

# Dodatak A. Programski jezik MikroJava

Ovaj dodatak opisuje programski jezik MikroJava koji se koristi u praktičnom delu kursa programskih prevodilaca (13E114PP1, 13S114PP1) 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, znakovne i logičke konstante (*int*, *char*, *bool*).
  - Osnovni tipovi: *int*, *bool*, *char* (Ascii)
  - Promenljive: globalne (statičke), lokalne, klasne (polja);
  - Promenljive osnovnih tipova sadrže vrednosti.
  - Strukturirani/referencijalni tipovi: jednodimenzionalni nizovi kao u Javi, unutrašnje klase (apstraktne i konkretne) sa poljima i metodama.
  - Promenljive referencijalnih tipova predstavljaju reference (sadrže adrese koje se ne mogu menjati eksplicitno);
  - Statičke metode u programu.
- Ne postoji garbage kolektor (alocirani objekti se samo dealociraju nakon kraja programa).
- Postoji nasleđivanje klasa i polimorfizam.
- Postoji redefinisane metoda.
- Metode unutrašnjih klasa su vezane za instancu i imaju implicitni parametar *this* (referenca na instancu klase za koju je pozvana metoda).
- Referenca "this" se implicitno deklarise u metodama unutrašnjih klasa kao prvi formalni argument tipa reference na klasu kojoj metoda pripada.
- Predeklarisane procedure su *ord*, *chr*, *len*.
- Metoda *print* ispisuje vrednosti svih osnovnih tipova.
- Od kontrolnih struktura postoji uslovno grananje (if-else), kao i ciklus (for).

## Primer programa

program P

```
const int size = 10;
abstract class Tab{
    int size;
    {
        abstract int getp(int i);
        abstract int getn(int i);
        int getSize() { return size;}
    }
}
class Table extends Tab{
    int pos[], neg[];
    {
        void putp (int a, int idx) { this.pos[idx]=a; }
        void putn (int a, int idx) { this.neg[idx]=a; }
        int getp (int idx) { return pos[idx]; }
        int getn (int idx) { return neg[idx]; }
    }
}
Table val;
{
void f(char ch, int a, int arg)
int x;
{
    x = arg;
}
void
main()
int x, i;
char c;
{ //----- Initialize val
val = new Table;
val.pos = new int [size];
val.neg = new int [size];
for (i = 0; i<size; i++)
{
    val.putp(0,i);
    val.putn(0,i);
}
f(c, x, i);

//----- Read values
read(x);
for (;x > 0;)
{
    if (0 <= x && x < size)
    {
        val.putp(val.getp(x)+1);
    } else
    if (-size < x && x < 0)
    {
        val.putn(val.getn(-x)+1);
    }
    read(x);
} }
}
```

## A.2 Sintaksa

Program	= "program" ident {ConstDecl   VarDecl   AbstractClassDecl   ClassDecl} "{" {MethodDecl} "}"
ConstDecl	= "const" Type ident "=" (numConst   charConst   boolConst) {, ident "=" (numConst   charConst   boolConst)} ";"
VarDecl	= Type ident "[" "[" "]" "{" , ident "[" "[" "]" } ";"
ClassDecl	= "class" ident ["extends" Type] "{" {VarDecl} [{" {MethodDecl} "}"] "}"
AbstractClassDecl	= "abstract" "class" ident ["extends" Type] "{" {VarDecl} [{" {MethodDecl}   AbstractMethodDecl} "}"] "}"
MethodDecl	= (Type   "void") ident "(" [FormPars] ")" {VarDecl} "{" {Statement} "}"
AbstractMethodDecl	= "abstract" (Type   "void") ident "(" [FormPars] ")" ";"
FormPars	= Type ident "[" "[" "]" "{" , Type ident "[" "[" "]" }
Type	= ident
Statement	= DesignatorStatement ";"   "if" "(" Condition ")" Statement ["else" Statement]   "for" "(" [DesignatorStatement] ";" [Condition] ";" [DesignatorStatement] ")" Statement   "break" ";"   "continue" ";"   "return" [Expr] ";"   "read" "(" Designator ")" ";"   "print" "(" Expr [ , numConst ] ")" ";"   "{" {Statement} "}"
DesignatorStatement	= Designator (Assignop Expr   "(" [ActPars] ")"   "++"   "--")
ActPars	= Expr { , Expr }
Condition	= CondTerm {    CondTerm }
CondTerm	= CondFact { && CondFact }
CondFact	= Expr [Relop Expr]
Expr	= ["-"] Term {Addop Term}
Term	= Factor {Mulop Factor}
Factor	= Designator ["(" [ActPars] ")"]   numConst   charConst   boolConst   "new" Type ["[" Expr "]" ]   "(" Expr ")"
Designator	= ident { "." ident   "[" Expr "]" }
Assignop	= "="
Relop	= "=="   "!="   ">"   ">="   "<"   "<="
Addop	= "+"   "-"
Mulop	= "*"   "/"   "%"

