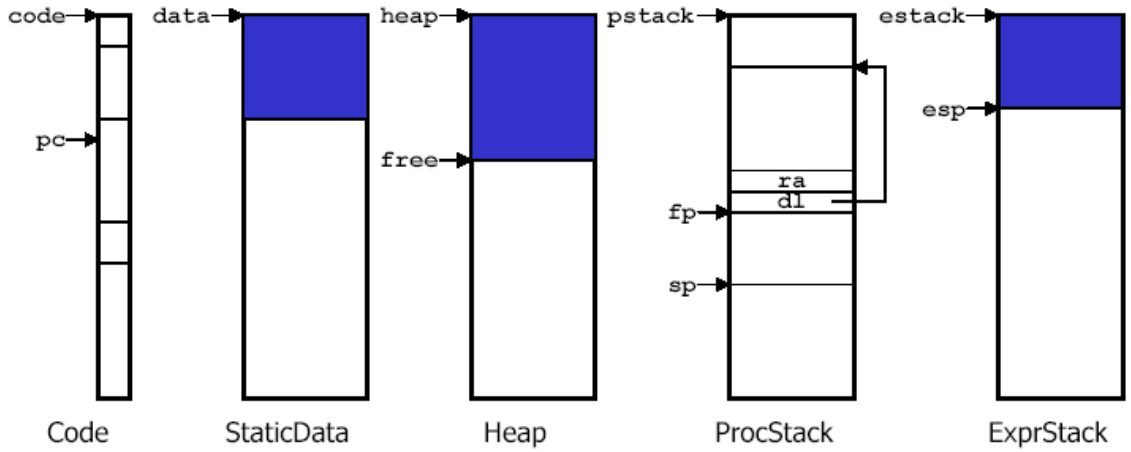
Mikrojava virtuelna mašina kao primer izvršnog okruženja

MikroJava VM koristi sledeće memorijske oblasti:



Code	Ova oblast sadrži kod metoda, preciznije bajt kod programa. U registru **pc** se nalazi indeks instrukcije VM koja se trenutno izvršava. Registar **mainpc** sadrži početnu adresu metode `main()`. Iz ove oblasti VM čita instrukcije i izvršava ih. Code oblast predstavlja niz bajtova, dok su sve ostale oblasti nizovi reči.
StaticData	U ovoj oblasti se nalaze (statički ili globalni) podaci glavnog programa (tj. glavne 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** (return address) je povratna adresa metode, **dl** (dynamic link) je kontrolna veza (pokazivač na aktivacioni zapis pozivaoca metode). Novoalocirani zapis se inicijalizuje nulama.
ExprStack	Ova oblast se koristi za skladištenje operanada i rezultata instrukcija. **ExprStack** je prazan posle svake MikroJava instrukcije. Stvarni argumenti metoda se, pre poziva metode, prosleđuju na stek izraza i kasnije uklanjaju `enter` instrukcijom pozvane metode. Ovaj stek izraza se takođe koristi za prosleđivanje povratne vrednosti metode njenom pozivaocu.

Svi podaci (globalne promenljive, lokalne promenljive, promenljive na heap-u) se inicijalizuju

null vrednošću (0 za `int`, `chr(0)` za `char`, `null` za reference).
Operandi MJVM 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.

- Skup instrukcija

Instrukcije MJ virtuelne mašine se mogu podeliti u 12 grupa:

1. Instrukcije za load i store lokalnih promenljivih

Ova grupa služi za stavljanje i skidanje lokalnih promenljivih na/sa steka izraza pri korišćenju tih promenljivih u telu metoda. Ako su lokalne promenljive prostog tipa, onda se na memorijskoj lokaciji koja im odgovara, nalazi trenutna vrednost promenljive, što znači da se na stek stavlja ta vrednost. Ako su lokalne promenljive tipa unutrasnje klase ili niza, onda se na memorijskoj lokaciji koja im odgovara nalazi adresa na heap-u objekta te klase, odnosno početna adresa niza. Znači, da se u takvoj situaciji na stek izraza stavlja adresa.

Primer:

kod instrukcije	instrukcija	operand	sadržaj steka izraza
1	**load**	**b**	...
			..., val

Ovom instrukcijom se vrednost lokalne promenljive stavlja na stek izraza. **b** označava operanda koji je veličine jednog bajta. Taj bajt je redni broj lokalne promenljive u grupi deklaracija lokalnih promenljivih metode u čijem telu se nalazimo. U ovom slučaju se u lokalne promenljive ubrajaju i argumenti metode, pa brojanje pozicije počinje od njih.

2. Instrukcije za load i store globalnih promenljivih

Ova grupa služi za stavljanje i skidanje globalnih promenljivih na/sa steka izraza pri korišćenju tih promenljivih u telu metoda. Ako je globalna promenljiva tipa klase ili niza, onda se na memorijskoj lokaciji koja joj odgovara nalazi adresa objekta na heap-u, tj. početna adresa niza.

Primer:

kod instrukcije	instrukcija	operand	sadržaj steka izraza
11	**putstatic**	**s**	..., val
			...

Ovom instrukcijom se sa steka izraza skida vršni element val i dodeljuje gobalnoj promenljivoj. **s** označava operanda koji je short int, tj. veličine 16 bita. Taj operand je redni broj globalne promenljive u grupi deklaracija globalnih promenljivih u glavnoj klasi.

kod instrukcije	instrukcija	operand	sadržaj steka izraza
12	**getstatic**	**s**	...
			..., val

Ovom instrukcijom se na steka izraza stavlja vrednost globalne promenljive. Operand s je redni

broj globalne promenljive u grupi deklaracija globalnih promenljivih u glavnoj klasi.

