Zbirka zadataka
Programski prevodioci
!!! RADNA VERZIJA !!!

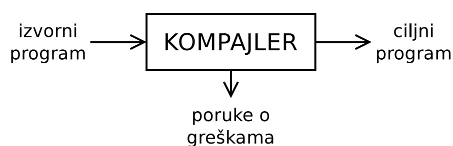
Zorica Suvajdžin Rakić
Miroslav Hajduković
Predrag Rakić
Žarko Živanov

5. decembar 2012

Glava 1

Kompajler

Kompajler je program koji čita program napisan na jednom programskom jeziku (izvorni jezik) i prevodi ga na ekvivalentni program napisan na drugom programskom jeziku (ciljni jezik) [1]. Proces prevođenja se zove kompajliranje (*compilation*) i ne sme da izmeni značenje izvornog programa.



Izvorni jezik (*source language*) je programski jezik visokog nivoa koji je prilagođen programerima (ljudima), dok je ciljni jezik (*target language*) programski jezik niskog nivoa koji zahtevaju mašine-računari.

Tokom kompajliranja, kompajler prijavljuje korisniku eventualno postojanje grešaka u izvornom programu.

Da bi kompajler mogao da prevede izvorni kod u neki drugi, potrebno je da poznaje strukturu izvornog jezika. Tada može prevoditi pojedinačne iskaze izvornog jezika u ekvivalentne ciljne iskaze.

1.1 Sintaksa programskog jezika i gramatika

Svaki programski jezik ima pravila koja određuju sintaksnu strukturu programa. Na primer, C program, se sastoji od definicija funkcija, funkcije od naredbi, naredbe od izraza, izrazi od simbola, itd. Sintaksa programskog

jezika je skup pravila koja definišu validne kombinacije simbola u tom jeziku. Sintaksa se opisuje gramatikom (*grammar*), obično pomoću BNF notacije (*Bacus-Naur Form*) [5].

Gramatika nekog programskoj jezika se sastoji od simbola (*terminal symbol*), pojmova (*nonterminal symbol*), pravila (*productions*) i polaznog pojma (*start symbol*) [7]. Gramatike nude velike prednosti i onima koji projektuju jezike i onima koji pišu kompajlere.

Sintaksa jezika opisuje formu validnog programa, ali ne sadrži informacije o značenju programa niti o rezultatima izvršavanja programa. Značenjem koje ima neka kombinacija simbola se bavi semantika. Semantika se opisuje neformalno (tekstualno) u opisu (uputstvu) programskog jezika ili mnogo ređe nekim formalizmom.

Ako je program sintaksno ispravan ne mora da znači da je i semantički ispravan. Na primer, `C` iskaz `a = b + 3;` je sintaksno ispravan ali semantički neispravan u slučaju da promenljiva `a` nije prethodno definisana u programu.

1.2 Faze kompajliranja

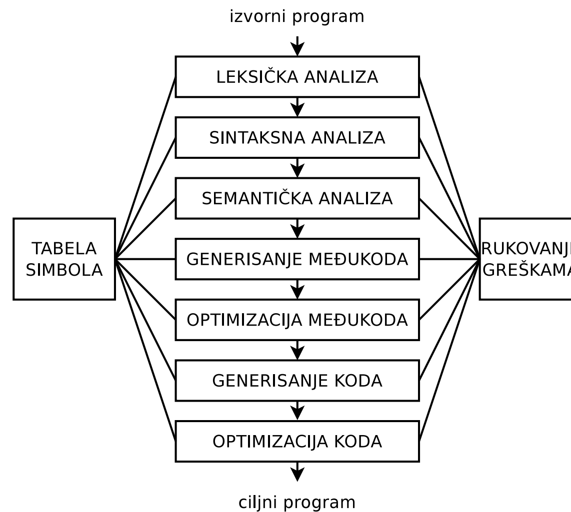
Pisanje (implementacija) kompajlera nije trivijalan posao i zato ga je zgodno podeliti u nekoliko faza. Konceptualno, ove faze se odvijaju sekvencijalno, međutim u praksi su često preklopljene. Faze međusobno komuniciraju tako što je izlaz iz prethodne faze ulaz u sledeću fazu. Uobičajena podela na faze je prikazana na slici 1.1 i opisana u nastavku.

Leksička analiza Ovo je početni deo čitanja i analiziranja programskog teksta. Tekst se pročita i podeli u tokene, tako da svaki token odgovara simbolu u programskom jeziku (npr. ključna reč, ime promenljive ili broj).

Sintaksna analiza Ova faza 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 Ova faza proverava konzistentnost programa: npr. da li je promenljiva koja se koristi prethodno deklarisana i da li se koristi u skladu sa svojim tipom - npr. korišćenje `boolean` vrednosti kao pokazivača.

Generisanje međukoda Program se prevodi na jednostavan međujezik (*intermediate language*) koji je nezavisan od mašine (*machine-independent*).



Slika 1.1: Faze kompajliranja

Generisanje koda Međukod se prevodi na asemblerski jezik (tekstualna prezentacija mašinskog jezika) za određenu arhitekturu mašine ili na mašinski jezik (binarna reprezentacija) [6].

U praksi, faze su često grupisane u *front end* i *back end*. *Front end* se sastoji od faza koje prvenstveno zavise od izvornog jezika i koje su velikom delom nezavisne od ciljne mašine. Ovde spadaju leksička i sintaksna analiza, semantička analiza i generisanje međukoda. *Back end* sadrži one delove kompajlera koji zavise od ciljne mašine, i koji generalno ne zavise od izvornog jezika, već samo od međujezika. Ovde spadaju faze generisanja koda i optimizacije. Da bi se napravio kompajler za isti izvorni jezik, ali za drugu ciljnu mašinu, dovoljno je preuzeti *front end* postojećeg kompajlera i napraviti redizajn *back end* dela kompajlera.

Nekoliko faza kompajliranja se obično implementira kao jedan prolaz (*pass*), a aktivnosti ovih faza se tokom prolaza međusobno prepliću. Jedan prolaz se sastoji od čitanja ulaznog fajla i zapisivanja izlaznog fajla. Pošto su operacije čitanja i pisanja fajlova spore, i pošto je veličina interne forme programa znatno veća i od izvornog i od ciljnog programa, poželjno je da kompajler ima što manje prolaza.

Sve faze kompajliranja su u vezi sa tabelom 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 seman-

tičku analizu i generisanje (među) koda.

1.3 Rukovanje greškama

Kada bi kompajler trebao da obrađuje samo ispravne programe, njegovo osmišljavanje i implementacija bi bili znatno pojednostavljeni. Međutim, programeri često pišu neispravne programe, pa dobri kompajleri treba da pomognu programeru u identifikovanju i lociranju grešaka. Programi mogu sadržati greške na različitim nivoima. Na primer, greške mogu biti:

– leksičke (pogrešno napisano ime, ključna reč ili operator)

– sintaksne (logički izraz sa nepotpunim parom zagrada)

– semantičke (operator primenjen na nekompatibilni operand)

– logičke (beskonačan rekurzivni poziv)

Rukovaoc greškama u parseru ima nekoliko ciljeva:

– da saopšti prisustvo grešaka jasno i ispravno

– da se oporavi od greške dovoljno brzo da bi mogao da detektuje naredne greške

– da ne usporava bitno obradu ispravnih programa.

U svakoj fazi kompajliranja se može pojaviti greška, pa zato svaka faza mora postupiti sa greškom tako, da se proces kompajliranja može nastaviti, sa ciljem detekcije još novih grešaka. Najvećim delom grešaka (koje kompajler može da otkrije) rukuju sintaksna i semantička analiza.

1.4 Implementacija

Kompajler je veoma kompleksan program. Za neke izvorne jezike se može napisati ručno, dok je za neke druge to gotovo nemoguće uraditi. Zato postoje generatori kompajlera (*compiler generator* ili *compiler-compiler*).

Neki od najpoznatijih i najkorišćenijih alata koji se koriste za generisanje skenera su:

- lex - generiše C kod [3]
- flex - generiše C, C++ kod [4]
- jlex - generiše Java kod

a za generisanje parsera su:

- yacc, yacc++ - generiše C odnosno C++ kod [2]
- bison, bison++ - generiše C odnosno C++ kod [4]
- CUP, JavaCC, CookCC - generiše Java kod
- ANTLR - generiše Ada95, ActionScript, C, C#, Java, JavaScript, Objective-C, Perl, Python i Ruby kod
- Coco/R - generiše Java, C#, C++, Pascal, Modula-2, Modula-3, Delphi, VB.NET, Python i Ruby kod
- BYACC, BYACC/J - generiše C, C++ i Java kod

1.5 Sadržaj kursa

Ovaj kurs ima za cilj da prikaže tehnike implementacije svih faza kompajliranja. Biće korišćeni alati flex i bison [4], što znači da će se generisati C skener i C parser, kao i da će sav dodatni kod biti pisan na C programskom jeziku. Kroz kurs će biti analiziran postojeći (delimično implementiran) kompajler za mC programski jezik, a na vežbama će biti dopunjavan novim sintaksnim konstrukcijama. mC kompajler prevodi mC kod na hipotetski asemblerski jezik.

Glava 2

Leksička analiza - skener

Leksički analizator (skener) je program namenjen za leksičku analizu (skeniranje) teksta. Skener radi tako što čita tekst, karakter po karakter, i pokušava da prepozna neku od zadatih reči. Kada prepozna reč (sekvencu karaktera) izvrši zadatu akciju.

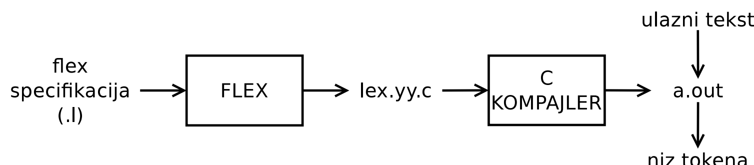
Skener se može napraviti (isprogramirati) ručno ili korišćenjem alata za generisanje skenera. Tokom ovog kursa ćemo za generisanje skenera koristiti program `flex`. `Flex` generiše skener na osnovu pravila zadatih u `flex` specifikaciji. Specifikaciju kreira korisnik u posebnoj datoteci koja po konvenciji 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.

`.l` datoteka se prosleđuje `flex`-u, koji po *default-u*, generiše skener na jeziku C, u globalnoj 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 može da se prevede i C i C++ kompajlerom (tj. može da se uključi u C ili u C++ program). Ako se za generisanje skenera, umesto programa `flex` pozove program `flex++`[1], skener biva generisan u C++ funkciji `yylex()` članici klase `yyFlexLexer` ili članici klase proizvoljnog imena koja nasleđuje klasu `yyFlexLexer`. Ovako generisan skener može da se uključi samo u C++ program.

Ovakav program predstavlja leksički analizator koji transformiše ulazni tekst u niz tokena (u skladu sa pravilima zadatim u specifikaciji).

[1] U pitanju je isti program, samo se drugačije ponaša kada mu se ime završava znakom '+'.

Korišćenje `flex`-a je prikazano na slici:



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.

2.1 Format `flex` specifikacije

Specifikacioni `flex` fajl se sastoji od tri dela, međusobno razdvojena oznakom `%%` (dva znaka procenat jedan do drugog):

1. definicije
2. pravila i
3. korisnički kod.

2.1.1 Definicije

Prvi deo se obično koristi za uključivanje zaglavlja i definicije promenljivih (pri čemu ovakav kod mora biti okružen specijalnim zagradama `%{` i `%}`). U ovom delu se navode još i regularne definicije, `flex` makroi, početni uslovi, kao i opcije za upravljanje radom generatora. Ovaj deo `flex` specifikacije nije obavezan.

`flex` nudi razne opcije za pravljenje skenera. Većina se može navesti kao `%option name` u prvom delu specifikacije, ili kao `--name` u komandnoj liniji prilikom poziva `flex`-a. Da bi se neka opcija isključila, pre imena opcije treba dodati string "`no`", kao na primer `%option noyywrap` ili `--noyywrap`.

Opcije koje ćemo koristiti su:

```
%option noyywrap ili --noyywrap
```

Ovom opcijom korisnik kaže `flex`-u da ne želi da definiše funkciju `yywrap()`. Ovu funkciju poziva skener kada dođe do kraja ulazne datoteke. U njoj korisnik može da opiše ponašanje skenera posle čitanja datoteke (npr. da nastavi čitanje neke druge datoteke). Podrazumeva se da će (ako `yywrap()` nije definisana) skener preuzeti i čitati samo jednu ulaznu datoteku i da će vratiti pozivaocu vrednost 0.

```
%option yylineno ili --yylineno
```

Zgodno je znati broj skenirane linije u ulaznom fajlu kada, na primer, treba prijaviti leksičku grešku. `flex` definiše promenljivu `yylineno` koja sadrži broj trenutne linije i automatski je ažurira svaki put kada pročita znak `\n` . Skener ne inicijalizuje ovu promenljivu, pa joj zato korisnik sam mora dodeliti vrednost 1 svaki put kada započinje čitanje ulaznog fajla.

2.1.2 Pravila

Drugi deo specifikacije sadrži niz pravila koja opisuju ponašanje skenera. Na osnovu ovih pravila `flex` generiše kod skenera. Svako pravilo se zadaje u obliku:

```
obrazac [ akcija ]
```

Obrasci se zadaju korišćenjem proširenog skupa regularnih izraza (datog u nastavku), a akcije blokom C (ili C++) koda koji će se izvršiti kada skener prepozna obrazac. Obrazac mora da počne u prvoj koloni, a akcija ne mora da postoji. Ako akcija postoji, njena definicija mora da počne u istom redu u kojem je definisan obrazac. Ovaj deo `flex` specifikacije je obavezan.

2.1.2.1 `flex` regularni izrazi

U tabeli u nastavku su dati operatori koji se koriste za pisanje regularnih izraza u `flex`-u.

regularni izraz	značenje
x	karakter x
"x"	x, čak i ako je x operator
\x	x, čak i ako je x operator
x?	opciono x (0 ili 1 instanca)
x*	0 ili više instanci x
x+	1 ili više instanci x
x|y	x ili y
[xy]	x ili y
[x-z]	karakteri x, y ili z
[^x]	bilo koji karakter osim x
.	bilo koji karakter osim newline
^x	x na početku linije
x$	x na kraju linije
(x)	x
x/y	x ali ako i samo ako iza njega sledi y
{xx}	pravilo za xx iz prvog dela specifikacije
x{m,n}	m do n pojava x

Da bi karakter koji predstavlja `flex` metasimbol mogao da ima i svoje prirodno značenje, ispred njega treba staviti kosu crtu (\) ili ga navesti pod navodnicima (“”). Tako, na primer, regularn i izraz `a\.b` opisuje string ''`a.b`'' (tačka se tretira kao običan znak), dok regularni izraz `a.b` opisuje string ''`a`'' iza kojeg sledi bilo koji znak osim `newline` iza kojeg sledi znak ‘‘`b`” (tačka se tretira kao operator).

2.1.3 Korisnički kod

U ovom opcionom delu korisnik može da definiše proizvoljan blok koda (funkcija `main()`, druge funkcije, globalne promenljive, ...) koji će biti prekopiran neizmenjen na kraj generisanog skenera.

Ovaj deo datoteke omogućava da se ceo program smesti u samo jednu (`.l`) datoteku. Kompleksni programi se svakako (zbog čitljivosti) raspoređuju u više izvornih datoteka, pa tada ovaj deo, uglavnom, i ne postoji.

2.1.4 Komentari

U `flex`-u se koriste komentari iz C programskog jezika:

jednolinijski `//`

višelinijski `/* ... */`

Komentari se pišu uvučeni (indentovani) bar za jedno prazno mesto.

2.2 `flex` skeneri

U ovom delu su prikazane osnovne karakteristike programa `flex` kroz nekoliko jednostavnih primera skenera. Primeri su implementirani na jeziku C.

2.2.1 Bez ijednog pravila

Najjednostavnija `flex` specifikacija (ovde nazvana `klot.l`) je prikazana u listingu 2.1:

Listing 2.1: klot.l

```
%%
```

Ovaj fajl sadrži samo jedan simbol od 2 karaktera - dva znaka procenat. Skener izgenerisan iz ove specifikacije (koja ne sadrži nijedno pravilo) prihvata karaktere sa standardnog ulaza i kopira ih na standardni izlaz (karakter po karakter). Iz ovog jednostavnog primera se vidi da će svi karakteri iz ulaznog teksta koji ne budu prepoznati (koji se ne uklapaju ni u jedan od zadatih obrazaca) biti ispisani na standardnom izlazu.

Skener se generiše, kompajlira i pokreće naredbama:

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

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

Svič `--noyywrap` prosleđen `flex`-u kaže da korisnik ne želi da definiše funkciju `yywrap()`.

U drugoj liniji pozivamo `gcc` kompajler i prosleđujemo mu `lex.yy.c` datoteku.

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 izvršni fajl sa imenom `a.out`.

Uz program `flex` dolazi i biblioteka `libl.a`. U ovoj biblioteci je definisana funkcija `main()` koja jedino poziva skener, odnosno funkciju `yylex()`:

```
int main() {
   while (yylex() != 0) ;
   return 0;
}
```

Ako u programu (u našem slučaju `klot.c`) nije definisana funkcija `main()` koristi se ona iz biblioteke. Svič `-l` govori linkeru da pogleda u biblioteku čije ime je navedeno iza sviča. Tom prilikom se iz imena biblioteke izostavljaju prva tri slova: 'lib' (jer se ona podrazumevaju[2]) i ekstenzija. Otuda, "`-l l`" u pozivu `gcc`-a.