## Leksičke Strukture

Ključne reči:	program, break, class, abstract, else, const, if, new, print, read, return, void, for, extends, continue
Vrste tokena :	ident = letter {letter   digit   "_"}. numConst = digit {digit}. charConst = "" printableChar "" boolConst = ("true"   "false").
Operatori:	+, -, *, /, %, ==, !=, >, >=, <, <=, &&,   , =, ++, --, ,, zarez, ., (, ), [, ], {, }
Komentari:	// do kraja linije

### 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.
- Tip logičke konstante (npr. True) je bool.

#### 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

#### Ili Predeklarisana imena

int tip svih celobrojnih vrednosti

char tip svih znakovnih vrednosti

bool logički tip

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** - standardna metoda; chr(i) vrši konverziju celobrojnog izraza *i* u karakter (char)

**ord** - standardna metoda; ord(ch) vrši konverziju karaktera *ch* u celobrojnu vrednost (int)

#### Opseg važenja

Opseg važenja (*scope*) predstavlja tekstualni doseg metode ili klase. Prostire se od početka definicije metode ili klase do zatvorene velike zagrade na kraju te definicije. Opseg važenja ne uključuje imena koja su deklarisan u opsezima koji su leksički ugnježdeni unutar njega. U opsegu se “vide” imena deklarisan unutar njega i svih njemu spoljašnjih opsega. Pretpostavka je da postoji veštački globalni opseg (universe), za koji je glavni program lokalni i koji sadrži sva predeklarisana imena.

Deklaracija imena u unutrašnjem opsegu *S* sakriva deklaraciju istog imena u spoljašnjem opsegu.

#### **Napomena**

- Indirektna rekurzija nije dozvoljena i svako ime mora biti deklarisan pre prvog korišćenja.
- Predeklarisana imena (npr. int ili char) mogu biti redeklarisani u unutrašnjem opsegu (ali to nije preporučljivo).

## A.4 Kontekstni uslovi

### Opšti kontekstni uslovi

- Svako ime u programu mora biti deklarirano pre prvog korišćenja.
- Ime ne sme biti deklarirano više puta unutar istog opsega.
- U programu mora postojati metoda sa imenom *main*. Ona mora biti deklarirana 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 | AbstractClassDecl} "{" {MethodDecl} "}"`.

---

**ConstDecl** = `"const" Type ident "=" (numConst | charConst | boolConst) ";"`.

- Tip terminala *numConst*, *charConst* ili *boolConst* 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.
  - Klasa koja nije apstraktna ne sme sadržati apstraktne metode.
  - Ukoliko klasa koja nije apstraktna nasleđuje apstraktnu klasu, mora implementirati sve apstraktne metode.
- 

**AbstractClassDecl** = `"abstract" "class" ident ["extends" Type] "{" {VarDecl} [{" {MethodDecl | AbstractMethodDecl} "}"] "}"`.

- Apstraktna klasa može nasleđivati neku drugu apstraktnu ili klasu koja nije apstraktna.
  - Apstraktna klasa se ne može instancirati.
- 

**AbstractMethodDecl** = `"abstract" (Type | "void") ident "(" [FormPars] ")" ";"`.

---

**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** = `DesignatorStatement ";"`.

---

**DesignatorStatement** = `Designator Assignop Expr ";"`.

- *Designator* mora označavati promenljivu, element niza ili polje unutar objekta.
- Tip neterminala *Expr* mora biti kompatibilan pri dodeli sa tipom neterminala *Designator*.

