Programski jezik ***miniC*** specifikacija i kompajler

Zorica Suvajdžin Rakić
Miroslav Hajduković

septembar 2015

Sadržaj

Predgovor

Praktikum "Programski jezik mini C - specifikacija i kompajler" upoznaje čitaoca sa podskupom programskog jezika C, nazvanog miniC, i implementacijom kompajlera za ovaj jezik. Materijal je napisan za potrebe kursa *Programski prevodioci* na Fakultetu tehničkih nauka, Univerziteta u Novom Sadu.

miniC jezik je minimalni podskup programskog jezika C, dovoljan da omogući edukativni prikaz implementacije kompajlera. miniC kompajler prevodi sa miniC jezika na hipotetski asemblerski jezik. Kompajler je implementiran pomoću alata `flex` i `bison`. Sav dodatni kod je napisan na programskom jeziku C.

Praktikum se sastoji iz dva dela. Prvi deo sadrži specifikaciju miniC jezika sa objašnjenjem sintakse i semantike i primerima miniC koda. Drugi deo praktikuma sadrži implementaciju miniC kompajlera u fazama: leksička, sintaksna, semantička analiza i generisanje asemblerskog koda. Na kraju svakog poglavlja su navedene vežbe za dodatni rad.

Kompletni primeri su numerisani, sa naznačenim imenom datoteke u kojoj se nalaze. Neproporcionalnim fontom, radi bolje preglednosti, navedeni su primeri, pokretanja programa i gramatika miniC jezika.

Zahvalnica

Autori se, na prvom mestu, zahvaljuju dr Žarku Živanovu, koji je svojim idejama i predlozima značajno doprineo unapređenju miniC kompajlera.

Autori se zahvaljuju dr Predragu Rakiću na korisnim savetima i promišljenim sugestijama u toku izrade materijala.

Autori se zahvaljuju Tari Petrić na ukazanim propustima i vrlo korisnim sugestijama u toku izrade materijala.

Autori se, na kraju (ali ne i najmanje) zahvaljuju studentima koji su imali u rukama *draft* verziju praktikuma na kursu *Programski prevodioci* i koji su našli brojne slovne i druge greške u prvim verzijama materijala. Njihovi komentari i sugestije su značajno uticale na oblik i sadržaj praktikuma.

Autori

Deo I

miniC jezik

Poglavlje 1

miniC jezik

miniC je podskup programskog jezika C. Dakle, miniC programe je moguće kompajlirati regularnim C kompajlerom. miniC je nastao odabirom osobina i koncepata C jezika koji su interesantni za kurs implementacije kompajlera. Međutim, ove osobine su uzete u određenoj meri, tako da olakšaju implementaciju jezika. Autori su se uzdržali od mnogih karakteristika C-a koji nepotrebno komplikuju implementaciju jezika, a koji, u edukativnom smislu, ne doprinose značajno.

U ovom delu, dat je opis miniC jezika (njegova sintaksa i semantika) i primer programa. Kompletna gramatika miniC jezika je navedena na kraju prvog dela (deo 1.4).

1.1 Leksika

Leksika se bavi opisivanjem osnovnih gradivnih elemenata jezika. Za prirodne jezike, to su reči, a za programske jezike, to su simboli. String simbola se zove leksema. Skup leksema i pravila njihovog formiranja predstavljaju leksiku jezika.

U miniC jeziku postoji nekoliko vrsta simbola. To su identifikatori i označeni i neoznačeni celobrojni literali.

Identifikator u miniC jeziku je malo pojednostavljen u odnosu na C jezik, pa se sastoji od malih slova, velikih slova i cifara, i ne sme započinjati cifrom. Primeri ispravnih miniC identifikatora su:

```
a, A, abc, number, Number, MyNumber, num2, num2str
```

a neispravnih: `3a`, `_x`, `my_number`.

Celobrojni označeni literal se sastoji od predznaka plus ili minus iza kojeg sledi jedna ili više cifara, dok se neoznačeni celobrojni literal sastoji od jedne ili više cifara iza kojih sledi malo ili veliko slovo “u”. Primeri literala su:

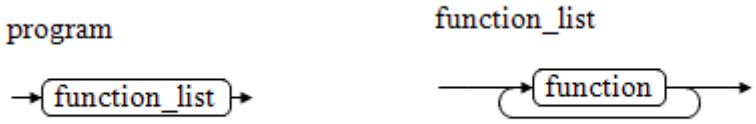
```
0, -123, 12345
5u, 1234U
```

1.2 Sintaksa

Sintaksa opisuje skup pravila za kombinovanje simbola u ispravne jezičke konstrukcije. Sintaksa se formalno opisuje gramatikom (vidi deo 1.4). U objašnjenjima sintaksnih konstrukcija, u zagradama je navedeno ime pojma u gramatici koje odgovara toj konstrukciji.

1.2.1 Program

Najmanji miniC program ima bar jednu funkciju.



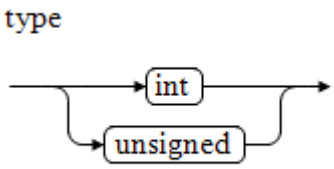
Slika 1.1: Sintagram programa i liste funkcija

Da bi program mogao da započne izvršavanje potrebna mu je `main()` funkcija. Povratna vrednost funkcije `main()` je celobrojnog tipa (`int`).

```
int main() {
  ...
}
```

1.2.2 Tipovi

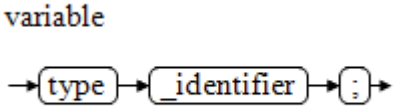
miniC jezik podržava samo `int` i `unsinged` tipove podatka, tj. označene i neoznačene celobrojne vrednosti.



Slika 1.2: Sintagram tipa podatka

1.2.3 Promenljive

Deklaracija promenljive (*variable*) sadrži ime tipa i ime promenljive a završava se separatorom ";".



Slika 1.3: Sintagram deklaracije promenljive

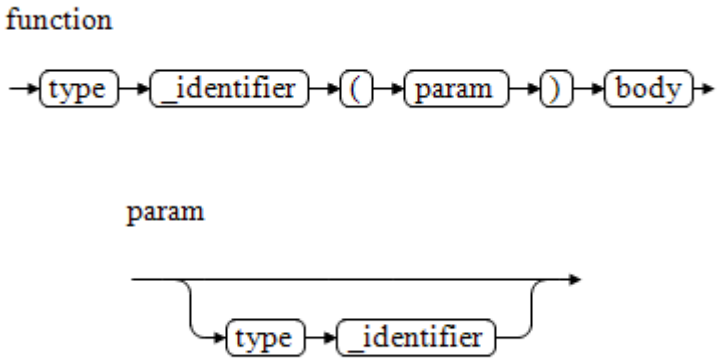
U jednoj miniC deklaraciji, moguće je deklarisati samo jednu promenljivu. Da bi se deklarisale dve promenljive, potrebno je napisati dve deklaracije. U primeru, prve tri deklaracije su ispravne, dok poslednja nije:

```
int counter;
int line;
unsigned n;

int counter, line;  //error
```

1.2.4 Funkcija

Zaglavlje miniC funkcije (*function*) se sastoji od tipa povratne vrednosti funkcije, imena funkcije i malih zagrada u kojima se može, a ne mora, navesti (jedan) parametar funkcije.



Slika 1.4: Sintagram definicije funkcije i parametra

Primeri sintaksno ispravnih miniC funkcija su dati u primeru:

```
int f() {
  ...
}

int f(unsigned a) {
  ...
}
```

Iza zaglavlja se navode vitičaste zagrade u kojima se piše telo funkcije (*body*). Telo funkcije se sastoji od deklaracija lokalnih promenljivih (*variable_list*) i od iskaza (*stmt_list*), u tom redosledu. Telo može, a ne mora, da sadrži lokalne promenljive. Isto tako, može a ne mora da sadrži iskaze, što znači da funkcija može imati prazno telo. Slede ispravni primeri miniC funkcija:

```
int f() {
}

int f(int a) {
  return a;
}

int f() {
  int x;
}

unsigned f() {
```

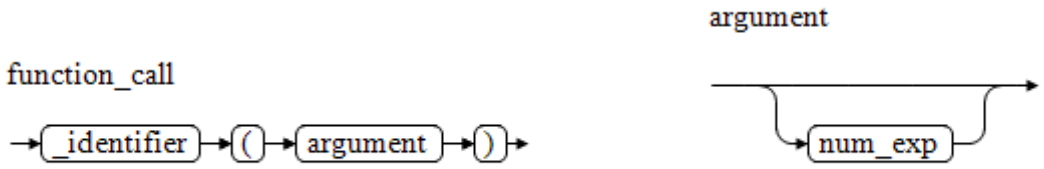
```
  unsigned x;
  x = 0u;
}
```

Unutar tela funkcije, prvo se navode deklaracije lokalnih promenljivih, pa tek onda iskazi. To znači, da nije ispravno napisati deklaraciju lokalne promenljive posle nekog iskaza:

```
int f() {
  int x;
  x = 0;
  int y;    //error
  y = 0;
}
```

1.2.4.1 Poziv funkcije

Poziv miniC funkcije (*function_ call*) se sastoji od imena funkcije i malih zagrada u kojima se može, a ne mora, navesti (jedan) argument.



Slika 1.5: Sintagram poziva funkcije

Slede (i ispravni i neispravni) primeri poziva miniC funkcija:

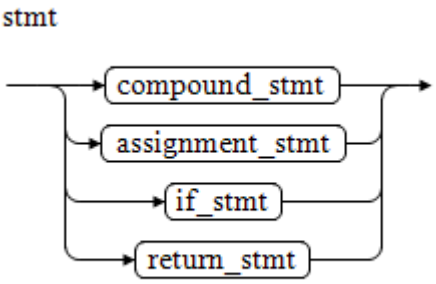
```
unsigned f0() {
  return 0u;
}

int f1(int a) {
  return a + a;
}

int f() {
  unsigned x;
  int y;
  x = f0();
  x = f0(5u);  //error, f0 nema parametar
  y = f1();    //error, f1 ima parametar
  y = f1(3);
}
```

1.2.5 Iskazi

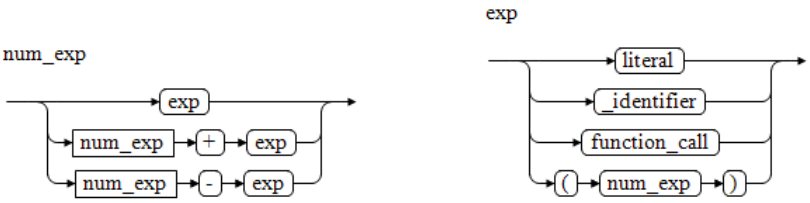
Iskazi koji mogu da se pojave u telu funkcije su: iskaz dodele, `if` iskaz, `return` iskaz i blok iskaza. U iskazima se koriste numerički izrazi.



Slika 1.6: Sintagram iskaza

1.2.5.1 Numerički izraz

Numerički izraz (*num_ exp*) se gradi od literala, promenljivih i poziva funkcija. Od aritmetičkih operacija, podržano je samo sabiranje i oduzimanje. Numerički izraz se može pisati i u malim zagradama.



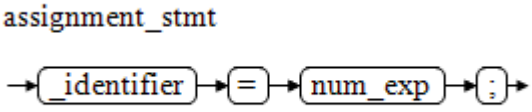
Slika 1.7: Sintagrami izraza

Primeri ispravnih miniC numeričkih izraza su:

```
a
0
line + 1
j + (k - 4)
f()
2 + f()
```

1.2.5.2 Iskaz dodele

Sa leve strane znaka jednako u iskazu dodele (*assignment_ stmt*) se može naći promenljiva, a sa desne strane numerički izraz.



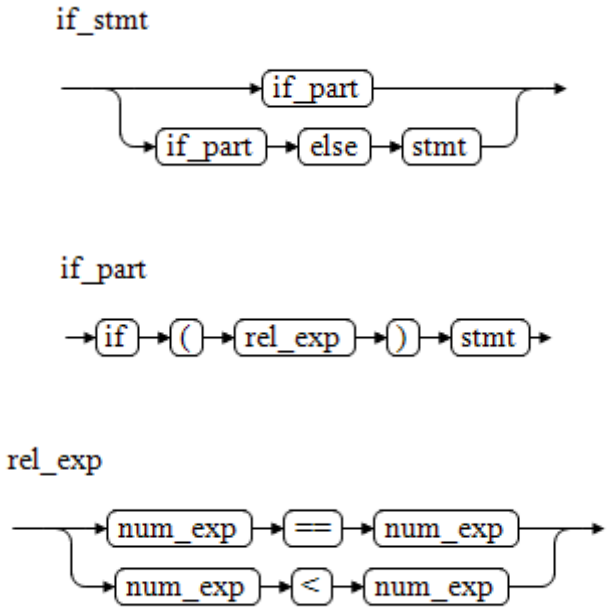
Slika 1.8: Sintagram iskaza dodele

Primeri pravilnih iskaza dodele u miniC jeziku su:

```
a = b;
counter = 0;
line = line + 1;
c = j + (k - 4);
m = f();
```

1.2.5.3 `if` iskaz

U `if` iskazu (*if_stmt*) se prvo navodi ključna reč `if`, zatim uslov u malim zagradama i onda iskaz. Opciono, `if` iskaz može da sadrži i `else` deo koji se sastoji od ključne reči `else` i iskaza. Uslov `if` iskaza, u miniC jeziku, sadrži relacioni izraz. Relacije koje su podržane u miniC jeziku su < i ==.



Slika 1.9: Sintagram `if` iskaza i relacionog izraza

Primeri pravilnih `if` iskaza su:

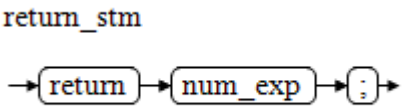
```
if(a < b)
  a = b;

if(a < b) {
  a = b;
}

if(a == b)
  counter = counter + 1;
else
  counter = counter - 1;
```

1.2.5.4 `return` iskaz

`return` iskaz (*return_stmt*) očekuje da se iza ključne reči `return` navede numerički izraz, čija će vrednost biti povratna vrednost funkcije u kojoj se nalazi.



Slika 1.10: Sintagram `return` iskaza

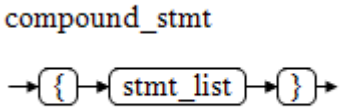
Primeri `return` iskaza u miniC jeziku su dati u sledećem primeru (prva 3 iskaza su ispravna, a poslednji nije):

```
return 0;
return a + 1 - (d + 5);
return f3(a);

return;  //error: return iskaz mora sadrzati izraz
```

1.2.5.5 Blok iskaza

Blok iskaza (*compound_ stmt*) čine iskazi napisani u vitičastim zagradama. Blok može biti i prazan.



Slika 1.11: Sintagram bloka iskaza

Primeri pravilnih blokova u miniC jeziku su:

```
{ }
{ a = 8; }
{
  if(a == b)
    counter = 0;
  a = 10;
}
```

U bloku se ne može deklarisati promenljiva, pa zato prva linija unutar bloka u sledećem primeru nije nedozvoljena u miniC jeziku:

```
{
  int a;  //error: u bloku se ne moze
          //deklarisati promenljiva
  a = 8;
}
```

1.2.6 Komentari

Komentari u miniC jeziku započinju sa dva znaka *slash* `//`, a u nastavku, do kraja linije (do znaka `\n`), sledi tekst komentara. Primer pravilnog komentara:

```
if(a == b)  // tekst komentara
  a = -b;
```

1.2.7 Primer miniC programa

miniC je dosta ograničen programski jezik, ali uprkos svojim ograničenjima može da se koristi za programiranje malih programa. Na primer, miniC program za izračunavanje apsolutne vrednosti označenog broja bi mogao da izgleda ovako:

```
int abs(int i) {
  int res;
  if(i < 0)
    res = 0 - i;
  else
    res = i;
  return res;
}

int main() {
  return abs(-5);
}
```

1.3 Semantika

Semantika jezika opisuje značenje sintaksno ispravnih konstrukcija jezika. Kod koji je sintaksno ispravan, ne mora biti i semantički ispravan. Na primer, ako se poziva funkcija koja nije prethodno definisana, to predstavlja semantičku grešku. Iako postoje mnogi formalizmi za zapis sematike programskih jezika neretko se za te potrebe u oblasti razvoja kompajlera koriste rečenice prirodnog jezika, tj. neformalni zapis.

