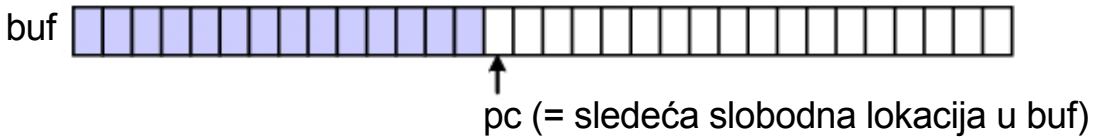
Generisanje koda u kompajleru za mikrojavu

Smeštanje koda u bafer

Struktura podataka

ceo kod držimo u nizu bajtova u memoriji, jer neke instrukcije zahtevaju naknadnu prepravku



Emitovanje instrukcija u bafer

put() emituje bajt, put2() dva bajta (short) a put4() četiri bajta (reč)

```
class Code {
    private byte[] buf = new byte[3000];
    public int     pc = 0;

    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);
    }

}
```

kodovi instrukcija deklarisani kao konst u klasi *Code*

```
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,
    ... ;
```

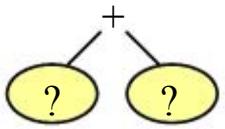
primer: emitovanje *load2*

```
Code.put(Code.load0 + 2);
```

Generisanje koda za pristup operandima

Primer

sabiranje dve vrednosti



treba generisati kod nalik na:

učitaj operand1 na estek
učitaj operand2 na estek
add

Instrukcije za učitavanje na estek zavise od vrste operanda

vrsta operanda	*instrukcija koju treba generisati*
• constant	const val
• lokalna promenljiva	load a
• globalna promenljiva	getstatic a
• polje objekta klase	getfield a
• element niza	aload
• vrednost na esteku	---

Obj zapis tabele simbola daje nam informaciju o vrsti operanda

Učitavanje vrednosti na e-stek

ulaz: vrednost opisana Obj zapisom (Con, Local, Static,...)
izlaz: emitovanje bajtkod instrukcije za učitavanje vrednosti na estek

```
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)  put(load_n + o.adr); // local variable
     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");
  }
}
```

zavisno od vrste Obj potrebno je generisati različite instrukcije

Gde se poziva Code.load()?

Razmotrimo smene za prepoznavanje korišćenja vrednosti obične promenljive:

```
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
                {:          Obj ob=Tab.find(id, idleft); // find prijavljuje grešku u slučaju da ne nadje id
                            RESULT=ob;
                :}
```

designator učestvuje i na levoj strani izraza dodele i na desnoj, tako da u njegovim smenama ne znamo da li treba raditi **load()** ili **store()**, pa se ne generiše kod. **Factor** je prvi od neterminala za izraze (ostali su **factor_list**, **term**, **term_list**, **expr**, **expr_list**) za koji znamo da se odnosi na izraze samo na desnoj strani iskaza dodele, pa se u njegovim smenama postavlja load().

Prevođenje izraza

Primer sabiranja dve vrednosti:

```
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; :};
```

Load() u smeni za factor dovodi vrednost na stek, tako da složeniji izrazi u kojima faktor učestvuje ne treba da generišu kod za dovođenje operanada na stek, nego samo za izvršenje operacija.

Atributi factora i ostalih neterminala za izraze prosleđuju tip (Struct objekat) da bi se mogle raditi semantičke provere u izrazima. Obj.Elem odnosi se na element niza. Ovaj tip postoji samo u izrazima, a ne koristi se pri deklarisanju promenljivih.

Prevođenje pristupa polju objekta klase

o.f

Konteksni uslovi (treba ih ugraditi u semantičke akcije)

IdentExprList$_0$ = IdentExprList$_1$ "." ident .

- *IdentExprList*$_1$ mora biti tipa klase.
- *ident* mora biti polje klase *IdentExprList*$_1$.

Semantičke akcije za generisanje koda

```
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; :}
```

Radi i u slučaju da je polje objekta drugi objekat (x.y.z), pogledati slučaj Obj.Fld u Code.load().