---

**DesignatorStatement = Designator ("++" | "--") ";"**.

- *Designator* mora označavati promenljivu, element niza ili polje objekta unutrašnje klase.
  - *Designator* mora biti tipa int.
- 

**DesignatorStatement = Designator "(" [ActPars] ")" ";"**.

- *Designator* mora označavati nestatičku metodu unutrašnje klase ili globalnu funkciju glavnog programa.
- 

**Statement = "break".**

- Iskaz break se može koristiti samo unutar *for* petlje. Prekida izvršavanje neposredno okružujuće petlje.
- 

**Statement = "continue".**

- Iskaz continue se može koristiti samo unutar *for* petlje. Prekida tekuću iteraciju neposredno okružujuće petlje.
- 

**Statement = "read" "(" Designator ")" ";"**.

- *Designator* mora označavati promenljivu, element niza ili polje unutar objekta.
  - *Designator* mora biti tipa int, char ili bool.
- 

**Statement = "print" "(" Expr ["," number] ")" ";"**.

- *Expr* mora biti tipa int, char ili bool.
- 

**Statement = "return" [Expr] .**

- Tip neterminala *Expr* mora biti ekvivalentan povratnom tipu tekuće metode/ globalne funkcije.
  - Ako neterminal *Expr* nedostaje, tekuća metoda mora biti deklarirana kao void.
  - Ne sme postojati izvan tela (statičkih) 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.
  - Tip uslovnog izraza Condition mora biti bool.
- 

**Statement="for" "(" [DesignatorStatement]; [Condition] ";" [DesignatorStatement] ")" Statement**

- Uslovni izraz Condition mora biti tipa bool.
  - Ako je navedena, prvo se izvršava naredba opisana prvim neterminalom DesignatorStatement, i to samo jednom (nije deo cikličnog izvršavanja).
  - Na početku svake iteracije proverava vrednost uslovnog izraza. Ukoliko ne postoji, podrazumeva se true. Vrednost true omogućava izvršavanje tela petlje (Statement).
  - Po završetku tela petlje (osim ako nije break), uvek se izvršava naredba opisana drugim neterminalom DesignatorStatement, a zatim ponovo proverava vrednost uslovnog izraza.
- 

**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}.**

- Potrebno je implementirati optimalno izračunavanje složenih kondicionala. Ukoliko je na osnovu parcijalnog dela složenog kondicionala poznato da će ceo uslov biti prihvaćen ili odbijen, ne izračunava se ostatak kondicionala.

---

**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 komatibilni.
  - Ako je *Addop* kombinovani aritmetički operator (+, -), *Expr* mora označavati promenljivu, element niza ili polje unutar objekta.
- 

**Term = Factor.**

---

**Term = Term Mulop Factor.**

- *Term* i *Factor* moraju biti tipa int.
  - Ako je *Mulop* kombinovani aritmetički operator (\*, /, %), *Term* mora označavati promenljivu, element niza ili polje unutar objekta.
- 

**Factor = Designator | numConst | charConst | boolConst | "(" Expr ")".**

---

**Factor = Designator "(" [ActPars] ")".**

- *Designator* mora označavati nestatičku 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* može biti unutrašnja 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.
- 

**Assignop = "=".**

Operator dodele vrednosti je desno asocijativan.