1.3.1 Standardni identifikatori

Standardni identifikatori su: rezervisane reči (`int`, `unsigned`, `if`, `else`, `return`) i identifikator `main`. Identifikator `main` je ime funkcije, za koju se podrazumeva da je definisana u izvršivom miniC programu. Izvršavanje miniC programa započinje izvršavanjem `main` funkcije. Definicija ove funkcije izgleda:

```
int main () {
   ...
}
```

Telo `main` funkcije definiše korisnik. Ako telo `main` (kao i svake druge) funkcije ne sadrži `return` iskaz, podrazumeva se da je povratna vrednost funkcije nedefinisana i da do povratka iz funkcije dolazi po izvršavanju poslednjeg iskaza iz njenog tela.

1.3.2 Opseg vidljivosti (*scope*)

Prema opsegu vidljivosti (područje važenja, doseg, vidljivost), identifikatori se razvrstavaju u globalne i lokalne.

Globalni identifikatori su imena funkcija. Oni su definisani na nivou programa (van funkcija). Opseg vidljivosti globalnih identifikatora je od mesta njihove definicije do kraja programskog teksta.

Lokalni identifikatori su imena lokalnih promenljivih i parametara i oni su definisani u okviru funkcija. Opseg vidljivosti lokalnih identifikatora je od mesta njihove definicije do kraja tela funkcije u kojoj su definisani. Znači, svaka funkcija poseduje svoje lokalne promenljive.

Identifikatori mogu biti korišćeni samo iza njihove definicije (to proizlazi iz opsega njihovog važenja).

Može se desiti da se isto ime deklariše u nekoliko ugnježdenih opsega vidljivosti. U tom slučaju, uobičajeno je da se koristi deklaracija imena koja je najbliža datoj upotrebi imena. Primer C iskaza:

```
{
  int x = 1;     //prva promenljiva x
  int y = 2;
  {
    int x = 3;   //druga promenljiva x
    y += x;
  }
  y += x;
}
```

U primeru postoje dve različite promenljive `x`, deklarisane u dva različita opsega vidljivosti. Druga promenljiva `x` je sakrila vidljivost prve. Zato će vrednost promenljive `y` na kraju ovog dela programa biti `6`.

1.3.3 Jednoznačnost identifikatora

Svi globalni identifikatori moraju biti međusobno različiti i svi lokalni identifikatori iste funkcije moraju biti međusobno različiti. Ako postoje identični globalni identifikatori i lokalni identifikatori neke funkcije, tada van te funkcije važe globalni, a unutar nje lokalni identifikatori.

Lokalni identifikatori raznih funkcija mogu biti identični.

Rezervisane reči smeju da se koriste samo u skladu sa svojom ulogom i na globalnom i na lokalnom nivou. Standardni identifikator `main` je rezervisan samo na globalnom nivou.

1.3.4 Provera tipova

Na upotrebu identifikatora utiče njegov tip. Na primer, tip identifikatora s leve strane iskaza pridruživanja određuje tip izraza sa desne strane ovog iskaza (leva i desna strana iskaza pridruživanja moraju imati isti tip).

Tip izraza iz `return` iskaza neke funkcije i tip ove funkcije moraju biti identični.

Tipovi korespondentnih parametara funkcije i argumenata iz njenog poziva moraju biti identični. Argumenti poziva funkcije moraju da se slažu po broju sa parametrima funkcije.

U istom relacionom izrazu smeju biti samo identifikatori istog tipa.

1.4 miniC gramatika

Gramatika miniC jezika je napisana korišćenjem BNF notacije. Vertikalna crta se koristi da ukaže na alternativna pravila i čita se "ili". Koristimo konvenciju da simboli jezika (*terminals*) započinju donjom crtom "_" da bi se razlikovali od pojmova (*nonterminals*).

Simboli (*terminals*)

```
        _letter ::= "a"|"A"|"b"|"B"|"c"|"C"|"d"|"D"
                  |"e"|"E"|"f"|"F"|"g"|"G"|"h"|"H"
                  |"i"|"I"|"j"|"J"|"k"|"K"|"l"|"L"
                  |"m"|"M"|"n"|"N"|"o"|"O"|"p"|"P"
                  |"q"|"Q"|"r"|"R"|"s"|"S"|"t"|"T"
                  |"u"|"U"|"v"|"V"|"w"|"W"|"x"|"X"
                  |"y"|"Y"|"z"|"Z"
         _digit ::= "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"
                  |"8"|"9"
    _identifier ::= letter ( letter | digit )*
   _int_literal ::= digit +
  _uint_literal ::= digit + ( "u" | "U" )
```

Pojmovi (*nonterminals*)

```
       program ::= function_list
  function_list ::= function
                  | function_list function
      function ::= type _identifier "(" param ")" body
          type ::= "int"
                  | "unsigned"
         param ::=
                  | type _identifier
          body ::= "{" variable_list stmt_list "}"
  variable_list ::=
                  | variable_list variable
      variable ::= type _identifier ";"
     stmt_list ::=
                  | stmt_list stmt
          stmt ::= compound_stmt
                  | assignment_stmt
                  | if_stmt
                  | return_stmt
  compound_stmt ::= "{" stmt_list "}"
assignment_stmt ::= _identifier "=" num_exp ";"
       num_exp ::= exp
                  | num_exp "+" exp
                  | num_exp "-" exp
           exp ::= literal
                  | _identifier
                  | function_call
                  | "(" num_exp ")"
       literal ::= _int_literal
                  | _uint_literal
  function_call ::= _identifier "(" argument ")"
      argument  ::=
                  | num_exp
       if_stmt ::= if_part
                  | if_part "else" stmt
```

```
    if_part ::= "if" "(" rel_exp ")" stmt
    rel_exp ::= num_exp "==" num_exp
              | num_exp "<" num_exp
return_stmt ::= "return" num_exp ";"
```

Razmak i kraj linije imaju funkciju separatora simbola.

Deo II

miniC kompajler (MICKO)

Glava 2

Karakteristike kompajlera

MICKO je kompajler koji prevodi sa jezika miniC (izvorni jezik) na hipotetski asemblerski jezik (ciljni jezik). Ime je nastalo kao akronim od **mi**ni**C** **ko**mpajler. MICKO je napravljen isključivo u edukativne svrhe. Implementiran je pomoću generatora skenera i parsera: *flex* i *bison* [1]. Sav dodatni kod je napisan na programskom jeziku C. Tokom kompajliranja, kompajler prijavljuje korisniku eventualno postojanje grešaka u izvornom miniC programu.

Slika 2.1: MICKO kompajler

MICKO je vrlo jednostavna implementacija miniC jezika, nastala kao odgovor na potrebe kursa *Programski prevodioci*, Fakulteta tehničkih nauka u Novom Sadu. Osnovni cilj je edukacija, a ne efikasnost koda (ovo se naročito odnosi na implementaciju tabele simbola).

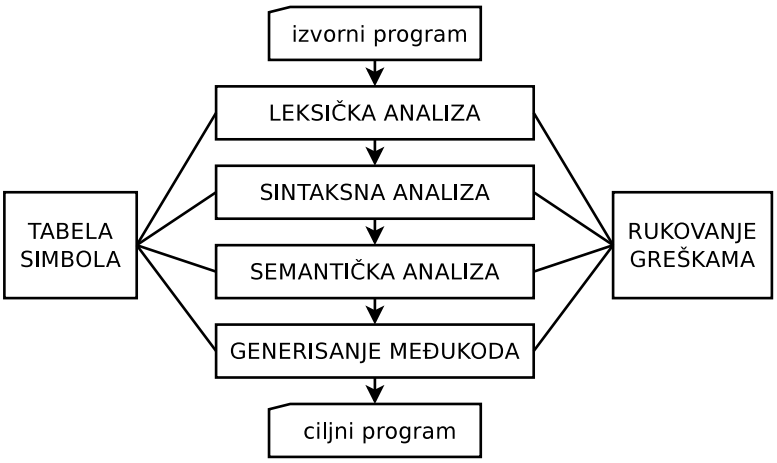
2.1 Faze kompajliranja

Implementacija miniC kompajlera je podeljena na uobičajene faze kompajliranja: leksičku, sintaksnu, semantičku analizu i generisanje koda. Optimizacija koda nije implementirana.

Leksička analiza je početni deo čitanja i analiziranja programskog teksta. Tekst se čita i deli na simbole programskog jezika (npr. ključna reč, ime promenljive ili broj).

Sintaksna analiza preuzima tokene koji su stvoreni tokom leksičke analize i proverava da li su navedeni u ispravnom redosledu. Ova faza se naziva parsiranje (*parsing*). Tokom parsiranja se pravi stablo koje se naziva stablo parsiranja (*parse tree*) koje odražava strukturu programa.

Semantička analiza proverava konzistentnost programa: npr. da li je promenljiva, koja se koristi, prethodno deklarisana i da li se koristi u skladu sa svojim tipom.



Slika 2.2: Faze kompajliranja

Generisanje međukoda prevodi program na jednostavan međujezik (*intermediate language*) koji je nezavisan od ciljne mašine (*machine-independent*), obično asemblerski jezik [2], u ovom slučaju hipotetski asemblerski jezik.

2.2 Tabela simbola

Sve faze kompajliranja koriste tabelu simbola. To je struktura podataka u kojoj se čuvaju sve informacije o svim simbolima koji su prepoznati u toku kompajliranja. Na osnovu ovih informacija moguće je uraditi semantičku analizu i generisanje asemblerskog koda. Na primer, uz ime funkcije, potrebno je čuvati i informaciju o tipu povratne vrednosti te funkcije (da bi se mogla uraditi provera tipova u `return` iskazu), broj parametara i tipovi parametara (da bi se mogla proveriti ispravnost broja argumenata i njihovih tipova prilikom poziva te funkcije).

2.3 Greške u kompajliranju

Tokom svih faza kompajliranja moguća je pojava greške u izvornom kodu, pa kompajler treba da pomoge programeru da ih identifikuje i locira. Programi mogu sadržati greške na različitim nivoima:

- leksičke (npr. pogrešno napisano ime, ključna reč ili operator),
- sintaksne (npr. relacioni izraz sa nepotpunim parom zagrada) ili
- semantičke (npr. operator primenjen na nekompatibilni operand).

Kompajler treba da:

- saopšti prisustvo grešaka jasno i ispravno,
- da se oporavi od greške dovoljno brzo da bi mogao da detektuje naredne greške i
- da ne usporava bitno obradu ispravnih programa.

2.4 Primer prevođenja

Kada se miniC kompajleru prosledi datoteka `abs.mc` (listing 2.1) sa funkcijom `abs()` koja računa apsolutnu vrednost broja, kompajler izgeneriše datoteku `abs.asm` koja sadrži ekviva-

lentan kod, napisan na hipotetskom asemblerskom jeziku (listing 2.2):

```
$./micko <abs.mc
$
```

Listing 2.1: abs.mc

```
1  int abs(int i) {
2    int res;
3    if(i < 0)
4      res = 0 - i;
5    else
6      res = i;
7    return res;
8  }
9
10 int main() {
11   return abs(-5);
12 }
```

Listing 2.2: abs.asm

```
abs:
        PUSH  %14
        MOV   %15,%14
        SUBS  %15,$4,%15
@abs_body:
@if1:
        CMPS  8(%14),$0
        JGES  @false1
@true1:
        SUBS  $0,8(%14),%0
        MOV   %0,-4(%14)
        JMP   @exit1
@false1:
        MOV   8(%14),-4(%14)
@exit1:
        MOV   -4(%14),%13
        JMP   @abs_exit
@abs_exit:
        MOV   %14,%15
        POP   %14
        RET
main:
        PUSH  %14
        MOV   %15,%14
@main_body:
        PUSH  $-5
        CALL  abs
        ADDS  %15,$4,%15
        MOV   %13,%0
        MOV   %0,%13
        JMP   @main_exit
@main_exit:
        MOV   %14,%15
        POP   %14
        RET
```

Pošto datoteka abs.mc sadrži leksički, sintaksno i semantički ispravan kod, kompajler je uspešno preveo program i izgenerisao datoteku sa ekvivalentnim asemblerskim programom. Ukoliko bi se kompajleru prosledila datoteka sa neispravanim kodom, on bi korisniku prijavio grešku. Za greške koje je kompajler predvideo, prijaviće tačnu lokaciju i detaljan opis greške i nastaviće parsiranje (npr. ulaz 2 i ulaz 4). Za greške koje nije predvideo, kompajler će ispisati samo poruku da postoji greška i završiće kompajliranje (npr. ulaz 1 i ulaz 3).

2.4.1 Neispravan ulaz 1

Na primer, ako bi se miniC kompajleru prosledio prethodni program, ali sa izmenjenom relacijom u if iskazu (linija 3, operator < zamenjen operatorom >):

```
if(i > 0)
  ...
```

kompajler bi prijavio leksičku grešku (a nakon toga i sintaksnu), jer miniC jezik ne podržava relacioni operator >:

```
$./micko <abs.mc
line 3: LEXICAL ERROR on char >
line 3: ERROR: syntax error
$
```

2.4.2 Neispravan ulaz 2

Ukoliko bi se miniC kompajleru prosledio `return` iskaz kojem nedostaje separator ";" na kraju iskaza (linija 7):

```
return res
```

kompajler bi prijavio sintaksnu grešku, ali bi se i oporavio od nje i nastavio parsiranje:

```
$./micko <abs.mc
line 8: ERROR: Missing ';' in return statement
$
```

2.4.3 Neispravan ulaz 3

Međutim, ukoliko bi mu se prosledilo zaglavlje funkcije u kojem iza definicije parametra nedostaje zatvorena mala zagrada (linija 1):

```
int abs(int i {
```

kompajler bi prijavio sintaksnu grešku i završio parsiranje, jer nema oporavak od ovakve sintaksne greške:

```
$./micko <abs.mc
line 1: ERROR: syntax error
$
```

2.4.4 Neispravan ulaz 4

Ako bi se u `abs` programu, u liniji 6, izmenilo ime promenljive iz `res` u `re`:

```
re = i;
```

kompajler bi prijavio semantičku grešku na poziciji sa leve strane znaka jednako (linija 6):

```
$./micko <abs.mc
line 6: ERROR: invalid lvalue 're' in assignment
$
```

Glava 3

Leksička analiza

Leksički analizator (skener) je program namenjen za leksičku analizu (skeniranje) teksta. Zadatak skenera je da pročita ulazni tekst i da ga podeli na simbole. Skener radi tako što čita tekst, karakter po karakter, i pokušava da prepozna neku od zadatih reči. Ukoliko naiđe na simbol čija pojava nije predviđena, treba da prijavi leksičku grešku.

Na primer, ako se u ulaznoj miniC datoteci nalazi kod:

```
if(a == 6)
  a = -b;
```

skener bi trebao da prepozna sledeće simbole:

`if` ključna reč

`(` leva mala zagrada

`a` identifikator

`==` relacioni operator

`6` celobrojni literal

`)` desna mala zagrada

`a` identifikator

`=` operator dodele

`-` operator unarni minus

`b` identifikator

`;` separator

Beline (u koje spadaju prazno mesto, tabulator i karakter za novi red) se preskaču, odnosno prepoznaju i ignorišu, jer imaju ulogu separatora simbola.