Pristup elementu niza

a[i]

Kontekstni uslovi koje treba ugraditi u semantičke akcije

IdentExprList$_0$ = IdentExprList$_1$ "[" Expr "]" .

- *IdentExprlist*$_1$ mora biti tipa niza.
- *Expr* mora biti tipa *int*.

Semantičke akcije za generisanje koda

```
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
                     {:  Obj ob=Tab.find(id, idleft); // find prijavljuje gresku u slucaju da ne nadje
                         RESULT=ob;
                     :}
```

provera indeksa se radi u vreme izvršavanja od strane MJVM ;

Generisanje koda za iskaz dodele

Ima četiri slučaja zavisno od designatora na levoj strani dodele

Statement ::= Designator "=" 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**

plave instrukcije su već generisane u okviru *Designatora.* **Bold** instrukcije treba generisati u okviru posmatrane smene. Za to se koristi funkcija Code.store()

Generisanje koda za iskaz dodele

Kontekstni uslovi

Statement = Designator "=" Expr ";".

- *Designator* mora biti promenljiva, element niza ili polje objekta klase.
- Tip *Expr* mora biti kompatibilan pri dodeli sa tipom *Designatora*.

Semantičke akcije

```
statement ::= designator:o EQUAL expr:t SEMI
        {: .../ provera uslova
            Code.store(o);
        :}
```

Uslovni i bezuslovni skokovi u bajtkodu

Bezuslovni skok

```
jmp offset
```

Uslovni skok

```
... load operand1 ...
... load operand2 ...
jeq offset
```

if (operand1 == operand2) jmp offset

jeq	jump on equal
jne	jump on not equal
jlt	jump on less than
jle	jump on less or equal
jgt	jump on greater than
jge	jump on greater or equal

```
static final int
   eq = 0,
   ne = 1,
   lt  = 2,
   le  = 3,
   gt  = 4,
   ge = 5;
```

u klasi *Code*

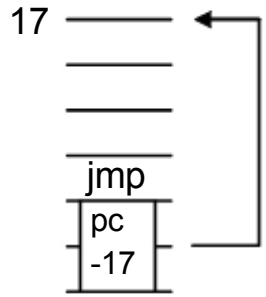
Emitovanje instrukcija skokova

```
Code.put(Code.jmp);
Code.put2(offset);
```

```
Code.put(Code.jeq + operator);
Code.put2(offset);
```

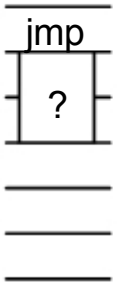
Uslovni i bezuslovni skokovi

Skokove unazad lako je izgenerisati

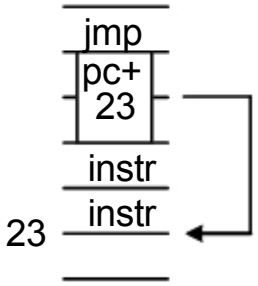


odredišna adresa je već poznata
(jer je kod na odredišnoj adresi već izgenerisan)

Skokovi unapred



odredišna adresa još nije poznata
⇒ ostaviti prazno
⇒ zapamtiti "adresu za prepravku"



“zakrpiti” kada odredišna adresa postane poznata (backpatching)

Metodi u klasi Code za generisanje skokova

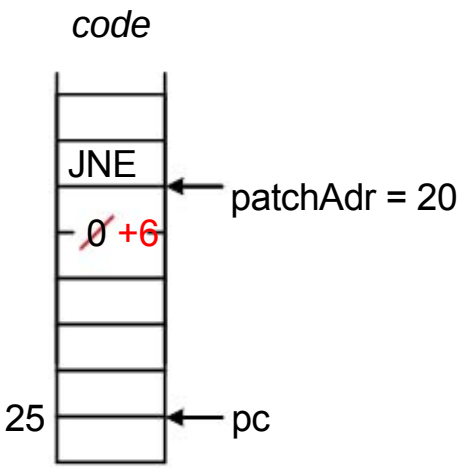
```
class Code {
   private static final int
      eq = 0, ne = 1, lt = 2, le = 3, gt = 4, ge = 5;
   private static int inverse[]= {ne, eq, ge, gt, le, lt};

   // 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);
   }
}
```

adresa je short veličina



Generisanje koda za iskaz while

Želimo da se izgeneriše sledeći kod:

```
while                top:
(Condition)                ... code for Condition ...
                           falseJump end
Statement                  ... code for Statement ...
                           jump top
                     end:
```