3. Instrukcije za load i store polja objekata

Ova grupa služi da se vrednost sa steka izraza dodeli polju nekog objekta, ili da se vrednost polja objekta stavi na vrh steka izraza.

Primer:

kod instrukcije	instrukcija	operand	sadržaj steka izraza
14	**putfield**	**s**	..., adr, val
			...

putfield s skida vrednost sa steka izraza i dodeljuje je polju odgovarajućeg objekta. Ova instrukcija podrazumeva da su na stek izraza prethodno stavljeni početna adresa na heap-u (adr) objekta čijem polju ćemo dodeljivati, i vrednost koju dodeljujemo (val). s je operand koji predstavlja poziciju polja (ofset) u objektu svoje klase.

4. Instrukcije za load konstanti

Ova grupa služi za stavljanje određene konstante na stek izraza.

Primer:

kod instrukcije	instrukcija	operand	sadržaj steka izraza
22	**const**	**w**	...
			..., val

Navedena instrukcija stavlja na stek izraza konstantu koja se navodi kao operand w. Operand je širine jedne reči, tj. 32 bita.

5. Aritmetičke operacije

Primer:

kod instrukcije	instrukcija	operand	sadržaj steka izraza
23	**add**	nema	..., val1, val2
			..., val1+val2

Navedena instrukcija skida sa steka izraza dva vršna elementa, sabira ih i stavlja rezultat na stek izraza.

6. Pravljenje objekata

Ova grupa služi za alociranje prostora na heap-u za nove objekte i nizove.

Primer:

kod instrukcije	instrukcija	operand	sadržaj steka izraza
32	**new**	**s**	...,
			..., adr

Ova instrukcija pravi novi objekat klase. Na heap-u se alocira prvi slobodan prostor od s bajtova (s je 16-obitni operand; s = broj_polja_u_objektu_klase * 4). Taj prostor se inicijalizuje nulama, pa se na stek izraza stavi njegova početna adresa.

7. Pristup nizu

Ova grupa služi za manipulaciju elementima nizova.

Primer:

kod instrukcije	instrukcija	operand	sadržaj steka izraza

34	**aload**	nema	..., adr, index ..., val

aload instrukcija obezbeđuje čitanje vrednosti odgovarajućeg elementa niza. Podrazumeva se da su prethodno na stek izraza stavljeni početna adresa niza na heap-u i indeks elementa kojem se pristupa. Ova naredba skida te dve vrednosti sa steka, vrši proveru da li je indeks u granicama adresiranog niza, učitava vrednost elementa i stavlja je na vrh steka izraza.

8. Operacije na steku

Ova grupa služi za neposredan rad sa stekom izraza.

Primer:

kod instrukcije	instrukcija	operand	sadržaj steka izraza
39	**pop**	nema	..., val ...

pop skida element sa vrha steka izraza. Skinuta vrednost nam je nedostupna.

9. Skokovi

Ova grupa služi za realizaciju uslovnih i bezuslovnih skokova u programu. Adresa skoka je relativna u odnosu na početak instrukcije skoka.

Primer:

kod instrukcije	instrukcija	operand	sadržaj steka izraza
42	**jmp**	**s**	ne utiče na njega

Navedena naredba realizuje bezuslovni skok na adresu pc+s, gde je s 16-obitni operand jmp instrukcije.

10. Pozivi metoda

Ova grupa služi za realizaciju sekvence pozivanja i sekvence povratka iz metode.

kod instrukcije	instrukcija	operand
49	**call**	**s**

Instrukcija call počinje formiranje aktivacionog zapisa metode koja se poziva, tako što na stek procedura stavlja povratnu adresu i vrši skok (ažurira se pc) na deo bajt koda koji odgovara pozvanoj metodi.

kod instrukcije	instrukcija	operand
50	**return**	nema

Instrukcija return skida sa steka procedura povratnu adresu i dodeljuje je registru pc, čime se kontrola vraća pozivaocu.

kod instrukcije	instrukcija	operandi
51	**enter**	**b1, b2**

Instrukcija enter je prva instrukcija u bajt kodu svake metode. Operand b1 predstavlja broj argumenata metode, a b2 zbir broja argumenata i broja lokalnih promenljivih. Enter nastavlja formiranje aktivacionog zapisa metode tako što formira polje kontrolne veze i alocira b2 mesta na steku procedura za stvarne parametre i lokalne promenljive. Stvarni parametri se skidaju sa steka izraza i kopiraju odmah posle polja kontrolne veze (prema vrhu steka) onim redosledom kojim su navedeni u definiciji metode. Ostatak alociranog prostora, između poslednjeg parametra i lokacije na koju ukazuje sp je rezervisan za lokalne promenljive.

kod instrukcije	instrukcija	operand
52	**exit**	nema

Instrukcija exit označava kraj obrade metode. Prostor za stvarne argumente i lokalne promenljive se dealocira sa steka procedura i restaurira se polje kontrolne veze.

kod instrukcije	instrukcija	operand
58	**invokevirtual**	w1 w2…wn -1 wn wn+1 …, adr

Ime virtuelne metode sastoji se od n znakova, koji se smeštaju u samu instrukciju. Svaki znak predstavlja jednu memorijsku reč. Poslednja reč je jednaka -1 (wn+1). Vrednost adr predstavlja adresu početka tabele virtuelnih funkcija u zoni StaticData. Instrukcija uklanja vrednost adr sa Expr steka. Instrukcija prvo uzme sa steka izraza adresu v-tabele, pristupi joj i poređenjem karaktera traži da li se u njoj nalazi ime metode koje se poklapa sa imenom koje je operand instrukcije. Ako je ime nađeno, iz v-tabele se čita adresa početka koda metode, koja se nakon stavljanja trenutne vrednosti PC-ja na stek procedura upisuje u PC. Time se prelazi na izvršenje koda odgovarajuće metode.