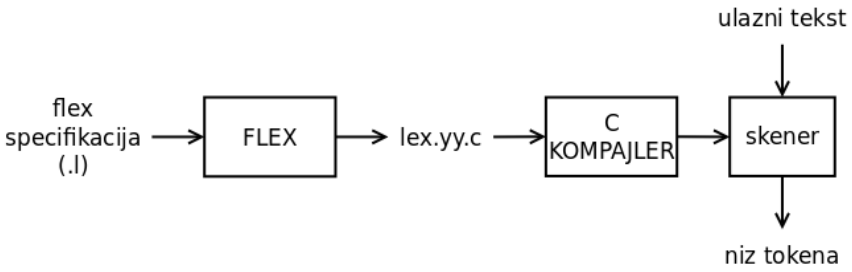
3.1 Implementacija skenera

Skener se može napraviti (programirati) ručno ili korišćenjem alata za generisanje skenera. Tokom ovog kursa se za generisanje skenera koristi program `flex`. `Flex` generiše skener na osnovu pravila zadatih u `flex` specifikaciji. Specifikaciju kreira korisnik u posebnoj datoteci koja po konvenciji ima ekstenziju `.l`. Ova datoteka sadrži pravila koja opisuju reči koje

skener treba da prepozna. Pravila se zadaju u obliku regularnih izraza. Svakom regularnom izrazu moguće je pridružiti akciju (u obliku `C` koda) koja će se izvršiti kada skener prepozna dati regularni izraz. Uobičajena akcija je vraćanje oznake simbola, odnosno tokena za taj simbol. Token je numerička oznaka klase (vrste) simbola. Na primer: token `NUMBER` označava bilo koji broj (i broj `2` i broj `359`), dok token `IF` označava jedino ključnu reč `if`.

`.l` datoteka se prosleđuje `flex`-u koji generiše skener na jeziku C, u funkciji `yylex()` u datoteci `lex.yy.c`. Akcije, pridružene regularnim izrazima u `.l` datoteci, su delovi `C` koda koji se direktno prenose (kopiraju) u `lex.yy.c`. Ovako generisan C kod se dalje prevodi C kompajlerom (tj. može da se uključi u C program). Ovakav program predstavlja leksički analizator koji transformiše ulazni tekst u niz tokena (u skladu sa pravilima zadatim u specifikaciji).

Korišćenje programa `flex` je prikazano na slici 3.1:



Slika 3.1: Korišćenje `flex`-a

Prilikom izvršavanja, skener će tražiti pojavu stringa u ulaznom tekstu koji odgovara nekom regularnom izrazu. Ako ga pronađe, izvršiće akciju (kod) koju je korisnik pridružio tom regularnom izrazu. U suprotnom, podrazumevana akcija za tekst koji ne odgovara ni jednom regularnom izrazu je kopiranje teksta na standardni izlaz.

Skener se generiše, kompajlira i pokreće naredbama:

```
$ flex scanner.l
$ gcc -o scanner lex.yy.c
$ ./scanner
```

U prvoj liniji se pokreće `flex` koji preuzima specifikaciju iz datoteke `scanner.l` i generiše skener u datoteci `lex.yy.c`.

U drugoj liniji se poziva `gcc` kompajler sa prosleđenom datotekom `lex.yy.c` (u kojoj se u funkciji `yylex()` nalazi skener). Svič `--o` prosleđen `gcc`-u omogućava korisniku da iza sviča navede ime programa koji će biti izgenerisan. Ukoliko se ne upotrebi ovaj svič, kompajler će napraviti izvršnu datoteku sa imenom `a.out`.

U trećoj liniji naredbom `./scanner` se izgenerisani skener pokreće u interaktivnom režimu, što znači da ulaz očekuje preko tastature, a da se program završava kada se na ulazu pojavi kombinacija `^D` (`CTRL+D`).

Ukoliko skener treba da skenira neku datoteku (na primer datoteku sa imenom `test.txt`), prilikom pokretanja programa treba upotrebiti redirekciju standardnog ulaza:

```
$ ./scanner <test.txt
```

3.2 miniC skener

Skener za miniC jezik je implementiran u datoteci `scanner.l` i dat na listinzima 3.1, 3.2 i 3.3.

Prvi deo specifikacije započinje uključivanjem dve opcije koje utiču na to kako će *flex* napraviti skener (listing 3.1):

```
%option noyywrap yylineno
```

`noyywrap` opcija znači da će skener skenirati samo jednu datoteku, i

`yylineno` opcija znači da će skener koristiti globalnu promenljivu *flex-a* `yylineno` koja sadrži broj trenutno skenirane linije.

Za neke simbole se, osim tokena, prosleđuje još neka vrednost koja bliže opisuje simbol (npr. za simbol *broj* se uz token `NUMBER` šalje i vrednost konkretnog broja, recimo `123`). U ovakvim situacijama, za smeštanje vrednosti, koja može biti različitog tipa - za različite simbole, se koristi unija. Unija se definiše u prvom delu specifikacije (listing 3.1), a koristi u drugom delu specifikacije (listing 3.2). Za miniC skener, potrebna je unija koja sadrži jedan celobrojni tip (`int i`) i jedan pokazivač na karakter (`char *s`). Promenljiva tipa unije, preko koje se, uz token, prosleđuje vrednost, se u *flex-u* zove `yylval`.

Prvi deo specifikacije sadrži enumeraciju `token`s sa definicijama tokena i niz `token_strings` koji sadrži imena tokena u obliku stringova koji se koriste prilikom ispisa tokena. U prvom delu je definisana i enumeracija `values` sa konstantama koje opisuju razne operatore i, opet, niz imena tih konstanti `value_strings` u obliku stringova (listing 3.1). Svi stringovi će se koristiti u `main` funkciji, prilikom prikazivanja prepoznatih tokena i vrednosti.

Listing 3.1: `scanner.l` - tokeni

```
%option noyywrap yylineno
%{
  union {
    int i;
    char *s;
  } yylval;

  enum tokens { _TYPE = 1, _IF, _ELSE, _RETURN,
                _LPAREN, _RPAREN, _LBRACKET, _RBRACKET,
                _SEMICOLON, _ASSIGN, _AROP, _RELOP, _ID,
                _INT_NUMBER, _UINT_NUMBER };
  char *token_strings[] = { "NONE", "_TYPE", "_IF",
                "_ELSE", "_RETURN", "_LPAREN","_RPAREN",
                "_LBRACKET", "_RBRACKET", "_SEMICOLON",
                "_ASSIGN", "_AROP", "_RELOP", "_ID",
                "_INT_NUMBER", "_UINT_NUMBER" };

  enum values { ADD, SUB, LT, EQ, INT, UINT};
  char *value_strings[] = { "ADD", "SUB",
                            "LT",  "EQ",
                            "INT", "UINT" };
```

Prvo pravilo preskače beline (listing 3.2). Sledeća tri pravila prepoznaju ključne reči `if`, `else` i `return`. Za ključne reči dovoljno je vezati samo tokene (nisu potrebne dodatne vrednosti koje bi ih detaljnije definisale).

Sledeća dva pravila prepoznaju označene i neoznačene celobrojne tipove. Za oba simbola se vezuje isti token (`_TYPE`), ali različite vrednosti uz token. Vrednosti su konstante (`INT` i `UINT`) koje opisuju prepoznate tipove (listing 3.2). Vrednost se prosleđuje kroz promenljivu `yylval` koja je definisana u prvom delu specifikacije kao unija (listing 3.1). Kako je konstanta celobrojnog tipa, njena vrednost se smešta u polje `i` promenljive `yylval`.

Dalje su navedena pravila koja prepoznaju delimitere (`(`, `)`, `{`, `}`), separatore (`;`) i operator dodele (`=`), koji su potputno opisani samo tokenima (listing 3.2).

Zatim slede pravila za prepoznavanje aritmetičkih i relacionih operatora (listing 3.2). Za ove operatore se, osim tokena, prosleđuje još i odgovarajuća konstanta kao vrednost simbola. Konstanta je potrebna da bi se operatori razlikovali međusobno, jer imaju isti token (na primer: konstanta `ADD` za operator `+`).

Zatim slede pravila za prepoznavanje identifikatora i literala. U ovim slučajevima se, pored tokena, prosleđuje još i string simbola, koji se smešta u polje `s` (tipa `char*`) promenljive `yylval` (listing 3.2). String prepoznatog simbola flex čuva u promenljivoj `yytext`. Regularni izraz za identifikator opisuje string koji počinje slovom (malim ili velikim: `[a-zA-Z]`), a iza njega se može naći slovo (malo ili veliko) ili cifra, 0 ili više puta: `[a-zA-Z0-9]*`. Minimalni identifikator se, znači, sastoji od jednog slova. Regularni izraz za označeni celobrojni literal opisuje string od minimalno jedne do maksimalno deset cifara: `[0-9]{1,10}`, ispred kojih se može, a ne mora, naći predznak plus ili minus: `[+-]?`. Regularni izraz za neoznačeni celobrojni literal ne prepoznaje predznak, već dozvoljava da se iza broja navede malo ili veliko slovo "u": `[uU]`.

U skeneru nam je bitan redosled pravila, tako da pravila za prepoznavanje ključnih reči treba staviti ispred pravila za identifikator (listing 3.2).

Linijski komentari se prepoznaju pomoću regularnog izraza koji prepoznaje dva znaka *slash* na početku stringa: `\/\/` (može i: `"//"`), a zatim bilo koji znak osim `\n` znaka, 0 ili više puta: `.*`.

Poslednje pravilo koje sadrži operator `.` (bilo koji znak osim `\n` znaka) služi da prepozna leksičku grešku. Bilo koji znak koji ne pripada nijednom prethodnom pravilu predstavlja grešku, koju treba saopštiti korisniku.

Listing 3.2: `scanner.l` - pravila

```
%%

[ \t\n]+        { /* skip */ }

"if"            { return _IF; }
"else"          { return _ELSE; }
"return"        { return _RETURN; }

"int"           { yylval.i = INT;  return _TYPE; }
"unsigned"      { yylval.i = UINT; return _TYPE; }

"("             { return _LPAREN; }
")"             { return _RPAREN; }
"{"             { return _LBRACKET; }
"}"             { return _RBRACKET; }
";"             { return _SEMICOLON; }
"="             { return _ASSIGN; }

"+"             { yylval.i = ADD; return _AROP; }
"-"             { yylval.i = SUB; return _AROP; }

"<"             { yylval.i = LT; return _RELOP; }
"=="            { yylval.i = EQ; return _RELOP; }

[a-zA-Z][a-zA-Z0-9]*  { yylval.s = yytext;
                        return _ID; }
```

```
[+-]?[0-9]{1,10}       { yylval.s = yytext;
                         return _INT_NUMBER;}
[0-9]{1,10}[uU]        { yylval.s = yytext;
                         return _UINT_NUMBER;}

\/\/.*          { /* skip */ }
.               { printf("line␣%d:␣LEXICAL␣ERROR␣"
                  "on␣char␣%c\n",yylineno, *yytext);}
```

Funkcija `main()` preuzima od skenera (funkcije `yylex()`) tokene i vrednosti simbola i prikazuje ih korisniku (listing 3.3). Za ispis se koriste stringovi iz niza `token_strings` i `value_strings` koji su definisani u prvom delu *flex* specifikacije (listing 3.1).

Listing 3.3: `scanner.l - main() funkcija`

```
%%

int main() {
  int tok;
  while(tok = yylex()) {
    printf("%s", token_strings[tok]);
    switch(tok) {
      case _ID:
      case _INT_NUMBER:
      case _UINT_NUMBER:
        printf(":␣%s", yylval.s); break;
      case _AROP:
      case _RELOP:
      case _TYPE:
        printf(":␣%s", value_strings[yylval.i]);break;
    }
    printf("\n");
  }
}
```

3.2.1 Primer upotrebe skenera

Ako bismo ovaj program pokrenuli sa redirekcijom standardnog ulaza na datoteku `test.c` sa listinga 3.4 i redirekcijom standardnog izlaza na datoteku `out.c`, njen sadržaj bi bio kao na listingu 3.5:

```
$ ./scanner <test.c >out.c
```

Listing 3.4: `test.c`

```
int boolval(int x) {
  if(x == 0)
    return 0;
  else
    return 1;
}
```

Listing 3.5: `out.c`

```
_TYPE: INT_TYPE
_ID: boolval
_LPAREN
_TYPE: INT_TYPE
_ID: x
_RPAREN
_LBRACKET
_IF
_LPAREN
_ID: x
_RELOP: EQ
_INT_NUMBER: 0
_RPAREN
_RETURN
_INT_NUMBER: 0
_SEMICOLON
_ELSE
_RETURN
_INT_NUMBER: 1
_SEMICOLON
_RBRACKET
```

Skener je ulaznu datoteku podelio na simbole, pri čemu je zanemario beline i komentare.

Ako bismo napisali uslov `if` iskaza sa nepodržanim relacionim operatorom `!=` umesto `==` (listing 3.6), skener bi nam prijavio leksičku grešku (listing 3.7):

Listing 3.6: `test2.c`

```
if(x != 0)
  return 0;
```

Listing 3.7: `out2.c`

```
_IF
_LPAREN
_ID: x
line 1: LEXICAL ERROR on char !
_ASSIGN
_INT_NUMBER: 0
_RPAREN
_RETURN
_INT_NUMBER: 0
_SEMICOLON
```

3.3 Vežbe

1. Proširiti miniC jezik i miniC skener `while` iskazom.
2. Proširiti miniC jezik i miniC skener `break` iskazom.
3. Proširiti miniC jezik i miniC skener `for` iskazom.
4. Proširiti miniC jezik i miniC skener `switch` iskazom.
5. Proširiti miniC jezik i miniC skener nizovima.
6. Proširiti relacione izraze miniC jezika i proširiti miniC skener.
7. Proširiti miniC jezik logičkim izrazima `&&` i `||` i proširiti miniC skener.
8. Proširiti miniC jezik i miniC skener definicijama funkcija sa više parametara, i pozivima funkcija sa više argumenata.

Glava 4

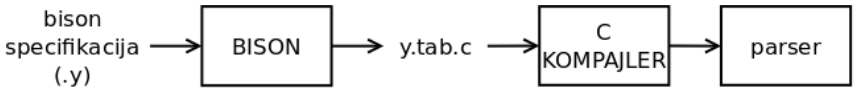
Sintaksna analiza

Sintaksa jezika opisuje pravila po kojima se kombinuju simboli jezika (npr. u deklaraciji promenljive, prvo se piše ime tipa, zatim ime promenljive i na kraju ";"). Sintaksa se opisuje gramatikom, obično pomoću BNF notacije.

Sintaksna analiza ima zadatak da proveri da li je ulazni tekst sintaksno ispravan. Ona to čini tako što preuzima niz tokena od skenera i proverava da li su tokeni navedeni u ispravnom redosledu. Ako jesu, to znači da je ulazni tekst napisan u skladu sa pravilima gramatike korišćenog jezika, tj. da je sintaksno ispravan. U suprotnom, sintaksna analiza treba da prijavi sintaksnu grešku i da nastavi analizu. Opisani proces se vrši u delu kompajlera koji se zove parser i naziva se parsiranje.

4.1 Implementacija parsera

Za implementaciju parsera, često se koristi generator parsera `bison`. Korišćenje `bison`-a je prikazano na slici 4.1.



Slika 4.1: Korišćenje `bison`-a

Prvi korak u generisanju parsera je priprema njegove specifikacije. Specifikaciju kreira korisnik u posebnoj datoteci koja po konvenciji ima ekstenziju `.y`. Ova datoteka sadrži gramatiku koju parser treba da prepozna. Pravila se zadaju u BNF obliku. Svakom pravilu je moguće pridružiti akciju (u obliku `C` koda) koja će se izvršiti kada parser prepozna dato pravilo.

`.y` datoteka se prosleđuje `bison-u`, koji kao izlaz, generiše `C` parser u funkciji yyparse() u datoteci `y.tab.c`. Ova datoteka se prosleđuje `C` kompajleru da bi se dobio program. Dobijeni program predstavlja parser koji proverava sintaksnu ispravnost ulaznog teksta na osnovu zadatih pravila gramatike.

Prilikom izvršavanja, parser komunicira sa skenerom tako što od njega traži sledeći token iz ulaznog teksta. Kada dobije token, parser proverava da li je njegova pojava na datom mestu dozvoljena (tj. da li je u skladu sa gramatikom). Ako jeste nastaviće parsiranje, a ukoliko nije, prijaviće grešku, pokušaće da se oporavi od greške i da nastavi parsiranje. U momentu kada prepozna celo pravilo, parser će izvršiti akciju koja je pridružena tom pravilu.

