kako se definiše. U ovom primeru, ulančavanje je na kraj ovog lanca kolizije. Pri kasnijoj pretrazi, opet izračunavamo heš Indeks određenog imena, pristupamo određenom slotu ovoga heš vektora i onda pretražujemo ovaj lanac kolizija da nađemo odgovarajući zapis. Ovo je praktično, u uslovima velike količine imena, možda najbrža varijanta. I ovde se takođe gubi redosled deklaracija. Ova komplikovanost je relativna, ako se koriste gotove biblioteke standardnih kolekcija, svejedno nam je da li ćemo uzeti heš li na stablo i tako dalje. Heš tabela je vrlo često korišćena struktura podataka za ove namene.



Kako pamtimo informacije o objektu u tabeli simbola? Svako deklarisanao ime se pamti u jednom objektom zapisu. Moramo znati o kojoj vrsti objekata se radi, tako da uvodimo niz simboličkih konstanti, znači po jednu za svaku različitu vrstu objekta: za konstante, programske promenljive, standardne tipove, korisnički definisane tipove, kao što su imena klasa, metoda i polja klasa. Neke informacije nezavisne su od vrste objekata, to jest sve vrste objekata imaju svoje ime, imaju informaciju o kojoj vrstu objekata je reč i tip u programu. Takođe, pošto smo usvojili organizaciju u vidu jednostruko ulančane liste, postoji još pokazivač na sledeći zapis. Neke informacije su specifične za određenu vrstu objekta: za konstante to je vrednost, za promenljive to je adresa u memoriji na kojoj će u vreme izvršavanja vrednost promenljive moći da se nađe, nivo deklarisanosti, da li su u pitanju lokalne promenljive metoda ili globalne promenljive programa.

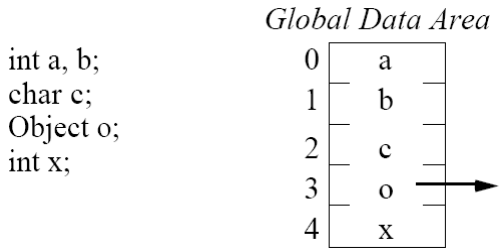
```
static final int
    Con = 0, // constants
    Var = 1, // variables, parameters
```

```
Typ = 2, // types
Meth = 3, // methods
Fld = 4; // fields
```

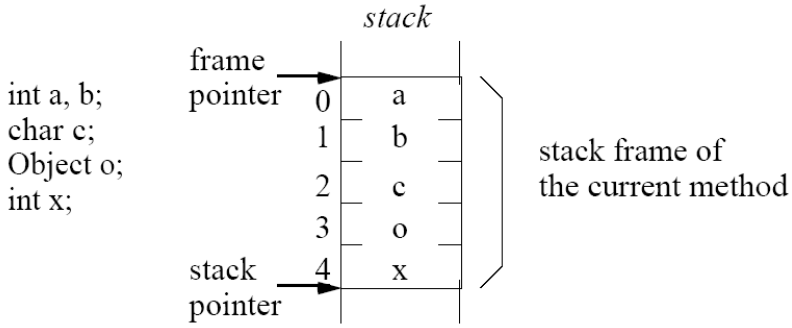
Za polja klase bitan je pomeraj u zapisu objekta klase u run-time-u, gde može da se određeno polje nađe; za metode pamtimo početnu adresu gde se nalazi mašinski kod provedenog metoda, pamtimo broj parametara metoda, formalne parametre pamtimo u listi objekata i takođe moramo imati listu lokalnih objekata metoda.

Adrese objekata u mikrojavi

Što se tiče adresa kao tipa informacija koji se pamti za neke vrste objekata, evo kratkog pregleda tipova adresa u mikrojavi. Što se tiče globalnih promenljivih u mikrojava virtuelnoj mašini, o kojoj ćemo naknadno više reći, sada ćemo reći samo toliko da se globalne promenljive čuvaju u posebnoj oblasti globalnih podataka i svaka mikrojava promenljiva zauzima po jednu memorijsku reč. Adrese promenljivih su relativne u odnosu na početak oblasti globalnih podataka i adrese se dodeljuju sukcesivno. Uprostili smo stvar u mikrojavi da promenljive nemaju različito memorijsku zauzeće, nego uvek fiksno po jedna memorijska reč, tako da ako imamo niz nekih globalnih objekata promenljivih znači njihove relativne adrese su stvari indeksi u nizu global data area, idu prosto sukcesivno jedna za drugom.



Što se tiče lokalnih promenljivih, u mikrojava virtuelnoj mašini, one se čuvaju na programskom steku, steku poziva u okviru metoda. Znači na steku se formira jedan zapis, koji se zove ili aktivacioni zapis, ili okvir određenog metoda. U trenutku kada se metod pozove na vrh tog steka poziva formira se taj okvir metoda.

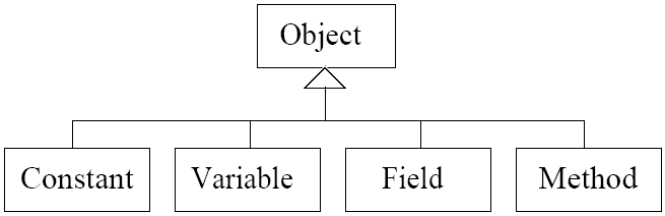


U runtime-u, da bismo znali gde se taj okvir metoda nalazi, njega ograničavaju s jedne strane stek pointer, to je vrh steka (ovako kako je pokazano stek raste naniže), a drugu granicu okvira metoda čini poseban registar koji se zove pokazivač okvira, odnosno frame pointer.

Da bi se pristupilo nekoj od lokalnih promenljivih, adresa se formira tako što se na trenutnu vrednost frame pointera doda poznati pomeraj. Inače ovaj pomeraj je konstanta koja je poznata kompajleru u vreme prevođenja programa. Vrednost frame pointera se u runtime-u formira, tako da kompajler ne zna njegovu vrednost, ali on zna ovaj pomeraj. Sada. ako se onaj isti niz deklaracija odnosi na lokalne promenljive, pošto svaka mikrojava promenljiva zauzima po jednu memorijsku reč i adrese promenljivih su relativne u odnosu na pokazivač okvira (frame pointer) dodeljuju se sukcesivno, znači redom čemo im dodeliti relativne adrese.

Predstavljanje mikrojava objekata u tabeli simbola

Pošto smo rekli da deo atributa objekta zavisi od vrste objekta, postavlja se pitanje koju ćemo strukturu podataka u Javi iskoristiti za beleženje objekta. Objektno-orijentisanom projektovanju bi odgovaralo da onaj zajednički deo atributa objekta, kao što su ime i vrsta zabeleži u nekoj osnovnoj klasi Obj, a zatim tu klasu proširujemo za svaku posebnu vrstu objekta, tako da imamo posebno izvedene klase za konstante, za promenio, polja i metode.



U implementaciji koju ovde dalje diskutujemo, nije primenjeno to rešenje. Rešenje koje je primenjeno je da formiramo samo jednu klasu Obj i da sve vrste informacije o svakoj vrsti objekata pamtimo u toj jednoj klasi. Ovo nije preporučeno sa stanovišta objektno-orijentisanog projektovanja, ali s druge strane u praksi možete naići na takva rešenja. Ovde je bio motiv da se uprosti kod koji kasnije diskutujemo za iimplementaciju pojedinih metoda.

```
class Obj {
    static final int Con = 0, Var = 1, Typ = 2, Fld = 3, Meth = 4;
    int kind; // Con, Var, Typ, Fld, Meth
    String name;
    Struct typ;
    Obj next;
    int adr; // Con: value; Var, Fld, Meth: address
    int level; // Var: declaration level; Meth: no.of. parameters
    Obj locals; // Meth: local objects
}
```

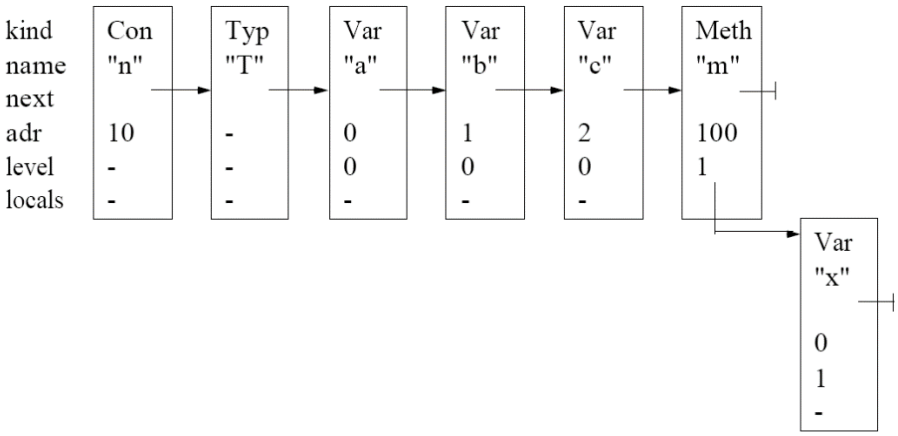
Znači imamo klasu Obj, imamo definisane konstante koje predstavljaju vrstu objekta. Vrsta objekta pamti se u polju kind i ona dobija vrednost neke od ovih konstanti. Zatim, za objekat se pamti ime i pamti se tip, programski tip objekta, to je objekat klase Struct u kome ćemo naknadno pričati. Ova tri podatka, kind, name i type su zajednički za sve vrste objekata. Još jedno zajedničko polje je next, to je pokazivač, to jest, referenca na sledeći zapis u onoj jednostrukoj ulančanoj listi koja odgovara određenom tekućem opsegu. Sada slede polja koja neke vrste objekata koriste. Polje other, ako je objekat vrste konstanta, predstavlja vrednost konstante, a ako je objekat vrste promenljiva, polje klase, ili metod, predstavlja adresu onako kako smo maločas diskutovati različite vrste adresa. Sledeće polje level za promenljive pamti nivo deklarisanosti, da li su lokalne

ili globalne, a za metode se u tom polju pamti broj njihovih formalnih parametara. I konačno imamo pokazivač na listu objekata koje koristi metod za svoje formalne parametre i lokalne nepromenljive, oni se zajednički ulančavaju u listu, pa ovo polje level u stvari kaže koliko članova liste se odnosi na formalne parametre, a ostatak su lokalne promenljive.

Sada kada znamo strukturu objekta, kako izgleda deo strukture koji se pamti u tabeli simbola za ovaj niz deklaracija?

```
const int n = 10;
class T {…}
int a, b, c;
void m (int x) {…}
```

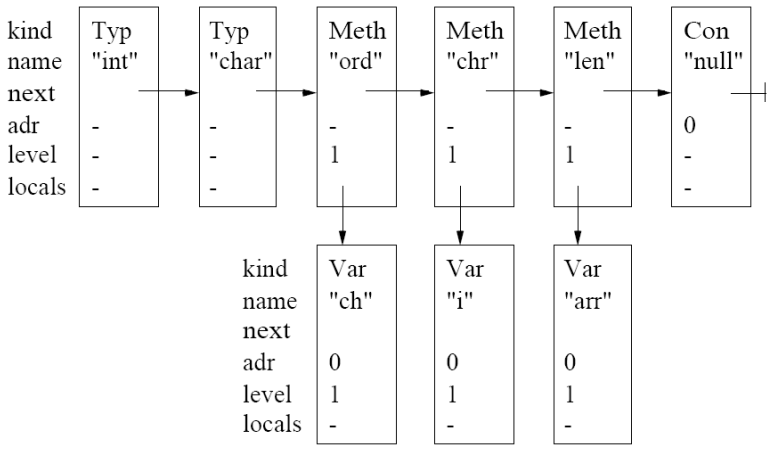
Znači, imaćemo za svako ime koje se ovde pominje, po jednu instancu klase Obj i popunjena su odgovarajuća polja prema vrsti objekta. Ovde je navedena vrsta objekta, ovde je navedeno ime, pokazivač na sledeći zapis, naravno poslednji objekat ima null, kraj liste, adresa znači za konstante predstavlja vrednost, za promenljive to je adresa. Vidimo po nivou da su ove tri promenjive globalne promenljive, imaju level nula, znači ovo je u oblasti globalnih podataka relativna adresa. Za metode, adresa gde je preveden kod metoda. Kodovi metoda čuvaju se u posebnom delu memorije mikrojava virtuelne mašine, oblasti programskog koda. Polje level metoda označava da ovaj metod m ima jedan formalni parametar i znači ova locals lista ukazuje na taj jedan objekat koji reprezentuje formalni parametar metoda m, to je znači x, on je isto vrste promenjiva i ima lokalnu adresu nula u okviru metoda i nivo je jedan, što znači da nije globalna promenljiva nego lokalna.



Preddeklarisana imena

Na početku prevođenja, tabela simbola nije prazna, nego sadrži već neka takozvana predeklarisana imena. Ta imena važe u celom programu. Ta imena odnose se na standardne tipove, o tipovima ćemo pričati u posebnoj lekciji. Zatim, odnose se na standardne metode: method ord() služi za konverziju znakovnog tipa u celobrojni, znači daje kod odredjenog znaka, metod chr() obrnuto, za određeni kod daje njemu odgovarajući znak. Metod len() da e dužinu nekog niza u mikrojava programu, znači broj njegovih elemenata. Konačno, postoji konstanta null, slično kao u Javi, znači kad uvedete promenljivu tipa neke klase, dok ne instancirate objekat klase, vrednost te promenljive je null.

Kako izgledaju strukture podataka u tabeli simbola za ta predeklarisana imena? Evo vidimo, znači niz objektnih zapisa, za svaku od ovih pomenutih imena. Mogu samo reći da ovo još nije kompletna slika, ovde nedostaju informacije o tipovima i takođe nedostaju informacije o opsezima. Opseg predeklarisanih imena naziva se Universe u mikrojavi i on je stalno aktivan kao krajnji spoljni opseg, nad njim nema nikakvog drugog opsega. Ovde vidimo još jednu pribelešku, a to je da za ova predefinisana imena i teorijski može i na drugi način da se reši njihovo korišćenje. a to je da se ona ne pamte u tabeli simbola kao predefinisana imena, nego da recimo na nivou gramatike uvedemo posebne ključne reči i posebne smene koje obrađuju sve ove objekte sa specijalnim značenjima. To bi dosta komplikovalo stvari, uvelo niz nekih specijalnih slučajeva pri pretraživanju imena i komplikovalo gramatiku. Tako da je ovo u suštini mnogo elegantnije rešenje, da se ova imena stave u tabelu simbola.



Zanimljivo je da u mikrojavi ključne reči, to su ove reči sa specijalnim predefinisanim značenjem, na primer kao što je int, nisu ujedno i rezervisane reči. Često se u literaturi o programskim jezicima izjednačava značenje ključne reči i rezervisane reči, ali postoji suptilna razlika: ključna reč je neka reč sa nekim predefinisanim specijalnim značenjem, kao što je to ovde recimo int ili konstanta null, a rezervisana reč je reč čije značenje ne sme da se redefiniše, znači ima neko određeno značenje i ono je konstantno. Međutim, zanimljivo je da u mikrojavi ključne reči nisu ujedno i rezervisane, odnosno nije sprečeno da programer uvede na primer svoju promenljivu koju bi nazvao baš int. Kakve bi to onda imalo posledice? Pa u tom opsegu gde on uvede promenljivu int, više ne bi mogao da deklariše nijednu celobrojnu promenljivu jer bi promenio značenje ove reči int. Znači to može da bude korisno ali i da stvori zabunu. Zato neki jezici to ograničavaju.

Opsezi važenja imena

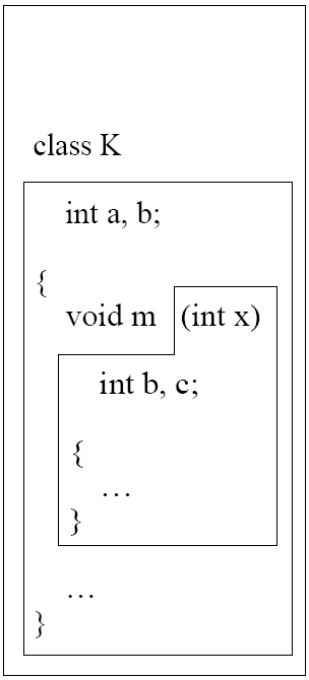
Sada ćemo videti kako se u tabeli simbola pamte i obrađuju informacije o opsezima imena u programu. U blok strukturiranim jezicima kao što smo videli, a u to spada i mikrojava, postoji hijerarhija opsega imena. Na najvišem nivou hijerarhije je opseg predeklarisanih imena, takozvani Universe. Zatim dolaze globalna imena, koja naravno važe u celom programu, znači ako deklarišete na nivou programa neku promenljivu, to je globalna promenjiva i njeno ime će biti vidljivo u celom programu. Zatim, ako deklarišete neku klasu, u stvari otvarate novi opseg, tako da sva polja klase i metodi te klase pripadaju opsegu te klase. I konačno, kada se deklariše neki metod, parametri i lokalna imena imaju opseg metoda. Usvojena je realizacija sa lokalnim pamćenjem tih kolekcija objekata, ono svaki opseg ima svoju posebnu kolekciju, listu Obj zapisa.

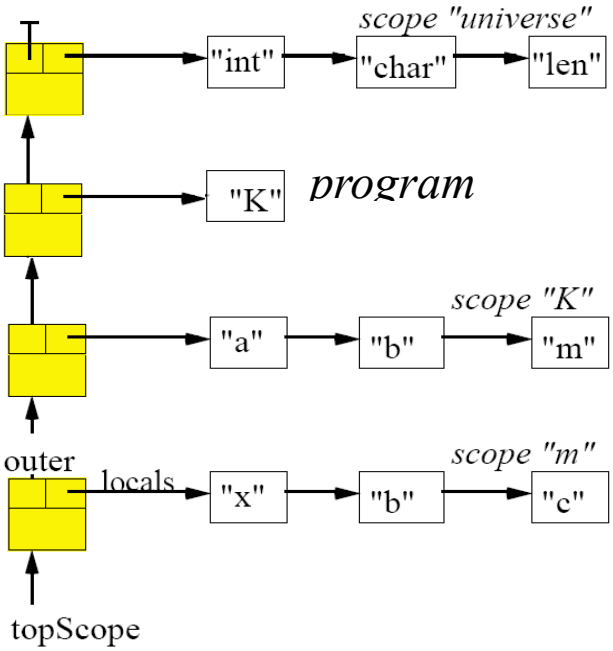
Za reprezentaciju nekog opsega uvodi se zapis Scope. Njegove komponente su; pokazivač na spoljni opseg, tako da, preko tih pokazivača se ovi objekti u toku rada kompajlera ulančavaju u jednostruku ulančanu listu, koja kreće od tekućeg opsega pa ide ka spoljnom opsegu, znači na kraju lista je uvek Universe. Zatim imamo pokazivač na listu objekata koji pripadaju ovom opsegu, znači koji su deklarisani u posmatranom opsegu i imamo broj objekata koji su deklarisani u opsegu. Vrednost ovog polja se zgodno koristi za alokaciju adrese jer videli smo da, za sve vrste objekata koji zahtevaju adresu, pošto svaka vrsta objekta uvek zauzima po jednu memorijsku reč, da prosto samo sekvencijalno dodeljujemo te adrese i za to uvek možemo iskoristiti znači ovu nVars promenljivu. Znači kada ubacujemo novi objekat, mi ga ulančanamo u ovu listu, uvećavavamo nVars, pridružimo objektu tu ažuriranu vrednost nVars.

```
class Scope {
    Scope outer; // to outer scope
    Obj locals; // to the objects of this scope
    int nVars; // number of variables in this scope (for addr. allocation)
}
Scope topScope; // this is an attribute of class Tab
```

Pored ovih Scope objekata, imamo takođe jednu promenljivu TopScope. Pošto sam rekao da se u toku rada kompajlera svi ovi opsezi ulančavaju u jednu listu koja pokazuje aktivne opsege, na početak te liste ukazuje ovaj TopScope pokazivač, koji je atribut cele tabele simbola i dostupan i ostatku kompajlera kroz interfejs tabele simbola.

Opsezi se, kao što sam rekao, ulančavaju u kolekciju tipa steka. Evo kako to izgleda na primeru, Ovaj primer ima nekakav program, taj deo koda nije prikazan, ovde bi u mikrojavi pisalo program ime programa i onda bi tek išla deklaracija klase.





Ako je tekuća linija posmatranja, znači dokle je kompajler došao sa obradom, unutar metoda m, u tom trenutku izgled tabele simbola biće kao na slici desno. Ovde su prikazani Scope i objektni čvorovi, nedostaju samo Struct čvorovi o tipovima, pošto o njima još nismo pričali.

U trenutku kada počinje rad kompajlera, formiraće se samo ovaj jedan scope, topScope bi ukazivao samo na ovaj jedan član i to je scope Universe, odnosno predefinisana imena. U trenutku kad se naiđe na ovde nedostajuću deklaraciju program ime programa, otvara se opseg programa i sve što je globalno, recimo ovde je ime ove klase K, ide u opseg programa. Znači u trenutku kada se neke opseg otvara, na vrh ovog scope steka dodaje se novi zapis i onda topScope ukazuje, umesto na ovaj, sad ukazuje na ovaj, a ovaj ukazuje na Universe.

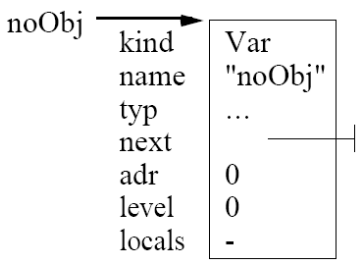
U trenutku kada naiđemo na deklaraciju klase K, još ovo ime K se ubacuje u programski opseg, a neposredno iza imena se otvara novi opseg klase K, znači formira se novi scope zapis i tako ulančanava u listu aktivnih opsega da je na početku liste. U opsegu klase K su polja klase K i ime metoda m. U trenutku kada prođe ime metoda M. otvara se lokalni opseg metoda M. Formalni parametar metoda M, kao i lokalne promenljive b i c ulančvaju se u lokalni opseg metoda M.

Pretraga u tabeli simbola

Razmotrićemo kako se, sa usvojenom organizacija tabele simbola, obavljaju osnovne operacije u tabeli simbola. Pretraga imena u tabeli simbola prilikom svakog korišćenja, realizuje se metodom find, znači to je deo interfejsa tabele simbola. Prema imenu, pretraga ide kroz listu aktivnih opsega. Kreće se od prvog elementa lista na koji ukazuje topScope. Ako ne nađemo traženo ime u toj listi objekata, preko atributa outer u zapisu Scope dolazimo do okružujućeg opsega i onda pretražujemo u njegovoj listi i tako dalje.

```
static Obj find (String name) {
    for (Scope s = topScope; s != null; s = s.outer)
        for (Obj p = s.locals; p != null; p = p.next)
            if (p.name.equals(name)) return p;
    error(name + " not found");
    return noObj;
}
```

Ovde je prikazan kod implementacija find metoda. Krene se od topScope-a. Sve dok postoji aktivan opseg, ako u tekućoj iteraciji ne nađemo ime prelazi se na sledeći okružujući opseg. Unutar jednog opsega ide se kroz locals listu, do god postoje elementi i poredi se zadato ime sa imenom objekta zapamtićenog u listi. Ako dođe do poklapanja, vraća se pronađeni objekat. Šta se dešava u slučaju da prođemo kroz sve opsege i ne nađemo traženo ime? Treba prijaviti grešku da ime nije nađeno, znači to je greška u programu. Vraća se posebna vrednost. Znači pošto treba vratiti zapis objekat, bilo bi očekivano da, ako ne nađemo ništa, vratimo null konstantu. Međutim ne vraćamo null, nego vraćamo posebnu vrstu zapisa, koja se zove noObj.



noObj je takođe jedna vrsta predeklarisanog objekta, koji nije ulančan u ove Scope-ove postojeće, jer ga nećemo tražiti po imenu, nego ćemo ga vraćati samo u slučaju greške. Kao što vidite, on je određene vrste promenljive i ima neko svoje specijalno ime.

Zašto vratiti ovakav poseban objekat greške a ne prosto null vrednost? Ovakav koncept obrade greške poznat je iz IEEE standarda brojeva u pokretnom zarezu. Aritmetičke operacije nad brojevima u pokretnom zarezu nekad mogu da rezultuju u grešci, na primer deljenje sa nulom i da postoje dva načina obrade greške, jedan je da se odmah signalizuje greška, a drugi je onaj takozvani tihi režim, kada se koristi vrednost koja se zove NaN (not a number). U tihom režimu, neka operacija, ako dođe do greške, neće signalizovati grešku, ali će kao svoj rezultat vratiti tu specijalnu vrednost NaN. Sada ako niz operacija izvodimo, znači neki složeni izraz i ne ispitujemo posle svake operacije da li je došlo do greške, nego tek na kraju kada izračunavamo vrednost celog izraza. Ako sledeća operacija u nizu na ulazu dobije NaN, ona će prosto samo na izlaz proslediti opet NaN, znači neće se raditi nikakva operacija i na taj način je omogućeno da izvedemo niz nekih operacija i tek na kraju ispitamo eksplicitno da li je došlo do greške, znači da li je rezultat neki validan ili NaN. I noObj prati sličnu logiku, mi možemo nekoliko stvari da uradimo nad tabelom simbola i tek na kraju ispitamo šta je objekat. Kao što vidimo, sami metodi tabele simbola imaju ugrađenu dijagnostiku, znači ako se pojavi neka greška, iniciraće prijavu greške krajnjem korisniku.

Ubacivanje imena u tabelu simbola

Sledeća operacija nad tabelom simbola je insert, to jest, ubacivanje nekog imena u tabelu simbola. Metod insert ima tri parametra, to su upravo oni zajednički atributi bez obzira na vrstu objekta, znači ime objekta, njegova vrsta i tip objekta, instanca klase Struct o kojoj ćemo naknadno pričati. Šta treba da uradi insert?

- Treba da proveri da li već postoji objekat istog imena u aktivnom opsegu, a to je onaj opseg na koji ukazuje topScope. Ako već postoji objekat istog imena, onda je reč ogrešci, ne može dvaput da se deklariše u jednom opsegu isto ime, bez obzira da li je ta druga deklacija iste vrste ili nije.
- Ako nije došlo do greške, kreiraće se nova instanca klase Obj, biće popunjena ova polja koja su zadata, biće mu dodeljena prva slobodna adresa, za šta se koristi nVars iz Scope zapisa.
- Automatski se dodeljuje i nivo deklaracije za promenljive, da li je globalna ili lokalna, pošto to tabela može preko tekućeg Scope-a da proveri i na kraju
- sam objekat se dodeljuje na kraj one liste objekata koja se pamti u topScope.lokals, znači tu je ta lista objekata i na njen kraj se dodaje promenljiva.

Postavlja se pitanje, gde se u ostatku kompajlera pozivaju ovi metodi find, insert i tako dalje? Naravno, oni se pozivaju kao posledica sintaksno-upravljanog prevođenja, odnosno semantičke analize programa. Ovde kako je prikazano, podrazumeva se semantička obrada putem akcionih simbola gramatike. Druga varijanta semantičke obrade je da parser izgradi sintaksno stablo programa, a onda da se posebnim mehanizmima, to onaj visitor šablon u objektno-orijentisanom programiranju, prolazi kroz stablo i na određenim mestima inicira određena obrada.

VarDecl = Type$_{typ}$ ident$_{name}$ (: Tab.insert(Obj.Var, name, typ) :)

{ "," ident$_{name}$ (: Tab.insert(Obj.Var, name, typ) :) }.

Znači ovo je prva varijanta, ako direktno ugradjujemo akcije u gramatičke smene, onda bi parser već pri parsiranju programa sprovodio ove akcije. Vaš projekat, domaći zadatak, nekad aje koristio ovaj način semantičke obrade, a sada se koristi onaj drugi, putem izgradnje i obilaska apstraktnog sintaksnog stabla. Može vrlo lako da se jedan način prepravi u drugi način.

Sada da vidimo unutar smene gde može da se inicira ubacivanje objekata u tabelu simbola. U deklaraciji promenljive, čim prođe ime promenljive, treba pozvati ubacivanje, vrsta objekta je promenljiva, ime se dobija kao atribut iz leksičke analize, kao atribut terminala IDENT, a tip se dobija kao atribut ovog gramatičkog simbola type. Za type postoje posebne smene koje će da razreše tip i koje će vratiti jedan strukturni zapis preko ovoga atributa, koji ćemo mi ovde iskoristiti u pozivu. Ovde je prikazana smena, znači konceptualan je primer. Smena je prikazana u proširenoj BNF notaciji, znači deklaracija promenljive može da ima tip int pa onda ide niz imena x, y, z, ovaj deo opisuje ta dodatna imena i posle svakog imena treba pozvati na isti način Tab.insert().

Obrada opsega

Što se tiče obrade opsega, tabela simbola treba da imam metode za otvaranje opsega i za zatvaranje opsega.

Za otvaranje opsega zadužen je metod openScope(). Treba da se napravi nova instanca Scope objekta i da se ulanča u listu aktivnih opsega, na početak te liste. Parser treba da poziva ovaj metod kada se naiđe na početak opsega, odnosno, to je početak programa ili čim prođe ime klase ili čim prođe ime metoda.

```
static void openScope() {
    Scope s = new Scope();
    s.outer = topScope; topScope = s;
}
```

Za zatvaranje opsega koristi se metod closeScope() koji izlančava tekući opseg iz lista aktivnih opsega, tako što prosto topScope namesti da pokazuje na sledeći u listi. Ovaj metod poziva se kada parser naiđe na kraj opsega, a to su situacije kraja programa, završne vitičaste zagrade od klase ili od metoda.

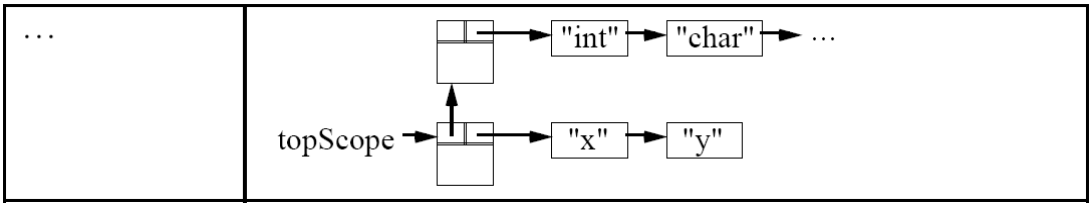
```
static void closeScope() {
    topScope = topScope.outer;
}
```

Na sledećoj strani imamo primer sa gramatičkim smenama. Imamo smenu za deklaraciju metoda, gde se ta smena sastoji od povratnog tipa metoda koji ima ovaj atribut strukturni čvor type, zatim ime metoda, to je token sa atributom stringom imena i znači čim prođe ime metoda potrebno je ubaciti to ime metoda u trenutno aktivni opseg, a posle toga otvoriti lokalni opseg metoda. Dosta je važan ovaj redosled poziva ove dve metode, da se ne uradi obrnuto. Sada ovde ide deo smene koji nije prikazan, odnosno prikazan je sa tri tačke, koji se odnosi na opis samog tela metoda, lokalne deklaracije i izvršni deo znači niz naredbi. Konačno smena se završava vitičastom

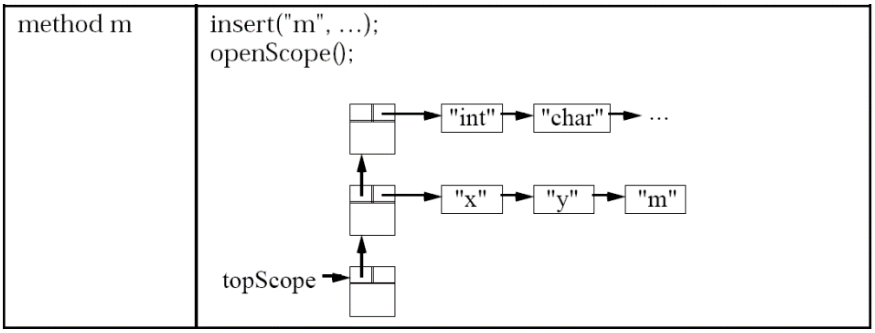
zagradom i na samom kraju te smene potrebno je zatvoriti lokalni opseg metoda, ali pre zatvaranja opsega, da nam ne bi nestala lista lokalnih objekata metoda koja se sastoji od formalnih parametara i lokalnih promenljivh, potrebno je zapamtiti je. Meth je onaj objekti zapis za tekući metod, koji je formiran pri pozivu insert. Znači to treba da zapamtimo u nekoj pomoćnoj promenljivoj, pa u njenom polju local treba zapamtiti tu listu iz tekućeg opsega pre nego što se on zatvori.

```
MethodDecl
= Type_typ
   ident_name (: meth=Tab.insert(Obj.Meth, name, typ);
                 Tab.openScope() :)
…
   "}"          (: meth.locals = Tab.topScope.locals;
                 Tab.closeScope(); :).
```

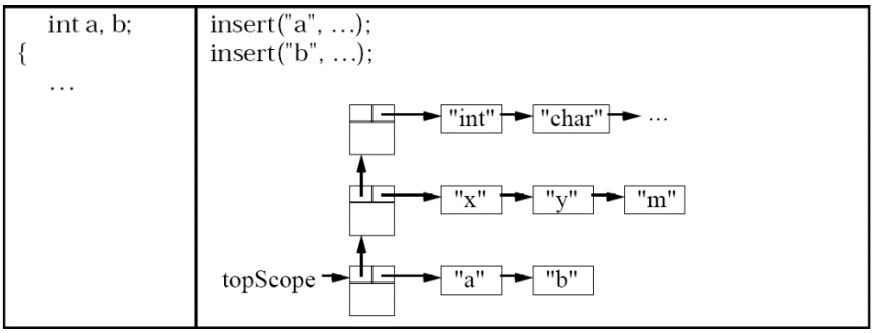
Prokažimo u nekoliko koraka sekvencu aktivnosti prilikom semantičke obrade definicije nekog mikrojava metoda m. Ako je ovo neka početna situacija,



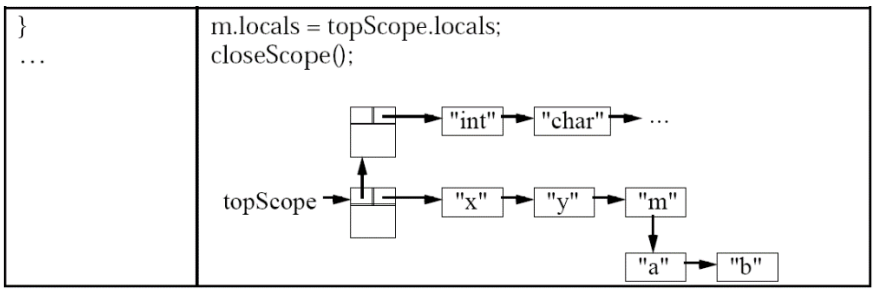
kada dolazi telo metoda m, onda se u tekući opseg (u onom prethodnom to je bio globalni opseg odgovara glavnom programu) ubaci ime metoda M. Posle toga formirano na vrhu steka opsega novi prazan opseg.



Sada kada dolaze lokalne promenljive, ubacujemo u taj opseg ovo a i b.



Kada na kraju tela metoda, dolazi do zatvaranja opsega, mi ćemo izlančati taj lokalni opseg iz ove liste, ali ćemo a i b zapamtiti kao atribute ovog objekta M, tako da ta informacija neće biti izgubljena.



Sa svim ovim dosadašnjim saznanjima, konačno možemo prikazati kompletan interfejs tabele simbola. Vidimo ova četiri osnovna metoda insert, find, openScope i closeScope sa svojim parametrima, vidimo da postoje direktni pokazivači na neke predefinisane objekte i tipove, vidimo ovaj topScope listu aktivnih opsega i takođe ostatak kompajlera može pročitati i o ugneždenosti tekućeg opsega, znači da li globalni opseg ili lokalni. U implementaciji koja je dostupna na sajtu kursa u vidu .jar biblioteke ima još par stvari koje ovde nisu pomenute, kao što su iteratori kroz objekte nekog opsega i slično. To su pomoćne stvari da možete tabelu simbola efikasno da upotrebite, a ne utiču na ovde izloženu konceptualnu priču.

```
class Tab {
    static Scope topScope; // current top scope
    static int level; // nesting level of current scope
    static Struct intTyp; // predeclared types
    static Struct charTyp;
    static Struct nullTyp;
    static Struct noTyp;
    static Obj chrObj; // predeclared objects
    static Obj ordObj;
    static Obj lenObj;
    static Obj noObj;
    static Obj insert (int kind, String name, Struct typ);
    static Obj find (String name);
    static void openScope();
    static void closeScope();
}
```

Tipovi u mikrojava tabeli simbola

U nastavku lekcije o konstrukciji tabele simbola za mikrojavu, bavićemo se predstavljanjem tipova u programu i proverama tipova. Svaki programski konstrukt, bilo da je deklaracija, izraz ili iskaz, ima svoj tip. Vrste tipova u mikrojavi su osnovni tipovi, znači oni koji ne mogu da se dalje razlože na prostije komponente -- celobrojni i znakovni. U nekim varijantama, kada se zadaje domaći zadatak, može biti i logički. Izvedeni tipovi u mikrojavi su nizovi i klase.

Tipovi imaju strukturu, odnosno imaju određene svoje osobine, imaju memorijsku zauzeće, znači koliko promenljiva određenog tipa zahteva memorije za pamćenje vrednosti, a za izvedene tipove tu su i druge osobine kao što su, kod nizova broj i tipovi elemenata niza, kod klase definicije

članova klase, polja i metoda i tako dalje. Tipove možemo predstavljati takozvanim izrazima tipa koji se obično definišu počeši od osnovnih tipova, da oni predstavljaju izraz tipa. Zatim, ime tipa, to je kod korisnički uvedenih imena, na primer klasa predstavlja izraz tipa, a zatim se izvedeni tipovi dobijaju primenom takozvanih konstruktora izvedenog tipa na neki izraz tipa.

Pogodan metod za predstavljanje izraza tipa su grafovi. U slučaju mikrojave, grafovi tipova sadrže Struct i Obj zapise. Obj zapise smo već definisali, a Struct zapis, odnosno klasu Struct mikrojava tabele simbola sada gledamo njenu strukturu. Prvo imamo niz simboličkih konstanti, koje se odnose na vrstu tipova. Za celobrojne, znakovne, nizove i klase i imamo naravno ako nije nikakav tip. Polje kind upravo pamti jednu od ovih konstanti, tako da onda znamo na koju vrstu tipa se odnosi odgovarajući Struct zapis. Ostala polja u ovom strukturom zapisu imaju interpretaciju zavisno od ovoga od vrste tipa. Polje tip elementa elemTyp je referenca na drugi strukturni zapis i kod nizova se koristi da upiše tip njegovih elemenata, a kod klasa se koristi kao pokazivač na natklasu, ako je klasa izvedena iz neke druge klase. Zatim imamo celobrojno polje N koje kod klase predstavlja broj članova klase i polja i metoda i zatim imamo pokazivač na listu objektnih zapisa koji predstavljaju članove klase. U listi prvo ide niz objektnih čvorova koji opisuju polja, a zatim i objektni čvorovi koji opisuju metode klase. Ovo još nije kompletna definicija ove klase Struct u smislu da klasa Struct poseduje i neke metode za proveru ekvivalentnosti tipova o čemu će naknadno biti reči.

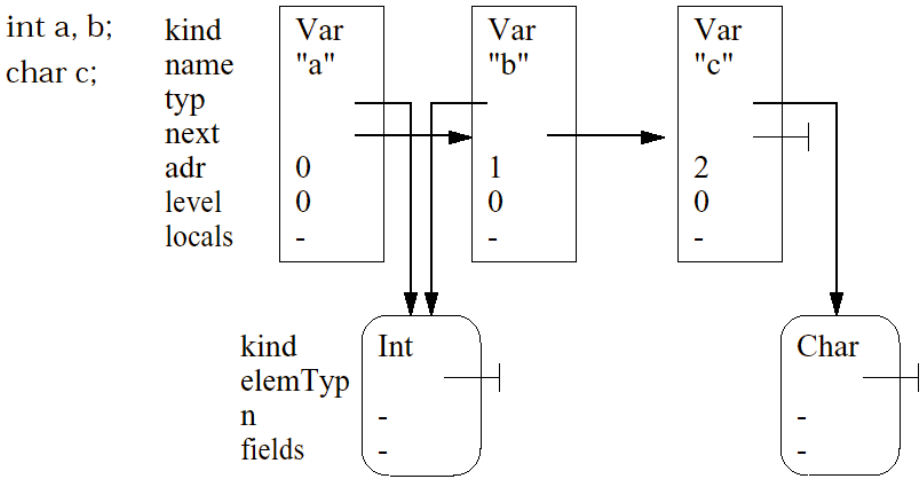
```
class Struct {
    static final int // kinds of types
        None = 0, Int = 1, Char = 2,
        Arr = 3, Class = 4;

    int kind; // None, Int, Char, Arr, Class

    Struct elemTyp; // Arr:elem.type,
                    // Class:superclass

    int n; // Class: number of fields
    Obj fields; // Class: list of fields
}
```

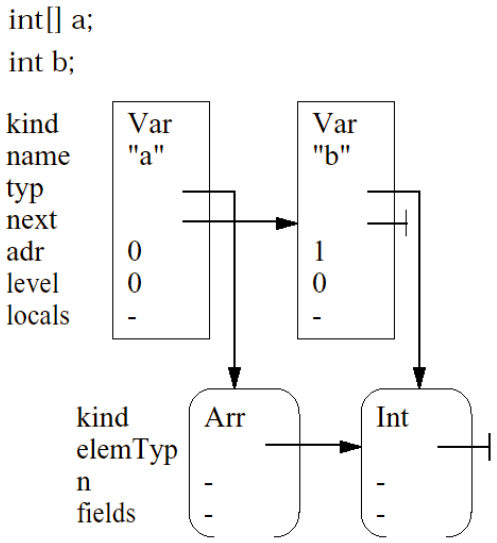
Sada da vidimo kako se u mikrojava tabeli simbola reprezentuju različitii tipovi. Krenućemo od najjednostavnijeg, odnosno ovih osnovnih tipova, celobrojnog i znakovnog. Na primer, da imamo ovaj niz globalnih deklaracija, znači dve celobrojne promenljive a i b, i znakovnu promenljivu c.



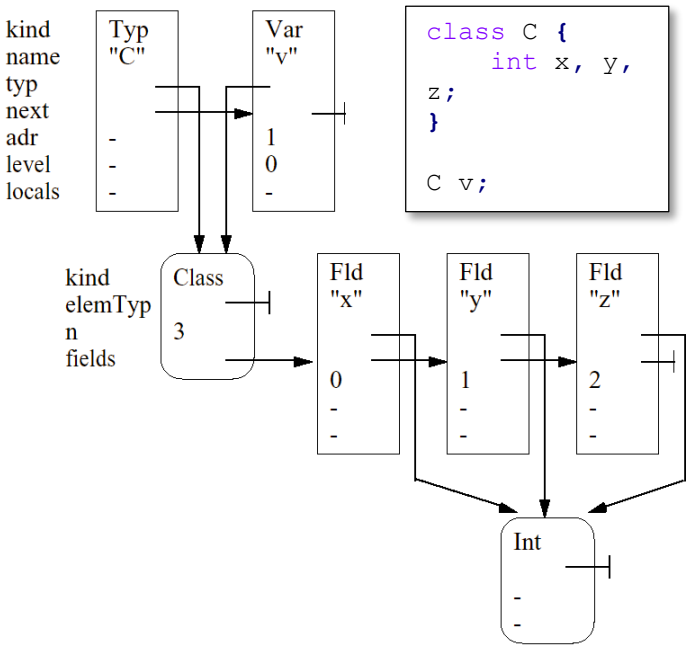
U tabeli simbola Imali bismo listu sastavljenu od objektnih čvorova. Svaki objektni čvor opisuje jednu od ovih promenljivih, znači promenljiva a ime promenljive, zatim ovo polje type loje dosad nismo prikazivali, koristi se da opiše tip promenljive i ono ukazuje na strukturni zapis.

Po konvenciji, objektne zapise uvak crtamo kao pravougaonike, a strukturne zapise crtamo kao pravougaonike zaobljenih uglova tako da odmah razlikujemo koja je koja vrsta čvora. Znači strukturni čvor vrste int opisuje osnovni celobrojni tip. Ova druga polja se ne koriste, elemTyp je null, a fields se ne koristi. Važno je napomenuti da u celoj tabeli simbola postoji samo jedna instanca ovoga strukturnog zapisa i sve promenljive, u konkretnom primeru a i b, koje su celobrojnog tipa ukazuju na ovaj isti čvor. To je videćemo posle, važno kod provere tipa, da li su dve promenljive istog odnosno nekog ekvivalentnog tipa. Promenljiva c isto ima svojih objektni zapis i njen tip ukazuje na strukturni zapis koji je vrste znak. Isto se ova druga polja ne koriste, isto sve promenljive u programu koje su znakovnog tipa ukazivaće na ovu jednu jedinstvenu instancu ovog strukturnog zapisa. Naravno, druga polja u ovim objektnim zapisima za promenljive su njihove adrese u globalnoj oblasti memorije koje su im dodeljene i vidimo da su u pitanju (level jednako nula) globalne promenljive.

Kako se predstavljaju nizovi u mikrojava tabeli simbola? Ovde imamo deklaraciju jednog celobrojnog niza a. Da bismo ga koristili u programu, potrebno je isto kao i u javi, upotrebiti operator new, da bi se dodelila ovoj promenljivoj vrednost, ali to je deo izvršnog koda, znači nije deklaracija. Promenljiva a je referenca na niz i imamo celobrojnu promenljivu b. U mikrojava tabeli sibola imamo listu sastavljenu od dva objektna čvora za reprezentovanje promenljivih a i b. Tip promenljive a ukazuje na strukturni zapis vrste niz, a elemTyp polje tog strukturnog zapisa, ukazuje na onaj zapis koji se odnosi na celobrojni osnovni tip.



Klase imaju složeniji zapis u tabeli simbola, pa ćemo kroz nekoliko primera, od jednostavnijih ka složenijim, da vidimo sve osobine ovoga zapisa. Prvi primer je najjednostavniji mogući, klasa C koja ima tri polja x, y, i z. Ona nema metode niti se nasleđuje od neke druge klase. Takođe imamo promenljivu v koja je tipa C. Naravno, za korišćenje promenljive v treba operatorom new joj prvo dodeliti instancu klase C, ali to je deo izvršnog koda, tako da nas ne zanima u ovom primeru.



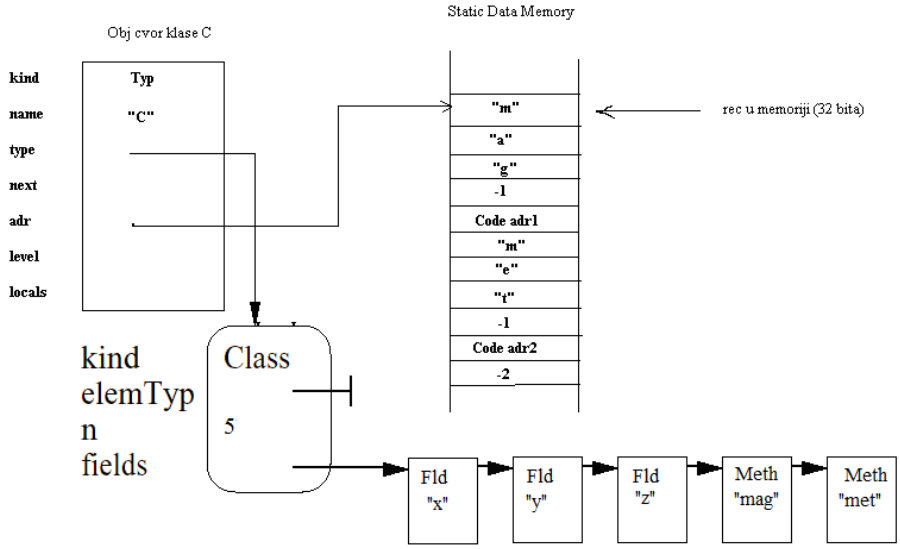
Kada parser naiđe na deklaraciju klase, on stvara odgovarajući objektni zapis u tabeli simbola. Taj objektni zapis pamtiće ime klase. U tom čvoru polje tipa pokazuje na novi zapis strukturnog tipa čije polje fields treba da pokazuje na polja klase čija se deklaracije obrađuje. Evo kako za ovaj primer izgleda grafovska struktura u tabeli simbola. Objektni čvor koji pamti ime klase i njegovo polje tipa pokazuje na strukturni čvor koji opisuje strukturu ove klase. Znači vrste je Class, elemTyp je null, pošto ova klasa ne proširuje nijednu drugu klasu. Klasa ima tri polja i pokazivač na listu objektnih čvorova za njih je fields. Lista polja sadrži po jedan objektni čvor za svako polje klase. Vrsta tog objektnog čvora je field (polje klase). U pitanju su celobrojna polja i njima su u polju adr dodeljeni pomeraji u objektnom zapisu u memoriji. Kada se instancira objekat ove klase, na pomeraju 0 će biti polje x, na pomeraju 1 polje y, a na pomeraju 2 polje z.