U trećoj liniji naredbom `./klot` pokrećemo izgenerisani skener u interaktivnom režimu, što znači da ulaz očekuje preko tastature, a izlaz iz programa se zadaje kombinacijom `^D`.

Ukoliko želimo skeneru da prosledimo datoteku (`test.txt`) na skeniranje, upotrebićemo redirekciju standardnog ulaza:

```
$ ./klot <test.txt
```

A ako pokrenemo skener u interaktivnom režimu, to može da izgleda ovako:

```
$ ./klot
tekst
tekst
se prekopira
se prekopira
na standardni izlaz
na standardni izlaz
^D
$
```

2.2.2 Obrasci

Obrasci opisuju sekvence (nizove) karaktera koje skener treba da prepozna. Kada je potrebno prepoznati samo jednu sekvencu karaktera (kao na primer

[2] Konvencija je da nazivi svih biblioteka počinju slovima 'lib'.

ključnu reč `if` ili kao u listingu 2.2 reč ``njam'') tu sekvencu je moguće eksplicitno zadati.

Ako se prethodno prikazan `flex` fajl `klot.l` preimenuje u `njam.l` i proširi jednim pravilom koje prepoznaje reč "njam" (i kada je prepozna ne radi ništa), dobije se `flex` konfiguracioni fajl prikazan na listingu 2.2:

Listing 2.2: njam.l

```
%%

njam
```

Ako se zatim, ponovo izgeneriše, kompajlira i pokrene, program `njam` se ponaša slično programu `klot` - većinu karaktera koje primi na ulazu on prosledi na izlaz. Ako se na ulazu pojavi niz karaktera ``njam'' oni neće biti prosleđeni na izlaz, već će nestati ("poješće" ih skener - otuda 'njam'). Jedna interaktivna sesija programa `njam` izgleda ovako:

```
$ ./njam
Ovo prolazi bez izmena.
Ovo prolazi bez izmena.
njam

Cunjam ovuda.
Cu ovuda.
```

Kada je potrebno prepoznati sve sekvence karaktera iz neke klase (kao na primer: "broj je niz od jedne ili više cifara"), tada se za specifikaciju obrasca koriste složeniji izrazi (iz proširenog skupa regularnih izraza). Sledi primer (listing 2.3) specifikacije skenera koji prepoznaje prirodne brojeve:

Listing 2.3: num.l

```
%%

[0-9]+
```

Uglaste zagrade opisuju klasu karaktera – bilo koji karakter u rasponu od ASCII koda 0 do ASCII koda 9 (tj. bilo koja cifra), a operator "+" kaže da se cifra može ponavljati jednom ili više puta (bar jedna mora postojati).

Izlaz ovog programa može da izgleda ovako:

```
$ ./num
1 i jedan jesu 2.
 i jedan jesu .
9 + 11 = 20
 +  =
```

2.2.3 Akcije

Akcija je C (ili C++) kod koji skener izvršava kada prepozna string koji odgovaraju obrascu.

Na listingu (2.4) je prikazan skener sa samo jednim pravilom koji prepoznaje beline (proizvoljno dugačke sekvence praznih mesta i tab–ova) i zamenjuje ih samo jednim praznim mestom.

Listing 2.4: ws.l

```
%%

[ \t]+       { putchar(' '); }
```

Obrazac je regularni izraz koji opisuje: “ili razmak ili tab jednom ili više puta”.

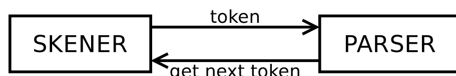
Akcija je poziv C funkcije koja ispisuje jedan karakter (u ovom slučaju prazno mesto) na standardni izlaz. Kako se ova akcija izvršava svaki put kada skener prepozna niz razmaka i/ili tabova, dobija se efekat zamene belina jednim praznim mestom.

Upotreba ovog programa:

```
$ ./ws
Neko pise   reci odvojene   jednim razmakom   a neko      ne.
Neko pise reci odvojene jednim razmakom a neko ne.
^D
$
```

2.2.4 Upotreba skenera sa parserom

Skener kreiran pomoću `flex`-a u kombinaciji sa parserom se ponaša na sledeći način: kada parser aktivira leksički analizator, on počinje sa čitanjem preostalog ulaznog teksta, jedan po jedan karakter, dok ne nađe najduži niz karaktera koji odgovara nekom obrascu. Tada, izvršava akciju koja je pridružena prepoznatom obrascu. Akcija može da vrati kontrolu parseru ukoliko se u njoj izvrši naredba `return`. Ako do toga ne dođe, leksički analizator nastavlja da čita ostatak ulaznog teksta, dok akcija ne izazove vraćanje kontrole parseru. Ponavljanje traženja stringova dok se kontrola eksplicitno ne vrati parseru, olakšava skeneru obradu praznina i komentara.



Skener parseru može da vrati token i neku vrednost koja ide uz token. Token je numerička oznaka kategorije reči (npr. token `IF` opisuje ključnu reč `if`, token NUM opisuje brojeve, token `ID` opisuje identifikatore odnosno imena). U slučajevima kada skener prepozna jednostavne reči, kao što su ključne reči (npr. `if`, `while`, ...) parseru je dovoljno proslediti (vratiti) token te ključne reči. U slučaju kada skener prepozna identifikator ili broj, parseru je osim tokena potrebno proslediti i konkretan string imena odnosno vrednost konkretnog broja. Ova vrednost se prosleđuje preko globalne promenljive `yylval`.

Na listingu 2.5 sledi primer prepoznavanja ključne reči `if`, prirodnih brojeva i reči:

Listing 2.5: union.l

```
/* Primer skenera koji pokazuje upotrebu unije */
%{
   enum { IF = 1, NUM, WORD };

   union {
      int n;
      char *s;
   } yylval;
%}

%%

if           { return IF; }
[0-9]+       { yylval.n = atoi(yytext); return NUM; }
[a-zA-Z]*    { yylval.s = yytext; return WORD; }

%%

int main() {
   int tok;
   while(tok = yylex()) {
      switch(tok) {
         case IF   : printf("IF"); break;
         case NUM  : printf("NUM: %d", yylval.n); break;
         case WORD : printf("WORD: %s", yylval.s); break;
      }
   }
}
```

Tokeni su definisani u enumeraciji u prvom delu `flex` specifikacije.

Promenljive `yylval` i `yytext` su globalne promenljive u `flex`-u. Prva se koristi za prosleđivanje vrednosti simbola parseru (uz token). Podrazumevani tip ove promenljive je `int`, a uobičajeno je da se ova promenljiva definiše kao unija (`union`). Unija je promenljiva koja može čuvati objekte različitih tipova i veličina, odnosno to je jedna promenljiva koja može sadržati vrednost jednog od nekoliko tipova (a programer je dužan da vodi računa o tome koji tip vrednosti se trenutno čuva u uniji). Druga promenljiva, `yytext`, je tipa `char*` i sadrži string poslednje prepoznate reči.

U trećem delu je navedena definicija funkcije `main`, koja sada ne poziva samo skener (kao što to radi bibliotečka implementacija funkcije `main`), već dodatno i štampa na ekran sve vrenosti koje dobije od skenera: tokene i vrednosti koje su prosleđene uz token.

Uz token `IF` se ne prosleđuje nikakva dodatna vrednost simbola, jer se ključna reč `if` može napisati na jedan jedini način.

Uz token `NUM` se prosleđuje konkretna vrednost broja (C funkcija `atoi` prihvata string i konvertuje ga u `int`).

Uz token `WORD` se prosleđuje konkretni string reči, jer se reči mogu različito napisati.

2.3 Primeri

2.3.1 Primer 1

Napraviti skener koji preuzme sve karaktere sa ulaza i ispiše:

- koliko je bilo reči
- koliko je bilo karaktera
- koliko je bilo redova.

Listing 2.6: wclc.l

```
/* Skener koji broji reci, karaktere i linije */

%option noyywrap

%{
   int chars = 0;
   int words = 0;
```

```
   int lines = 0;
%}

%%

[a-zA-Z]+    { words++; chars += strlen(yytext); }
\n           { chars++; lines++; }
.            { chars++; }

%%

main() {
   yylex();
   printf("chars: %d, words: %d, lines: %d.\n",
             chars, words, lines);
}
```

Prvi deo specifikacije sadrži opciju za isključivanje funkcije `yywrap`. Isti efekat smo u ranijim primerima postizali navođenjem sviča `--noyywrap` u komandnoj liniji prilikom poziva `flex`-a.

U prvom delu specifikacije se nalaze i definicije 3 promenljive koje će čuvati broj karaktera, reči i linija.

Prvi obrazac `[a-zA-Z]+` opisuje reč. Karakteri u zagradama opisuju jedno malo ili veliko slovo, a znak `+` opisuje jednu ili više pojava slova. Znači, ceo obrazac opisuje niz slova, tj. reč. Akcija inkrementira broj viđenih reči i karaktera. Promenljiva `yytext` pokazuje na string iz ulaznog teksta koji je prepoznat na osnovu tog obrasca. U ovom slučaju, ne zanima nas koji je to string, već koliko karaktera ima u njemu.

Drugi obrazac `\n` opisuje znak za novi red, a pridružena akcija inkrementira broj karaktera i broj redova.

Treći obrazac je regularni izraz koji opisuje bilo koji znak osim znaka za novi red. Akcija inkrementira broj karaktera.

`main` funkcija poziva skener, tj. funkciju `yylex()` i ispisuje vrednosti 3 brojača. Ako skeneru ne damo ulaznu datoteku, on će ulazni tekst čitati sa tastature:

```
$ flex wclc.l
$ gcc lex.yy.c
$ ./a.out
Eci peci pec
ti si mali zec
^D
chars: 28, words: 7, lines: 2.
```

```
$
```

Prvo kažemo `flex`-u da prevede skener na `C`, zatim kompajliramo `lex.yy.c` (`C` program koji je izgenerisan), pokrenemo program i otkucamo mali ulaz za skener. Izgleda da radi.

Neki pravi program koji broji reči i karaktere bi imao malo drugačiju definiciju reči: niz karaktera koji nisu belina, pa bi `flex` pravilo moglo da izgleda ovako:

```
[^ \t\n\r\f\v]+    { words++; chars += strlen(yytext); }
```

Operator `^` na početku zagrada znači "prepoznaj bilo koji karakter osim onih koji su navedeni u zagradama". Ovo pokazuje snagu i fleksibilnost `flex`-a - lako je napraviti izmene u obrascu i pustiti `flex` da brine kako to utiče na izgenerisani kod skenera.

2.3.2 Primer 2

Napraviti skener koji prepravlja ulazni tekst tako da ne dozvoljava beline pre tačke, a ubacuje samo jedan razmak posle tačke.

Listing 2.7: wsdot.l

```
/* Skener koji ne dozvoljava pojavu belina ispred tacke,
   i dozvoljava samo jedan space iza tacke. */

%%

[ \t\n]*"."[ \t\n]*    { printf(". "); }
.                      { ECHO; }
```

Ovaj skener treba da reaguje na beline pre i posle tačke, a sve ostale znake da "propušta" na izlaz. Zato pravimo dva pravila. Prvo pravilo opisuje postojanje 0 ili više belina pre tačke, iza kojih sledi tačka, iza koje sledi niz od 0 ili više belina. Akcija koja se izvršava u slučaju da se pronađe tačka okružena belinama je da se ceo string zameni stringom "`. `", čime zadovoljavamo zahtev zadatka da posle tačke treba da stoji samo jedno prazno mesto.

Drugo pravilo kaže da svi karakteri koji ne odgovaraju prvom pravilu treba da prođu na izlaz neizmenjeni. To se dobije akcijom koja radi eho ulaznog stringa na izlaz (`ECHO`). Ovaj program bi se ponašao isto i da smo izostavili

drugo pravilo, jer flex ima predefinisano ponašanje, da svaki karakter koji se ne uklapa ni u jedno pravilo ispiše na izlaz.

Primer pokretanja ovog programa:

```
$ ./wsdot
Prva recenica .Druga recenica.   Treca recenica.Cetvrta    .
Prva recenica. Druga recenica. Treca recenica. Cetvrta.
^D
$
```

2.3.3 Primer 3

Napraviti skener koji briše komentare iz C i C++ programa.

Listing 2.8: comments.l

```
/* Skener koji brise komentare iz C i C++ programa */

%x COMMENT

%%

    /* Blok komentari */
"/*"           { BEGIN COMMENT;
                 /* predji u stanje COMMENT */ }

<COMMENT>.|\n  { /* preskoci tekst komentara */ }

<COMMENT>"*/"  { BEGIN INITIAL;
                 /* vrati se u normalno stanje */ }

    /* Linijski komentari */
"//".*
```

U prvom delu se nalazi i definicija (ekskluzivnog) stanja `%x COMMENT`. To znači da kada je to stanje aktivno, samo obrasci koji su vezani za to stanje se mogu prepoznavati.

Prva tri pravila služe za prepoznavanje blok komentara. Prvo pravilo ulazi u stanje `COMMENT` kada prepozna `/*`, a drugo pravilo se vraća nazad u normalno `INITIAL` stanje kada prepozna `*/`. Treće pravilo prepoznaje sve što se nalazi između, odnosno tekst komentara.

Ako bismo dodali još i pravilo

```
<COMMENT><<EOF>> {
        printf("line␣%d:␣Unterminated␣comment\n", yylineno);
        return 0;
    }
```

mogli bismo da detektujemo i prijavimo nezavršene komentare.

Poslednje pravilo opisuje jednolinijske komentare (dva `slash` znaka iza kojih može da sledi proizvoljan broj znakova).

Za ulaznu datoteku koja sadrži jednu C funkciju:

```
/* vraca vrednost 1 ako se vrednost prvog parametra
   nalazi u navedenim granicama. Inace vraca 0. */
int interval(int x, int lower, int upper) {
   if(x >= lower && x <= upper)
      return 1;   // vrati bool vrednost true
   else
      return 0;   // vrati bool vrednost false
}
```

program daje izlaz:

```
int interval(int x, int lower, int upper) {
   if(x >= lower && x <= upper)
      return 1;
   else
      return 0;
}
```

2.3.4 Primer 4

Napraviti skener koji u ulaznom tekstu pronalazi binarne brojeve i prevodi ih u dekadne brojeve. Primer binarnog broja je `0B0110`.

Listing 2.9: 2to10.l

```
/* Program koji prevodi binarne brojeve u dekadne */

%{
   unsigned val = 0;
```

```
   char *i;
%}

%%

0[bB][01]{1,32}    {  for(i = yytext+2; *i!=0; ++i) {
                         val <<= 1;
                         val += *i - '0';
                      }
                      printf("%d", val);
                      val = 0;
                   }
.                  {  ECHO;  }
```

Prvi deo specifikacije sadrži definicije dve promenljive: celobrojnu neoznačenu `val` koja će služiti za konverziju vrednosti i pokazivačku promenljivu `i` koja će služiti kao iterator `for` petlje.

Prvo pravilo prepoznaje binarne brojeve kao znak `'0'` zatim malo ili veliko slovo `b`, a zatim minimalno jedna a maksimalno 32 binarne cifre. Akcija koja prati ovo pravilo radi konverziju binarnog u dekadni broj. Iterator for petlje počinje konverziju od trećeg znaka ulaznog stringa, jer preskače "`0b`". Promenljiva val se resetuje nakon ispisa da bi bila spremna za ispravnu konverziju sledećeg broja.

Drugo pravilo preusmerava sve ostale znake na izlaz.

Primer pokretanja bi mogao da izgleda:

```
$ ./2to10
123
123
0b01
1
0B0110
6
^D
$
```

Prvi broj `123` se ispisuje neizmenjen jer nije binarni broj.

Varijacije primera: prevesti heksa broj u dekadni.

2.3.5 Primer 5

Napraviti skener koji čita `float` vrednosti sa ulaza dok se na ulazu ne pojavi neka od komandi:

- reč “sum” - vraća pozivaocu sumu unetih vrednosti
- reč “mean” - vraća pozivaocu srednju vrednost unetih vrednost
- reč “quit” - dovodi do završetka programa

Listing 2.10: cmd.l

```
/* Skener koji preuzima niz float vrednosti
   i kada stigne komanda, izvrsi je.
   Komande su SUM, MEAN ili QUIT */

%{
   double sum = 0;
   int count = 0;
%}

%%

[0-9]+\.[0-9]*   { sum += atof(yytext);
                   count++; }

"sum"            { printf("Sum: %f", sum);
                   sum = 0;
                   count = 0; }

"mean"           { printf("Mean: %f", sum/count);
                   sum = 0;
                   count = 0; }

"quit"           { return 0; }

.
```

Prvo pravilo prepoznaje `float` brojeve kao jedna ili više cifara iza čega sledi tačka iza čega opciono sledi 0 ili više cifara. To znači da su tačka i cifra ispred tačke obavezni a cifre iza tačke nisu. Akcija dodeljena ovom pravilu sabira vrednost broja u promenljivoj `sum`.

Drugo i treće pravilo opisuju prepoznavanje stringa komande i izvršavaju akciju: ispisuju rezultat komande i resetuju promenljive.

Četvrto pravilo vraća vredost `0` što dovodi do kraja skeniranja i time realizuje komandu `‘‘quit’’`.