`Bison` i `flex`, odnosno izgenerisani skener i parser, se mogu udružiti u jedan program. Generisanje skenera i parsera i pokretanje programa se odvija ovako:

```
$ bison -d syntax.y
$ flex scanner.l
$ gcc -o syntax syntax.tab.c lex.yy.c
$ ./syntax <test.c
```

Prvo se pokrene `bison` koji napravi parser u datotekama `syntax.tab.c` i `syntax.tab.h`. Opcija `-d` generiše **d**efinicije tokena u `.h` datoteci. Zatim se pokrene `flex` koji napravi skener u datoteci `lex.yy.c`. Na kraju se pozove `gcc` kompajler da napravi izvršni program `syntax`. Poslednja linija sadrži pokretanje programa sa ulaznom datotekom `test.c`.

4.2 miniC parser

Skener za miniC jezik je već viđen u prethodnom poglavlju knjige (vidi listinge 3.1, 3.2 i 3.3). Drugi deo njegove *flex* specifikacije se može ovde iskoristiti u kombinaciji sa parserom za miniC. Treći deo njegove specifikacije ovde nije od interesa, jer će `main()` funkciju sadržati miniC parser.

Sve konstante koje su potrebne skeneru i parseru su smeštene u posebnoj datoteci (`defs.h`, listing 4.1) radi lakšeg održavanja koda.

Listing 4.1: `defs.h`

```
//tipovi podataka
enum types { NO_TYPE, INT, UINT };

//konstante aritmetickih operatora
enum arops { ADD, SUB, MUL, DIV, AROP_NUMBER };

//konstante relacionih operatora
enum relops { LT, GT, LE, GE, EQ, NE, RELOP_NUMBER };
```

Enumeracija `types` sadrži konstante koje opisuju tipove. miniC jezik podržava samo `int` i `unsigned` tipove.

Enumeracija `arops` sadrži konstante koje opisuju aritmetičke operatore sabiranja, oduzimanja, množenja i deljenja, a enumeracija `relops` sadrži konstante koje opisuju relacione operatore (<, >, <=, >=, == i !=). Iako miniC jezik podržava samo operacije sabiranja i oduzimanja, i relacione operatore < i ==, ovde su navedene konstante za sve aritmetičke i relacione operatore jezika C, da bi olakšali studentima proširivanje postojećeg miniC kompajlera.

Specifikacija miniC jezika pripremljena za `bison` je data na listingu 4.2.

Tip globalne promenljive `yylval` je definisan kao unija koja sadrži jedan ceo broj (`i`) i jedan pokazivač na string (`s`). Polja unije su baš ovakva, jer se uz tokene, iz miniC skenera, prosleđuju vrednosti koje su ili celobrojnog tipa ili string.

Tokeni koje šalje skener, a koje prima parser, se moraju definisati pomoću ključne reči `%token`. Uz svaki token se može definisati tip vrednosti koja se prosleđuje sa njim, između operatora < i >. Tokeni koji imaju vrednosti celobrojnog tipa su `_TYPE`, `_AROP` i `_RELOP` (njihova vrednost je konstanta koja opisuje vrstu tipa, aritmetičkog, odnosno relacionog operatora), a tokeni koji imaju vrednost tipa string su `_ID`, `_INT_NUMBER` i `_UINT_NUMBER` (njihova vrednost je string imena ili broja).

U drugom delu `bison` specifikacije navedena je gramatika miniC programskog jezika, kao što je navedeno u delu 1.4. To znači da će parser proveravati sintaksnu ispravnost miniC programa (listing 4.2).

Listing 4.2: `syntax.y`

```
%{
    #include <stdio.h>
    #include "defs.h"

    int yyparse(void);
    int yylex(void);
    int yyerror(char *s);
    extern int yylineno;
%}

%union {
   int i;
   char *s;
}

%token <i> _TYPE
%token _IF
%token _ELSE
%token _RETURN
%token <s> _ID
%token <s> _INT_NUMBER
%token <s> _UINT_NUMBER
%token _LPAREN
%token _RPAREN
%token _LBRACKET
%token _RBRACKET
%token _ASSIGN
%token _SEMICOLON
%token <i> _AROP
%token <i> _RELOP

%%

program
   :  function_list
   ;

function_list
   :  function
   |  function_list function
   ;

function
   :  type _ID _LPAREN parameter _RPAREN body
   ;

type
   :  _TYPE
   ;

parameter
   :  /* empty */
   |  _TYPE _ID
   ;
```

```
body
   :  _LBRACKET variable_list statement_list _RBRACKET
   ;

variable_list
   :  /* empty */
   |  variable_list variable
   ;

variable
   :  type _ID _SEMICOLON
   ;

statement_list
   :  /* empty */
   |  statement_list statement
   ;

statement
   :  compound_statement
   |  assignment_statement
   |  if_statement
   |  return_statement
   ;

compound_statement
   :  _LBRACKET statement_list _RBRACKET
   ;

assignment_statement
   :  _ID _ASSIGN num_exp _SEMICOLON
   ;

num_exp
   :  exp
   |  num_exp _AROP exp
   ;

exp
   :  literal
   |  _ID
   |  function_call
   |  _LPAREN num_exp _RPAREN
   ;

literal
   :  _INT_NUMBER
   |  _UINT_NUMBER
   ;

function_call
   :  _ID _LPAREN argument _RPAREN
   ;

argument
   :  /* empty */
   |  num_exp
   ;
```

```
if_statement
   :  if_part
   |  if_part _ELSE statement
   ;

if_part
   :  _IF _LPAREN rel_exp _RPAREN statement
   ;

rel_exp
   :  num_exp _RELOP num_exp
   ;

return_statement
   :  _RETURN num_exp _SEMICOLON
   ;

%%

int yyerror(char *s) {
   fprintf(stderr, "line␣%d:␣ERROR:␣%s\n", yylineno, s);
   return 0;
}

int main() {
   return yyparse();
}
```

Funkcija `main()` jedino poziva parser (tj. funkciju `yyparse()`).

Čim naiđe na prvu sintaksnu grešku, parser će završiti parsiranje, jer nije implementiran nikakav oporavak od greške.

4.2.1 Primer upotrebe parsera

Ako parseru sa listinga 4.2 prosledimo datoteku `abs.mc` (listing 2.1), parser neće ispisati ništa, jer ova datoteka sadrži sintaksno ispravan miniC program:

```
$./syntax <abs.mc
$
```

Međutim, ako izmenimo prvu liniju, tako što obrišemo tip parametra `int`:

```
int abs(i) {
```

kompajler će prijaviti sintaksnu grešku, jer simboli koji su navedeni u ovoj liniji ne čine sintaksno ispravno zaglavlje funkcije:

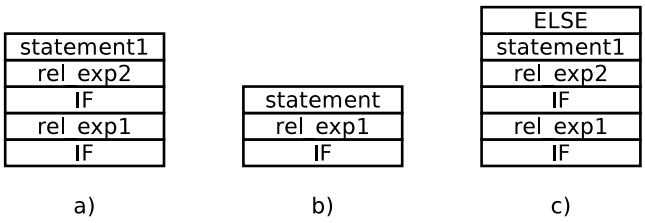
```
$./syntax <abs.mc
line 1: ERROR: syntax error
$
```

4.2.2 IF-ELSE konflikt

Nakon kompajliranja primera sa listinga 4.2, `bison` će prijaviti postojanje jednog konflikta, koji se javlja zbog `if` pravila:

```
$ make
bison -d syntax.y
syntax.y: conflicts: 1 shift/reduce
flex syntax.l
gcc -o syntax lex.yy.c syntax.tab.c
$
```

`Bison` je detektovao stanje u kom može da uradi i *shift* i *reduce* akciju [3] (da preuzme token ili da uradi redukciju po nekom pravilu), a nema na osnovu čega da odluči koju akciju da izvrši. Ova situacija se desi kada se na steku zatekne sledeće stanje `if rel_exp1 if <*> rel_exp2 statement1` (slika 4.2.a) i ukoliko parser kao sledeći token dobija `else` (i posle toga njemu pripadajući iskaz `statement2`).



Slika 4.2: IF-ELSE konflikt (stanje na steku stanja)

U tom trenutku se može obaviti:

1. *reduce* akcija nad poslednja 3 elementa steka (slika 4.2.b), nakon čega će se `else` token i njemu pripadajući iskaz `statement2` priključiti prvom `if` iskazu, ili se može obaviti
2. *shift* akcija, u kojoj se na stek prebacuje `else` token (slika 4.2.c), nakon čega će uslediti iskaz `statement2` na steku, a tada je moguća redukcija poslednja 4 elementa sa steka. Time će se `else` token i njemu pripadajući iskaz `statement2` priključiti drugom `if` iskazu.

Kada se `bison` nalazi u stanju u kom mora nešto da odluči (*shift* ili *reduce*), a nema na osnovu čega, on primenjuje pravilo koje kaže da u takvoj situaciji bira *shift* akciju. Korisniku to može ali ne mora da odgovara.

Kod IF-ELSE konflikta, ispravna akcija je *shift*, jer semantika `if` iskaza kaže da se `else` deo pridružuje najbližem prethodnom `if` iskazu bez `else` dela. Jedino iz ovog razloga možemo ignorisati poruku o ovom konfliktu prilikom kompajliranja `.y` datoteke.

4.2.3 Oporavak od greške na primeru miniC parsera

Kada `bison`-ov parser detektuje grešku, ispiše string **“`syntax error`”** i završi parsiranje (završi program). Ukoliko korisnik želi da parser detektuje više od jedne greške, mora implementirati detekciju i oporavak greške. `Bison` nudi dve vrste opravka od greške. Obe će biti pokazane na primeru sa listinga 4.2.

Prvi primer oporavka koristi `error` token (listing 4.3) i biće primenjen na `if` pravilo. Ovaj token je postavljen između zagrada, tako da će parser moći da obavi redukciju po ovom pravilu, ukoliko dođe do sintaksne greške unutar uslova (tj. unutar relacionog izraza). Nakon toga može da izvrši pridruženu akciju u kojoj se ispisuje adekvatna poruka o lokaciji greške. Akcija sadrži i poziv makroa `yyerrok` koji signalizira parseru da je oporavak završen i da može da nastavi parsiranje.

Listing 4.3: Oporavak od greške pomoću `error` tokena

```
if_part
  : _IF _LPAREN rel_exp _RPAREN statement
  | _IF _LPAREN error _RPAREN
      { yyerror("Error␣in␣if␣condition\n"); yyerrok; }
     statement
  ;
```

Drugi primer oporavka je primenjen na `return` pravilo i ima oblik pravila (*production rule*) (listing 4.4). Pravilu za `return` iskaz je dodato još jedno pravilo, u kom nedostaje karakter tačka-zarez na kraju iskaza. Ako se u ulaznom miniC programu pojavi `return` iskaz bez tačke-zarez na kraju, parser će obaviti redukciju po ovom pravilu i izvršiti pridruženu akciju u kojoj ispisuje poruku sa preciznim opisom greške. Ako se u ulaznom kodu pojavi kompletan `return` iskaz (koji sadrži karakter tačka-zarez na kraju iskaza) parser će proći kroz pravilo koje opisuje ispravan `return` iskaz. U ovoj varijanti oporavka od greške nema potrebe pozivati makro `yyerrok`, jer parser nije u vanrednom stanju zbog greške - on ovakvu situaciju ne vidi kao grešku, već kao najobičnije pravilo koje treba redukovati.

Listing 4.4: Oporavak od greške pomoću pravila

```
return_statement
  : _RETURN num_exp _SEMICOLON
  | _RETURN num_exp
      { yyerror("Missing␣\';\'␣in␣return␣statement\n");}
  ;
```

U praksi obe vrste oporavka treba koristiti umereno, jer uvode nova pravila, čime se značajno povećava obim i kompleksnost parsera.

4.2.3.1 Primeri oporavka od greške

Ako malo izmenimo `abs` program (listing 4.5), tako da napravimo dve sintaksne greške - u dva iskaza izostavimo separator “;”:

Listing 4.5: `abs2.mc`

```
int abs(int i) {
  if(i < 0)
    return 0 - i;
  else
    return i    //error
}

int main() {
  int x;
  x = -5        //error
  return abs(x);
}
```

parser će se, u ova dva slučaja, ipak drugačije ponašati. U prvom slučaju, parser ima pripremljen oporavak od greške (još jednim pravilom bez separatora), pa će ispisati grešku i nastaviti parsiranje. U drugom slučaju, parser nema pripremljen oporavak od situacije kada nedostaje separator na kraju iskaza dodele, pa mora da prijavi grešku i završi parsiranje.

Da su ova dva slučaja greške bila u obrnutom redosledu, parser bi prijavio samo jednu grešku (jer bi morao da završi parsiranje).

4.3 Vežbe

1. Proširiti miniC jezik i miniC parser `while` iskazom. Primer:

```
while(a > 0)
  a = a - 1;
```

2. Proširiti miniC jezik i miniC parser `break` iskazom. Primer:

```
break;
```

3. Proširiti miniC jezik i miniC parser `for` iskazom. Primer:

```
for(a = 0; a < x; a++)
  y = y + a * 2;
```

4. Proširiti miniC jezik i miniC parser `switch` iskazom. Primer:

```
switch (state) {
  case 10: { state = 1; } break;
  case 20: state = 2;
  default: state = 0;
}
```

5. Proširiti miniC jezik i miniC parser nizovima. Primer:

```
int n[3];
n[0] = 0;
n[1] = n[0] + 13;
```

6. Proširiti miniC jezik i miniC parser globalnim promenljivim. Primer:

```
int a;
int f() { }
```

7. Proširiti deklaraciju promenljive njenom inicijalizacijom, tako da parser prihvata i `int a = 0;`
8. Izmeniti miniC parser tako da u telu funkcije bude moguće naizmenično pisati deklaracije promenljivih i iskaze, na primer:

```
int f() {
  int a;
  a = 0;
  int b;
  b = 0;
}
```

9. Proširiti relacione izraze operatorima `<=`, `>`, `>=`, `!=`.
10. Proširiti miniC jezik logičkim izrazima `&&` i `||` i proširiti miniC parser.
11. Proširiti miniC jezik i miniC parser definicijama funkcija sa više parametara, kao i pozivima funkcija sa više argumenata.

Glava 5

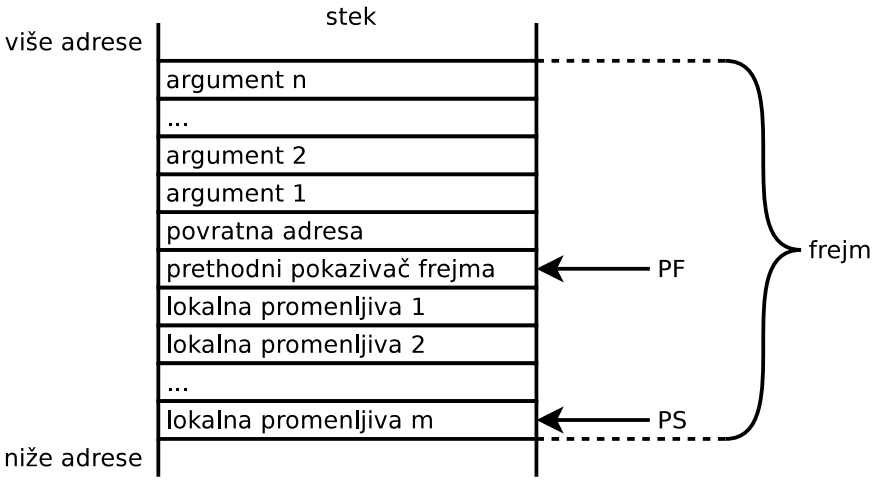
Semantička analiza

Semantička analiza je faza kompajliranja u kojoj se proverava značenje (semantika) programskog teksta (koda). Kod koji je sintaksno ispravan, ne mora biti i semantički ispravan. Na primer, ako se promenljiva tipa `int` poredi sa promenljivom tipa `boolean`, kompajler treba da prijavi semantičku grešku.