Dinamički metodi klase

Sledeći primer vezan za klase ilustruje opisivanje klasa koje imaju i uobičajene dinamičke metode. Svaka klasa osim glavnog programa koji isto možemo tretirati kao jednu klasu što se tiče tabele simbola, znači svaka klasa ima virtuelnu tabelu metoda, takozvanu v-tabelu. v-tabela se čuva u statičkoj oblasti memorije. Adresa prve reči v-tabele u statičkoj oblasti memorije se pamti u objektnom čvoru klase, u polju adr. Razmotrićemo primer jedne konkretne klase C koje ima tri polja i dva dinamička metoda, mag i met.

```
class C {
    int x,y, z;
    int mag() {
        return 3;
    }
    void met(int q) {
        q=q+1;
    }
}
```

Ovoj klasi odgovara ovaj graf u tabeli simbola.



Ime klase se pamti u objektnom čvoru. Polje type ukazuje na strukturni zapis koji opisuje strukturu klase, a polje adr ukazuje na početnu adresu u statičkoj oblasti memorije virtuelne tabele ove klase. Treba napomenuti da je virtuelna tabela klase jedna struktura podataka čiji sadržaj kompajler definiše, ali koja svoje korišćenje u stvari i egzistenciju ima tek u vreme izvršavanja programa. Ova statička memorijska oblast je u stvari jedna oblast memorije u mikrojava virtuelnoj mašini. Ova tabela sa svojim sadržajem ne koristi se tokom prevođenja u kompajleru, nego tokom izvršavanja programa. Kompajler mora da obezbedi da se na početku izvršavanja programa deo statičke memorije napuni ovim podacima i kasnije, tokom izvršavanja programa, kada se pozivaju dinamički metodi klase, ova tabela se koristi da razreši poziv. Znači kompajler o njoj samo zna šta koji sadržaj treba da se u nju smesti i na kojoj adresi u statičkoj memoriji.

Kako izgleda struktura virtuelne tabele u mikrojavi? Ona ima niz ulaza, za svaki metod klase. U konkretnom primeru imamo dva metoda. Prvi ulaz je evo ide od ovoga 'm' do ovoga codeadr1. Drugi ulaz ove tabele ide od ove ćelije ispod do ovoga codeadr2. Kraj tabele (kraj liste metoda) signalizuje ova konstanta -2. Ulaz za određeni metod sadrži niz znakova koji predstavljaju ime metoda, znači 'm' 'a' 'g'. Kraj niza znakova označava se konstantnom -1.

Svaki znak se pamti posebno u celoj memorijskoj reči od 32 bita. To je tako urađeno, deluje kao neko manje rasipanje memorije, ali u suštini to u toku interpretacije programa obezbeđuje bržu interpretaciju, a pošto ukupan broj metoda svih klasa nije tako velik, znači nije toliko veliko trošenje memorije na ovaj detalj. Ime metoda završava konstantnom -1. Poslednji element u ulazu v-tabele koji odgovara određenom metodu je početna adresa metoda. Znači kompajler prevodi metod u mikrojava bajtkod i smešta ga u programsku oblast memorije virtuelne mašine, a početnu adresu prevedenog koda upisuje u ovaj ulaz virtuelnw tabele. Iza tog ulaza ide ulaz za drugi metod met klase, prvo niz znakova koji predstavljaju ime metoda, a zatim konstanta -1 označava kraj tog niza znakove i početna adresa koda za prevedeni metod met i kraj -2 virtuelne tabele.

To je znači bila struktura virtuelne tabele. Ostalo je da se opiše struktura same klase. Imamo strukturni zapis klase. Klasa u ovom primeru ima ukupno pet svojih elemenata, znači tri su polja i dva su metodi, to je jednako dužini liste fields objektnih čvorova. U listi objektnih čvorova, na početku liste prvo idu sva polja, znači x, y, z a onda slede redom za svaki metod po jedan objektni čvor. Ono što na ovom dijagramu nije prikazano, to je da ovaj drugi metod ima formalni parametar u i onda bi tu postojao dodatni objektni čvor tipa promenljive koji je vezan za ovaj metod met. Znači taj detalj nije prikazan, da bi se kompletirala ova slika. Treba još prisetiti se i posebno napomenuti nešto što takođe ono što nije prikazano na slici, a što svaki dinamički metod ima, to je implicitan parametar this. Argumentu this odgovara isti strukturni zapis kao i klasi unutar koje se metoda deklariše. Znači ono što fali naravno i ovde i ovde je pokazivač na dodatni objektni čvor, koji bi opisivao znači to ime this i unutar tog objektnog čvora, polje type ukazivalo bi na upravo ovaj strukturni čvor koji odgovara ovoj klasi. Iako parametri metoda nisu ovde prikazani, biće prikazani u primeru u nastavku.

Nasleđivanje klasa

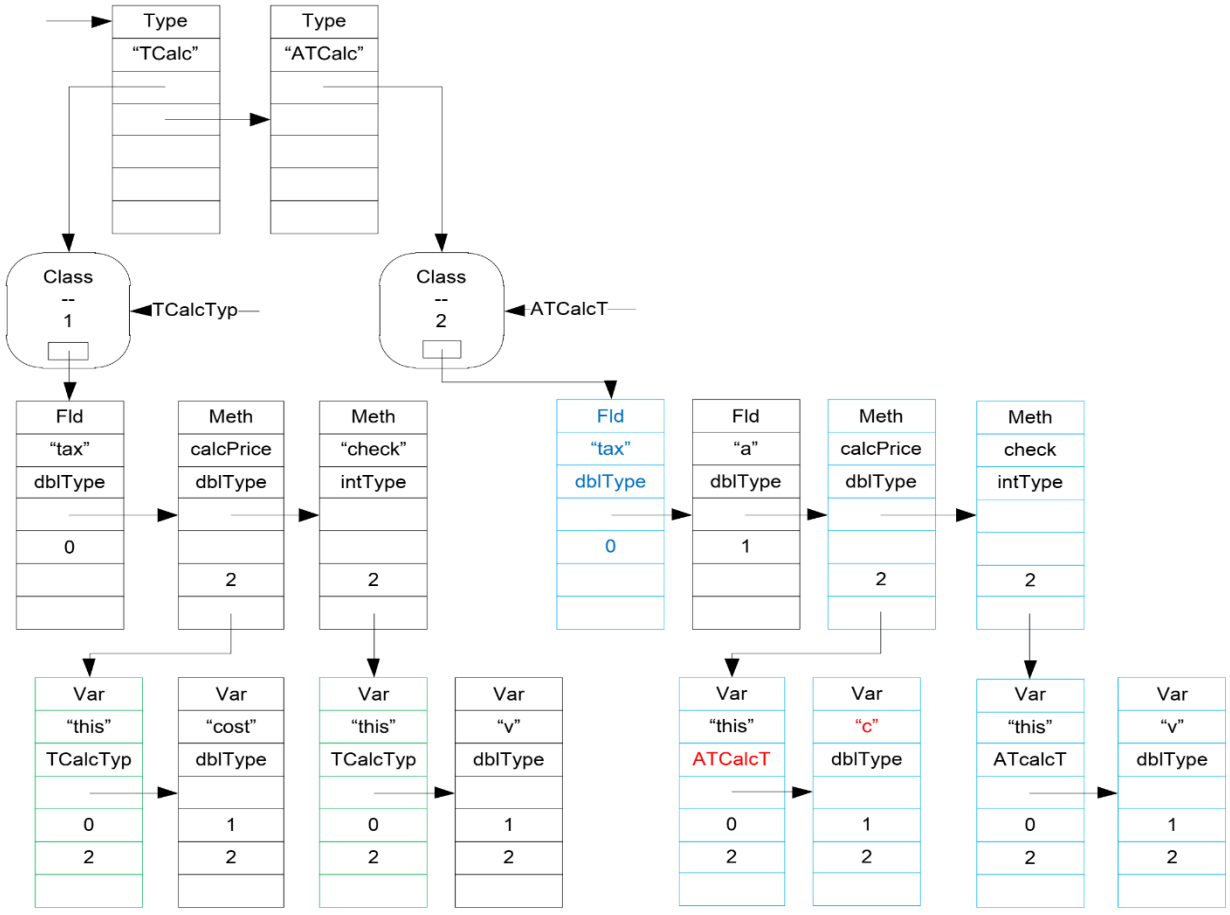
Finalni primer za predstavljanje tipa klase je najsloženiji i ilustruje nasleđivanje klasa. Znači u ovom primeru imamo dve klase, osnovnu klasu TCalc koja ima jedno polje tax, ovde za potrebe primera uvedeno kao da postoji osnovni tip double (realan broj). Znači imamo polje tax osnovnog tipa double i imamo metode calcPrice i check. Ovaj primer je tek u sintaksi mikrojave jer sintaksa mikrojave zahteva i taj detalj, to smo izostavljali zbog jednostavnosti iz prethodnih primera, da kada prođu deklaracije svih polja i krenu deklaracije metoda, mora se otvoriti dodatan par vitičastih zagrada i onda ide niz deklaracija metoda. Znači imamo metode calcPrice check u ovoj osnovnoj klasi TCalc. Klasu TCalc proširuje klasa ATCalc. znači ona je nasleđena iz klase TCalc i pored ovoga što nasleđuje polje tax i pomenuta dva metoda, dodaje sopstveno polje a i takođe vidimo da redefiniše nasleđeni metod calcPrice, znači pruža sopstvenog implementaciju ovoga metoda.

```
class TCalc {
    double tax;
    {   double calcPrice(double cost) {
           return cost*(1+tax);
        }
        int check(double v) {
           return d>0;
        }
    }
}
class ATCalc extends TCalc {
    double a;
    {   double calcPrice(double c) {
           return (cost*a)*(1*tax);
        }
    }
}
```

Da vidimo kakva struktura podataka u mikrojavinoj tabeli simbole odgovara ovim deklaracijama. Kao što vidite, primer je dosta elaboriran, ali ovde nijedan detalj nije izostavljen. Imamo za ime prve klase prvi objektni čvor i onda je to uvezano na ime druge klase, drugi objekti čvor. Struktura prve klase opisana je ovim strukturnim čvorom. Ovo označava da na nekim drugim mestima imamo u suštini pokazivače na ovaj čvor i ovde je nazvan prosto taj pokazivač, nije crtana

eksplicitna grana orijentisana odavde dovde da se ne bi opterećivala slika, nego je prosto označeno, gde god piše TCalcTyp, u stvari postoji pokazivač ka ovom strukturnom zapisu. Znači ovaj strukturni zapis odgovara klasi TCalc, a drugi strukturni zapis odgovara klasi ATCalc.

Rekli smo da je osnovna klasa TCalc, ona ima svoje jedno polje tax tipa double. Tipu double bi odgovarao jedan strukturni zapis koji ovde nije nacrtan, to bi bio jedan Struct čvor osnovnog tipa. Potom vidimo da u listi članova klase posle polja Idu metodi calcPrize i check. Ovde im je povratni tip za svaki metod, a ovde su sada eksplicitno prikazani parametri, koristi se polje locals da pamti listu objektnih ćvorova za parametre metoda. Evo vidimo implicitan parametar this, koji za calcPrice metod ukazuje na ovaj strukturni čvor za TCalc klasu i posle toga ide eksplicitan parametar cost koji je osnovnog tipa double. Što se tiče drugog metoda check, isto imamo implicitan parametar this, zato je on ovom zelenkastom bojom nacrtan, da se naglasi da je to implicitan parametar. Svejedno, moramo ga imati u tabeli simbola isto ukazuje na TCalc tip i eksplicitan parametar v osnovnog tipa double.



Kako tačno ide procesiranje deklaracije klase kada imamo i situaciju nasleđivanja, odnosno proširivanja neke osnovne klase? U slučaju da klasa nasleđuje drugu klasu, prvo se u tabeli simbola pronalazi čvor natklase, pa se pravi kopija čvorova koji predstavljaju polja, znači prvo se samo obrađuju polja ne i metodi natklase i te kopije se redom stavljaju u tabelu simbola kao polja potklase.

Znači kada se obrađuje ova deklaracija ATCalc, vidi se da je ona nasleđena iz TCalc, pronalazi se klasa TCalc, prvo njeno ime, pa zatim ova struktura, pa se onda dolazi do ovoga polja. Pravi se fizička kopija ovog objektnog čvora i popunjava se istim podacima, znači to sad postaje polje nasleđene klase. Ovde je plavičastom bojom označena činjenica da je ovo nasledjeno polje.

Posle toga se obrađuju polja koja deklariše sama potklasa i stvaraju novi objektni čvorovi, koji se nadovezuju na čvorove nasleđenih polja, po onom ranije opisanom principu proširivanja na desno, ka višim memorijskim adresama. U našem primeru ova izvedena klasa ima sopstveno polje a, znači ova lista se nastavlja, tako da je sledeći objektni čvor polje a.

Kada su sva polja stavljena u tabelu simbola dolazi na red ubacivanje Obj čvorova dinamičkih metoda. Ti čvorovi se nadovezuju na čvorove polja, opet će se prvo pogledati čvorovi natklase u tabeli simbola, napraviti kopije čvorova njenih metoda i kopije formalnih argumenata i lokalni promenljivih i redom nadovezati na čvorove polja.

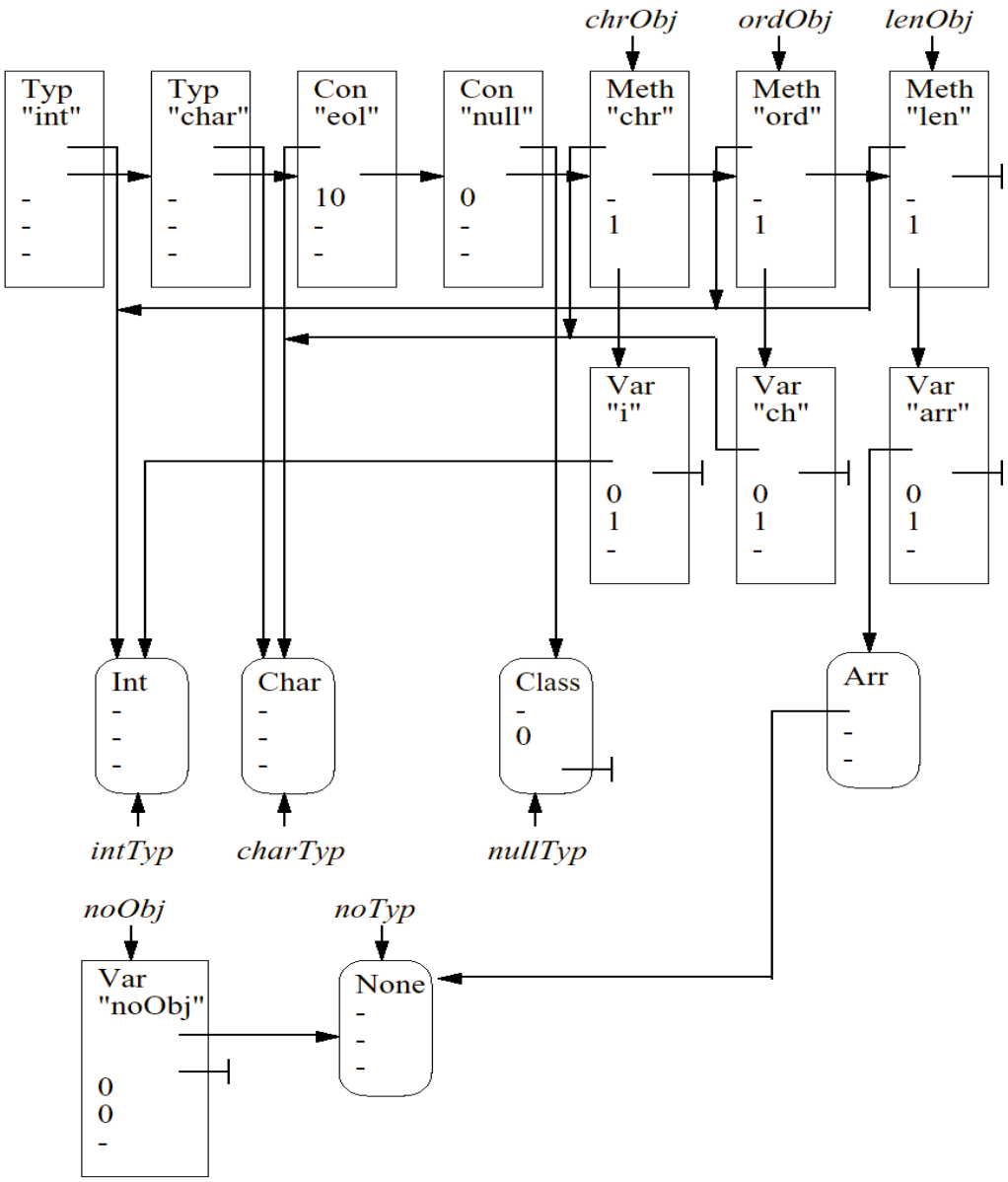
Posle toga idu metodi koji deklariše sama potklasa. Potpis svake metode deklarisane u potklasi se prvo uporedi sa potpisima metoda nasledjenih iz natklase, da bi se utvrdilo da li je metod redefinisan. Ako nije, za taj metod se pravi novi čvor u tabeli simbola, ako jeste, neće se praviti novi čvor, već će biti modifikovan postojeći čvor nasleđen iz potklase koji odgovara toj metodi. Čvor se modifikuje zato što je moguće da redefinisani metod nema ista imena formalnih argumenata i broj i tipove lokalnih promenljivih kao metoda koju redefinišemo. Takođe nedefinisani metod ima zaseban kod u programskoj memoriji, pa treba promeniti polje adr čvora metoda.

Evo opet da se vratimo na primer i pogledamo kakvo procesiranje vezano za metode ove nasleđene klase imamo. Prvo je rečeno da se prepisuju nasleđeni metodi, to su calcPrice i check. Pristupa se strukturi osnovne klase i krene se kroz ovu listu, sad se obrađuju metodi i vrši se fizičko kopiranje ova dva objektna čvora za calcPrice i check, ali takođe i ovih povezanih čvorova, znači čitavog ovog dela grafa koji opisuje njihove formalne metode. Tim kopiranjem smo dobili calcPrice sa Obj čvorom za implicitni parametar this i inicijalno je bio objektni čvor koji se zvao cost. Ovde je finalno stanje, pošto je ovaj metod overajdovan, onda je došla do izražaja modifikacija podataka i modifikovane vrednosti su upisane ovde. Inicijalno je ovde pisalo, kad smo izvršili bukvalnu kopiju, this je imao tip TCalc, a lokalna promenljiva se zvala cost. Takođe kod metoda check smo bukvalno prekopirali ovaj objektni čvor i ova dva objektna čvora za njegove parametre i vidimo da je tu samo jedna izmena bila u tipu parametra this, inače drugih izmena nije bilo.

To su bili nasleđeni metodi i posle toga se proverava da li sama izvedena klasa ima neke svoje metode. Da je postojao neki poseban metod sasvim nov, on bi se dalje nadovezivao u isti iza ovih metoda. Pošto u ovom primeru nemamo nekih potpuno novih metoda, nego imamo redefinisani metod calcPrice, znači nećemo dodavati u ovu listu nove objektne čvorove. nego ćemo naći ovaj postojeći objektni čvor za calcPrice i izvršiti modifikaciju potrebnih podataka. Crvenim su naznačene potrebne modifikacije. Parametar this više nije tipa osnovne klase, nego mora biti tipa izvedene klase. Zo inače važi i da ovaj drugi metod koji nije overajdovan, ali svejedno uvek parametar this treba posle kopiranja izmeniti njegov tip.

Dodatno vidimo da u ovoj donjoj calcPrice ne postoje lokalne promenljive, inače bismo dodavali nove čvorove, nego Postoji samo jedan formalni parametar, međutim u izvornoj varijanti funkcije se nazivao cost, a u promenjenoj varijanti funkcije se naziva c. Zato moramo ime u ovom objektnom čvoru izmeniti u c. Time je obrađena kompletna deklaracije izvedene klase i dobijen finalni graf u tabeli simbola.

Preddefinisana imena



Gornja slika prikazuje početno stanje tabele simbola sa predefinisanim imenima i njihovim tipovima. Mi smo ranije već prikazivali preddefinisana imena, ali prikaz nije bio kompletan zato što su nedostajali svi ovi strukturni zapisi koje tada još nismo objašnjavali. Sada vidimo kompletnu sliku, gde imamo sve potrebne čvorove. Vidimo takođe da postoje u interfejsu tabele simbola i direktne reference na neke predefinisane tipove i tip kostante null (konstanta null ima svoj poseban

tip nullTyp). Za ugrađene tipove imamo za njihova imena (int, char) i objektne čvorove. I ovo smo ranije objašnjavanjali, konstanta eol, konstanta null tipa NullTyp i imamo metode za konverziju između znakova i celobrojnih vrednosti i obrnuto i metod koji određuje dužinu nekog niza, to je broj elemenata niza. Zanimljiv je recimo taj detalj na taj metod len kao parametar prihvata nekakav niz i taj niz, kao formalni parametar, znači opisan je kao niz, ima strukturni čvor array, međutim zanimljivo je da je tip njegovih elemenata takozvani noType, poseban ugrađeni tip. Ovaj noObj je ona posebna vrednost za koju smo rekli da se javlja prilikom raznih grešaka, kada recimo find ne nađe neku promenljivu.

Međutim, zašto je formalni parametar ovog Len metoda ovaj niz koji ima kao tip elementa noType? Videćemo da je to isto jedna posebna vrednost, uvedena zato što ova len metoda može da se primeni na bilo koji niz. Kakav god niz da deklarišemo, sa bilo kojom vrstom elemenata, mi na njega možemo da primenimo len. Videćemo da pri proverama tipa, kada pozivamo len, onda formalne parametre zamenjujemo stvarnim i moramo proveriti da li se stvarni parametar po tipu može dodeliti formalnom parametru. U toj proveri videćemo da postoji poseban slučaj, da ovaj noTyp odgovara bilo kom tipu elementa bilo kojeg niza. Zbog toga ovo nije mogao biti celobrojni, ili neki konkretan tip, nego je morala biti posebna vrednost noTyp.

Sistem tipova

Ostalo je da vidimo kako se vrše provere tipova u mikrojavi. Generalno gledano, sistem tipova je skup pravila kako se dodeljuju tipovi različitim delovima programa, raznim deklaracijama, izrazima, iskazima. Za mikrojava jezik, sistem tipova definisan je delovima specifikacije odeljkom A.3 Semantika i odeljkom A.4 Kontekstni uslovi. Možemo pogledati taj deo specifikacije mikrojave. U A.3 odeljku, koji definiše semantiku mikrojave, vidimo da se tip referenca odnosi na nizove i klase u mikrojavi. Zatim, definišu se tipovi konstanti, celobrojne i znakovne, zatim definiše se kada se tipovi smatraju ekvivalentnima.

Dva tipa podataka su ekvivalentna, ako imaju isto ime, ili ako su oba nizovi, a tipovi njihovih elemenata su ekvivalentni. Videćemo kasnije da će ova i još nekoliko sledećih rečenica imati svoju direktnu implementaciju u proveri tipova u metodama koji se koriste u tabeli simbola. Dva tipa podataka su kompatiblna, ako su ekvivalentna, ili ako je jedan od njih tipa reference, a drugi je null tipa. Ova definicija se koristi kada, na primer, upoređujemo dve promenljive x i y. Da bi kompajler preveo to poređenje, prvo mora da se uveri da poredite kompatibilne promenljive.

Takođe imamo kompatibilnost tipova podataka pri dodeli. Izvorišni tip je kompatibilan pri dodeli sa odredišnim tipom dst, ako su izvorište i odredište ekvivalentni, ili ako je odredište tip reference, a izvorište tipa null, ili ako je odredište referenca na osnovnu klasu, a izvorište referenca na izvedenu klasu. Ovo je potrebno pri obradi eksplicitnih dodela, recimo u naredbi dodele x=y, ili pri pozivu metoda kada se formalnom parametru dodeljuje stvarni parametar. U svim tim situacijama mora se proveriti da li su izvorište i odredište kompatibilni.

Deo ove semantičke specifikacije se odnosi na definiciju šta su sve predeklarisana imena u mikrojavi, opseg važenja i tako dalje. Sledeća sekcija su kontekstni uslovi, koji se dele na neke opšte kontekstne uslove i onda imamo, za svaku konstrukciju jezika, to nam je u ovom trenutku zanimljivije, šta sve treba proveriti. Tipično su to provere koje su deo semantičke analize kompajlera pri prevođenju programa. To je definisano za različite jezičke konstrukcije,

deklaracije, izraze, iskaze. Ima puno tih kontekstnih uslova, mi sada ne možemo da gledamo sve redom. Možemo razmotriti, na primer, za if naredbu je speficicirano, ukoliko je vrednost uslovnog izraza true, izvršavaju se naredbe u if grani, u suprotnom izvršavaju se naredbe u else grani, ako je navedena. Tip uslovnog izraza condition mora biti bool. Znači to je recimo uslov koji kompajler mora da proveri.

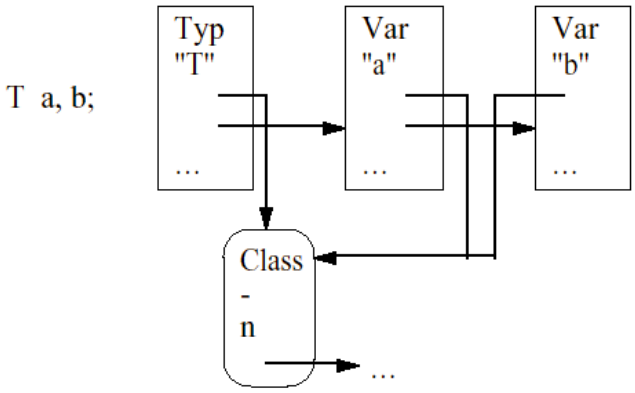
U izrazima, recimo, ako se koristi operator sabiranja i operandi su expression i term, oni moraju biti tipa int. U svakom slučaju tipovi za expression i term moraju biti kompatibilni.

Proverom tipa implementira su određeni sistem tipova. Provera može biti statička, koju vrši kompajler u fazi semantičke analize, ili dinamička, ako se izvodi u vreme izvršavanja programa. Naravno dinamičke provere mogu se vršiti pod uslovom da objekti u programu nose informacije o tipu u vreme izvršavanja. Recimo, u C++u mora biti omogućen RTTI mehanizam, tako da onda su tek na raspolaganju određeni operatori koji vrše dinamičke provere. Jezik se naziva strogo tipiziranim, ako definicija jezika, podržana statičkim i dinamičkim proverama tipova, garantuje da u izvršavanju programa neće doći do neprimećene greške upotrebe pogrešnog tipa u raznim jezički konstrukcijama.

Jezici sa klasičnim prevođenjem, kao što su java, C++, C# i tako dalje spadaju u strogo tipizirane jezike. Njih ćete prepoznati po tome što zahtevaju od svake promenljive da bude deklarisana sa određenim tipom pre korišćenja, a suprotnost njima su slabo tipizirani jezici, to su tipično razni skript jezici, gde se ne zahteva da promenljive budu deklarisane pre korišćenja, nego prostom dodelom vrednosti, znači ako u jednom delu programa napišete x = "neki string", uvodite promenljivu x i ona će u tom trenutku imati tip stringa, ako u sledećoj liniji napišete x = 1, znači dodelili ste novu vrednost promenljivoj i promenili njen tip koji je sad celobrojni.

Ekvivalentnost tipova

Kada se za dva izraza tipa može reći da predstavljaju međusobno zamenljive tipove? To nam je potrebno u nekim programskim situacijama kada imamo eksplicitne dodele vrednosti, kad imamo upoređivanje vrednosti, u aritmetičkim izrazima i tako dalje. Postoje u opštem slučaju dve vrste ekvivalencije tipova: prva je po imenu. Dva tipa su ekvivalentna ako su reprezentovana istim čvorom tipa u grafu, to jest imaju isto ime. Kada se pominje graf misli se na mikrojava reprezentaciju tipova grafovima sa strukturnim i objektnim čvorovima.

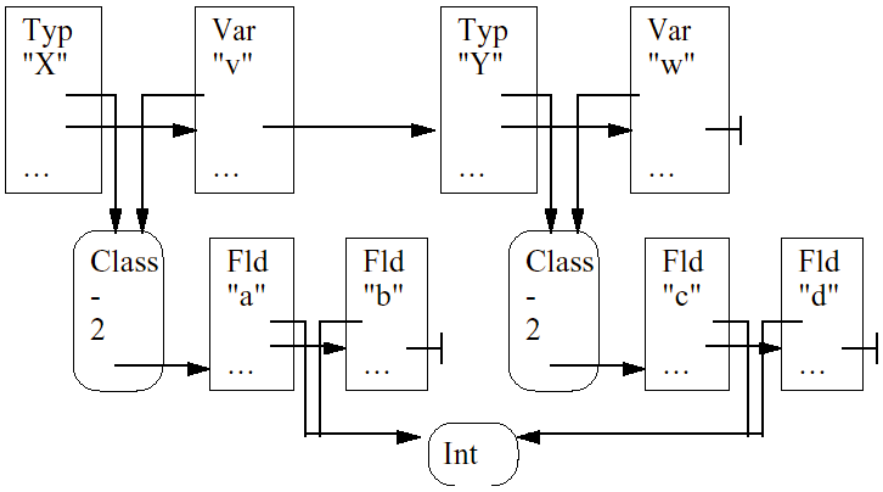


Druga vrsta ekvivalencije je strukturna ekvivalencija: dva tipa se ekvivalentna tako imaju istu strukturu, odnosno topologiju grafa. U mikrojavi, ekvivalentnost tipova je po imenu, sa izuzetkom nizova. Dva niza su ekvivalentna, ako su im tipovi elemenata ekvivalentni.

Evo kroz primere da ilustrujemo ove dve vrste ekvivalentnosti. Ekvivalentnost po imenu znači da su tipovi reprezentovati jednim istim čvorom u grafu tipova, na primer, evo kada uvedemo jednom dekoracijom dve promenljive nekog tipa T, u njihovoj reprezentaciji u mikrojava tabeli simbola postojaće dva objektno čvora za a i b i njihovo polje type će ukazivati na potpuno isti strukturni čvor. To znači, ako ukazuju na identičan čvor, da su ekvivalentni po imenu.

Ekvivalentnost tipova po strukturi: rekli smo da u opštem slučaju, ne znači da se ekvivalentni po imenu, odnosno njihovi tipovi ne ukazuju na isti čvor u grafu tipova. Evo jedan konkretan primer. U ovom primeru imamo klasu X koja ima dva celobrojna polja i klasu Y koja isto ima dva celobrojna polja. Imena polja nisu ista u klasama X i Y, niti su naravno imena klasa ista. I zatim, imamo promenljivu v tipa X, odnosno reference na klasu X i promenljive w tipa Y.

```
class X {int a, b;}
class Y {int c, d;}
X v;
Y w;
```



Kada to reprezentujemo onim strukturama koje smo objašnjavali, imaćemo promenljivu v, njen tip će biti reprezentovan ovim strukturnim zapisom klase i sa dva objektna zapisa polja a i b i ona su tipa int. Čitav ovaj deo grafa opisuje tip promenljive v. S druge strane tip promenljive w opisuje ovaj ovde deo grafa, znači isto jedan strukturni čvor i dva objektna čvora koji isto ukazuju na tip int. Rekli smo da bismo ovom primeru ustanovili ekvivalenciju v i w po strukturi, to znači da moramo ustanoviti jedan prema jedan korespondenciju između grafova koji opisuju njihove tipove, znači između svih čvorova i između svih grana. Pri tome, sadržaji čvorova ne moraju se potpuno poklapati. Vidimo da imena promenljiva nisu ista, ali sama struktura grafa je identična. Ovaj strukturni čvor odgovara ovom. ovaj objektni čvor ovom ovde, ovaj objektni čvor ovom ovde i konačno, ovaj čvor na koji ukazuju ovi objektni čvorovi je isti.

Gde važi ova ekvivalencija po strukturi? Ovaj primer je samo u sintaksi mikrojave, ali ne uklapa se u semantiku mikrojave. Znači, u mikrojavi ove promenljive v i w ne bi bile ekvivalentnih tipova. Međutim, primer bi bio ispravan u Cu uz malu sintaksnu izmenu, da se umesto klasa upotrebe strukture. U Cu, dva tipa strukture (zapisa) su ekvivalentni, ako imaju isti broj polja istih tipova.

Provera tipova

Sada ćemo videti kako je implementirana provera tipova u mikrojavi. To je nekoliko metoda koji su pridruženi klasi Struct. Oni imaju direktan opis u mikrojava specifikaciji u odeljku Semantika i kontekstni uslovi. Da krenemo od slajda gde su ovi pomoćni metodi. Rečeno je da je tip reference u mikrojavi klasa ili niz. Pošto su metode pridružene Struct zapisu, to znači da se ispituje, kada pozovete ovaj metod, this objekat, instanca struct zapisa nad kojom smo pozvali ovu proveru. Kod ovih drugih metoda zahtevaju se dva Struct objekta, jedan je this kojim pozivate metod, a drugi se prosleđuje kao parametar other.

Metod equals je osnovna ekvivalencija, koja se ne koristi posle direktno kao provera u kontekstnim uslovima, ali služi kao osnova za druga dva metoda. Ta druga dva metoda implementiraju konkretne provere.

```
class Struct {
    ...
    // checks, if two types are equal (structural equivalence for array,
    // name equivalence otherwise)
    public bool Equals (Struct other) {
        if (kind == Kinds.Arr)
            return  other.kind == Kinds.Arr &&
elemType.Equals(other.elemType);
        return other == this;
    }

    public bool IsRefType() {
        return kind == Kinds.Class || kind == Kinds.Arr;
    }
}
```

Da vidimo ovu osnovnu proveru: dva tipa su ekvivalentna, ako ukazuju na identičan čvor, to je ona ekvivalencija po imenu. Postoji samo jedan izuzetak, a to je za nizove: dva niza su ekvivalentna, ako su im ekvivalentni tipovi njihovih elemenata. Tako da vidimo ovde, ova donja grana je taj opšti slučaj, ako je other == this, ako se radi o identičnom objektu, a ovaj if obezbeđuje još ovaj poseban slučaj za nizove, znači ako je this čvor vrste niz, onda ispitujemo da li je i other čvor vrste niz i ako jeste, onda ispitujemo za njihove tipove elemenata da li su ekvivalentni. To se realizuje rekurzivnim pozivom metoda equals nad elemTyp, isto strukturnim zapisom i nad other.elemTyp. Ovim je omogućeno da recimo imamo nizove koji nisu sastavljeni od osnovnih tipova kao što je celobrojni ili znakovni, nego može biti niz nizova i slično, i svejedno će ova raditi i u takvim situacijama.

Znači ova osnovna equals definicija ekvivalentnosti tipova se koristi u ovim operativnim metodama koji se direktno koriste u semantičkoj analizi. Prvi od tih metoda je CompatibleWith. On odgovara onim situacijama kada dve promenljive treba da uporedite a nije u pitanju dodela jedne vrednosti drugoj. To može biti u nekim aritmetičkim izrazima (x + y), ili u relacionim izrazima (x > y), poziva se ova CompatibleWith provera. Jedan čvor je this, drugi čvor je other. Osnovna generička provera je preko Equals metoda, da li je this ekvivalentan sa other i imamo i određene specijalne slučajeve, znači ako ne prođe ta generička provera, onda proveravamo da li je this onog posebnog ugrađenog nullTyp tipa. a da je other tipa reference, ili obrnuto, da je other nullTyp tipa, a this tipa reference.

```
class Struct {
    ...
    // checks, if two types are compatible (e.g. in comparisons)
    public bool CompatibleWith (Struct other) {
        return this.Equals(other) ||
            this == Tab.nullType && other.IsRefType() ||
            other == Tab.nullType && this.isRefType();
    }
}
```

Zašto su nam potrebna ova dva izuzetka? Da omogućimo provere da li je neka promenljiva, koja obično referencira neku klasu, da li je jednaka null (na primer, ako joj još nismo dodelili neki konkretan objekat).

Ima još jedan metod za upoređivanje dva tipa, a to je kompatibilnost pri dodeli, u situacijama kada izvorišnu promenljivu dodeljujemo odredišnoj promenljivoj, pri čemu this reprezentuje tip izvorišne promenljive, a kroz parametar dest se dobija tip odredišne promenljive. Kako je ovaj metod implementiran?

```
class Struct {
    ...
    // checks, if this can be assigned to dest (explicit. assignment,
parameters)
    public bool AssignableTo (Struct dest) {
        if   (  this.Equals(dest) ||
            this == Tab.nullType && dest.IsRefType() ||
            kind == Kinds.Arr && dest.kind == Kinds.Arr &&   // for len()
                dest.elemType == Tab.noType)
              return true;
         for (Struct s = elemType; s != null; s = s.elemType) {
              if (s.equals(dest))
                    return true;
          }
          return false;
    }
}
```

On je kao što se vidi najsloženiji od svih ovih metoda, ima nekoliko elemenata. Prvo se radi opšta provera preko metoda equal, znači da li su, po onoj baznoj definiciji, ta dva tipa ekvivalentna. Ako jesu ekvivalentni, biće vraćeno true. Ako nisu ekvivalentni po baznoj definiciji, imamo specijalne slučajeve. da je this ugrađenog null tipa, a odredište tipa reference. to je znači da omogući eksplicitnu dodelu x = null promenljivama tipa reference, to su nizovi i klase.

Imamo još jedan specijalan slučaj koji se odnosi na nizove, znači ako je this odnosno izvorište tipa niza i odredište tipa niza i ako je odredišni tip elemenata ugrađeni tip noTyp. Razjašnjava nam ovaj komentar svrhu ovog specijalnog slučaja, a to je da bi se omogućila implementacija ugrađene metode len, koja vraća broj elemenata bilo kog niza. Da bi se tu mogao proslediti bilo koji niz, sa bilo kojim tipom elemenata, mi moramo obezbediti da bilo koji tip elementa bude ekvivalentan sa noTyp jer to je tip formalnog parametra ove len ugrađene funkcije. znači formalni parametar je niz čiji su elementi noTyp i onda takav niz mora da bude ekvivalentan sa bilo kojim realnim nizom. To je sve ova prva grana.

Ako ne uspeju sve te provere, imamo ovaj nastavak sa for. Nastavak sa for je potreban da bismo obezbedili mogućnost da dodelimo objekat izvedene klase (potklase) promenljivoj tipa natklase i to može biti njena direktna potklasa, ili u grafu nasleđivanja može da bude i više čvorova između bazne klase koja je više u grafu nasleđivanja i izvedene klase. Tako da ova for petlja nam obezbeđuje te provere, znači ako ne pripadaju direktno istoj klasi, onda krenemo od this.elemTyp. To polje, rekli smo, ukazuje na natklasu klase this, ako takva postoji, elemTyp će biti različito od null. Znači idemo na natklasu i pošto je to for petlja znači mi ćemo se prošetati, ako ta natklasa ima svoju natklasu i tako dalje. To formira čitav lanac. Ova for petlja ide po celom tom lancu element po element. Ispitujemo da li odredišni čvor po tipu odgovara toj natklasi. Znači odredište može biti bilo koja od natklasa klase this i onda je to ispravno za dodelu. Ako prođemo kroz celu for petlju, a nema tog poklapanja, onda ćemo na kraju vratiti false, da ta dva tipa nisu kompatibilna za dodelu.

Tipovi u drugim OO jezicima

Za kraj lekcije ostalo je da pomenemo još neke stvari u vezi tipova i provera tipova koje mikrojava nema, ali neki drugi jezici imaju. U izrazima gde se koriste operandi različitih tipova, na primer, sabiranje celobrojne vrednosti sa realnom, mora se vrednost jednog operanda konvertovati u drugi tip, da bi se mogla operacija sprovesti jer operacije su tipično definisane, ako su binarne, nad operandima isto tipa. Često se te konverzije rade tako da se jedan od operanada konvertuje u širi tip, onaj koji može predstavljati veći skup vrednost ili u većoj preciznosti.

Konverzija može biti eksplicitna ili implicitna. Eksplicitnu konverziju mora programer eksplicitno da zada, u Covskoj familiji jezika to se obično radi tako što ispred izraza, stavite u zagradi tip i time zahtevate od kompajlera da generiše kod koji će vrednost iz njenog originalnog tipa prevesti u tip naveden u zagradi. Kompajler pri tome naravno mora da izvrši proveru da li je ovaj oritinalni tip kompatibilan za ovu operaciju.

Neki jezici u svojim semantičkim pravilima definišu implicitne konverzije (engl. coercion) gde sam komajiler zna da konvertuje jednu vrednost u drugu u nekim situacijama. Još da pomenemo da pored termina redefinisanja funkcija, one situacije kada jedna klasa nasleđuje drugu klasu, pa nasleđuje metod iz bazne klase i onda ga redefinišemo (overajduje), tako što pruža sopstvenu implementaciju, postoji i preklapanje funkcija koje (engl. overloading) a to je da u jednom istom opsegu važenja imena možete imati isto ime funkcije koje se odnosi na različite funkcije. Na primer u C++u možete imati jednu funkciju koja prima celobrojni parametar i njeno telo je jedno i možete imati neku sasvim drugu funkciju koja prima parametar nekog drugog tipa i radi neku sasvim drugu obradu.

```
int fja( int x ) { x = 1; }
int fja ( T& t ) { t.end = true; }
```

To može biti dozvoljeno i sistemom tipova jezika je onda definisano kako se razrešava ime u zavisnosti od konteksta upotrebe, znači kompajler mora jednoznačno da može da odredi da li se upotrebili prvu ili drugu funkciju. On to određuje tako što gleda tip stvarnog parametra i gleda da ga mapira na neki od tipova formalnih parametara i tako dalje. To može da zakomplikuje pravila provere tipa i sistem tipova, ali neki jezici znači pružaju i tu mogućnost.

Izvršno okruženje

Uvod

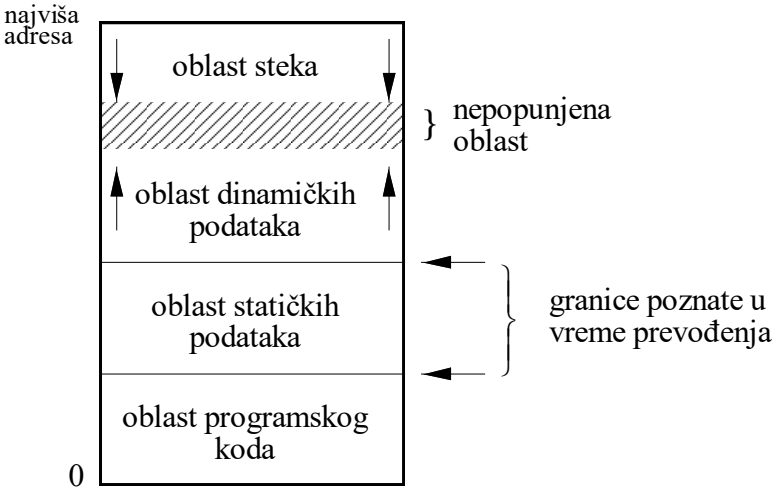
U ovom poglavlju biće reči o izvršnom okruženju programa. Taj pojam odnosi se na podršku pri izvršavanju programa na višem programskom jeziku i obuhvata sledeća razmatranja:

- Izvršna organizacija memorije -- kako se predstavljaju različiti tipovi podataka i kako se dodeljuje memorija promenljivama različitih memorijskih klasa u programu? Kako se pristupa ovim promenljivama?
- Pravila vidljivosti -- Kako upotreba određene promenljive uparuje sa odgovarajućom deklaracijom? Kako se iz određene procedure pristupa promenljivoj koja nije definisana u toj proceduri (nelokalnoj promenljivoj)?
- Poziv procedura – Kako se generiše kod za poziv i povratak iz potprograma? Kako se pri pozivu procedura prenose stvarni parametri? Kako se realizuju rekurzivne procedure?
- I druge stvari, znači izvršno okruženje obuhvata podršku obradi izuzetaka, podršku simboličkom debagovanju, dinamičkom povezivanju i tako dalje.

Obim ovog poglavlja ne dozvoljava da govorimo o svim ovim aspektima, ali ćemo pomenuti najvažnije od njih.

Izvršna organizacija memorije

Kako se deli operativna memorija vreme izvršavanja programa? Ovo je jedna tipična slika. Na ovoj slici memorijske adrese rastu od dole na gore, znači dole najniža gore najviša adresa. Operativna memorija deli se na neke logičke zone koje imaju različitu namenu.



Slika 1

Na prethodnoj slici prikazano je da memorija počinje oblašću programskog koda gde se nalazi prevedeni binarni mašinski kod, koji mora da se napuni neposredno pre početka izvršavanja programa. Zatim na to je nadovezana oblast statičkih podataka, znači globalnih promenljivih,

virtuelnih tabela i drugog, podataka koji su prisutni od početka izvršavanja programa i sve vreme izvršavanja programa. Granice obe ove oblasti su fiksne i poznate kompajleru u vreme prevođenja programa.

Ostatak memorije predstavljaju oblasti koje se po potrebi alociraju i to su oblast dinamičkih podataka, odnosno heap i oblast procesorskog steka. Heap se koristi za promenljive kojima se u programu eksplicitno dodeljuje, a takođe može dinamički da se eksplicitno i implicitno oduzima memorija, znači to su razne konstrukcije recimo u Javi, C++u operator new. Razlika između Jave i C++ je što C++ ima eksplicitnu dealokaciju sa delete, a Java ima implicitno "sakupljanje smeća". Znači sve to utiče na sadržaji i veličinu ove oblasti dinamičkih podataka. Ona u principu raste na gore (ka višim memorijskim adresama) od oblasti statičkih podataka, kako se memorijski prostor dodeljuje promenljivama.

Deo memorije od vrha adresnog prostora programa pa naniže, koristi se za oblast izvršnog steka procesora, koja se koristi tipično za podršku pozivanju potprograma i lokalne promenljive. Deo memorije je naravno neiskorišćen i ove dve oblasti idu u susret jedna drugoj. Obično postoji neka hardverska provera tih granica, da se memorija ne prepuni. Najčešće u savremenim uslovima koristi se u operativnom sistemu virtuelna memorija, tako da proces (program u izvršavanju) dobija svoj virtuelni adresni prostor i znači sopstvene adrese koje po nahođenju može da koristi. Naravno realna slika obično je za nijansu složenija, na primer, postoji prostor za razne dinamičke biblioteke i tako dalje, deo memorije se koristi za potrebe operativnog sistema, ali ovo je jedna konceptualna slika koja nas zadovoljava za naša razmatranja.

U nastavku, kada upotrebimo termin statički, to se odnosi na vreme kada radi kompajler znači veličine su poznate u vreme prevođenja, dok termin dinamički se odnosi na veličine koje nisu poznate u vreme prevođenja, nego se menjaju tokom izvršavanja programa.

Memorijske klase promenljivih

Vrednost neke promenljive u toku izvršavanja programa može se čuvati na različitim mestima. Neke promenjive mogu biti u procesorskom registru, one se nazivaju registarske promenljive. Zatim, neke promenljive, njihove vrednosti, mogu da se čuvaju u oblasti statičkih podataka, to su takozvane statičke promenljive. U oblasti steka su takozvane automatske promenljive, to je terminologija iz C/C++ i u oblasti dinamičkih podataka takozvane dinamičke promenjive. Znači, promenljive poseduju različite memorijske (engl. *storage*) klase.