Poslednje pravilo sve ostale znake prosleđuje na izlaz.

Primer pokretanja programa:

```
$ ./cmd
1.1 2.2 sum
Sum: 3.300000
5.5 3.5 6. mean
Mean: 5.000000
quit
```

2.3.6 Primer 6

Napraviti skener koji procesira tekst i ako naiđe na datum u obliku `dd/mm/yyyy`, menja ga u oblik `dd-mmm-yyyy`.

Listing 2.11: date.l

```
/* Skener koji menja formu datuma
   iz dd/mm/yyyy  u  dd-mmm-yyyy  */

%option noyywrap yylineno

%{
   char *months[] = { "", "jan", "feb", "mar", "apr", "maj",
      "jun", "jul", "avg", "sep", "okt", "nov", "dec" };
%}
%%

[0-9]{2}\/[0-9]{2}\/[0-9]{4}    { printf("%d-%s-%d",
   atoi(yytext), months[atoi(yytext+3)], atoi(yytext+6)); }
```

Evo jednog izvršavanja ovog programa:

```
$ ./date
Danas je utorak 06/11/2012 godine.
Danas je utorak 6-nov-2012 godine.
^D
$
```

2.3.7 Primer 7

Tekst se sastoji od 0 ili više rečenica. Rečenica započinje rečju koja počinje velikim slovom, zatim mogu da slede i male i velike reči i na kraju stoji tačka.

Skener za ovaj zadatak, znači, treba da prepozna:

– reči koje počinju velikim slovom: prvo jedno veliko slovo a zatim 0 ili više malih slova: `[A-Z][a-z]*`

– reči koje počinju malim slovom: 1 ili više malih slova: `[a-z]+`

U nastavku je data implementacija ovog skenera:

Listing 2.12: text.l

```
/* Skener za tekst */

%option noyywrap yylineno
%{
   char* yylval;
   #define _DOT            1
   #define _CAPITAL_WORD   2
   #define _WORD           3

   static char* tokens[] = { "",
                             "_DOT␣␣␣␣␣␣␣␣␣",
                             "_CAPITAL_WORD",
                             "_WORD␣␣␣␣␣␣␣␣" };
%}

%%

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

"."             { return _DOT; }
[A-Z][a-z]*     { yylval = yytext; return _CAPITAL_WORD; }
[a-z]+          { yylval = yytext; return _WORD; }

.               { printf("\nline␣%d:␣LEXICAL␣ERROR␣on␣char␣%c",
                          yylineno, yytext[0]); }

%%

main() {
   int token = -1;
   while((token = yylex()) != 0) {
      printf("\n%16s␣TOKEN:␣%s", yytext, tokens[token]);
      if(token == _CAPITAL_WORD || token == _WORD) {
         printf("␣value:␣%s", yylval);
      }
   }
}
```

Prvi deo specifikacije sadrži opcije za isključivanje funkcije `yywrap` i za uključivanje brojanja linija `yylineno`. U nastavku su definisani tokeni sa

vrednostima 1, 2 i 3. Dalje su definisani stringovi koji odgovaraju imenima tokena, i služe za štampanje izlaza programa. Vrednosti tokena odgovaraju indeksima njihovih stringova.

Prvo pravilo preskače beline. Drugo pravilo prepoznaje tačku i vraća samo token.

Treće i četvrto pravilo prepoznaju reči i vraćaju i token i string simbola kroz promenljivu `yylval`. Zato je promenljiva `yylval` u prvom delu specifikacije definisana kao `char*` .

Poslednje pravilo služi za prijavljivanje leksičke greške: ako je neki znak dospeo u ovo pravilo, znači da nije odgovarao nijednom prethodnom pravilu, a to znači da njegova pojava nije ni predviđena.

Funkcija `main` prihvata tokene i vrednosti simbola i ispisuje ih.

Jedan primer izvršavanja ovog programa može biti:

```
$ ./a.out
Ovo je tekst. Za skeniranje.

             Ovo TOKEN: _CAPITAL_WORD value: Ovo
              je TOKEN: _WORD         value: je
           tekst TOKEN: _WORD         value: tekst
               . TOKEN: _DOT
              Za TOKEN: _CAPITAL_WORD value: Za
      skeniranje TOKEN: _WORD         value: skeniranje
               . TOKEN: _DOT
^D
$
```

2.4 mikroC skener

MikroC jezik je podskup C programskog jezika. Gramatika koja ga opisuje je data u prilogu.

Skener za mikroC jezik je naveden u nastavku:

Listing 2.13: mcscanner.l

```
/* Leksicki analizator za mikroC
   (ispisuje prepoznate tokene) */

%option noyywrap yylineno
%{
   union {
```

```
      int i;
      char *s;
   } yylval;

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

   enum { ADD, SUB, EQ, NE };
   char *value_strings[] = { "ADD", "SUB", "EQ", "NE" };
%}

%%

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

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

"("                     { return _LPAREN; }
")"                     { return _RPAREN; }
"{"                     { return _LBRACKET; }
"}"                     { return _RBRACKET; }
";"                     { return _SEMICOLON; }
"="                     { return _ASSIGN; }
"+"                     { yylval.i = ADD; return _AROP; }
"-"                     { yylval.i = SUB; return _AROP; }
"=="                    { yylval.i = EQ; return _RELOP; }
"!="                    { yylval.i = NE; return _RELOP; }

[a-zA-Z][a-zA-Z0-9]* { yylval.s = yytext; return _ID; }
[+-]?[0-9]{1,10}      { yylval.s = yytext; return _INT_NUMBER;}

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

%%

int main() {
   int tok;
   while(tok = yylex()) {
```

```
      printf("%s", token_strings[tok]);
      if(tok == _ID || tok == _INT_NUMBER)
         printf(":␣%s\n", yylval.s);
      else
         if(tok == _AROP || tok == _RELOP)
            printf(":␣%s\n", value_strings[yylval.i]);
         else printf("\n");
   }
}
```

Prvi deo specifikacije sadrži enumeraciju sa definicijama tokena i niz imena tokena u obliku stringova koji se koriste prilikom ispisa tokena, kao i enumeraciju sa konstantama koje opisuju razne operatore i, opet, niz imena tih konstanti u obliku stringova.

Prvo pravilo preskače beline. Drugo pravilo prepoznaje celobrojni tip, a sledeća tri ključne reči.

Dalje su navedena pravila koja prepoznaju separatore i operator dodele. Zatim slede pravila za prepoznavanje aritmetičkih i relacionih operatora. Za ove operatore se osim tokena prosleđuje još i odgovarajuća konstanta kao vrednost simbola. Vrednost se prosleđuje kroz promenljivu `yylval` koja je definisana u prvom delu specifikacije kao unija. Sadrži jedan celobrojni tip i jedan pokazivač na karakter. Kako je konstanta celobrojnog tipa, njena vrednost se smešta u polje `i` promenljive `yylval`.

Zatim slede pravila za prepoznavanje identifikatora i konstanti. U oba slučaja se pored tokena, prosleđuje još i string kao vrednost simbola, koji se smešta u polje `s` (tipa `char*`).

Funkcija `main` preuzima od skenera tokene i vrednosti simbola i ispisuje ih.

Ako bismo ovom programu prosledili ulaznu datoteku:

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

izlaz bi bio:

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

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

Glava 3

Sintaksna analiza - parser

Sintaksa jezika opisuje pravila po kojima se kombinuju simboli jezika (npr. u `while` iskazu se prvo navede ključna reč `while`, zatim se u malim zagradama navodi logički izraz, a iza zagrada slede naredbe koje čine telo petlje). 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. sintaksno je ispravan. U suprotnom, sintaksna analiza treba da prijavi sintaksnu grešku i da nastavi analizu. Opisani proces se dešava u delu kompajlera koji se zove parser i naziva se parsiranje. U toku parsiranja gradi se stablo parsiranja.

3.1 Vrste parsera

Generalno, postoje dve vrste parsera, odnosno dve metode parsiranja koje na različit način grade stablo parsiranja:

top-down koje grade stablo parsiranja od korena prema listovima i

bottom-up koje kreću od svojih listova i kreću se prema korenu.

U oba slučaja, ulaz parsera se skenira sa leva na desno, jedan po jedan karakter. Najefikasnije *top-down* i *bottom-up* metode rade samo sa podklasom gramatika, a samo neke od tih podklasa, kao što su LL i LR gramatike, su dovoljno izražajne da opišu većinu sintaksnih konstrukcija u programskim jezicima. Ručno napisani parseri obično rade sa LL gramatikama, dok

se parseri za veću (širu) klasu LR gramatika prave uz pomoć alata za njihovo automatsko generisanje. Tako se parseri, s obzirom na to kojoj vrsti gramatika su namenjeni, mogu podeliti i na:

LL parseri pogodni za programiranje, namenjeni LL gramatikama

LR parseri pogodni za automatsko generisanje alatima, namenjeni LR gramatikama

Prvo slovo "L" ("left"), u oba slučaja, označava da se ulaz čita s leva na desno. Drugo slovo "L" ("left") označava da se primenjuje izvođenje s leva, a drugo slovo "R" ("right") označava da se koristi izvođenje s desna.

Osnovni izlaz iz parsera je informacija da li je ulazni tekst sintaksno ispravan (da ili ne). U praksi, tokom parsiranja, moguće je sprovesti i obrade kao što su sakupljanje informacija o simbolima u tabeli simbola, provera tipova i drugi oblici semantičke analize i generisanje međukoda.

3.2 LR parseri

Problem sa LL parsiranjem je taj da postoji veliki broj gramatika koje se ne mogu automatski pretvoriti u LL gramatike.

LR parseri su vrsta *bottom-up* metoda za parsiranje koje prihvataju mnogo veću klasu gramatika nego LL parsiranje, mada i dalje ne sve gramatike. Osnovna prednost LR parsiranja je da je potrebno manje intervencija da bi se gramatika dovela do prihvatljivog oblika za LR parsiranje nego što je to slučaj sa LL parsiranjem. Osim toga, LL parseri traže od gramatike da bude nedvosmislena, dok LR parseri omogućavaju razrešavanje dvosmislenosti pomoću deklaracije redosleda operatora .

LR parseri su *table-driven bottom-up* parseri i koriste 2 vrste akcija koje se odnose na ulazni tekst i stek:

***shift*:** simbol se pročita iz ulaznog strima i stavi se na stek

***reduce*:** gornjih N elemenata steka čuvaju simbole identične sa N simbola koji se nalaze sa desne strane određenog pravila. Reduce akcijom se ovih N simbola zamenjuje pojmom koji se nalazi sa leve strane tog pravila.

Zbog ovih akcija LR parseri se nazivaju još i *shift-reduce* parseri.

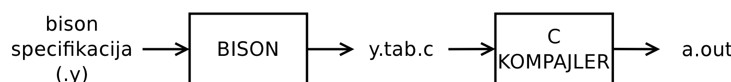
Osim ove 2 akcije postoje još 2 koje se koriste u procesu LR parsiranja:

***accept*:** ova akcija se izvršava kada je pročitan ceo ulazni tekst, a na steku se nalazi samo jedan pojam i to početni pojam gramatike

***error*:** ako ulazni tekst nije u skladu sa gramatikom, parser neće moći da primeni ni *shift* ni *reduce* akciju, pa će se zaustaviti i prijaviti grešku

3.3 Generatori LR parsera

U toku ovog kursa, koristićemo `bison` generator LR parsera. Korišćenje `bison`-a je prikazano na slici:



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` program `y.tab.c`. Kod koji je izgenerisao `bison`, sadrži tabelarnu reprezentaciju dijagrama dobijenih iz opisa pravila. Akcije pridružene pravilima u `.y` datoteci su delovi `C` koda koji se direktno prenose (kopiraju) u `y.tab.c`. Na kraju, `y.tab.c` se prosleđuje `C` kompajleru da bi se kao izlaz dobio program `a.out`. Ovaj program predstavlja parser koji proverava sintaksnu ispravnost ulaznog teksta (u skladu sa pravilima gramatike).

Prilikom izvršavanja, parser će od skenera tražiti naredni token iz ulaznog teksta i proveriti da li je njegova pojava na tom mestu dozvoljena (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 parser prepozna celo pravilo, parser će izvršiti akciju koja je pridružena tom pravilu.

3.4 Format bison specifikacije

`bison` specifikacija se sastoji od tri dela, međusobno razdvojena oznakom `%%` (dva znaka procenat jedan do drugog):

1. definicije
2. pravila i
3. korisnički kod.

3.4.1 Definicije

Prvi deo se obično koristi za uključivanje zaglavlja i definicije promenljivih (pri čemu ovakav kod mora biti okružen specijalnim zagradama `%{` i `%}`). U ovom delu se definišu i tokeni tako što se ime tokena navede iza kljune reči `%token`. Uopšteno, u ovom delu se navode opcije za upravljanje radom generatora. Ovaj deo fajla nije obavezan.

3.4.2 Pravila

Drugi deo specifikacije sadrži gramatiku, odnosno pravila koja opisuju sintaksu jezika. Gramatika svakog jezika obuhvata: simbole, pojmove, pravila i polazni pojam. Na osnovu ovih pravila `bison` generiše kod parsera. Svako pravilo se zadaje u obliku:

```
pravilo [ akcija ]
```

Pravila se pišu na formalan način u BNF obliku, a akcije blokom C (ili C++) koda koji će se izvršiti kada skener prepozna pravilo. Obrazac mora da počne u prvoj koloni, a akcija ne mora da postoji. Ako akcija postoji, njena definicija mora da počne u istom redu u kojem je definisan obrazac. Ovaj deo opisa `bison` programa je obavezan.

Pravila (*production*) određuju dozvoljene načine ređanja/pisanja pojmova i simbola:

```
pojam → pojmovi i/ili simboli
```

Leva strana pravila (pre znaka →) sadrži pojam koji može biti zamenjen sekvencom pojmova i/ili simbola koje sadrži desna strana pravila (iza znaka →). To znači da se leva strana pravila može zameniti desnom stranom pravila, i obrnuto. Jedan od pojmova predstavlja polazni pojam i on se navodi kao prvo pravilo.

Na primer, `IF-ELSE` iskaz u C programskom jeziku ima formu:

```
IF ( Expr ) stmt ELSE stmt
```

Drugim rečima, to je konkatenacija: ključne reči `IF`, otvorene zagrade `(`, izraza `Expr`, zatvorene zagrade `)`, iskaza `stmt`, ključne reči `ELSE`, i na kraju, još jednog iskaza `stmt`.

Sintaksa `IF-ELSE` iskaza napisana BNF notacijom ima izgled:

```
stmt → IF ( Expr ) stmt ELSE stmt
```

Strelica → se čita "može imati oblik". Više pravila koja imaju istu levu stranu, na primer:

```
list → + digit
list → - digit
list → digit
```

mogu se grupisati zajedno, tako što se razdvoje vertikalnom linijom koja ima značenje "ili":

```
list → + digit | - digit | digit
```

Ako se ovo napiše malo drugačije, postaje malo čitljivije:

```
list → + digit
     | - digit
     | digit
```

`Bison` ovu gramatiku prihvata u malo modifikovanom obliku BNF notacije: umesto strelice se piše dvotačka i na kraju grupe pravila se piše znak tačka-zarez:

```
list : + digit
     | - digit
     | digit
     ;
```

3.4.3 Korisnički kod

U ovom opcionom delu korisnik može da definiše proizvoljan blok koda (funkcija `main()`, druge funkcije, globalne promenljive, ...) koji će bez izmena biti prekopiran na kraj generisanog parsera.

3.5 Kako `bison` parsira ulaz

`Bison` preuzima gramatiku iz specifikacije i pravi parser koji prepoznaje "rečenice" odnosno iskaze te gramatike.

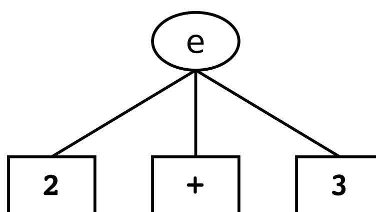
Programi mogu biti sintaksno ispravni ali semantički neispravni (na primer, C program koji `string` dodeljuje `int` promenljivoj). `Bison` rukuje samo sintaksnom a sve ostale validacije su ostavljene korisniku.

Kao što je ranije opisano, gramatika je niz pravila koje parser koristi da prepozna sintaksno ispravan ulaz. Na primer, evo jedne verzije gramatike za jednostavni kalkulator:

```
e : NUMBER "+" NUMBER
  | NUMBER "-" NUMBER
  ;
```

gde pojam `e` opisuje izraz (*expression*). `NUMBER`, `'+'` i `'-'` su simboli. Vertikalna crta (`|`) znači da za isti pojam postoje dve različite mogućnosti, odnosno u ovom slučaju, izraz može biti ili sabiranje ili oduzimanje. Svi pojmovi koji se koriste u gramatici moraju biti definisani, odnosno mora postojati bar jedno pravilo u kom se taj pojam nalazi na levoj strani. Na levoj strani pravila se nikada ne sme naći token (to je greška).

Uobičajeni način da se predstavi isparsiran tekst je stablo. Na primer, ako se parsira ulaz: `2+3` ovom gramatikom, stablo izgleda kao na slici 3.1.