---

**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 polja.
- Izvorni kod programa ne sme biti veći od 8 KB.

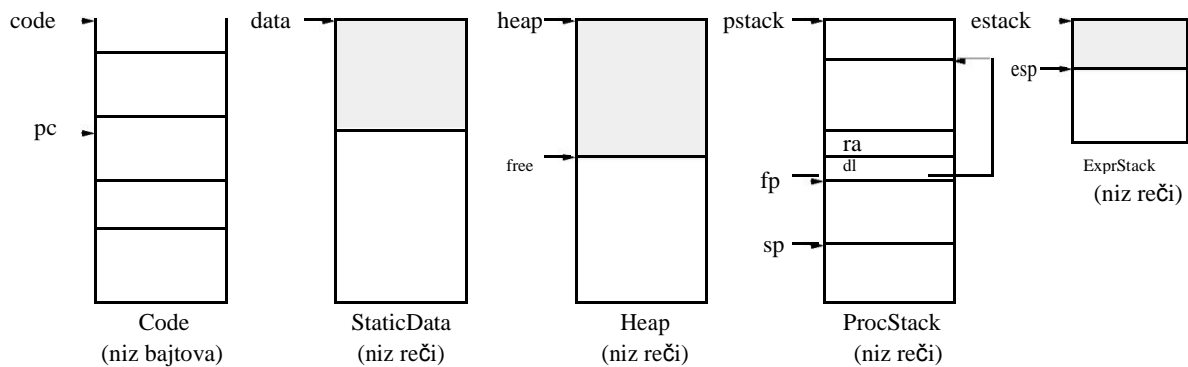


## Dodatak B. MikroJava VM

Ovaj dodatak opisuje arhitekturu MikroJava virtuelne mašine koja se koristi u praktičnom delu kursa programskih prevodilaca (IR4PP1) 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 <i>pc</i> se nalazi indeks instrukcije koja se trenutno izvršava. Registar <i>mainpc</i> sadrži početnu adresu metode <i>main()</i> .
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. <i>free</i> 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 <i>char</i> 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. <i>ra</i> je povratna adresa metode, <i>dl</i> je dinamička veza (pokazivač na aktivacioni zapis pozivaoca metode). Novoalocirani zapis se inicijalizuje nulama.
ExprStack	Ova oblast se koristi za skladištenje operanada instrukcija. <i>ExprStack</i> je prazan posle svake MikroJava instrukcije. Argumenti metoda se prosleđuju na stek izraza i kasnije uklanjaju <i>Enter</i> 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

<i>opcode</i>	<i>instr.</i>	<i>opds</i>	<i>ExprStack</i>	<i>značenje</i>
1	<b>load</b>	b	... ..., val	<u>Load</u> push(local[b]);
2..5	<b>load_n</b>		... ..., val	<u>Load</u> (n = 0..3) push(local[n]);
6	<b>store</b>	b	..., val ...	<u>Store</u> local[b] = pop();
7..10	<b>store_n</b>		..., val ...	<u>Store</u> (n = 0..3) local[n] = pop();

### Instrukcije za load i store globalnih promenljivih

11	<b>getstatic</b>	s	... ..., val	<u>Load statičke promenljive</u> push(data[s]);
12	<b>putstatic</b>	s	..., val ...	<u>Store statičke promenljive</u> data[s] = pop();

### Instrukcije za load i store polja objekata

13	<b>getfield</b>	s	..., adr ..., val	<u>Load polja objekta</u> adr = pop()/4; push(heap[adr+s]);
14	<b>putfield</b>	s	..., adr, val ...	<u>Store polja objekta</u> val = pop(); adr = pop()/4; heap[adr+s] = val;

## Instrukcije za load konstanti

15..20	<b>const_n</b>	...	<u>Load konstante (n = 0..5)</u> push(n)
21	<b>const_m1</b>	..., -1	<u>Load konstante -1</u> push(-1)
22	<b>const</b>	w ..., val	<u>Load konstante</u> push(w)

## Aritmetičke operacije

23	<b>add</b>	..., val1, val2 ..., val1+val2	<u>Sabiranje</u> push(pop() + pop());
24	<b>sub</b>	..., val1, val2 ..., val1-val2	<u>Oduzimanje</u> push(-pop() + pop());
25	<b>mul</b>	..., val1, val2 ..., val1*val2	<u>Množenje</u> push(pop() * pop());
26	<b>div</b>	..., val1, val2 ..., val1/val2	<u>Deljenje</u> x = pop(); push(pop() / x);
27	<b>rem</b>	..., val1, val2 ..., val1%val2	<u>Ostatak pri celobrojnem deljenju</u> x = pop(); push(pop() % x);
28	<b>neg</b>	..., val ..., - val	<u>Promena predznaka</u> push(-pop());
29	<b>shl</b>	..., val ..., val1	<u>Aritmetičko pomeranje ulevo</u> x = pop(); push(pop() << x);
30	<b>shr</b>	..., val ..., val1	<u>Aritmetičko pomeranje udesno</u> x = pop(); push(pop() >> x);
31	<b>inc</b>	b1, b2 ...	<u>Inkrementiranje</u> local[b1] = local[b1] + b2;