11. Ulaz/Izlaz

Ova grupa služi za čitanje i ispis sa/na standardni izlaz.

12. Instrukcija trap

trap generiše run-time grešku. Zavisno od vrednosti operanda b, ispisuje se poruka o grešci i prekida izvršavanje programa.

kod instrukcije	instrukcija	operand	sadržaj steka izraza
57	**trap**	**b**	ne utiče na njega

Prevođenje izraza

Zadatak 1.
Za svaki od simbola u listingu Mikrojava programa, odrediti da li postoji u vreme prevođenja ili u vreme izvršavanja programa, kao i gde je alociran u memoriji.

```
class A
final int max = 12;    // Konstanta
char c; int i;         // globalne promenljive
class B { int x, y; }  // lokalna klasa
{
      void foo() int[]iarr; B b; int n;
      {
        b=new B;
        b.x = 2;
        b.y = 3;
        arr[2] = b.x + max;
      }
}
```

Rešenje:
Simboli nađeni u kodu i njihove lokacije u memoriji su prikazani u tabeli.

Ime simbola	Memorija	Adresa(offset)
A	-	-
max	Code memory-	12 - vrednost
c	StaticData	0
i	StaticData	1
B	-	-
B.x	-	0
B.y	-	1
iarr	ProcStack	0
iarr[2]	Heap	Val(iarr)+2*sizeof(int)
b	ProcStack	1
b.x	Heap	Value(fp+1) + 0
b.y	Heap	Value(fp+1) + 1
n	ProcStack	2

Zadatak 2.
Napisati MJ asemblerski kod koji obrađuje telo metode foo, bez enter, exit i return sekvenci..

```
class A
final int max = 12;    // Konstanta
int i;             // globalna promenljive
class B { int x, y; }  // lokalna klasa
{
      void foo() int[]iarr; B b; int n;
      {
        i = max;
        b = new B();
        iarr[0] = -1;
      }
}
```

Rešenje:
Za izvršavanje naredbe i = max, potrebno je učitati promenljivu max na stek izraza i sačuvati ga u promenljivu i koja se nalazi u statičkoj memoriji na lokaciji 0. Za ovu operaciju su potrebne sledeće naredbe:

```
const 12
putstatic_0
```

Za izvršavanje naredbe b = new B(), potrebno je alocirati memoriju na Heap-u za objekat klase B veličine dve reči, a potom adresu alocirane memorije sa steka izraza upisati u promenljivu b u okviru procedure foo.

```
new 2
store_1
```

Za izvršavanje naredbe iarr[0] = -1, potrebno je učitati vrednost -1 na stek izraza, postaviti adresu i indeks eleenta u koji će vrednost =1 biti smeštena i sačuvati ga na lokaciju na koju pokazuje iaar[0] .

```
load_0
const_0
const_m1
astore
```

Zadatak 3.
Ako su date sledeće Mikrojava globalne deklaracije:

```
class A{
   int i;
}
class B{
    A[5] a;
}
class C{
    B b;
}
C c;
int res;
```

Prikazati bajtkod za izraz:

```
res = c.b.a[2].i;
```

Prikazati sadržaj steka izraza prilikom izvršavanja koda. Pretpostavka je da je promenljiva c ispravno inicijalizovana.

Rešenje

Instrukcija	Stek izraza
getstatic 0	c
getfield 0	c.b
getfield 0	c.b.a
const_2	c.b.a, 2
aload	c.b.a[2]
getfield 0	c.b.a[2].i
putstatic 1	

Prevođenje kontrolnih struktura

Zadatak 1.

Prikazati generisani kod za telo funkcije prikazane na listingu , bez enter, exit i return sekvenci.

```
class A
final int max = 12;
char c; int i;
class B { int x, y; }
{ void foo () int[] iarr; B b; int n;
      {
            if (i <= n) n=0;
      }
}
```

Rešenje:

Pri prevođenju if kontrole prvo se generiše kod za izračunavanje uslova. Ako uslov nije ispunjen generiše se instrukcija skoka između ove dve grupe koda kojom se preskače telo if kontrole. Iza koda se generiše kod koji odgovara telu if kontrole.

Kompleten kod je:

```
10: getstatic 1
13: load_2
14: jgt 5 (--> 19)
17: const_0
18: store_2
19: ...
```

Zadatak 2.

Prikazati generisani kod za telo funkcije prikazano na listingu, bez enter, exit i return sekvenci.

```
class A
final int max = 12;
char c; int i;
class B { int x, y; }
{ void foo () int[] iarr; B b; int n;
      {
            if (i <= n && n < 0) n=0;
      }
}
```

Rešenje:

Pri prevođenju if kontrole za svaki uslov povezan operatorom **and** se generiše kod koji ga izračunava. Iza svakog uslova se generiše upit koji u slučaju da uslov nije tačan preskače celu if kontrolu, inače nastavlja dalje

Kompleten kod je:

```
10: getstatic 1
13: load_2
14: jgt 10 (--> 24)
17: load_2
18: const_0
19: jge 5 (--> 24)
22: const_0
23: store_2
24: ...
```

Zadatak 3.
Prikazati generisani kod za telo funkcije prikazano na listingu, bez enter, exit i return sekvenci.

```
class A
final int max = 12;
char c; int i;
class B { int x, y; }
{ void foo()
      int[] iarr; B b; int n;
      {
            if (i <= n || n < 0) n=0;
      }
}
```

Rešenje:

Pri prevođenju if kontrole za svaki uslov povezan operatorom **or** se generiše kod koji ga izračunava. Iza svakog uslova se generiše upit koji u slučaju da je uslov tačan preskače kod za ispitivanje uslova i prelazi na kod koji odgovara telu if kontrole.