Memorijske klase promenljivih u nekim programskim jezicima

Kako u programskim jezicima programer specificira memorijsku klasu promenljive? To je ili implicitno određeno nekim jezičkim konstrukcijama, na primer u Paskalu promenljive deklarisane u glavnom programu smeštaju se statički, lokalne promenljive procedura uključujući parametre, smeštaju se na steku, a dinamičke promenljive se pozivom funkcije new alociraju (dodeljuje im se proctor u memoriji i počinju da egzistiraju).

U C-u se globalne promenljive, deklarisane izvan procedura i funkcija smeštaju statički. Lokalne promenljive procedura i njihovi parametri smeštaju se na steku. Postoje funkcije za memorijsku alokaciju malloc, calloc i slične, kojima se može u dinamičkoj oblasti dodeliti memorija. Prilikom deklarisanja promenljive, C ima različite može se navesti ključna reč specifikator memorijske klase, kao što je register, static ili auto za lokalne promenljive koje se smeštaju na steku. Međutim, recimo, možete u potprogramu uvesti lokalnu promenljivu, a onda u deklaraciji staviti static i C

kompajler će znati da tu promenljivu, ako je lokalna za funkciju, čuva u statičkoj oblasti i na taj način, da njena vrednost preživljava između različitih poziva funkcija. Ako navedete memorijsku klasu register kompajler će nastojati da vrednost te promenljive čuva u nego procesorskom registru to nije garantovano to je samo preporuka programera kompajleru, a kompajler sam odlučuje u postupku generisanja koda i optimizacije, gde će da stavi promenljivu.

Način pristupa skalarnoj (na primer, celobrojnoj) promenljivoj

U zavisnosti od memorijske klase, u kodu će se promenljivoj pristupati različitim načinima adresiranja. Recimo, registarskim promenljivama pristupaće se registarskim direktnim adresiranjem. Statičkim promenljivama pristupa se memorijskim direktnim adresiranjem jer su i veličina i početna adresa promenljive su poznati u vreme prevođenja programa. Automatskim, odnosno promenljivama koje se dodeljuju na steku koristi se za pristupanje njima varijanta indeksnog adresiranja, koja se najčešće zove bazno adresiranje. To je u svojoj suštini vrednost u nekom registru koja je poznata u runtime-u, plus neka konstanta koja je poznata kompajleru, znači saberu se te dve vrednosti i dobije se finalna adresa. U nastavku će o ovome biti više reči. Konačno, dinamičkim promenljivama se pristupa nekom vrstom indirektnog adresiranje. Znači adrese tih promenljivih takođe, pošto su dinamički dodeljuju, ne mogu biti poznate kompajleru, nego tipično postoje pokazivači ili reference u programu, oni čuvaju adrese takvih promenljivih i onda pristupom preko pokazivača pristupamo vrednostima tih promenljivih.

Studija slučaja: Arhitektura procesora Intel 80x86

Za konkretne primere kojima ćemo ilustrovati razne koncepte izvršnog okruženja koristićemo asemblerski kod za Intelovu x86-icu, onu najstariju 16-bitnu varijantu. Sadašnji procesori koriste 64-bitnu varijantu i kod je donekle različit, ali u suštini nije bitno drugačiji, pošto je evoluirao iz ove najstarije 16-bitne varijante. Ovo treba shvatite kao konceptualne primere, znači tipično sada kod 64-bitnih procesora AX bi se zvao RAX i nekada je bio širine 16 bita, sada je 64. Nekadašnja x86 imala je nekoliko registara opšte namene, to su AX, BX, CX i DX i takozvane indeksne registre SI i DI. Adresiranje je na nivou bajta i na steku su se čuvale 16-bitne veličine. Za rad sa stekom koriste se 16-bitni registri SP (engl. *Stack Pointer*) i BP (engl. *Base Pointer*). Stek raste prema nižim memorijskim lokacijama. Registar stek pointer ukazuje na poslednju popunjenu lokaciju.

Sledeća tabela predstavlja mali izbor iz seta instrukcija, koliko će nama trebati za primere koje pokazujemo. Znači imamo instrukcije za rad sa stekom push i pop kojima operand može biti registar ili memorijska lokacija. Zatim poziv potprograma -- tekuća vrednost brojača naredbi ide na stek i vrši se skok na adresu specificiranu operanom, koji opet može biti dat ili kao vrednost registra, ili kao neka direktna memorijska adresa. Naredba za povratak iz potprograma skida vrednost sa vrha steka i to ide u brojač naredbi IP. Postoji opcioni parametar c, to je jedna celobrojna vrednost. Ako je c zadato, onda se stek pointer pre skidanja povratne vrednosti sa steka uvećava za vrednost specificiranu operandom c. Videćemo u nastavku da ta varijanta naredbe služi da se sa steka dealociraju parametri poziva potprograma, znači ima svoju ulogu u sekvenci poziva i povratka iz potprograma. Konačno, koristićemo mali broj aritmetičkih naredbi. Procesor 8086 je bio u svojoj osnovi dvoadresna mašina, tako da su bila dva operanda i recimo naredba sabiranja dst, src sabira vrednost operanda src sa operandom dst i rezultat opet smešta u lokaciju dst. Slično

tome naredba oduzimanja i naredba direktnog kopiranja iz izvorišta u odredište. Dakle ovo je neki mali skup naredbi koji je dovoljan za naše primere.

Tabela 1

Instrukcija	Objašnjenje
PUSH r/m	stavlja vrednost operanda na stek
POP r/m	skida vrednost sa vrha steka i dodeljuje je operandu
CALL r/m	stavlja tekuću vrednost brojača naredbi IP na stek i vrši skok na adresu specificiranu operandom
RET [c]	SP uvećava za vrednost specificiranu operandom c; skida vrednost sa vrha steka i dodeljuje brojaču naredbi IP
ADD dst, src	Sabira vrednost operanda src sa operandom dst i rezultat smešta u dst
SUB dst, src	Oduzima vrednost operanda src sa operandom dst i rezultat smešta u dst
MOV dst, src	Dodeljuje operandu dst vrednost operanda src

Oznake korišćene za operande imaju sledeće značenje:

- r/m označava registarsko direktno ili memorijsko direktno adresiranje
- [c] označava da je operand opcioni. Ako je naveden, zadat je neposrednim adresiranjem; ako nije naveden, podrazumeva se vrednost 0.
- src kod MOV naredbe označava proizvoljan način adresiranja, a kod ADD i SUB naredbi registarsko direktno, memorijsko direktno ili neposredno adresiranje
- dst kod MOV označava proizvoljan načina adresiranja osim neposrednog, a kod ADD i SUB naredbi registarsko direktno ili memorijsko direktno adresiranje. Ako je jedan od operanada MOV, ADD i SUB naredbi u memoriji, drugi mora biti u registru.

Načini adresiranja kod 8086

Načini adresiranja procesora 8086 koje ćemo koristiti u navedenim naredbama, navedeni su u sledećoj tabeli.

Tabela 2

Primeri sintakse	Objašnjenje	Naziv
100	vrednost operanda je 100	neposredno
labela [100]	adresa operanda je zadata vrednost	memorijsko direktno
AX BP	vrednost operanda je sadržaj registra	registarsko direktno
[BP] [SI] [DI]	adresa operanda je sadržaj nekog od registara BP, SI, DI.	registarsko indirektno
[BP+4] [DI-100] [SI-20]	adresa operanda je sadržaj registra ± konstanta	bazno ili indeksno

Može da se piše konstanta, znači to je neposredno adresiranje. Ta konstanta se direktno ugrađuje u kod naredbe i služi kao konstantna vrednost. Zatim, memorijsko direktno adresiranje može da se

specificira ili konstantnom adresom, onda se ona stavlja u uglaste zagrade, znači čitamo za fiksne adrese, ili se simbolički najčešće u asembleru navodi neko ime koje zamenjuje neku konkretnu vrednost. Sledeće je registarsko direktno, specificira se navođenjem imena raznih registara. Registarsko indirektno znači da registar sadrži adresu i da, koristeći tu adresu, pristupa se memoriji i na toj lokaciji se u stvari čuva vrednost, u toj varijanti je ime registra navedeno u uglasnim zagradama. Konačno, postoji I bazno ili indeksno adresiranje. Kod tog načina adresiranja, adresa operanda se dobija kao zbir sadržaja registra i plus ili minus neke konstante.

Primer

U nekoj C proceduri promenljive r, s, a i d deklarisane su na sledeći način:

```
register int r;  /* registarska celobrojna promenljiva */
static int s;    /* statička celobrojna promenljiva */
auto int a;      /* automatska celobrojna promenljiva */
static int *d;   /* statička promenjiva tipa pokazivača na ceo broj */
```

Pod pretpostavkom da je dužina celobrojnog podatka 16-bita i da se pokazivač d inicijalizuje pozivom `d=(int *)malloc(sizeof(int))`, koji asemblerski kod odgovara naredbama dodele:

```
r=s;
a=*d;
```

Pretpostavimo da je promenljivoj r dodeljen procesorski registar SI, da je adresa promenljive s u statičkoj oblasti (određena u vreme prevođenja programa) jednaka 200. Prevodilac za naredbu dodele `r = s;` generiše sledeći asemblerski kod:

```
MOV     SI,[0200]
```

Izvorišni (desni) operand zadat je memorijskim direktnim, a odredišni registarskim adresiranjem. Vrednost se čita iz memorijske lokacije 200 i upisuje u odredišni registar SI.

Razmotrimo sada drugu naredbu dodele a = *d. Potrebno je prvo pristupiti, preko pokazivača d, promenljivoj koja se čuva u dinamičkoj memoriji (*d što označava vrednost promenljive na koju ukazuje d). Pošto je d statički alocirana, recimo da je ona na nekoj adresi 202, da bi smo pročitali vrednost pokazivača koristimo memorijsko direktno adresiranje i vrednost pokazivača upisujemo na primer u BX registar, a zatim čitamo indirektnim adresiranjem, koristeći registar BX, znači pročitamo na šta ukazuje pokazivač d i vrednost sa te memorijske lokacije upisaćemo u drugi registar AX.

```
MOV BX,[0202]          ; u BX ide sadržaj promenljive d
MOV AX,[BX]            ; u AX ide sadržaj *d
```

Sadržaj registra AX predstavlja vrednost izvorišnog operanda i tu vrednost treba upisati u odredište. Pošto je odredište lokalna promenljiva, uskoro ćemo to detaljnije videti, ali lokalnim promenljivama se uvek pristupa takozvanim baznim adresiranjem. To je vrsta indeksnog adresiranja gde je registar uvek base pointer BP. Svaka promenljiva ima svoj konstantni pomeraj u odnosu na lokaciju BP koji je poznat u vreme prevođenja programa. Pod pretpostavkom da je a jedina automatska (lokalna) promenljiva u posmatranoj proceduri, njen pomeraj u odnosu na registar BP će biti -2.

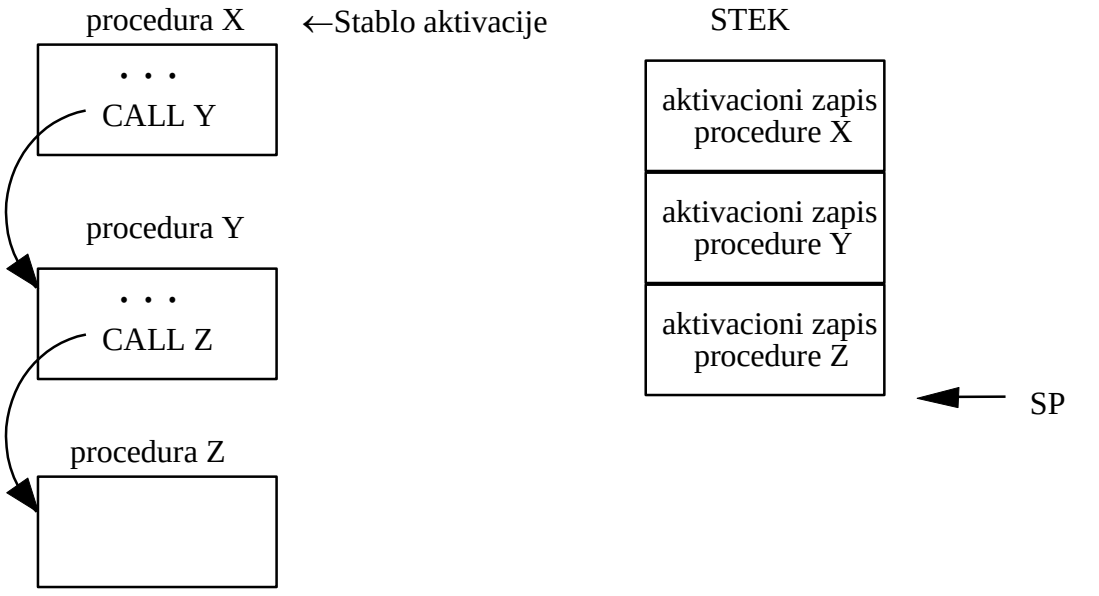
```
MOV [BP-02],AX         ; a:=*d
```

Zbog ograničenja arhitekture ovog Intel-ovog procesora, ne može se realizovati jednom instrukcijom čitanje indirektno BX, pa smeštanje na ovu adresu na koju ukazuje BP-2. To bi bila instrukcija transfera iz memorije u memoriju, a u jednoj mašinskoj instrukciji 8086 može biti samo jedno memorijsko adresiranje, znači memorija može biti samo izvorište ili samo odredište, a ne može biti i izvorište i odredište.

Pozivanje i povratak iz potprograma

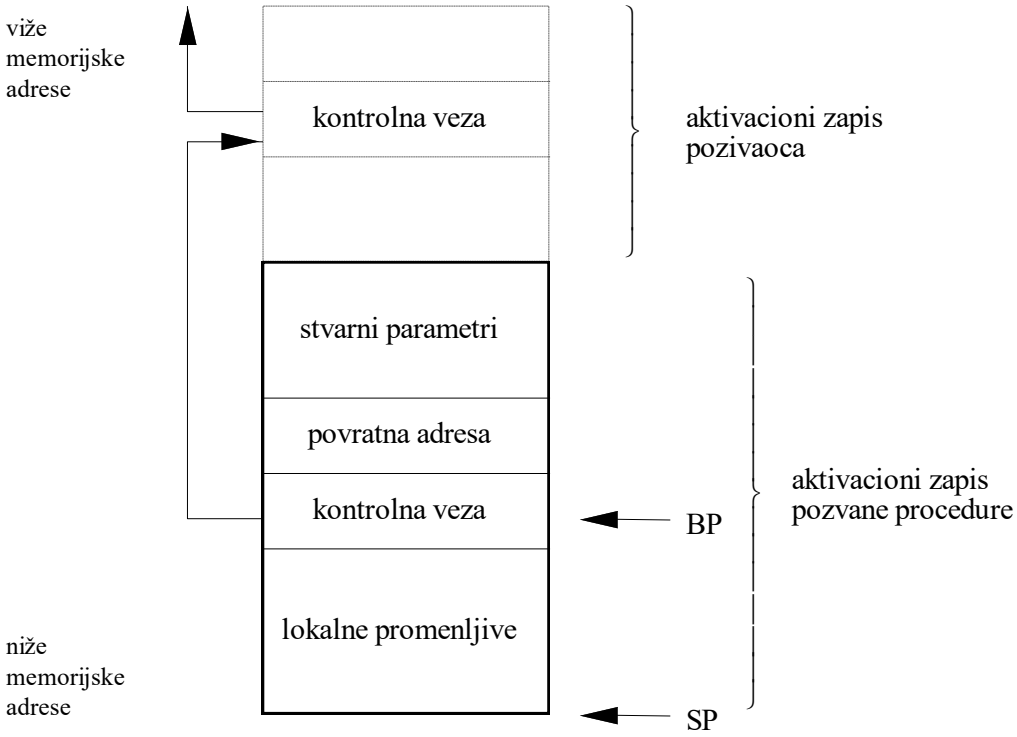
Sada ćemo detaljnije razmotriti kako se u mašinskom kodu realizuje pozivanje i povratak iz potprograma i koje memorijske strukture odgovaraju čuvanju lokalnih promenljivih. Pri svakom pozivu procedure u toku izvršavanja programa, na vrh steka se stavlja blok podataka koji koristi pozvana procedura u svom radu. Ovaj blok se zove aktivacioni zapis ili okvir, na engleskom activation record ili frame. Aktivacioni zapis na steku ostaje sve dok pozvana procedura ne vrati kontrolu pozivaocu, kada se aktivacioni zapis pozvane procedure skida sa steka.

Evo jednog primera koji to ilustruje. Čim se iz glavnog programa pozove procedura X, na steku se formira aktivacioni zapis procedure X. Uskoro ćemo razmotriti strukturu aktivacionog zapisa. On čuva sve bitne podatke za izvršavanje procedure X. Ako sada u toku izvršavanja procedure X pozovemo proceduru Y, na vrhu steka, znači iznad aktivacionog zapisa procedure X, formira se novi blok podataka, aktivacioni zapis procedure Y. On se formira u trenutku kada pozivamo proceduru Y i ostaće na steku su sve do god je aktivan taj poziv Y. Ako se iz Y zove sada neka treća procedura Z, na vrh steka iznad Y dolazi aktivacioni zapis procedure Z. Kako se završavaju ovi pozivi? Kada procedura Z završi rad i vrati se u Y, sa vrha steka prvo će biti uklonjen aktivacioni zapis procedure Z. Kada se izvrši povratak iz Y u X, sa steka će biti uklonjen aktivacioni zapis Y i konačno, kada procedura X završi rad i vrati se u glavni program, sa steka će biti uklonjen i aktivacioni zapis procedure X.



Struktura aktivacionog zapisa

U nastavku ćemo da konkretizujemo prethodnu priču. Prvo ćemo razmotriti varijantu koja odgovara C jeziku, a to je varijanta kada imamo model izvršavanja gde imamo procedure, a promenljive ili mogu biti lokalne za tekuću proceduru, ili globalne promenljive koje se, kao što smo već izložili, čuvaju u statičkoj memoriji. To je u stvari relativno jednostavna situacija. Jeziku C odgovara struktura aktivacionog zapisa prikazana na slici koja sledi. Na ovoj slici punom liniju je prikazan aktivacioni zapis tekuće pozvane procedure, a isprekidanom linijom je prikazan drugi aktivacioni zapis procedure koje je pozvala tekuću proceduru.



Na dnu slike su niže memorijske adrese, a memorija raste na gore. Vrh steka je ovde gde ukazuje stek pointer SP. U trenutku kada pozivalac, čiji aktivacioni zapis je prikazan isprekidanom linijom, pozove tekuću proceduru, prilikom tog poziva formira se na vrhu steka blok podataka koji čini aktivacioni zapis pozvane procedure. U tom aktivacionom zapisu, na najvišim adresama su stvarni parametri koje pozivalac prenosi pozvanoj proceduri potiskujući ih na stek, zatim, u trenutku kada se izvrši mašinska naredba call, na stek ide povratna adresa, zatim na stek ide polje kontrolne veze. Polje kontrolne veze je u suštini stari sadržaj BP, znači do tog trenutka BP je ukazivao na aktivacioni zapis pozivaoca i to konkretno upravo na polje kontrolne veze u pozivaocu, a onda u jednom trenutku mi ažuriramo BP da ukazuje na aktivacioni zapis pozvane procedure, a staru vrednost BP čuvamo na steku.

Registar BP ukazuje na polje kontrolne (ili dinamičke) veze i služi za pristup stvarnim parametrima i lokalnim promenljivama procedure. Stvarnim parametrima pristupa se baznim adresiranjem [BP+pomeraj] gde je vrednost pomeraja za svaki od parametara poznata u vreme prevođenja. Lokalnim promenljivama pristupa se baznim adresiranjem [BP-pomeraj] gde je vrednost pomeraja takođe poznata u vreme prevođenja.

Polje kontrolne veze, odnosno mesto u aktivacionom zapisu na koje ukazuje BP, možemo shvatiti kao pokazivač na odgovarajuće polje kontrolne veze u aktivacionom zapisu pozivaoca. I taj pozivalac, ako je njega neko pozvao isto ima jedan takav pokazivač. Niz tih pokazivača zovemo lanac kontrolnih veza. Kontrolna veza korsti se kod normalnog izvršavanja programa, kada posle povratka iz procedure treba restaurirati staru vrednost BP. Osim toga, može biti od koristi recimo i pri debagovanju programa. Kada prekinete program u dibageru i želite da pogledate stek, dibager može da koristi ovo polje kontrolne veze da bi vam prikazao koji sve aktivacioni zapisi se u tom trenutku nalaze na steku i tome slično.

Pored polja kontrolne veze, na kraju deo prostora odvojen za smeštaj lokalnih promenljivih procedure i znači tu je sad vrh steka, odnosno stek pointer. Ovaj aktivacioni zapis se formira na početku poziva i egzistira na steku do god se izvršava kod pozvane procedure. Pri završetku izvršavanja pozvane procedure, ovaj zapis se dealocira sa steka i u suštini onda na vrhu steka ostaje naravno aktivacioni zapis pozivaoca. To znači da lokalne promenjive imaju svoj život od trenutka poziva do trenutka povratka iz tekućeg poziva.

Sekvenca pozivanja

Sekvenca pozivanja je niz mašinskih naredbi koje pri pozivu neke procedure izvršavaju pozivalac i pozvana procedura da bi uspostavili aktivacioni zapis pozvane procedure. Sekvencu pozivanja zajednički izvršavaju jedan deo sekvence procedura pozivalac, a drugi deo sekvence pozvana procedura. Ako poznajete strukturu aktivacionog zapisa, lako je znati koji deo formiranja zapisa radi pozivalac, a koji pozvani, ako pogledate gde je povratna adresa, jer kada dođe povratna adresa na stek, kontrola se prenosi sa pozivaoca na pozvanog. Znači ceo ovaj deo zapisa koji je više na steku, zaključno sa povratnom adresom, njega formira pozivalac. Sve što je niže od toga mora da formira pozvana procedura. Ako taksativno nabrojimo aktivnosti, njih započinje procedura pozivalac

- Na stek se redom stavljaju stvarni parametri za pozvanu proceduru. Kod jezika kao što je C, stvarni parametri idu od poslednjeg ka prvom, znači obrnutim redosledom, prvi ide poslednji na stek. To je zato što C dozvoljava promenljiv broj parametara pri pozivu i ima mehanizam da obezbedi da napišete potprogram koji možete pozvati sa promenljivim brojem parametara. Neke bibliotečke funkcije, recimo funkcija printf je dobar primer, mogu da primaju promenljiv broj parametara i onda taj redosled od poslednjeg ka prvom obezbeđuje da je uvek poznato na kom pomeraju je prvi parametar, pa posle toga znači uvećavanjem pomeraja može se doći do ostalih parametara.
- Izvršava se CALL naredba koja na stek stavlja povratnu adresu i prenosi kontrolu pozvanoj procedure.

Pozvana procedura:

- Na stek stavlja sadržaj registra BP (radi se o polju kontrolne veze).
- Sadržaj registra SP, koji u tom trenutku ukazuje na kontrolnu vezu, kopira u BP.
- Od sadržaja registra SP oduzima broj N koji je poznat u vreme prevođenja. Radi se o broju memorijskih lokacija potrebnih za smeštaj lokalnih promenljivih. Na ovaj način se na steku alocira potreban prostor za lokalne promenljive.

Nakon završetka sekvence pozivanja, izvršava se kod pozvane procedure za inicijalizaciju lokalnih promenljivih i tako dalje. Znači ovo je onaj nevidljiv deo koda koji se u programu na višem programskom jeziku i ne vidi, nego tek znači kad se taj kod završi, ide kod koji je prevedeni kod sa višeg programskog jezika.

Sekvenca povratka

Na kraju izvršavanja potprograma, prilikom povratka iz pozvane u pozivajuću proceduru, izvršava se sekvenca povratka. U suštini radi se o potpuno komplementarnoj aktivnosti prethodne sekvence pozivanja, jer sve što je tada na steku alocirano sada treba ukloniti obrnutim redom. Sekvencu povratka započinje pozvana procedura:

- Sadržaj registra BP kopira se u registar SP čime se sa steka oslobađa prostor za lokalne promenljive i one efektivno prestaju da postoje.
- Sa vrha steka (koji u tom trenutku ukazuje na polje kontrolne veze pozvane procedure) skida se vrednost kontrolne veze i smešta u registar BP. Time je ovaj registar postao spreman za upotrebu u proceduri-pozivaocu.
- Naredba RET skida sa vrha steka povratnu adresu i smešta u brojač naredbi IP. Time se kontrola predaje pozivaocu, na mestu neposredno iza mašinske naredbe call.

Pozivalac još mora da, dodavanjem odgovarajuće vrednosti na SP, sa steka ukloni stvarne parametre i time je završen poziv. Nastavlja se naravno izvršavanje nekog koda koji je bio posle poziva.

Primer

Napisati 80x86 asemblerski potprogram ekvivalentan datom programu na C-u. Koja sekvenca asemblerskog koda odgovara pozivu `primer(10,20)` u glavnom programu?

```
void primer(int a,b) {
     int c;
     c=a;
}
```

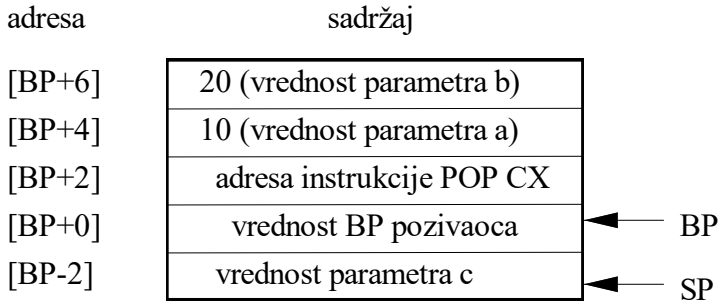
Što se tiče samog poziva koji je deo glavnog programa, potrebno je prvo na stek staviti stvarne parametre. Parametri se stavljaju redom s desna na levo, prvo 20, pa 10. Zatim ide poziv i kada se kontrola vrati u glavni program moraju se ukloniti parametri sa steka.

```
; smeštanje drugog parametra na stek
MOV AX,20
PUSH AX
; smeštanje prvog parametra na stek
MOV AX,10
PUSH AX
; poziv procedure primer
CALL primer
; skidanje parametara sa steka, vrednost CX se ne koristi
; alternativa navedenim je operacija ADD SP,4
POP CX
POP CX
; kraj poziva
```

Kod za proceduru primer dat je u nastavku. Procedura treba da formira kontrolnu vezu, namesti BP na vrednost koja njoj treba, alocira prostor od dva bajta (16bitni int) za lokalnu promenljivu c, dohvati stvarni parametar a u pomoćni registar, upiše dohvaćenu vrednost u lokalno c. Zatim sledi standardna sekvenca povratka, dealoiciranje lokalnih promenljivih, restauriranje stare vrednosti BP sa steka i povratak u program.

```
; smeštanje sadržaja BP u polje kontrolne veze
PUSH BP
; nova vrednost BP treba da ukazuje na kontrolnu vezu
MOV BP,SP
; alociranje prostora za lokalnu promenljivu c
SUB SP,2
; u AX ide vrednost stvarnog parametra a
MOV AX,[BP+04]
; u lokalnu promenljivu c ide vrednost iz AX
MOV [BP-02],AX
; oslobađanje prostora dodeljenog promenljivoj c
MOV SP,BP
; restauracija vrednosti BP iz kontrolne veze
POP BP
; povratak u pozivajuću proceduru
RET
```

Na sledećoj slici prikazan je izgled aktivacionog zapisa procedure primer. Do svih drugih delova aktivacionog zapisa može se stići preko BP sa pozitivnim ili negativnim pomerajem. Ako je recimo pomeraj negativan, dolazi se do lokalnih promenljivih, a ako je pomeraj pozitivan može se doći do stvarnih parametara. Standardno, prvi stvarni parametar je na BP+4, a prva lokalna promenjiva je na BP-2 kod ovog 16-bitnog procesora. Te konstante se proporcionalno uvećavaju, ako 32-bitni ili 64-bitni kod.



Postavlja se pitanje da li je registar BP neophodan za pristup parametrima i lokalnim promenljivama procedure. Drugim rečima, da li se za pristup može koristiti bazno adresiranje sa registrom SP i odgovarajućim pomerajem (koji bi u ovom slučaju bio uvek pozitivan) i time izbeći rezervacija posebnog registra? Odgovor je pozitivan. Na primer, GNU kompajler za C (gcc) ima opciju –omit-frame-pointer kojom se kompajleru sugeriše da koristi upravo taj pristup. Prednosti tog rešenja su da BP postaje raspoloživ za druge namene (privremeni smeštaj vrednosti neke promenljive) i da se ne mora generisati kod za formiranje kontrolne veze na početku svake procedure. Međutim, ovakav pristup zahteva da kompajler bude u stanju da isprati svaku promenu registra SP tokom izvršavanja procedure, da bi prilagodio pomeraje. U nekim situacijama, jedna od njih je korišćenje Lunix-ovske bibliotečke funkcije alloca koja na steku rezerviše varijabilni

broj lokacija u vreme izvršavanja, to nije moguće, tako da kod tih procedura mora da se koristi BP. Takođe ovo rešenje ometa debagovanju programa.

Postavlja se takođe pitanje: zašto se parametri i lokalne promenljive alociraju na steku umesto da se koriste registri procesora ili neka druga oblast memorije? Naslednici 8086 procesora, specifično x86-64 arhitektura, koristi prednost većeg broja registara opšte namene (pored proširenih rax, rcx, rdx, rbx, rsp, rbp, rsi, rdi ima i dodatne r8, r9, r10, r11, r12, r13, r14, r15) da određen broj argumenata prenese kroz registre, ostali se prenose na steku. Takođe je moguće kompajleru da optimizije generisani kod da se neke lokalne promenljive čuvaju u procesorskim registrima. Ipak ako je broj parametara veći (od četiri za Windows, odnosno šest za Linux sisteme), takođe ako se koristi rekurzija koja nije završna (engleski tail recursion), neophodno je koristiti stek za čuvanje vrednosti registara na opisani način. Jezici koji dozvoljavaju rekurziju moraju obezbediti način da u određenom trenutku izvršavanja programa jednoj istoj proceduri odgovara veći broj različitih aktivacionih zapisa u memoriji. da bi svaka aktivacija mogla imati sopstvene vrednosti parametara i lokalnih promenljivih.

Rekurzija

Kako se realizuju rekurzivni potprogrami? Stek je jako važan za realizaciju rekurzije u programskim jezicima. Evo konkretng primera. Data je C funkcija `fakt`. Pratiti tok izvršavanja poziva `fakt(3)` i odrediti izgled procesorskog steka u trenutku neposredno pre prvog povratka iz funkcije.

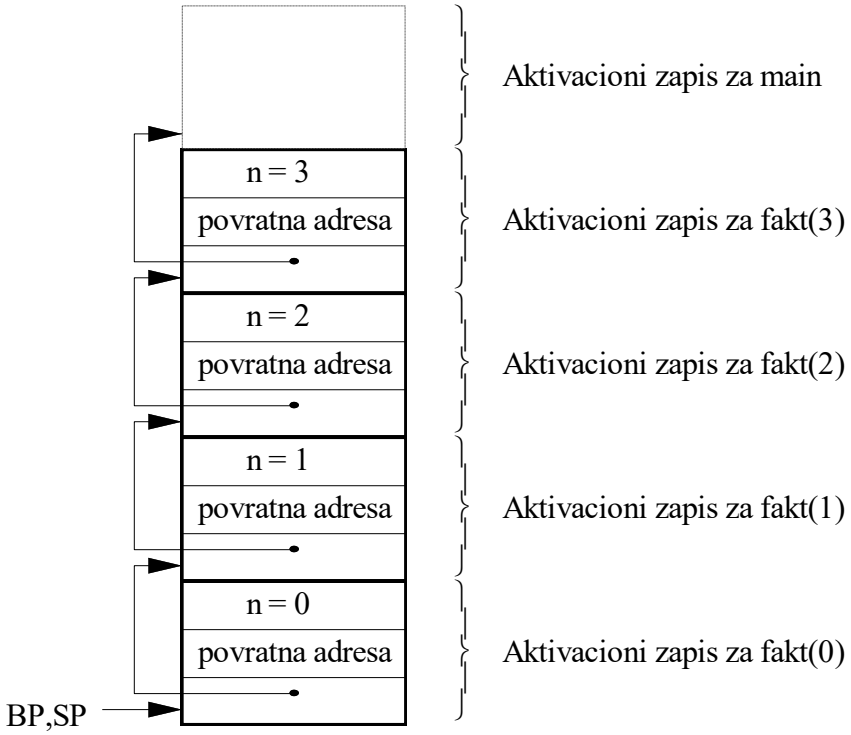
```
int fakt(int n) {
    return (n>0)?(n * fakt(n-1)):1;
}
```

Razmotrimo aktivnosti koje se odvijaju tokom izvršavanja poziva `fakt(3)`:

- Iz glavnog programa poziva se funkcija `fakt` sa parametrom n = 3. Pri ovome se na steku formira aktivacioni zapis funkcije `fakt`. Prenos parametra je po vrednosti, pa pozivalac pre naredbe CALL na stek stavlja vrednost 3.
- U funkciji `fakt`, pre nego što se izvrši `return` naredba mora se izračunati vrednosti izraza navedenog u okviru te naredbe. Radi se o uslovnom izrazu. S obzirom da je n >0 potrebno je izračunati proizvod `n * fakt(n-1)`. Da bi se odredio drugi faktor proizvoda, poziva se funkcija fakt, ovaj put sa vrednošću parametra n - 1 = 2. Ovaj poziv se realizuje kao i svaki drugi poziv funkcije, formiranjem novog aktivacionog zapisa na steku. Ovaj aktivacioni zapis uključuje i parametar, vrednost 2 tako da za novu aktivaciju funkcije `fakt` važi da je n =2.
- Kontrola izvršavanja se ponovo prenosi na početak funkcije `fakt`. Pre povratka iz funkcije, ponovo se mora izračunati vrednost drugog faktora proizvoda navedenog u naredbi return. Stoga ponovo sledi poziv funkcije `fakt`, ovaj put sa vrednošću parametra 1. Formira se novi aktivacioni zapis na vrhu steka koji u polju parametra ima vrednost 1, tako da za novu aktivaciju funkcije važi n = 1.
- Izvršavajući ponovo funkciju `fakt` od početne naredbe, ustanovljava se da je n >0 te stoga opet treba izračunati drugi faktor proizvoda `n * fakt(n-1)`. Zato sledi poziv `fakt(0)` pri čemu se na vrhu steka formira novi aktivacioni zapis.
- U novoj aktivaciji važi da je n =0 pa nije ispunjen uslov n>0. Zato je vrednost uslovnog izraza jednaka 1. U trenutku neposredno pred povratak iz ove aktivacije funkcije `fakt`, na steku se

još uvek nalaze sva četiri aktivaciona zapisa funkcije `fakt` kao što je prikazano na slici na sledećoj strani. Po izvršenju naredbe `return`, sa vrha steka skida se poslednji stavljeni aktivacioni zapis. Funkcija `fakt` može da prosledi rezultat pozivaocu u nekom procesorskom registru, ili alternativno preko steka. U ovom drugom slučaju u okviru polja parametara potrebno je rezervisati lokaciju za smeštaj rezultata (na slici je prikazan slučaj kada se rezultat vraća u registru, po standardnoj konvenciji je to registar AX).

- Po povratku iz četvrte u treću aktivaciju funkcije `fakt` računa se proizvod vrednosti parametra n=1 i vraćenog rezultata `fakt(0)` = 1 i dobijeni rezultat 1 vraća pozivaocu, pri čemu se uklanja aktivacioni zapis sa vrha steka.
- Na isti način, druga aktivacija računa proizvod parametra n=2 i vraćenog rezultata `fakt(1)`=1 i rezultat 2 vraća pozivaocu, pri čemu se skida naredni aktivacioni zapis sa vrha steka.
- Prva aktivacija računa konačni rezultat 3 * 2 = 6 i vraća ga glavnom programu, pri čemu se sa steka skida i poslednji aktivacioni zapis funkcije `fakt`. Time je u potpunosti obrađen poziv `fakt(3)`.



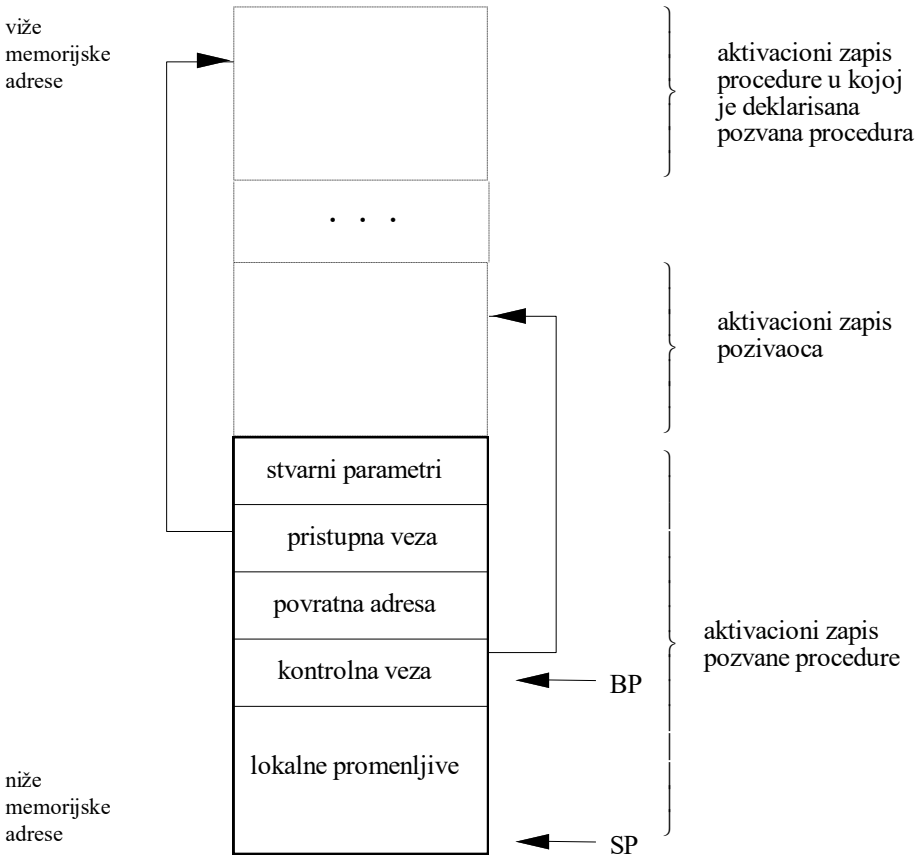
Statičko okruženje za nelokalne promenljive realizovano pristupnim vezama

Postoje programski jezici, jedan od klasičnih primera je Pascal, ali i neki noviji programski jezici kao što su objective C, koji je koristan za programiranje OSX aplikacija i takođe gnu C/C++ koji isto ima tu nestandardnu ekstenziju, da se dozvoljava ugneždavanje definicija procedura i funkcija. i pristup lokalnim promenljivama neke procedure/funkcije od strane neke druge procedure/funkcije.

Prema pravilu leksičkog opsega važenja deklaracije promenljivih, u nekoj Pascal-skoj proceduri, pored lokalnih promenljivih, vidljive su i sve promenljive deklarisane u okružujućim procedurama ako nemaju isto ime kao lokalne promenljive. Ugneždena funkcija može dakle da koristi i promenljive koje su lokalne recimo za spoljnu funkciju, to znači može da pristupi promenljivama koje nisu njene, a s druge strane nisu ni globalne. Takav pristup zovemo nelokalni pristup promenljivama. Znači mi na neki način moramo da obezbedimo u runtime-u pristup tim nelokalnim promenljivama. Način realizacije ovih jezičkih osobina naziva se statičko okruženje za nelokalne promenljive.

Izgled aktivacionog zapisa pogodnog za procedure na Pascal-u i srodnim jezicima dat je na slici koja sledi. Kao i u slučaju Ca, aktivacioni zapis pozivaoca je naznačen isprekidanim linijama. Punim linijama prikazan je aktivacioni zapis pozvane procedure. Tu se nalaze polja koja smo i ranije imali: stvarni parametri, povratna adresa, kontrolna veza, lokalne promenljive. U odnosu na aktivacioni zapis za C procedure, razlika je u polju *pristupne veze*.

Polje pristupne veze se definiše kao pokazivač na jedan aktivacioni zapis koji se već nalazi niže na steku, ali znamo da je i kontrolna veza u suštini pokazivač na aktivacioni zapis. Koja je razlika između kontrolne i pristupne veze? Dok kontrolna veza ukazuje na aktivacioni zapis onoga ko je u runtime-u neposredno pozvao tekuću proceduru, polje pristupne veze neke procedure ukazuje na aktivacioni zapis (preciznije, na polje kontrolne veze toga zapisa) *neposredno okružujuće* procedure u listingu programa. Ova procedura, u opštem slučaju, razlikuje se od procedure pozivaoca u vreme izvršavanja programa. Polje pristupne veze potrebno je da bi se realizovao pristup nelokalnim promenljivama koje su deklarisane u drugim procedurama.



Postavlja se pitanje kako ćemo formirati pristupnu vezu? Videli smo da u sekvenci pozivanja potprograma treba svako polje definisati, u njega upisati neku konkretnu vrednost. Kompajler mora generisati kod koji će postaviti prilikom poziva vrednost u to polje pristupne veze. Ako pogledamo strukturu zapisa sa prethodne slike, vidimo da se pristupna veza nalazi pre povratne adrese što nam govori da to polje mora da definiše pozivalac.

Algoritam formiranja pristupne veze

Razmotrimo tri moguća slučaja kada Pascal-ska procedura A poziva proceduru B:

i) Deklaracija B je ugneždena u deklaraciji A; kaže se da je B deklarisana na (za jedan) višem *leksičkom nivou ugneždavanja* od A.

```
procedure A;
    procedure B; begin...end;
begin B end.
```

U ovom slučaju, pristupna veza procedure B ukazivaće na aktivacioni zapis procedure A.

ii) Deklaracija B je ugneždena u istoj proceduri X (ili u glavnom programu) kao i deklaracija A; tada važi da su A i B na istom leksičkom nivou ugneždavanja.

```
procedure X;
    procedure B; begin...end;
    procedure A; begin B end;
begin ... end.
```

U ovom slučaju, pristupna veza A kopira se u polje pristupne veze B, tako da obe ukazuju na aktivacioni zapis procedure X.

iii) Leksički nivo ugneždavanja A je veći od nivoa B. Razlika leksičkih nivoa A i B označena sa N može da bude veća od jedan;

```
procedure X
  procedure B;
     procedure Y
            procedure A; begin B end;
    begin A; end;
  begin Y; end;
begin B; end;
```

u ovom slučaju potrebno je slediti lanac od pristupnih veza počev od pristupne veze A i u polje pristupne veze za B iskopirati sadržaj N-tog polja u lancu. Za N jednako 1, to je polje na koje ukazuje polje pristupne veze za A i tako dalje.

Sekvenca pozivanja

Kako izgleda sekvenca pozivanja u ovom slučaju ovih pristupnih veza? Ona je donekle različita u odnosu na sekvencu koja se primenjuje u jeziku C. Pri pozivu procedure, pozivalac radi sledeće:

- na stek redom stavlja stvarne parametre (uobičajeno za Pascal je da kranje levi parametar ide prvi)
- na stek stavlja vrednost koja definiše sadržaj polja pristupne veze pozvane procedure prema gore navedenim pravilima (na primer, u slučaju i, na stek se stavlja sadržaj registra BP)

- Izvršava se CALL naredba koja na stek stavlja povratnu adresu i prenosi kontrolu pozvanoj proceduri

Pozvana procedura:

- Na stek stavlja sadržaj registra BP (radi se o polju kontrolne veze).
- Sadržaj registra SP, koji u tom trenutku ukazuje na kontrolnu vezu, kopira u BP.
- Od sadržaja registra SP oduzima broj N koji je poznat u vreme prevođenja. Radi se o broju memorijskih lokacija potrebnih za smeštaj lokalnih promenljivih. Na ovaj način se na steku alocira potreban prostor za lokalne promenljive.

Nakon završetka sekvence pozivanja izvršava se kod pozvane procedure.

Sekvenca povratka

Prilikom povratka iz pozvane u pozivajuću proceduru izvršava se *sekvenca povratka*, koju u potpunosti realizuje pozvana procedura:

- Sadržaj registra BP kopira se u registar SP čime se sa steka oslobađa prostor za lokalne promenljive i one efektivno prestaju da postoje.
- Sa vrha steka (koji u tom trenutku ukazuje na polje kontrolne veze pozvane procedure) skida se vrednost kontrolne veze i smešta u registar BP. Time je ovaj registar postao spreman za upotrebu u proceduri-pozivaocu.
- Naredba RET *n* skida sa vrha steka povratnu adresu i smešta u brojač naredbi IP. Ova naredba zatim uvećava registar SP za vrednost *n* čime se sa steka skidaju stvarni parametri i pristupna veza. Nakon toga se kontrola predaje pozivaocu.

Primer

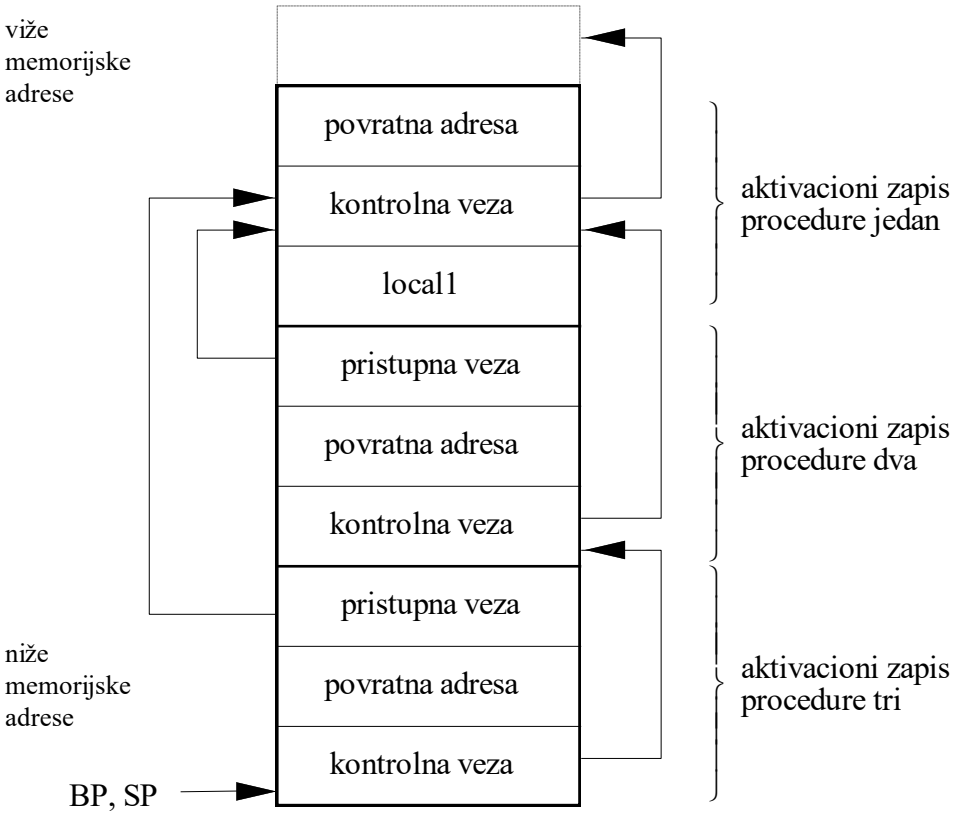
Da vidimo konkretan primer: odrediti 80x86 asemblerski ekvivalent datog Pascal programa. Kako izgleda procesorski stek u trenutku neposredno pre prvog povratka iz procedure?

```
program glavni;
    procedure jedan
    var local1: integer;
        procedure tri; forward;
        procedure dva;
        begin {dva}
            tri  {II slučaj kod formiranja pristupne veze}
        end; {dva}
        procedure tri;
        begin {tri}
            local1:=10;
        end; {tri}
    begin {jedan}
        dva {I slučaj kod formiranja pristupne veze}
    end; {jedan}
begin {glavni}
    jedan
end. {glavni}
```

Program se zove glavni, unutar njega imamo definisanu proceduru jedan koja ima celobrijnu promenljivu local1, a unutar procedure jedan, data je takozvana forward deklaracija, znači da će tek biti deklarisana procedura tri. Potom ide deklaracija procedure dva. Procedure dva i tri su obe unutar procedure jedan.