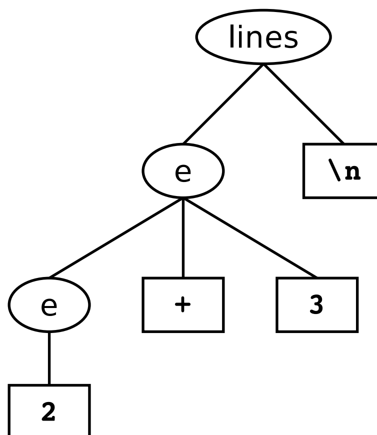
Slika 3.1: Stablo parsiranja za izraz `2+3`

Parser ne kreira automatski ovo stablo kao strukturu podataka, iako je relativno jednostavno da korisnik to uradi.

Pravila mogu da pozivaju, direktno ili indirektno, sama sebe pa se zovu rekurzivna pravila. Time je omogućeno parsiranje proizvoljno dugačkih ulaznih nizova. Ako prethodnu gramatiku malo izmenimo, dobićemo:

```
lines :
      | lines e '\n'
      ;
e     : e "+" NUMBER
      | e "-" NUMBER
      | NUMBER
      ;
```

gde `e` opisuje izraz a `lines` opisuje (ništa ili) redove koji sadrže po jedan izraz (i znak za novi red). `NUMBER`, `'\n'`, `'+'` i `'-'` su simboli, a `e` i `lines` su pojmovi. Ako se parsira ulaz `2+3'\n'` ovom gramatikom, stablo izgleda kao na slici 3.2.

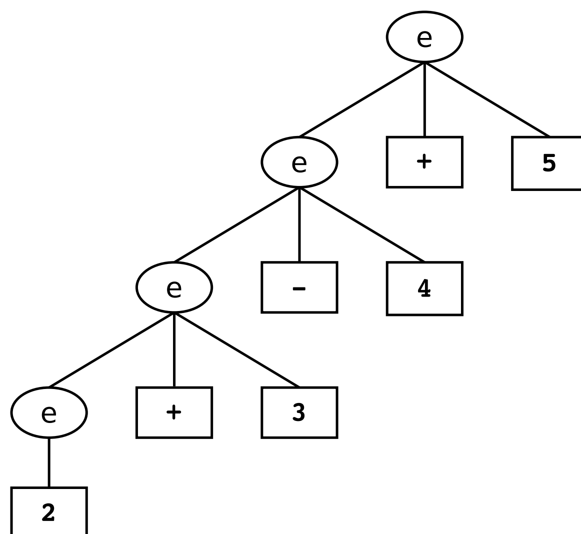


Slika 3.2: Stablo parsiranja za ulaz `2+3\n`

U ovom primeru, `2+3` je izraz, a `e'\n'` je iskaz. Svaka gramatika ima početni pojam, onaj koji mora da bude koren stabla parsiranja. U ovoj gramatici početni pojam je `lines`. Da bi neki pojam bio proglašen početnim, treba da bude naveden kao prvo pravilo u `bison` specifikaciji ili se ime početnog pojma navede iza ključne reči `%start`, u prvom delu `bison` specifikacije, na primer:

```
%start lines
```

U našem primeru pravilo `e` je sada rekurzivno definisano, i to levo rekurzivno. To znači da se ime pravila nalazi prvo navedeno na desnoj strani pravila. Zbog ove osobine, moguće je parsirati izraz `2+3-4+5` ponovljenom primenom pravila `e` (deo stabla parsiranja koji se odnosi na ovaj ulazni izraz prikazano je na slici 3.3).



Slika 3.3: Deo stabla parsiranja za izraz `2+3-4+5`

Evo `flex` i `bison` implementacije ove gramatike: potreban nam je i skener za prepoznavanje simbola i parser za prepoznavanje iskaza.

Listing 3.1: calc1.l

```
/* Jednostavni kalkulator */

%{
   #include "calc1.tab.h"
%}

%%

[ \t]+
[0-9]+     { yylval = atoi(yytext); return NUMBER; }
"+"        { return PLUS; }
"-"        { return MINUS; }
\n         { return NEWLINE; }
.          { printf("Unknown char %c\n", *yytext); }
```

U prvom delu specifikacije potrebno je uključiti datoteku `calc1.tab.h` koju proizvodi `bison`, jer se u njoj nalaze izgenerisani tokeni koji se koriste u skeneru.

U pravilima se prepoznaje broj kao niz cifara, i uz token se prosleđuje konkretna vrednost broja (podrazumevanti tip promenljive `yylval` je `int`). Za ostale simbole (`'+', '-' i '\n'`) se prosleđuje samo token. Za sve ostale karaktere prijavljuje se greška.

Listing 3.2: calc1.y

```
%{
   #include <ctype.h>
   #include <stdio.h>
   int yyparse(void);
   int yylex(void);
%}

%token NUMBER
%token PLUS
%token MINUS
%token NEWLINE

%%

lines
   :
   |  lines NEWLINE
   |  lines e NEWLINE
   ;

e
   :  e PLUS NUMBER
   |  e MINUS NUMBER
   |  NUMBER
   ;

%%

int main() {
   return yyparse();
}

int yyerror(char *s) {
   fprintf(stderr, "%s\n", s);
   return 0;
}
```

U prvom delu su definisana 4 tokena za 4 simbola gramatike. Pravilo `lines`

opisuje prazan red ili red u kojem se nalazi izraz iza kog sledi znak za novi red. Izraz je rekurzivno definisan kao sabiranje ili oduzimanje.

`main()` funkcija jedino poziva parser. Ukoliko dođe do sintaksne greške, pozvaće se funkcija `yyerror()` koja ispisuje poruku o greški.

Generisanje skenera i parsera, i pokretanje programa se odvija ovako:

```
$ bison -d calc1.y
$ flex --noyywrap calc1.l
$ gcc -o calc1 calc1.tab.c lex.yy.c
$ ./calc1
2+3
5-2
^D
$
```

Prvo se pokrene `bison` sa opcijom `-d` (za generisanje `.h` fajla sa **d**efinicijama) koji će napraviti `calc1.tab.c` i `calc1.tab.h`. Zatim se pokrene `flex` koji napravi `lex.yy.c`. Na kraju se pozove `gcc` kompajler da napravi izvršni program `calc1`. Pokretanjem programa vidimo da kalkulator prihvata izraze, ali ne izvršava sabiranje i oduzimanje jer pravilima još nismo pridružili akcije, tj. nismo mu rekli šta da radi kada prepozna izraz i liniju.

3.5.1 Akcije

Hajde da pravilima u prethodnoj `bison` specifikaciji dodamo akcije (3.3):

Listing 3.3: calc2.y

```
%{
   #include <ctype.h>
   #include <stdio.h>
   int yyparse(void);
   int yylex(void);
%}

%token NUMBER
%token PLUS
%token MINUS
%token NEWLINE

%%

lines
   :
   |  lines NEWLINE
```

```
   |  lines e NEWLINE    { printf("%d\n", $2); }
   ;

e
   :  e PLUS NUMBER      { $$ = $1 + $3; }
   |  e MINUS NUMBER     { $$ = $1 - $3; }
   |  NUMBER             { $$ = $1; }
   ;

%%

int main() {
   return yyparse();
}

int yyerror(char *s) {
   fprintf(stderr, "%s\n", s);
   return 0;
}
```

U akcijama se koriste meta-promenljive čija imena počinju znakom `'$'` a iza toga sledi broj. Ovaj broj označava redni broj simbola ili pojma na desnoj strani pravila. Tako se meta-promenljiva `$1` odnosi na vrednost pojma/simbola koji se nalazi prvi naveden na desnoj strani pravila. Vrednost simbola je u parser stigla preko promenljive `yylval` a poslao ju je skener. Vrednost pojmova se definiše u parseru, u pravilu za dotični pojam. Meta-promenljiva `$$` se odnosi na pojam sa leve strane pravila, odnosno sadrži njegovu vrednost. To znači da kada želimo da definišemo vrednost nekog pojma, u njegovom pravilu ćemo imati akciju `{ $$ = ... ; }`.

Krenućemo od poslednjeg pravila za izraz `e : NUMBER` kojem smo dodali akciju `{ $$ = $1; }`. Ova akcija kaže da vrednost pojma `e` dobija istu onu vrednost koju ima simbol `NUMBER`, a to je konkretna vrednost prepoznatog broja (jer smo u skeneru uz token `NUMBER` preko `yylval` prosledili i vrednost broja).

U pravilu za oduzimanje dodali smo akciju `{ $$ = $1 - $3; }`, gde ćemo fizički oduzeti 2 broja. `$1` sadrži vrednost prvog broja (to je vrednost pojma `e`) a `$3` sadrži vrednost drugog broja (koji se nalazi na trećoj poziciji u pravilu pa se njegovoj vrednosti pristupa preko meta-promenljive `$3`). Vrednosti pojma `e` (kojoj se pristupa preko meta-promenljive `$$`) se dodeljuje vrednost oduzimanja. Slično se dešava i u pravilu za sabiranje.

U pravilu za `lines` dodali smo akciju `{ printf("%d\n", $2); }` koja nakon prepoznavanja jedne linije sa izrazom ispisuje vrednost izraza. Ovu akciju ne

treba dodati u ostala dva pravila za `lines` jer su to pravila za prepoznavanje praznih linija, bez izraza, pa nema ni izračunavanja ni ispisivanja.

3.5.2 *Shift-reduce* parsiranje

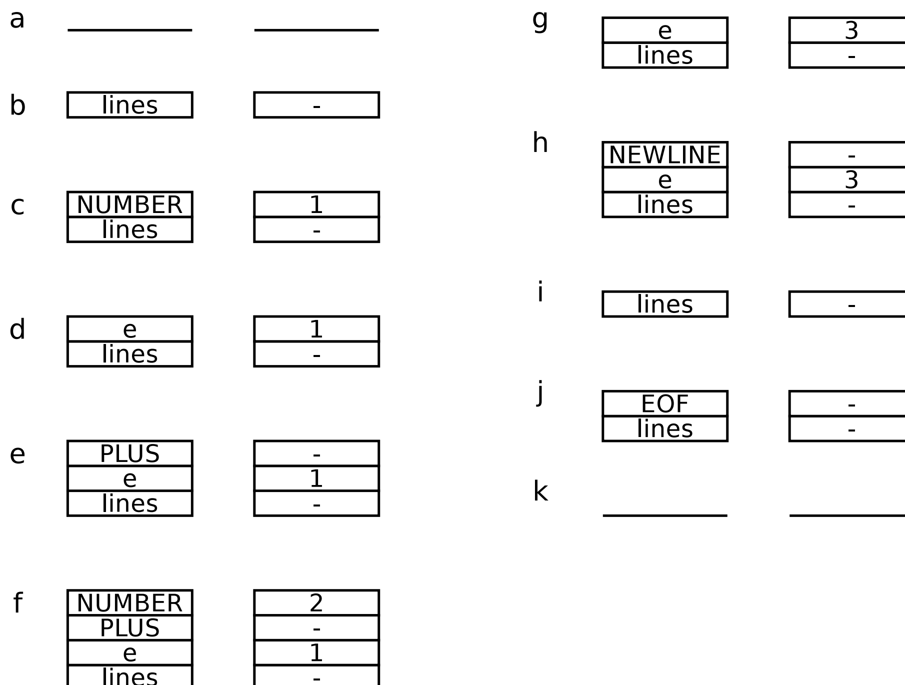
`Bison` radi tako što gleda da li postoji pravilo (desna strana pravila) koje odgovara tokenima koje je do sada primio. Kada `bison` napravi parser, on kreira skup stanja, od kojih svako odražava poziciju u pravilu koje se parsira. Dok parser čita tokene, svaki put kada pročita token koji ne kompletira pravilo, on ga stavi na interni stek i pređe u novo stanje. Ova akcija se zove *shift*. Kada parser pronađe sve simbole koji čine desnu stranu pravila, on ih sve skine sa steka, i stavi pojam sa leve strane pravila na stek, i pređe u novo stanje (zameni desnu stranu pravila na steku levom). Ova akcija se zove *reduce*, jer smanjuje broj elemenata na steku. Kad god `bison` prepozna pravilo (*reduce*) on izvrši akciju koja je pridružena tom pravilu.

Da vidimo kako će parser parsirati ulaz `1+2`\n. Pre preuzimanja prvog znaka, dešava se redukcija po prvom pravilu, koje ništa pretvara u `lines` (3.4.a, 3.4.b). Pošto ne može da izvrši ni jednu više redukciju, poziva skener, i dobija od njega prvi simbol iz ulaznog teksta, a to je `NUMBER`, sa vrednošću `1`. Simbol `NUMBER` šiftuje na stek stanja, a na stek vrednosti stavlja vrednost `1` (stek sada izgleda kao 3.4.c).

Zatim proverava da li neka sekvenca na vrhu steka odgovara desnoj strani nekog pravila. Konstatuje da `NUMBER` čini desnu stranu pravila za `e`, pa izvršava redukciju. Sa steka stanja skida `NUMBER` i menja ga pojmom `e`, a sa steka vrednosti skida `1` i stavlja vrednost pojma `e`. Vrednost pojma se računa u akciji koja je pridružena tom pravilu i koja se izvršava u toku redukcije. U toj akciji piše da vrednost pojma `e` (`$$`) dobija vrednost simbola `NUMBER` (`$1`) a to je vrednost `1` (vidi 3.4.d).

Pošto ne može da izvrši više nijednu redukciju, parser se ponovo obraća skeneru za token. Dobija token `PLUS` kojeg šiftuje na stek stanja (vidi 3.4.e). Uz token `PLUS` iz skenera nije poslata nikakva vrednost. U ovom trenutku elementi na vrhu steka ne formiraju nijedno pravilo, pa parser poziva skener i dobija token `NUMBER` i vrednost `2` (vidi 3.4.f).

Sada poslednja 3 elementa na vrhu steka čine desnu stranu pravila `e` za sabiranje izraza. Parser može da uradi redukciju po tom pravilu, što znači da skida 3 elementa sa steka stanja i stavlja pojam `e`, a sa steka vrednosti skida 3 elementa i stavlja novu vrednost `3`. Ovu novu vrednost pojma `e` je dobio izvršavanjem akcije uz ovo pravilo, koja opisuje da se vrednost pojma



Slika 3.4: Primer rada parsera za ulaz `1+2\n`

`e` dobija kao zbir prvog operanda (`$1`) - u našem slučaju vrednost `1`, i drugog operanda (`$3`) - u našem slučaju vrednost `2`. Vrednosti simbola `NUMBER` se pristupa preko meta-promenljive `$3` jer se simbol nalazi na trećem mestu na desnoj strani pravila, i istovremeno je treći element koji je skinut sa steka (vidi 3.4.g).

Parser, dalje, šiftuje simbol `NEWLINE` (vidi 3.4.h) i sada je u mogućnosti da uradi redukciju po pravilu `lines : lines e NEWLINE`. U procesu redukcije skida po 3 elementa sa svakog steka i ostavlja jedan pojam `lines` na steku stanja (vidi 3.4.i).

Nakon toga, šiftuje `EOF` znak jer je stigao do kraja datoteke, i u prilici je da uradi *accept* akciju koja je moguća kada se na steku nalazi samo polazni pojam a iza njega sledi `EOF` znak (3.4.j, 3.4.k). U ovom slučaju parser objavljuje uspešno parsiranje.

3.5.3 Oporavak od greške

Kada `bison`-ov parser detektuje grešku, ispiše string "`syntax error`" i završi parsiranje (završi program). Ukoliko korisniku to nije dovoljno, `bison` nudi mogućnost oporavka od greške pomoću tokena `error`.

Specijalni pseudo-token `error` označava mesto oporavka od greške. Kada parser detektuje grešku, počinje da odbacuje simbole sa steka sve dok ne dostigne mesto gde će `error` token biti validan. Zatim odbacuje ulazne tokene sve dok ne pronađe jedan koji može šiftovati u trenutnom stanju i tada nastavlja parsiranje od te tačke. Ako parsiranje ponovo detektuje grešku, odbacuje se još simbola sa steka i ulaznih tokena sve dok ne postane moguć nastavak parsiranja ili dok se stek ne isprazni. Da bi se izbeglo puno neadekvatnih poruka o greškama, parser, posle prve greške, prestane da prijavljuje poruke sve dok ne uspe da šiftuje tri tokena jedan za drugim.

U nastavku je dat primer kalkulatora koji detektuje grešku unutar izraza, oporavlja se od greške i nastavlja parsiranje (3.4).

Listing 3.4: calc3.y

```
%{
   #include <ctype.h>
   #include <stdio.h>
   int yyparse(void);
   int yylex(void);
%}

%token NUMBER
%token PLUS
%token MINUS
%token NEWLINE

%%

lines
   :  /* prazna linija */
   |  lines NEWLINE
   |  lines e NEWLINE       { printf("%d\n", $2); }
   |  lines error NEWLINE   { yyerror("reenter last line:\n");
                              yyerrok; }
   ;

e
   :  e PLUS NUMBER         { $$ = $1 + $3; }
   |  e MINUS NUMBER        { $$ = $1 - $3; }
   |  NUMBER                { $$ = $1; }
   ;
```

```
%%

int main() {
   return yyparse();
}

int yyerror(char *s) {
   fprintf(stderr, "%s\n", s);
   return 0;
}
```

Dodali smo još jedno pravilo sa tokenom `error`. Pozicija ovog tokena opisuje pojavu bilo kakve sintaksne greške unutar linije, pre znaka za novi red. Ukoliko se na ulazu pojavi greška, parser će uraditi redukciju po ovom pravilu, i u našem slučaju, ispisati poruku korisniku da ponovo unese izraz. Nakon unosa, parser nastavlja sa parsiranjem.