Kompleten kod je:

```
10: getstatic 1
13: load_2
14: jle 8 (--> 22)
17: load_2
18: const_0
19: jge 5 (--> 24)
22: const_0
23: store_2
24: ...
```

Zadatak 4.

Prikazati generisani kod za telo funkcije prikazano na listingu, bez enter, exit i return sekvenci.

```
class A
final int max = 12;
char c; int i;
class B { int x, y; }
{ void foo()
      int[] iarr; B b; int n;
      {
            if (i <= n || n < 0 && i>0) n=0;
      }
}
```

Rešenje:

```
10: getstatic 1
13: load_2
14: jle 15 (--> 29)
17: load_2
18: const_0
19: jge 12 (--> 31)
22: getstatic 1
25: const_0
26: jle 5 (--> 31)
29: const_0
30: store_2
31: ...
```

Zadatak 5.

Prikazati generisani kod za telo funkcije prikazano na listingu, bez enter, exit i return sekvenci.

```
class A
final int max = 12;
char c; int i;
class B { int x, y; }
{ void foo()
      int[] iarr; B b; int n;
      {
            if (i<= n ) n=0 else n=1;
      }
}
```

Rešenje:

```
10: getstatic 1
13: load_2
14: jgt 8 (--> 22)
17: const_0
18: store_2
19: jmp 5 (--> 24)
22:const_1
23: store_2
24: ...
```

Zadatak 6.
Prikazati generisani kod za telo funkcije prikazano na listingu, bez enter, exit i return sekvenci.

```
class A
final int max = 12;
char c; int i;
class B { int x, y; }
{ void foo()
      int[] iarr; B b; int n;
      {
            do n++ while (i<=n);
      }
}
```

```
10: inc 2 1
13: getstatic 1
16: load_2
17: jle -7 (--> 10)
20: ...
```

Zadatak 7.
Prikazati generisani kod za telo funkcije prikazano na listingu.

```
class A
final int max = 12;
char c; int i;
class B { int x, y; }
{ void foo()
      int[] iarr; B b; int n;
      {
            while(i<=n) n++;
      }
}
```

Rešenje:

```
10: getstatic 1
13: load_2
14: jgt 9 (--> 23)
17: inc 2 1
20: jmp -10 (--> 10)
23: ...
```

Prevođenje poziva metoda

Zadatak 1.

Prikazati generisani kod za telo funkcije *m* prikazano na listingu. Prikazati redosled mašinskih instrukcija koje se izvršavaju pri pozivu funkcije m.

```
int p = 5,q =6;
int m (int i, int j) int k;
{
      p = j;
      k = i;
    return k;
}
void main ()
{
      q = m(p,1);
}
```

Analiza problema:
Prilikom pozivanja funkcije odgovornost onog ko poziva je na stek izraza postavi stvarne parametre koji odgovaraju fukciji i da pozove funkciju. Odgovornost funkcije je da stvarne parametre prebaci u aktivacioni zapis u prostor za lokalne promenljive. Ako ima povratne vrednosti funkcija je pre zavrsetka stavlja na stek izraza.
Odgovornost pozivaoca je da pročita povratnu vrednost sa vrha steka izraza i da je smesti u odgovarajuću lokaciju.

Rešenje:

```
0:  enter 2,3
3:  load_1     // p = j;
4:  putstatic_0
5:  load_0     // k = i;
6:  store_2
7:  load_2     // return k;
8:  exit
9:  return
10: getstatic_0       // m(p,1);
11: const_1
12: call -12       // push pc na p-stack, goto enter 2,3 (adresa 0)
15: putstatic 1       //k = result
```

Redosled instrukcija je dat u tabeli:

Naredba	Instukcija	Stek izraza	p	q	i	j	k
			5	6			
m(p,1)	getstatic 0	5	5	6			
	const_1	5 1	5	6			
	call -20	5 1	5	6			
	enter 2,3		5	6	5	1	
p = j	load_1	1	5	6	5	1	
	putstatic 0		1	6	5	1	
k = i	load_0	5	1	6	5	1	
	store_2		1	6	5	1	5
return k	load_2	5	1	6	5	1	5
	exit	5	1	6			
	return	5	1	6			
q = m(p,1)	putstatic 1	5	1	5			

Zadatak 2.
Dat je MikroJava program:

```
class P
 final int kon = 5;
 int a;
 class c {
    int s;
 }

 c gl;

{

 int m(int f)
    int y;
 {
    return f;
 }
 void main()
    int x;
    c kl;
 {
    gl = new c;
    gl.s = kon + x;
    a = x;
    m(a);
 }
}
```

Napisati MikroJava bajt kod koji odgovara datom programu.