Kako u toku izvršavanja ide redosled poziva? Glavni program poziva proceduru jedan, procedura jedan poziva svoju ugneždenu proceduru dva. Procedura dva poziva proceduru tri koja je na istom nivou ugneždenja kao i dva. Procedura tri pristupa nelokalnoj promenljivoj local1 iz procedure jedan dodeljujući joj vrednost 10 i onda redom ide redosled povrataka, iz tri u dva, iz dva u jedan, iz jedan u glavni program.

Pre nego što pogledamo detalje generisanog koda možemo pogledati izgled steka na sledećoj slici. Kod Paskala glavnom programu ne treba neki poseban aktivacioni zapis. Postoji samo jedna lokacija koji izigrava neki degenerisani aktivacioni zapis, zato što promenljive glavnog programa idu u statički deo memorije, ne čuvaju se na steku.



Kada se iz glavnog programa pozove procedura jedan, njen aktivacioni zapis se formira. U ovom konkretnom primeru, polja koja taj zapis ima su povratna adresa, pristupna veza ne treba jer glavni program nema svoj aktivacioni zapis, tako da procedura koja je neposredno u glavnom programu nema pristupnu vezu, nego samo ima povratnu adresu, kontrolnu vezu i promenljivu local1. To je aktivacioni zapis procedure jedan.

Kada se iz procedure jedan pozove procedura dva, ona ima pristupnu vezu i ona ukazuje na jedan, pošto je dva u listingu ugneždena u jedan. Kod procedure jedan se u stvari poklapaju pristupna i kontrolna veza, što je samo koincidencija. Znači sa desne strane na slici su prikazane kontrolne veze, sa leve strane su prikazane pristupne. Procedura dva nema svoje lokalne promenljive.

Sada kada se iz procedure dva pozove procedura tri na vrhu steka se formira aktivacioni zapis procedure tri. Iskopira se pristupna veza procedure dva u pristupnu vezu procedure tri, jer one su na istom leksičkom nivou i obe ukazuju u stvari na aktivacioni zapis procedure jedan. Kontrolna veza procedure tri je prema onom redosledu pozivanja znači tri ukazuje na dva, dva ukazuje na jedan, jedan ukazuje na onaj rudimentarni aktivacioni zapis celog programa. Znači slika prikazuje izgled aktivacionih zapisa pre nego što se realizuje prvi povratak iz bilo koje procedure.

Ostalo je da razmotrimo mašinski kod koji se generiše. On se generiše prema redosledu kako se nailazi u listingu programa na izvršna tela pojedinih procedura. Ovde prvo se u listingu nailazi na izvršno telo procedure dva, koja poziva proceduru tri.

procedura dva

```
; sadržaj registra BP definise sadržaj polja kopntrolne veze
PUSH BP
; BP ukazuje na polje kontrolne veze
MOV BP,SP
; počinje sekvenca poziva procedure tri
; uspostavlja se sadržaj polja kontrolne veze u aktivacionom zapisu
; procedure tri,
; kopiranjem sadržaja polja kontrolne veze u akt. zapisu procedure dva
; (ii slučaj algoritma)
PUSH [BP+04]
; poziv procedure tri
CALL tri
; sekvenca povratka iz procedure dva
; dealocira se deo aktivacionog zapisa iza polja kontrolne veze
MOV SP,BP
; restaurira se sadržaj registra BP potreban pozivaocu i dealocira
; polje kontrolne veze sa steka
POP BP
; povratak pozivaocu uz dealociranje polja pristupne veze
RET 0002
```

procedura tri

```
; definisanje polja kontrolne veze
PUSH  BP
; BP ukazuje na polje kontrolne veze
MOV BP,SP
; prevod naredbe local1 := 10
MOV DI,[BP+04]
MOV [DI-02],10*
; sekvenca povratka
MOV SP,BP
POP BP
RET 0002
```

procedura jedan

```
; definisanje polja kontrolne veze
PUSH BP
; BP ukazuje na polje kontrolne veze
```

* Na 8086 asembleru sintaksa ove instrukcije glasi MOV SS:WORD PTR [DI-2],10
SS: označava da se pristupa segmentu steka (podrazumevana vrednost za registar DI je segment podataka DS),
WORD PTR označava da se prenosti jedna reč, a ne jedan bajt.

```
MOV BP,SP
; alociranje prostora za promenljivu local1
SUB SP,+02
; definisanja pristupne veze za proceduru dva (slučaj i)
PUSH BP
; poziv procedure dva
CALL dva
; sekvenca povratka
MOV SP,BP
POP BP
RET
```

glavni program - sekvenca poziva

```
; polje pristupne veze nije potrebno proceduri jedan pošto su
; promenljive glavnog programa statički alocirane
CALL    jedan
```

Kod je detaljno iskomentarisan. Najzanimljiviji je deo programa koji upisuje vrednost 10 u nelokalnu promenljivu local1 u proceduri tri ima sledeći izgled:

```
MOV DI,[BP+4]
MOV [DI-2], 10*
```

Prva naredba dodeljuje registru DI sadržaj polja pristupne veze, odnosno adresu polja kontrolne veze u aktivacionom zapisu procedure jedan. Sada se u drugoj naredbi promenljivoj local1 pristupa indeksnim adresiranjem uz korišćenje registra DI i pomeraja poznatog u vreme prevođenja programa.

Kod C-a se problem pristupa nelokalnim promenljivama ne javlja, jer su sve promenljive kojima se pristupa u proceduri ili lokalne (kada im se pristupa baznim adresiranjem uz korišenje registra BP) ili su globalne, statički alocirane (pa su ima adrese poznate u vreme prevođenja programa).

Postvalja se pitanje, da li je pristupna veza neophodna ili se za pristup nelokalnim promenljivama mogu koristiti kontrolne veze? U cilju nalaženja odgovora osmatrajmo opet program iz tačke a). U slučaju korišćenja kontrolnih veza, naredba dodele u proceduri tri mogla bi se realizovati na sledeći način:

```
; u DI ide sadržaj polja kontrolne veze aktivacionog zapisa procedure tri
MOV DI,[BP]
; u DI ide sadržaj  polja kontrolne veze aktivacionog zapisa procedure dva
MOV DI,[DI]
; DI ukazuje na kontrolnu vezu procedure jedan pa se uz poznati
; pomeraj pristupa promenljivoj local1
MOV [DI-2],10
```

Razmotrimo, međutim slučaj koji bi nastao da procedura jedan pozove proceduru tri posle završetka procedure dva. Pošto je procedura dva završena, sa steka su uklonjeni aktivacioni zapisi procedura jedan i tri pa je na steku ostao samo aktivacioni zapis procedure jedan. Pri ponovnom pozivu procedure tri, na vrh steka se stavlja aktivacioni zapis za ovu novu aktivaciju procedure tri. Sada bi naredba dodele morala biti realizovana na sledeći način uz korišćenje kontrolnih veza:

```
; u DI ide sadržaj polja kontrolne veze procedure tri
MOV DI,[BP]
; DI ukazuje na kontrolnu vezu procedure jedan pa se uz
; poznati pomeraj pristupa promenljivoj local1
MOV [DI-2],10
```

Prema tome, potreban je jedan nivo indirekcije manje jer se aktivacioni zapis procedure jedan sada nalazi na steku neposredno ispod aktivacionog zapisa procedure tri. Drugim rečima, naredba dodele u proceduri tri se različito prevodi u zavisnosti sa kog mesta je pozvana procedura tri što je nedopustivo. Pristupna veza je, u skladu sa tim, neophodna.

Statičko okruženje realizovano displejima

U opštem slučaju, ako se promenljivoj deklarisanoj na leksičkom nivou M pristupa iz procedure deklarisane na nivou N kao nelokalnoj promenljivoj (te je N > M), potrebno je slediti lanac od N-M pristupnih veza da bi se došlo do aktivacionog zapisa u kome se nalazi promenljiva.

Efikasniji pristup nelokalnim promenljivama realizuje se korišćenjem *displeja*. Za svaki leksički nivo na kome se može nalaziti nelokalna promenljiva čuva se poseban pokazivač na aktivacioni zapis odgovarajuće procedure deklarisane na tom nivou; niz tih pokazivača naziva se displej.

Za pristup nelokalnoj promenljivoj putem displeja uvek su dovoljne dve naredbe, bez obzira na kojem se leksičkom nivou nalazi nelokalna promenljiva:

- Pva naredba u registar dovodi sadržaj odgovarajueg pokazivača koji predstavlja adresu aktivacionog zapisa u kome se nalazi nelokalna promenljiva.
- Druga naredba vrši pristup koristeći bazno indeksiranje sa poznatim pomerajem.

Gde se čuva ceo taj niz displeja? To zavisi od realizacije. Ako ima dovoljno registara procesora, najefikasnije ih je čuvati u registrima procesora. Ako ne, onda u statičkoj zoni memorije, ili čak u aktivacionom zapisu procedure. Ovu treću varijantu realizuje x86 procesor, ako koristimo posebne naredbe za hardversku podršku za sekvence pozivanja potprograma, kao što ćemo ubrzo videti.

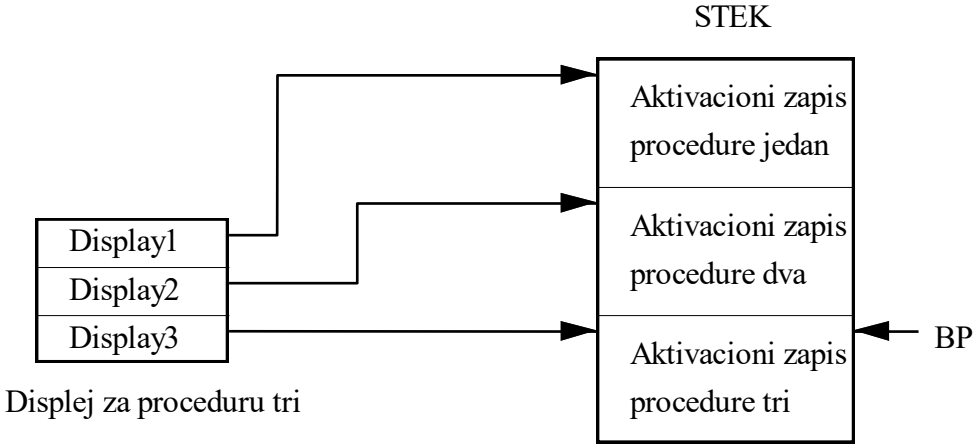
Primer korišćenja displeja

Ponovićemo primer vrlo sličan prethodnom na Paskalu. Opet imamo glavni program i procedure jedan, dva i tri. Procedura jedan, kao i u ranim primeru, ima lokalnu promenljivu local1. Iz glavnog programa se poziva procedura jedan. Iz procedure jedan se poziva procedura dva. Iz procedure dva se poziva procedura tri. U proceduri tri se onda pristupa toj promenljivoj local1 kao nelokalnoj. Sve je to isto kao u ranijem primeru, jedina razlika je što sad procedura tri nije na istom nivou kao dva, nego je ugneždena u dva.

```
program glavni;
    procedure jedan;
    var local1: integer;
        procedure dva;
            procedure tri;
            begin {tri}
                local1:=10
            end; {tri}
        begin {dva}
            tri
        end; {dva}
    begin {jedan}
        dva
    end; {jedan}
begin {glavni}
    jedan
```

```
end. {glavni}
```

Druga razlika je što ćemo umesto pristupnih veza koristiti koncept displeja. Sledeća slika prikazuje okruženje koje se formira u toku izvršavanja ovog programa pre prvog povratka iz procedure. Praktično bismo imali displej od tri elementa, ne računamo glavni program. Znači procedura jedan je na leksičkom nivou 1, procedura dva je na leksičkom nivou 2, a procedura tri je na leskičkom nivou 3. U trenutku izvršavanja procedure tri imamo izgled kaon a slici, znači na steku su ta tri aktivaciona zapisa za jedan, dva i tri, a u nizu displeja koristimo tri elementa. Prvi element pokazuje na aktivacioni zapis procedure jedan, drugi pokazuje na aktivacioni zapis procedure dva, jer ona je okružujuća za tri i konačno treći element pokazuje na tekući aktivacioni zapis same procedure tri, kao što i BP ukazuje. Znači mi možemo lokalnim promenljivama za tri dok izvršavamo tu proceduru pristupiti ili preko BP ili preko displeja tri.



Kako izgleda prevedeni kod? Ne moramo sve gledati, samo da vidimo kako izgleda u proceduri tri prevod naredbe dodele local1 = 10. Znači to je nelokalna promenljiva za proceduru tri, a ona je lokalna za proceduru jedan. Treba iskoristiti prvi element displeja, dovesti ga u neki registar i onda preko poznatog pomeraja doći do local1 promenljive i u nju upisati 10.

```
MOV DI, Display1 ; u SI ide adresa aktivacionog zapisa procedure jedan
MOV [DI-4], 10   ; u promenljivu local1 se upisuje 10
                 ; (pomeraj se zna u vreme prevođenja)
```

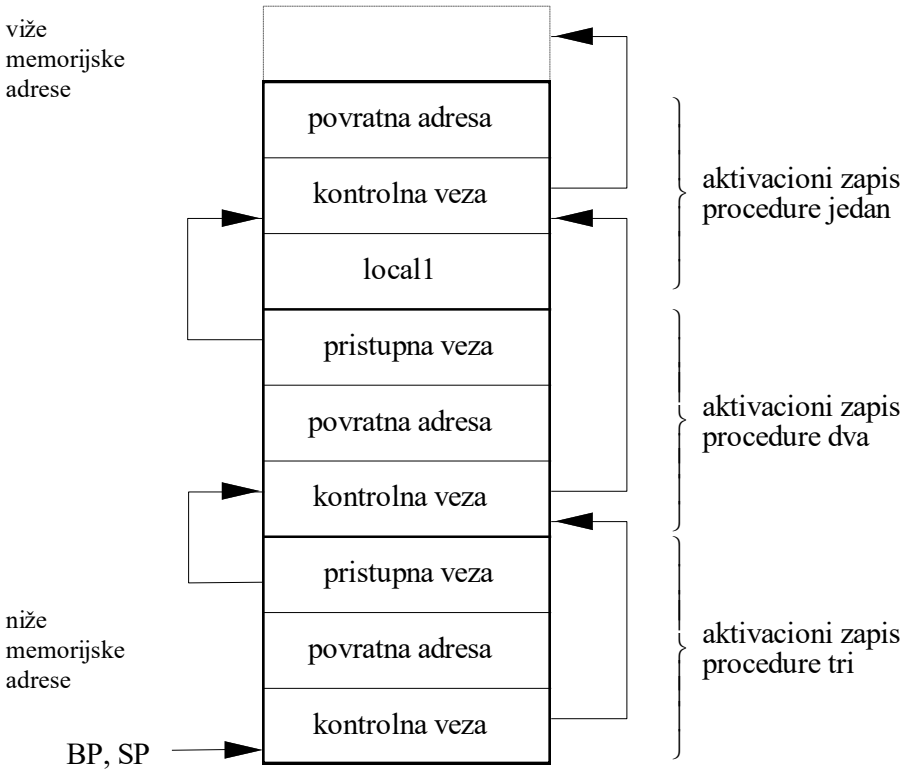
Ako bismo koristili pristupne veze kao ranije, ovde je sad situacija drukčija utoliko što je tri ugneždeno u dva, a dva ugneždeno u jedan, znači moramo slediti lanac pristupnih veza. Sekvenca asemblerskih instrukcija koja odgovara naredbi local1:=10 bila bi:

```
    ; u DI ide sadržaj polja pristupne veze procedure tri
    MOV DI,[BP+4]
    ; u DI ide sadržaj polja pristupne veze procedure dva
    MOV DI,[DI+4]
    ; DI ukazuje na kontrolnu vezu procedure jedan pa se uz poznati
    ; pomeraj pristupa promenljivoj local1
    MOV [DI-2],10
```

Prva asemblerska naredba uzima u DI sadržaj pristupne veze procedure tri, onda moramo koristeći tu pristupnu vezu da pristupimo aktivacionom zapisu procedure dva i opet preko poznatog pomeraja da dođemo do pristupne veze procedure dva, a ona tek ukazuje na jedan i onda tek

koristeći taj DI upisati u lokalnu promenljivu vrednost 10. U slučaju da je razlika leksičkih nivoa veća od 2, u gornju sekvencu bio bi dodat odgovarajući broj instrukcija `MOV DI,[DI+4]`.

Izgled steka za ovu varijantu u trenutku kada se izvršava procedura tri dat je na sledećoj slici. Prednost displeja u odnosu na lanac pristupnih veza je u efikasnijem pristupu nelokalnim promenljivama; nedostatak je u potencijalno povećanoj režiji u sekvenci pozivanja procedure, kada je potrebno definisati sadržaj displeja.



Algoritam za postavljanje displeja

Pri svakom pozivu i povratku ažuriraju se registri displeja da odgovaraju trenutno aktivnoj proceduri (onoj čiji je aktivacioni zapis na vrhu steka). Procedura na nivou ugneždavanja N koristi N registara displeja. U svakoj situaciji dovoljno je ažurirati jedan registar displeja (to je onaj koji odgovara nivou ugeždavanja N na kome se nalazi pozvana procedura P):

- Pri pozivu, stari sadržaj displej registra N čuva se u aktivacionom zapisu procedure P.
- U displej registar N stavlja se pokazivač na upravo formirani aktivacioni zapis procedure P.
- Pri povratku iz P, sačuvana vrednost displej registra N restaurira se.

Procesorska podrška višim programskim jezicima

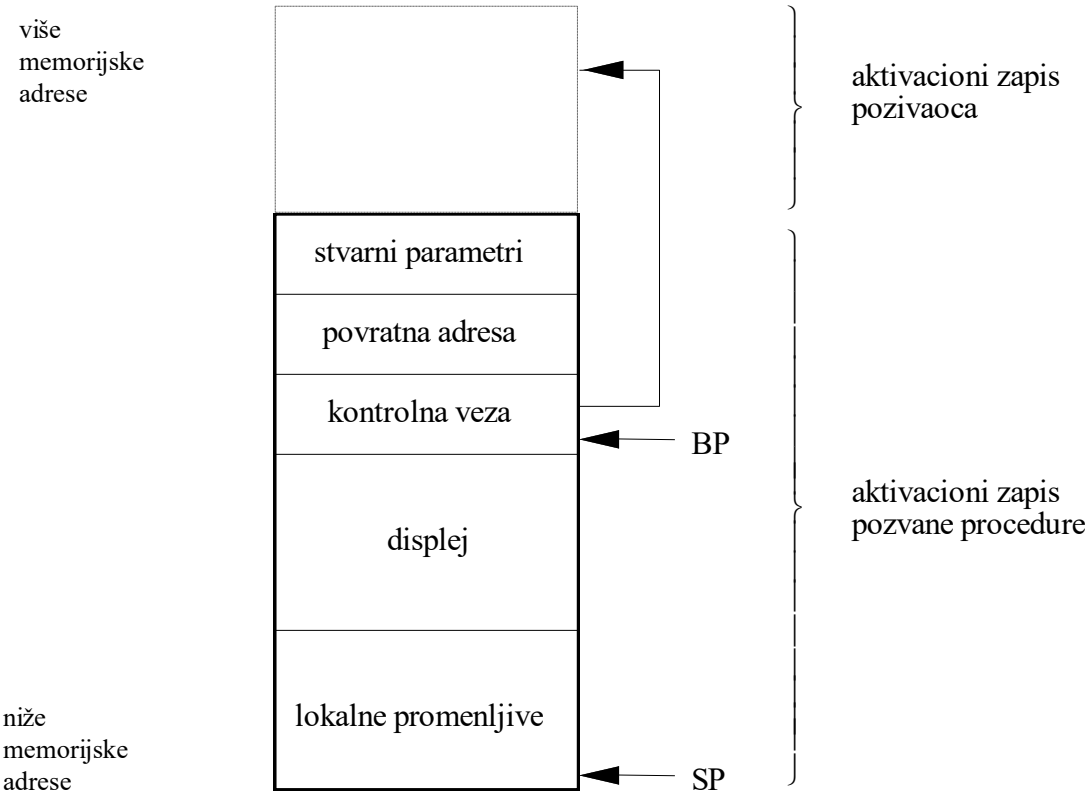
Naslednici procesora 8086 imaju dve mašinske instrukcije za podršku izvršnoj organizaciji memorije kod viših programskih jezika:

- ENTER Size, Level
- LEAVE

Naredba enter daje podršku za deo sekvence pozivanja koji radi pozvana procedura, a leave odgovara podrški za deo sekvence povratka koji vrši pozvana procedura i ovo u suštini znatno uprošćava kod procedure u tim standardnim delovima koje imaju sve procedure na početku koda i na kraju. Uvek ćete u asemblerskom kodu na početku videti enter, a na kraju, neposredno pre return, instrukciju leave.

Struktura aktivacionog zapisa za ENTER i LEAVE

Na sledećoj slici prikazan je izgled aktivacionog zapisa koji odgovara ovim naredbama. U zapisu imamo stvarne parametre, povratnu adresu, kontrolnu vezu. Zatim je iskorišćen deo aktivacionog zapisa posle kontrolne veze za smeštaj niza displej pokazivača i potom idu lokalne promenljive. Naravno, kod jezika kao što je C kome ne treba displej, ovo se ne alocira i ne koristi, ali kod jezika kao što je Pascal ovo se koristi. Naredba enter podržava obe te varijante zavisno kakve joj parametre zadamo.



Naredba ENTER Size, Level

Naredba ENTER realizuje deo sekvence pozivanja koji izvršava pozvana procedura. Ova naredba ima dva operanda, *Size* i *Level*. *Size* (16-bitna veličina) definiše koliko memorije treba alocirati u aktivacionom zapisu za lokalne promenljive. *Level* (8-bitna veličina) predstavlja leksički nivo pozvane procedure.

Ova naredba izvršava se kao prva naredba pozvane procedure na sledeći način:

1. Sadržaj registra BP ide na stek (čime se definiše polje kontrolne veze)
2. U registar BP upisuje se sadržaj registra SP (čime je BP postavljen da ukazuje na polje kontrolne veze).
3. Ako je Level = 0 preskače se sledeći korak (Level = 0 važi za procedure u C-u kod koga se ne javlja problem pristupa nelokalnim promenljivim).
4. Označimo vrednost Level -1 sa N, a sadržaj polja kontrolne veze sa A. Na stek se smešta N reči koje se kopiraju redom počev od lokacije A-2 ka nižim memorijskim lokacijama. Potom se na stek smešta adresa polja kontrolne veze. Ovime je uspostavljen displej pozvane procedure.
5. Registar SP umanjuje se za vrednost operanda Size čime se alocira prostor za lokalne promenljive pozvane procedure.

Naredba LEAVE

Naredba LEAVE realizuje deo sekvence povratka koji izvršava pozvana procedura. Ova naredba neposredno prethodi naredbi RET i izvršava se na sledeći način:

1. Sadržaj registra BP kopira se u registar SP. Ovime se dealocira sa steka deo aktivacionog zapisa pozivaoca ispod kontrolne veze.
2. Sa steka se skida sadržaj polja kontrolne veze i smešta u registar BP. Time je BP namešten za pozivaoca, a polje kontrolne veze pozvane procedure je uklonjeno sa steka.

Znači naredba LEAVE u stvari menja one dve naredbe mov SP, BP i pop BP. Posle leave ide naravno naredba return.

Primer

Evo na konkretnom primeru da vidimo kakav se kod generiše kada se koriste naredbe enter i leave. Primer je potpuno isti kao prethodni sa displejima. Znači imamo glavni program i procedure jedan, dva i tri. Glavni poziva jedan, jedan dva, dva poziva tri. Tri pristupa ovoj promenljivoj koja je lokalna u proceduri jedan. Tamo smo videli konceptualni primer sa displejima. Ovde sada gledamo konkretan kod kada se koriste x86 naredbe enter i leave.

```
program glavni;
    procedure jedan;
    var local1: integer;
        procedure dva;
            procedure tri;
            begin {tri}
                local1:=10
            end; {tri}
        begin {dva}
            tri
        end; {dva}
    begin {jedan}
        dva
    end; {jedan}
begin {glavni}
    jedan
end. {glavni}
```

Za dati Pascal program, asemblerski ekvivalent uz korišćenje naredbi ENTER i LEAVE bi imao sledeći izgled:

procedura dva

```
; nema lokalnih promenljivih a leksički nivo je jednak 2
ENTER 0,2
; poziv procedure tri
CALL     tri
; sekvenca povratka iz procedure dva
; dealocira se deo aktivacionog zapisa iza polja kontrolne veze
LEAVE
; povratak pozivaocu uz dealociranje polja pristupne veze
; nema stvarnih parametara za dealociranje pa je upotrebljena
; naredba RET bez argumenta
RET
```

procedura tri

```
; nema lokalnih promenljivih a leksički nivo je jednak 3
ENTER 0,3
; prevod naredbe local1 := 10
; u DI se kopira pokazivač koji odgovara leksičkom nivou 1
MOV DI,[BP-02]
; u kativacionom zapisu procedure jedan displej ima samo jedan pokazivač
; promenljiva local1 nalazi se odmah ispod toga pokazivača
MOV [DI-04],10
; sekvenca povratka
LEAVE
RET
```

procedura jedan

```
; rezervišu se dva bajta za promenljivu local1
; leksički nivo je jednak 1
ENTER 2,1
; poziv procedure dva
CALL    dva
; sekvenca povratka
LEAVE
RET
```

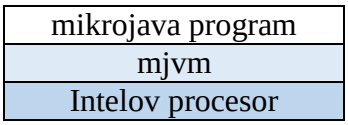
glavni program - sekvenca poziva

```
CALL    jedan
```

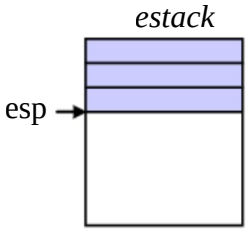
Prvo ide kod procedure dva, pripremna naredba je enter 0, 2. Procedura dva nema lokalnih promenljivih, znači nula bajtova se koristi u aktivacionom zapisu za njih. Ona je na leksičkom nivou dva. Zatim ide poziv procedure tri. Posle toga ide nareba leave i naredba return. Procedura tri takođe nema lokalnih promenljivih i nalazi se leksičkom nivou 3. Zatim ide prevod naredbe local1 = 10, gde smo videli da je u varijanti sa displejima potrebna samo jedna pripremna naredba da dovedemo odgovarajući displej. Sada se displeji nalaze na steku, tako da BP-02 odgovara tom displeju na leksičkom nivou 1. Sada, DI nam ukazuje na aktivacioni zapis jedan, onda DI- 4 u ovom slučaju upisuje u local1 konstantu 10 i imamo leave i ret. Konačno procedura jedan ima svoju lokalnu promenljivu, tako da dva bajta na steku treba opredeliti za to i nalazi se na leksičkom nivou 1. Ona poziva proceduru dva i imamo samo posle sekvencu povratka leave i ret. Glavni programima ima samo poziv procedure jedan.

Arhitektura mikrojava virtuelne mašine

U ovom poglavlju predstavljamo jedan konkretan primer izvršnog okruženja koje se koristi u projektu mikrojave, dajemo osnovne podatke o mikrojava virtuelnoj mašini (skraćeno MJVM). U specifikaciji mikrojave u dodatku knjige postoji odeljak o mikrojava virtuelnoj mašini, gde su dati dodatni detalji. Šta je generalno virtuelna mašina? Možemo je shvatiti kao jedan procesor, koji nije implementiran u hardveru, nego u softveru, što znači da bi se program pisan za mikrojava virtuelnu mašinu mogao izvršavati, mora postojati sloj softvera, takozvani runtime koji interpretira mikrojava instrukcije na realnom Intelovom procesoru.

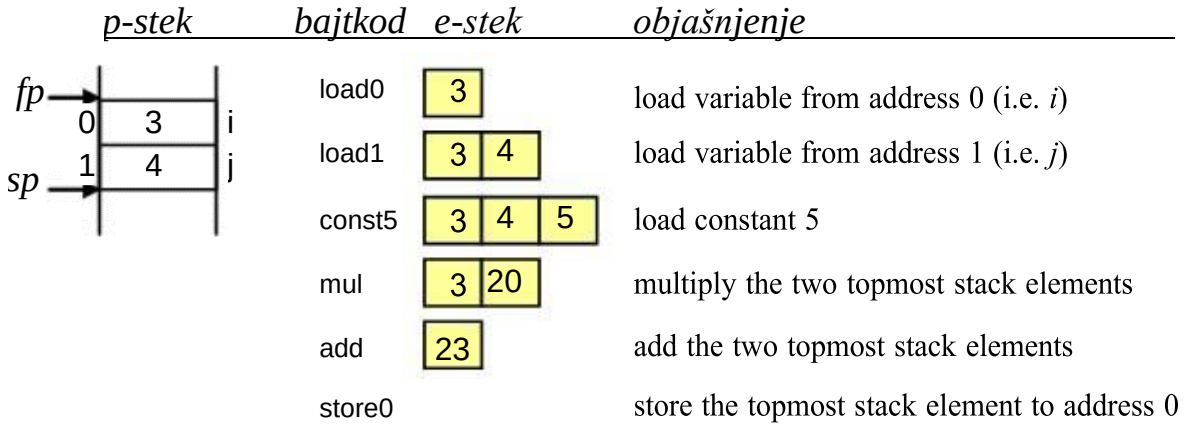


Taj koncept se puno koristi u savremenom softveru, znači primeri realnih virtuelnih mašina su Java virtuelna mašina, Microsoft-ov Common Language Runtime u dotnet okruženju. Jedna od prvih virtuelnih mašina istorijski bio je paskalski p-kod. Mikrojava virtuelna mašina osmišljena je po ugledu na pravu Java virtuelnu mašinu, ali je znatno uprošćena, oslobođena raznih detalja koji su potrebni Javi kao jeziku i koji bi komplikovali koncept. Mikrojava virtuelna mašina je stek mašina, ili nula adresna mašina, a to znači da ona nema procesorskih registara, nego ulogu registara preuzima takozvani stek izraza ili expression stack (e-stack).



To je struktura podataka last in first out, koja je realizovana preko jednog niza konačne veličine. Pošto je osnovna memorijska jedinica u mikrojava virtuelnoj mašini jedna memorijska reč veličine četiri bajta, kod mikrojava virtuelne mašine sve adrese se izražavaju na nivou memorijske reči. Na primer, e-stek može biti 32 registra i interno postoji ESP znači vrh tog steka izraza, koji služi za kontrolu steka. Kako radi stek mašina, evo na jednom konkretnom primeru kako izgleda prevod naredbe dodele i = i + j * 5, u mikrojava bajtkodu, ako su i i j lokalne promenljive, to znači da će one biti smeštene u okviru (engl. frame) tekuće procedure. Postoji još jedan stek u mikrojava virtuelnoj mašini, takozvani procesorski ili programski stek. p-stek, i na njemu jedan deo prostora se odvaja za okvir, znači vrh programskog steka na njega ukazuje stek pointer, a za pristup lokalnim promenljivima se koristi se još jedan interni registar mikrojava virtuelne mašine, pod nazivom frejm pointer i onda se lokalne promenljive čuvaju na pomerajima koji su relativni u odnosu na frejm pointer. Recimo i može biti prva lokalna promenljiva, znači na pomeraju 0, ovde

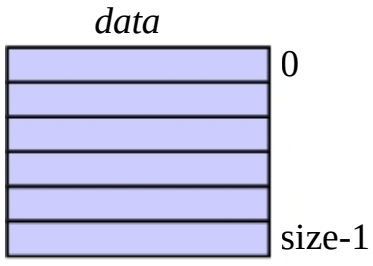
je prikazana sa konkretnom vrednošću 3 a j sledeća lokalna promenljiva, pošto svaka zauzima fiksno po jednu memorijsku ćeliju, to je pomeraj 1 i na primer ima vrednost 4. Kako glasi prevod ovog iskaza na jezik mikrojava virtuelne mašine, to jest na mikrojava bajtkod?



Prva instrukcija je load 0, to je instrukcija koja čita lokalnu promenljivu 0 i stavlja na stek izraza. Zatim instrukcija load1 čita drugu lokalnu promenljivu, to je j, i stavlja na stek izraza. Ako gledamo sadržaj steka na njega je prvo išla vrednost 3, zatim iznad toga stavljamo vrednost 4. Zatim instrukcija const stavlja određenu konstantu koja je zadata na expression stack znači konstantu 5. Posle toga aritmetička instrukcija množenja, nisu joj potrebni operandi, jer ona svoje operande uzima sa vrha e-steka, pomnoži ih i rezultat opet stavlja na e-stek. Posle toga instrukcija add isto to uradi, znači uzme dva operanda s vrha e-steka, sabere ih i rezultat stavi na e-stack. Finalna instrukcija store0 radi obrnuto od load, znači uzima vrednost sa e-steka i smešta u lokalnu promenljivu 0, znači ažuriramo i, a na kraju iskaza, kao što je bilo i na početku, e-stek je prazan.

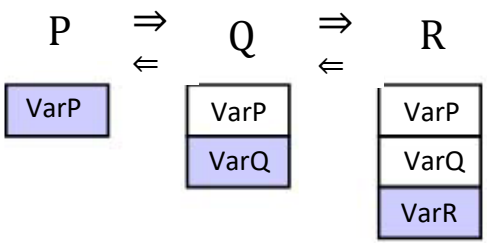
Memorijske oblasti MJVM

Mikrojava virtuelna mašina ima nekoliko oblasti podataka. To su stek izraza ili e-stek, programski stek ili p-stek, zatim promenljive glavnog programa čuvaju se u oblasti globalnih podataka. To je implementirano kao jedan niz reči u mikrojava virtuelnoj mašini. Osnovna implementacija mikrojava virtuelne mašine je interpreter mikrojava bajtkoda i napisan u Javi, tako da mu je naravno potreban Java runtime. Vektor data je oblast fiksne veličine. Globalne promenljive postoje tokom čitavog izvršavanja programa. Svaka promenljiva zauzima po jednu memorijsku reč i adresa globalne promenljive je u stvari indeks u ovom data nizu.



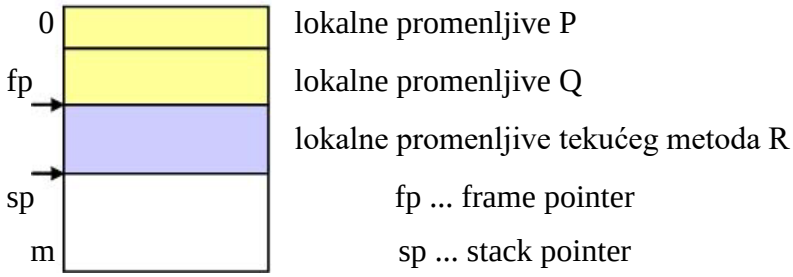
Kao što imamo load naredbu bajtkoda za lokalne promenljive, tako za globalne promenljive imamo recimo getstatic 2, to znači učitati globalni podatak na adresi 2 na e-stek. Znači ovo je

potpuno analogno load, samo što ne učitava lokalnu promenljivu, nego učitava globalnu promenljivu na e-stek.



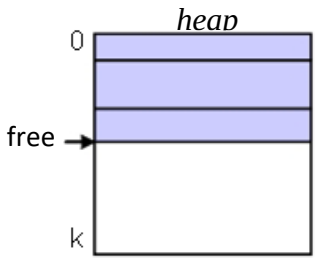
Zatim postoji oblast lokalnih promenljivih na p-steku. Svaki poziv metoda dobija sopstveni okvir na steku. Oni se pri pozivu metoda stavljaju na p-stek, a skidaju sa p-steka prilikom povratka iz odgovarajućeg metoda. Ovde je to prikazano, ako metod P, kada njega glavni program pozove, na vrhu steka se nalaze njegove lokalne promenljive, ako P pozove metod Q iznad okvira za P se formira okvir za Q u kome se nalaze lokalne promenljive za Q. Ako se u toku poziva Q pozove neki R, dalje se znači formira okvir za R na vrhu steka i onda sva ta tri okvira, do god pozivi nisu su završeni, egzistiraju. Kada se završi izvršavanje metoda R, sklanja se njegov okvir sa steka. Kada se vratimo u Q situacija je opet ova. Kada se iz Q vratimo u P, situacija je opet kao na slici.

Drugim rečima, ovde je to prikazano sa malo više detalja, ako je p-stek implementiran kao jedan niz memorijskih reči u Javi, sa indeksima 0 do M, postoji stek pointer, interni registar mikrojava vrituelne mašine, koji naravno nije vidljiv iz bajtkoda direktno, nego se koristi u interpreteru. Stek pointer ukazuje dokle smo popunili ovaj niz, a frejm pointer je druga granica tekućeg okvira. Lokalne promenljive se adresiraju relativno u odnosu na frejm pointer, svaka promenljiva, to smo već ranije rekli, zauzima fiksno po jednu reč, da bismo pojednostavili rad i generisanje koda. Videćemo kasnije skup instrukcija, šta sve možemo da radimo sa lokalnim promenljivim.



Sledeća oblast podataka mikrojava virtuelne mašine je oblast dinamičkih podataka -- heap. To je takođe implementirano kao jedan poseban niz i takođe postoji interni pointer free, da znamo dokle smo popunili heap.

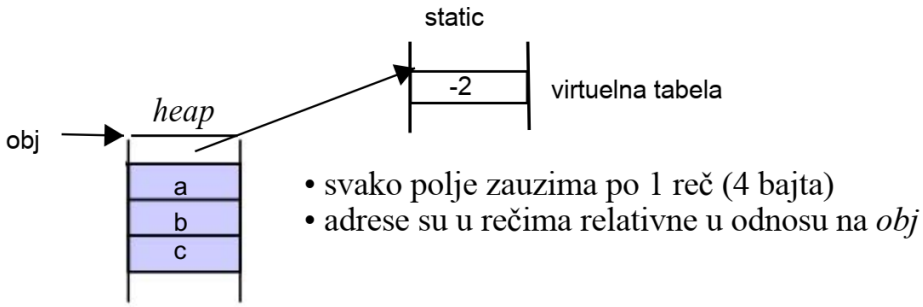
Heap se koristi pri instanciranju novih objekata neke klase, znači kada se u program upotrebi operator new i za instanciranje nizova, slično kao i u Javi. U ovoj osnovnoj mjvm koja je javno dostupna na sajtu kursa, jednom alocirani objekti se ne dealociraju, znači niti ima eksplicitan dealokator, niti ima sakupljač smeća, kao što postoji u pravoj Javi.



Jedinica alokacije je i dalje je jedna memorijska reč i adrese su na nivou memorijske reči. Na sledećoj slici se ilustruje rad sa heap-om, na primer ako imamo neku klasu i obj je tipa X i alociramo novi objekat, znači instancu X. Naravno promenljiva obj će negde drugde da se nalazi, a ne na heap-u. to je ili lokalna ili globalna promenljiva, ovde nije prikazan taj detalj. Međutim, na heap-u će se ovim operaterom new, alocirati dovoljan prostor, znači za svako polje po jedna memorijska reč i dodatna memorijska reč kao pokazivač na statičku (data) oblast gde se formira virtuelna tabela klase X. Pošto nema nijedan metod, virtuelna tabela je prazna, ima samo onaj kraj virtuelne tabele -2.

Objekat klase

```
class X {
   int a, b;
   char c;
}
X obj = new X;
```



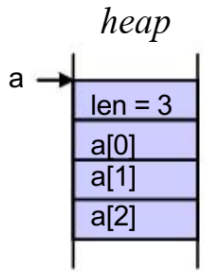
Ako je u pitanju niz, imamo dva tipa nizova: niz celih brojeva i znakovni niz. Ima male razlike u alokaciji, znači kada se sa new alocira recimo celobrojni niz veličine 3, to znači da će se na heap-u u stvari 4 memorijske lokacije alocirati, jer na adresi a, znači prva memorijska ćelija, iskoristiće se da se zapamti broj elemenata niza i onda tek od adrese a+1 pamtiće a[0].

Znači indeksi idu kao u Javi, ali prva ćelija je rezervisana za dužinu niza, tako da ima još tri ćelije za elemente niza i kod celobrojnih nizova svaki element zauzima tačno po jednu memorijsku reč. Što se tiče znakovnih nizova, prva memorijska ćelija na ovoj adresi koju vrati operator new, ta ćelija na adresi c isto se koristi da zapamti broj elemenata niza (5 u primeru). Međutim, kod znakovnih nizova koristi se pakovanje, tako da u jednoj memorijskoj reči mogu da stanu četiri znaka.

Tako da je recimo nizu dužine 5, potrebno samo dve memorijske reči plus jedna za dužinu niza. U drugoj memorijskoj reči su četiri znaka, a u trećoj memorijskoj reči se pamti još onaj jedan, peti znak.

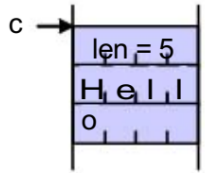
Niz

```
int[] a;
a = new int[3];
```



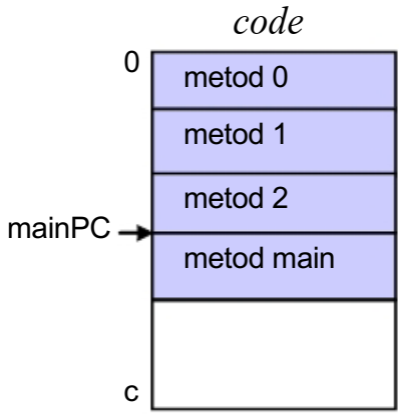
- dužina niza se pamti na početku
- svaki element zauzima po 1 reč

```
char[] c = new char[5];
```



- *char* nizovi su nizovi bajtova
- alociranje prostora je na nivou reči

Oblast programskog koda je u MJVM realizovana vektorom code. Kod se jedini u bajtovima upisuje i naknadno koristi. Metodi su smešteni redom, kako je ih je kompajler preveo, jedan za drugim, a promenljiva MainPC ukazuje na početak programa, odnosno gde je preveden main metod iz glavnog programa. Postoji naravno kao interni registar i program counter, kada MJVM interpretira program, da zna koja je tekuća instrukcija bajtkoda.



Pregled skupa instrukcija MJVM

Sledi kratak pregled i objašnjenje skupa instrukcija mikrojava virtuelne mašine. Inspiracija za bajtkod mikrojava virtuelne mašine je bila prava Java virtuelna mašina i instrukcije MJVM bajtkoda blisko prate instrukcije pravog Java bajtkoda, uz određeno uprošćenje, recimo kod pravog Java bajtkoda unutar instrukcije pamti se tip operanda sa kojima instrukcija radi (u donjem primeru celobrojni ili u pokretnom zarezu), jer Java ima verifikator bajtkoda, koji može da proveri integritet programa pre izvršavanja, dok je mikrojava u tom smislu jednostavnija, ne kodira se tip operanda kao deo instrukcije, ali van toga instrukcije imaju istu funkcionalnost.

Mikrojava	Java	
load0	iload0	fload0
load1	iload1	fload1
add	iadd	fadd

Mikrojava program je niz instrukcija, a svaka instrukcija počinje operacionim kodom (1 bajt), a iza toga može biti promenljiv broj operanada (0, 1 ili 2) pri čemo operandi mogu imati različit broj bajtova 1, 2 ili 4. Postoje možda izuzeci od toga, recimo za poziv dinamičkih metoda klasa, znači realizaciju polimorfnih poziva, taj metod može da zauzima i više bajtova.

Code = {Instruction}.	opcode ... 1 bajt
Instruction = opcode {operand}	operand ... 1, 2 ili 4 bajta

Primeri nekih instrukcija:

- add instrukcija sa 0 operanada, videli smo da većina aritmetičkih instrukcija samo ima opkod, zato što sve operande i rezultat smešta na e-stek.
- load 7 instrukcija sa jednim operandom, koja dohvata lokalnu promenljivu na adresi 7 i stavlja na e-stek.
- enter 0,2 ima dva operanda i sličnu semantiku kao Intel-ova enter instrukcija, znači za podršku pozivanju potprograma.

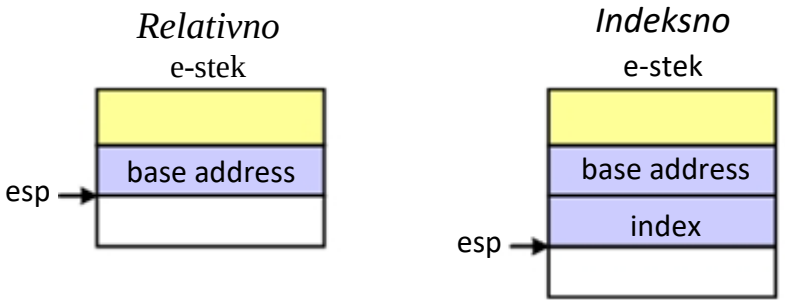
Što se tiče načina adresiranja postoje različiti naćini adresiranja kojima se može pristupiti operandima.

način adresiranja	***primer instrukcije***	***objašnjenje***
Neposredno	const 7	za konstante
Lokalno	load 3	za lokalne promenljive na *psteku*
Statičko	getstatic 3	za globalne promenljive u *data* oblasti
Stek	add	za vrednosti na *esteku*
Relativno	getfield 3	za polja objekata (load heap[pop() + 3])
Indeksno	aload	za elemente nizova (load heap[pop()+1+pop()]

Neposebno adresiranje je za konstante i vrednost konstante ugrađena je u sam bajtkod, znači postoji posebna instrukcija recimo const 7, koja služi tome da konstantu 7 stavi na vrh e-steka. Lokalnim promenljivama se pristupa posebnim instrukcijama, znači da se dohvati vrednost sa p-steka iz tekućeg okvira, zatim postoji statičko adresiranje da se dohvate globalne promenljive, statički alocirane. Onda postoji stek adresiranje, znači to je na neki način podrazumevano ako nema operanada, onda se barata sa e-stekom.

Relativno adresiranje je za polja objekata; tipično se na e-stek mora dovesti početna adresa zapisa, a onda se u okviru instrukcije dodaje relativni pomeraj, da bi se onda cela adresa izračunala kao zbir, onoga što je na vrhu steka plus relativnog pomeraja. Znači to je za pristup poljima objekta (instance klase).

Indeksno adresiranje je za nizove. Kod te vrste adresiranja se obično na e-stek stavljaju i početna adresa niza i za indeks i, pošto smo rekli da je na samom početku niza broj elemenata, onda zbir ta dva podatka sa vrha steka plus jedan daje adresu gde se nalazi i-ti element niza.



Sledi pregled celog instrukcijskog seta. Ovo takođe može još i kompletnije da se nađe u mikrojava specifikaciji. Prvo ćemo objasniti format prikaza instrukcija i šta znače pojedine kolone. Prva kolona služi da se definiše mnemonik instrukcije i ako instrukcija ima operand, onda će ovde odvojeno pisati operand i biće upotrebljeno b, s ili w slovo da označi u koliko bajtova se pamti, u okviru instrukcijskog koda operand. Znači b je 1 bajt, s je short, dva bajta i w (word) četiri bajta. Recimo, instrukcija load zahteva operand veličine jednog bajta, što znači da može biti do 256 lokalnih promenljivih.

Instrukcije za load i store lokalnih promenljivih

opcode	***instr.***	*opds*	*ExprStack*	*značenje*
1	**load**	b	… …, val	Load push(local[b]);
2..5	**load_n**		… …, val	Load (n = 0..3) push(local[n]);
6	**store**	b	…, val …	Store local[b] = pop();
7..10	**store_n**		…, val …	Store (n = 0..3) local[n] = pop();

Sledeća kolona označava stanje steka izraza, ona ima dve vrste za svaku instrukciju. Prva vrsta je stanje e-steka pre izvršavanja instrukcije, a druga vrsta je stanje e-steka posle izvršavanja. Recimo instrukcija load ne zahteva ništa na e-steku pre izvršavanja instrukcije, a posle izvršavanja instrukcije, na vrh steka (vrh je uvek skroz desno) biće stavljena vrednost.