Semantička analiza se, najčešće, implementira u parseru, tako što se pravilima dodaju semantičke provere. Da bi se ove provere implementirale, potrebno je znanje o tome kako izgledaju organizacija memorije i arhitektura računara za koju se generiše kod.

5.1 Organizacija memorije za miniC

Globalni identifikatori su statični – postoje za sve vreme izvršavanja programa i za njih se mogu rezervisati memorijske lokacije u toku kompajliranja. Lokalni identifikatori su dinamični – postoje samo za vreme izvršavanja funkcija i za njih se zauzimaju memorijske lokacije na početku izvršavanja funkcija. Ove lokacije se oslobađaju na kraju izvšavanja funkcija, pa se zato lokalnim identifikatorima dodeljuju memorijske lokacije sa steka. Deo steka koji se zauzima za izvršavanje neke funkcije se zove (stek) frejm. Tipični frejm izgleda kao na slici 5.1.



Slika 5.1: Stek frejm

Stek frejm, redom, sadrži:

- argumente poziva funkcije (od poslednjeg ka prvom),
- povratnu adresu (od koje će se nastaviti izvršavanje koda nakon poziva funkcije),
- pokazivač prethodnog stek frejma (frejma funkcije iz koje se poziva ova funkcija) i
- lokalne promenljive (od prve ka poslednjoj).

Pokazivač frejma PF pokazuje na početak frejma, a pokazivač PS pokazuje na vrh steka. PF ostaje isti, dok se PS menja tokom izvršavanja funkcije.

Pošto će generator koda generisati kod na hipotetskom asemblerskom jeziku, potrebno je upoznati se sa arhitekturom naredbi i registara (vidi 6.1). Ukupno ima 16 registara. Oznaka registra se sastoji od oznake `%` i rednog broja registra: `%0`, `%1`, ..., `%15`. Registri od `%0` do `%12` imaju opštu namenu i služe kao radni registri. Po konvenciji, registar `%13` je rezervisan za povratnu vrednost funkcije, registar `%14` služi kao pokazivač frejma (PF), a registar `%15` služi kao pokazivač steka (PS).

5.2 Tabela simbola

Tabela simbola (*symbol table*) je struktura podataka koja čuva identifikatore iz kompajliranog programa i sve informacije o tim identifikatorima. Za semantičku analizu, ona je neophodna, jer sadrži sve informacije na osnovu kojih se mogu sprovesti semantičke provere.

Na primer, u programu sa listinga 5.1, postoji samo `main()` funkcija, sa jednim parametrom i jednom lokalnom promenljivom:

Listing 5.1: `add.mc`

```
int main(int p) {
  int x;
  x = 100;
  return p + x;
}
```

Tokom kompajliranja ove funkcije, u tabelu simbola treba zapisati da postoji (slika 5.2):

- identifikator `main` koji predstavlja funkciju, čiji povratni tip je `int` i koja ima jedan parametar tipa `int`

- identifikator `p` tipa `int`, koji predstavlja parametar, prvi po redu

- identifikator `x` tipa `int`, koji predstavlja lokalnu promenljivu, prvu po redu

- literal 100 tipa `int`.

Za program sa listinga 5.1, sadržaj tabele simbola (na kraju parsiranja funkcije `main()`) bi bio:

STRING SIMBOLA	VRSTA SIMBOLA	TIP SIMBOLA	ATRIBUT SIMBOLA	TIP PARAMETRA FUNKCIJE
main	FUN	INT_TYPE	1	INT_TYPE
p	PAR	INT_TYPE	1	-
x	VAR	INT_TYPE	1	-
100	LIT	INT_TYPE	-	-

Slika 5.2: Primer tabele simbola

I globalni i lokalni identifikatori mogu biti smešteni u istu tabelu simbola. Globalni su prisutni u tabeli simbola sve vreme kompajliranja, dok su lokalni prisutni samo u toku kompajliranja funkcije kojoj pripadaju. Izlaskom iz opsega vidljivosti, iz tabele simbola se brišu identifikatori vidljivi (samo) u tom opsegu. Na primer, na kraju parsiranja funkcije, brišu se svi lokalni identifikatori funkcije.

Pošto u tabeli simbola istovremeno mogu postojati identični globalni i lokalni identifikatori, radi njihovog razlikovanja, u tabeli mora biti naznačena vrsta identifikatora (da li je funkcija, promenljiva, parametar, argument, ...). Za svaki identifikator mora postojati i oznaka njegovog tipa (da li je `int`, `unsigned`, `char`, ...). Za parametre i lokalne promenljive mora postojati njihova relativna pozicija na steku u odnosu na početak frejma, tj. njihov redni broj (ova informacija je potrebna za generisanje koda).

Takođe, radi uniformnosti, u tabeli simbola mogu biti smeštene i oznake radnih registara. One se smeštaju u tabelu u vreme njene inicijalizacije. Registar `%0` se smešta u element sa indeksom `0`, .., a `%12` u element sa indeksom `12`. Registri su prisutni u tabeli simbola sve vreme kompajliranja. Registri se koriste u fazi generisanja koda, kada se u njih smeštaju međurezultati izraza.

5.2.1 Implementacija tabele simbola

Za potrebe kursa koristi se vrlo jednostavna implementacija tabele simbola: tabela je organizovana kao niz struktura koje opisuju osobine identifikatora (slika 5.2).

Datoteka `defs.h` (listing 5.2) koja je uvedena u poglavlju sintaksne analize (listing 4.1) je proširena enumeracijom `kinds` koja sadrži konstante koje opisuju vrste simbola. Vrednosti konstanti su stepeni broja 2 (1, 2, 4, 8, ... tj. sadrže samo po 1 setovan bit) da bi se nad njima mogla primeniti *bitwise* logika. Ovakav zapis klase simbola olakšava pretragu tabele simbola.

Listing 5.2: `defs.h` - enumeracija `kinds`

```
// vrste simbola (moze ih biti maksimalno 32)
enum kinds { NO_KIND = 0x1, REG = 0x2, LIT = 0x4,
             FUN = 0x8, VAR = 0x10, PAR = 0x20 };
```

U ovu datoteku smeštena su i dva makroa koja olakšavaju prijavu greške ili upozorenja: `err(args ...)` i `warn(args ...)`, kao i nekoliko konstanti konstante za simulaciju bool tipa (`TRUE`, `FALSE`), dužina tabele simbola, tj. broj elemenata tabele: `SYMBOL_TABLE_LENGTH`, dužina bafera za smeštanje poruka o greškama: `CHAR_BUFFER_LENGTH`, oznaka da simbol nema atribut `NO_ATTR`, redni broj poslednjeg radnog registra `LAST_WORKING_REG`, redni broj registra koji čuva povratnu vrednost funkcije `FUN_REG`.

Listing 5.3: `defs.h` - dodate konstante

```
#define bool int
#define TRUE  1
#define FALSE 0

#define SYMBOL_TABLE_LENGTH   64
#define CHAR_BUFFER_LENGTH   128
#define NO_ATTR               -1
#define LAST_WORKING_REG      12
#define FUN_REG               13
```

5.2.1.1 Struktura jednog elementa tabele simbola

Struktura jednog elementa tabele simbola se sastoji od polja (listing 5.4):

`name` string identifikatora
`kind` vrsta simbola - sadrži vrednost iz enumeracije `kinds` (listing 5.2)
`type` tip simbola - sadrži vrednost iz enumeracije `types` (listing 5.2)
`attr` atribut - sadrži različite vrednosti za različite vrste simbola
- za lokalnu promenljivu, u ovom polju stoji redni broj promenljive
- za parametar, u ovom polju stoji redni broj parametra
- za identifikator funkcije, u ovom polju se čuva broj parametara[1]
- za ostale identifikatore, ovo polje nije popunjeno.

`ptyp` tip parametra - sadrži vrednost iz enumeracije `types` (listing 5.2)

Listing 5.4: `symtab.h` - struktura elementa tabele simbola

```
typedef struct sym_entry {
   char *   name;          // ime simbola
   unsigned kind;          // vrsta simbola
   unsigned type;          // tip vrednosti simbola
   int      attr;          // dodatni attribut simbola
   unsigned ptyp;          // tip parametra funkcije
```

5.2.1.2 Operacije za rad sa tabelom simbola

Operacije za rad sa tabelom simbola se nalaze u datotekama `symtab.h` i `symtab.c`.

Funkcije za ubacivanje u tabelu simbola

Funkcija koja služi za ubacivanje simbola u tabelu simbola se zove `insert_symbol()`. Parametri funkcije su string, vrsta, tip i atribut simbola. Funkcija vraća indeks ubačenog elementa u tabeli simbola. Funkcija radi tako što prvo proveri da li se simbol sa takvim imenom već nalazi u tabeli simbola, pa ako se ne nalazi ubacuje ga, a u suprotnom prijavljuje grešku o dupliranosti simbola.

```
int insert_symbol(char *name, unsigned kind,
                  unsigned type, int attr);
```

Funkcija koja služi za ubacivanje novog literala u tabelu simbola se zove `insert_literal()`. Parametri funkcije su string i tip literala. Funkcija će ubaciti novi literal ako on već ne postoji u tabeli simbola. U suprotnom neće prijavljivati grešku, jer je dovoljna jedna instanca literala u tabeli (svi literali se tretiraju kao lokalni za funkciju).

```
int insert_literal(char *str, unsigned type);
```

Funkcije za pretraživanje tabele simbola

Funkcija koje pretražuje tabelu simbola se zove `lookup_symbol()`. Parametri funkcije su ime i vrsta simbola koji se traži. Funkcija vraća indeks elementa tabele simbola na kom je pronašla simbol, ili `-1` ukoliko ga nije našla. Pretraga tabele simbola je potrebna u semantičkim proverama, kada se npr. proverava da li je promenljiva u nekom izrazu prethodno deklarisana ili ne.

[1] miniC funkcija podržava maksimalno jedan parametar

```
int lookup_symbol(char *name, unsigned kind);
```

Funkcija koje pretražuje tabelu simbola i traži literal u njoj se zove `lookup_literal()`. Parametri funkcije su ime i tip literala koji se traži. Funkcija vraća indeks elementa tabele simbola na kom je pronašla literal, ili `-1` ukoliko ga nije našla.

```
int lookup_literal(char *name, unsigned type);
```

Ažuriranje elementa tabele simbola

Set i *get* metode kojima se menjaju i čitaju vrednosti određenih polja elementa tabele simbola su:

```
char*    get_name(int index);
unsigned get_kind(int index);
unsigned get_type(int index);
void     set_attr(int index, int attr);
unsigned get_attr(int index);
void     set_ptyp(int index, unsigned type);
unsigned get_ptyp(int index);
```

Funkcije za brisanje simbola iz tabele simbola

Funkcija koja briše deo tabele simbola i koja se koristi za brisanje lokalnih simbola funkcije se zove `clear_symbols()`. Parametar ove funkcije je indeks prvog elementa koji treba da se obriše. Funkcija će obrisati sve simbole koji se nalaze između ovog indeksa i poslednjeg popunjenog elementa tabele simbola.

```
void clear_symbols(unsigned begin_index);
```

Funkcije za rad sa tabelom simbola u celini

Funkcije za rad sa tabelom simbola u celini su funkcija za prikaz svih popunjenih elemenata tabele: `print_symtab()`, funkcija za inicijalizaciju tabele simbola: `init_symtab()` i funkcija za brisanje svih elemenata tabele simbola: `clear_symtab()`.

```
void print_symtab(void);
void init_symtab(void);
void clear_symtab(void);
```

5.3 miniC parser sa semantičkim proverama

Semantičke provere se mogu implementirati kao zasebne funkcije ili kao deo akcije u nekom pravilu u parseru.

```
bool check_types(int first_index, int second_index);
```

Funkcija `check_types()` vrši proveru tipova. Dva parametra su indeksi dva elementa u tabeli simbola. Funkcija pristupa elementima na datim indeksima i poredi njihove tipove. Vraća vrednost `TRUE` ako su isti ili `FALSE` ako su različiti.

miniC parser sa semaničkim proverama koristi prethodno opisanu tabelu simbola (listinzi 5.2, 5.4) a impelementiran je u datoteci `semantic.y`.

Prvi deo specifikacije parsera (dat na listingu 4.2), je proširen definicijama promenljivih koje koristi parser u toku provere semantike (listing 5.5). To su broj grašaka `error_count`, broj upozorenja `warning_count`, broj lokalnih promenljivih `var_num`, indeks u tabeli simbola na kom se nalazi trenutno parsirana funkcija `fun_idx` i indeks u tabeli simbola na kom se nalazi trenutno parsirani poziv funkcije `fcall_idx`:

Listing 5.5: `semantic.y` - promenljive

```
  int error_count = 0;
  int warning_count = 0;
  int var_num = 0;
  int fun_idx = -1;
  int fcall_idx = -1;
```

U nastavku su definisani i tipovi pojmova: iza ključne reči `%type` se navodi tip iz unije (`<i>`), a zatim imena svih pojmova koji će biti tog tipa (listing 5.6). Ako je tip pojma `<i>`, to znači da će njegova vrednost biti celobrojnog tipa.

Listing 5.6: `semantic.y` - tipovi pojmova

```
%type <i> type num_exp exp literal parameter
%type <i> function_call argument rel_exp
```

Dva tokena: `ONLY_IF` i `_ELSE` su definisana upotrebom deklaracije %nonassoc koja kaže da ovi tokeni nemaju asocijativnost. Ova dva tokena se koriste za potrebe razrešavanja dvosmislenosti *if-else* iskaza.

Listing 5.7: `semantic.y` - tokeni za razrešavanje dvosmislenosti

```
%nonassoc ONLY_IF
%nonassoc _ELSE
```

Pravila u drugom delu specifikacije parsera su proširena akcijama koje sadrže semantičke provere.

Semantička provera, koju treba obaviti nakon prepoznavanja celog programa, je da li postoji `main` funkcija i da li je njen tip `int`. Provera se vrši pretragom tabele simbola. Ako se ne nađe simbol `main` koji predstavlja funkciju, prijavi se semantička greška. Drugi deo provere se radi samo ako simbol `main` postoji u tabeli: proveri se njegov tip i prijavi semantička greška ako nije `int` (listing 5.8).

Listing 5.8: `semantic.y` - `program`

```
program
  : function_list
      {
        int idx;
        if((idx = lookup_symbol("main", FUN)) == -1)
          err("undefined␣reference␣to␣'main'");
        else
          if(get_type(idx) != INT)
            warn("return␣type␣of␣'main'␣is␣not␣int");
       }
  ;
```

Pojam `type`, dobija vrednost koja stiže uz token `_TYPE`, a kojoj se pristupa preko meta promenljive $1 [3]. To je ili vrednost konstante `INT`, ili vrednost konstante `UINT` (vidi skener na listingu 3.2, pravila za "`int`" i "`unsigned`") u zavisnosti od prepoznatog tipa.

Listing 5.9: `semantic.y` - tip

```
type
  : _TYPE
      {  $$ = $1; }
  ;
```

Tokom parsiranja definicije funkcije vrši se nekoliko semantičkih akcija (listing 5.10). Čim pristigne ime funkcije, ono se smešta u tabelu simbola zajedno sa informacijama o tipu povratne vrednosti funkcije (vrednost pojma `type`, tj. vrednost meta-promenljive `$1`, koja nosi konstantu `INT` ili `UINT`) i vrsti identifikatora (konstanta `FUN`). Promenljiva `fun_idx` dobija indeks elementa u tabeli simbola u koji je smešteno ime funkcije.

Nakon parisranja definicije parametra u element tabele simbola sa indeksom `fun_idx` koji čuva informacije o funkciji, u polje `attr`, smešta se podatak o broju parametara funkcije. Ovaj podatak se čita kao vrednost meta-promenljive `$5`, tj. kao vrednost pojma `parameter`. Promenljiva `var_num`, koja broji lokalne promenljive jedne funkcije, se resetuje, jer sada sledi ulazak u telo (nove) funkcije.