Rešenje

Bajt kod odgovara kodu metoda i stavlja se u Code memory. Bajt kod za metode generišemo redom kojim su njihove definicije navedene u programu. Brojevi navedeni na početku svake instrukcije predstavljaju redni broj bajta u Code memory na kojem ta instrukcija počinje. U ovom zadatku smo zbog jednostavnosti pretpostavili da je svaka instrukcija veličine jednog bajta. U realnosti veličina instrukcije varira. Operacioni kod instrukcije uvek zauzima jedan bajt. Ostatak instrukcije čine operandi, ako postoje, čije su veličine definisane u opisu instrukcija.
VM će izvršavati program počevši od bajt koda za metodu main. Na početak bajt koda za main ukazuje registar mainPC.

```
/* Pocinje bajt kod za metodu m. Metoda ima jedan stvarni argument i
jednu lokalnu promenljivu. */

0: enter 1, 2

/* Povratna vrednost metode se ocekuje na steku izraza. Znaci, da je
potrebno da se na stek izraza stavi vrednost stvarnog argumenta f. U
bajt kodu se i argumenti metoda tretiraju kao lokalne promenljive,
sto znaci da je f lokalna promenljiva na poziciji 0. */

1: load 0
2: exit
3: return

/* Pocinje bajt kod za metodu main. */

4: enter 0, 2

/* Na heap-u se alocira prostor za objekat klase c. Klasa c ima samo
jedno polje, pa je velicina objekta jedna rec, tj. 4 bajta. Nakon
izvrsenja naredbe new, na steku izraza se nalazi adresa
novoformiranog objekta klase c. */

5: new 4

/* Adresa novoformiranog objekta tipa klase c se skida sa steka
izraza i dodeljuje globalnoj promenljivoj gl, koja je na poziciji 1.
Na poziciji 0 je globalna promenljiva a. */

6: putstatic 1

/* Na stek izraza se stavlja adresa objekta gl. */

7: getstatic 1

/* Na stek izraza se stavlja vrednost konstante kon. */

8: const 5

/* Na stek izraza se stavlja vrednost lokalne promenljive x na
poziciji 0. */
```

```
9: load 0

/* Vrsi se sabiranje kon i x. Nakon izvrsenja ove naredbe, na vrhu
steka se nalazi zbir, a odmah ispod njega pocetna adresa objekta gl.
Time su se stekli uslovi za dodelu polju s objekta gl. */

10: add

/* Rezultat operacije sabiranja, koji se nalazi na vrhu steka izraza,
se dodeljuje polju s, na poziciji 0, objekta gl. */

11: putfield 0

/* Na stek izraza se stavlja vrednost lokalne promenljive x na
poziciji 0. */

12: load 0

/* Vrednost lokalne promenljive x se skida sa vrha steka i stavlja u
globalnu promenljivu a. */

13: putstatic 0

/* Stvarni argument u pozivu metode m je globalna promenljiva a na
poziciji 0. Pre samog poziva metode, stvarni argumenti se moraju
staviti na stek izraza. */

14: getstatic 0

/* Poziva se metoda m. Argument poziva je pomeraj koji se dodaje na
trenutnu vrednost registra pc da bi se skocilo na pocetak bajt koda
za metodu m. */

15: call -15

/* Nakon povratka iz metode m, na steku izraza se nalazi njena
povratna vrednost. Posto se ta vrednost nikom ne dodeljuje, mora se
ukloniti sa steka izraza. */

16: pop

/* Povratak iz metode main. */

17: exit
18: return
```

Virtuelne metode, nasleđivanje i polimorfizam

Zadatak 1
Dat je sledeći program na programskom jeziku Mikrojava.

a) Izgenerisati instrukcije za poziv virtuelne metode calc.
b) Generisati kompletan bajtkod za dati program.
c) Prikazati sekvencu bajtkoda koja generiše tabelu virtuelnih funkcija.
d) Prikazati strukturu objekata i tabela virtuelnih funkcija u memoriji u vreme izvršavanja programa.

```
class OOMJ
    class TCalc { // TaxCalculator
        int tax; // [0,100] interval
        {
            int calc (double cost) { return cost*(1+tax); }
            int tax() { return tax; }
        }
    }
    class ATCalc extends TCalc {
        // Tax including amortization
        int a; // [0,100] interval
        {
            // @Overriden
            int calc (double c) { return (c*a)*(1*tax);}
            int amtz() { return a; }
        }
    }
{
    void main()
        TCalc c1, c2;
        int cost, price;
    {
        cost = 1000;
        c1 = new  Tcalc;  c1.tax = 10;
        c2 = new ATCalc; c2.tax = 18; c2.a =2;
        print(c1.calc (cost));
        print(c2.calc (cost));
    }
}
```

Rešenje:
a) pozivanje dinamičkih metoda
Neka je data sledeća sekvenca programskog koda:

```
c2 = new ATCalc;
c2.tax = 18; c2.a = 2;
cost = 100;
price = c2.calc(cost);
```

Ovoj sekvenci odgovara sledeća sekvenca asemblerskih instrukcija MikroJava virtuelne mašine:

new *3* /* klasa ATCalc ima dva polja i jos jedno skriveno polje za adresu v-tabele.*/
store_*1* /* Pretpostavka je da je c2 lokalna promenljiva metode main na poziciji 1. */

```
                /* INICIJALIZACIJA objekta. */
load_1          /* Stavlja se adresa objekta na expr stek. */
const XXX       /*Pretpostavka je da v-tabela za ATCalc počinje na adresi XXX u Static Data*/
                /*    (podesavanje pointera na tabelu virtuelnih funkcija. */
putfield_0      /* Vrednost XXX je ubačena u nulto polje objekta. */
load_1          /* c2.tax = 18; */
const 18
putfield_1

load_1          /* c2.a= 2;  */
const_2
putfield_2

const 100       /*cost = 100*/
store_2
                /* PRIPREME ZA POZIV DINAMIČKE METODE. */
                /* price = c2.calc(cost); */
load 1          /* Na stek izraza se stavlja adresa objekta c2 (prvi stvarni argument, tj. this!) */
load 2          /* Stavljanje drugog argumenta cost. */
load 1          /* Pristupa se pointeru v-tabele objekta. */
getfield_0      /* Stavlja se adresa v-tabele klase objekta c2 na expr. stek. */
invokevirtual 'c' 'a' 'l' 'c' -1  /* poziv metode calc. (-1 označava kraj instrukcije). */
store_3         /* rezultat calc metode se sa expr steka sacuva u promenljivu price. */
```