Makro `yyerrok` u akciji kaže parseru da je oporavak završen i da se naredne poruke o greškama mogu prijavljivati.

U cilju detekcije što više grešaka moguće je dodati puno pravila koja opisuju greške, ali u praksi retko postoji više od nakoliko pravila sa greškama.

3.5.4 Šta `bison`-ov LALR(1) parser ne može da parsira

LALR(1) je LR parser koji ima 1 *lookahead* token - vidi jedan token unapred.

Iako je LALR parsiranje vrlo snažno, postoje gramatike koje ne može da isprocesira. Ne može da rukuje dvosmislenim gramatikama i gramatikama kojima je potrebno više od jednog *lookahead* tokena da kaže da li je video pravilo.

Dvosmislene su one gramatike kod kojih je moguće za isti ulazni tekst napraviti više od jednog stabla parsiranja. Kada `bison`-ov parser detektuje ovakvu situaciju u gramatici, on prijavi konflikt.

3.5.5 Konflikti i njihovo razrešavanje

Za sve osim najjednostavnijih gramatika, korisnik generatora LR parsera treba da očekuje prijavu konflikata kada prvi put propusti gramatiku kroz generator. Ovi konflikti mogu biti uzrok dvosmislenosti gramatike ili ograničenja metode parsiranja. U oba slučaja, konflikti se mogu eliminisati prepravljanjem gramatike ili deklarisanjem prednosti operatora. `Bison` pruža mogućnost

da se prednost operatora opiše odvojeno od pravila, što čini gramatiku i parser manjim i jednostavnijim za održavanje.

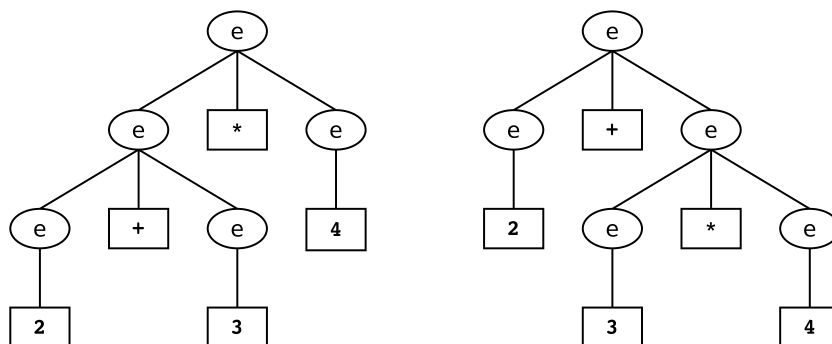
Većina generatora može pružiti informacije pomoću kojih se može locirati mesto u gramatici na kom se nalazi problem. Kada generator prijavi konflikt, on će reći u kom stanju (u tabeli prelaza parsera) se to desilo. Ovo stanje može biti ispisano u (jedva) čitljivom obliku kao skup NFA stanja. Pokretanjem `bison`-a sa opcijom (svičem) `-v` (*verbose*) generiše se datoteka `name.output` u kojoj se nalazi spisak konflikata.

3.5.5.1 Primer konflikta

Ako prethodnu gramatiku za izraze proširimo operacijama množenja, deljenja i unarnim minusom

```
e : e "+" e
  | e "-" e
  | e "*" e
  | e "/" e
  | "-" e
  | NUMBER
  ;
```

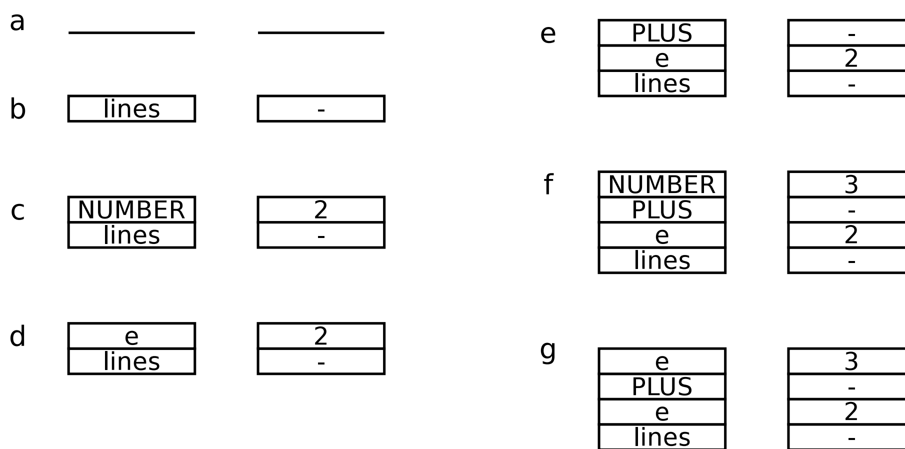
dobićemo dvosmislenu gramatiku. Na primer, ulaz `2+3*4` može značiti `(2+3)*4` ili `2+(3*4)`, a ulaz `3-4-5-6` može značiti `3-(4-(5-6))` or `(3-4)-(5-6)` i još mnogo sličnih mogućnosti. Slika 3.5 pokazuje dva moguća stabla parsiranja za primer `2+3*4`.



Slika 3.5: Dva moguća stabla parsiranja za primer `2+3*4`

3.5.5.2 Rešenje konflikta

Ako ovakvu gramatiku prosledimo `bison`-u, on će nam saopštiti da postoji više *shift/reduce* konflikata. To su stanja gde on može da iradi i *shift* i *reduce* akciju (da šiftuje token ili da uradi redukciju po nekom pravilu) a ne zna za šta da se odluči. Kada parser parsira prethodni primer `2+3*4` on prolazi kroz sledeće korake:



Slika 3.6: Primer konflikta za izraz `2+3*4`

U ovom trenutku, parser vidi `*` i može da uradi redukciju poslednja 3 elementa `(2+3)` po pravilu `e : e '+' e` a može da uradi i šiftovanje operatora `*`, očekujući da će moći kasnije da uradi redukciju po pravilu `e : e '*' e`.

Problem je nastao zbog toga što `bison`-u nismo ništa rekli o prioritetu i asocijativnosti operatora. Prioritet upravlja redosledom izvršavanja izraza. Množenje i deljenje imaju prednost nad sabiranjem i oduzimanjem, tako da izraz `a+b*c` ima značenje `a+(b*c)`, a `d/e-f` zmači `(d/e)-f`. U bilo kojoj gramatici izraza, operatori su grupisani u nivoe prioriteta od najnižeg ka najvišem. Postoje dva načna da se definišu prioritet i asocijativnost u gramatici: implicitno i eksplicitno. Implicitna definicija prioriteta podrazumeva definiciju zasebnih pojmova za svaki nivo prioriteta. Naša gramatika bi u ovom slučaju izgledala:

```
mul_exp : num_exp "*" num_exp
        | num_exp "/" num_exp
        ;
```

```
num_exp : exp "+" exp
        | exp "-" exp
        ;
exp     : NUMBER
        ;
```

Međutim, `bison` pruža mogućnost i eksplicitnog definisanja prioriteta operatora. Dodavanjem ovih linija u prvi deo `bison` specifikacije, kažemo parseru kako da razreši konflikte:

```
%left '+' '-'
%left '*' '/'
%right UMINUS
%%
exp : NUMBER
    | exp "+" exp
    | exp "-" exp
    | exp "*" exp
    | exp "/" exp
    | "-" exp %prec UMINUS
    ;
```

Svaka od ovih deklaracija definiše nivo prioriteta a redosled navođenja `%left`, `%right`, and `%nonassoc` declaracija definiše redosled prioriteta od najnižeg ka najvišem. One kažu `bison`-u da su `+` i `-` levo asocijativni i sa najnižim prioritetom, da su `*` i `/` levo asocijativni i sa višim prioritetom i da je `UNIMUS`, pseudotoken za unarni minus, desno asocijativan i ima najviši prioritet.

`Bison` dodeljuje svakom pravilu prioritet krajnje desnog tokena na desnoj strani pravila. Ako taj token nema pridružen prioritet, ni pravilo nema svoj prioritet. Kada bison naiđe na *shift/reduce* konflikt, konsultuje tabelu prioriteta i, ako sva pravila koja učestvuju u konfliku imaju dodeljen prioritet, koristi taj prioritet za razrešavanje konflikta.

U našoj gramatici, svi konflikti se pojavljuju u pravilima oblika `exp OP exp`, tako da definisanje prioriteta ova 4 operatora omogućuje razrešavanje svih konflikata. Ovaj parser koji koristi eksplicitne definicije prioritera operatora je malo manji i brži od onog koji sadrži dodatna pravila sa implicitnom definicijom prioriteta, jer ima manje pravila za redukciju.

Pravilo za negaciju sadrži `%prec UMINUS`. Jedini operator u ovom pravilu je - (minus), koji ima nizak prioritet. Međutim kada ga koristimo kao unarni

minus, želimo da ima veći prioritet od množenja i deljenja. Oznaka `%prec` kaže `bison`-u da korisiti prioritet `UMINUS` za ovo pravilo.

U nastavku sledi kompletan primer kalkulatora implementiran sa oporavkom od greške i prioritetom operatora.

Listing 3.5: calc4.l

```
/* Kompletan kalkulator */

%{
   #include "calc4.tab.h"
%}

%%

[ ]+

[0-9]+         { yylval = atoi(yytext); return NUMBER; }
"+"            { return PLUS; }
"-"            { return MINUS; }
"*"            { return MULTIPLY; }
"/"            { return DIVIDE; }
\n             { return NEWLINE; }
.              { printf("Unknown char %c\n", *yytext); }
```

Listing 3.6: calc4.y

```
%{
   #include <ctype.h>
   #include <stdio.h>
   int yyparse(void);
   int yylex(void);
%}

%token NUMBER
%token PLUS
%token MINUS
%token MULTIPLY
%token DIVIDE
%token NEWLINE

%left  PLUS MINUS
%left  MULTIPLY DIVIDE
%right UMINUS

%%

lines
   :  /* prazna linija */
```

```
   |  lines NEWLINE
   |  lines e NEWLINE       { printf("%d\n", $2); }
   |  lines error NEWLINE   { yyerror("reenter␣last␣line:\n");
                              yyerrok; }
   ;

e
   :  e PLUS e              { $$ = $1 + $3; }
   |  e MINUS e             { $$ = $1 - $3; }
   |  e MULTIPLY e          { $$ = $1 * $3; }
   |  e DIVIDE e            { $$ = $1 / $3; }
   |  MINUS e %prec UMINUS  { $$ = -$2; }
   |  NUMBER                { $$ = $1; }
   ;

%%

int main() {
   return yyparse();
}

int yyerror(char *s) {
   fprintf(stderr, "%s\n", s);
   return 0;
}
```

3.5.5.3 Kada ne treba koristiti prioritet operatora

Prioriteti operatora se mogu koristiti za razrešavanje konflikata u gramatici, ali to obično nije dobra ideja. U gramatikama sa izrazima, lako je pronaći i shvatiti konflikte, a efekat primene prioriteta je jasan. U drugim situacijama, pravila prioriteta će rešiti konflikte, ali može biti izuzetno teško razumeti kakav uticaj će imati na gramatiku.

Prioritet operatora treba koristiti u dve situacije: u gramatikama koje opisuju izraze, i u slučaju IF-ELSE konflikta koji će uskoro biti objašnjen. Inače, treba popraviti gramatiku tako da konflikt nestane. Postojanje konflikta znači da bison ne može da napravi parser za gramatiku jer je ona dvosmislena. To znači da postoji više mogućih stabala parsiranja i da je parser odabrao jedno od njih. Osim u prethodna dva slučaja, ovo obično pokazuje da postoji problem u definiciji gramatike.

3.5.6 Leva i desna rekurzija

Kada se piše rekurzivno pravilo, rekurzivna referenca se može staviti na levi kraj ili na desni kraj pravila, na primer:

```
exp_list : exp_list ',' exp ;  /* leva rekurzija */
exp_list : exp ',' exp_list ;  /* desna rekurzija */
```

U većini slučajeva, gramatika se može pisati na oba načina. `Bison` rukuje levom rekurzijom mnogo efikasnije nego desnom. To je zato što na svom steku čuva sve simbole koji su dotad viđeni za sva delimično isparsirana pravila. Ako se koristi desno rekurzivna verzija pravila `exp_list` i na ulazu se pojavi lista od 10 izraza, kad se bude pročitao 10-ti izraz, na steku će biti već 20 elemenata: izraz i zarez za svih 10 izraza. Kada se lista završi, svi ugnježdeni `exp_list` pojmovi će biti redukovani, s desna na levo. S druge strane, ako se koristi levo rekurzivna varijanta pravila `exp_list`, pravilo `exp_list` će biti redukovano posle svakog `exp`, tako da lista nikad neće imati više od tri elementa na steku.

3.6 Primeri

3.6.1 Primer 1

Tekst se sastoji od 0 ili više rečenica. Rečenica započinje rečju koja počinje velikim slovom, zatim mogu da slede i male i velike reči i na kraju stoji tačka. Potrebno je izračunati broj reči i rečenica koje se pojave u ulaznom tekstu i ispisati sve reči.

Tekst gramatika u BNF obliku izgleda ovako:

```
text     :
         | text sentence
         ;
sentence : CAPITAL_WORD words DOT
         ;
words    :
         | words WORD
         | words CAPITAL_WORD
         ;
```

Skener smo već imali prilike da vidimo u prethodnom delu zbirke (2.12):

Listing 3.7: count2.l

```
/* parser za tekst koji broji reci i recenice */

%option noyywrap yylineno
%{
   #define YYSTYPE char*
   #include "count2.tab.h"
%}

%%

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

"."          { return _DOT; }
[A-Z][a-z]* { yylval = strdup(yytext); return _CAPITAL_WORD;}
[a-z]+       { yylval = strdup(yytext); return _WORD; }

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

Parser treba da prebroji reči i rečenice, pa nam zato trebaju 2 brojačke promenljive: `word_counter` i `sentence_counter`. Brojač rečenica se inkrementira svaki put kada se prepozna cela rečenica, a to je u akciji koja sledi iza pravila `sentence`. Brojač reči se inkrementira svaki put kada parser dobije od skenera token za neku reč, a to je na 3 mesta: kada dobije prvu reč rečenice `_CAPITAL_WORD`, kada dobije `_WORD` u sredini rečenice i kada dobije `_CAPITAL_WORD` u sredini rečenice.

Listing 3.8: count2.y

```
/* Parser za tekst koji broji reci i recenice */

%{
   #include <stdio.h>
   #define YYSTYPE char*
   int yylex(void);
   int yyparse(void);

   int word_counter = 0;
   int sentence_counter = 0;
   extern int yylineno;
%}

%token   _DOT
%token   _CAPITAL_WORD
%token   _WORD
```

```
%%

text
   :  /* empty text */
   |  text sentence
   ;

sentence
   :  _CAPITAL_WORD words _DOT
          {  word_counter++;
             sentence_counter++;
             printf("%s\n", $1);
             free($1); }
   ;

words
   :  /* empty */
   |  words _WORD
          {  word_counter++;
             printf("%s\n", $2);
             free($2); }

   |  words _CAPITAL_WORD
          {  word_counter++;
             printf("%s\n", $2);
             free($2); }
   ;

%%

main() {
   yyparse();
   printf("Total words: %d.\n", word_counter);
   printf("Total sentences: %d.\n", sentence_counter);
}

yyerror(char *s) {
   fprintf(stderr, "line %d: SYNTAX ERROR %s\n", yylineno, s);
}
```

Ispis reči se opet vrši na prethodno opisana 3 mesta, svaki put kada stigne neka reč. Nakon svakog ispisa reči sledi oslobađanje memorije koju je u skeneru zauzela funkcija `strdup`. Za tokene `_WORD` i `_CAPITAL_WORD` u skeneru smo pored tokena prosledili i vrednosti simbola preko promenljive `yylval`. Te vrednosti smo definisali kao pokazivač na string koji vraća funkcija `strdup`. Ova funkcija alocira memoriju za string i u njega kopira string na koji pokazuje `yytext` promenljiva. Zato je potrebno da, kada nam ti stringovi

više ne trebaju, u parseru uradimo dealokaciju dotične memorije.

Na kraju programa, iz `main` funkcije se ispišu sadržaji brojačkih promenljivih.

Kada se pokrene ovaj program izlaz će izgledati ovako:

```
$ ./count2
Danas je predivan dan.
je
predivan
dan
Danas
^D
Total words: 4.
Total sentences: 1.
$
```

Osim prve reči, sve reči će ispisati u redosledu u kom su se pojavile u ulaznom tekstu, a prvu reč će ispisati na kraju. Ovo se dešava zato što se kod koji vrši ispis prve reči u rečenici nalazi u akciji koja se izvršava tek kada se završi parsiranje cele rečenice (pa se i ispis prve reči vrši kada je prepoznata cela rečenica).

U tekstu zadatka se ne traži da se reči ispišu u redosledu u kom su se pojavile u ulaznom tekstu, pa ovo rešenje možemo smatrati zadovoljavajućim. Ali, hajde da u sledećem primeru rešimo i redosled ispisivanja reči.

3.6.2 Primer 2

Skener je isti kao u prethodnom zadatku i nećemo ga menjati.

Jedina izmena je u pravilu `sentence`. Dodali smo akciju odmah nakon preuzimanja tokena `_CAPITAL_WORD` i njoj smo dodali ispisivanje reči. Znači, u ovom slučaju, prva reč u rečenici će biti ispisana čim dospe u parser, a ne, kao u prethodnom primeru, kada se prepozna cela rečenica.

Listing 3.9: count3.y

```
/* Parser za tekst koji broji reci i recenice */

%{
   #include <stdio.h>
   #define YYSTYPE char*
   int yylex(void);
   int yyparse(void);

   int word_counter = 0;
```

```
   int sentence_counter = 0;
   extern int yylineno;
%}

%token   _DOT
%token   _CAPITAL_WORD
%token   _WORD

%%

text
   :  /* empty text */
   |  text sentence
   ;

sentence
   :  _CAPITAL_WORD
         {  word_counter++;
            printf("%s\n", $1);
            free($1); }
      words _DOT
         {  sentence_counter++;  }
   ;

words
   :  /* empty */
   |  words _WORD
         {  word_counter++;
            printf("%s\n", $2);
            free($2); }

   |  words _CAPITAL_WORD
         {  word_counter++;
            printf("%s\n", $2);
            free($2); }
   ;

%%

main() {
   yyparse();
   printf("Total␣words:␣%d.\n", word_counter);
   printf("Total␣sentences:␣%d.\n", sentence_counter);
}

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

Kada se pokrene ovaj program izlaz će izgledati ovako:

```
$ ./count3
Danas je lep dan.
Danas
je
lep
dan
^D
Total words: 4.
Total sentences: 1.
$
```

3.6.3 Primer 3

Proširiti prethodnu tekst gramatiku tako da omogući da se jedna ili više reči piše u malim zagradama. Ispred prve reči ne može da se pojavi zagrada. Prazne zagrade su dozvoljene. Zagrade moraju biti u paru. Mogu da postoje ugnježdene zagrade (zagrade u zagradama).