Poslednja kolona tabele daje objašnjenje efekta izvršavanja instrukcije, da se u lokalnoj oblasti na adresi b, to je u stvari relativna adresa preko frame pointera, pročita vrednost i da se onda potisne na e-stek. Zanimljivo je da load instrukcija ima više od jednog opkoda i dolazi u ova dva varijeteta, prvi je ovaj koji smo upravo objasnili sa jednim operandom, ali postoji i varijanta sa nula operanada. Za nju je izdvojeno četiri različita opkoda, za n jednako 0, 1, 2 i 3. Za razliku od prve varijate gde je blanko između load i N, ove ostale četiri se pišu sa spojenim N. Znači piše se load0, load1 i tako dalje, kao jedna reč. To su posebni operacioni kodovi. Te instrukcije imaju potpuno isto značenje kao ova load sa odvojenim operandom, ali zbog efikasnosti izvršavanja, znači ideja

je da ćemo najčešće baratati sa prve četiri lokalne promenjive, onda one imaju tu privilegiju da imaju svoje posebne opkodove, odnosno ovakva instrukcija se izvršava malo brže od load sa blankom, zato što interpreteru treba manje vremena da pročita i dekoduje samo jedan bajt, nego kada mora da pročita poseban operand. Isto tako važi za store. Store ima komplementan efekat, znači sa vrha e-steka skida vrednost i onda upisuje u lokalni okvir na relativnu adresu b. Isto ima ove varijante sa spojenim operandima od nula do tri. Što se tiče ovih globalnih promenljivih, imamo analogne getstatic i putstatic. One nemaju svoje skraćene varijante, nego tu je operand short, znači njih možemo imati 2^16 to jest 65.535 globalnih podataka.

Instrukcije za load i store polja objekata

13	**getfield** s	…, adr …, val	Load polja objekta adr = pop()/4; push(heap[adr+s]);

14	**putfield** s	…, adr, val
		…
			Store polja objekta
			val = pop(); adr = pop()/4;
			heap[adr+s] = val;

Za pristup poljima objekata, instanci klasa, koristi se getfield da dovuče vrednost iz polja objekta na e-stek. Ta instrukcija zahteva, pre njenog izvršavanja, da se početna adresa objekta u heap-u stavi na e-stek, a onda se zadaje ova instrukcija. Ona ima jedan short operand koji predstavlja pomeraj da bi se došlo do polja u okviru zapisa objekta. Efekat izvršavanja te instrukcije je da se pokupi adresa sa e-steka, da se sabere adresa plus pomeraj i da se onda pristupi heap-u i pročita se ta lokacija i sadržaj postavi na e-stek.

Instrukcija putfield ima na obrnuto dejstvo. Zahteva da pre izvršavanja instrukcije i početna adresa objekta i vrednost koju želimo da upišemo u polje, budu na e-steku i kao rezultat njenog izvršavanja, te dve vrednosti će biti skinute sa e-steka i u heap će biti na adr+s lokaciju upisana ta vrednost val.

Instrukcije za load konstanti

15..20	**const_n**	… …, val	Load konstante (n = 0..5) push(n)
21	**const_m1**	… …, -1	Load konstante -1 push(-1)
22	**const** w	… …, val	Load konstante push(w)

Rekli smo za razne konstante postoji opšta instrukcija const w, znači konstante su na širini cele memorijske reči od 32bita. Efekat izvršavanja instrukcije je da se ta konstanta postavi na vrh e-steka. Postoje dve posebne varijante, konstante od 0 do 5 su povlašćene i to se znači piše kao jedna reč, recimo const0, znači da dovučemo konstantu 0 na vrh e-steka. Takođe konstanta -1 je isto jedna od povlašćenih konstanti.

	code	*bytes*	*estack*
x = y;	load1 store0	1 1	y -
gx = gy;	getstatic 1 putstatic 0	3 3	gy -
p.x = p.y;	load2 load2 getfield 2 putfield 1	1 1 3 3	p p p p p.y -

Ovde ima nekoliko primera, kako se prevode naredbe dodele, ako su u pitanju dve lokalne promenljive x i y, onda se naravno pristupa tekućem okviru na p-steku. Instrukcija load1 je bez operanada, to je ona povlašćena varijanta, znači da se dovuče prva lokalna promenljiva i stavi na mesto nulte lokalne promenljive preko e-steka.

Dodela gx = gy, radi sa globalnim promenljivama, dovlači se prva globalna promenljiva na e-stek, a onda stavlja na mesto nulte globalne promenljive. Kolona bytes navodi sa koliko bajtova se kodira pojedinačna instrukcija. Što se tiče kodiranja, tačno koji bajtovi služe za kodiranje instrukcija definisano je u okviru bibliotečke klase Code i tamo je kroz konstante, mnemonike, direktno povezan svaki opkod sa konkretnim vrednostima.

U nekom objektu p, kome je inače zapamćena adresa u lokalnoj promenljivoj 2, želimo da vrednost polja y objekta p upišemo u polje x. To zahteva četiri instrukcije mikrojava bajtkoda, da se dovuče adresa objekta p na e-stek, trebaće nam još jednom ta adresa objekta p, jer jedna treba za čitanje polja, a druga za upis u drugo polje. I onda getfield 2 koristi jednu od ovih dovučenih p. U poslednjoj koloni je naveden sadržaj e-steka, koji se menja kako se koja instrukcija izvršava. Znači skinuće se drugo p sa vrha steka, biće sabrano sa 2 i pročitaće se heap na adresi p + 2. Ono što se pročita sa heap-a, a to je vrednost ovog p.y polja, staviće se na vrh e-steka. Posle toga e-stek sadrži adresu p i vrednost p.y. Zatim, putfield 1 je instrukcija koja upisuje u polje 1 tog objekta p, odnosno skinuće obe ove vrednosti sa e-steka i sabraće p sa 1 i na adresu heap[p+1] upisaće ovo py. e-stek će ostati prazan.

Aritmetičke instrukcije

Različite aritmetičke instrukcije imaju nula operanada, jer sve svoje operande i rezultat koriste na e-steku. Sabiranje zahteva dve vrednosti na vrhu e-steka i tokom izvršavanje instrukcije te dve vrednosti se skinu sa steka, saberu i rezultat se stavi na stek. Oduzimanje, množenje, deljenje, deljenje po modulu, kod koga se dobija ostatak pri deljenju, promena predznaka, pomeranje vrednosti, moraju biti dve vrednosti na e-steku, jedno je vrednost, a drugo je za koliko bitskih mesta se vrši pomeranje. Pošto je vrednost 32bitna, znači ima smisla da ovo drugo polje ide od 1 do 32 i umesto ove dve vrednosti, rezultat se smešta na stek. znači sve su to binarne operacije.

U nastavku je dat primer računanja izraza x + y * 3, zahteva dovođenje x i y na stek, ako su to lokalne promenljive, koriste se dve load instrukcije, zatim se dovodi konstanta 3, zatim mul

skida y i 3 i njihov proizvod stavlja na e-stek. Add instrukcija nema argumenata, znači skida x i međurezultat y*3 i finalni zbir potom stavlja na e-stek.

	code	*bytes*	*estack*
x + y * 3	load0	1	x
	load1	1	x y
	const3	1	x y 3
	mul	1	x y*3
	add	1	x+y*3

Pravljenje objekata

32	**new**	s	…	Novi objekat
			…, adr	alocirati oblast od s bajtova; inicijalizovati oblast nulama; push(adr(oblast));
33	**newarray**	b	…, n	Novi niz

			…, adr	n = pop();
				if (b==0)
				alocirati niz sa n
				elemenata veličine
				bajta;
				else if (b==1)
				alocirati niz sa n
				elemenata veličine reči;
				inicijalizovati niz nulama;
				push(adr(niz));

Za stvaranje novih objekata koristi se bajtkod instrukcija new sa jednim operandom koji je short tipa, znači dva bajta. Ne zahteva na e-steku nikakav sadržaj pre izvršavanja instrukcije. Efekat izvršavanja je da se alocira s memorijskih reči na heap-u, da se popuni ta novo dodeljena oblast nulama i da na e-stek ide početna adresa te oblasti.

Za alociranje nizova u mikrojavi koristi se newarray bajtkod instrukcija, koja ima jedan operand veličine jednog bajta i zahteva vrednost N na e-steku pre izvršavanja. Ovo b se odnosi na tip elemenata, njegove ispravne vrednosti su 0 ili 1. Ako je b jednako nula, onda želimo znakovni niz, a za b jednako 1, celobrojne nizove. N je broj elemenata niza koji hoćemo da alociramo. Efekat izvršavanja instrukcije biće alociranje onoliko reči na heap-u koliko je neophodno, znači za znakovne nizove biće pakovano po četiri i + 1 znači reč za broj elemenata. Oblast heap-a koja koja se dodeli će biti alocirana nulama, ali onda će na početak, znači upravo na ovu adresu na heap-u koja se vraća, biće smešten broj N. Početna adresa cele te oblasti, odnosno tačno adresa gde je smeštena ova konstanta N, se stavlja na e-stek, to je znači povratna vrednost iz ove instrukcije.

U nastavku su dati primeri sa kreiranjem objekta neke klase. U prvom primeru, kada se zada instrukcija new, želimo četiri lokacije. Na e-steku ćemo dobiti neku adresu p i onda ćemo je upisati u lokalnu promenljivu sa store 0. U drugom primeru, kada alociramo novi niz, onda moramo zadati

broj elemenata na e-steku i pozvati newarray 1 ako je celobrojni niz. Nama će instrukcija alocirati na heap-u odgovarajući broj, to je 5 + 1, reči; smestiće početnu adresu a na e-stek a skinuće vrednost 5. Potom moramo sa store1 upisati adresu a u lokalnu promenljivu za kasnije referenciranje elemenata niza.

mikrojava	*code*	*bytes*	*stack*	
Person p = new Person;	new 4	3	p	// assume: size(Person) = 4 words
	store0	1	-	
int[] a = new int[5];	const5	1	5	
	newarray 1	2	a	
	store1	1	-	

Pristup nizu

34	**aload**	…, adr, index
		…, val
			Load elementa niza (+ provera indeksa)
			i = pop(); adr = pop()/4+1;
			push(heap[adr+i]);

35	**astore**	…, adr, index, val
		…
			Store elementa niza (+ provera indeksa)
			val = pop(); i = pop();
			adr = pop()/4+1;
			heap[adr+i] = val;

Za pristup elementima niza, odnosno za čitanje i-tog elementa niza imamo aload instrukciju odnosi za celobrojne nizove, odnosno njenu baload varijantu za čitanje elementa znakovnih nizova. I jedna i druga nemaju operande, ali zahtevaju pre izvršavanja instrukcije dve vrednosti na e-steku. To je početna adresa niza i indeks i-tog elementa. Izvršavanjem instrukcije skinu se te dve vrednosti, pristupi se heap-u na indeksu adr+i+1, dodavanje jedinice je zbog toga što se veličina niza pamti u nultom elementu. Pročitana vrednost smešta se na e-stek. Varijanta baload razlikuje su utoliko što se i deli sa 4, jer su po četiri znaka smeštena u jednu reč. Isto je poenta da i-ti znak reči pročitamo, znači adresa+1+i/4 i još onda određeni bajt smeštamo na vrh e-steka. Naravno taj bajt se proširi u četiri bajta, kao cela reč.

Astore instrukcija služi za upis u i-ti element nekog niza. Moraju tri podatka da se zadaju pre izvršavanja: početna adresa adr, indeks (i) i vrednost koji upisujemo. Znači sve se to pokupi sa e-steka, upiše se u heap na indeksu adr+i+1 upravo ta vrednost val.

Varijanta za znakovne nizove je bastore. Ona samo malo manipuliše sa ulazna tri podatka i naravno upisuje samo jedan bajt. Za svaki niz možemo dobiti njegovu dužinu (broj elemenata), ovo je direktna podrška za ugrađeni method len, za njega postoji posebna bajtkod instrukcija arraylength, koja zahteva početnu adresu niza na e-steku i ona pristupi heap-u na toj adresi, pročita reč i tu vrednost stavi na vrh e-steka umesto adrese koju pokupi.

Ako bismo imali dodelu a[i] = b[i+1], vidimo to je niz ovih bajtkod instrukcija, znači prvo stavimo a na e-stek, zatim stavimo i e-stek, zatim sa load1 stavimo b na e-stek, zatim ponovo stavimo i na e-stek, sada imamo a i b i na e-steku, stavimo konstantu 1 na e-stek, sa add dobijemo i+1, aload iskoristi vrednosti b i i+1 da pročita i-plus-prvi element niza b i stavi na e-stek i konačno, upravo ta tri podatka a, i, b[i+1] iskoristi ova astore instrukcija da ažurira a[i] i isprazni e-stek.

mikrojava	*code*	*bytes*	*stack*
a[i] = b[i+1];	load0	1	a
	load2	1	a i
	load1	1	a i b
	load2	1	a i b i
	const1	1	a i b i 1
	add	1	a i b i+1
	aload	1	a i b[i+1]
	astore	1	-

Operacije na steku

39	**pop**	…, val …		Skidanje elementa sa vrha steka dummy = pop();

Imamo i instrukciju pop koja skida sa vrha e-steka vrednost i "zaboravlja", znači ako nam ne treba vrednost, nego da samo skinemo jedan element sa vrha e-steka.

Skokovi

42	**jmp**	s		Bezuslovni skok pc = pc + s;

43..48	**j<cond>**	s	…, val1, val2
			…
				Uslovni skok (eq, ne, lt, le, gt, ge)
				y = pop(); x = pop();
				if (x cond y) pc = pc + s;

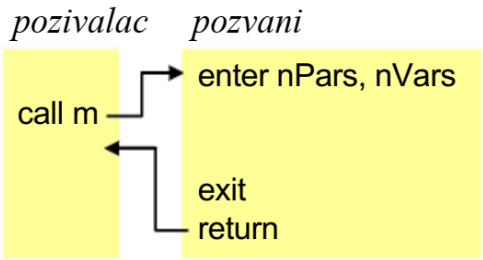
Za realizaciju skokova imamo instrukciju bezuslovnog uskoka, koja kao svoj short operand ima adresu skoka. Adresa skoka je relativna u odnosu na početak instrukcije skoka. Postoje i uslovni skokovi, gde se u operacionom kodu kodira jedan od nekoliko uslova jednakost, nejednakost, strogo manje, manje jednako, strogo veće, veće do jednako. Ta instrukcija zahteva dva operanda koji će se porediti relacionim operatorom. Ima nekoliko opkodova, jer se uslov kodira u opkodu. Instrukcija se izvršava tako što ove dve vrednosti skine sa e-steka, uporedi ih operatorom i, ako je uslov zadovoljen, onda se skače na odredišnu adresu. Ako uslov nije zadovoljen, izvršavanje se nastavlja sekvencijalno od sledeće instrukcije.

Pozivanje metoda

49	**call**	s	Poziv metode

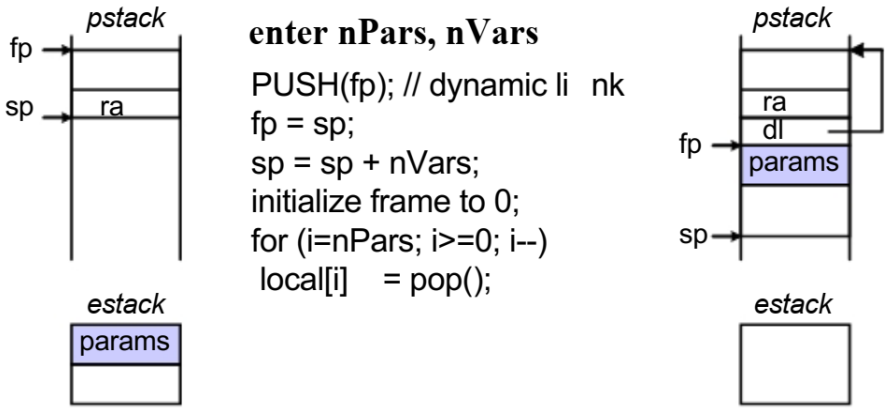
```
                                    PUSH(pc+3); pc := pc + s;
50      return                      Povratak iz metode
                                      pc = POP();
51      enter b1, b2                Početak obrade metode
                                      psize = b1;
                                      lsize = b2; // u rečima
                                      PUSH(fp);
                                      fp = sp; sp = sp + lsize;
                                    Inicijalizovati okvir nulama
                                      for (i=psize-1; i>=0; i--)
                                      local[i] = pop();
52      exit                        Kraj obrade metode
                                      sp = fp; fp = POP();
```

Što se tiče pozivanja metoda u mikrojava virtuelnoj mašini, sledi objašnjenje za statičke metode kada nije polimorfni mehanizam u igri. Onda imamo nešto slično što imaju i realni procesori. Imamo instrukciju poziva potprograma, gde se adresa zadaje operandom. Instrukcija se izvršava tako što povratnu adresu, znači adresu naredbe odmah iza call, stavlja na programski stek (to je ovaj PUSH), a onda izvršavanje se nastavlja od zadate adrese s.



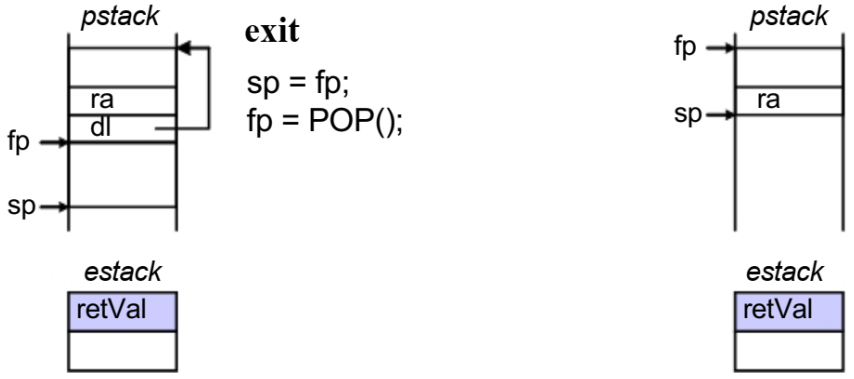
Zatim imamo enter instrukciju, koja zahteva dva operanda i u nastavku ćemo objasniti malo bolje tačan efekat instrukcije. Pozivalac kodira call m instrukciju i onda kada prenosimo kontrolu pozvanom potprogramu, na početku svakog pozvanog potprograma (to je slično kao kod Intel-ovog x86) imamo enter instrukciju, koja ima dva operanda: prvi operand odnosi se na ukupan broj stvarnih parametara, a drugi operand odnosi se na veličinu potrebnu za pamćenje lokalnih promenljivih metoda. Pozivalac mora da, pre ove call instrukcije, na vrh p-steka redom stavi parametre znači stvarne parametre poziva i onda izvršava call instrukciju.

Kada se kontrola prenese kod pozvanog potprograma izvršava se instrukcija enter. Ovo je izgled programskog i expression steka u trenutku kad je pozvan enter. Tada fp i sp još ukazuju na okvir pozivaoca, samo je na vrh p-steka naredbom call stavljena povratna adresa ra. Na sledećoj slici levo je izgled stekova neposredno pre izvršavanja enter, a desno je izgled p-steka i e-steka pošto se enter izvrši.



Kakav tačno efekat ima enter? Enter ima složeni efekat ažuriranja i jednog i drugog steka. Prvo se na p-stek stavlja stara vrednost frame pointera, to u suštini odgovara dinamičkoj vezi, zatim se vrednost stek pointera upiše u frame pointer, tako da se postavlja frame pointer za okvir pozvanog potprograma. Zatim se p-stek pointer uveća za vrednost ovog drugog parametre nVars, čime se alocira prostor za lokalne promenljive. nVars je broj lokacija koje zahtevaju svi parametri, plus sve lokalne promjenjive metoda. Zatim se inicijalizuje sadržaj tog alociranog dela na nula i onda se prebace svi parametri sa e-steka redom u lokalne promenljive i to je kompletan efekat enter instrukcije.

Iza enter ide bajtkod onoga što programer isprogramira za neki metod. Sekvenca povratka se realizuje sa exit i return. Exit je vrlo slična leave instrukciji kod x86 i ima zadatak da u suštini obrne efekat procesiranja enter instrukcije. Instrukcija exit je jednostavnija od enter i nema nikakve parametre. Ona zatiče p-stek koji je ostavio enter, plus što, ako je u pitanju funkcija, povratna vrednost funkcije uvek se stavlja na e-stek, a ovo drugo je sve ono kako ostavio enter. Exit samo treba da uradi sp = fp, čime delaocira sve lokalne promenljive i opet vraća stek pointer da ukazuje na polje dinamičke veze, odnosno stari fp. Potom se pročita stari fp iz tog polja dinamičke veze i postavi kao što odgovara pozivaocu. I to je efekat exit. Kada se sa steka skine polje dinamičke veze, na na vrhu p-steka je povratna adresa. Instrukcija return će pročitati tu povratnu adresu sa p-steka i preneti kontrolu izvršavanja iza call. Sada pozivalac može sa e-steka da pokupi povratnu vrednost funkcije, ako funkcija ima rezultat.



Kako izgleda u bajtkodu neki poziv nekog potprograma koji vraća rezultat? Treba pripremiti stvarne parametre, tako što se potisnu na e-stek, recimo iz nekih lokalnih promenljivih a i b. Pozove

se potom metod m i, ako taj metod m vraća neki rezultat, on se sa store upiše u odgovarajuću lokalnu promenljivu.

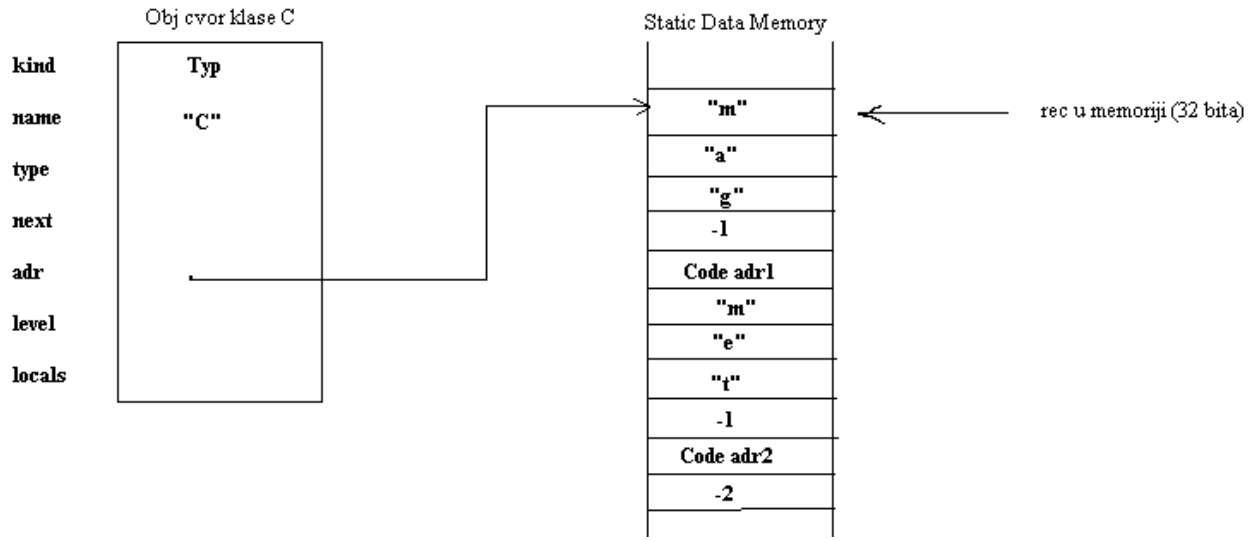
mikrojava	*bajtkod*
c = m(a+b);	load a
	load b
	call m
	store c

Poziv dinamičkih metoda

Sada da objasnimo kako se realizuje u mikro java virtuelnoj mašini poziv dinamičkih metoda klase. Rekli smo da svaka klasa, osim glavne koja predstavlja program, ima virtuelnu tabelu metoda, v-tabelu. Ona se čuva u statičkoj oblasti podataka, a adresa prve reči v-tabele se pamti u samom kompajleru u okviru Obj čvora klase, u polju other. Recimo da u mikrojava izvornom kodu imamo nekakvu klasu C koja ima neka polja x, y i z i neka dva metoda, dinamička, mag i met.

```
class C {
    int x,y, z;
    int mag() {
        return 3;
    }
    void met(int q) {
        q=q+1;
    }
}
```

U kompajleru će biti formirana njena virtuelna tabela, biće i alocirana u delu statičke memorije, u smislu rezervacije adresa, a fizičko popunjavanje se naravno odlaže za trenutak izvršavanja programa. Kompajler zna gde se nalazi za klasu C virtuelna tabela.



Sadržaj v-tabele biće od dva zapisa, za metod mag redom ime metoda, rekli smo svako slovo se pamti u čitavoj memorijskoj reči, znači tri memorijske reči za ime metoda zatim jedna reč za

konstantu -1 (kraj imena metoda) i jedna memorijska reč za početnu adresu metoda mag kada ga komparater prevede. Onda ide drugo ime metoda met, -1 kraj imena i adresa drugog metoda i -2 označava kraj ove virtuelne tabele. Znači to je struktura koju kompajler formira, a koja u runtime-u mora da se napuni u statičku oblast memorije i koristi se pri pozivu ovih dinamičkih metoda.

Evo da vidimo sad kroz primer, kako ide korišćenje objekta i poziv dinamičkog metoda. Rekli smo da se operator new, koji alocira memoriju na heap-u, koristi za pravljenje i alokaciju objekata neke klase. Sada ćemo videti, sasvim konkretno, za ovaj fragment mikrojava programa, instanciranje novog objekta klase C koju smo ranije definisali i poziv metoda met klase C sa stvarnim parametrom a.

```
C c=new C;  // klasa C napred definisana
int a;
c.met(a);
```

Kakav bajtkod mora da se generiše? Kao što vidite, bajtkod je podugačak, postoji kompleksna obrada iza toga.

```
new 16   /* klasa C ima tri polja, plus jos jedno za adresu v-tabele, znaci 4 puta 4 bajta */
putstatic 0  /* pretpostavka je da je c statička promenljiva glavne klase na poziciji 0 */
getstatic 0
const 32     /* pretpostavka je da v-tabela za C počinje na adresi 32 u Static Data */
putfield 0   /* 32 je ubačeno u nulto polje objekta kojem pristupa referenca c */
/* poziv metode met */
getstatic 0 /* na stek izraza se stavlja c koje poziva metodu, pa prema tome predstavlja prvi stvarni
argument, tj. this */
getstatic 1 /* stavljanje drugog stvarnog argumenta. Pretpostavka je da je a statička promenljiva glavne
klase na poziciji 1 */
getstatic 0    // na stek se
getfield 0     // stavlja adresa v-tabele klase objekta na koji ukazuje referenca c
invokevirtual 'm' 'e' 't' -1  /* poziv metode met. -1 označava kraj instrukcije */
```

Prvi deo bajtkoda odnosi se na pravljenje novog objekta klase C. Ukupno nam treba 16 bajtova, jer imamo tri polja x, y i z, svako po četiri bajta to je 12 bajtova, plus četiri bajta za pokazivač na v-tabelu, ukupno 16 bajtova. Pozivom bajtkod instrukcije new na e-stek će doći adresa odgovarajućeg zapisa na heap-u. Sada treba zapamtiti tu početnu adresu, pod pretpostavkom da je ovo c statička promenljiva na indeksu 0, zapamtićemo tu početnu adresu u c. Potrebno je još upisati u nultu reč novog objekta na heap-u pokazivač na virtuelnu tabelu. Znači moramo ponovo dovesti početnu adresu objekta na e-stek, staviti konstantu 32. Pretpostavka je da je kompajler smestio virtuelnu tabelu na adresu 32 statičke oblasti. putfield 0 koristi početnu adresu na heap-u sa e-steka, pomeraj je 0, znači da baš upiše u nultu reč tog objekta ovu konstantu 32. Na taj način smo napravili novi objekat i on sada ima dinamički tip C, ova konstanta 32 u suštini reprezentuje i dinamički tip tog objekta, a za sva ova druga njegova polja, pošto je to podrazumevani konstruktor, vrednosti su im 0. Znači ovih pet instrukcija bajtkoda služi da realizuje ovu jednu liniju koda.

Posle toga još imamo ovu drugu liniju koda, gde pozivamo metod met. Kako teče poziv metoda met? Iz globalne promenljive c čitamo adresu objekta i stavljamo na e-stek, to je implicitni stvarni parameter this. Pod pretpostavkom da je neka statička promenljiva na poziciji jedan, vrednost a stavljamo na e-stek. Sada su na e-steku vrednosti oba stvarna parametra, this i a. Zatim nam je

potrebna adresa v-tabele objekta, a nju ćemo dobiti tako što opet stavimo adresu celog objekta na e-stek i sa getfield 0 dohvatimo to nulto polje objekta, koje je ujedno adresa početka v-tabele. I onda sledi ovaj glavni poziv invokeVirtual, posebna instrukcija mikrojava bajtkoda za realizaciju dinamičkih poziva. Ta instrukcija kao svoje argumente ima niz bajtova koji predstavljaju ime metoda i kraj argumenata je -1, znači kraj imena. Ta instrukcija prvo uzme sa vrha steka adresu v-tabele, pristupi toj tabeli i pretragom u ulazima v-tabele traži poklapanje ovog imena, koje je zadato kao argument, sa nekom imenom u v-tabeli. Ako je ime nađino iz v-tabele se čita početna adresa koda metoda, teluća vrednost pc-a upisuje se na stek kao povratna adresa, a onda se skače na adresu koja je pročitana iz v-tabele.

Ulaz/Izlaz

53	**read**	…	Operacija čitanja
		…, val	readInt(x); push(x);
54	**print**	…, val, width	Operacija ispisa
		…	width = pop(); writeInt(pop(), width);

Ulaz/izlaz je vrlo elementaran, zato što je osnovna namena mikrojave da se napiše kompajler, a ne da se pišu neki elaborirani programi. Tako da ima samo osnovni ulaz u obliku čitanja jednog celobrojnog podatka, ili čitanja jednog znakovnog podatka. Instrukcija read ne zahteva ništa na steku, prekida izvršavanje programa da možete da ukucate broj, koji će runtime da konvertuje u ceo broj i onda će ta vrednost da se nađe, posle izvršavanja instrukcije, na vrhu e-steka. Izlaz je realizovan sa print. On zahteva vrednost koja se štampa i na kojoj širini, znači koliko znakova na izlazu da se ta vrednost kao string predstavi, znači te dve vrednosti se pokupe sa steka i ispisuju na terminalu.

Za znakovne promenljive imamo bread i bprint, slično kao ove dve, samo se odnose na jedan znak.

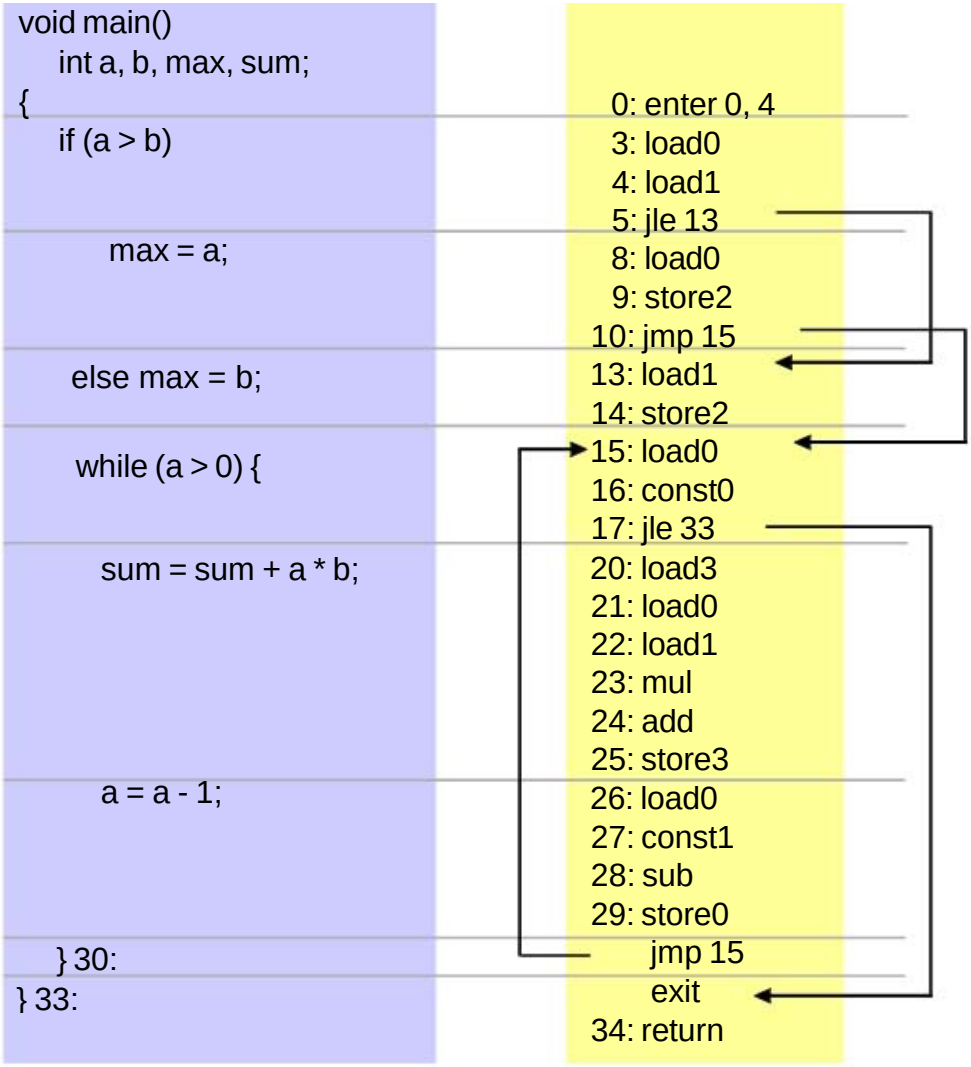
Ostalo

57	**trap**	b	Generiše run time grešku zavisno od vrednosti b se ispisuje odgovarajuća poruka o grešci i prekida izvršavanje programa

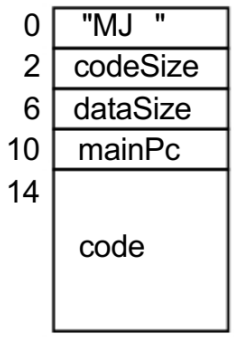
Postoji instrukcija softverskog prekida trap, koja ima jedan operand, koji je povezan sa različitim porukama o grešci koje runtime izdaje. Efekat izvršavanje instrukcije trap je da se prekine izvršavanje programa i da se izda poruka o grešci koja zavisi od ovoga parametera b. trap se koristi za jednu konkretnu namenu: kako kompajler prevodi telo metoda, na samom kraju tela uvek stavlja jednu trap instrukciju. To je ako slučajno programer uspe da programski nekako zaobiđe naredbu return, da se naleti na ovaj trap koji bi onda izazvao grešku u izvršavanju, da se ne dozvoli da program odluta na neku slučajnu lokaciju.

Primer prevođenja

Ovde jedan kompleksniji primer prevođenja čitavog jednog mikrojava metoda. U telu metoda koriste se if i while konstrukcije. Naravno sve mora da se prevede na nivo uslovnih i bezuslovnih skokova u bajtkodu. Promenljive a, b, max i sum redom su smeštene na adrese 0, 1, 2 i 3.



Na kraju da kažemo da je definisano, kao što prava Java ima .class fajlove, tako i mikrojava ima svoj objektni fajl, nije definisana standardna ekstenzija, ali najčešće koristimo .obj. To je fajl koji na ulazu zahteva mikrojava runtime, Kada kompajler prevede program, on će generisati jedan objektni fajl. Struktura objektnog fajla je takva da prva dva bajta u tom fajlu treba da budu slova MJ, kao neka magična konstanta da se zna da je u pitanju sadržaj objektnog fajla.



Sledeća četiri bajta su veličina prostora programskog koda, zatim sledeća 4 bajta su veličina statičkih podataka, znači koliko da runtime alocira za sve metode i za sve globalne promenljive, pošto su to fiksne veličine. Sledeća četiri bajta je adresa gde se u code oblasti nalazi main program odakle treba početi izvršavanje. I još jedino što u fajlu kreće od 14 bajta pa nadalje, to je prevedeni bajtkod. Veličina ovog dela je promenljiva, u zavisnosti od sadržaja prevedenog programa. Tako da je to sve što se pamti u objektnom fajlu. Možda je taj format mogao i da se proširi, recimo treba primetiti da nema nikakav početni sadržaj statičke oblasti, znači alocira se samo ta statičko oblasti i napuni nulama, a na potom morate u izvršnom kodu puniti virtuelne tabele, mada je njihov sadržaj fiksan.

Načini predstavljanja međukoda

U nekom vrlo jednostavnom kompajleru, bilo bi zamislivo da se iz semantičkih akcija ugrađenih u parser, ili obilaskom apstraktnog stabla, neposredno generiše kod. Međutim, realni kompajleri to nikada ne rade, nego koriste jedan ili više oblika međukoda (engleski intermediate representation, skraćeno IR) u transformaciji ulaznog programa u finalni mašinski kod.

Postoji više prednosti da se koristi međukod, u odnosu na varijantu da se direktno generiše mašinski kod. Prednji kraj kompajlera, to jest, sintaksno-semantički analizator i generator međukoda su mašinski nezavisni. Kompajler se lako prilagođava novoj mašini izmenom zadnjeg kraja kompajlera, generatora mašinskog koda iz međukoda. Ako prednji krajevi kompajlera za različite jezike generišu isti medukod, moguće je za ciljnu mašinu napraviti samo jedan generator koda za sve jezike. Optimizacije je moguće sprovoditi na međukodu, čime se pravi mašinski nezavisni optimizator.

U ovoj lekciji sagledaćemo nekoliko načina predstavljanja međukoda. Lekcija je uglavnom zasnovana na "Dragon book" knjizi i govorićemo na konceptualnom nivou, izostavićemo razne implementacione detalje.

Neki od načina su sintaksno stablo ili aciklični graf. Videćemo da graf može da se koristi da se identifikuju i eliminišu ponavljanja usled zajedničkih podizraza. Zatim postfiksna notacija troadresni kod, graf toka kontrole na nivou bazičnih blokova i takozvana statička forma jedinstvene dodele, odnosno static singla assignment form, skraćeno ssa forma.

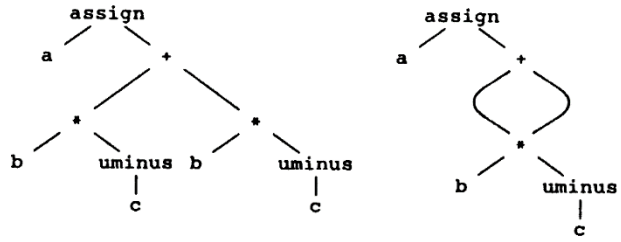
Predstavljanje međukoda putem sintaksnog stabla ili acikličkog grafa

Primer

Što se tiče predstavljanja međukoda putem sintaksnog stabla, odnosno akcikličkog grafa, možemo odmah sagledati na primeru. Slika govori kao hiljadu reči. Ovde je namerno izabran jedan iskaz dodele, a:= b * -c + b * -c, gde se promenljivoj a dodeljuje ovaj izraz koji, opet namerno izabrano, ima dva identična podizraza, b *-c koji se sabiraju. Ako predstavimo izraz sintaksnim stablom, u listovima stabla biće imena promenljivih, a u unutrašnjim čvorovima stabla biće operatori. Operatori su aritmetičke operacije u izrazu, takođe i dodela promenljivoj a je operator koji je upravo u korenu stabla.

U stablu vidimo da svaki čvor ima najviše jednog roditelja, odnosno praktično svaki ima tačno jednog, a koren nema roditelja, dok u acikličnom grafu čvor stabla potencijalno može da ima više od jednog roditelja. U acikličnom grafu vidimo da je uočena činjenica da se podizraz b*-c ponavlja i onda je čvor koji predstavlja taj podizraz, to jest, čvor množenja iskorišćen i kao levi sin i kao desni sin čvora koji reprezentuje sabiranje, tako da to kasnije omogućava generisanje koda da se samo jednom računa izraz i onda da se taj podizraz upotrebi tamo gde god treba, u konkretnom slučaju dva puta.

Stablo, odnosno graf za iskaz: a:= b * -c + b * -c



(a) Syntax tree. (b) Dag.

Atributivno-translaciona gramatika za iskaz dodele koja konstruiše sintaksno stablo:

1. $<S>_s \rightarrow id_n \{MKLEAF\}_{x,y,z} := <E>_{s1} \{MKNODE\}_{a,b,c,d}$
   x ← "id" y ← n a ← "assign" b ← z c ← s1 s ← d
2. $<E>_s \rightarrow <E>_{s1} + <E>_{s2} \{MKNODE\}_{a,b,c,d}$
   a ← "+" b ← s1 c ← s2 s ← d
3. $<E>_s \rightarrow <E>_{s1} * <E>_{s2} \{MKNODE\}_{a,b,c,d}$
   a ← "*" b ← s1 c ← s2 s ← d
4. $<E>_s \rightarrow – <E>_{s1} \{MKNODE\}_{a,b,c,d}$
   a ← "uminus" b ← s1 c ← NULL s ← d
5. $<E>_s \rightarrow (<E>_{s1})$
   s ← s1
6. $<E>_s \rightarrow id_n \{MKLEAF\}_{x,y,z}$
   x ← "id" y ← n s ← z

Ako želimo da sagledamo kako možemo semantičkim akcijama izgenerisati stablo, odnosno graf, ovde je dat jedan jednostavan primer gramatike koja sa nekoliko smena opisuje jedan ovakav iskaz dodele i neke izraze. Prva smena opisuje da je id terminal koji opisuje promenljivu, <E> opisuje izraze i onda imamo nekoliko smena za izraze. Pošto je gramatika konceptualna, vidimo da je dvosmislena, znači morao bi se dalje razrađivati za konkretno pasiranje, a pošto je ovo samo koncept, znači izrazi su <E> + <E>, <E> * <E>, - <E> i (<E>), znači dovoljno da ovaj naš primer reprezentuje.

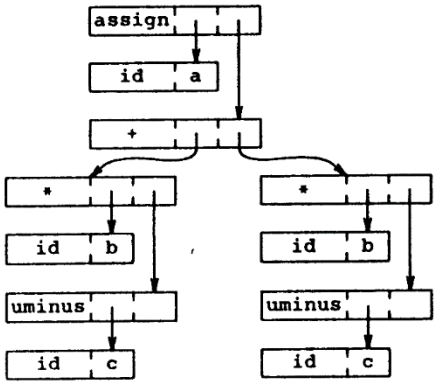
Imamo dve vrste semantičkih akcija, MKNODE i MKLEAF. Akcija MKLEAF ima tri argumenta x, y, z konstruiše čvor list sintaksnog stabla, pri čemu nasleđeni atribut x označava tip čvora, a nasleđeni atribut y ime promenljive koje je predstavljena tim čvorom, ili pointer u tabeli simbola gde se nalaziti ta promenljiva. Sintetizovani atribut z je pokazivač na konstruisani čvor. Znači z je u suštini izlaz kada semantičke analiza aktivira tu akciju, stvara se čvor stabla i dobija se pokazivač na novo konstruisani čvor.

Dosta slična akcija MKNODE za unutrašnje čvorove ima četiri argumenta a, b, c i d, konstruiše unutrašnji čvor sintaksnog stabla. Nasleđeni atribut a označava o kojoj operaciji je reč, a nasleđeni atributi b i c, su pokazivači na čvorove naslednike, Sintetazovani atribut d je pokazivač na kostruisani čvor. Stablo se konstruiše od dna ka vrhu, tako da onda već imamo naslednike ovoga čvora. U samoj gramatici, neterminal <E> koji predstavlja izraz ili delove izraza, ima jedan sintetizovani atribut koji pretstavlja pokazivač na deo sintaksnog stabla koji odgovara izrazu <E>. Neterminal <S> ima sintetizovani atribut koji predstavlja pokazivač na kompletno sintaksno

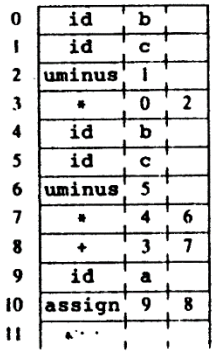
stablo. Terminali id koji predstavlja ime promenljive ima atribut koji je to ime ili pokazivač na tabelu simbola gde se nalazi ta promenljiva.

Gramatika može da se koristi i za konstrukciju stabla i za konstrukciju grafa. U čemu je jedina razlika? Ako želimo da konstruišemo aciklički graf, onda kada god pozovemo akciju MKNODE, potrebno je da se u okviru tog poziva ispitaju parametri s kojima smo pozvali, a to su ovi nasleđeni atributi a,b,c i da se njihove vrednosti keširaju za već napravljene čvorove. Sada ako imamo novi poziv i utvrdimo da smo već napravili čvor sa potpuno istim ulaznim parametrima a, b i c, onda ne treba da pravimo novi čvor, nego treba da vratimo pokazivač na već napravljeni čvor i onda ćemo u stvari dobiti ovu strukturu acikličkog grafa.

Implementacija sintaksnog stabla (grafa)



stablo zapisa



vektor zapisa

Što se tiče konkretne implementacije, odnosno struktura podataka za reprezentaciju sintaksnog stabla, odnosno grafa, ona može biti dinamička (stablo zapisa) gde naravno dinamički alociramo zapise i koristimo memorijske pokazivače, tako da realizujemo one grane stabla odnosno grafa. Uobičajena struktura za reprezentaciju stabala.

Alternativno, možemo, ako želimo da izbegnemo dinamičku alokaciju iz nekog razloga, to je u današnje vreme relativno retko, ali znači moguće je alocirati vektor u memoriji i onda da zapis predstavljaju pojedine elemente tog vektora. Kad god nam treba novi zapis, pamtimo indeks dokle smo iskoristili vektor i novi zapis uvećava indeks iskorišćenog prostora i vraćamo taj poslednji indeks. Umesto pokazivača, grane grafa bile bi reprezentovane indeksima gde se nalaze odgovarajući zapisi sinova stabla odnosno grafa. Naravno, pošto ima dve vrste zapisa, jedan ima dve komponente, to je list, a drugi ima tri komponente, to je unutrašnji čvor, mi prosto možemo uvesti da svaki ima tri komponente, ali kod listova ne koristimo ovu jednu komponentu.

Predstavljanje međukoda putem postfiksne notacije

Primer

Postfiksna notacija za iskaz: a:= b * -c + b * -c

a b c uminus * b c uminus * + assign

Ponovo za isti iskaz dodele iz prethodnog primera, postfiksna notacija je ovde data u ovoj drugoj liniji. U postfiksnoj notaciji, uvek prvo idu operandi, pa onda ide aritmetička operacija. Generalno postfiksna notacija pogodna je za izvršavanje na stek mašini.

Kako se ovakav jedan izraz izvršava? Prosto ga procesiramo sa leva na desno. Svaki put kada naiđe operand znači promenljiva, radimo push na nekakav aritmetički stek, znači ove tri bi išle na stek. Ako naiđe operator, ako je unarni operator, skidamo skidamo s vrha steka jedan element, uradimo operaciju i rezultat operacije ponovo stavljamo na stek. Ponovo kada dođe sledeći, recimo ovaj binarni operator množenja, skinuli bismo dva operanda sa steka, izvršili operaciju i stavili rezultat na stek. Tako se vrlo lako procesira ceo izraz i to je u suštini osnova rada stek mašine. Mi ćemo ovaj koncept postfiksne notacije vrlo razraditi u narednim poglavljima, jer mikrojava virtuelna mašina, koju mi koristimo u kursu i u projektu, kao i prava Java virtuelna mašina je u stvari vrsta stek mašine i upravo radi na ovom konceptu.

Ako želimo da sagledamo kako semantičkim akcijama u gramatici dobiti postfiksnu notaciju, to je relativno lako: opet imamo slično gramatiku kao maločas, koja opisuje ove iskaze dodele i izraze kao primer. Dovoljno je u ovim smenama koje predstavljaju izraze staviti akcije koje reprezentuju operatore na sam kraj smene, jer onda će se prvo emitovati operandi, pa onda tek operacija i naravno ovde imamo ID reprezentuje razne operande, ovde su samo uvedeni a, b, c, koliko se pojavljuju u primeru. Isto se i oni emituju na izlaz, čim se na ulazu prepozna neki operand, odmah se emituje odgovarajuća akcija koja ga reprezentuje na izlazu.

Ako imamo sintaksno stablo, a želimo da konvertujemo u postfiksnu notaciju, onda treba prosto obići to stablo u postorder obilasku -- prvo se obrade levi i desni naslednik čvora, pa onda na kraju i sam čvor.