Nakon parsiranja celog tela funkcije (nakon prepoznavanja pojma `body`) iz tabele simbola se brišu svi simboli koji su lokalni za funkciju. Ovo sme da se radi (i potrebno je), jer je opseg vidljivosti lokalnih simbola samo do kraja funkcije u kojoj su deklarisani. Ovim je omogućeno da svaka funkcija ima svoje lokalne promenljive, koje mogu biti (potpuno) iste kao u nekoj drugoj funkciji. Funkcija `clear_symbols()`, koja briše lokalne simbole, se poziva sa argumentom `fun_idx+1`, jer brisanje tabele simbola treba početi nakon imena funkcije. Ime funkcije je globalni identifikator i zato treba da bude prisutan u tabeli simbola tokom parsiranja celog programa.

Listing 5.10: `semantic.y` - definicija funkcije

```
function
  : type _ID
      {
        fun_idx = insert_symbol($2, FUN, $1, NO_ATTR);
      }
    _LPAREN parameter _RPAREN
      {
        set_attr(fun_idx, $5);
        var_num = 0;
      }
    body
      {
        // izbaci iz tabele simbola sve lokalne simbole
        if(get_last_element() > fun_idx)
          clear_symbols(fun_idx + 1);
      }
  ;
```

Nakon parsiranja pojma `parameter`, ime parametra (vrednost `$2`) se ubacuje u tabelu simbola kao `PAR` sa tipom koji nosi pojam `type` (vrednost `$1`) i sa rednim brojem `1` (polje `attr` parametra čuva redni broj parametra) (listing 5.11). Tip parametra (vrednost `$1`) se zapisuje u polju `ptyp` simbola funkcije u tabeli simbola (jer polje `ptyp` za funkciju čuva tip njenog parametra). Vrednost pojma parameter se postavlja ili na `0` (ako nije bilo parametara) ili na `1` ukoliko je detektovan (jedan) parametar.

Listing 5.11: `semantic.y` - parametar

```
parameter
```

```
  : /* empty */
      { $$ = 0; }

  | type _ID
      {
        insert_symbol($2, PAR, $1, 1);
        set_ptyp(fun_idx, $1);
        $$ = 1; // samo 1 parametar
      }
  ;
```

Nakon parsiranja pojma `variable`, odnosno deklaracije promenljive, ime promenljive (vrednost `$2`) se ubacuje u tabelu simbola kao `VAR` sa tipom koji nosi pojam `type` (vrednost `$1`). Vrednost brojača lokalnih promenljivih se inkrementira i zapisuje se kao atribut ovog simbola u tabeli simbola, zato što lokalna promenljiva na mestu atributa u tabeli simbola čuva svoj redni broj (listing 5.11).

Ukoliko je u tabeli simbola pronađen parametar sa istim imenom, prijavljuje se semantička greška.

Listing 5.12: `semantic.y` - promenljiva

```
variable
  : type _ID _SEMICOLON
      {
        if(lookup_symbol($2, PAR) != -1)
           err("redefinition␣of␣'%s'", $2);
        else
           insert_symbol($2, VAR, $1, ++var_num);
      }
  ;
```

Nakon parsiranja iskaza dodele, proverava se da li je identifikator koji se nalazi sa leve strane jednakosti promenljiva ili parametar (listing 5.13). Ako nije, prijavljuje se semantička greška, jer je na tom mestu dozvoljeno samo ime promenljive ili parametra (a ne, recimo, ime funkcije). Dodatno, vrši se i provera tipova leve i desne strane jednakosti, jer bi oni trebali biti isti. Funkciji `check_types` se prosleđuje indeks pronađenog identifikatora sa leve strane jednakosti i indeks u tabeli simbola gde se nalazi rezultat numeričkog izraza sa desne strane jednakosti (vrednost `$3`).

Listing 5.13: `semantic.y` - iskaz dodele

```
assignment_statement
  : _ID _ASSIGN num_exp _SEMICOLON
      {
        int idx = -1;
        if((idx = lookup_symbol($1, (VAR|PAR))) == -1)
          err("invalid␣lvalue␣in␣assignment");
        if(!check_types(idx, $3))
          err("incompatible␣types␣in␣assignment");
      }
  ;
```

Nakon parsiranja aritmetičkog izraza (pojam `num_exp`), vrši se provera tipova operanada, pa ako nisu isti, prijavljuje se semantička greška (listing 5.14). Provera se vrši pozivom funkcije `check_types()` sa argumentima: indeks prvog operanda u tabeli simbola (`$1`) i indeks drugog operanda u tabeli simbola (`$3`).

Listing 5.14: `semantic.y` - numerički izraz

```
num_exp
  : exp
  | num_exp _AROP exp
      {
        if(!check_types($1, $3))
          err("invalid operands to "
              "arithmetic operation");
      }
  ;
```

Nakon parsiranja identifikatora u izrazima (`exp`), vrši se provera postojanja dotičnog imena, tj. njegovog prisustva u tabeli simbola (listing 5.15). Ovu proveru obavlja funkcija `lookup_symbol()`. Ako ga nema u tabeli, znači da nikada nije deklarisan, pa treba prijaviti grešku, jer se ne može koristiti promenljiva koja nije prethodno deklarisana.

Ako se izraz nalazi u zagradama (poslednja varijanta pojma `exp`), njegova vrednost treba da se prenese kao vrednost celog izraza.

Listing 5.15: `semantic.y` - izraz

```
exp
  : literal
  | _ID
      {
        if(($$ = lookup_symbol($1, (VAR|PAR))) == -1)
          err("'%s' undeclared", $1);
      }
  | function_call
  | _LPAREN num_exp _RPAREN
      {  $$ = $2; }
  ;
```

Kada se isparsira literal, treba ga ubaciti u tabelu simbola kao lokalni simbol, a pojmu `literal` dodeliti vrednost: indeks u tabeli simbola na kom je smešten literal (listing 5.16).

Listing 5.16: `semantic.y` - literal

```
literal
  : _INT_NUMBER
      {  $$ = insert_literal($1, INT); }

  | _UINT_NUMBER
      {  $$ = insert_literal($1, UINT); }
  ;
```

Tokom parsiranja poziva funkcije (pojam `function_call`, listing 5.17) proverava se da li je navedeni identifikator ime neke od postojećih funkcija, pa ako nije, prijavljuje se greška da se poziva nepostojeća funkcija. Kako je usvojeno, povratna vrednost funkcije će biti smeštena u registru `%13`, pa je potrebno postaviti vrednost pojma `function_call` na indeks `13` u tabeli simbola (gde se nalazi registar `%13`). Registru `%13` se postavlja tip povratne vrednosti funkcije koja se kasnije koristi u izrazima.

Listing 5.17: `semantic.y` - poziv funkcije

```
function_call
  : _ID
      {
        if((fcall_idx = lookup_symbol($1, FUN)) == -1)
```

```
            err("'%s' is not a function", $1);
      }
    _LPAREN argument _RPAREN
      {
        if (get_attr(fcall_idx) != $4)
           err("wrong number of arguments to function '%s'",
              get_name(fcall_idx));
        //povratna vrednost funkcije se nalazi u %13
        set_reg_type(FUN_REG, get_type(fcall_idx));
        $$ = FUN_REG;
      }
  ;
```

Tokom parsiranja argumenta, treba proveriti da li je njegov tip isti kao tip odgovarajućeg parametra (listing 5.18). Ako nije, treba prijaviti grešku. Tip parametra se dobija pozivom funkcije `get_ptyp(fcall_idx)`, dok se tip argumenta dobija pozivom funkcije `get_type($1)` (`$1` je indeks elementa u tabeli simbola koji sadrži rezultat numeričkog izraza koji je prosleđen na mestu argumenta).

Vrednost pojma `argument` dobija vrednost `1` koja označava da je to prvi po redu (i jedini) argument. Ukoliko argumenta nema, vrednost ovog pojma postaje `0` (oznaka da nema argumenata).

Listing 5.18: `semantic.y` - argument

```
argument
  : /* empty */
    { $$ = 0; }

  | num_exp
    {
      if(get_ptyp(fcall_idx) != get_type($1))
        err("incompatible type for argument in '%s'",
          get_name(fcall_idx));
      $$ = 1;
    }
  ;
```

Tokom parsiranja relacionog izraza odnosno pojma `rel_exp` (vidi listing 5.19) potrebno je proveriti tipove operanada. Semantika jezika kaže da oni moraju biti isti. To se postiže pozivom funkcije `check_types()` sa argumentima poziva: indeks prvog operanda u tabeli simbola (`$1`) i indeks drugog operanda u tabeli simbola (`$3`).

Listing 5.19: `semantic.y` - relacioni izraz

```
rel_exp
  : num_exp _RELOP num_exp
      {
        if(!check_types($1, $3))
          err("invalid operands to relational "
            "operator");
      }
  ;
```

Nakon prepoznavanja `return` iskaza (listing 5.20), potrebno je proveriti tip izraza koji se vraća iz funkcije: semantika nalaže da ovaj tip mora biti isti kao povratni tip funkcije. Funkcija koja proverava tipove `check_types()` se poziva sa argumentima: indeks u tabeli simbola na kom se nalazi ime funkcije (`fun_idx`, na kom stoji informacija o povratnom tipu funkcije) i indeks izraza koji se vraća iza funkcije.

Listing 5.20: `semantic.y` - `return` iskaz

```
return_statement
  : _RETURN num_exp _SEMICOLON
      {
        if(!check_types(fun_idx, $2))
          err("incompatible␣types␣in␣return");
      }
  ;
```

Funkcije `yyerror()` i `warning()` služe za prijavljivanje greške ili upozorenja korisniku.

Listing 5.21: `semantic.y` - korisničke funkcije parsera

```
int yyerror(char *s) {
  fprintf(stderr, "\nline␣%d:␣ERROR:␣%s", yylineno, s);
  error_count++;
  return 0;
}

void warning(char *s) {
  fprintf(stderr,"\nline␣%d:␣WARNING:␣%s", yylineno, s);
  warning_count++;
}
```

Funkcija `main` vrši inicijalizaciju tabele simbola (`init_symtab()`) pre početka parsiranja i brisanje tabele simbola (`clear_symtab()`) nakon parsiranja. Na kraju, prijavljuje broj grešaka i upozorenja u toku parsiranja.

Listing 5.22: `semantic.y` - `main` funkcija

```
int main() {
  int synerr;
  init_symtab();

  synerr = yyparse();

  clear_symtab();

  if(warning_count)
    printf("\n%d␣warning(s).\n", warning_count);

  if(error_count)
    printf("\n%d␣error(s).\n", error_count);

  if (synerr)
    return -1;
  else
    return error_count;
}
```

5.3.1 Primer upotrebe parsera sa semantičkim proverama

Ako parseru sa semantičkim proverama prosledimo datoteku `abs.mc` (listing 2.1), parser će ispisati da nema (semantičkih) grešaka, jer ova datoteka sadrži sintaksno i semantički ispravan miniC program:

```
$./semantic <abs.mc
0 errors
$
```

Međutim, ako izmenimo argument u pozivu `abs()` funkcije, tako što umesto označenog literala `-5` navedemo neoznačen literal `2u`:

```
return abs(2u);
```

kompajler će prijaviti semantičku grešku, jer tip prvog argumenta nije isti kao tip prvog parametra `abs()` funkcije:

```
$./semantic <abs.mc
line 11: ERROR: incompatible type for argument in 'abs'
1 errors
$
```

Ako izmenimo ime `main()` funkcije u `main2()`, kompajler će prijaviti semantičku grešku, jer ne postoji obavezna `main()` funkcija:

```
$./semantic <abs.mc
line 13: ERROR: undefined reference to 'main'
1 errors
$
```

Ukoliko izmenimo tip lokalne promenljive `res` iz `int` u `unsigned`, kompajler će prijaviti semantičke greške na svim mestima gde se ta promenljiva koristi, jer će na tim mestima provera tipova biti neuspešna:

```
$./semantic <abs.mc
line 4: ERROR: incompatible types in assignment
line 6: ERROR: incompatible types in assignment
line 7: ERROR: incompatible types in return
3 errors
$
```

5.4 Vežbe

Kako bi trebali da se prošire parser i tabela simbola da bi se implementirale sledeće nove osobine miniC jezika:

1. globalne promenljive
2. naizmenična pojava deklaracije promenljive i iskaza, u telu funkcije
3. definicija funkcije sa više parametara i poziv funkcije sa više argumenata
4. definicija promenljive sa inicijalizacijom, na primer: `int a = 5;`
5. `while` iskaz
6. `break` iskaz
7. `for` iskaz
8. `switch` iskaz
9. nizovi
10. novi relacioni operatori `<=`, `>`, `>=`, `!=`
11. povezati relacione izraze logičkim operatorima `&&` i `||`

Glava 6

Generisanje koda

Generisanje koda je faza kompajliranja u kojoj se proizvodi datoteka sa ekvivalentnim programom napisanim na ciljnom programskom jeziku. Ciljni jezik je jezik niskog nivoa, i može biti neki mašinski ili asemblerski jezik. Ciljni jezik na koji će (u primeru) biti preveden miniC programski jezik, je hipotetski asemblerski jezik. Ovaj asemblerski jezik je vrlo jednostavan i zgodan za predstavljanje faze generisanja koda.

6.1 Hipotetski asemblerski jezik

Podrazumeva se da registri i memorijske lokacije zauzimaju po 4 bajta. Ukupno ima 16 registara. Oznaka registra se sastoji od oznake `%` i rednog broja registra: `%0`, `%1`, ..., `%15`. Registri od `%0` do `%12` imaju opštu namenu i služe kao radni registri. Registar `%13` rezervisan je za povratnu vrednost funkcije. Registar `%14` služi kao pokazivač frejma. Registar `%15` služi kao pokazivač steka.

Labele započinju malim slovom iza koga mogu da slede mala slova, cifre i podcrta '_' (alfabet je 7 bitni ASCII). Iza labele se navodi dvotačka, a ispred sistemskih labela se navodi znak '`@`'.

6.1.1 Operandi

neposredni operand odgovara celom (označenom ili neoznačenom) broju: `$0` ili `$-152` a njegova vrednost vrednosti tog broja, dok `$lab` odgovara adresi labele `lab`.

registarski operand odgovara oznaci registra, a njegova vrednost sadržaju tog registra, npr. `%0`.

direktni operand odgovara labeli, npr. `a`. Njegova vrednost odgovara adresi labele, ako ona označava naredbu i koristi se kao operand naredbe skoka ili poziva potprograma. Ako direktni operand odgovara labeli koja označava direktivu i ne koristi se kao operand naredbe skoka ili poziva potprograma, njegova vrednost odgovara sadržaju adresirane lokacije.

indirektni operand odgovara oznaci registra navedenoj između malih zagrada: `(%0)`, a njegova vrednost sadržaju memorijske lokacije koju adresira sadržaj registra.

indeksni operand započinje celim (označenim ili neoznačenim) brojem ili labelom iza čega sledi oznaka registra navedena između malih zagrada: `-8(%14)` ili `4(%14)` ili `tabela(%0)`. Njegova vrednost odgovara sadržaju memorijske lokacije koju adresira zbir vrednosti broja i sadržaja registra, odnosno zbir adrese labele i sadržaja registra.

Operandi se dele na:

ulazne neposredni, registarski, direktni, indirektni i indeksni i
izlazne registarski, direktni, indirektni i indeksni.

6.1.2 Naredbe

Neke naredbe postoje u 3 varijante za 3 različita tipa podataka, koje su označene sa `x`. `x` može biti `S` (*signed*) za označene tipove, `U` (*unsigned*) za neoznačene, i `F` (*float*) za realne (mašinska normalizovana forma).