Listing 3.10: count4.l

```
/* parser za tekst koji broji reci i recenice */

%option noyywrap yylineno
%{
   #define YYSTYPE char*
   #include "count4.tab.h"
%}

%%

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

"."           { return _DOT; }
"("           { return _LPAREN; }
")"           { return _RPAREN; }
[A-Z][a-z]*   { yylval = strdup(yytext); return _CAPITAL_WORD;}
[a-z]+        { yylval = strdup(yytext); return _WORD; }

.             { printf("\nline %d: LEXICAL ERROR on char %c",
                        yylineno, *yytext); }
```

U skener su dodata pravila za prepoznavanje otvorene zatvorene male zagrade.

Listing 3.11: count4.y

```
/* Parser za tekst koji broji reci i recenice */

%{
   #include <stdio.h>
   #define YYSTYPE char*
   int yylex(void);
   int yyparse(void);

   int word_counter = 0;
   int sentence_counter = 0;
   extern int yylineno;
%}

%token   _DOT
%token   _CAPITAL_WORD
%token   _WORD
%token   _LPAREN
%token   _RPAREN

%%

text
   :  /* empty text */
   |  text sentence
   ;

sentence
   :  _CAPITAL_WORD
         {  word_counter++;
            printf("%s\n", $1);
            free($1); }
      words _DOT
         {  sentence_counter++;  }
   ;

words
   :  /* empty */
   |  words _WORD
         {  word_counter++;
            printf("%s\n", $2);
            free($2); }

   |  words _CAPITAL_WORD
         {  word_counter++;
            printf("%s\n", $2);
            free($2); }

   |  words _LPAREN words _RPAREN
```

```
    ;

%%

main() {
    yyparse();
    printf("Total␣words:␣%d.\n", word_counter);
    printf("Total␣sentences:␣%d.\n", sentence_counter);
}

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

Kada se pokrene ovaj program izlaz će izgledati kao na sledećem listingu:

```
$ ./count4
Danas (utorak) je lep dan (nije hladno (i vedro je)) ().
Danas
utorak
je
lep
dan
nije
hladno
i
vedro
je

Nezavrsene (zagrade u tekstu.
Nezavrsene
zagrade
u
tekstu
line 2: SYNTAX ERROR syntax error
Total words: 14.
Total sentences: 1.
$
```

Prva rečenica je sintaksno ispravna. Testirali smo jednu reč u zagradama, više reči u zagradama, ugnježdene zagrade, prazne zagrade. Druga rečenica je sintaksno neispravna, jer ne sadrži zatvorenu zagradu, i zato je parser ispisao poruku da je detektovao sintaksnu grešku, i završio svoju aktivnost.

3.6.4 Primer 4

Proširiti osnovnu tekst gramatiku datumima u obliku `dd/mm/yyyy`. Napraviti parser koji procesira ulazni tekst i menja formu datuma u `dd-mmm-yyyy`. Ostatak teksta se ne menja (ispisuje se na izlaz u neizmenjenom obliku).

Listing 3.12: date.l

```
%option noyywrap yylineno
%{
   #include "date.tab.h"
%}

%%

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

"/"            { return _SEPARATOR; }
[0-9]{2}       { yylval.i = atoi(yytext); return _2D;}
[0-9]{4}       { yylval.i = atoi(yytext); return _4D; }

"."            { return _DOT; }
[A-Z][a-z]*    { yylval.s = yytext; return _CAPITAL_WORD; }
[a-z]+         { yylval.s = yytext; return _WORD; }
```

U skeneru se kao deo datuma prepoznaju simboli od 2 cifre i od 4 cifre. Za njih su vezani tokeni `_2D` i `_4D` a vrednosti ovih simbola su konkretne vrednosti brojeva, koje se smeštaju u polje `i` unije `yylval`. Kao separator ovih simbola definisan je znak "/" i za njega je vezan token `_SEPARATOR`. Ostali simboli su, kao u prethodnim primerima, simboli tekst gramatike, jedina razlika je što se ovde stringovi reči prenose preko polja `s` unije `yylval`.

Kada želimo da promenljiva `yylval` prenosi različite tipove vrednosti (više od jednog tipa podatka, na primer, i `int` i `char *`), onda je treba definisati kao uniju. `Bison` u ovu svrhu nudi ključnu reč `%union`. Unija u ovom parseru treba da ima jedan `integer` (jer se uz tokene `_2D` i `_4D` šalju konkretni brojevi) i jedan pokazivač na `string` (jer se uz tokene `_CAPITAL_WORD` i `_WORD` šalju stringovi).

Kada se promenljiva `yylval` definiše kao unija, za svaki token koji ima vrednosti, se mora reći kog tipa je ta vrednost. Zato uz definiciju tokena `_2D` i `_4D` stoji oznaka `<i>`, a uz definiciju tokena `_CAPITAL_WORD` i `_WORD` stoji oznaka `<s>`.

Listing 3.13: date.y

```
/* Parser koji pretvara datume iz dd/mm/yyyy u dd-mmm-yyyy */
```

```
%{
   #include <stdio.h>
   int yylex(void);
   int yyparse(void);
   extern int yylineno;
   char *months[] = { "", "jan", "feb", "mar", "apr", "maj",
      "jun", "jul", "avg", "sep", "okt", "nov", "dec" };
%}

%union {
   int i;
   char *s;
}

%token   _SEPARATOR
%token   <i> _2D
%token   <i> _4D
%token   _DOT
%token   <s> _CAPITAL_WORD
%token   <s> _WORD

%%

text
   :  /* empty text */
   |  text sentence
   ;

sentence
   :  _CAPITAL_WORD
         {  printf("%s", $1); }
      words _DOT
         {  printf(".␣"); }
   ;

words
   :  /* empty */

   |  words _WORD
         {  printf("␣%s", $2); }

   |  words _CAPITAL_WORD
         {  printf("␣%s", $2); }

   |  words date
   ;

date
```

```
    :   _2D
          {  printf(" %d-", $1); }
        _SEPARATOR _2D
          {  printf("%s-", months[$4]); }
        _SEPARATOR _4D
          {  printf("%d", $7); }
    ;

%%

int main() {
   yyparse();
   printf("\n");
}

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

Parser je proširen u pravilu `words`, tako da prima i datume, bilo gde u sredini rečenice. Čim se prepoznaju dve cifre koje označavaju dan, izvršava se akcija koja ispisuje taj broj na ekran, a iza njega novi separator. Zatim, kada se prepoznaju sledeće dve cifre koje označavaju mesec, pomoću niza stringova `months`, definisanog u prvom delu specifikacije, se ispisuje tekstualna forma meseca i novi separator. Kada se prepoznaju 4 cifre za godinu, ispišu se na ekran. Sve reči (i tačka) koje se prepoznaju u tekstu se neizmenjene ispišu na izlaz.

3.6.5 Primer 5

Napraviti program koji analizira ulazni fajl sa klimatskim podacima za jedan mesec i ispisuje koliko je bilo dana sa temperaturom iznad 12 stepeni Celzijusa. Fajl se sastoji od više slogova, a slog ima sledeću strukturu:

```
temperatura:  13
pritisak:     1004.9
pravac vetra: E
brzina vetra: 3
vlaznost:     67
```

Listing 3.14: meteo.l

```
%option noyywrap yylineno
```

```
%{
   #include "meteo.tab.h"
%}

%%

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

"temperatura"     { return _TEMPERATURA; }
"pritisak"        { return _PRITISAK; }
"pravac vetra"    { return _PRAVAC_VETRA; }
"brzina vetra"    { return _BRZINA_VETRA; }
"vlaznost"        { return _VLAGA; }
":"               { return _DVOTACKA; }
[0-9]+            { yylval.i = atoi(yytext); return _INT;}
[0-9]+\.[0-9]*    { yylval.f = atoi(yytext); return _FLOAT; }
E|NE|SE|W|NW|SW   { return _VETAR; }
```

Skener prepoznaje sve ključne reči iz sloga i dvotačku i za njih šalje samo token. Prilikom prepoznavanja celog i razlomljenog broja šalje odgovarajući token (`_INT` ili `_FLOAT`) i vrednost broja. Unija sadrži tipove `int` i `float` da bi mogle da se prenesu konkretne vrednosti ovih brojeva. Zato je tokenu `_INT` dodeljen tip `<i>`, a tokenu `_FLOAT` tip `<f>`. Token `_VETAR` se šalje kada skener prepozna neku kombinaciju znakova koja označava smer vetra (`E`, `NE`, `SE`, `W`, `NW`, `SW`).

Primenjena gramatika opisuje da se jedan fajl sastoji od jednog ili više slogova, a da jedan slog redom sadrži: opis temperature, pritiska, pravca vetra, brzine vetra i vlažnosti. Deo sloga koji opisuje temperaturu se sastoji prvo od ključne reči "`temperatura`", zatim sledi dvotačka a zatim ceo broj koji sadrži vrednost temperature.

Listing 3.15: meteo.y

```
/* Parser koji broji dane cija je t > 12C */

%{
   #include <stdio.h>
   int yylex(void);
   int yyparse(void);
   extern int yylineno;
   int temp = 0;
%}

%union {
   int i;
   float f;
}
```

```
%token   _TEMPERATURA
%token   _PRITISAK
%token   _PRAVAC_VETRA
%token   _BRZINA_VETRA
%token   _VLAGA
%token   _DVOTACKA

%token   <i> _INT
%token   <f> _FLOAT
%token   _VETAR

%%

fajl
   :  slog
   |  fajl slog
   ;

slog
   :  temperatura pritisak pravac_vetra brzina_vetra vlaga
   ;

temperatura
   :  _TEMPERATURA _DVOTACKA _INT
         {  if($3 > 12) temp++;  }
   ;

pritisak
   :  _PRITISAK _DVOTACKA _FLOAT
   ;

pravac_vetra
   :  _PRAVAC_VETRA _DVOTACKA _VETAR
   ;

brzina_vetra
   :  _BRZINA_VETRA _DVOTACKA _INT
   ;
vlaga
   :  _VLAGA _DVOTACKA _INT
   ;

%%

int main() {
   yyparse();
   printf("U␣ovom␣mesecu␣bilo␣je␣%d␣dana␣sa␣t␣>␣12C.\n", temp);
}
```

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

Da bi izračunali koliko je dana bilo sa temperaturom preko 12 stepeni, treba nam promenljiva u kojoj ćemo brojati takve dane (promenljiva `temp`). Nakon prepoznavanja tokena `_INT` (što je ujedno i kraj pravila `temperatura`) smeštamo akciju koja inkremetira promenljivu `temp` ako je vrednost temperature `> 12`. Vrednosti broja pristupamo preko meta-promenljive `$3` (jer želimo da pročitamo vrednost uz token `_INT` koji se nalazi na 3-ćoj poziciji na desnoj strani pravila). Ispis broja dana se vrši iz `main` funkcije, nakon parsiranja svih slogova.

3.7 mC parser

Datoteka `defs.h` sadrži sve konstante koje su potrebne skeneru i parseru (drugim datotekama).

2.13

Listing 3.16: defs.h

```
#ifndef DEFS_H
#define DEFS_H

//tipovi podataka (moze ih biti maksimalno 8)
enum { NO_TYPE, INT_TYPE };

//konstante aritmetickih operatora
enum { ADD, SUB };

//konstante relacionih operatora
enum { EQ, NE };

#endif
```

U skeneru nam je bitan redosled pravila, tako da pravila za prepoznavanje ključnih reči treba staviti ispred pravila za identifikator. Skener je već viđen u prethodnom delu zbirke ().

Listing 3.17: syntax.l

```
/* Skener za mikroC */
%option noyywrap yylineno
```

```
%{
    #include <string.h>
    #include "syntax.tab.h"
    #include "defs.h"
%}

%%

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

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

"("                  { return _LPAREN; }
")"                  { return _RPAREN; }
"{"                  { return _LBRACKET; }
"}"                  { return _RBRACKET; }
";"                  { return _SEMICOLON; }
"="                  { return _ASSIGN; }
"+"                  { yylval.i = ADD; return _AROP; }
"-"                  { yylval.i = SUB; return _AROP; }
"=="                 { yylval.i = EQ; return _RELOP; }
"!="                 { yylval.i = NE; return _RELOP; }

[a-zA-Z][a-zA-Z0-9]* { yylval.s = yytext; return _ID; }
[+-]?[0-9]{1,10}     { yylval.s = yytext; return _INT_NUMBER;}

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

%%
```

Tip globalne promenljive `yylval` je definisan kao unija koja sadrži jedan ceo broj i jedan pokazivač na string. Tokeni koji imaju vrednostu celobrojnog tipa su `_AROP` i `_RELOP`, a tokeni koji imaju vrednost tipa string su `_ID` i `_INT_NUMBER`.

U drugom delu `bison` specifikacije navedena je gramatika mC programskog jezika, što znači da će parser proveravati sintaksnu ispravnost mC programa.

Listing 3.18: syntax.y

```
// Parser za mC
%{
    #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 _TYPE
%token _IF
%token _ELSE
%token _RETURN
%token <str> _ID
%token <str> _INT_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 _RPAREN body
   ;

type
   :  _TYPE
   ;

body
   :  _LBRACKET variable_list statement_list _RBRACKET
```

```
    |  _LBRACKET statement_list _RBRACKET
    ;

variable_list
    :  variable _SEMICOLON
    |  variable_list variable _SEMICOLON
    ;

variable
    :  type _ID
    ;

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
    :  constant
    |  _ID
    |  function_call
    |  _LPAREN num_exp _RPAREN
    ;

constant
    :  _INT_NUMBER
    ;

function_call
```

```
    :   _ID _LPAREN _RPAREN
    ;

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, "\nline␣%d:␣ERROR:␣%s", yylineno, s);
    return 0;
}

int main() {
    printf("\nSTART\n");
    yyparse();
    printf("\nSTOP\n");
    return 0;
}
```

Čim naiđe na prvu sintaksnu grešku, parser će završiti parsiranje, tj. završiće izvršavanje funkcije `yyparse()`.

3.7.1 Oporavak od greške na primeru mC parsera

Ukoliko korisnik želi da parser detektuje više od jedne greške, mora implementirati detekciju i oporavak greške. `Bison` nudi dve vrste opravka od greške, pa ćemo ih pokazati na prethodnom primeru (3.18).

Prvi primer oporavka je primenjen na `if` pravilo i koristi *error* token (vidi 3.4). Ovaj token je postavljen između zagrada, tako da, ukoliko dođe do sintaksne greške unutar zagrada relacionog izraza, parser će moći da obavi

redukciju po ovom pravilu. 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.

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

Drugi primer oporavka je primenjen na `return` pravilo i ima oblik pravila (*production rule*). Pravilu za `return` iskaz je dodato još jedno pravilo u kom nedostaje znak tačka-zarez na kraju iskaza. Ako se u ulanom mC 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 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.

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

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

3.8 Vežbe

1. Proširiti kalkulator tako da prihvata i linijske komentare.
2. Proširiti kalkulator tako da prihvata i heksa i decimalne brojeve. U skeneru dodati obrazac `0x[a-f0-9]+` za prepoznavanje heksa cifara, a u akciji koristiti funkciju `strtol` za konverziju stringa u broj koji se

prenosi preko `yylval`, i vratiti token `NUMBER`. Prilagoditi poziv funkcije `printf` tako da može da štampa rezultat i kao decimalni i kao heksa broj.