Promenljivoj c moze biti dodeljen objekat tipa klase C, ali i tipa bilo koje potklase klase C. Znači, da se za vreme prevođenja ne zna kog će dinamičkog tipa c zaista biti. Međutim, u nultom polju objekta koji c referencira nalazi se adresa v-tabele odgovarajućeg dinamičkog tipa, tj. klase. Time je obezbeđeno da bude pozvana ispravna metoda u zavisnosti od tipa objekta, a ne tipa reference (polimorfno ponašanje).

b) Kompletan kod za dati program:

```
0:  enter 2,2      // TCalc::calc, including this.
1:  load_1
2:  const_1
3:  load_0
4:  getfield_1 // field 0 is reserved for vftPtr.
5:  add
6:  mul
7:  exit
8:  return

9:  enter 1,1  // TCalc::tax
10: load_0
11: getfield_0
12: exit
13: return

14: enter 2,2      // ATCalc::calc
15: load_1
16: load_0
17: getfield_2
```

```
18: mul
19: const_1
20: load_0
21: getfield_1
22: mul
23: mul
24: exit
25: return

26: enter 1,1      // ATCalc::amtz
27: load_0
28: getfield_2
29: exit
30: return

// Virtual Function Table initialization code.
// TCalc class VFT
31: const 99       // 'c' ⇔ 99
36: putstatic 0    // TCalc VFT start address (0 - static data)
39: const 97       // 'a' ⇔ 97
44: putstatic 1
47: const 108      // 'l' ⇔ 108
52: putstatic 2
55: const 99       // 'c' ⇔ 99
60: putstatic 3
63: const_m1       // -1 (method name terminator)
64: putstatic 4
65: const 0        // adr(TCalc::calc)
70: putstatic 5

73: const 116      // 't' ⇔ 116
78: putstatic 6
81: const 97       // 'a' ⇔ 97
86: putstatic 7
89: const 120      // 'x' ⇔ 120
94: putstatic 8
97: const_m1       // -1 (method name terminator)
98: putstatic 9
101:    const 9        // adr(TCalc::tax)
106:    putstatic 10
109:    const_m2       // -2 (TCalc VFT terminator)
110:    putstatic 11

// ATCalc class VFT
114:    const 99       // 'c' ⇔ 99
119:    putstatic 12   // ATCalc VFT start address (12 - static data)
122:    const 97       // 'a' ⇔ 97
127:    putstatic 13
130:    const 108      // 'l' ⇔ 108
135:    putstatic 14
138:    const 99       // 'c' ⇔ 99
143:    putstatic 15
146:    const_m1       // -1 (method name terminator)
151:    putstatic 16
154:    const 14       // adr(ATCalc::calc)
```

```
159:   putstatic 17

162:   const 116     // 't' ⇔ 116
167:   putstatic 18
170:   const 97      // 'a' ⇔ 97
175:   putstatic 19
178:   const 120     // 'x' ⇔ 120
183:   putstatic 20
186:   const_m1      // -1 (method name terminator)
187:   putstatic 21
190:   const 9       // adr(TCalc::tax)
193:   putstatic 22

196:   const 97      // 'a' ⇔ 97
201:   putstatic 23
203:   const 109     // 'm' ⇔ 109
208:   putstatic 24
211:   const 116     // 't' ⇔ 116
216:   putstatic 25
219:   const 122     // 'z' ⇔ 122
222:   putstatic 26
225:   const_m1      // -1 (method name terminator)
226:   putstatic 27
229:   const 26      // adr(ATCalc::amtz)
234:   putstatic 28
237:   const_m2      // -2 (ATCalc VFT terminator)
238:   putstatic 29

//main function bytecode
241:   const 1000
245:   store_3
246:   new 2
249:   store_0
250:   load_0
251:   const_0
252:   putfield_0    // init object's vftPtr
253:   load_0
256:   const 10
257:   putfield_1    // c1.tax = 10
258:   new 3
261:   store_1
262:   load_1
263:   const 12
266:   putfield_0    // init object's vftPtr
267:   load_1
268:   const 18
271:   putfield_1    // c2.tax = 18
272:   load_1
273:   const_2
274:   putfield_2    // c2.a = 2
275:   load_0     // First argument: this
276:   load_2     // Second argument: cost
277:   load_0
278:   getfield_0    // vftPtr -> Expr stack.
279:   invokevirtual 'c' 'a' 'l' 'c' -1 // each operand 4B
```

```
300:   print
301:   load_1     // First argument: this
302:   load_2     // Second argument: cost
303:   load_1
304:   getfield_0
305:   invokevirtual 'c' 'a' 'l' 'c' -1
326:   print
```

c) Prikaz sekvence bajtkoda koja generiše tabelu virtuelnih funkcija, a izvršava se neposredno pre prve instrukcije funkcije main. Posto u glavnom programu nije bilo globalnih promenljivih, tabela virtuelnih funkcija je samo iz tog razloga smeštena od adrese 0 u statičkoj zoni memorije.