## Pravljenje objekata

32	<b>new</b>	s ..., adr	<u>Novi objekat</u> alocirati oblast od s bajtova; inicijalizovati oblast nulama; push(adr(oblast));
33	<b>newarray</b>	b ..., n ..., adr	<u>Novi niz</u> 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	<b>aload</b>	..., adr, index ..., val	<u>Load elementa niza</u> (+ provera indeksa) i = pop(); adr = pop()/4+1; push(heap[adr+i]);
35	<b>astore</b>	..., adr, index, val ...	<u>Store elementa niza</u> (+ provera indeksa) val = pop(); i = pop(); adr = pop()/4+1; heap[adr+i] = val;
36	<b>baload</b>	..., adr, index ..., val	<u>Load elementa niza bajtova</u> (+ provera indeksa) i = pop(); adr = pop()/4+1; x = heap[adr+i/4]; push(byte i%4 of x);
37	<b>bastore</b>	..., adr, index, val ...	<u>Store elementa niza bajtova</u> (+ 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	<b>arraylength</b>	..., adr ..., len	<u>Dohvatanje dužine niza</u> adr = pop(); push(heap[adr]);

## Operacije na steku

39	<b>pop</b>	..., val ...	<u>Skidanje elementa sa vrha steka</u> dummy = pop();
40	<b>dup</b>	..., val ..., val, val	<u>Udvajanje elementa na vrhu steka</u> x = pop(); push(x); push(x);
41	<b>dup2</b>	..., v1, v2 ..., v1, v2, v1, v2	<u>Udvajanje prva dva elementa na vrhu steka</u> y = pop(); x = pop(); push(x); push(y); push(x); push(y);

## Skokovi

Adresa skoka je relativna u odnosu na početak instrukcije skoka.

42	<b>jmp</b>	s	<u>Bezuslovni skok</u> pc = pc + s;
43..48	<b>j&lt;cond&gt;</b>	s ..., val1, val2 ...	<u>Uslovni skok</u> (eq, ne, lt, le, gt, ge) y = pop(); x = pop(); if (x cond y) pc = pc + s;

11

## Pozivi metoda (PUSH i POP se odnose na stek procedura)

49	<b>call</b>	s	<u>Poziv metode</u> PUSH(pc+3); pc := pc + s;
50	<b>return</b>		<u>Povratak iz metode</u> pc = POP();

```
51      enter b1, b2
```

## Početak obrade metode

```
psize = b1; lsize = b2; // u rečima
PUSH(fp); fp = sp; sp = sp + lsize;
inicijalizovati akt. zapis svim
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);  
// vrši 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  W1,W2,...,Wn,Wn+1  ..., adr
                                     ...

```

## Poziv virtualne metode

ime metode ima n znakova;  
 ovi znakovi su deo same instrukcije, i nalaze se u rečima  $w_1, w_2, \dots, w_n$ ;  
 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.

## Kombinovani operatori

```
59  dup_x1    ..,val2, val1    ...,val1, val2, val1
```

Umetanje kopije vršne vrednosti ispod druge vrednosti sa vrha steka izraza.

vrednost sa vrha steka se kopira i ubacuje ispod druge vrednosti sa vrha steka izraza.

```
60      dup_x2    val1, val2, val3    ...,val3, val1, val2, val3
                                ...
```

Umetanje kopije vršne vrednosti ispod treće vrednosti sa vrha steka izraza.  
vrednost sa vrha steka se kopira i ubacuje ispod treće vrednosti sa vrha steka izraza.

### 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.