3. Proširiti kalkulator operatorima na nivou bita, kao što su `AND` i `OR`.

4. Napisati gramatike za svaki od ovih jezika:

   (a) Svi nizovi reči "da" i "ne" koji sadrže isti broj reči "da" i reči "ne" (u bilo kom redosledu).

   (b) Svi nizovi reči "da" i "ne" koji sadrže duplo više reči "da" od "ne".

Glava 4

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 `string` poredi sa promenljivom tipa `float`, kompajler treba da prijavi semantičku grešku . Semantika programskog jezika se u praksi opisuje neformalno, iako postoje razne formalne metode za opis semantike.

4.1 Semantička pravila za mC jezik

4.1.1 Standardni identifikatori

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

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

Telo `main` funkcije definiše korisnik. Ako telo `main` (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.

4.1.2 Opseg vidljivosti (*scope*)

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

Globalni identifikatori su imena globalnih promenljivih i 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 ime koje je najbliže toj upotrebi imena. Opseg vidljivosti deklaracije je podstablo sintaksnog stabla a ugnježdene deklaracije su ugnježdena podstabla. Najbliža deklaracija imena je ona deklaracija koja odgovara najmanjem podstablu koji obuhvata upotrebu imena. Primer C iskaza:

```
{
  int x = 1;
  int y = 2;
  {
    double x = 3.14;
    y += (int)x;
  }
  y += x;
}
```

U primeru postoje 2 različite promenljive `x`, deklarisane u 2 različita opsega vidljivosti. Druga promenljiva `x` je sakrila vidljivost prve. Zato će vrednost promenljive `y` na kraju ovog dela programa biti `6`.

4.1.3 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.

4.1.4 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). Isto važi i za konstante.

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. Isto važi i za konstante.

4.1.5 Organizacija memorije za mC

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 4.1.

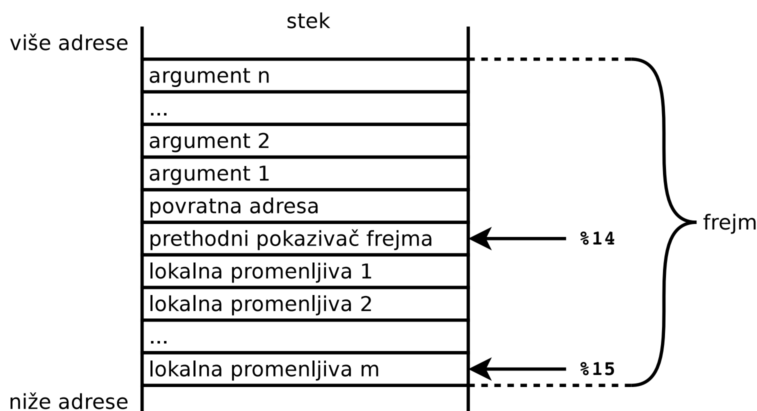
4.2 Tabela simbola

Kompajler mora da beleži informacije o imenima i njima pridruženim objektima tako da se upotreba imena veže za dobru deklaraciju. Ovo se radi pomoću tabele simbola (*symbol table*).

Tabela simbola čuva imena i sve informacije o tim imenima. Neke od informacija o simbolima koje su potrebne kompajleru su:

– ime

– vrsta simbola



Slika 4.1: Stek frejm

- tip simbola
- za ime funkcije: broj parametara i tipovi parametara
- za argumente/parametre: redni broj argumenta/parametra
- indikator opsega vidljivosti (*scope*)
- indikator prostora imena (*namespace*)

Za rad sa tabelom simbola potrebne su sledeće operacije:

- ubaciti novo ime u tabelu simbola
  - ako takvo ime ne postoji u tabeli simbola - ubaci ga
  - ako takvo ime već postoji u tabeli, prijavi grešku o duplikatu
  - novo ime ubaciti u tekući opseg vidljivosti
- vezati neku informaciju za određeno ime
  - imenu funkcije dodeliti tip povratne vrednosti, oznaku da je u pitanju funkcija, broj parametara, tipove parametara ...
  - promenljivim, parametrima, argumentima dodeliti tip, ...
- potražiti ime u tabeli simbola da bi se pristupilo njegovim informacijama

  - ako traženo ime nije definisano u tabeli simbola - prijaviti grešku
  - pretragu započeti u tekućem opsegu vidljivosti, pa ići prema globalnom

– ulazak u novi opseg vidljivosti

  - postaviti indikatore opsega vidljivosti

– izlazak iz opsega vidljivosti i vraćanje sadržaja tabele simbola u stanje koje je bilo pre ulaska u opseg

4.2.1 Implementacija tabele simbola

Za potrebe kursa koristi se vrlo jednostavna implementacija tabele simbola: tabela je organizovana kao niz struktura koje opisuju ime (4.2). Struktura sadrži opis imena, vrste, tipa i atributa simbola.

STRING SIMBOLA	VRSTA SIMBOLA	TIP SIMBOLA	ATRIBUT SIMBOLA
main	FUNCTION	int	-
x	LOCAL_VAR	int	1
y	LOCAL_VAR	int	2
0	CONSTANT	int	-

Slika 4.2: Tabela 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 njihove 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 registar `%14`, tj. njihov redni broj.

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.

Izlazak iz opsega vidljivosti briše svoj deo tabele simbola. Za jednostavne jezike, kao što je mC, ovo nije važno, jer kada se jednom izađe iz njega, opseg se više ne koristi. Međutim, osobine jezika kao što su klase ili moduli mogu zahtevati postojanje njihovih struktura u tabeli simbola i nakon izlaska iz njihovog opsega vidljivosti.

4.2.1.1 Pitanje efikasnosti

Problem efikasnosti se ogleda u operaciji pretraživanja tabele simbola (*lookup*): ako se pretraga radi linearno, u najgorem slučaju, vreme pretrage je proporcionalno veličini tabele simbola. Uobičajeno rešenje ovog problema je heširanje (*hashing*). Imena se heširaju (procesiraju) u brojeve koji se koriste da indeksiraju niz. Svaki element niza je linearna lista koja sadrži imena sa istim heš ključem. Uz dovoljno veliku heš tabelu, ove liste će biti vrlo kratke, tako da je vreme pretrage u osnovi konstantno.

4.2.1.2 mC tabela simbola

Datoteka `defs.h` (4.1) sadrži konstante koje se koriste u tabeli simbola.

Enumeracija `types` sadrži konstante koje opisuju tipove. Za sada mC jezik podržava samo jedan tip `int`.

Enumeracija `kinds` 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.

Enumeracija `arops` sadrži konstante koje opisuju aritmetičke operatore sabiranja i oduzimanja, a enumeracija `relops` sadrži konstante koje opisuju relacione operatore (`==` i `!=`).

Listing 4.1: defs.h

```
#ifndef DEFS_H
#define DEFS_H

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

#define SYMBOL_TABLE_LENGTH   64
#define CHAR_BUFFER_LENGTH    128
#define NO_ATTRIBUTE          -1
#define LAST_WORKING_REGISTER 12
```

```
#define FUNCTION_REGISTER     13
#define printerror(args...) sprintf(char_buffer, args), \
                            yyerror(char_buffer)
#define printwarning(args...) sprintf(char_buffer, args), \
                              warning(char_buffer)

extern char char_buffer[CHAR_BUFFER_LENGTH];

//tipovi podataka
enum types { NO_TYPE, INT_TYPE };

// vrste simbola (moze ih biti maksimalno 32)
enum kinds { NO_KIND = 0x1, REGISTER = 0x2, CONSTANT = 0x4,
   FUNCTION = 0x8, LOCAL_VAR = 0x10 };

//konstante arithmetickih operatora
enum arops { ADD, SUB };

//konstante relacionih operatora
enum relops { EQ, NE };

#endif
```

Struktura jednog elementa tabele simbola se sastoji od

- stringa imena,
- vrste simbola - sadrži vrednost iz enumeracije `kinds`, vidi 4.1
- tipa simbola - sadrži vrednost iz enumeracije `types`, vidi 4.1
- atributa - sadrži različite vrednosti za različite vrste simbola
  - funkcija - broj parametara
  - lokalna promenljiva - redni broj promenljive

Listing 4.2: symtab.h

```
#ifndef SYMTAB_H
#define SYMTAB_H

// Element tabele simbola
typedef struct sym_entry {
   char *name;            // ime simbola
   unsigned kind;         // vrsta simbola
   unsigned type;         // tip vrednosti simbola
```

```
   int attribute;          // dodatni attribut simbola
} SYMBOL_ENTRY;

// Vraca indeks prvog sledeceg praznog elementa.
int get_next_empty_element(void);

// Vraca indeks poslednjeg zauzetog elementa.
int get_last_element(void);

// Ubacuje simbol sa datom vrstom simbola i tipom simbola
// i vraca indeks ubacenog elementa u tabeli simbola ili -1.
int insert_symbol(char *name, unsigned kind, unsigned type);

// Proverava da li se simbol vec nalazi u tabeli simbola,
// ako se ne nalazi ubacuje ga, ako se nalazi ispisuje gresku.
// Vraca indeks elementa u tabeli simbola.
int try_to_insert_id(char *name, unsigned kind, unsigned type);

// Ubacuje konstantu u tabelu simbola (ako vec ne postoji).
int try_to_insert_constant(char *str, unsigned type);

// Vraca indeks pronadjenog simbola ili vraca -1.
int lookup_id(char *name, unsigned kind);

// Vraca indeks pronadjene konstante ili vraca -1.
int lookup_constant(char *name, unsigned type);

// set i get metode za polja elementa tabele simbola
char*    get_name(int index);
unsigned get_kind(int index);
unsigned get_type(int index);
void     set_attribute(int index, int attribute);
unsigned get_attribute(int index);
void     set_register_type(int register_index, unsigned type);

// Brise elemente tabele od zadatog indeksa
void clear_symbols(unsigned begin_index);

// Brise sve elemente tabele simbola.
void  clear_symtab(void);

// Ispisuje sve elemente tabele simbola.
void print_symtab(void);
unsigned logarithm2(unsigned value);

// Inicijalizacija tabele simbola.
void init_symtab(void);

#endif
```

4.3 mC parser sa semantičkim proverama

Datoteka `sem.c` (4.3) sadrži dve funkcije koje vrše semantičke provere. Prva proverava da li u programu postoji `main` funkcija i da li je njen tip `int`. To radi tako što proveri da li se u tabeli simbola nalazi simbol `main`, pa ako se ne nalazi prijavi semantičku grešku. Drugi deo provere se radi ako simbol postoji u tabeli, pa se za njega proveri tip i prijavi semantička greška ukoliko je različit od `int`.

Druga funkcija, `check_types()`, koja vrši proveru tipova, ima dva parametra: indekse dva elementa u tabeli simbola. Ova funkcija pristupi elementima na datim indeksima i poredi njihove tipove. Vraća vrednost `TRUE` ako su isti ili `FALSE` ako su različiti.

Listing 4.3: sem.c

```
#include <stdio.h>
#include "sem.h"

// Proverava da li postoji 'main' funkcija
// i da li joj je tip 'int'.
void check_main() {
   int index;
   if((index = lookup_id("main", FUNCTION)) == -1)
      printerror("undefined reference to 'main'");
   else
      if(get_type(index) != INT_TYPE)
         printwarning("return type of 'main' is not int");
}

// Proverava tipove 2 elementa u tabeli simbola.
bool check_types(int first_index, int second_index) {
   unsigned t1 = get_type(first_index);
   unsigned t2 = get_type(second_index);
   if(t1 == t2 && t1 != NO_TYPE)
      return TRUE;
   else
      return FALSE;
}
```

Sledi mC parser (`semantic.y` 4.7) koji koristi prethodno pomenutu tabelu simbola i semantičke provere iz `sem.c` (4.3).

U prvom delu specifikacije, definisani su tipovi pojmova, koji slede iza ključne reči `%type`. Nakon toga se navodi tip iz unije (`<i>`) a zatim imena svih pojmova koji će biti tog tipa.

Listing 4.4: semantic.y-1

```
// Parser za mC sa semantickom analizom
%{
   #include <stdio.h>
   #include "defs.h"
   #include "symtab.h"
   #include "sem.h"

   int yyparse(void);
   int yylex(void);
   int yyerror(char *s);
   void warning(char *s);

   extern int yylineno;
   int error_count = 0;
   char char_buffer[CHAR_BUFFER_LENGTH];
   int var_num = 0;
   int function_index = -1;
   int function_call_index = -1;
%}

%union {
   int i;
   char *s;
}

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

%type <i> type variable num_exp exp constant
%type <i> function_call rel_exp

%%
```

Semantička provera koju treba obaviti nakon prepoznavanja celog programa je da li postoji `main` funkcija i kog je tipa. Zato se funkcija koja to radi

(`check_main()`) poziva iz akcije koja je pridružena pojmu `program`.

Listing 4.5: semantic.y-2

```
program
   :  function_list
         {  check_main();  }
   ;

function_list
   :  function
   |  function_list function
   ;
```

Pojam `type`, dobija uvek vrednost konstante `INT_TYPE` jer, za sada, programski jezik mC podržava jedino tip `int`.

Listing 4.6: semantic.y-3

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

function
   :  type _ID
         {
            function_index = try_to_insert_id($2,FUNCTION,$1);
            var_num = 0;
         }
      _LPAREN _RPAREN body
         {
            // izbaci iz tabele simbola sve lokalne simbole
            if(get_last_element() > function_index)
               clear_symbols(function_index + 1);
         }
   ;
```

Tokom parsiranja definicije funkcije vrši se nekoliko semantičkih akcija. 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`) i vrsti identifikatora (konstanta `FUNCTION`). Promenljiva `function_index` dobija indeks elementa u tabeli simbola u koji je smešten novi simbol. Promenljiva koja broji lokalne promenljive jedne funkcije se resetuje, jer sada sledi ulazak u telo 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 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 koja briše lokalne simbole `clear_symbols()` se poziva sa argumentom `function_index+1` jer brisanje treba početi nakon imena funkcije. Ime funkcije je globalni identifikator i treba da bude prisutan u tabeli simbola tokom parsiranja celog programa.

Nakon parsiranja pojma `variable`, odnosno deklaracije promenljive, ime promenljive (vrednost `$2`) se ubacuje u tabelu simbola kao `LOCAL_VAR` sa tipom koji nosi pojam type (`$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 4.7: semantic.y-4

```
body
   :  _LBRACKET variable_list statement_list _RBRACKET
   |  _LBRACKET statement_list _RBRACKET
   ;

variable_list
   :  variable _SEMICOLON
   |  variable_list variable _SEMICOLON
   ;

variable
   :  type _ID
        {
           $$ = try_to_insert_id($2, LOCAL_VAR, $1);
           var_num++;
           set_attribute($$, var_num);
        }
   ;
```

Nakon parsiranja iskaza dodele proverava se da li je identifikator koji se nalazi sa leve strane jednakosti lokalna promenljiva. Ako nije prijavljuje se semantička greška, jer je na tom mestu dozvoljeno samo ime promenljive (a ne, recimo, ime funkcije). Dodatno, vrši se i provera tipova koji se nalaze sa leve i desne strane jednakosti, jer bi 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 (`$3`).

Listing 4.8: semantic.y-5

```
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
         {
            int i = -1;
            if( (i = lookup_id($1, LOCAL_VAR)) == -1 )
               printerror("invalid␣lvalue␣in␣assignment");
            if(!check_types(i, $3))
               printerror("incompatible␣types␣in␣assignment");
         }
   ;
```

Nakon parsiranja aritmetičkog izraza (pojam `num_exp`), vrši se provera tipova operanada, pa ako nisu isti, prijavljuje se semantička greška. 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 4.9: semantic.y-6

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

Nakon parsiranja identifikatora u izrazima (`exp`) vrši se provera postojanja dotičnog imena, tj. njegovog prisustva u tabeli simbola. 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

(`exp`) nalazi u zagradama, njegova vrednost treba da se prenese na vrednost celog izraza.

Listing 4.10: semantic.y-7

```
exp
   :  constant
   |  _ID
         {
            if( ($$ = lookup_id($1, LOCAL_VAR)) == -1)
               printerror("'%s'␣undeclared", $1);
         }
   |  function_call
   |  _LPAREN num_exp _RPAREN
         {  $$ = $2; }
   ;
```

Kada se isparsira konstanta, treba je ubaciti u tabelu simbola kao lokalni simbol, a pojmu `constant` dodeliti vrednost: indeks u tabeli simbola na kom je smeštena konstanta.

Listing 4.11: semantic.y-8

```
constant
   :  _INT_NUMBER
         {  $$ = try_to_insert_constant($1, INT_TYPE); }
   ;
```

Tokom parsiranja poziva funkcije (pojam `function_call`) proverava se da li je navedeni identifikator ime neke od postojećih funkcija, pa ako nije, prijavljuje grešku da se poziva nepostojeća funkcija. Kako je dogovoreno, 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`) i registru postaviti tip povratne vrednosti funkcije.

Listing 4.12: semantic.y-9

```
function_call
   :  _ID _LPAREN _RPAREN
         {
            if( (function_call_index =
               lookup_id($1, FUNCTION)) == -1 )
               printerror("'%s'␣is␣not␣a␣function", $1);

            // povratna vrednost funkcije se uvek nalazi u %13
            set_register_type(FUNCTION_REGISTER,
               get_type(function_call_index));
            $$ = FUNCTION_REGISTER;
```

```
        }
    ;
```

Tokom parsiranja relacionog izraza (pojma rel_exp) potrebno je proveriti tipove operanada jer 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 4.13: semantic.y-10

```
if_statement
    :   if_part
    |   if_part _ELSE statement
    ;

if_part
    :   _IF _LPAREN rel_exp _RPAREN statement
    ;

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

Nakon prepoznavanja return iskaza potrebno je proveriti tip izraza koji se vraća iz funkcije, jer bi trebao 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 (function_index) i indeks izraza koji se vraća iza funkcije.