1. <S> → <ID> := <E> {assign}
2. <E> → <E> + <E> {+}
3. <E> → <E> * <E> {*}
4. <E> → – <E> {uminus}
5. <E> → (<E>)
6. <E> → <ID>
7. <ID> → a {a}
8. <ID> → b {b}
9. <ID> → c {c}

Predstavljanje međukoda putem troadresnog koda

Sledeći način predstavljanja međukoda je putem troadresnog koda. To je jedan oblik koda koji donekle podseća na mašinski kod i to znači za neku troadresnu mašinu, ali apstrahovan je u smislu da ne mora da se vodi računa o konkretnim lokacijama promenljivih i konkretnim načinima adresiranja, koji inače postoje u realnim procesorima. Evo ovde nabrojani su oblici i vrste iskaza, koji tipično postoje u takvom jednom troadresnom kodu, prema [Dragon Book].

1. Naredbe dodele oblika x := y op z, gde je op binarna aritmetička ili logička operacija.
2. Naredbe dodele oblika x := op y, gde je op unarna operacija. Osnovne unarne operacije uključuju unarni minus, logičku negaciju, operatore pomeranja i operatore konverzije koji, na primer, konvertuju broj sa fiksnom zarezom u broj sa pokretnim zarezom.

3. Naredbe kopiranja oblika x := y gde se vrednost y dodeljue x.

4. Bezuslovni skok goto L. Troadresna naredba sa oznakom L je sledeća koja se izvršava.

5. Uslovni skokovi kao što je if x relop y goto L. Ova instrukcija primenjuje relacioni operator (<, =, >=, itd.) na x i y i izvršava naredbu sa oznakom L sledeću ako x stoji u odnosu relop na y. Ako ne, troadresna naredba koja sledi if x relop y goto L se izvršava sledeća, kao u uobičajenoj sekvenci.

6. param x i call p, n za pozive procedure i return y, gde je y, koje predstavlja vraćenu vrednost, opciono. Njihova tipična upotreba je kao niz troadresnih naredbi

   param x_1
   param x_2
   . . .
   param x_n
   call p, n

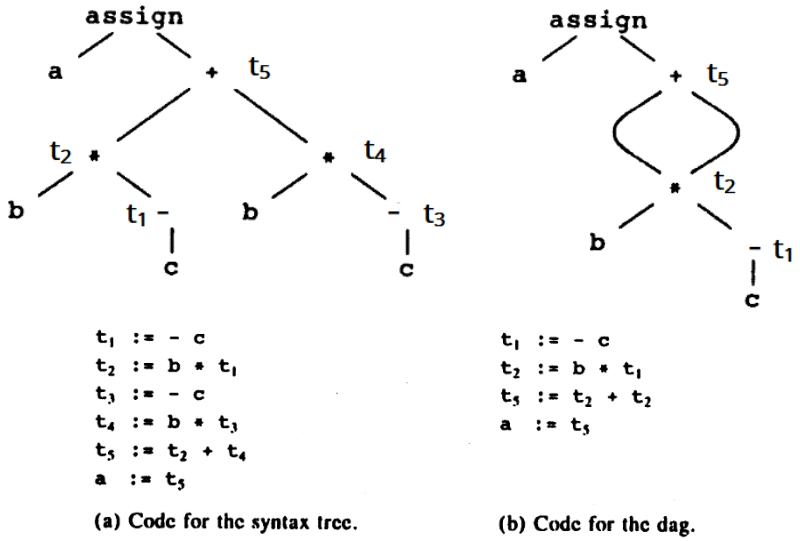
   generisanih kao deo poziva procedure p(x_1, x_2,… x,,). Ceo broj n koji označava broj stvarnih parametara u pozivu "call p, n" nije suvišan, jer pozivi mogu biti ugnežđeni.

7. Indeksirane dodele oblika x := y[i] i x[i] := y..Prva postavlja x na vrednost u lokaciji memorije koja je i jedinica posle lokacije y. Naredba x[i] := y postavlja sadržaj lokacije koja je i jedinica posle x na vrednost y. U obe naredbe. x, y i i odnose se na podatke.

8. Dodeljivanje adresa i pokazivača oblika x :=&y, x := *y i *x := y. Prva od njih postavlja vrednost x da bude lokacija y. Pretpostavlja se da je y ime, možda privremena promenljiva, koje označava izraz sa l-vrednošću kao što je A[i, jJ, a x je ime pokazivača ili privremena promenljiva. To jest, r-vrednost x je l-vrednost (lokacija) nekog objekta. U iskazu x := •y, verovatno je y pokazivač ili privremena promenljiva čija je r-vrednost lokacija. R-vrednost x je jednaka sadržaju te lokacije. Konačno, •x : = y postavlja r-vrednost objekta na koji ukazuje x na r-vrednost y.

Sa navedenim tipovima troadresnih naredbi može da se pokrije jedan osnovni troadresni međukod. Za iskaz dodele a = b*-c + b*-c, u ranijem primeru dobili sintaksno stablo, odnosno aciklički graf. Možemo na bazi tih predstava stabla odnosno grafa, da dobijemo troadresni međukod. U osnovi, mi bismo numerisali čvorove stabla, tako što krenemo od korena, pa numerišemo redom čvorove, uvek biramo kao sledeći čvor čiji su svi roditelji već numerisani i onda kad dobijemo određenu sekvencu brojeva, idemo suprotno, što znači da naredbe generišemo od listova stabla ka korenu. Pri tome čvorovima dodelimo privremene promenjive t_i, recimo vrednost dela izraza -c, kad se upotrebi unarni minus nad c, ćemo upisati u t_1. U čvoru pored može da se napiše t1, njemu smo dodelili privremenu promenljiv t_1. Zatim ovom ovde gore čvoru smo dodelili privremenu promenjivu t_2 i tako dalje.

Sada kad krenemo od listova ka korenu, uz upotrebu tih privremenih promenljivih koje reprezentuju vrednosti u pojedinim čvorovima stabla, za ovo stablo dobijemo troadresni međukod prikazan ispod stabla, a za aciklički graf dobijemo još jednostavniji međukod sa manje naredbi, zato što je iskorišćena činjenica da se ovaj izraz b*-c koji je jednom izračunat i rezultat smešten t_2, vidimo da je $t_5 = t_2+t_2$, znači ne računamo ga zaista dva puta kao u stablu, nego iskoristimo već izračunati međurezultat.



Troadresni međukod može da se izgeneriše, ako je potrebno, direktno semantičkim akcijama iz gramatike, tako što se ubaci generisanje troadresnih naredbi u svakoj smeni. Sledi gramatika koja za iskaz dodele generiše troadresni kod:

1. <S> → id_n := <E>$_s$ {GEN3}$_{a,b,c}$
   a ← n b ← ":=" c ← s

2. <E>$_s$ → <E>$_{s1}$ + <E>$_{s2}$ {GEN1}$_{a,b,c,d,e}$
   (s, a) ← newtemp b ← ":=" c ← s1 d ← "+" e ← s2

3. <E>$_s$ → <E>$_{s1}$ * <E>$_{s2}$ {GEN1}$_{a,b,c,d,e}$
   (s, a) ← newtemp b ← ":=" c ← s1 d ← "*" e ← s2

4. <E>$_s$ → – <E>$_{s1}$ {GEN2}$_{a,b,c,d}$
   (s, a) ← newtemp b ← ":=" c ← "uminus" d ← s1

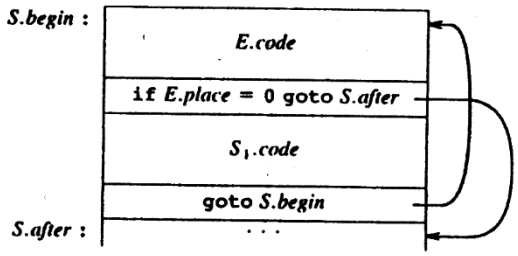
5. <E>$_s$ → (<E>$_{s1}$)
   s ← s1

6. <E>$_s$ → id_n
   s ← n

Terminal id ima atribut koji predstavlja naziv odgovarajuće promenljive. Neterminal <E> ima sintetizovani atribut koji predstavlja naziv privremene promenljive koja sadrži međurezultat za deo izraza uparen sa <E>. Svaki poziv funkcije newtemp rezerviše novu privremenu promenljivu. Akcija {GEN} od svojih argumenata sastavlja jednu troadresnu naredbu (ima promenljiv broj nasleđenih atributa). U prvoj smeni je generisanje ove dodele vrednosti izraza promenljivoj, u smenama gde su izrazi je generisanje odgovarajućih troadresnih naredbi sa binarnim operacijama, u četvrtoj smeni je generisanje troadresne naredbe sa unarnom operacijom. Parametri semantičke akcije su operacija, operandi i dodela.

Do sada smo kao primere navodili isključivo izraze, u nastavku možemo da vidimo i na primeru jedne naredbe kontrole toka, konkretno while petlje, kako generisati troadresni kod za nju. Sledeća šematska slika prikazuje šta želimo da postignemo pri generisanju koda. Na početku petlje

generišemo simboličku labelu s.begin koja služi da se vratimo u sledećoj iteraciji na kod za računanje ovog uslovnog izraza petlje koji ispituje uslov ostanka u petlji i to je ovaj e.code.



Gramatička smena za iskaz while koja generiše troadresni kod prema slici je sledeća:

<S> → while $\{GENL\}_{l1}$ <E>$_t$ $\{GEN5\}_{a,b,c,d}$ <S> $\{GEN4\}_{g,h}$ $\{GENL\}_{l2}$
(l1,h) ← newlabel (d,l2) ← newlabel
a ← "if" b ← t c ← "= 0 goto" g ← "goto"

GENL – generiše definiciju labele u međukodu
GENC – generiše izvršne naredbe međukoda

Neterminal <S> predstavlja iskaz tela petlje, a <E> predstavlja uslovni izraz. Iz <E> se generiše, izraz za proveru uslova ostanka u petlji. Prva akcija GENL ubačena ispred <E>, odnosi se na to da se u međukodu izgeneriše labela s.begin sa prethodne slike. Zatim obradom <E>, to ne treba znači semantički obrađivati u ovoj smeni, nego kako se budu prepoznavale smene za <E>, generisaće se *e.code*, troadresni kod za računanje izraza. Zatim GEN5 akcija generiše uslovni if u međukodu, koji treba da ispita rezultat *e.place* izraza <E>, da li je jednak nuli, i ako jeste treba da izvrši skok izlaza izvan petlje. Izlaz iz petlje će biti reprezentovan labelom *s.after*. Zatim, kada se bude procesirao neterminal <S> na desnoj strani, biće izgenerisano telo petlje. Posle generisanja tela petlje, GEN4 akcija služi da izgeneriše u troadresnom kodu bezuslovan skok na sledeću iteraciju, odnosno na labelu *s.begin*, da bi se računao uslov ostanka u petlji za sledeću iteraciju. Konačno ima još jedna GENL akcija koja generiše mesto, odnosno poziciju *s.after* labele koju smo iskoristili u prethodnom uslovnom skoku. Znači, sa ukupno četiri semantičke akcije, moguće je izgenerisati sve što je potrebno od troadresnog koda za obradu ove while petlje.

Implementacija troadresnih naredbi

Kada razmatramo strukture podataka za implementaciju i reprezentaciju troadresnih naredbi u programu, moguće varijante prema [Dragon book] su:

- *Četvorke*, odnosno zapisi sa četiri polja gde se operandi predstavljaju pokazivačima na tabelu simbola.
- *Trojke* -- u tim zapisima izostavljeno je polje za rezultat, nego se redni broj naredbe navodi kada se koristi međurezultat koji ta naredba izračunava.
- *Indirektne trojke* -- ne koriste se redni brojevi trojki, nego umesto njih pokazivači na memorijsku adresu gde je smeštena trojka čiji se rezultat koristi na mestu operanda. Uvodi se još dodatan vektor pokazivača koji definiše redosled trojki, omogućujući jednostavnu izmenu redosleda - samo se ažurira vektor, a sadržaj u trojkama ostaje neizmenjen.

Problem sa trojkama je što nije lako izmeniti redosled u sekvenci naredbi, ako želimo recimo da dve naredbe zamene mesto, to nije lako uraditi ako je kod reprezentovan trojkama. Zato se kod

indirektnih trojki koristi još jedan nivo indirekcije. Sada ćemo to sve videti na jednom konkretnom primeru.

Primer

Naš raniji primer troadresnog međukoda, koji je bio dobijen na bazi sintaksnog stabla, ima ukupno šest troadresnih naredbi,

```
t1 := - c
t2 := b * t1
t3 := - c
t4 := b * t3
t5 := t2 + t4
 a := t5
```

Ako ovaj kod reprezentujemo četvorkama, to su zapisi sa četiri komponente: prva komponenta je operacija, druga i treća komponenta su argumenti operacije, (naravno ako je u pitanju unarna operacija, drugi argument ne koristimo) i konačno, u četvorkama se eksplicitno reprezentuje i rezultat troadresne naredbe. Kod trojki izostavljamo polje rezultata eksplicitno, nego Indeks, odnosno adresa samog zapisa, u stvari služi i kao oznaka za rezultat. Tako da, recimo, ako druga troadresna naredba koristi međurezultat generisan u prvoj troadresnoj naredbi, mi ovde koristimo adresu prve troadresne naredbe, odnosno adresu 0 kao jedan od operanada. Ovde nije lako zameniti redosled naredbi, zbog tih ubačenih indeksa. Zašto bi se uopšte vršila neka pretumbacija ovog redosleda troadresnih naredbi? To može da bude rezultat optimizacije. Optimizator može da zaključi da treba da izmeni redosled, recimo približi korišćenje rezultata njegovom generisanju da bi manje opterećivao procesorske registre i tako dalje.

	op	*arg 1*	*arg 2*	*result*
(0)	uminus	c		t_1
(1)	*	b	t_1	t_2
(2)	uminus	c		t_3
(3)	*	b	t_3	t_4
(4)	+	t_2	t_4	t_5
(5)	:=	t_5		a

četvorke

	op	*arg 1*	*arg 2*
(0)	uminus	c	
(1)	*	b	(0)
(2)	uminus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	assign	a	(4)

trojke

Postoji znači potreba nekada se preraspodele naredbe u sekvenci i onda je pogodna implementaciona forma takozvane indirektne trojke. Sledeća slika predstavlja istu trojku kao i ranije, ali uvodimo još jedan dodatan vektor. Ovaj vektor određuje redosled naredbi, tako da ovde, u ovom konkretnom primeru, vidimo da prva naredba u sekvenci ima indeks 14 u ovom drugom vektoru. Ovo je indeks naredbe kako je ona originalno generisana, a ovaj drugi vektor daje redosled tih naredbi kako će ga kasnije recimo generator koda uzeti u obzir.

	statement
(0)	(14)
(1)	(15)
(2)	(16)
(3)	(17)
(4)	(18)
(5)	(19)

	op	*arg 1*	*arg 2*
(14)	uminus	c	
(15)	*	b	(14)
(16)	uminus	c	
(17)	*	b	(16)
(18)	+	(15)	(17)
(19)	assign	a	(18)

indirektne trojke

Preraspoređivanje naredbi u ovoj varijanti traži samo ažuriranje ovog vektora, znači samo bismo promenili redosled, u desnom vektoru zapisa ne bismo ništa morali da menjamo, za razliku od običnih trojki, kada bismo promenili redosled nekih naredbi, morali bismo da tražimo po poljima gde se naredbe koriste da izmenimo njihove indekse.

Predstavljanje međukoda grafom toka kontrole programa na nivou osnovnih blokova

Naredbe, recimo, troadresnog koda u kompajleru se nikada ne razmatraju kao prosta sekvenca, nego se obično pravi struktura podataka koja se zove graf toka kontrole na nivou osnovnih blokova. Da bismo definisali tu strukturu podataka, prvo ćemo definisati osnovne blokove.

Osnovni blok (engl. basic block) je skup iskaza međukoda koji se izvršavaju u sekvenci (bez grananja) izuzev krajnjeg iskaza u bloku, koji može biti uslovni ili bezuslovni skok. U sekvenci naredbi međukoda sledećim algoritmom identifikujemo osnovne blokove:

- Prvo se odrede *vođe*, to jest prvi iskazi osnovnih blokova. To je ili prvi iskaz u programu, ili iskaz koji predstavlja odredište nekog uslovnog ili bezuslovnog skoka, ili iskaz koji neposredno sledi bezuslovni ili uslovni skok.
- Kada odredimo sve iskaze vođe, isparcelišemo ceo ostatak koda tako što svakom vođi pridružimo iskaze koji ga slede do prvog sledećeg vođe, isključujući naravno tog sledećeg vođu, ili do kraja programa.

Jednom kada isparcelišemo sekvencu narebi na osnovne blokove, nad njima dodatno možemo uspostaviti međusobne relacije, tako što ceo program predstavimo u vidu *grafa toka kontrole na nivou osnovnih blokova*. Čvorovi ovoga grafa su osnovni blokovi koje smo identifikovali, a grane uvodimo po sledećim pravilima: od bloka B1 postoji orijentisana grana ka bloku B2, ako:

- B2 neposredno sledi B1 u nekoj izvršnoj sekvenci, odnosno ako postoji uslovan ili bezuslovan skok od poslednjeg iskaza B1 ka vođi B2, ili
- B2 neposredno sledi B1 u međukodu, a B1 nema na kraju bezuslovni skok.

Znači kontrola se ili sekvencijalno prenosi sa jednog bloka na drugi, ili nekim eksplicitnim skokom.

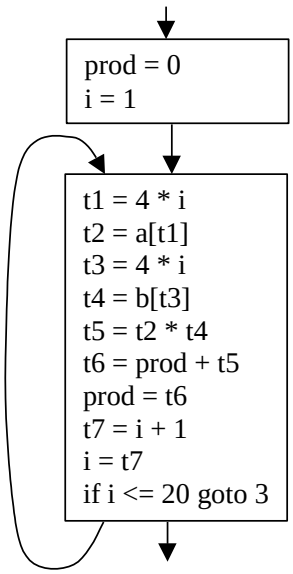
Primer

Sve će biti jasnije na primeru, znači ako u izvornom kodu pođemo od nekog konstrukta iz višeg programskog jezika, do while petlje i nekih dodatnih naredbi izračunavanja vrednosti nekoga izraza, koji u višem programskom jeziku može biti komplikovan izraz, to će na nivou troadresnog koda sve da se svede na trodresne naredbe, a ova do while petlja će se svesti na uslovni goto.

```
prod = 0;
i = 1;
do {
    prod = prod + a[i] * b[i];
    i = i + 1;
} while ( i <= 20 );
```

Ovo je sekvenca od 12 troadresnih naredbi koja odgovara polaznom fragmentu izvornog koda.

```
1. prod = 0
2. i = 1
3. t1 = 4 * i
4. t2 = a[t1]
5. t3 = 4 * i
6. t4 = b[t3]
7. t5 = t2 * t4
8. t6 = prod + t5
9. prod = t6
10. t7 = i + 1
11. i = t7
12. if i <= 20 goto 3
```



Prvo odredimo osnovne blokove, tako što vidimo da je početak programa, iskaz 1, prvi vođa, a zatim vidimo da u sekvenci imamo skok na 3, tako da iskaz 3 predstavlja isto vođu. Prvi osnovni blok prema tome, obuhvata iskaze 1 i 2, do ovog drugog vođe, ali njega nećemo uključiti. Od iskaza 3 do 12, pošto je sve sekvencijalno izvršavanje, samo naredba 12 je skok, znači predstavljaju drugi osnovni blok.

Imamo ukupno dva bloka i sad među njima uspostavimo grane. Vidimo da od naredbe 2 ka naredbi 3 imamo sekvencijalno izvršavanje (prva grana) i takođe imamo iskaz 12, uslovni skok koji predstavlja povratnu granu za ovaj drugi osnovni blok. Na taj način smo dobili ovaj graf toka kontrole na nivou osnovnih blokova. To je bitna struktura podataka, koju praktično svaki kompajler koristi i sve obrade koje se rade unutra jednog osnovnog bloka zovu se lokalne obrade, recimo lokalne optimizacije ili lokalno generisanje koda gleda isključivo granice jednog osnovnog bloka, a globalne optimizacije su kada se gleda ceo ovaj graf toga kontrole.

Predstavljanje međukoda statičkom formom jedinstvene dodele (engl. Static Single Assignment form – SSA)

Sledeći način predstavljanja međukoda je *statička forma jedinstvene dodele*, odnosno engleski static single assignment form, skraćeno SSA. U programu koji je u SSA formi, svakoj dodeli promenljive daje se jedinstveno ime i sve upotrebe te promenljive, koje koriste vrednost te dodele, se preimenuju da reflektuju dodeljeno ime. Ovo bi bila neka neformalna definicija.

Ako je kod sekvencijalan, prevođenje u SSA oblik je lako izvesti, vidimo da ovde imamo dve dodele promenljivoj V, naravno to programer nikad tako ne bi pisao u sekvenci ali je moguće dobiti kao rezultat nekog generisanja medjukoda i moramo svakoj dodeli dati jedinstveno ime dodavanjem indeksa.

V = V + 4		$V_0 = 4$
X = V + 5	posle preimenovanja	$X = V_0 + 5$
V = 6		$V_1 = 6$
Y = V *7		$Y = V_1 *7$

Promenljive X i Y čak i ne moramo da indeksiramo, ali za promenljivu V, njenu prvu dodelu obeležavamo za V0, a drugu dodelu sa V1. Znači vršimo to preimenovanje da ne bismo imali isto ime promenljive u različitim dodelama vrednosti. Ovde gde koristimo prvu vrednost pišemo V0, a ovde gde koristimo drugu vrednost pišemo V1 i sve deluje vrlo jednostavno.

Međutim, komplikacije nastaju kada imamo grananja u kodu. Razmotrimo ovaj relativno jednostavan fragment gde na bazi nekog if-a, u then delu imamo jednu dodelu X-u, u else delu drugu dodelu X-u i onda iza if-a imamo dodelu Y = X.

```
if (...)
      X = 5;
else
      X = 3;

Y = X;
```

Koja će tačno vrednosti biti dodeljena promenljivoj Y, da li 5 ili 3, zavisi od toka izvršavanja programa, znači statički kompajler ne može da odredi šta će biti ovde. U nekom scenariju izvršavanja programa može biti dodeljena vrednost 5, u drugom scenariju izvršavanja programa može biti dodeljena vrednost 3. Lako je preimenovati prvi X u X0, drugi X u X1, ali onda ne možemo reći da li da pišemo Y = X0 ili Y = X1. Tu dilemu rešavamo takozvanim φ funkcijama, znači uvodimo novu dodelu X-u (X2) kao rezultat φ funkcije, koja ima argumente mogućih vrednosti X-a, znači X0 i X1. Onda imamo dodelu Y = X2.

```
if (...)
      X0 = 5;
else
      X1 = 3;
X2 = φ(X0, X1);
Y0 = X2;
```

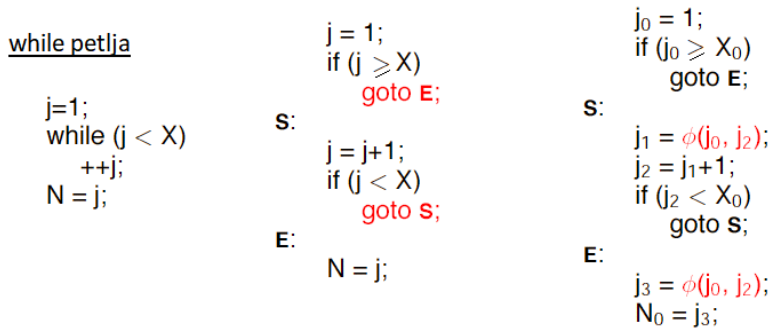
Šta ova φ funkcija tačno znači? Vrednost φ funkcije jednaka je X0, ako kontrola u poslednji bazični blok ulazi iz then dela if-a, odnosno vrednost φ funkcije jednaka je X1, ako kontrola ulazi kroz else deo gornjeg if-a.

U opštem slučaju, ako φ funkcija ima n argumenata V_1, V_2, …, V_N, u osnovnom bloku B, sa N blokova prethodnika u grafu kontrole toka i ti prethodnici su redom P_1, P_2,... P_N, vrednost ove $\varphi(V_1, V_2, \ldots, V_N)$ funkcije iznosi V_i, ako tok izvršavanja prenosi kontrolu programa sa i-tog prethodnika na blok B. Znači φ funkcija za neku promenljivu V ima tačno toliko argumenata koliko posmatrani blok ima ulaznih grana sa dodelama promenljivoj V.

U nastavku dajemo još jedan primer sa φ funkcijama. Ako u izvornom kodu na višem programskom jeziku imamo ovakav fragment sa while petljom kao na strani 264 u levoj koloni, onda moguće je prevesti tu while petlju na nivo troadresnog međukoda, što je prikazano u srednjoj koloni. Imamo na dva mesta ispitivanje uslova izlaska iz petlje. Prvi if je u stvari ispitivanje negacije, da li uopšte treba da uđemo u petlju. Ako uslov nije ispunjen onda ćemo odmah iskočiti iza tela petlje, a naravno ako uđemo u petlju na kraju svake iteracije, imamo ispitivanje uslova ostanka u petlji.

Treća kolona je međukod u SSA formi. Da bi bilo u SSA formi, moramo preimenovati sve dodele, vidimo da tu postoji više dodela promenljivoj j, tako da imamo j_0, početnu dodelu, zatim j_1

odgovara korišćenju j u petlji, znači u blok koji reprezentuje telo petlje možemo stići ili tako što ulazimo u prvu iteraciju, ili što se vraćamo na sledeću iteraciju, znači ima dva prethodnika, tako da nam je tu potrebna φ funkcija, koja ima argumente ili j_0 ili j_2 iz prethodne iteracije. Rezultat φ funkcije ćemo obeležiti sa j_1 i on se koristi u izrazu kojim se računa j_2. Takođe, kada izađemo iz petlje, finalni blok ima dva prethodnika. Jedan prethodnik je kada preskačemo izvršavanje petlje, a drugi prethodnik je ako smo ušli u petlju i uslov za sledeću iteraciju više nije zadovoljen. Tako da nam je i tu potrebna jedna φ funkcija, na bazi inicijalnog j_0 i j_2 koji se generiše u petlji. U krajnje desnoj koloni je finalna varijanta koda u SSA.



Formalno, program je u obliku SSA ako:

1. Svaka promenljiva poseduje tačno jednu dodelu vrednosti.

2. Dodela promenljivoj *dominira* svim upotrebama te promenljive.

Pojam dominacije se definiše za grafove, znači u grafu toka kontrole na nivou iskaza, ne može se od ulaznog čvora doći do čvora upotrebe promenljive, a da se ne prođe kroz čvor dodele toj promenljivoj. To znači čvor dodele dominira čvorom upotrebe, ne može biti zaobiđen.

Ostala je malo nejasna semantika φ funkcije u vreme izvršavanja programa -- kako generisati kod za nju. Sva sreća, φ funkcija nije nešto što je ostaje u run-time-u, nego to je jedna formalizacija koja kompajleru samo služi za određene potrebe. Pre izvršavanja nekih optimizacija, kod se prevede u SSA formu, a zatim, pošto se optimizacija izvrši i dobije finalni optimizovani kod, vrši se vraćanje natrag iz SSA forme u međukod koji nema φ funkcije i na taj način ih se otarasimo.

"Vraćanje" iz SSA forme nazad u običan međukod

Kako se vrši to vraćanje iz SSA forme nazad u običan međukod? Najjednostavnija varijanta je da φ funkciju sa n argumenata u bloku B, treba zameniti sa n običnih dodela, po jednom na kraju svakog bloka prethodnika bloku B u grafu. To je vrlo jednostavan način kako eliminišemo φ funkciju, znači sa tih n dodela koje preselimo na krajeve blokova prethodnika, međutim ovim se nekada dobije nefikasan kod sa suvišnim dodelama, ali postoje dodatne optimizacije koje nećemo razmatrati, da se vidi koje od tih dodela su stvarno neophodne, a koje nisu.

Primer sa osnovnim blokovima

Na sledećoj strani dat je završni primer, u izvornom kodu imamo for petlju koja obrađuje elemente niza a. Izvorni kod preveden je u troadresni međukod. U tom troadresnom međukodu prvo treba identifikovati osnovne blokove. Ovde je znači vođa prvi iskaz međukoda v=g1, zatim vođa je takođe ovaj iskaz koji koji je odredište skoka, vidimo da je označen labelom L1 i takođe iskaz koji sledi if pošto je definisano da iskaz koji neposredno sledi bezuslovni ili uslovni skok takođe spada

u vođe. Tako da sada u sekvenci iskaza imamo tri osnovna bloka. Prvi osnovni blok obuhvata prve tri naredbe. Drugi osnovni blok je if, sam za sebe i treći osnovni blok obuhvata sve naredbe od ovoga t4=&a do goto L1 zaključno. U desnoj koloni data je predstava na nivou grafa toka kontrole. Granama su pokrivene sve varijante gde se kontrola prenosi između osnovnih blokova. Konačno, ako još treba da uvedemo SSA formu, vidimo da je, gde god postoje ulazne grane, recimo u if-u se koristi v i pošto ovaj blok B2 ima ulaz ili od gore ili iz goto, ovo v može biti ili ova dodela v1, ili ova dodela v3. Ovde je ažuriranje v, tako da nam je potrebna jedna φ funkcija i uz uvođenje te φ funkcije i dodeljivanja indeksa raznim promenljivama, da bi se tačno označilo koja dodela se gde koristi, dobijamo finalni graf toka na nivou osnovih blokova pri čemu je u samim blokovima kod u SSA formi.

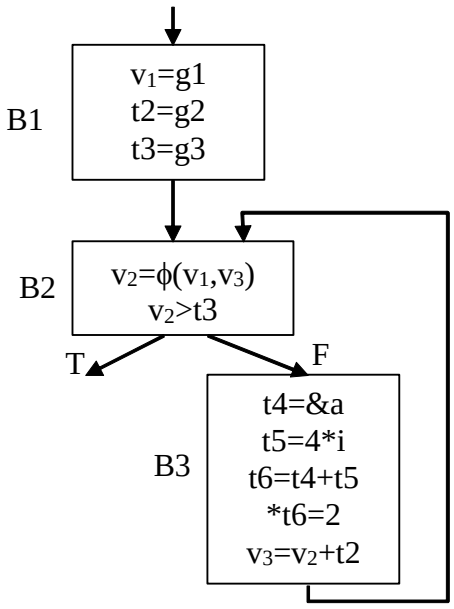
Izvorni kod

```
for (v=g1; g3; v+=g2 )
        a[i]=2;
```

međukod (četvorke)

```
        v=g1
        t2=g2
        t3=g3
L1:     if v > t3 goto L2
        t4 = &a
        t5 = 4*i
        t6=t4+t5
        *t6=2
        v=v+t2
        goto L1
L2:
```

međukod (SSA forma)



Pogodnosti za korišćenje SSA forme su u eksplicitnoj predstavi informacija vezanih za tok podataka što olakšava optimizacije koje su zasnovane na analizi toka podataka. Mane SSA forme su u tome što je ograničena na skalarne vrednosti (ceo niz se mora tretirati kao veliki scalar) i uvođenje suvišnih dodela vrednosti kada se konvertuje iz SSA u običan međukod.

Generisanje koda u kompajleru

Ulaz u generator koda je međukod programa, izlaz je mašinski kod za ciljnu mašinu. Moguće varijante izlaza su apsolutizovani mašinski kod, zatim objektni kod o kome je više detalja dato u predmetu sistemski softver, asemblerski izvorni kod, ako izlaz potom ide u asembler.

Sad nešto opšte o funkcijama generatora koda. Prva važna funkcija je izbor naredbi, to jest mapiranje međukoda u mašinski kod. U ovoj funkciji podrazumeva se fiksirano mapiranje promenljivih u memoriju i neograničen broj registra, to se rešava kao poseban problem. Znači ovde je fokus na izboru operacija i načina adresiranja. Druga funkcija generatora koda je raspoređivanje naredbi, odnosno instruction scheduling. Tu se radi promena redosleda naredbi da se smanje kašnjenja, upotreba registara i tome slično. Podrazumeva se fiksirani program, to jest skup instrukcija. Tu igraju ulogu i hardverski elementi, kao što je pajplajn i slično. Treća funkcija generatora kada je dodela registara, odnosno alokacija, što je odlučivanje o mapiranju promenljivih u memoriju i registre procesora. Znači tu je moguće privremeno deljenje iste lokacije od strane više promenljivih, pošto su registri procesora scarce, odnosno redak resurs.

Ono što komplikuje problematiku, to je da ove tri funkcije generisanja koda ne mogu nezavisno da se rešavaju, nego su tesno povezane. Možemo sagledati samo neke probleme generisanja koda, ako se tome pristupi suviše naivno. Recimo, ako imamo dve naredbe, A=B+C, D=A+E i ako potpuno nezavisno, znači za ovu prvu naredbu izgeneriše mo to su ove prve tri naredbe asemblerskog koda, da u registar AX dovedemo B, da dodamo na to C i da onda u memorijsku lokaciji a zapamtimo rezultat i, na sličan način, ova druga naredba se rešava sa donje tri, odmah ćemo uočiti da u kompletnoj generisanoj sekvenci koda imamo nepotrebno pamćenje, naravno pod pretpostavkom da nam a kasnije ne treba, nepotrebno se beleži a u memoriju, i plus naravno, ova druga instrukcija dovođenja ponovo vrednosti a u AX je potpuno nepotrebna u svim scenarijima. Ova prva je nepotrebna, ako nam a dalje ne treba.

Primer
Generisati mašinski kod za naredbe:

```
a = b + c
d = a + e
```

Generisani kod:

```
MOV AX, b
ADD AX, c
MOV a, AX      } nepotrebno
MOV AX, a
ADD AX, e
MOV d, AX
```

Drugi problem je problem izbora naredbi. Recimo nekakav izraz. ako programer napiše a = a+1 svakako može da se prevede recimo na x86 asembleru ili kao inkrementiranje vrednosti a, ili znatno komplikovanije sa tri naredbe, znači da se dovede a u AX, da se upotrebi sabiranje sa

konstantom jedan i da se ponovo AX smešta u a. Jasno je već intuitivno, da je ova sekvenca manje efikasna od ove prve, da zauzima i više memorije i potrebno je veće vreme izvršavanja.

Primer
Izraz a = a +1
može se prevesti na (bar) dva načina:

```
INC a       MOV AX, a
            ADD AX, 1
            MOV a, AX
```

Sledeći primer je problem dodele registara promenljivim, odnosno izbor skupa promenljivih koje će biti u registrima u datoj tački izvršavanja programa, što ima neposredan odraz na efikasnost koda, odnosno broj pristupa memoriji jer je pristupanje registrima neuporedivo kod savremenih procesora efikasnije nego pristup memoriji. Rešavanje ovoga problema je eksponencijalno zavisno od broja promenljivih, znači dokazuje se da je problem NP kompletan i onda naravno se daju neki suboptimalni algoritmi, koji mogu da efikasno urade, ali tu znači vidi se da je to složen problem.

Ciljna mašina – podskup x86 asemblera

Za ovu lekciju, ciljna mašina će nam biti podskup x86 asemblera. Radi se o dvodresnoj mašini.

Asemblerske naredbe su oblika operacija i onda argumenti su odredište i izvorište:

OP destination, source

Takva naredba ima značenje da rezultat ide u odredište, a izvorni operandi su prvi je isto dat tim odredišnim operandom, pa operacija, pa izvorišni operand:

destination := destination OP source

x86 ima to ograničenje da istovremeno ne mogu i izvoriše i odredište biti memorijska lokacija, odnosno jedan od dva operanda mora biti u registru. Operacije će za naše potrebe biti ove neke osnovne aritmetičke: sabiranje, oduzimanje, množenje, deljenje i takođe kopiranje vrednosti iz izvorišta u odredište. Množenje i deljenje donekle su specifične zato što zahtevaju da se koristi par registara. Kod množenja, rezultat množenja ide u određene registre: gornji deo u DX, donji deo rezultata u AX. Pošto je u pitanju 16bitna mašina znači rezultat je 32bitni, a izvorišni (prvi) operandi je uvek fiksno AX, a samo se zadaje drugi operand. Što se tiče deljenja, tu je deljenik u paru registara dx:AX, zadaje se delilac, a rezultat fiksno ide u AX, dok ostatak pri deljenju ide u registar DX. x86 ovaj koji mi razmatramo, ima 4 registra opšte namene, AX, BX, CX i DX i indeksne koji imaju neke specifične ramene, SI i DI i BP. Adresiranje je na nivou bajta, memorijska reč je dva bajta. U našem primerima, svi operandi i rezultati operacija će biti 16bitni. Što se načina adresiranja tiče, ova tabela nam daje kratko podsećanje. Imamo memorijsko direktno adresiranje, to jest, apsolutno gde se adresa zadaje kao neka memorijska lokacija. Zatim registarsko gde je operand zadat kao registar koji se koristi, indeksno, to je vrednost registra plus minus neka konstanta se koristi kao pokazivač u memoriju gde je smešten operand. Zatim registarsko indirektno, tu nema konstante nego samo vrednost registra je pokazivač u memoriju i neposredno, gde se zadaje konstanta C. Kod registarskog indirektnog adresiranja mogu se koristiti samo registri ovi navedeni BX, SI i DI, kod indeksnog samo registri BX, BP, SI i DI.

Tabela ** Načini adresiranja kod razmatarnog podskupa x86

Način adr.	Sintaksa	Objašnjenje
Apsolutno	M	Simbolička mem. adresa M

Registarsko	R	Registar R
Indeksno (r.relat)	[R±c]	Konstanta c ± sadržaj reg. R
Indirektno reg.	[R]	Sadržaj registra R
Neposredno	c	Konstanta c

Jednostavan algoritam generisanja koda za jedan bazični blok

Mi ćemo ilustrovati znači klasično generisanje koda algoritmom iz knjige aho, lan, sethi, ullman, Compilers -- principles, techniques and tools, poznatom Dragon book, gde je opisan jedan vrlo jednostavan algoritam generisanja koda, koji radi na nivou jednog bazičnog bloka. U prvom koraku toga algoritma, kao preduslov za primenu samog algoritma, određuju se informacije o životu i sledećem korišćenju za svaku pojavu promenljivih u bazičnom bloku. Da definišemo te pojmove: promenjiva X je živa u iskazu I programa, ako postoji putanja od nekog iskaza J, po kojoj se X ne menja, a u iskazu J promenljiva X se koristi kao operand. Tada iskaz J predstavlja sledeće korišćenje promenljive X. U suprotnom, za promenljivu X se kaže da je mrtva.

Kako ćemo izračunati ove informacije o životu i sledećem korišćenju promenljivih? Ako se ne vrši globalna analiza na kompletnom grafu toka kontrole, konzervativna pretpostavka je da su na izlazu iz bazičnog bloka, sve korisničke promenljive žive, a privremene promenljive mrtve. Izuzeci su ako je blok u petlji, a da je indeks petlje neka privremena promenljiva. Ali to je da kažem neki specijalni slučaj. Ove informacije sačuvamo u tabeli simbola za svaku promenljivu. To je jedna posebna struktura, koja nam sad samo treba za izračunavanje ovih informacija. Zvaćemo je tabela simbola, zato što za svaku promenljivu čuvamo tu informaciju. Konzervativna pretpostavka znači da naš kod će funkcionalno sigurno biti ispravan, a može biti neoptimalan.

Postupak izračunavanja (unutar bazičnog bloka):

1. Ako se ne vrši globalna analiza, konzervativna pretpostavka je da su na izlazu iz bazičnog bloka sve korisničke promenljive žive, a privremene mrtve (postoje izuzeci). Ove informacije sačuvamo u tabeli simbola (T.S) za svaku promenljivu.
2. Iskazi u bazičnom bloku obrađuju se redosledom od poslednjeg ka prvom. Kada naiđemo na i-tu četvorku
   i. x := y op z ;

   iz T.S. pokupimo informacije o x, y, z i prikačimo i-toj četvorci.
   u T.S. ažuriramo x = (m , -), to jest, x je mrtvo i nema sledeće korišćenje
   u T.S. ažuriramo: y, z = (ž , i), to jest, y i z su živi a iskaz i predstavlja sled. kor.

U drugom koraku, primenjujemo sam algoritam. Iskazi u bazičnom bloku obrađuju se redosledom od poslednjeg ka prvom. Znači, kada naiđemo na i-tu četvorku X međukoda, x = y op z, iz tabele simbola prvo treba da pokupimo informacije o x, y i z i prikačimo i-toj četvorci. Zatim u tabeli simbola ažurirano da je x mrtvo, jer to što sad stavljamo u tabelu simbola, pošto idemo unazad kroz kod, znači važiće pre ove naredbe x. Ako X ima dodelu ovde, nadalje više ne važi ta stara vrednost koja je došla do ovog trenutka ovde, znači ova dodela Xu ubija tu staru vrednost. Tako da ažuriramo da je X mrtvo i nema sledeće korišćenje, a za y i z ažuriramo da su živi i da imaju sledeće korišćenje baš u i-toj naredbi. Ilustrovaćemo algoritam na jednom konkretnom primeru, recimo da je u izvornom kodu zadata neka naredba dodele s nekim kompleksnijim izrazom.

Primer

Generisati mašinski kod za sledeću instrukciju :

d := (a – b) + (a – c) + (a – c).

Međukod :				
	1. t := a – b	t = (ž , 3)	a = (ž , 2)	b = (ž , -)
	2. u := a – c	u = (ž , 3)	a = (ž , -)	c = (ž , -)
	3. v := t + u	v = (ž , 4)	t = (m , -)	u = (ž , 4)
	4. d := v + u	d = (ž , -)	v = (m , -)	u = (m , -)

Tabela simbola

a	b	c	d	t	u	v
~~(ž , -)~~	~~(ž , -)~~	~~(ž , -)~~	~~(ž , -)~~	~~(m , -)~~	~~(m , -)~~	~~(m , -)~~
~~(ž , 2)~~	(ž , 1)	(ž , 2)	(m , -)	~~(ž , 3)~~	~~(ž , 4)~~	~~(ž , 4)~~
(ž , 1)				(m , -)	~~(ž , 3)~~	(m , -)
					(m , -)	

Na nivou međukoda, mogli bismo ovaj izraz predstaviti s ove četiri četvorke. Znači prvo izračunamo A minus B dodelimo nekoj privremenoj promenjivoj T. zatim izračunam a minus C, dodelimo nekpj privremenoj promenljivoj u. Ako uočimo da postoje zajednički podizrazi, ne moramo to dvaput računati nego saberemo ovaj deo, dobijamo v i konačno, kada saberemo taj međurezultat sa ovim prvim T, dobijamo finalno vrednost korisničke promenljive d.

Sada treba da izračunamo, za ovu sekvencu međukoda, informacije o životu i sledećem korišćenju. Moramo uvesti tabelu simbola sa svim promenljivama. Prve četiri su korisničke promenljive, ove druge tri su privremene promenljive. Početno stanje u tabeli simbola odgovara tački izlaza iz ovog bazičnog bloka. Rekli smo po konzervativnoj pretpostavci, sve korisničke promenljive su žive, nemamo informaciju o sledećem korišćenju, to je negde u nastavku programa, ali nama je fokus samo na ovom bazičnom bloku, a privremene promenljive su mrtve, jer ih je kompajler uveo, tako da znamo da sigurno van bloka neće više biti korišćene.

Krećemo od poslednje četvorke, kačimo ono što trenutno piše za svaku pojavu promenljive u četvrtoj četvorci iz tabele simbola kupimo trenutnu informaciju, znači da je d živo, a u i v su mrtvi. Kačimo znači taj podatak za četvrtu četvorku, ažurirano tabelu simbola da je d mrtvo, a v i u živi, a sledećim korišćenjem u četvorci broj 4.

Time smo završili sa ovom poslednjom četvorkom, prelazimo na pretposlednju. Iz tabele simbola izvlačimo trenutne informacije o v, t i u; dobili smo da su v i u živi sa korišćenjem u 4, a t je još ona inicijalna informacija da je mrtvo. Sada ažuriramo znači da je v mrtvo, a t i u živi sa sledećim korišćenjem u 3.

Prelazimo na drugu četvorku, izvlačimo informacije o u, znači da je živo u 3, a i c da su živi, ali ne znamo gde je sledeće korišćenje, i ažurirano tabelu simbola da je u mrtvo, a a i c živi sa sledećim korišćenjem u 2.

Konačno prelazimo na prvu četvorku. Iz tabele simbola kačimo da je t živo sa sledećim korišćenjem u 3, da je a živo sa sledećim korišćenjem u 2, da je b živo, ali ne znamo gde je sledeće korišćenje; sada ažuriramo da je t mrtvo, a da su a i b živi sa sledećim korišćenjem u 1. Ovo je ujedno finalni rezultat, znači anotirali smo sve promenljive koje se pojavljuju u međukodu, a ovaj finalni sadržaj ove tabele simbola odgovara stanju na ulazu u bazični blok.

Pre nego što pređemo na generisanje koda, da se upoznamo sa strukturama podataka i samim algoritmom. Generator koda koristi deskriptor registra za svaki registar i deskriptor adresa za svaku promenljivu. Deskriptor registra sadrži spisak imena promenljivih čije vrednosti se nalaze u pojedinim registrima i na početku bloka uvek podrazumeva da su registri prazni. Znači za određeni registar recimo AX, tu mogu biti na primer u nekom trenutku vrednosti promenljivih a, b i c. Zašto može više od jedne promenljive? Pa jasno, zato što postoje naredbe dodele X = Y i tme slično, znači jedna vrednost može da se odnosi na više promenljivih.

Primer desktiptora registara: AX : a, b, c

Deskriptor adresa pokazuje u kojim sve registrima i memorijskim lokacijama se nalazi vrednost neke promenljive. Na početku bloka sve korisničke promenljive su u memoriji, znači to je isto pretpostavka između blokova, da uvek korisničke promenljive moramo čuvati u memoriji. Recimo, vrednost promenljive a može se nekom datom trenutku naći u registrima AX i BX, mada je to relativno mala šansa i u nekoj memorijskoj lokaciji a'. Memorijske lokacije ćemo označavati simbolički, imenom promenljive pa ovaj dodatni superskript, da nam sugeriše koja je promenjiva i da je reč o memorijskoj lokaciji.

Primer deskriptora adresa: a : AX, BX, a'

Sam algoritam generisanja koda formulisan je za pojedinačne četvorke. Recimo, i-ta četvorka x = y op z. Prvo, postoji poseban deo algoritma, odvojen u ovaj potprogram getreg i njegov zadatak je da odredi lokaciju za destination, to je lokacija L u koji će biti smešten rezultat x.

Algoritam generisanja koda za i) x := y <u>op</u> z

1. Poziva se potprogram getreg da bi se dobila lokacija L u koji će biti smešten rezultat x. Obično funkcija vraća registar, a ređe fizičku lokaciju operativne memorije
2. Ako y nije u L onda se generiše mašinska instrukcija:
   MOV L, y'
   gde je y' tekuća lokacija za y (najbolje ako je neki registar).
3. Generiše se mašinska instrukcija
   <u>op</u> L, z'
   gde je z' tekuća lokacija za z (najbolje ako je neki registar).
   Dodati L u adresni deskriptor za x. Ako je L registar, dodati x u reg. deskriptor za L.
4. Ako y i z nisu živi u i-toj četvorci, udaljiti ih iz deskriptora registara u kojima se trenutno nalaze.
5. Posle procesiranja svih četvorki, na kraju bazičnog bloka, generisati instrukciju:
   MOV m, R
   za sve promenljive koje su žive, a nalaze se isključivo u nekom od registara R.