Naredba poređenja brojeva postavlja bite status registra u skladu sa razlikom prvog i drugog ulaznog operanda

```
CMPx ulazni operand, ulazni operand
```

Naredba bezuslovnog skoka smešta u programski brojač vrednost ulaznog operanda (omogućujući tako nastavak izvršavanja od ciljne naredbe koju adresira ova vrednost)

```
JMP ulazni operand
```

Naredbe uslovnog skoka smeštaju u programski brojač vrednost ulaznog operanda samo ako je ispunjen uslov određen kodom naredbe (ispunjenost uslova zavisi od bita status registra)

```
JEQ ulazni operand
JNE ulazni operand
JGTx ulazni operand
JLTx ulazni operand
JGEx ulazni operand
JLEx ulazni operand
```

Naredbe rukovanja stekom omogućuju smeštanje na vrh steka vrednosti ulaznog operanda, odnosno preuzimanje vrednosti sa vrha steka i njeno smeštanje u izlazni operand (podrazumeva se da `%15` služi kao pokazivač steka, da se stek puni od viših lokacija ka nižim i da `%15` pokazuje vrh steka)

```
PUSH ulazni operand
POP izlazni operand
```

Naredba poziva funkcije smešta na vrh steka zatečeni sadržaj programskog brojača, a u programski brojač smešta vrednost ulaznog operanda:

```
CALL ulazni operand
```

Naredba povratka iz potprograma preuzima vrednost sa vrha steka i smešta je u programski brojač

```
RET
```

Aritmetičke naredbe omogućuju sabiranje, oduzimanje i množenje ulaznih operanada (uz izazivanje izuzetka ako rezultat ne može da stane u izlazni operand) kao i deljenje prvog ulaznog operanda drugim i smeštanje količnika u izlazni operand

```
ADDx ulazni operand, ulazni operand, izlazni operand
SUBx ulazni operand, ulazni operand, izlazni operand
MULx ulazni operand, ulazni operand, izlazni operand
DIVx ulazni operand, ulazni operand, izlazni operand
```

Naredba za prebacivanje vrednosti kopira vrednost ulaznog operanda u lokaciju izlaznog operanda

```
MOV ulazni operand, izlazni operand
```

Naredba konverzije celog broja u razlomljeni broj omogućuje da se vrednost ulaznog operanda, koja je celi broj, konvertuje u vrednost izlaznog operanda, koja je ekvivalentni razlomljeni broj u mašinskoj normalizovanoj formi

```
TOF ulazni operand, izlazni operand
```

Naredba konverzije razlomljenog broja u celi broj omogućuje da se vrednost ulaznog operanda, koja je razlomljeni broj u mašinskoj normalizovanoj formi, konvertuje u vrednost izlaznog operanda, koja je ekvivalentni celi broj, ako je konverzija moguća, inače se izaziva izuzetak

```
TOI ulazni operand, izlazni operand
```

6.1.3 Direktive

direktiva zauzimanja memorijskih lokacija omogućuje zauzimanje onoliko uzastopnih memorijskih lokacija koliko je navedeno u operandu

```
WORD broj
```

6.2 Primeri ekvivalencije

Pre nego što se krene u implementaciju generisanja koda, potrebno je imati šeme ekvivalencije. To su šabloni koji opisuju kako se određeni iskazi miniC jezika pišu na hipotetskom asemblerskom jeziku. Ovi šabloni ne smeju izmeniti značenje polaznog miniC iskaza.

Primeri (mogućih) ekvivalencija između miniC koda i hipoteskog asemblerskog koda, koji slede na listinzima, napisani su u dve kolone. U prvoj koloni se nalazi miniC kod koji se prevodi, a u drugoj se nalazi ekvivalentni asemblerski kod. Primeri sadrže i predloge načina generisanja ekvivalentnog koda. U svim navedenim primerima se podrazumeva da su `a`, `b` i `c` celobrojne označene lokalne promenljive, deklarisane tim redom.

Na osnovu slike 5.1 na kojoj je prikazana organizacija stek frejma, vidi se da se lokalne promenljive smeštaju ispod pokazivača frejma `%14`. Za pristup ovim lokacijama koristi se indeksno adresiranje pomoću registra `%14`. Tako je lokacija prve promenljive: `-4(%14)` a druge `-8(%14)` (memorijsku lokaciju adresira zbir vrednosti broja i sadržaja registra). Lokacije argumenata počinju na drugoj lokaciji iznad pokazivača frejma (neposredno iznad pokazivača frejma se nalazi povratna adresa), pa se prvom argumentu pristupa na adresi `8(%14)`, drugom na adresi `12(%14)`, itd.

6.2.1 Iskaz pridruživanja

Primeru iskaza pridruživanja odgovara izgenerisani kod:

Listing 6.1: primer ekvivalencije za jednostavan iskaz pridruživanja

```
a = b - a;                          SUBS  -8(%14),-4(%14),%0
                                    MOV   %0,-4(%14)
```

Za smeštanje međurezultata izraza koriste se radni registri. Podrazumeva se da su svi radni registri na početku slobodni. U prvoj naredbi `SUBS` se zauzima (prvi slobodan) radni registar `%0` i u njega se smešta razlika `b - a` (prvi operand naredbe `SUBS` je lokacija lokalne promenljive `b`, drugi operand lokacija lokalne promenljive `a`, a izlazni operand je registar `%0`). U naredbi `MOV` se vrednost iz radnog registra `%0` kopira u lokaciju lokalne promenljive `a`, i oslobađa se registar `%0`.

Listing 6.2: primer ekvivalencije za iskaz pridruživanja

```
a = (a - b) + (a - c) - c;          SUBS   -4(%14),-8(%14),%0
                                    SUBS   -4(%14),-12(%14),%1
                                    ADDS   %0,%1,%0
                                    SUBS   %0,-12(%14),%0
                                    MOV    %0,-4(%14)
```

U prvoj naredbi `SUBS` se zauzima radni registar `%0`, a u drugoj registar `%1`. U naredbi `ADDS` (linija 3) se oslobađaju radni registri `%0` i `%1`, a ponovo se zauzima radni registar `%0`. U naredbi `SUBS` (linija 4) se oslobađa i ponovo zauzima radni registar `%0`, a u naredbi `MOV` se oslobađa radni registar `%0`.

Radni registar se zauzima za smeštanje rezultata svakog aritmetičkog izraza (`num_exp`). Radni registar se oslobađa čim se preuzme njegova vrednost.

Pošto se međurezultati izraza koriste u suprotnom redosledu od onog u kome su izračunati, radni registri, koji se koriste za smeštanje međurezultata, se zauzimaju i oslobađaju po principu steka. Kao "pokazivač steka registara" koristi se promenljiva `free_reg_num`, koja sadrži broj prvog slobodnog radnog registra. Zauzimanje radnog registra se sastoji od preuzimanja vrednosti promenljive `free_reg_num` i njenog inkrementiranja, a oslobađanje radnog registra se sastoji od dekrementiranja ove promenljive. Treba napomenuti da je broj registra istovremeno i indeks elementa tabele simbola.

Broj zauzetog radnog registra služi kao vrednost sintaksnog pojma (`num_exp`) koji odgovara izrazu čiji rezultat radni registar sadrži. Prekoračenje broja radnih registara (`free_reg_num > 12`) predstavlja fatalnu grešku u radu kompajlera.

6.2.2 Funkcija

6.2.2.1 Definicija funkcije

Za smeštanje povratne vrednosti funkcije (po dogovoru) se koristi registar `%13`. Funkcija, nakon poziva, lokalne promenljive i argumente čuva na stek frejmu (vidi sliku 5.1). Listing 6.3 sadrži primer ekvivalencije za definiciju funkcije.

Listing 6.3: primer ekvivalencije za definiciju funkcije

```
int f(int p) {                      f:
   int a;                              PUSH %14
```

```
   return p + a;                    MOV  %15,%14
}                                   SUBS %15,$4,%15
                                 @f_body:
                                    ADDS 8(%14),-4(%14),%0
                                    MOV  %0,%13
                                    JMP  @f_exit
                                 @f_exit:
                                    MOV  %14,%15
                                    POP  %14
                                    RET
```

Ekvivalentni asemblerski kod za funkciju započinje labelom koja ima isti identifikator kao funkcija, u ovom slučaju `f:`. Naredbom `PUSH` na stek se smešta "stari" pokazivač frejma, dok se pomoću naredbe `MOV` postavlja "novi" pokazivač frejma (registar `%14` sada pokazuje na vrh steka, kao i `%15`). Pomoću naredbe `SUBS` se zauzima prostor na steku za lokalnu promenljivu `a` (pokazivač steka se pomera jednu lokaciju, tj. 4 bajta niže). Promenljiva `var_num` služi kao brojač lokalnih promenljivih. Veličina prostora za lokalne promenljive se određuje kao `var_num * 4`. Prostor se zauzima samo ako funkcija ima lokalnih promenljivih (`var_num > 0`).

Ekvivalentni asemblerski kod tela funkcije započinje labelom koja sadrži ime funkcije sa postfiksom `_body`, u ovom slučaju `@f_body:`. U naredbi `ADDS` se računa povratna vrednost funkcije, a naredbom `MOV` se ta vrednost smešta u registar `%13` (jer je to dogovoreno mesto za povratnu vrednost funkcije). Ako funkcija ne sadrži `return` iskaz, kao povratna vrednost funkcije služi zatečeni sadržaj registra `%13`, koji je nepoznat u vreme definisanja funkcije.

Ekvivalentni asemblerski kod kraja funkcije započinje labelom koja sadrži ime funkcije sa postfiksom `_exit`, u ovom slučaju `@f_exit:`. Naredbom `MOV` se oslobađa prostor za lokalne promenljive (pokazivač steka se vraća u poziciju u kojoj je bio pre zauzimanja lokalnih promenljivih), a u naredbi `POP` se u pokazivač frejma vraća njegova prethodna vrednost ("stari" pokazivač frejma). Naredbom `RET` će doći do preusmeravanja toka izvršavanja na mesto odakle je funkcija pozvana.

6.2.2.2 Poziv funkcije

Primeru poziva funkcije (u okviru iskaza dodele) odgovara kod sa listinga 6.4.

Listing 6.4: primer ekvivalencije za poziv funkcije (a)

```
a = f(a + b);                    ADDS -4(%14),-8(%14),%0
                                 PUSH %0
                                 CALL f
                                 ADDS %15,$4,%15
                                 MOV  %13,-4(%14)
```

U prvoj liniji (u naredbi `ADDS`) se računa vrednost argumenta poziva funkcije i smešta u radni registar (pretpostavka je da su svi radni registri slobodni). Naredbom `PUSH` se vrednost argumenta smešta na stek. Naredbom `CALL` se poziva funkcija, a pomoću naredbe `ADDS` se oslobađa prostor koji je zauzimao argument (pokazivač steka se pomera 1 lokaciju naviše). Naredbom `MOV` se isporučuje povratna vrednost funkcije.

Ako se na mestu argumenta pojave literali ili promenljive, kao u slučaju poziva u primeru sa listinga 6.5, onda nisu potrebni radni registri, jer se vrednosti literala i promenljivih direktno mogu smeštati na stek.

Listing 6.5: primer ekvivalencije za poziv funkcije (b)

```
a = f(1);
```

```
PUSH $1
CALL f
ADDS %15,$4,%15
MOV  %13,-4(%14)
```

Izvorni miniC kompajler, MICKO, ne podržava pojavu poziva funkcije na mestu argumenta.

6.2.3 `if` iskaz

Primeru `if` iskaza odgovara kod sa listinga 6.6.

Listing 6.6: primer ekvivalencije za `if` iskaz

```
if(a < b)
  a = 1;
else
  a = 2;
```

```
@if0:
   CMPS -4(%14),-8(%14)
   JGES @false0
@true0:
   MOV  $1,-4(%14)
   JMP  @exit0
@false0:
   MOV  $2,-4(%14)
@exit0:
```

U ekvivalentnom asemblerskom kodu se pojavljuju labele `@if`, `@true`, `@false` i `@exit`. Neke od njih su potrebne, jer su ciljevi naredbi skokova (to su `@false` i `@exit`), dok su labele `@if` i `@true` nepotrebne, ali zgodne za praćenje i proveru izgenerisanog koda.

Labela `@if` označava početak `if` iskaza, a labela `@exit` njegov kraj. Izvršavanje `if` iskaza započinje proverom uslova (naredba `CMP`) i naredbom skoka (na `@true` ili `@false` labelu) u zavisnosti od ispunjenosti uslova. Neposredno iza labele `@true` se nalazi kod koji će se izvršiti ukoliko je uslov tačan i iza toga bezuslovan skok na kraj `if` iskaza (odnosno na `@exit` labelu), da izvršavanje ne bi "propalo" na `@false` labelu. Neposredno iza `@false` labele se nalazi kod koji će se izvršiti ukoliko uslov nije tačan.

Labele u ekvivalentnom (i izgenerisanom) kodu moraju biti jedinstvene. Svaka labela se završava brojem koji sadrži promenljiva `lab_num` (aktuelni broj labele). Jednoznačni brojevi se dobijaju inkrementiranjem promenljive `lab_num` za svaki sledeći iskaz.

Ukoliko bi se u prethodnom primeru izostavio `else` deo, u ekvivalentnom kodu bi nestao deo koda iza `@false` labele, kao na listingu 6.7.

Listing 6.7: primer ekvivalencije za `if` iskaz bez `else` dela

```
if(a < b)
  a = 1;
```

```
@if0:
   CMPS -4(%14),-8(%14)
   JGES @false0
@true0:
   MOV  $1,-4(%14)
   JMP  @exit0
@false0:
@exit0:
```

6.3 miniC parser sa generisanjem koda

Datoteke `codegen` (`.c` i `.h`) sarže funkcije za generisanje koda. Deo generisanja je smešten u ovim funkcijama, a deo u parseru (`micko.y`). Najčešće korišćene funkcije su opisane u nastavku.

Funkcije koje generišu labele su:

```
void gen_sslab(char *str1, char *str2);
void gen_snlab(char *str, int num);
```

Prva od njih generiše labele čija se imena sastoje od dva stringa, kao što je, na primer: `@main_exit:`. Druga funkcija generiše labele čija se imena sastoje od stringa i broja, kao što je, na primer: `@if0:`.

Za generisanje naredbe poređenja predviđena je funkcija `gen_cmp()` čiji su parametri indeksi operanada u tabeli simbola:

```
void gen_cmp(int operand1_index, int operand2_index);
```

Funkcija `gen_mov()` generiše `MOV` naredbu, a parametri su joj indeks ulaznog i indeks izlaznog operanda u tabeli simbola:

```
void gen_mov(int input_index, int output_index);
```

Funkcija za generisanje aritmetičkih naredbi ima 3 parametra: prvi predstavlja oznaku operacije (konstanta `ADD` ili `SUB`, vidi `defs.h`), dok su drugi i treći parametar indeksi operanada u tabeli simbola:

```
int gen_arith(int stmt, int op1_index, int op2_index);
```

Ako drugi ili treći argument poziva predstavljaju registre, ove naredbe ih oslobađaju. Za izlazni operand ovih naredbi (za smeštanje rezultata izvršavanja `ADD` ili `SUB` naredbe) se zauzima prvi slobodan registar.

Funkcija, koja generiše poziv funkcije prihvata indeks elementa tabele simbola u kom se nalazi funkcija čiji poziv treba izgenerisati:

```
void gen_fcall(int name_index);
```

Generisanje koda se realizuje u parseru, tako da se faza generisanja koda, praktično, odvija zajedno sa sintaksnom i semantičkom analizom. Sledi `micko.y` datoteka: parser sa semantičkim proverama i generisanjem hipotetskog asemblerskog koda. Parser pre početka parsiranja napravi `.asm` datoteku i zatim u nju generiše kod. Ukoliko parsiranje prođe bez greške, ova datoteka će biti validna. Ako se pojave greške u toku parsiranja, parser će obrisati ovu datoteku, jer sigurno nije validna.