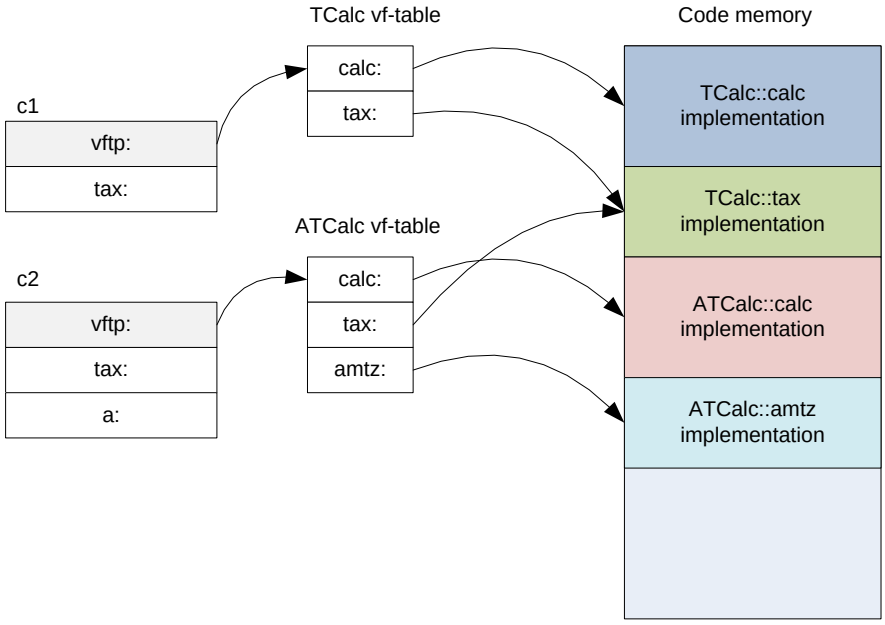
```
// Virtual Function Table initialization code.
// TCalc class VFT
31: const 99       // 'c' ⇔ 99
36: putstatic 0    // TCalc VFT start address (0 - static data)
39: const 97       // 'a' ⇔ 97
44: putstatic 1
47: const 108      // 'l' ⇔ 108
52: putstatic 2
55: const 99       // 'c' ⇔ 99
60: putstatic 3
63: const_m1       // -1 (method name terminator)
64: putstatic 4
65: const 0        // adr(TCalc::calc)
70: putstatic 5
73: const 116      // 't' ⇔ 116
78: putstatic 6
81: const 97       // 'a' ⇔ 97
86: putstatic 7
89: const 120      // 'x' ⇔ 120
94: putstatic 8
97: const_m1       // -1 (method name terminator)
98: putstatic 9
101:   const 9        // adr(TCalc::tax)
106:   putstatic 10
109:   const_m2      // -2 (TCalc VFT terminator)
110:   putstatic 11

// ATCalc class VFT
114:   const 99       // 'c' ⇔ 99
119:   putstatic 12   // ATCalc VFT start address (12 - static data)
122:   const 97       // 'a' ⇔ 97
127:   putstatic 13
130:   const 108      // 'l' ⇔ 108
135:   putstatic 14
138:   const 99       // 'c' ⇔ 99
143:   putstatic 15
146:   const_m1       // -1 (method name terminator)
151:   putstatic 16
154:   const 14       // adr(ATCalc::calc)
159:   putstatic 17
```

```
162:   const 116      // 't' ⇔ 116
167:   putstatic 18
170:   const 97       // 'a' ⇔ 97
175:   putstatic 19
178:   const 120      // 'x' ⇔ 120
183:   putstatic 20
186:   const_m1       // -1 (method name terminator)
187:   putstatic 21
190:   const 9        // adr(TCalc::tax)
193:   putstatic 22

196:   const 97       // 'a' ⇔ 97
201:   putstatic 23
203:   const 109      // 'm' ⇔ 109
208:   putstatic 24
211:   const 116      // 't' ⇔ 116
216:   putstatic 25
219:   const 122      // 'z' ⇔ 122
222:   putstatic 26
225:   const_m1       // -1 (method name terminator)
226:   putstatic 27
229:   const 26       // adr(ATCalc::amtz)
234:   putstatic 28
237:   const_m2       // -2 (ATCalc VFT terminator)
238:   putstatic 29
```

d) Izgled strukture objekata klasa i tabela virtuelnih funkcija.



Zadatak 2
Proširiti implementaciju parsera, tako da se omogući generisanje tabela virtuelnih funkcija za unutrašnje klase.
Rešenje:

Generisanje tabele virtuelnih funkcija za osnovnu klasu.

```
{:  List<Byte> MethodTable = new ArrayList();
    void addWordToStaticData (int value, int address)
    {
        MethodTable.add(new Byte((byte)Code.const_));
        MethodTable.add(new Byte( (byte)( (value>>16)>>8 ) ) );
        MethodTable.add(new Byte((byte)(value>>16)));
        MethodTable.add(new Byte((byte)(value>>8)));
        MethodTable.add(new Byte((byte)value));
        MethodTable.add(new Byte((byte)Code.putstatic));
        MethodTable.add(new Byte((byte)(address>>8)));
        MethodTable.add(new Byte((byte)address));
    }
    void addNameTerminator()
    {
        addWordToStaticData(-1, Code.dataSize++);
    }
    void addTableTerminator()
    {
        addWordToStaticData(-2, Code.dataSize++);
    }
    void addFunctionAddress(int functionAddress)
    {
        addWordToStaticData(functionAddress, Code.dataSize++);
    }
    void addFunctionEntry(String name, int functionAddressInCodeBuffer)
    {
        for (int j=0; j<name.length(); j++)
        {
            addWordToStaticData((int)(name.charAt(j)), Code.dataSize++);
        }
        addNameTerminator();
        addFunctionAddress(functionAddressInCodeBuffer);
    }
:}

class IDENT:id LBRACE LBRACE class_method_declaration_list RBRACE RBRACE
{:
    // atributi klase koji se nalaze u unutrasnjem opsegu se
      // pridruzuju polju fields struct cvora
    CurrentClass.type.fields=Tab.topScope.locals;
    CurrentClass.type.n=Tab.topScope.nVars;
    Obj ob=Tab.topScope.locals;
    // Skip fields.
    for (int i=0; i<CurrentClass.type.n; i++) ob=ob.next;
    while (ob!=null)
    {
        // Generate method entry.
        addFunctionEntry (ob.name, ob.adr);
        ob=ob.next;
    }
    // Generate VFT terminator (-2).
    addTableTerminator();
    Tab.closeScope();
:}
```