Funkcija getreg, samo joj ime kaže, preferentno vraća registar, a ređe fizičku lokaciju operativne memorije, videćemo kasnije detalje. Pošto je taj destination ujedno i levi operand, ako levi operand y nije u L, onda se generiše pripremna mašinska instrukcija dovođenja u L vrednosti toga y, gde je y' tekuća lokacija za y. Inače, ako može da se iskoristi registar, bolje registar, ali ako nije registar onda može i memorija, U trećem koraku, generiše se sama mašinska instrukcija čiji opkod odgovara ovoj operaciji, gde je destination L, a izvorišni operand je zadat adresiranjem z', gde je z' tekuća lokacija za z, preferetno registar. Posle samog generisanja ovih naredni, potrebno je ažurirati adresne i registarske deskriptore, tako što se L doda u adresni deskriptor za x, da reflektuje činjenicu da se vrednost x sad čuva L. Takođe ako je L registar, treba x da se doda u registarski deskriptor za L. Ako y i z nisu živi u i-toj četvorci, tu su nam potrebne informacije o životu promenljivih, udaljiti ih iz deskriptora registara u kojima se trenutno nalaze. Prvih pet koraka se

obavlja za svaku četvorku, a korak 6 na kraju bazičnog bloka, posle obrade svih četvorki. Potrebno je sagledati da li postoje promenljive, na izlasku iz bazičnog bloka, koje su žive, a nalaze se isključivo u registru, nisu još u memoriji. Potrebno je znači generisati za svaku od takvih promenljivih dodatnu instrukciju čuvanja njene vrednosti iz registra u memoriju, znači v' je memorijska lokacija za promenljivu v.
Potprogram getreg na ulazu od podataka koristi tekuću četvorku za koju se generiše kod, dodatnu indikaciju da li je neophodan registar, znači da li mora da vrati registar, ili može i memoriju. Na izlazu getreg daje registar R ili memorijsku lokaciju za ovaj levi operand y.

Getreg (četvorka x:=y OP z , fleg da li je neophodan registar):izlaz: registar R ili mem. lokacija za y

1. Ako je y jedina promenljiva u nekom registru R i y je mrtvo u posmatranoj četvorci, izbaciti R iz adr. deskriptora za y i vratiti R
2. Ako (1) ne uspe, vratiti prazan registar (ako ima takav).
3. Ako (2) ne uspe, i ako x ima sledeću upotrebu u tom bloku ili ako je op operator koji obavezno zahteva registar (npr. indeksiranje) pronaći neki zauzeti registar. Sa MOV upisati u memoriju vrednosti svih promenljivih iz R koje nisu u mem., izbaciti R iz adr. deskriptora tih promenljivih i vratiti R.
4. Ako tačka (3) ne uspe, vratiti memorijsku lokaciju za x.

Algoritam je formulisan u četiri tačke, one idu po nekom prioritetu, odnosno tačka jedan označava najpovoljniju, najoptimalniju situaciju, a onda znači polako klizimo, ako ne mogu da se ispune preduslovi za jedan, idemo na dva, pa ako ne može ni dva, na tri, pa ako ne može tri idemo na četiri. Znači najbolja situacija je da je y jedina promenljiva u nekom registru i da je y mrtvo u posmatranoj četvorci. Znači više nam neće biti ta vrednost potrebna, onda možemo osloboditi R, odnosno, možemo ga vratiti kao izlaz i izbaciti R iz adresnog deskriptora za y, znači osloboditi ga za naredna korišćenja i vratiti R. Znači to je najpovoljnije, da iskoristimo neki već zauzet registar koji može da se oslobodi od y-ona, posle toga će da se koristi naravno za rezultat x. Međutim ako svi ovi preduslovi nisu ispunjeni, onda pokušavamo da nađemo i vratimo prazan registar, ako takav ima. Naravno, taj prazan registar moraće onom pripremnom naredbom da se napuni sa y. Ako ne može ni prazan registar da se nađe, a x ima sledeću upotrebu u tom bloku, ili ako ovaj operator obavezno zahteva registar, mora da se pronađe neki zauzeti registar i onda sa jednom ili više mov naredbi potrebno je osloboditi registar od onih promenljivih koje su u tom registru, a nisu u memoriji. Znači to ćemo videti iz deskriptora toga registra. Kada sve te promenljive sačuvamo u memoriji, izbacujemo ih iz tog registarskog deskriptora, oslobadjamo registar i vraćamo taj, sada slobodan, registar. Konačno, tačka četiri, ako ne može ni tačka 3, onda se vraća memorijska lokacija za x.
Sada ćemo ilustrovati na onom našem primeru sa četiri četvorke, kako da primenimo ovaj jednostavan algoritam. Na početku vidimo da su registri prazni, a da su korisničke promenljive u memoriji. Fokusirano se na prvu četvorku. Pozivom getreg potrebno je dobiti lokaciju za smeštaj levog operanda a. Uvidom u deskriptore vidimo da je a u memoriji i da su svi registri prazni, znači tačka 1 ne može da uspe, ne možemo osloboditi nijedan registar. Ali tačka 2 može da uspe. Inače pretpostavka pod kojom ćemo primeniti algoritam je da ćemo koristiti samo dva registra AX i BX. i ovde ona informacije o životu promenljivih, da ne bi se vraćali na onaj prethodni slajd, podvukao sam znači koje su mrtve, to su ova ovde t i ova v i u. Sve druge promenljive su žive. Tako da znači prva tačka getreg procedure ne uspeva, ali druga tačka getreg će nam vratiti slobodan registar, to je registar AX.

četvorke	mašinski kod	Deskriptori registara registri prazni	Adresni deskriptori a, b, c i d u memoriji
t := a - b	**MOV** AX, a' **SUB** AX, b'	AX sadrži t	t u AX, a, b, c u memoriji
u := a - c	**MOV** BX, a' **SUB** BX, c'	AX sadrži t BX sadrži u	t u AX, a, b, c u memoriji u u BX
v := t + u	**ADD** AX, BX	AX sadrži v BX sadrži u	u u BX, a, b, c u memoriji v u AX
d := v + u	**ADD** AX, BX	AX sadrži d	d u AX, a, b, c u memoriji
	MOV d', AX	registri prazni za novi bazični blok	a, b, c u memoriji

Potom moramo pripremnom naredbom dovući vrednost A u registar AX i obaviti operaciju konkretno znači oduzimanja. Za drugi operand moramo naći gde se nalazi. On je samo u memoriji tako da koristimo memorijsku lokaciju. Posle ove naredbe, moramo ažurirati deskriptore. AX će sadržati rezultat t i naravno za svaku promenljivu moramo reći gda je koja, znači osim t, ove druge su u memoriji.

Zatim, za sledeću naredbu, opet nam je potrebno za levi operand da odaberemo registar. Ne može biti ispunjen prvi slučaj algoritma da oslobodimo neki registar u kome se već nalazi a, pošto se a ne nalazi u registru. Drugi slučaj može da uspe, a to je da vratimo slobodan registar, a to je registar BX koji još imamo na raspolaganju kao slobodan. Znači moraćemo imati pripremnu naredbu da dovučemo vrednost a u registar BX i potom da obavimo opet ovu glavnu operaciju, a to je oduzimanje od BX ovog drugog operanda. c se nalazi samo u memoriji tako da moramo iskoristtii memorijsku lokaciju. Posle generisanja naredbe, ažuriramo deskriptore, znači sad od ranije imamo da je t u AX. Ovo što je sad novo, to je da je u (rezultat) u BX i znači ovde za promenljivu u naznačuejmo da se njena vrednost nalazi u registru BX.

Za treću četvorku, moramo sa getreg da dobijemo lokaciju za ovaj levi operand. Levi operand, kao što vidimo, nalazi se u registru, drugo mrtav je u ovoj tački, znači ta vrednost kada je iskoristimo više neće biti korišćena i to je onaj najpovoljniji slučaj algoritma da možemo da iskoristimo registar u kome se, pošto se ništa drugo ne nalazi osim t u tom registru. Znači iskoristićemo registar Ax i getreg vraća registar AX. Pošto se levi operand već nalazi u tom registru, nije potrebna nikakva pripremna naredba, već samo ova osnovna operacija, add AX. Drugi operand u vidimo da se nalazi u registru, tako da je i to je povoljno, iskoristićemo registar BX. Sada ažuriramo deskriptore, znači sada vidimo da, jer smo naravno izbacili vrednost t iz AX, a rezultat v je upisan u AX. Drugo se nije promenilo od ranije, samo što ovde sad znamo da je ve u AX, t više ne figuriše jer je mrtvo, znači nemamo tu vrednost.

Konačno poslednja četvorka. Potrebna nam je lokacija za levi operand v, vidimo da je v u registru i takođe da je v mrtvu, tako da opet možemo vratiti registar AX i osloboditi ga od v. Ne treba nam pripremna naredba, odmah emitujemo sabiranje. Drugi operand u sadrži se u registru BX. Sada ažuriramo deskriptore, znači sad je rezultat d u AX, u možemo izbaciti zato što je mrtvo posl ove naredbe, znači više nam nije bitna ta vrednost, zato ga nema više u BX. Evo vidimo pošto su sve privremene promenljive mrtve, njih više i ne pominjemo, nego ovde samo ove korisničke promenljive. Znači d je u registru ove druge su u memoriji i sada dolazi ona tačka 6 algoritma,

znači šta raditi na kraju bazičnog bloka. Proveriti da li neka od tih promenljivih koje žive na izlasku, a to je korisničko promenljiva d, nije još u memoriji, znači moramo da sačuvamo vrednost d iz AX u memoriju.
I tako dalje, generišemo još tu, da kažem, naredbu pospremanja i posle ažuriranja deskriptora dobijamo stanje kao na samom početku, da su svi registri prazni, a A, B,C i d su u memoriji.
Ovaj slajd je više informativan u smislu da nismo takve zadatke davali na ispitu, ali moguće je naravno i za ove druge vrste naredbi međukoda definisati kako se generiše kod. Za razne naredbe međukoda koje se najčešće dobijaju pri prevođenju indeksiranja nizova, znači a = b[i], ona će koristiti neke vrste indeksnih adresiranja. Ovde su date opcije u zavisnosti od toga da li je indeks i u registru, ili je Index i u memoriji, ili je i lokalna promenljiva na steku sa pomerajem d. Za b se podrazumeva da je neka memorijska lokacija, početna adresa tog niza.

Pravila generisanja koda za pristup elementima niza:

Naredba	i je u registru R		i je u memoriji M		i je na steku, pomeraj di	
	Kod	Cena	Kod	Cena	Kod	Cena
a:=b[i]	MOV R', [b+R]	2	MOV R, M MOV R, [R+b]	4	MOV R, [BP+di] MOV R, [R+b]	4
a[i]:=b	MOV [a+R], b	3	MOV R, M MOV [R+a], b	5	MOV R, [BP+di] MOV [R+a], b	5

Ovde su date i nekakve cene. Obično generatori koda kad biraju između više mogućnosti, znači to u onom našem jednostavnom algoritmu nije figurisalo, ali realni generatori koda, ako ima više mogućnosti da izabere neku naredbu, svakoj naredbi je dodeljena neka cena koliko košta njena primena, odnosno niža cena znači da je naredba efikasnija, da ima manje pristupa memoriji, manje procesorski ciklusa za izvršavanje. Vidimo da kada je i već u registru, onda možemo iskoristiti indeksno adresiranje da izračunamo b+i, i naravno, potreban je još jedan registar r' za smeštaj rezultata. Ako je i u memoriji, moramo pripremno prvo dovesti i iz memorije u registar, a onda isto iskoristi registarsko adresiranje, pri čemo ovde vidimo da može odmah da se iskoristi taj isti registar, jer on je samo privremeno upotrebljen za dohvatanje b[i]. I naravno ako je i na steku, onda moramo putem baznog adresiranja da dohvatimo i. Naravno opet moramo da iskoristimo neki registar za to i onda opet se koristi ova osnovna naredba da upišemo u ovo a. Postoji varijanta kada je obrnuto A[i]=b i slično se analizira. Što se upotrebe pokazivača tiče, ako preko nekog pokazivača p pristupamo nekoj lokaciji i upisujemo u a, onda ćemo koristiti, ako je p već u nekom registru, registarsko indirektno adresiranje. Ako je p u memoriji, moramo prvo da ga dovedemo u registar, pa onda koristimo registarsko indirektno, ili ako je na steku moramo ga isto tako dovesti u registar, pa onda koristiti registarsko indirektno adresiranje. U ova druga dva slučaja imamo i tu pripremnu naredbu.

Pravila generisanja koda pri upotrebi pokazivača:

Naredba	p je u registru R		p je u memoriji M		p je na steku,pomeraj dp	
	Kod	Cena	Kod	Cena	Kod	Cena
a:=*p	MOV a, [R]	2	MOV R, M MOV R, [R]	3	MOV R, [BP+dp] MOV R, [R]	3
*p:=a	MOV [R], a	2	MOV R, M MOV [R], a	4	MOV R, [BP+dp] MOV [R], a	4

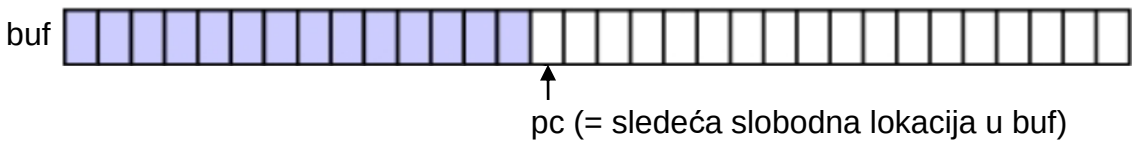
Generisanje mikrojava koda

U ovom poglavlju biće reči o tome kako generisati kod u kompajleru za mikrojavu. Za podršku generisanju koda postoji bibliotečka klasa Code čiju implementacija je javno dostupna na sajtu kursa navedenom u uvodnom delu. Klasa Code namenjena je da se, koristeći javne metode put, put2 i put4 generiše kod i smešta u jednu internu strukturu – bafer buf, koja je u stvari niz bajtova. Interno postoji jedan brojač (Program Counter) PC, koga ne treba naravno mešati sa PCem virtuelne mašine kada se bajtkod izvršava, PC je prosto jedan interni brojač dokle se popunio bafer buf.

```
class Code {
  private byte[] buf = new byte[3000];
  public int pc = 0;

  static final int
    load = 1, load0 = 2, load1 = 3, load2 = 4, load3 = 5,
    store = 6, store0 = 7, store1 = 8, store2 = 9, store3 = 10,
    getstatic = 11,
    ... ;

  public static void put (int x); {
    buf[pc++] = (byte)x;
  }
  public static void put2 (int x) {
    put(x >> 8); put(x);
  }
  public static void put4 (int x) {
    put2(x >> 16); put2(x);
  }
}
```



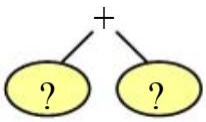
Svaki put kada se emituje sa put jedan ili više bajtova bajtkoda, smeštaju se od lokacije na koju ukazuje PC, a potom se PC ažurira. Metodi put, put2 i put4 redom, služe da u bafer buf ubace 1, 2 ili 4 bajta sadržaja. Da bi se olakšalo kodiranje mikrojava naredbi, postoji niz definisanih simboličkih konstanti u klasi Code koje odgovaraju operacionim kodovima mikrojava naredbi onako kako su one definisane u specifikaciji mikrojave. Ovo je jedan primer, kako se recimo ova jednobajtni metod put koristi:

```
Code.put( code.load0 + 2)
```

emituje bajtkod instrukciju load2, veličine jednog bajta, za koju znamo da služi za dohvatanje druge lokalne promenljive i smeštanje na vrh e-steka.

Pristup operandima

Počećemo od objašnjenja kako generisati kod za mikrojava izraze, za uzimanje vrednosti operanada u izrazima. Ako krenemo od nekog izraza ili dela izraza, koji predstavlja sabiranje dve vrednosti u mikrojavi, potrebno je izgenerisati kod koji učitava prvi operand na e-stek, zatim učitava drugi operand na e-stek i onda se generiše naredba add.



Treba generisati kod nalik na:
učitaj operand1 na estek
učitaj operand2 na estek
add

U zavisnosti od vrste operanda, menja se i kod koji treba da se generiše. Ako je operand konstanta, onda treba mikrojava naredba const sa vrednošću konstante da se izgeneriše. Ako je u pitanju lokalna promenljiva, naredba treba da je load i adresa lokalne promenljive. Ako je u pitanju globalna promenljiva, potrebno je generisati getstatic. Ako je u pitanju polje objekta klase, treba generisati getfield. Ako je u pitanju element niza, treba naredba aload. Ako je vrednost već na e-steku i samo je treba iskoristiti, onda naravno ne treba generisati ništa. Da bi kompajler znao o kojoj vrsti operanda je reč, on će gledati zapis o objektu iz tabele simbola.

vrsta operanda	*instrukcija koju treba generisati*
konstanta	**const val**
lokalna promenljiva	**load a**
globalna promenljiva	**getstatic a**
polje objekta klase	**getfield a**
element niza	**aload**
vrednost na esteku	--

Zbog toga postoji u klasi Code jedan pomoćni metod koji se zove load i koji prima kao svoj stvarni parametar taj objektni zapis na ulazu, a efekat pozivanja toga metoda je emitovanje odgovarajuće bajtkod instrukcije za učitavanje vrednosti na e-stek. Ovaj metod dosta olakšava generisanje koda za pristup operandima izraza, samo ga treba na odgovarajućem mestu tokom semantičke obrade pozvati. U nastavku je prikazana implementacija toga metoda, koji kao što smo rekli predstavlja deo bibliotečke klase Code, da bismo se upoznali sa time kako funkcioniše ovaj metod.

U metodu load, na bazi vrste kind objekta o imamo switch razgranatu strukturu. Za svaku vrstu objekata imamo odgovarajuću case granu, gde možemo videti kako dalje u metodu se koriste ove put naredbe, da se emituju u bafer koda, kako opkodovi, tako i operandi odgovarajućih instrukcija. Recimo kad gledamo slučaj Obj.var, dodatno mora da se proveri level, znači da li je u pitanju globalna ili lokalna promenljiva. Vidimo, na primer, ako je level jednako 0, da se emituje operacioni kod getstatic, a da se u objektu Obj polje adr koristi da se emituje adresa te globalne konstante i vidimo (to mora naravno da odgovara mikrojava specifikaciji) da je to na širini dva bajta. I tako dalje, vidimo na primer i dalje u toj grani, Obj.var, ako nije globalna nego lokalna

konstanta, onda se dodatno proverava da ne pristupamo nekoj od prve četiri lokalne promenljive, pa se onda emituje posebno efikasan operacioni kod, koji odgovara mnemoniku load zajedno sa konstantom da bi se izvršavalo brže. A ako nije jedna od prve četiri konstante, onda se emituje onaj drugi, sporiji load i još se kao drugi bajt naredbe emituje adresa konstante.

```
public static void load (Obj o) {
    switch (o.kind) {
        case Obj.Con:
            if (o.type == Tab.nullType) put(const_n + 0);
            else loadConst(o.adr);
            break;
        case Obj.Var:
            if (o.level==0) // global variable
                 { put(getstatic); put2(o.adr); break; }
            if (0 <= o.adr && o.adr <= 3) // local variable
                 put(load_n + o.adr);
            else { put(load); put(o.adr); }
            break;
        case Obj.Fld:
            put(getfield); put2(o.adr); break;
        case Obj.Elem:
            if (o.type.kind == Struct.Char) put(baload); else put(aload);
            break;
        default:
            error("Greska: nelegalan operand u Code.load");
    }
}
```

Ključno, i ono što nas u stvari najviše zanima, jeste gde smestiti poziv metoda Code.load. Da bismo razumeli kako tačno funkcioniše gramatika za mikrojava izraze, u nastavku ćemo analizirati deo gramatike mikrojava jezika koji se odnosi na izraze.

Prevođenje izraza

Priličan broj smena u mikrojava gramatici odnosi se na izraze. Te smene, na neki način, imaju hijerarhiju. Na dnu te hijerarhije je smena koju vidimo dole u nastavku za ident_expression_list, gde na desnoj strani imamo terminal ident, to je token koji dolazi iz skenera i koji u sebi nosi Id atribut, ime identifikatora koji je upotrebljen u izrazu. To može biti, ako se iskoristi ova smena, nekakva lokalna ili globalna promenljiva. Počevši od te donje smene, tu treba na bazi imena pozvati metod find tabele simbola da bi se našao odgovarajući objektni zapis. Ova varijanta odgovara slučaju da se semantičke akcije ugrađuju direktno u parser u gramatiku, ali može lako da se preradi i za varijantu kada se koristi apstraktno sintaksno stablo. Za tu drugu varijantu čitalac se upućuje na sajt kursa na ETFu čiji je link dat u uvodnom delu, da pogleda primer mini domaćeg koji je detaljno razrađen u vežbama i u prilogu vezanom za laboratoriju. Ovde se zbog konciznijeg zapisa koristi varijanta kada su odgovarajuće semantičke akcije ubačene direktno gramatiku.

```
factor ::= designator:o
           {: ... // semantičke provere videti u primeru mini domaćeg
               Code.load(o); RESULT=o.type;
           :}
designator ::= ident_expr_list:o
           {: RESULT=o; :};
```

```
ident_expr_list ::= IDENT: id
            {:  // find prijavljuje grešku u slučaju da ne nadje id
                Obj ob=Tab.find(id, idleft);
                RESULT=ob;
            :}
```

Iz gornje smene vidimo da, kada nađemo u tabeli simbola objekat ob, preko promenljive RESULT, dodeljujemo neterminalu identifier_expression_list taj objekat i u suštini taj objekat se dalje prenosi, vidimo u smeni iznad, designator se menja sa identifier_expression_list i sad taj identifier_expression_list ima atribut o koji se onda prosleđuje u RESULT, odnosno sada to postaje atribut designator-a. U sledećoj smeni iznad, factor se menja sa designator, dolazi kao atribut o designator-a referenca na taj nađeni objekat.

Treba uočiti da je ovaj neterminal designator značajan po tome što on učestvuje i na levoj strani iskaza dodele i na desnoj. Tako da u njegovim smenama ne znamo da li treba učitavati operand (bajtkod instrukcijom load ili sličnim), ili se designator koristi u kontekstu odredišta, onda u stvari treba da generišemo store i njoj sličnu mikrojava instrukciju. Zato na nivou designator-a ne generišemo nikakav kod, nego dolazimo do smene za factor.

Factor je prvi od neterminala za izraze, a hijerarhijski iznad njega su fakctor_list, term, term_list, expression i expression_list ali factor je prvi za koji znamo da se odnosi na izraze koji se pojavljuju samo na desnoj strani iskaza dodele. Tako da u smeni za factor smemo da generišemo bajtkod za učitavanje operanda na e-stek, tako da upravo u smeni za factor, sa prosledjenim objektom, zovemo Code.load. U tom trenutku se generiše mikrojava instrukcija koja dohvata odgovarajući operand. Vdimo ovde još dodatnu semantičku akciju RESULT = o.type. Zašto je to potrebno? Kao što smo već promenuli, iznad factora u izrazima koriste se razni neterminali factor_list, term i tako dalje. Očigledno i ti neterminali imaju neku semantičku vrednost, neki atribut i to je u stvari tip promenljive, u smislu mikrojava tipa podataka.

Zašto je potrebno da oni imaju tip, ako smo već izgenerisali kod za pristup vrednosti? Potrebno je da radimo semantičke akcije prepoznavanja da li je pravilan tip u izrazima, jer sada, kada se ovaj factor upotrebi u nekom složenijem izrazu, recimo, sabiranju dve vrednosti, kompajler mora na nekom mestu imati proveru da li se one poklapaju po tipu, da li se smeju sabirati i tako dalje. Zbog toga znači u više nivoe smena prosleđujemo tip objekta. Sada ćemo analizirati semantiku smena za mikrojava izraze koje su više u hijararhiji (i stablu izvođenja) od factor-a, kao što su factor_list, term i term_ list. U nastavku je prikazan fragment mikrojava gramatike koja sadrži niz tih smena iznad factora. Nisu navedene sve gramatičke smene za određene neterminale, jer ilustrujemo semantičku obradu za samo neke osnovne tipove izraza, a kasnije ćemo posebno pomenuti još dodatne vrste izraza i odgovarajuće dodatne smene.

```
term_list ::= term_list: t1 addop: op term: t2
            {: // semantička provera kompatibilnosti tipova operanada t1 i t2
                Code.put( op );
                RESULT=t1;
            :}
          | term:t {: RESULT=t; :} ;

addop ::= PLUS {: RESULT=Code.add :}
          | MINUS {: RESULT=Code.sub :};

term ::= factor_list:t {: RESULT=t; :};

factor_list ::= factor:t {: RESULT=t; :};
```

Ovde je Ilustrovano za primer addop, operacija sabiranja i oduzimanja. Na nivou smene *term_list → term list addop term* potrebno je ugraditi semantičku proveru kompatibilnosti tipova operanada t1 i t2, jer tipovi objekata dolaze kroz ove t1 i t2 atribute. Umesto komentara u akciji, treba staviti pravi Java kod, a zatim ide generisanje koda za računanje vrednosti izraza, to je Code.put(op), op se koristi Code.add konstanta ili Code.sub konstanta, u zavisnosti od operacije. Ovaj Code.put emituje mikrojava naredbu add ili sub koja nema svojih argumenata, tako da se tu vrši generisanje koda za sabiranje ako je upotrebljena ova smena za sabiranje. Dalje prosleđujemo semantički atribut term_list-a na levoj strani, to je RESULT koji preuzima vrednost od levog operanda. Taj zbir, taj međurezultat kada se dalje upotrebljava u složenijim izrazima, takođe je podložan proveri tipa i on preuzima tip svog levog operanda, jer ako je ovde ispravno prošao semantičku proveru, onda u suštini oba operanda će imati isti tip, tako da je dozvoljeno proslediti tip bilo kog od operanada.

Pristup polju objekta klase

Sada ćemo videti neke specijalne slučajeve generisanja koda za dohvatanje operanada izraza, a to je recimo ako u izrazima upotrebimo objekte klase i njihova polja. Na primer, ako je jedan od operanada u izrazu o.f, onda imamo sledeći fragment mikrojava gramatike koji se odnosi na onaj, najniži u hijerarhiji, deo izraza kada prepoznajemo sam operand.

```
IdentExpr_list ::= IdentExpr_list:o DOT IDENT:id
          {: ...// ovde treba da ide provera uslova da li je o instanca klase
             Code.load(o); // adr o na estek za kasnije getfield ili putfield
             Obj f=Tab.findField(id, o.type); RESULT=f;
          :}
| IDENT:id
          {: Obj o=Tab.find(id, idleft); // find prijavljuje grešku ako nema id
             RESULT=o;
          :}
```

Do sada smo pominjali smenu ident_expression_list → IDENT i ta situacija je odgovarala običnoj skalarnoj promenljivoj, međutim, neterminal ident_expression_list ima i drugu smenu, a to je ova iznad nje: ident_expr_list → ident_expr_list DOT ident. DOT je terminal koji odgovara upotrebi operatora tačke i još jedan IDENT koji je ime polja. Ako upotrebimo polja u izrazima, onda ćemo jedan ili više puta upotrebiti ovu rekurzivnu smenu. Moguća je višestruka upotreba operatora tačka za pristup polju.

Kako ide semantička obrada te situacije? Već smo videli obradu osnovne smene ident_expr_list → IDENT, vrednost atributa id koji nam dostavlja tokenizator, je ime identifikatora u vidu stringa. Ono se koristi pri pozivu metoda find u tabeli simbola, da bi metod find vratio objekat. Ako je u pitanju ovaj izraz o.f, metod find za ovaj deo o vratiće objekat koji je instanca klase i pošto to prosleđujemo putem RESULT-a na nivo ident_expression_list, onda, kada upotrebimo tačku, aktivira se gornja smena i mi kroz neterminal `IdentExpr_list` i njegov atribut o dobijamo objekat koji je nađen pozivom find metoda. Tamo gde piše komentar da treba da se proveri uslov da li je o instanca klase, znači tu umesto tog komentara treba naravno ubaciti kod za proveru koji je tip objekta o, znači da li je u pitanju klasa, a zatim ide generisanje koda za pristup objektu klase. Ta obrada se sastoji od toga da se pozove Code.load koji će staviti na e-stek adresu,

znači neće generisati kompletan getfield ili putfield nego će samo uraditi pripremu za adresu. Sada je ključno da se posle Code.load mora pozvati poseban metod findField u tabeli simbola (što je takođe bibliotečka klasa). sa imenom polja id. Iz ovog primera o.f, id ima vrednost "f", a o.type je tip od o u izrazu o.f. Metod findField ide u tabeli simbola kroz graf klase i traži njeno odgovarajuće polje, tako da mi dobijamo kao rezultat tog findField, ako je uspešan, objektni čvor koji za polje odgovarajuće klase. Vidimo onda da RESULT dobija vrednost tog objektnog čvora i dalje se znači prosleđuje ta vrednost. Tako da će se ponovo zvati na nekom mestu Code.load što smo videli u onim ranijim smenama i onda će tek biti generisana instrukcija bajtkoda, znači ako je Code.load pozvan u prethodno opisanoj smeni za factor emitovaće se getfield, a, kako ćemo kasnije videti, ako je pozvan metod Code.store na nivou odredišta u naredbi dodele emitovaće se putfield.

Pristup elementu niza

Specijalna obrada slična ovoj prethodnoj, je pristup elementu niza u izrazu a[i]. Postoji još jedna posebna smena za neterminal ident_expression_list koju do sada nismo videli, a to je ona koja na desnoj strani ima ident_expressin_list pa iza toga u uglastim zagradama izraz, LSQUARE expr RSQUARE.

```
ident_expr_list ::= ident_expr_list: o LSQUARE
            {:  if (o.type.kind==Struct.Arr) {
                    Code.load(o); // za poc.adr.niza na estek
                    // dalje prenosimo element niza
                    RESULT=new Obj(Obj.Elem, "", o.type.elemType);
                } else ...; // u slučaju greške vratiti noObj
            :}
            expr RSQUARE // ovaj expr je na esteku ostavio indeks elem niza
        |
            IDENT: id
            {:  // find prijavljuje gresku u slucaju da ne nadje
                Obj ob=Tab.find(id, idleft);
                RESULT=ob;
            :}
```

Prema kontekstnim uslovima u specifikaciji mikrojave iz Priloga A, mora se proveriti da li je taj ident_expression_list na desnoj strani tipa niza, a expr, to jest indeks elementa niza, mora biti celobrojnog tipa. To su dve provere koje moraju da se ugrade u semantičke akcije, koje su u akcijama zbog konciznosti navedene samo kao komentari, nije dat njihov kod.

Kako izgledaju semantičke akcije za generisanje koda? Na sličan način kao i za polja objekta, prvo ova osnovna smena za ident_expresion_list nalazi onaj osnovni objekat koji u tabeli simbola reprezentuje niz, to je u stvari ovo a u izrazu a[i] i zatim se aktivira, kada programer u izrazu upotrebi uglaste zagrade, ova gornja smena ident_expression_list → ident_expression_list LSQUARE expr RSQUARE. LSQUARE je otvorena vitičasta zagrada, onda ide prva semantička obrada, potom ide indeksni izraz expr na desnoj strani smene i RSQUARE odnosno zatvorena vitičasta zagrada. Znači u ovoj smeni semantička akcija nije na samom kraju smene, nego kad prođe otvorena vitičasta zagrada. Semantička akcija se sastoji iz toga da ispitamo da li je u ovom prosleđenom objektu, koji dolazi kao atribut o neterminala ident_expression_list na desnoj strani smene, da li je tip toga objekta tip niza. Ako je to u redu, onda pozivamo Code.load koji će, u slučaju kada mu se prosledi ova vrsta objekta, postaviti početnu adresu niza na e-stek.

Posle toga, ovo je zanimljiva situacija, šta predstavlja semantička vrednost leve strane smene RESULT? Ovo je jedina situacija kada van tabele simbola pravimo instancu objektnog čvora, koji se inače koriste u tabeli simbola i interno ih generišu metodi tabele simbola. Međutim mi ovde eksplicitno pravimo instancu Obj posebne vrste, a to je Obj.Elem. On reprezentuje jedan element nekog niza, nema ime, a njegov tip je tip elementa niza, što vidimo na bazi ovoga objekta koji nam je došao, koji je vrste niza. Zašto nam je potrebna ova posebna vrsta objekta koji se inače ne pamti u tabeli simbola, niti smo do sada spominjali vrstu Obj.Elem? Ta vrsta objekta nam je potrebna isključivo u ovoj semantičkoj obradi izraza, da bi nosila informacije o tome da je izraz tipa elementa nekog niza, što pomaže kasnijem (drugom) pozivu Code.load metoda da izgeneriše pravilan kod, znači da umesto običnog load za pristup skalarnoj promenljivoj upotrebi operacioni kod aload za pristup elementu niza.

U semantičkoj akciji gornje smene, else grana se odnosi na situaciju semantičke greške, recimo kada je programer napisao a[i], pri čemu a nije nikakav niz. Neterminal expr koji se takođe pojavljuje na desnoj strani ove smene je na e-steku ostavio svoju vrednost, odnosno inicirao je generisanje koda koji na estek-u ostavlja vrednost za indeks elementa niza. Kada parser završi sa ovim semantičkim akcijama i krene na više hijerarhijske nivoe smena za izraze, emitovan je bajtkod da na e-steku imamo i početnu adresu niza i vrednost indeksa i-tog elementa niza, upravo ono što je potrebno aload i astore instrukcijama. Provera indeksa, znači da li je vrednost *i* u granicama niza, se radi u vreme izvršavanja od strane mikrojava virtuelne mašine (to je sastavni element semantike aload i astore instrukcija). Ovime smo obradili kako generišemo vrednosti izraza na desnim (izvorišnim) stranama raznih iskaza dodele. U nastavku ćemo posvetiti pažnju odredištu iskaza dodele.

Generisanje koda za iskaz dodele

Iskaz dodele je kada napišete promenljiva jednako izraz, odnosno u gramatici imamo smenu statement → designator '=' expr. U zavisnosti od toga šta je designator, u sledećoj tabeli je prikazano kakav bajtkod treba izgenerisati. Kod mora biti takav da se vrednost expr-a ostavlja na e-steku i na kraju sekvence koda ovo što je boldovano, u stvari je deo koji je potrebno izgenerisati u tekućoj gramatičkoj smeni. Italikom je prikazan kod izgenerisan pri prepoznavanju izraza expr.

localVar = expr;	globalVar = expr;	obj.f = expr;	a[i] = expr;
…load expr… **store localVar**	*…load expr…* **putstatic globalVar**	load obj *…load expr…* **putfield f**	load a load i *…load expr…* **astore**

Kada je u pitanju lokalna promenljiva, vidimo da je boldovani kod jedna store naredba. Ako je u pitanju globalna skalarna promenljiva, onda je to putstatic naredba.

Nešto je složenije ako je u pitanju dodela polju neke klase. Onda pripremu za dodelu objektu, označenu podvučeno, moramo izgenerisati pre prepoznavanja expr. Naredba load obj je u stvari

stavljanje na stek adrese objekta na heap-u, jer to mora ići pre vrednosti koja predstavlja desnu stranu dodele, a onda na kraju moramo još izgenerisati naredbu putfield sa ofsetom polja f.

Slično tome i kod nizova. Ako se dodeljuje nekom elementu a[i] vrednost nekog izraza, onda moramo dovesti na stek prvo početnu adresu tog niza, zatim vrednost indeksa, zatim vrednost koja će biti dodeljena elementu i onda samo na kraju treba izgenerisati astore naredbu bez argumenata.

Međutim, nama u stvari tu na neki način malo pomažu, kao što ćemo sada videti, one obrade koje smo već radili na nivou identifier_expression_list. Evo da se fokusiramo konkretno na ovu smenu statement → designator '=' expression.

```
statement ::= designator:o EQUAL expr:t SEMI
            {: ...// provera uslova
                Code.store(o);
            :}
```

Kontekstni uslovi koji pišu u specifikaciji mikrojave, provere koje kompajler mora da uradi, da bi to sve bilo semantički ispravno, je da designator mora biti promenljiva, element niza ili polje objekta klase, a tip izraza expresion mora biti kompatibilan pri dodeli sa tipom designator. Kao što je opisano u poglavlju o mikrojava tabeli simbola, postoji poseban metod u tabeli simbola koji proverava tu kompatibilnost pri dodeli. Semantička obrada ove smene svodi se samo na to da se pozove Code.store za objekat o. Objekat o se dobija kao semantički atribut neterminala designator. Metod Code.store u klasi Code implementiran je da na bazi vrste prosleđenog objekta emituje ovu završnu boldovanu naredbu bajtkoda. dok se podvučeni deo emituje, kao što sam rekao, već pri prepoznavanju ident_expression_list, jer i kada se prepoznaje designator prolazi se kroz te smene na najnižem nivou tako da su te pripremne naredbe već emitovane. Potom se emituje za izraz expr kod koji treba da dovede vrednost na vrh steka i onda se tek prepoznaje smena koju sada razmatramo, koja onda emituje ove store naredbe. Znači redosled prepoznavanja smena obezbeđuje ispravni redosled generisanja bajtkoda.

Skokovi u bajtkodu

Kako obrađujemo skokove u mikrojava bajtkodu? Rekli smo da za bezuslovni skok postoji naredba jump i za uslovne skokove postoji niz naredbi sa različitim relacionim operatorima provera jednakosti i nejednakosti, strogo manje, manje jednako, strogo veće ili veće do jednako i uslovni skok. Kod za uslovni skok mora biti takav da se dovede vrednost jednog operanda na vrh steka, vrednost drugog operanda na vrh steka i onda se emituje kod uslovnog skoka. Skokovi su inače u implementaciji relativni u odnosu na tekući brojač naradbi PC mikrojava virtuelne mašine koji ukazuje na tekuću naredbu koja se interpretira (naredbu skoka).

bezuslovni skok	uslovni skok
jmp offset	*…load operand1…* *…load operand2…* **jxx offset**

Ovde vidimo i razne konstante koje su u Code klasi definisane, za razne uslove.

```
static final int
  eq = 0, // jump on equal
  ne = 1, // jump on not equal
  lt = 2, // jump on less than
  le = 3, // jump on less or equal
  gt = 4, // jump on greater than
  ge = 5; // jump on greater or equal
```

Konstanta jeq kojoj se dodaje kod nekog od ovih operatora daje pravilan bajtkod za pravilan relacioni operator, to je ovaj primer dole desno.

bezuslovni skok	uslovni skok
Code.put(Code.jmp); Code.put2(offset);	Code.put(Code.jeq + operator); Code.put2(offset);

Code.put(Code.jeq + plus operator) daje tačno operacioni kod. Ako se kao operator recimo upotrebi gt, dobija se kao vrednost tog zbira kod za jgt instrukciju. Pomeraj offset je uvek dva bajta, tako da i uslovni i bezuslovni skokovi zahtevaju da se emituje i dvobajtni ofset. Odmah se postavlja pitanje vrednosti ofseta, to jest, adrese odredišta skoka. Ako su u pitanju skokovi unazad, recimo tako realizujemo kod petlji prelaz na sledeću iteraciju, onda je situacija relativno čista. Pošto smo početak petlje u tom trenutku već preveli, mi znamo i možemo zapamtiti početnu adresu.

Međutim, skokovi unapred deluju teži za realizaciju (kod if naredbi i slično), zato što mi znamo da tek kad kompajler obradi taj deo iskaza koji je odredište skoka, onda ćemo znati gde treba skočiti, ali u trenutku kada emitujemo samu instrukcije za skok, nama odredišna adresa još nije poznata. Osnovna ideja je da mi moramo da izgenerišemo kompletnu bajtkod naredbu skoka, znači da emitujemo i opkod i neku praznu vrednost (nula) za odredište skoka. Ali pamtimo adresu te izgenerisane naredbe skoka, to jest preciznije tog njenog dela sa operandima, ta dva bajta gde se nalazi odredište skoka, da bismo kasnije, kada saznamo pravu adresu na koju treba skočiti, primenili takozvanu backpatching tehniku, odnosno da naknadno promenimo taj deo naredbe skoka.

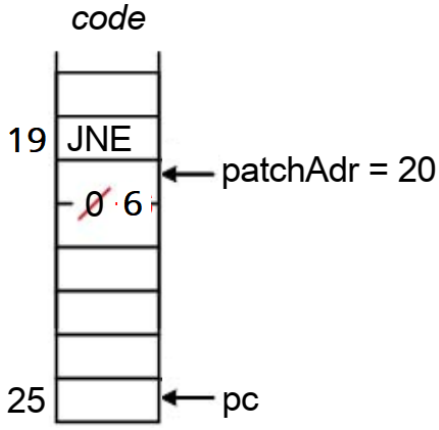
```
class Code {
    private static final int
        eq = 0, ne = 1, lt = 2, le = 3, gt = 4, ge = 5;
    private static int inverse[]= {ne, eq, ge, gt, le, lt};
    // generisanje bezuslovnog skoka na adr
    void putJump (int adr) {
        put(jmp); put2(adr-pc+1);
    }
    // za uslovni skok unapred ostaviti adresu nula
    void putFalseJump (int op, int adr) {
        put(jeq + inverse[op]); put2(adr-pc+1);
    }
    // zakrpiti patchAdr tako da je odredište tekući pc
    void fixup (int patchAdr) {
        put2(patchAdr, pc - patchAdr + 1);
```

```
    }
}
```

Za podršku emitovanju koda za realizaciju skokova, u klasi Code postoje metodi: putJump za generisanje bezuslovnog skoka na odgovarajuću adresu (zadajete apsolutnu adresu putJump metodu, a on interno preračuna koji pomeraj treba da emituje u bajtkod); drugi metod je putFalseJump, on je za uslovne skokove. Njemu zadajete originalni kod uslova u vidu konstante koja je definisana u klasi Code i apsolutnu adresu tog uslovnog skoka. Sam metod će izgenerisati skok za negaciju tog uslova (za to postoji pomoćni niz inverse znači za svaki uslov mapira njegov obrnuti uslov) i emitovaće odgovarajuću relativnu adresu skoka u bajtkod. Zašto metod negira originalni uslov skoka? Često mi znamo određeni uslov, ali treba njegova negacija. Na primer, kada obrađujete if naredbu sa nekim uslovom, u kodu emitujete skok za za ispunjenu negaciju tog uslova, kojim se preskače then grana i ide na else granu.

Za podršku backpatching tehnici, postoji metod fixup koji, ako zapamtimo adresu za prepravku (patch adresu), a to je u stvari trenutni sadržaj polja pc klase Code plus jedan, kada pozivamo metod putFalseJump da generišemo skok unapred. Pre nego što ga pozovemo, Code.pc je za 1 manji pošto jump bajtkod naredba ima tri bajta, a nama treba adresa ofseta iza opkod bajta. Znači pre poziva putFalseJump, ako zapamtimo pc, posle nam treba pc+1 kao patch adresa, Metod fixup treba pozvati u trenutku kad kompajler obrađuje upravo odredišnu adresu tog skoka koji smo već generisali. Tako da fixup-u ne treba davati taj drugi argument, odredišnu adresu skoka, nego on podrazumeva uvek tekući PC kao odredište skoka.

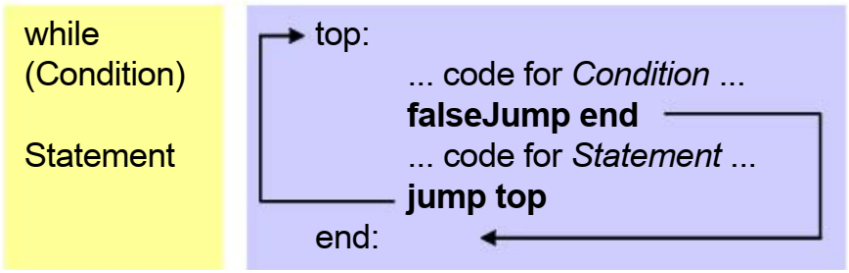
U primeru backpatchinga na sledećoj slici, u baferu koda patch adresa 20 ukazuje upravo iza opkoda skoka i ako se fixup pozove kada je PC na odredišnoj adresi 25, on će upisati iza ovog jne (bez obzira na prethodni sadržaj tih bajtova) rezultat izraza: odredišna adresa manje patch adresa plus 1, konkretno, 25 - 20 + 1 = 6, upisaće 6 iza jne, što će omogućiti da taj uslovni skok, ako je uslov ispunjen, prenosi kontrolu na adresu 25, u vreme kada se bajtkod bude izvršavao u mikrojava virtuelnoj mašini.



Generisanje koda za iskaz while

Kako se generiše kod za neke naredbe toka u višem programskom jeziku, konkretno mikrojavi? Ako u izvornom mikrojava kodu imamo while uslov, pa telo while petlje reprezentovano sa statement, na desnoj strani slike na sledećoj strani vidimo kostur ili šablon bajtkoda koji treba izgenerisati da bi se implemenitirao while. Moramo imati prvo jedno odredište skoka top, zapamćenu lokaciju, koja u stvari predstavlja odredište povratnog skoka kada idemo na novu iteraciju. Zatim treba izgenerisati bajtkod koji će računati uslov. Zatim moramo imati jedan skok

za negaciju uslova koji iskače iz petlje. Zatim će biti generisan kod, obradom statement-a od strane parsera, za implementaciju tela petlje. I na kraju tela petlje treba imati ovaj skok unazad na ovu top lokaciju.

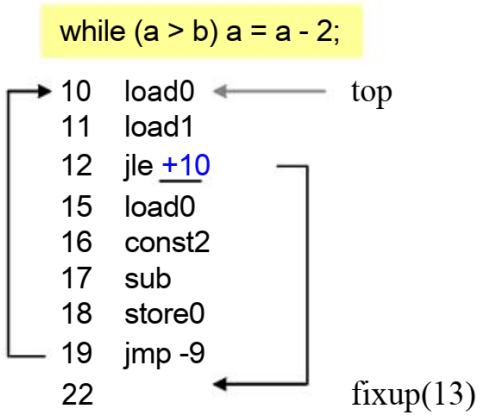


U gramatici imamo sledeću smenu za statement, koja na desnoj strani ima token WHILE, njemu možemo dodeliti semantičku vrednost top i tu vrednost ćemo iskoristiti da pamtimo ovu lokaciju top gde treba kasnije vršiti skok. Zatim imamo otvorenu zagradu, pa onda condition koji je poseban deo gramatike koji opisuje uslovne izraze, hijerarhijski još iznad expression (njegove smene imaju oblik expression logički operator expression).

```
Statement ::=
        WHILE: top            {: top = Code.pc :}
        LPAREN Condition: op  {: Code.putFalseJump(op, 0);
                                adr = Code.pc - 2; :}
        RPAREN
        Statement             {: Code.putJump(top);
                                Code.fixup(adr); :}
```

Taj condition ima svoju semantičku vrednost op koja reprezentuje operator koji treba ugraditi. Pošto mi znamo da je to uslovni izraz, condition smene uparuju neki izraz tipa x op y. Obradom condition, imaćemo na e-steku i x i y, a sam condition će nam kao semantički atribut op vratiti relacioni operator. Odmah posle condition treba pozvati helper metod u klasi Code putFalseJump sa negacijom operatora op i odredištem skoka 0, jer mi u tom trenutku ne znamo gde skačemo. Deo semantičke akcije iza putFalseJump je da zapamtimo adr = Code.pc-2, pošto PC u tom trenutku ukazuje neposredno generisanog skoka koji ima tri bajta, znači ako smanjimo PC za 2 upravo ćemo zapamtiti adresu ona dva bajta koja specificiraju ofset pri skoku, što će kasnije trebati za fixup. Znači promenljiva adr pamti upravo podatak potreban za fixup. Potom u smeni prolazi desna zagrada, zatim prolazi statement koji generiše svoj kod i na samom kraju ove smene, potrebno je emitovati bezuslovni skok, pozivom Code.putJump na ovu zapamćenu vrednost top. Potom treba pozvati fixup sa zapamćenom patch adresom, jer baš u tom trenutku, čim smo emitovali ovaj skok unazad, imamo adresu gde treba da iskočimo iz petlje, znači neposredno iza tog skoka unazad.

U nastavku je dat jedan konkretan primer prethodne priče. Ako je u mikrojavi napisano while A > B, telo petlje je A = A-2, tu će biti izgenerisan redom sledeći kod: deo za condition, kad se bude prepoznavao, izgenerisaće load0 za dovođenje a na vrh e-steka, zatim load1 za dovođenje b na vrh e-steka, zatim će biti aktiviran ovaj metod putFalseJump, vidimo da je uslov strogo veće, a putFalseJump će staviti negirani uslov, odnosno manje do jednako.