U prvom delu `bison` specifikacije dodate su promenljive: `lab_num` - brojač labela i `output` - izlazni fajl u koji će biti izgenerisan hipotetski asemblerski program:

Listing 6.8: `micko.y` - promenljive u parseru

```
int lab_num = -1;
FILE *output;
```

Generisanje koda za definiciju funkcije podrazumeva generisanje (vidi listing 6.3):

1. labele koja se zove isto kao funkcija
2. smeštanja starog pokazivača frejma (registar `%14`) na stek:

```
PUSH %14
```

3. postavljanja novog pokazivača frejma (`%14`): prebacivanjem vrednosti stek pokazivača u pokazivač frejma:

   ```
   MOV %15,%14
   ```

4. tela funkcije (na čijem početku će se izgenerisati zauzimanje prostora za lokalne promenljive)
5. `exit` labele funkcije
6. oslobađanja prostora zauzetog za lokalne promenljive: pomeranjem stek pokazivača na frejm pokazivač:

   ```
   MOV %14,%15
   ```

7. vraćanja starog pokazivača frejma:

   ```
   POP %14
   ```

8. naredbe `RET`, koja će sa steka skinuti povratnu adresu i dovesti do preusmeravanja toka izvršavanja na deo koda iz kog je funkcija pozvana.

Prva tri elementa generisanja koda treba da se izvrše na početku parsiranja funkcije, tj. odmah nakon prepoznavanja imena funkcije (listing 6.9). Telo funkcije se generiše tokom parsiranja pojma `body`. Na kraju parsiranja funkcije se generišu elementi 5-8.

Listing 6.9: `micko.y` - definicija funkcije

```
function
  : type _ID
      {
        fun_idx = insert_symbol($2, FUN, $1, NO_ATTR);

        code("\n%s:", $2);
        code("\n\t\t\tPUSH\t%%14");
        code("\n\t\t\tMOV␣\t%%15,%%14");
      }
    _LPAREN parameter _RPAREN
      {
        set_attr(fun_idx, $5);
        var_num = 0;
      }
    body
      {
        if (get_last_element() > fun_idx)
          clear_symbols(fun_idx + 1);

        gen_sslab($2,"_exit");
        code("\n\t\t\tMOV␣\t%%14,%%15");
        code("\n\t\t\tPOP␣\t%%14");
        code("\n\t\t\tRET");
      }
  ;
```

Ukoliko se u telu funkcije (pojam `body`) nalaze lokalne promenljive (pojam `variable_list`), potrebno je izgenerisati zauzimanje prostora za njih na steku (listing 6.10). Zauzimanje prostora se vrši pomeranjem stek pokazivača za onoliko lokacija koliko ima lokalnih promenljivih (svaka lokacija zauzima 4 bajta). Zato se u ovu svrhu generiše oduzimanje vrednosti `4*var_num` od registra `%15` (`var_num`: vidi listing 5.5).

Zbog preglednosti izgenerisanog koda, na početku tela funkcije, generiše se `body` labela funkcije.

Listing 6.10: micko.y - telo funkcije

```
body
  : _LBRACKET variable_list
      {
        if (var_num)
          code("\n\t\t\tSUBS\t%%15,$%d,%%15", 4*var_num);
        gen_sslab(get_name(fun_idx), "_body");
      }
    statement_list _RBRACKET
  ;
```

Kao ekvivalent iskaza dodele, generiše se MOV naredba (listing 6.11). Funkciji gen_mov() se, kao prvi argument, prosleđuje $3 koji sadrži indeks elementa tabele simbola na kom se nalazi ulazni operand naredbe MOV, tj. rezultat izraza sa desne strane znaka jednakosti. Drugi argument je indeks elementa tabele simbola na kom se nalazi identifikator sa leve strane znaka jednakosti, koji predsavlja izlazni operand naredbe MOV.

Listing 6.11: micko.y - iskaz dodele

```
assignment_statement
  : _ID _ASSIGN num_exp _SEMICOLON
      {
        int i;
        if( (i = lookup_symbol($1, (VAR|PAR))) == -1 )
          err("invalid␣lvalue␣'%s'␣in␣assignment", $1);
        else
          if(!check_types(i, $3))
            err("incompatible␣types␣in␣assignment");
        gen_mov($3, i);
      }
  ;
```

Za generisanje aritmetičkih operacija koristi se funkcija gen_arith() koja generiše neku varijantu ADD ili SUB naredbe (listing 6.12). Kao prvi argument prosleđuje se oznaka aritmetičke operacije (konstanta ADD ili SUB, koja se nalazi u meta-promenljivoj $2), kao drugi argument se prosleđuje vrednost $1 koja sadrži indeks prvog operanda (rezultata num_exp izraza) u tabeli simbola, a kao treći argument se prosleđuje vrednost $3 koja sadrži indeks drugog operanda (rezultata exp izraza) u tabeli simbola. Funkcija gen_arith() zauzima prvi slobodan registar, u njega smešta rezultat operacije i vraća indeks tog registra u tabeli simbola.

Listing 6.12: micko.y - aritmetičke operacije

```
num_exp
  : exp

  | num_exp _AROP exp
      {
        if(!check_types($1, $3))
          err("invalid␣operands:␣arithmetic␣operation");
        $$ = gen_arith($2, $1, $3);
      }
  ;
```

U pravilu za izraze (exp), samo pravilo, koje opisuje poziv funkcije, sadrži generisanje koda (listing 6.13). Kada se poziv funkcije nađe negde u izrazu, njenu povratnu vrednost treba kopirati u prvi slobodan registar. Ovo je neophodno, jer je moguće da se u izrazu ponovo zatekne poziv neke funkcije koji će njenu povratnu vrednost ponovo smestiti u registar %13 (i

time poništiti prethodnu vrednost). Zato povratne vrednosti funkcija u izrazima treba čuvati u registrima. Vrednost pojma `exp` postaje indeks elementa u tabeli simbola, u kom se nalazi rezultat izvršavanja funkcije.

Listing 6.13: `micko.y` - izrazi

```
exp
  : literal

  | _ID
      {
        if( ($$ = lookup_symbol($1, (VAR|PAR))) == -1)
          err("'%s' undeclared", $1);
      }

  | function_call
      {
        $$ = take_reg();
        gen_mov(FUN_REG, $$);
      }

  | _LPAREN num_exp _RPAREN
      { $$ = $2; }
  ;
```

Za generisanje poziva funkcije koristi se funkcija `gen_fcall()` koja generiše `CALL` naredbu (listing 6.14). Posle poziva funkcije treba obrisati deo steka na kom su bili argumenti funkcije. Za to je zadužena funkcija `gen_clear_args()` kojoj se prosledi broj argumenata, a ona generiše `ADDS` naredbu kojom pomeri pokazivač steka za onoliko lokacija naviše koliko ima argumenata. Vrednost pojma `function_call` postaje registar `%13`, a njegov tip se postavlja na povratni tip funkcije.

Listing 6.14: `micko.y` - poziv funkcije

```
function_call
  : _ID
      {
        if((fcall_idx = lookup_symbol($1, FUN)) == -1)
          err("'%s' is not a function", $1);
      }
    _LPAREN argument _RPAREN
      {
        if (get_attr(fcall_idx) != $4)
          err("wrong number of arguments to function '%s'",
            get_name(fcall_idx));
        gen_fcall(fcall_idx);
        gen_clear_args($4);
        set_reg_type(FUN_REG, get_type(fcall_idx));
        $$ = FUN_REG;
      }
  ;
```

U toku parsiranja pojma `argument` treba generisati naredbu `PUSH` kojom će se argument poziva funkcije smestiti na stek (listing 6.15). Kako se parsiranje ovog pojma odvija pre generisanja poziva funkcije, to će naredba `PUSH` prethoditi naredbi `CALL` u asemblerskom programu.

Listing 6.15: `micko.y` - argument

```
argument
```

```
  : /* empty */
      { $$ = 0; }

  | num_exp
      {
        if(get_ptyp(fcall_idx) != get_type($1))
           err("incompatible␣type␣for␣argument␣in␣'%s'",
              get_name(fcall_idx));
        gen_push($1);
        $$ = 1; // samo 1 argument
      }
  ;
```

U okviru `if` iskaza treba izgenerisati više asemblerskih naredbi (listing 6.16). Na početku `if` iskaza generiše se labela `@ifX`, gde je `X` redni broj iskaza u programu. Brojač (`lab_num` (vidi 6.2.3)) sa ovom vrednošću se na početku generisanja `if` iskaza inkrementira, da bi označio sledeći iskaz.

Listing 6.16: `micko.y` - `if` iskaz - 1

```
if_part
  : _IF _LPAREN
      {
        lab_num++;
        gen_snlab("if", lab_num);
      }
    rel_exp
      {
        code("\n\t\t\t%s\t@false%d",
             get_opjump($4),lab_num);
        gen_snlab("true", lab_num);
      }
    _RPAREN statement
      {
        code("\n\t\t\tJMP␣\t@exit%d", lab_num);
        gen_snlab("false", lab_num);
      }
  ;
```

Na mestu relacionog izraza će se izgenerisati odgovarajuća `CMP` naredba (vidi listing 6.18). Nakon toga se generiše skok sa suprotnim uslovom na odgovarajuću `@false` labelu. Ako je, na primer, u relaciji naveden operator `==`, izgenerisaće se skok `JNE`. Niz `opposite_jumps[]` sadrži stringove za generisanje naredbi skoka (vidi `defs.h`). U nastavku se generiše `true` labela sa brojem `lab_num`. Iza `true` labele, u toku parsiranja pojma `statement`, biće izgenerisani iskazi koji čine `then` telo `if`-a. Posle toga, treba izgenerisati bezuslovni skok `JMP` na kraj `if` iskaza, čime je obezbeđeno da se, odmah nakon `then` tela, neće izvršiti i `else` telo `if`-a. Zatim sledi generisanje `false` labele, iza koje će uslediti generisanje `else` tela (ako postoji).

Na kraju `if` iskaza, generiše se `exit` labela, kao oznaka kraja ekvivalentnog asemblerskog koda `if` iskaza (listing 6.17).

Listing 6.17: `micko.y` - `if` iskaz - 2

```
if_statement
  : if_part %prec ONLY_IF
      { gen_snlab("exit", lab_num); }

  | if_part _ELSE statement
      { gen_snlab("exit", lab_num);}
  ;
```

Ako bi se u izvornom kodu pojavila situacija da `then` ili `else` telo `if`-a sadrže drugi `if` ili neki drugi iskaz, koji u svom generisanju koda takođe koristi brojač `lab_num`, bilo bi potrebno sačuvati vrednost ovog brojača pre ulaska u novi iskaz, i vratiti vrednost posle parsiranja novog iskaza (zbog nastavka parsiranja spoljašnjeg `if`-a). Ove vrednosti se mogu čuvati na nekom steku labela. Izvorni miniC kompajler MICKO, ne podržava pojavu ugnježdenih `if` iskaza.

Ekvivalent relacionog izraza (`rel_exp`) u hipotetskom asemblerskom jeziku je naredba poređenja `CMP`. Za generisanje ove naredbe poziva se funkcija `gen_cmp()` sa argumentima koji sadrže indekse elemenata u tabeli simbola u kojima se nalaze operandi operacije poređenja (listing 6.18). Vrednost pojma `rel_exp` postaje redni broj naredbe skoka u nizu `opposite_jumps[]` (vidi `defs.h`).

Listing 6.18: `micko.y` - relacioni izraz

```
rel_exp
  : num_exp _RELOP num_exp
      {
        if(!check_types($1, $3))
          err("invalid␣operands␣to␣relational␣operator");
        $$ = $2 + (get_type($1) - 1) * RELOP_NUMBER;
        gen_cmp($1, $3);
      }
  ;
```

U okviru `return` iskaza treba izgenerisati `MOV` naredbu koja će rezultat izraza `num_exp` prebaciti u registar `%13` (`FUN_REG`), jer je to dogovoreno mesto za smeštanje povratne vrednosti funkcije (listing 6.19). Iza toga se generiše bezuslovni skok (`JMP`) na `exit` labelu, koja označava kraj tela funkcije, čime se realizuje semantika `return` iskaza.

Listing 6.19: `micko.y` - `return` iskaz

```
return_statement
  : _RETURN num_exp _SEMICOLON
      {
        if(!check_types(fun_idx, $2))
          err("incompatible␣types␣in␣return");
        gen_mov($2, FUN_REG);
        code("\n\t\t\tJMP␣\t@%s_exit",
             get_name(fun_idx));
      }
  ;
```

Funkcija `main()` pre početka parsiranja obavi incijalizaciju tabele simbola i kreira i otvori `.asm` datoteku za generisanje izlaznog koda. Zatim poziva parser sa generisanjem koda, pa nakon parsiranja briše tabelu simbola i zatvara datoteku. Ako je u toku parsiranja bilo grešaka, briše izlaznu datoteku jer nije validna i prijavljuje korisniku ukupan broj grešaka.

Listing 6.20: `micko.y` - `main()` funkcija

```
int main() {
  int synerr;
  init_symtab();
  output = fopen("output.asm", "w+");

  synerr = yyparse();

  clear_symtab();
  fclose(output);
```

```
  if(warning_count)
    printf("\n%d warning(s).\n", warning_count);

  if(error_count) {
    remove("output.asm");
    printf("\n%d error(s).\n", error_count);
  }

  if (synerr)
    return -1;
  else
    return error_count;
}
```

6.3.1 Primer upotrebe parsera sa generisanjem koda

Ako se parseru sa generisanjem koda prosledi datoteka `abs.mc` (listing 2.1), kompajler će izgenerisati datoteku `abs.asm` koja sadrži ekvivalentan kod, napisan na hipotetskom asemblerskom jeziku (listing 2.2).

6.4 Vežbe

Proširiti miniC parser tako da implementira generisanje koda za nove osobine miniC jezika:

1. globalne promenljive
2. definicija promenljive sa inicijalizacijom `int a = 5;`
3. definicija funkcije sa više parametara i poziv funkcije sa više argumenata
4. mogućnost generisanja koda za ugnježdene `if` iskaze
5. `while` iskaz

Listing 6.21: primer ekvivalencije za `while` iskaz

```
while (a < b)                    @while0:
  b = b - a;                         CMPS a,b
                                     JGES @false0
                                  @true0:
                                     SUBS b,a,%0
                                     MOV  %0,b
                                     JMP  @while0
                                  @false0:
                                  @exit0:
```

6. `break` i `continue` iskazi (podrazumeva se da se smeju naći samo unutar `while` iskaza)

Listing 6.22: primer ekvivalencije za `continue` iskaz

```
while(a < 5) {                   @while0:
   if(a == b)                        CMPS a,$5
      continue;                      JGES @false0
   a = a + 1;                    @true0:
}                                @if1:
                                     CMPS a,b
                                     JNE  @false1
                                 @true1:
                                     JMP  @while0    //continue
```

```
    JMP  @exit1
@false1:
@exit1:
    ADDS a,$1,%0
    MOV  %0,a
    JMP  @while0
@false0:
@exit0:
```

7. `for` iskaz

Listing 6.23: primer ekvivalencije za `for` iskaz

```
for(i = 0; i <= 5; i++)
   suma = suma + i;
```

```
    MOV  $0,i
@for0:
    CMPS i,$5
    JGTS @exit0
    ADDS suma,i,%0
    MOV  %0,suma
    ADDS i,$1,i
    JMP  @for0
@exit0:
```

8. `switch` iskaz

Listing 6.24: primer ekvivalencije za `switch` iskaz

```
switch(state) {
   case 10 : state = 1; break;
   case 20 : state = 2; break;
   default : state = 0;
}
```

```
@switch0:
    JMP  @test0
@case0_0:
    MOV  $1,state
    JMP  @exit0
@case0_1:
    MOV  $2,state
    JMP  @exit0
@default0:
    MOV  $0,state
    JMP  @exit0
@test0:
    CMPS state,$10
    JEQ  @case0_0
    CMPS state,$20
    JEQ  @case0_1
    JMP  @default0
@exit0:
```

9. logički izrazi sa operatorima `&&` i `||`
10. slogovi
11. nizovi
12. blokovi iskaza unutar tela funkcije

Listinzi

Slike

Indeks

Bibliografija

[1] J. Levine. *flex & bison*. Oreilly Series. O'Reilly Media, 2009.

[2] Steven S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[3] Predrag Rakic Zorica Suvajdzin Rakic. *flex & bison*. FTN, 2013.