Generisanje tabele virtuelnih funkcija za izvedenu klasu.

```
CLASS IDENT:id EXTENDS IDENT:cl LBRACE
{:
    Struct newClassNode=new Struct(Struct.Class);
    newClassNode.n=0;
    newClassNode.fields=null;

    superclass=Tab.find(cl);
    // ubacuje se novi simbol i otvara novi opseg
    newClassObjNode=Tab.insert(Obj.Type, id, newClassNode);
    newClassObjNode.adr=Code.dataSize;
    newClassObjNode.type.elemType=superclass.type;

    Tab.openScope();
    Tab.topScope.nVars=0;
    if (superclass != Tab.noObj)
        if (superclass.kind != Obj.Type)
            Tab.error("Error in line "+
                (((Yylex)CUP$parser$parser.getScanner()).linija()+1)+
                " : "+cl+" is not a class name!");
        else
        {
            // Copy fields from the super class to the current scope.
            Obj ob = superclass.type.fields;
            for (int i=0; i<superclass.type.n; i++)
            {
                Obj o=Tab.insert(Obj.Fld, ob.name, ob.type);
                if (!Tab.duplicate) o.adr=Tab.topScope.nVars;
                ob=ob.next;
            }
        }
:}
LBRACE // Method definition start.
{:
    // Copy methods from the super class to the scope of subclass.
    newClassObjNode.type.n = Tab.topScope.nVars;
    Obj ob2 = superclass.type.fields;
    for (int i=0; i<superclass.type.n; i++) ob2=ob2.next;
    while (ob2!=null) {
        Obj o=Tab.insert(Obj.Meth, ob2.name, ob2.type);
        o.adr=ob2.adr;
        o.level=ob2.level;
        if (ob2.locals!=null)
            o.locals=ob2.locals.copyLocals();
        else
            o.locals=null;

        ob2=ob2.next;
    }
:}
 class_method_declaration_list RBRACE RBRACE // End of class definition.
 {:
    Obj ob=Tab.topScope.locals;
    for (int i=0; i<newClassObjNode.type.n; i++) ob=ob.next;
    while (ob!=null)
    {
        // Generate method entry.
        addFunctionEntry (ob.name, ob.adr);
        ob=ob.next;
    }
    addTableTerminator();// Generate VFT terminator (-2).
```

```
    // atributi klase koji se nalaze u unutrasnjem opsegu se
    // pridruzuju polju fields struct cvora
    newClassObjNode.type.fields=Tab.topScope.locals;
    Tab.closeScope();
    glob=true;
:}
```

Obrada deklaracije metode unutrašnje klase.

```
class_meth_header ::= return_type: rt IDENT: id
{:
Obj ob=Tab.topScope.locals;
override=false;
boolean g=false;
inner=true;
for (int i=0; i<CurrentClass.type.n; i++) ob=ob.next;
while (ob!=null)
{
    if (ob.name.equals(id))
        if (ob.type==rt)
        {
            ob.adr=Code.pc; // change the address field.
            ob.locals.type=CurrentClass.type; // adjust argument this.
            override=true;
            objekat=ob;
            Tab.overriddenMethod=ob;
            auxi2=objekat.locals.next;
            // Adjust references for checking argument list.
            if (auxi2==null)
                auxi1=objekat.locals;
            else
                auxi1=auxi2;
            check=1;
            q=true;
        }
        else
        { Tab.error ("Error Method "+id+
             " must have the same return type as method in superclass! ");
            q = false;
            g=true;
            objekat=Tab.noObj;
        }
        ob=ob.next;
    }
    if (override && q)
    {
        end=new Label();
        Code.put(Code.enter);
        Code.put(1);
        Code.put(1);
    }
    if (!override&&!g)
    {
        objekat=Tab.insert(Obj.Meth, id, rt);
        end=new Label();
        objekat.level=1;
        objekat.locals=null;
        bjekat.adr=Code.pc;
        q=true; // q je true samo ako je return type regularan
        Code.put(Code.enter);
        Code.put(1);
```

```
        Code.put(1);
        Tab.openScope();
        Tab.topScope.nVars = 0;
        ob = Tab.insert(Obj.Var, "this", CurrentClass.type);
        ob.level = 2;
        ob.adr = 0;
    }
:};
```

Kod za inicijalizaciju tabele virtuelnih funkcija.

```
ClassMethDecl ::= meth_header LPAREN RPAREN LBRACE
{:
    if (isMain)
    {
        // Store vft-creation byte code before the first instruction
        // in main function.
        Object ia[]=MethodTable.toArray();
        for (int i=0; i<ia.length; i++)
        Code.buf[Code.pc++]=((Byte)ia[i]).byteValue();
        MethodTable.clear();
    }
:} stmt_list RBRACE
{:
    objekat.locals=Tab.topScope.locals;
    Tab.closeScope();
    if ( ret )
        end.here();
    else if ( !ret )
    {
        Code.put(Code.trap);
        Code.put(1);
    }
    Code.put(Code.exit);
    Code.put(Code.return_);
    ret=false;
:}
```

Početak generisanja koda tela metode.

```
meth_header ::= return_type: rt IDENT: id
{:
    methodObjNode=Tab.insert(Obj.Meth, id, rt);
    end=new Label();
    methodObjNode.level=0;
    methodObjNode.locals=null;
    methodObjNode.adr=Code.pc;
    if ( rt==Tab.noType && id.equals("main") )
    {
        nMain++,
        Code.mainPc=Code.pc;
    }
    Code.put(Code.enter);
    Code.put(0); // Izmestiti na drugo mesto.
    Code.put(0); // Izmestiti na drugo mesto.
    Tab.openScope();
    Tab.topScope.nVars=0;
:};
```