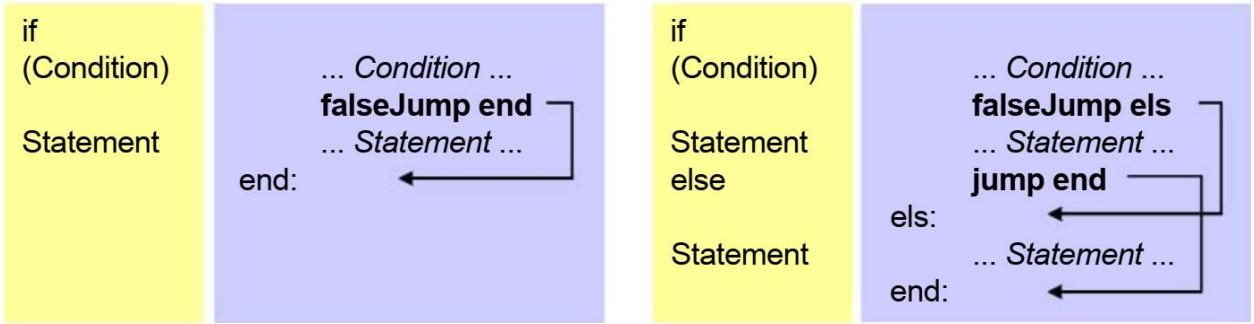


U tom trenutku naravno adresa će biti 0, ovo što je prikazano to je kasnije prepravljena adresa. Posle toga skoka koji predstavlja iskok iz petlje, ide prevođenje tela petlje koje će izgenerisati kod za ovu dodelu a=a-2, znači biće dovedeno a sa load0 na vrh e-steka, konstanta 2, sub naredba oduzimanja i store0, znači upis u a. To je telo petlje i na kraju biće izgenerisano bezuslovni skok unazad na zapamćenu početnu adresu. U bajtkodu su adrese relativne, znači biće ubačen operand -9. PC inače ne ukazuje na prvu adresu posle jump-a nego na sam jump kada se izvršava jump. Tako da PC jednako 19, pa operand -9, daje odredišnu adresu 10, što je u stvari početak ove sekvence koda.

Konačno biće pozvana fixup 13. To znači na lokaciji 13 prepraviti da bude skok na lokaciju 22 relativni, a to znači treba dodati plus 10, jer kad se bude vršio uslovni skok, znači PC je 12 + 10 jednako 22. U generisanju koda važan je pravilan tajming da se na pravo mesto u gramatici ubaci prava semantička akcija.

Generisanje koda za iskaz if

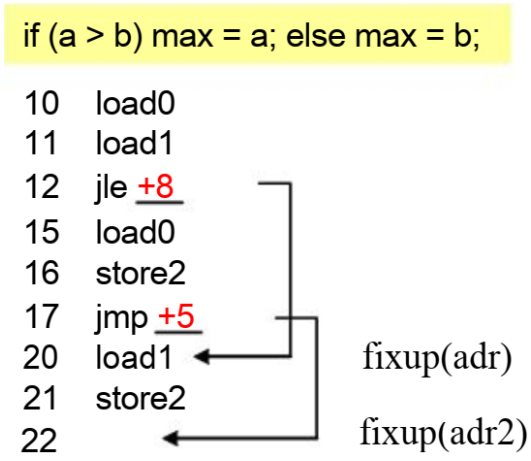
Za iskaz if, Imamo varijantu bez else i varijantu sa else. U varijanti bez else, imamo da se prvo izgeneriše kod za računanje uslova, zatim mora da se stavi jedan false jump koji preskače then deo if-a ako uslov nije ispunjen. To je skok unapred, znači mora d se koristi fixup i onda će statement izgenerisati svoj deo koda.



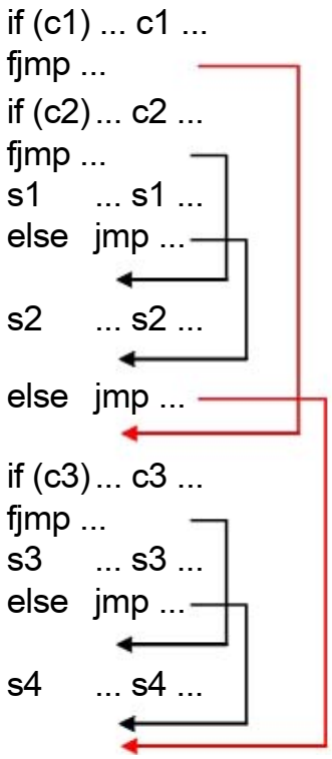
U varijanti kada imamo i else deo, vrlo slično je ovom prethodnom, jedino što posle then grane imamo bezuslovan skok koji će preskočiti else granu. Ovaj false jump će u stvari skakati na početak else grane, a bezuslovan skok će skakati na kraj celog if-a. U nastavku imamo gramatičku smenu koja opisuje if-ove u mikrojava gramatici.

```
Statement ::=
  IF
     LP Condition: op RP {: Code.putFalseJump(op, 0);
                            int adr = Code.pc - 2; :}
     Statement
     ELSE                {: Code.putJump(0);
                            int adr2 = Code.pc - 2;
                            Code.fixup(adr); :}
     Statement           {: Code.fixup(adr2); :}
```

Ovde je prikazana varijanta sa else. znači if, leva zagrada, condition ima svoj op, relacioni operator, koji treba ugraditi u uslovnu naredbu, potom desna zagrada, onda ide statement, to je then deo if-a, onda ide else i statement koji opisuje else deo if-a. Prema ranijem šablonu (desno), condition kada se prepoznaje u smeni, on će izgenerisati svoj deo koda. Potom, čim prođe ova desna zagrada zatvorena, pozivamo putFalseJump i pamtimo adresu fixup, zatim prolazi statement, generiše se ovaj kod za then deo if-a, onda kada prođe else, to je pravo mesto za semantičku akciju da izgenerišemo bezuslovan skok. Moramo takođe da zapamtimo adresu za fixup jer i to je skok unapred. Neposredno iza tog bezuslovnog skoka je odredišta ovog uslovnog skoka (na početak else dela) i tu je pravo mesto da pozovemo prvi fixup, a kad prođe ovaj statement u else delu i kada se izgeneriše kompletan kod, tu je pravo mesto da se pozove fixup za bezuslovan skok na kraju then dela. I ovde imate isto konkretan primer za jednu if naredbu mikrojava programa, kakav kod na osnovu ovih akcija treba da se izgeneriše.



Uvek treba voditi računa znači da if-ovi u programu mogu da se ugneždavaju. Ovako kako smo napisali semantičke akcije, bez ikakvih promena mogu da rade i za ugneždene if-ove. Treba samo voditi računa o ovim zapamćenim fixup adresama, da svaki if ima svoju instancu, da ne bi slučajno pri ugneždavanju desilo da unutrašnji if briše neki podatak iz spoljnog, oprezno koristiti eventualne globalne strukture i tako dalje.



Poziv metoda

Obradićemo poziv statičkih metoda, za dinamičke je primer dat u poglavlju o arhitekturi mikrojava virtuelne mašine. Već smo ranije videli, kada smo pričali o mikrojava virtuelnoj mašini, da za poziv metoda treba staviti stvarne parametre na stek, emitovati naredbu call i posle toga, ako metod vraća neku vrednost, pokupiti tu vrednost sa e-steka.

mikrojava	*bajtkod*
c = m(a+b);	load a
	load b
	call m
	store c

Koji deo gramatike odgovara pozivu metoda? To je deo gramatike za izraze. U izrazu programer može da upotrebi poziv metoda i onda povratnu vrednost metoda koristi u izrazu kao i sve druge vrednosti, znači neterminal factor ima posebnu smenu za pozivanje metoda.

```
factor ::= designator:o LPAREN act_pars RPAREN
            {:  .../// provera da li je o metod i da li ima povratni tip
                int dest_adr=o.adr-Code.pc; // relativna adresa
                Code.put(Code.call); Code.put2(dest_adr);
                RESULT=o.type;
            :}
act_pars ::= expr_list;
expr_list ::= expr_list COMMA expr
                    {: .../* provera broja i tipa stvarnih parametara */ :}
            | expr {: .../* provera broja i tipa stvarnih parametara */ :}
```

To je ova smena koja na desnoj strani ima designator, znači ime metoda koji se poziva, pa otvorena leva zagrada, pa stvarni parametri, pa zatvorena zagrada. Stvarni parametri su lista izraza i imamo posebnu smenu za listu izraza, koja je rekurzivna. Postoje i druge smene koje imaju poziv metoda. Poziv metoda može biti iskaz sam za sebe, pri čemu onda, ako taj metod vraća neku vrednost, ona se zanemaruje, mora samo da se skine sa e-steka.

```
statement ::= designator:o LPAREN act_pars RPAREN SEMI
```

Da se vratimo na prethodno prikazane smene za izraze. Šta u stvari treba sve od semantičkih akcija obraditi? Možemo da krenemo od dna, od ove liste izraza koji predstavljaju stvarne parametre. Treba u tim smenama proveriti broj i tip stvarnih parametra, umesto odgovarajućeg komentara koji stoji u akciji. Moramo na neki način zapamtiti koji smo metod pozvali. U smeni za factor imamo designator na desnoj strani, njegov atribut o je objekat koji je vraćen iz tabele simbola i koji predstavlja ime metoda i ima polje locals za pamćenje formalnih parametara metoda. Kada proveravamo broj stvarnih parametara, moramo znači pristupiti tim informacijama i onda sravniti da li je sve u redu (isti broj parametara i parametar po parametar da li tip formalnog parametra odgovara tipu stvarnog paramentra). Na nivou ove smene za factor, prvo treba na bazi ovoga o atributa proveriti da li je o metod i da li ima povratni tip, jer je upotrebljen u izrazu, znači očekuje se da ima povratni tip. I call method upotrebljava relativnu adresu, ovde to vidimo iz ovog koda, znači izračunamo relativnu adresu tako što od tekućeg pc-ja oduzmemo destination_address, a destination_address dobijamo iz o.adr i onda emitujemo poziv metoda. Takođe, na bazi o.type moramo kroz RESULT vratiti semantički atribut ovog factor-a za kasnije provere tipa.

Obrada deklaracije metoda

Naravno metodi pored poziva imaju i deklaracije u okviru glavnog programa, ili u okviru klasa se definišu metodi klasa. U nastavku su date smene za deklaracije metoda u glavnom programu.

```
method_dec ::= return_type_ident:o LPAREN form_pars:n RPAREN local_var_list
LBRACE
                {:  // generisanje koda
                    if (o.name.equals("main")) {
                        Code.mainPc = Code.pc;
                        if (method_type!=Tab.noType)
                            report_error("metod main mora biti void");
                    }
                    o.level = n; // za metode, broj parametara
                    o.adr = Code.pc;
                    Code.put(Code.enter); Code.put(o.level);
                    Code.put(Tab.topScope.nVars);
                :}
            stmt_list RBRACE
                {:  if (method_type!=Tab.noType && !returnExists) {
                        report_error("Metod mora imati return iskaz "+
                                     "jer nije deklarisan sa void");
                    }
                    returnExists=false;
                    o.locals = Tab.topScope.locals; Tab.closeScope();
                    // generisanje koda
                    if (method_type==Tab.noType) {
                        Code.put(Code.exit); Code.put(Code.return_);
                    } else { // postaviti trap na kraj tela funkcije,
                        // da izazove run time grešku ako se zaobiđe return
```

```
                Code.put(Code.trap); Code.put(1);
            }
        :};
```

Deklaracija metoda na desnoj strani počinje povratnim tipom, znači return_type_ident je posebna smena koja daje i povratni tip i ime metoda. Onda ide otvorena zagrada, pa idu formalni parametri. Lista formalnih parametara se takođe opisuje posebnom smenom. Potom ide zatvorena zagrada, pa onda ide lista lokalnih promenljivih, pa otvorena vitičasta zagrada, pa statement_list, znači to je u suštini izvršno telo metoda, pa zatvorena vitičasta zagrada. To je sama smena, ovo drugo su sve ubačene semantičke akcije.

Što se semantike tiče, čim prođu formalni parametri, lokalne promenljive i leva vitičasta zagrada, a pre statement_list, zgodan je trenutak za ubacivanje semantičke akcije u varijanti kada se direktno koriste semantiče akcije. Iz gornjeg koda vidimo šta su tu sve semantičke akcije: ako je metod baš main method, onda treba zapamtiti, u polju mainPC u klasi Code, vrednost tekućeg PCa, jer to će kasnije biti ugrađeno u objektni fajl. Klasa Code ima i pomoćni metod write za ispisivanje bafera u objektni fajl i on će znači upisati i vrednost ovog polja, odakle kasnije mjvm treba da počne da izvršava program. Metod main mora biti tipa void, tako da postoji i ta semantička provera. U delu mikrojava specifikacije vezanom za semantiku treba videti koje sve provere postoje, tamo je to po smenama i zapisano.

Vidimo dalje da za postrani metod, atribut o neterminala `return_type_ident` daje objektni čvor tog metoda, putem kog će metod biti zapamćen u tabeli simbola. Pošto su već prošli parametri, o.level daje broj parametara. Pošto znamo da će u listu o.locals biti na kraju naređani i parametri i lokalne promenljive, atribut n neterminala `form_pars` nam govori koliko početnih članova liste su parametri. A o.adr treba da je početna adresa tela metoda Tek će slediti generisanje koda, znači pravo vreme je da se zapamti početna adresa metoda iz tekućeg pc-ja u klasi Code.

U poglavlju o arhitekturi mjvm opisano je da telo svakog metoda standardno počinje bajtkod naredbom enter, koja ima dva parametra, broj parametara metoda i veličina njegovog p-stek frejma, prostora za parametre i lokalne promenjive u zbiru. Vidimo kako se oni ovde postavljaju iz o.level i iz broja oblekata u top scope. Trenutno otvoreni opseg je opseg parametara i lokalnih promenljivih metoda, ali on još nije prikačen metodu. Zato ne pristupamo njemu kao o.locals nego još uvek putem topScope u tabeli simbola. Tek kada se zatvori taj opseg, kada prođe cela ova smena, onda će o.locals da se definiše.

Postoji još jedna velika semantička akcija na samom kraju smene, kada se već izgeneriše kompletan kod metoda i prođe zatvorena vitičasta zagrada. Od semantičkih provera, treba proveriti da li je ovo bio metod koji nešto vraća i vidimo da ovde postoji return_exists logička promenljiva, koja je deo parserskog koda i koja se postavlja na nekom drugom mestu, u stvari kada parser obrađuje statement_list, ako naiđe return iskaz, globalna promenljiva return_exists biće postavljena na true, da je u statement_listu-u bila neka return naredba. Znači ako ta promenljiva nije postavljena na true, a metod je deklarisan da vraća neku vrednost, onda je to greška, jer bi morao da postoji return sa nekom povratnom vrednošću. Posle toga, globalnu promenljivu return_exists ponovo postavljamo na false da bi bila spremna za obradu nekog sledećeg tela metoda.

Potom u završnoj akciji imamo ono što je diskutovano u odeljku tabela simbola, da treba pre zatvaranja lokalnog opsega metoda, zapamtiti njegove lokalne promenljive i parametre u o.locals. Ako je metod deklarisan sa void, onda treba izgenerisati samo onaj završni kod kojim se završava

telo svakog metoda, par naredbi exit i return bez parametara, inače, to ne radimo za metode koji nešto vraćaju, jer onda je na programeru odgovornost da svuda stavi eksplicitan return. Jedino kod metoda koji su void, programer ne mora nikakav return da stavi, jer se podrazumeva da na kraju, kada se zatvoriti vitičasta zagrada, to je implicitni return. Tako da u else grani odgovarajuće akcije, ako je metod deklarisan da vraća neku vrednost, kompajler neće eksplicitno generisati retun naredbu. Međutim, generisaće instrukciju softverskog prekida trap, tako da ako je slučajno programer zaboravio da stavi eksplicitni return, u run-time-u će se naleteti na ovaj trap i izdati odgovarajuća poruka.

Obrada formalnih parametara

U deklaraciji metoda koju smo upravo opisali, jedna od povezanih smena su formalni parametri. U nastavku su prikazane smene koje se odnose na deklarisanje formalnih parametara, pri čemu, na najnižem nivou smena, jedan parametar se deklariše tipom type i imenom IDENT parametra. Potom smene za parameter_list opisuju da možemo imati više parametara razdvojenih zarezima.

Potrebno je formalne parametre uneti u tabelu simbola, kao promenljive u lokalnom opsegu metoda. Njihov ukupan broj treba da se ovaj prebroji i vidimo da ova smena na najvišem nivou, formal_parameters ima taj broj n, jer to kasnije nam treba za enter naredbu da bismo znali koju vrednost da joj prosledimo.

```
form_pars ::= parameter_list:n {: RESULT = n; :}
            |
            /* epsilon */ {: RESULT = new Integer(0); :};

parameter_list ::= parameter_list:n COMMA parameter
                   {: RESULT = new Integer(n.intValue() + 1); :}
            |
            parameter {: RESULT = new Integer(1); :};

parameter ::= type: t IDENT: id
                   {: Tab.insert(Obj.Var, id, idleft, t); :}
            |
            type:t IDENT:id LSQUARE RSQUARE
                   {: Tab.insert(Obj.Var, id, idleft, new Struct(Struct.Arr, t)); :};
```

Na nivou smene za parameter vidimo da, kada dobijemo tip t (prepoznavanjem posebnih smena koje vraćaju instancu struct čvora), dok skener uz terminal IDENT vraća ime identifikatora kao string, onda pozivamo insert metodu tabele simbola gde se kroz parametre zadaje da je vrsta objekta promenljiva, da je ime id i da je tip t. Dodatni parametar id.left odnosi se na jedan implementacioni detalj, koji do sada možda nismo pominjali u priči o tabeli simbola, radi se o poziciji, znači broju linije u izvornom programu. Skener kada iz ulaza izdvaja token po token, stalno ažurira tu id.left poziciju u izvornom fajlu, i to se kasnije prosleđuje parseru. Od parsera možete dobiti poziciju za svaki simbol, terminal i neterminal u smeni, jer on to dobija od skenera i pamti te pozicije. Ovde sada prosleđujemo inset metodi da, ako treba da javi neku grešku, onda ona u okviru teksta greške ispiše i tu poziciju da bi programer mogao lakše da se orijentiše gde je nastala greška.

Za parameter postoji i smena za obradu situacije kad je stvarni parametar neki niz. Onda se zadaje tip, ime niza, otvorena zatvorena uglasta zagrada i iz akcije vidimo da se u tabelu ubacuje objekat vrste promenljiva, ali njegov tip t je tip niza.

Iskaz return

U nastavku je prikazana smena za iskaz Return. Vidimo da se u semantičkoj akciji smene pojavljuje globalna promenljiva return_exists, koju, kada se u program pojavi return iskaz, treba postaviti na true. Druga globalna promenljiva koja se u akciji koristi je method_type. Kada počne deklaracija metoda, vrednost povratnog tipa (referenca na Struct čvor tabele simbola) se pamti u method_type i ta vrednost na ovom mestu služi da proverimo da li smo slučajno upotrebili return u metodu koji je deklarisan sa void, to je greška. Ili, ako metod nije deklarisan sa void, pošto je ovo varijanta return sa izrazom expr, onda moramo proveriti da li tip izraza odgovara tipu koji je naveden u deklaraciji metoda. Runtime kod koji se generiše za iskaz return su bajtkod naredbe exit i return.

```
Statement ::= RETURN expr:t SEMI
                {:  returnExists=true;
                    if (method_type==Tab.noType)
                        report_error("metod ne sme imati return sa "+
                                     "izrazom jer je deklarisan sa void");
                    if (!t.assignableTo(method_type))
                        report_error("tip izraza nekompatibilan sa "+
                                     "deklaracijom metoda");
                    Code.put(Code.exit);
                    Code.put(Code.return_);
                :}
```

Ovime je završen prikaz nekoliko karakterističnih vrsta iskaza koji su izabrani da bismo objasnili principe generisanja koda u mikrojava kompajleru. Za dodatne detalje i varijantu upotrebe apstraktnih sintaksnih stabala i vizitora za generisanje koda, čitaoucu se preporučuje da prouči primer mini domaćeg koji se može naći na sajtu kursa iz prevodilaca na ETFu, čiji je link naveden u uvodnom poglavlju.

Prilog A: Specifikacija mikrojave

Programski jezik Mikrojava se koristi u praktičnom delu kursa programskih prevodilaca 1 na Elektrotehničkom fakultetu u Beogradu. Mikrojava je slična Javi, ali je mnogo jednostavnija.

A.1 Opšte osobine jezika

• Mikrojava program počinje ključnom rečju program i ima statička polja, statičke metode i unutrašnje klase koje se mogu koristiti kao (korisnički) tipovi podataka.

• Glavna metoda Mikrojava programa se uvek zove *main*(). Kada se poziva Mikrojava program izvršava se ta metoda.

• Postoje:

- Celobrojne i znakovne konstante (*int, char*).
- Osnovni tipovi: *int, char* (ASCII).
- Promenljive: globalne (statičke), lokalne, klasne (polja).
- Promenljive osnovnih tipova sadrže vrednosti.
- Strukturirani/referencijalni tipovi: jednodimenzionalni nizovi kao u Javi i unutrašnje klase (u nastavku samo klase) sa poljima i metodama.
- Promenljive referencijalnih tipova predstavljaju reference (sadrže adrese koje se ne mogu menjati eksplicitno).
- Statičke metode u programu. Dinamičke metode klasa.

- Svaka klasa može naslediti neku drugu klasu, osim same sebe. Microjava je jezik jednostrukog nasleđivanja.
- Nasleđivanje u MikroJavi ima iste principe kao i u Javi. Objekat potklase je takođe tip njegove superklase. Referenca potklase može se dodeliti bilo kojoj referenci superklase. Čin pretvaranja reference potklase u referencu superklase naziva se konverzija na više (engl. upcast).
- Metodi superklase mogu se redefinisati (engl.override) u njenoj potklasi. Zbog toga se vezivanje metoda unutrašnje klase javlja u vreme izvršavanja, na osnovu vrste objekta. Ovo je polimorfizam (takođe se naziva dinamičko vezivanje ili kasno vezivanje).
- Unutar metoda instance, ime polja odnosi se na polje instance trenutnog objekta, pod pretpostavkom da polje nije skriveno parametrom metode. Ako je skriveno, možemo pristupiti polju instance preko this.fieldName. U telu unutrašnjih metoda this predstavlja referencu na objekat koji je pozvao metodu.
- Preklapanje metoda (engl. overloading) nije podržano u Mikrojavi.
- Za razliku od jave, u mikrojavi ne postoji preddefinisana klasa Object iz koje su sve ostale klase implicitno izvedene.
- Ne postoji garbage kolektor (alocirani objekti se dealociraju samo nakon završetka izvršavanja programa).

- Preddeklarisane funkcije su *ord*, *chr*, *len*.
- Metod *print* ispisuje vrednosti svih osnovnih tipova.
- Od kontrolnih struktura postoji uslovno grananje (if-else), kao i ciklus (while).

Primer mikrojava programa

```
program DeliveryShop
        const int SIZE = 100;
        const int DEADLINE = 7;
        const char ANSWERP = 'Y';
        const char ANSWERN = 'N';
        const Table {
                int pos[];
                int neg[];

        {
                void initialize () int i;
                { // ---------- Initialize Table object
                        pos = new int[SIZE]; neg = new int[SIZE]; i = 0;
                        while (i < SIZE) {
                                pos[i] = 0; neg[i] = 0; i++;
                        }
                }

                void setRating (int time) int x;
                {
                        read(x);
                        if (x >= 0 && x <= SIZE)
                        if (time >=0 && time <= DEADLINE) pos[x] = pos[x] + 5;
                        else if (time < 2*DEADLINE) neg[x] = -3;
                        else neg[x] = -15;
                }

                char permission () int s, j;
                {
                        s=0; j=0;
                        while (j < SIZE) { s = s + pos[j] + neg[j]; j++; }
                        if (s < 0) return ANSWERN;
                        else return ANSWERP;
                }

        }
        }

        class Box  {
                int weight; int width; int height; int depth;
                int unitPrice;
```

```
{
        void Box1 (int weight, int width, int height, int depth, int unitPrice)
        {
                this.weight = weight;
                this.width = width;
                this.height = height;
                this.depth = depth;
                this.unitPrice = unitPrice;
        }

        int volume ()
        {
                return weight*width*height;
        }

        int totalPrice()
        {
                return weight*unitPrice + 10;
        }
}
}

class GiftBox extends Box {
        int kind;
{
        void GiftBox1 (int we, int wi, int he, int de, int up, int k)
        {
                this.Box1(we, wi, he, de, up); kind = k;
        }

        int totalPrice ()          // overridden method int ad;
        {
                if (kind == 1) ad = 5;
                else if (kind == 2) ad = 10;
                else if (kind == 3) ad=15;
                return weight*unitPrice + 10 + ad;
        }
}
}

class Customer {
        int idNum; Table val;
}

{ // Slede deklaracije globalnih metoda
```

```
Table tableConstructor ()
        Table t;
{
        t = new Table; return t;
}

void main ()
        Customer John;
        Box b;
        GiftBox gb;
        int bill;
        int time;
{
        John = new Customer;
        John.idNum = 0;
        John.val = tableConstructor(); John.val.initialize();
        gb = new GiftBox;
        gb.GiftBox1(2, 500, 500, 500, 40, 2);
        b = gb; // upcasting
        if (John.val.permission() == ANSWERP) bill = b.totalPrice(); // polymorphism
        else { print('E'); print('R'); print('R'); print('O'); print('R'); }
        read(time);
        John.val.setRating(time);
}
}
```

A.2 Sintaksa

```
Program     = "program" ident {ConstDecl | VarDecl | ClassDecl} "{"
              {MethodDecl} "}".
ConstDec1   = "const" Type ident "=" (number | charConst) ";".
VarDec1     = Type ident ["[" "]"] {"," ident ["[" "]"]} ";".
ClassDec1   = "class" ident ["extends"] "{" {VarDecl} ["{" {MethodDec1} "}"] "}".
MethodDec1  = (Type | "void") ident "(" [FormPars] ")" {VarDecl} "{" {Statement} "}".
FormPars    = Type ident ["[" "]"] {"," Type ident ["[" "]"]}.
Type        = ident.
Statement   = Designator ("=" Expr | "(" [ActPars] ")" | "++" | "--") ";"
              | "if" "(" Condition ")" Statement ["else" Statement]
              | "while" "(" Condition ")" Statement
              | "break" ";"
              | "return" [Expr] ";"
              | "read" "(" Designator ")" ";"
              | "print" "(" Expr ["," number] ")" ";"
              | "{" {Statement} "}".
ActPars     = Expr {"," Expr}.
Condition   = CondTerm {"||" CondTerm}.
CondTerm    = CondFact {"&&" CondFact}.
```

```
CondFact    = Expr Relop Expr.
Expr        = ["-"] Term {Addop Term}.
Term        = Factor {Mulop Factor}.
Factor      = Designator ["(" [ActPars] ")"]
            | number
            | charConst
            | "new" Type ["[" Expr "]"]
            |"(" Expr ")".
Designator  = ident {"." ident | "[" Expr "]"}.
Relop       = "==" | "!=" | ">" | ">=" | "<" | "<=".
Addop       = "+" | "-".
Mulop       = "*" | "/" | "%".
```

Leksičke structure

Ključne reči:	break, class, const, else, extends, if, new, print, program, read, return, void, while
Vrste tokena:	ident = letter {letter | digit | "_"}. number = digit {digit}. charConst = "'" printableChar "'".
Operatori:	+, -, *, /, %, ==, !=, >, >=, <, <=, &&, ||, =, ++, --, ;, comma, . (,), [,], {, }
Komentari:	// to the end of the line

A.3 Semantika

Svi pojmovi u ovom dokumentu, koji imaju definiciju, su podvučeni da bi se naglasilo njihovo posebno značenje. Definicije tih pojmova su date u nastavku.

Tip reference

Nizovi i klase su tipa reference.

Tip konstante

- Tip celobrojne konstante (npr. 17) je int.
- Tip znakovne konstante (npr. 'x') je char.

Ekvivalentni tipovi podataka

Dva tipa podataka su ekvivalentna
- ako imaju isto ime, ili
- ako su oba nizovi, a tipovi njihovih elemenata su ekvivalentni.

Kompatibilni tipova podataka

Dva tipa podataka su kompatibilna
- ako su ekvivalentni, ili
- ako je jedan od njih tip reference, a drugi je tipa *null*.

Kompatibilnost tipova podataka pri dodeli

Tip *src* je kompatibilan pri dodeli sa tipom *dst*

- ako su *src* i *dst* ekvivalentni,
- ako je *dst* tip reference, a *src* je tipa *null*,
- ako je dst referenca na osnovnu klasu, a src referenca na izvedenu klasu

Predeklarisana imena

Int	tip svih celobrojnih vrednosti
char	tip svih znakovnih vrednosti
null	null vrednost promenljive tipa klase ili (znakovnog) niza simbolički označava referencu koja ne pokazuje ni na jedan podatak
eol	kraj reda karaktera (odgovara znaku ′\n′); print(eol) vrši prelazak u novi red
chr	standardan metod; chr(i) vrši konverziju celobrojnog izraza *i* u karakter (char)
ord	standardan metod; ord(ch) vrši konverziju karaktera *ch* u celobrojnu vrednost (int)
len	standardan metod; lrn(a) vraća broj elemenata niza a

Opseg važenja

Opseg važenja (*scope*) predstavlja tekstualni opseg metode ili klase. Prostire se od početka definicije metode ili klase (odmah posle imena) do zatvorene velike zagrade na kraju te definicije. Opseg važenja ne uključuje imena koja su deklarisana u opsezima koji su leksički ugnežđeni unutar njega. U opsegu se "vide" imena deklarisana unutar njega i svih njemu spoljašnjih opsega. Pretpostavka je da postoji "veštački" globalni opseg (universe), za koji je glavni program lokalan i koji sadrži sva preddeklarisana imena.

Deklaracija imena u unutrašnjem opsegu *S* sakriva deklaraciju istog imena u spoljašnjem opsegu.

Napomena

- Indirektna rekurzija nije dozvoljena i svako ime mora biti deklarisano pre prvog korišćenja.
- Preddeklarisana imena (npr. int ili char) mogu biti redeklarisana u unutrašnjem opsegu (ali to nije preporučljivo).
- Identifikator this nije ključna reč, ali nije preporučljivo koristiti ga, sem u dinamičkim metodama onako kako je objašnjeno.

A.4 Kontekstni uslovi

Opšti kontekstni uslovi

- Svako ime u programu mora biti deklarisano pre prvog korišćenja.
- Ime ne sme biti deklarisano više puta unutar istog opsega.
- U programu mora postojati metoda sa imenom *main*. Ona mora biti deklarisana kao void metoda bez argumenata.

Kontekstni uslovi za standardne metode

chr(e) *e* mora biti izraz tipa int.
ord(c) *c* mora biti tipa char.
len(a) *a* mora biti niz ili znakovni niz.

Kontekstni uslovi za MikroJava smene

Program = ″program″ ident {ConstDecl | VarDecl | ClassDecl} ″{″ {MethodDecl} ″}″.

ConstDecl = ″const″ Type ident ″=″ (numConst | charConst) ″;″.
- Tip terminala *numConst* ili *charConst* mora biti ekvivalentan tipu *Type*.

VarDecl = Type ident [″[″ ″]″] {″,″ ident [″[″ ″]″]} ″;″.
ClassDecl = ″class″ ident [″extends″ Type] ″{″ {VarDecl} [″{″ {MethodDecl} ″}″] ″}″.
- Tip *Type* prilikom izvođenja klase iz druge klase mora biti unutrašnja klasa glavnog programa.

MethodDecl = (Type | ″void″) ident ″(″ [FormPars] ″)″ {VarDecl} ″{″ {Statement} ″}″.
- Ako metoda nije tipa void, mora imati iskaz return unutar svog tela (uslov treba da se proverava u vreme izvršavanja programa).
- Globalne funkcije nemaju implicitan parametar this.

FormPars = Type ident [″[″ ″]″] {″,″ Type ident [″[″ ″]″]}.
Type = ident.
- *ident* mora označavati tip podataka.

Statement = Designator ″=″ Expr ″;″.
- *Designator* mora označavati promenljivu, element niza ili polje unutar objekta.
- Tip neterminala *Expr* mora biti kompatibilan pri dodeli sa tipom neterminala *Designator*.

Statement = Designator (″++″ | ″--″) ″;″.
- *Designator* mora označavati promenljivu, element niza ili polje objekta unutrašnje klase.
- *Designator* mora biti tipa int.

Statement = Designator ″(″ [ActPars] ″)″ ″;″.
- *Designator* mora označavati nestatičku metodu unutrašnje klase ili globalnu funkciju glavnog programa.

Statement = ″break″.
- Iskaz break se može koristiti samo unutar *while* petlje. Prekida izvršavanje neposredno okružujuće *while* petlje.

Statement = ″read″ ″(″ Designator ″)″ ″;″.
- *Designator* mora označavati promenljivu, element niza ili polje unutar objekta.
- *Designator* mora biti tipa int ili char.

Statement = ″print″ ″(″ Expr [″,″ numConst] ″)″ ″;″.
- *Expr* mora biti tipa int ili char.

Statement = ″return″ [Expr] .

- Tip neterminala *Expr* mora biti ekvivalentan povratnom tipu tekuće metode/ globalne funkcije.
- Ako neterminal *Expr* nedostaje, tekuća metoda mora biti deklarisana kao void.
- Ne sme postojati izvan tela metoda, odnosno globalnih funkcija.

Statement = ″if″ ″(″ Condition ″)″ Statement [″else″ Statement].

- Naredba if – ukoliko je vrednost uslovnog izraza *Condition* true, izvršavaju se naredbe u if grani, u suprotnom izvršavaju se naredbe u else grani, ako je navedena.

Statement = ″while″ ″(″ Condition ″)″ Statement .

- Pre izvršavanja tela petlje Statement proverava se zadati uslov. Ukoliko je uslov ispunjen izvršava se Statement, dok se u suprotnom izlazi iz petlje. Posle izvršavanja Statement kontrola se ponovo prenosi na proveru uslova ostanka u petlji.

ActPars = Expr {″,″ Expr}.

- Broj formalnih i stvarnih argumenata metode mora biti isti.
- Tip svakog stvarnog argumenta mora biti kompatibilan pri dodeli sa tipom svakog formalnog argumenta na odgovarajućoj poziciji.

Condition = CondTerm {″||″ CondTerm}.
CondTerm = CondFact {″&&″ CondFact}. CondFact = Expr Relop Expr.

- Tipovi oba izraza moraju biti kompatibilni.
- Uz promenljive tipa klase ili niza, od relacionih operatora, mogu se koristiti samo != i ==.

Expr = Term.
Expr = ″-″ Term.

- *Term* mora biti tipa int.

Expr = Expr Addop Term.

- *Expr* i *Term* moraju biti tipa int. U svakom slučaju, tipovi za Expr i Term moraju biti kompatibilni.

Term = Factor.
Term = Term Mulop Factor.

- *Term* i *Factor* moraju biti tipa int.

Factor = Designator | numConst | charConst | ″(″ Expr ″)″.
Factor = Designator ″(″ [ActPars] ″)″.

- *Designator* mora označavati metodu unutrašnje klase ili globalnu funkciju glavnog programa.

Factor = ″new″ Type ″[″ Expr ″]″.

- Tip neterminala *Expr* mora biti int.

Factor = ″new″ Type.

- Neterminal Type mora da označava unutrašnju klasu (korisnički definisani tip).

Designator = Designator ″.″ ident .

- Tip neterminala *Designator* mora biti klasa (*ident* mora biti ili polje ili metoda objekta označenog neterminalom *Designator*).

Designator = Designator ″[″ Expr ″]″.
- Tip neterminala *Designator* mora biti niz.
- Tip neterminala *Expr* mora biti int.

Relop = ″==″ | ″!=″ | ″>″ | ″>=″ | ″<″ | ″<=″.
Addop = ″+″ | ″-″.
- Operatori su levo asocijativni.

Mulop = ″*″ | ″/″ | ″%″.
- Operatori su levo asocijativni.

A.5 Implementaciona ograničenja

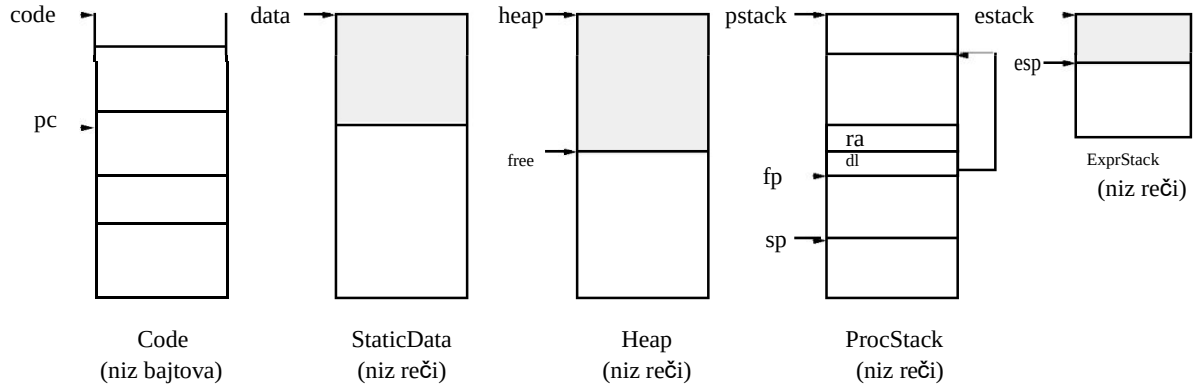
- Ne sme se koristiti više od 256 lokalnih promenljivih.
- Ne sme se koristiti više od 65536 globalnih promenljivih.
- Klasa ne sme imati više od 65536 članova (polja i metoda).
- Izvorni kod programa ne sme biti veći od 8 KB.

Dodatak B: Mikrojava virtuelna mašina

Ovaj dodatak opisuje arhitekturu Mikrojava virtuelne mašine koja se koristi u praktičnom delu kursa programskih prevodilaca 1 na Elektrotehničkom fakultetu u Beogradu. MikroJava VM je slična Java VM, ali ima znatno manje instrukcija. Neke instrukcije su takođe pojednostavljene. Dok kod Java VM punilac razrešava imena operanada iz skladišta konstanti (constant pool), dotle MikroJava VM koristi fiksne adrese operanada. U instrukcijama Java bajt koda kodirani su i tipovi njihovih operanada, tako da se može proveriti konzistentnost predmetnog fajla (object file). Instrukcije MikroJava bajt koda ne kodiraju tipove operanada.

B.1 Organizacija memorije

MikroJava VM koristi sledeće memorijske oblasti:



Code — Ova oblast sadrži kod metoda. U registru *pc* se nalazi indeks instrukcije koja se trenutno izvršava. Registar *mainpc* sadrži početnu adresu metode *main()*.

StaticData — U ovoj oblasti se nalaze (statički ili globalni) podaci glavnog programa (npr. klase koju kompajliramo). To je u stvari niz promenljivih. Svaka promenljiva zauzima jednu reč (32 bita). Adrese promenljivih su indeksi pomenutog niza.

Heap — Ova oblast sadrži dinamički alocirane objekte i nizove. Blokovi u heap-u se alociraju sekvencijalno. *free* pokazuje na početak slobodnog dela heap-a. Dinamički alocirana memorija se oslobađa samo na kraju izvršenja programa. Ne postoji sakupljanje đubreta.

Svako polje unutar objekta zauzima jednu reč (32 bita). Nizovi čiji su elementi tipa *char* su nizovi bajtova. Njihova dužina je umnožak broja 4. Pokazivači su bajt ofseti u heap-u. Objekti tipa niza počinju "nevidljivom" rečju koja sadrži dužinu niza.

ProcStack — U ovoj oblasti VM pravi aktivacione zapise pozvanih metoda. Svaki zapis predstavlja niz lokalnih promenljivih, pri čemu svaka zauzima jednu reč (32 bita). Adrese promenljivih su indeksi niza. *ra* je povratna adresa metode, *dl* je

dinamička veza (pokazivač na aktivacioni zapis pozivaoca metode). Novoalocirani zapis se inicijalizuje nulama.

ExprStack — Ova oblast se koristi za skladištenje operanada instrukcija. *ExprStack* je prazan posle svake MikroJava instrukcije. Argumenti metoda se prosleđuju na stek izraza i kasnije uklanjaju *Enter* instrukcijom pozvane metode. Ovaj stek izraza se takođe koristi za prosleđivanje povratne vrednosti metode pozivaocu metode.

Svi podaci (globalne promenljive, lokalne promenljive, promenljive na heap-u) se inicijalizuju null vrednošću (0 za *int*, chr(0) za *char*, *null* za reference).

B.2 Skup instrukcija

U sledećim tabelama su navedene instrukcije MikroJava VM, zajedno sa njihovim kodovima i ponašanjem. Treća kolona tabela prikazuje sadržaj *ExprStack*-a pre i posle svake instrukcije, na primer

…, val, val
…, val

znači da opisana instrukcija uklanja dve reči sa *ExprStack*-a i stavlja novu reč na njega. Operandi instrukcija imaju sledeće značenje:

b je bajt
s je short int (16 bitova)
w je reč (32 bita).

Promenljive tipa *char* zauzimaju najniži bajt reči, a za manipulaciju tim promenljivim se koriste instrukcije za rad sa rečima (npr. *load*, *store*). Niz čiji su elementi tipa *char* predstavlja niz bajtova i sa njima se manipuliše posebnim instrukcijama.

Instrukcije za load i store lokalnih promenljivih

opcode	***instr.***	*opds*	*ExprStack*	*značenje*
1	**load**	b	… …, val	Load push(local[b]);
2..5	**load_n**		… …, val	Load (n = 0..3) push(local[n]);
6	**store**	b	…, val …	Store local[b] = pop();
7..10	**store_n**		…, val …	Store (n = 0..3) local[n] = pop();

Instrukcije za load i store globalnih promenljivih

11	**getstatic** s		… …, val	Load statičke promenljive push(data[s]);

12	**putstatic** s	…, val …	Store statičke promenljive data[s] = pop();	

Instrukcije za load i store polja objekata

13	**getfield** s	…, adr …, val	Load polja objekta adr = pop()/4; push(heap[adr+s]);

14	**putfield** s	…, adr, val
		…
			Store polja objekta
			val = pop(); adr = pop()/4;
			heap[adr+s] = val;

Instrukcije za load konstanti

15..20	**const_n**	… …, val	Load konstante (n = 0..5) push(n)
21	**const_m 1**	… …, -1	Load konstante -1 push(-1)
22	**const** w	… …, val	Load konstante push(w)

Aritmetičke operacije

23	**add**	…, val1, val2 …, val1+val2	Sabiranje push(pop() + pop());
24	**sub**	…, val1, val2 …, val1-val2	Oduzimanje push(-pop() + pop());
25	**mul**	…, val1, val2 …, val1*val2	Množenje push(pop() * pop());
26	**div**	…, val1, val2 …, val1/val2	Deljenje x = pop(); push(pop() / x);
27	**rem**	…, val1, val2 …, val1%val2	Ostatak pri celobrojnom deljenju x = pop(); push(pop() % x);
28	**neg**	…, val …, -val	Promena predznaka push(-pop());
29	**shl**	…, val …, val1	Aritmetičko pomeranje ulevo x = pop(); push(pop() << x);
30	**shr**	…, val …, val1	Aritmetičko pomeranje udesno x = pop(); push(pop() >> x);
31	**inc** b1, b2	… …	Inkrementiranje local[b1] = local[b1] + b2;

Pravljenje objekata

32	**new** s	… …, adr	Novi objekat alocirati oblast od s bajtova; inicijalizovati oblast nulama; push(adr(oblast));

33	**newarray** b	…, n
		…, adr
			Novi niz
			n = pop();
			if (b==0)
			alocirati niz sa n elemenata veličine bajta;
			else if (b==1)
			alocirati niz sa n elemenata veličine reči;
			inicijalizovati niz nulama;
			push(adr(niz));

Pristup nizu

34	**aload**	…, adr, index
		…, val
			Load elementa niza (+ provera indeksa)
			i = pop(); adr = pop()/4+1;
			push(heap[adr+i]);

35	**astore**	…, adr, index, val
		…
			Store elementa niza (+ provera indeksa)
			val = pop(); i = pop();
			adr = pop()/4+1;
			heap[adr+i] = val;

36	**baload**	…, adr, index …, val	Load elementa niza bajtova (+ provera indeksa) i = pop(); adr = pop()/4+1; x = heap[adr+i/4]; push(byte i%4 of x);

37	**bastore**	…, adr, index, val
		…
			Store elementa niza bajtova (+ provera indeksa)
			val = pop(); i = pop();
			adr = pop()/4+1;
			x = heap[adr+i/4];
			set byte i%4 in x;
			heap[adr+i/4] = x;

38	**arraylength**	…, adr
		…, len
			Dohvatanje dužine niza
			adr = pop();
			push(heap[adr]);

Operacije na steku

39	**pop**	…, val …	Skidanje elementa sa vrha steka dummy = pop();
40	**dup**	…, val …, val, val	Udvajanje elementa na vrhu steka x = pop(); push(x); push(x);
41	**dup2**	…, v1, v2	Udvajanje prva dva elementa na

				vrhu steka

			…, v1, v2, v1, v2	y = pop(); x = pop();
				push(x); push(y); push(x);
				push(y);

Skokovi

Adresa skoka je relativna u odnosu na početak instrukcije skoka.

42	**jmp**	s		Bezuslovni skok pc = pc + s;

43..48	**j<cond>**	s	…, val1, val2
			…
				Uslovni skok (eq, ne, lt, le, gt, ge)
				y = pop(); x = pop();
				if (x cond y) pc = pc + s;

Pozivi metoda (PUSH i POP se odnose na stek procedura)

49	**call**	s	Poziv metode PUSH(pc+3); pc := pc + s;
50	**return**		Povratak iz metode pc = POP();

51	**enter** b1, b2		Početak obrade metode
			psize = b1;
			lsize = b2; *// u rečima*
			PUSH(fp);
			fp = sp; sp = sp + lsize;
			Inicijalizovati okvir nulama
			for (i=psize-1; i>=0; i--)
			local[i] = pop();

52	**exit**		Kraj obrade metode sp = fp; fp = POP();

Ulaz/Izlaz

53	**read**	… …, val	Operacija čitanja readInt(x); push(x); *// cita sa standardnog ulaza*
54	**print**	…, val, width …	Operacija ispisa width = pop(); writeInt(pop(), width); *// vrsi ispis na standardni izlaz*
55	**bread**	… …, val	Operacija čitanja bajta readChar(ch); push(ch);
56	**bprint**	…, val, width	Operacija ispisa bajta

			…	width = pop(); writeChar(pop(), width);

Ostalo

57	**trap**	b		Generiše run time grešku zavisno od vrednosti b se ispisuje odgovarajuća poruka o grešci; prekid izvršavanja;
58	**invokevirtual**	$w_1,w_2,...,w_n,w_{n+1}$	…, adr …	Poziv virtuelne metode ime metode ima n znakova; ovi znakovi su deo same instrukcije, i nalaze se u rečima $w_1,w_2,...,w_n$; reč w_{n+1} je jednaka -1 i označava kraj instrukcije; instrukcija prvo ukloni adr sa steka izraza; adr je adresa u statičkoj zoni memorije gde počinje tabela virtuelnih funkcija za klasu objekta čija metoda je pozvana; ako se ime metode u instrukciji pronađe u tabeli virtuelnih funkcija, instrukcija vrši skok na početak tela date metode.

B.3 Format predmetnog fajla

2 bajta: ″MJ″
4 bajta: veličina koda u bajtovima
4 bajta: broj reči rezervisan za globalne podatke
4 bajta: mainPC: adresa metode main() relativna u odnosu na početak code oblasti memorije
n bajtova: code oblast (n = veličina koda specificirana u header-u)

B.4 Runtime greške

1 Nedostaje return iskaz u telu funkcije.

Literatura

[1] H. Mössenböck, Compiler Construction course notes at the Oxford Brookes University, May 2004.

[2] D. Velašević, D. Bojić, Zbirka zadataka iz programskih prevodilaca I, Elektrotehnički fakultet, 2000.

[3] A. W. Appel, Jens Palsberg, Modern Compiler Implementation in Java, Second Editon, Cambridge University Press, 2002;

[4] C. Fischer, R. Cytron, R. LeBlanc, Crafting a Compiler, Addison-Wesley 2010;

[5] A. Aho, M. Lam, R. Sethi, J. D. Ullman, Compilers: Principles, Techniques and Tools, 2ed, Addison-Wesley, 2007.

[6] Dušan Stanković, Unapređenje generatora parsera CUP za rad sa apstraktnim sintaksnim stablima, diplomski rad, Elektrotehnički fakultet Univerziteta u Beogradu, oktobar 2009.