Semantičke akcije

```
Statement ::=
 WHILE: top                    {: top = Code.pc :}
   LPAREN Condition: op        {:  Code.putFalseJump(op,  0);
   RPAREN                          adr = Code.pc - 2; :}
   Statement                   {: Code.putJump(top);
                                   Code.fixup(adr); :}
```

Primer

```
while (a > b) a = a - 2;

10   load0      <-- top
11   load1
12   jle +10
15   load0
16   const2
17   sub
18   store0
19   jmp -9
22                          fixup(13)
```

Generisanje koda za iskaz if

Želimo da se izgeneriše sledeći kod

```
if                          ... Condition ...
(Condition)                 falseJump end
                            ... Statement ...
Statement                   end:
```

```
if                          ... Condition ...
(Condition)                 falseJump els
                            ... Statement ...
Statement                   jump end
else                        els:
                            ... Statement ...
Statement                   end:
```

Semantičke akcije

```
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); :}
```

Primer

```
if (a > b) max = a; else max = b;
```

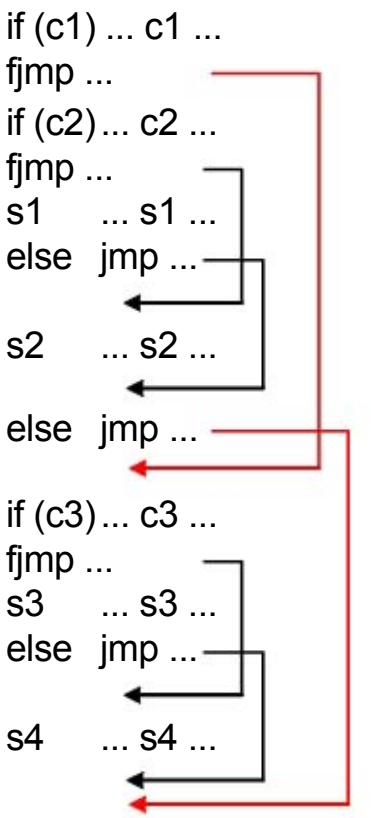
```
10   load0
11   load1
12   jle +8
15   load0
16   store2
17   jmp +5
20   load1
21   store2
22
```

fixup(adr)

fixup(adr2)

Radi i za ugnežđene strukture

```
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); :}
```



Generisanje koda za poziv metoda

Izvorni program	Generisani kod	
c = m(a, b);	load a load b	parametri se prosleđuju na *e-steku*
	call m	
	store c	funkcija vraća vrednost na *e-steku*

Semantičke akcije

```
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 */  :};
```

Slično se realizuju i druge smene za poziv metoda npr.

statement ::= designator:o LPAREN act_pars RPAREN SEMI

U okviru expr_list ne treba load() za smeštanje stvarnog parametra na e-stek jer su ga smene za expr već generisale.

Obrada deklaracije metoda

```
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 funkciju na kraj tela funkcije,
                           // da izazove run time grešku ako se zaobiđe return
                          Code.put(Code.trap); Code.put(1);
                      }
                :};
```

method_type i returnExists su članovi klase parser, služe za prenos informacija iz drugih smena, pogledati primer mini domaćeg treći deo

Obrada formalnih parametara

- Unose se u tabelu simbola (kao promenljive u lokalnom opsegu metoda)
- Broje se, ta vrednost se prosleđuje kao atribut neterminala

```
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)); :};
```

Obrada iskaza 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_);
         :}
```

method_type i returnExists deklarišemo kao članove klase parser, služe za prenos informacija iz drugih smena, pogledati primer mini domaćeg treći deo