Listing 4.14: semantic.y-11

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

Funkcija main vrši inicijalizaciju tabele simbola pre početka parsiranja i brisanje tabele simbola nakon parsiranja.

Listing 4.15: semantic.y-12

```
%%

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);
}

int main() {
   printf("\nSTART\n");
   init_symtab();
   yyparse();
   clear_symtab();
   printf("\n%d errors\nSTOP\n", error_count);
   return error_count;
}
```

4.4 Vežbanje

4.4.1 Vežba 1

Odaberite neki programski jezik koji dobro znate i odredite koji od sledećih objekata dele prostor imena: promenljive, funkcije, procedure i tipovi. Ako postoji više vrsta imenovanih objekata u jeziku (labele, konstruktori, moduli, itd.) uzmite i njih u obzir.

4.4.2 Vežba 2

U nekim programskim jezicima identifikatori su *case-insensitive*, pa se npr. `size` i `SiZe` odnose na isti identifikator. Opišite kako bi se tabela simbola mogla napraviti *case-insensitive*.

Glava 5

Generisanje koda

Generisanje koda je faza kompajliranja u kojoj se proizvodi datoteka sa ekvivalentnim programom napisanim na ciljnom programskom jeziku.

U ovom kursu, ciljni jezik na koji će biti preveden mC programski jezik, je hipotetski asemblerski jezik.

5.1 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 '`@`'.

5.1.1 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.

direktni operand odgovara labeli. 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

5.1.2 Naredbe

Neke naredbe postoje u 3 varijante za 3 različita tipa podatka, koje su očnačene 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 potprograma 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 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

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

naredba konverzije celog broja u razlomljeni broj vrednost ulaznog operanda je celi broj, a vrednost izlaznog operanda je ekvivalentni razlomljeni broj u mašinskoj normalizovanoj formi

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

naredba konverzije razlomljenog broja u celi broj vrednost ulaznog operanda je razlomljeni broj u mašinskoj normalizovanoj formi, a vrednost izlaznog operanda je ekvivalentni celi broj ako je konverzija moguća, inače se izaziva izuzetak

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

5.1.3 Direktive

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

```
WORD broj
```

5.2 Primeri generisanja koda

5.2.1 Globalne promenljive

Za svaku promenljivu treba generisati asemblersku direktivu sa odgovarajućom labelom. Za globalnu promenljivu

```
int a;
```

treba generisati kod

```
a: WORD 1
```

Labela "`a`" ne započinje znakom "`@`" jer odgovara identifikatoru koga je zadao korisnik.

5.2.2 Iskaz pridruživanja

Primer iskaza pridruživanja (podrazumeva se da su `a`, `b`, `c`, `d` i `e` celobrojne označene globalne promenljive)

```
a = (-a + b) * (c + d) - e
```

Generisani kod za prethodni iskaz:

```
SUBS $0,a,%0    linija 1
ADDS %0,b,%0    linija 2
ADDS c,d,%1     linija 3
MULS %0,%1,%0   linija 4
SUBS %0,e,%0    linija 5
MOV  %0,a       linija 6
```

Za smeštanje međurezultata izraza koriste se radni registri. Podrazumeva se da su svi radni registri na početku slobodni. U liniji 1 se zauzima radni registar `%0`. On se oslobađa i ponovo zauzima u liniji 2, a u liniji 3 se zauzima radni registar `%1`. U liniji 4 se oslobađaju radni registri `%0` i `%1`, a ponovo se zauzima radni registar `%0`. U liniji 5 se oslobađa i ponovo zauzima radni registar `%0`, a u liniji 6 se oslobađa radni registar `%0`.

Radni registar se zauzima za smeštanje rezultata svakog aritmetičkog izraza sa: dva operanda i jednim operatorom (`num_exp`, `mul_exp`) unarnim operatorom i jednim operandom (`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 koji odgovara izrazu čiji rezultat radni registar sadrži. Prekoračenje broja radnih registara (`free_reg_num > 12`) predstavlja fatalnu grešku u radu kompajlera.

5.2.3 Funkcija

5.2.3.1 Definicija funkcije

Primer funkcije:

```
int f(int x, int y) {
   int z;
   return x + y;
}
```

Za smeštanje povratne vrednosti funkcije koristi se radni registar `%13`. Funkcija, nakon poziva, lokalne promenljive i parametre čuva u stek frejmu: 4.1. Definiciji funkcije iz primera odgovara izgenerisani kod:

```
f:                              linija 1
   PUSH %14                     linija 2
   MOV  %15,%14                 linija 3
   SUBS %15,$4,%15              linija 4
@f_body:                        linija 5
   ADDS 8(%14),12(%14),%0       linija 6
   MOV  %0,%13                  linija 7
   JMP  @f_exit                 linija 8
@f_exit:                        linija 9
   MOV  %14,%15                 linija 10
   POP  %14                     linija 11
   RET                          linija 12
```

U linijama 2 i 3 se postavlja pokazivač frejma, a u liniji 4 se zauzima prostor na steku za lokalnu promenljivu `z`. Promenljiva `var_num` služi kao brojač lokalnih promenljivih. Veličina prostora za lokalne promenljive se određuje kao `var_num * 4`. Prostor se zauzima samo ako je `var_num > 0`. U linijama 6 i 7 se računa povratna vrednost funkcije i smešta u registar `%13`. Ako funkcija ne sadrži `return` iskaz, kao povratna vrednost funkcije služi zatečeni sadržaj registra `%13`, koji je nepoznat u vreme definisanja funkcije. U liniji 10 se oslobađa prostor za lokalne promenljive, a u liniji 11 se vraća prethodna vrednost u pokazivač frejma.

5.2.3.2 Poziv funkcije

Primeru poziva funkcije (podrazumeva se da su a, b, c i d celobrojne označene globalne promenljive):

```
a = f(a + b, c - d)
```

odgovara izgenerisani kod:

```
ADDS a,b,%0       linija 1
SUBS c,d,%1       linija 2
PUSH %1           linija 3
PUSH %0           linija 4
CALL f            linija 5
ADDU %15,$8,%15   linija 6
MOV  %13, a       linija 7
```

U linijama 1 i 2 se računaju argumenti i čuvaju u radnim registrima (pretpostavka je da su svi radni registri slobodni). U linijama 3 i 4 argumenti se smeštaju na stek, a u liniji 6 oslobađa se prostor koji su zauzimali argumenti. Veličina oslobađanog prostora je `arg_num * 4` i on se oslobađa samo ako je `arg_num > 0`. U liniji 7 se isporučuje povratna vrednost funkcije.

Ako su argumenti konstante ili promenljive, kao u slučaju poziva (podrazumeva se da su a i b celobrojne označene globalne promenljive)

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

onda nisu potrebni radni registri, jer se konstante i promenljive direktno mogu smeštati na stek:

```
PUSH b
PUSH $1
CALL f
ADDU %15,$8,%15
MOV  %13,a
```

Pošto argumentima odgovaraju ili radni registri ili konstante ili promenljive, svakom argumentu je pridružen indeks elementa tabele simbola koji je rezervisan ili za pomenuti radni registar, ili za pomenutu konstantu ili za

pomenutu promenljivu. Pošto se argumenti na stek smeštaju od poslednjeg ka prvom, zgodno je na posebnom steku argumenata kompajlera čuvati njihove indekse. Ovi indeksi se smeštaju na stek argumenata od prvog ka poslednjem argumentu, tako da su pripremljeni za generisanje naredbi koje smeštaju argumente na stek. Za čuvanje broja argumenata iz poziva funkcije zgodno je uvesti poseban stek broja argumenata kompajlera (ovaj stek omogućuje da se pamte brojevi argumenata kada se u pozivu neke funkcije kao argument javi opet poziv funkcije).

Ako su argumenti pozivi novih funkcija (podrazumeva se da je `a` celobrojna označena globalna promenljiva, a da funkcija `f2` ima jedan parametar tipa `int`)

```
a = f(1,f2(3));
```

onda generisani kod izgleda ovako:

```
PUSH $3           linija 1
CALL f2           linija 2
ADDU %15,$4,%15   linija 3
MOV  %13,%0       linija 4
PUSH %0           linija 5
PUSH $1           linija 6
CALL f            linija 7
ADDU %15,$8,%15   linija 8
MOV  %13,a        linija 9
```

Kada se u pozivu neke funkcije kao argument javi opet poziv funkcije, prvo se izvršava poziv te funkcije da bi se izračunala vrednost odgovarajućeg argumenta. U linijama 1, 2 i 3 se izvršava poziv funkcije `f2` da bi se izračunala vrednost drugog argumenta poziva funkcije `f`.

U slučaju pojave poziva funkcije na mestu argumenta poziva neke druge funkcije zgodno je uvesti poseban stek poziva funkcija kompajlera. Ovaj stek omogućuje da se pamte indeksi (imena) funkcija u tabeli simbola čiji pozivi su se javili kao argument poziva neke funkcije.

5.2.3.3 Iskaz poziva funkcije

Iskazu poziva funkcije

```
f(1,2);
```

odgovara izgenerisani kod:

```
PUSH $2
PUSH $1
CALL f
ADDU %15,$8,%15
```

5.2.4 `if` iskaz

5.2.4.1 if iskaz sa else delom

Primeru `if` iskaza

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

(podrazumeva se da su `a` i `b` celobrojne označene globalne promenljive) odgovara izgenerisani kod:

```
@if0:               linija 1
   CMPS a,b         linija 2
   JLES @false0     linija 3
@true0:             linija 4
   MOV  $1,a        linija 5
   JMP  @exit0      linija 6
@false0:            linija 7
   MOV  $2,a        linija 8
@exit0:             linija 9
```

Labele u generisanom kodu moraju biti jedinstvene. Svaka labela se završava brojem koji sadrži promenljiva (aktuelni broj labele). Jednoznačni brojevi se dobijaju inkrementiranjem promenljive `lab_num`. Broj uz labelu `if` se čuva jer se koristi i uz labele `@true` i `@exit`. Promenljiva sadrži aktuelni broj `@false` labele. Zaseban brojač je neophodan pošto u jednom logičkom izrazu može biti više `false` labela.

5.2.4.2 if iskaz sa else delom

Primeru `if` iskaza

```
if(a > b && c > d)
   a = 1;
else
   a = 2;
```

(podrazumeva se da su `a`, `b`, `c` i `d` celobrojne označene globalne promenljive) odgovara izgenerisani kod (podrazumeva se da se generisanje koda nastavlja na prethodni primer)

```
@if1:
   CMPS a,b
   JLES @false1
   CMPS c,d
   JLES @false1
@true1:
   MOV  $1,a
   JMP  @exit1
@false1:
   MOV  $2,a
@exit1:
```

5.2.4.3 if iskaz sa else delom

Primeru `if` iskaza

```
if(c > d || a > c)
   a = 1;
else
   a = 2;
```

(podrazumeva se da su `a`, `b`, `c` i `d` celobrojne označene globalne promenljive) odgovara izgenerisani kod (podrazumeva se da se generisanje koda nastavlja na prethodni primer)

```
@if2:
   CMPS c,d
   JGTS @true2
@false2:
   CMPS a,c
   JLES @false3
@true2:
   MOV  $1,a
   JMP  @exit2
@false3:
   MOV  $2,a
@exit2:
```

5.2.4.4 if iskaz sa else delom

Primeru `if` iskaza

```
if(a > b && c > d || a > c && b > d || a > e)
   a = 1;
else
   a = 2;
```

(podrazumeva se da su `a`, `b`, `c`, `d` i `e` celobrojne označene globalne promenljive) odgovara izgenerisani kod:

```
@if0:              linija 1
   CMPS a,b        linija 2
   JLES @false0    linija 3
   CMPS c,d        linija 4
   JGTS @true0     linija 5
@false0:           linija 6
   CMPS a,c        linija 7
   JLES @false1    linija 8
   CMPS b,d        linija 9
   JGTS @true0     linija 10
@false1:           linija 11
   CMPS a,e        linija 12
   JLES @false2    linija 13
@true0:            linija 14
```

```
   MOV  $1,a      linija 15
   JMP  @exit0    linija 16
@false2:          linija 17
   MOV  $2,a      linija 18
@exit0:           linija 19
```

Pošto se kao `then` iskaz može pojaviti novi `if` iskaz i tako dalje, neophodno je sačuvati zatečenu vrednost promenljive `lab_num`, jer će ona biti izmenjena u toku generisanja koda za novi `if` iskaz (zbog sekvencijalnog generisanja koda zatečena vrednost će biti naknadno korišćena). Iz istih razloga se mora sačuvati i vrednost promenljive `false_lab_num` pošto se ona koristi uz poslednju labelu `@false` u delu koda koji još nije izgenerisan za spoljašnji `if` iskaz.

Za čuvanje vrednosti ovih promenljivih se koristi poseban stek labela kompajlera. Na njega se ove vrednosti smeštaju pre tretiranja `then` iskaza, a nakon tretiranja se preuzimaju sa steka labela. Isto važi i za vrednost promenljive `lab_num` pre i nakon tretiranja `else` iskaza.

Od linije 2 do linije 13 je opisano određivanje vrednosti logičkog izraza. Generisanje naredbi poređenja je vezano za pojam `rel_exp`. Vrednost ovog pojma je vrednost relacionog operatora, da bi se na osnovu nje mogla izgenerisati ispravna naredba uslovnog skoka. Njeno generisanje je vezano za `&&` operator pojma `and_exp`, `||` operator pojma `log_exp` ili za pojam `log_exp`.

5.2.4.5 if iskaz bez else dela

Primeru `if` iskaza

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

(podrazumeva se da su `a` i `b` celobrojne označene globalne promenljive) ogovara izgenerisani kod:

```
@if0:
   CMPS a,b
   JLES @false0
@true0:
```

```
   MOV  $1,a
   JMP  @exit0
@false0:
@exit0:
```

5.2.4.6 if iskaz bez else dela

Primeru `if` iskaza

```
if(a > b && c > d || a > c && b > d || a > e) a = 1;
```

(podrazumeva se da su `a`, `b`, `c`, `d` i `e` celobrojne označene globalne promenljive) odgovara izgenerisani kod:

```
@if0:              linija 1
   CMPS a,b        linija 2
   JLES @false0    linija 3
   CMPS c,d        linija 4
   JGTS @true0     linija 5
@false0:           linija 6
   CMPS a,c        linija 7
   JLES @false1    linija 8
   CMPS b,d        linija 9
   JGTS @true0     linija 10
@false1:           linija 11
   CMPS a,e        linija 12
   JLES @false2    linija 13
@true0:            linija 14
   MOV  $1,a       linija 15
   JMP  @exit0     linija 16
@false2:           linija 17
@exit0:            linija 18
```

Od linije 2 do linije 13 je opisano određivanje vrednosti logičkog izraza.

5.2.5 while iskaz

Primeru `while` iskaza

```
while (a != b)
   if (a > b)
      a = a - b;
   else
      b = b - a;
```

(podrazumeva se da su `a` i `b` celobrojne označene globalne promenljive) odgovara izgenerisani kod (za generisanje koda primenjuje se pristup objašnjen u `if` iskazu):

```
@while0:
   CMPS a,b
   JEQ  @false0
@true0:
@if1:
   CMPS a,b
   JLES @false1
@true1:
   SUBS a,b,%0
   MOV  %0,a
   JMP  @exit1
@false1:
   SUBS b,a,%0
   MOV  %0,b
@exit1:
   JMP  @while0
@false0:
@exit0:
```

5.2.6 break iskaz

Primeru `break` iskaza (podrazumeva se da su `a` i `b` celobrojne označene globalne promenljive)

```
while(a < 5) {
   if(a == b)
      break;
   a = a + 1;
}
```

odgovara izgenerisani kod (podrazumeva se da se `break` iskaz sme naći samo unutar `while` iskaza):

```
@while0:
   CMPS a,$5
   JGES @false0
@true0:
@if1:
   CMPS a,b
   JNE  @false1
@true1:
   JMP  @exit0   //break
   JMP  @exit1
@false1:
@exit1:
   ADDS a,$1,%0
   MOV  %0,a
   JMP  @while0
@false0:
@exit0:
```

5.2.7 continue iskaz

Primeru `continue` iskaza (podrazumeva se da su `a` i `b` celobrojne označene globalne promenljive)

```
while(a < 5) {
   if(a == b)
      continue;
   a = a + 1;
}
```

odgovara izgenerisani kod (podrazumeva se da se `continue` iskaz sme naći samo unutar `while` iskaza):

```
@while0:
   CMPS a,$5
   JGES @false0
@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:
```

5.2.8 Logički izrazi

Pridruživanju vrednosti logičkih izraza iskazu

```
bool = a > b && c > d || a > c && b > d || a > e;
```

(podrazumeva se da su sve promenljive globalne, celobrojne i označene) odgovara kod:

```
   CMPS a,b
   JLES @false0
   CMPS c,d
   JGTS @true0
@false0:
   CMPS a,c
   JLES @false1
   CMPS b,d
   JGTS @true0
@false1:
   CMPS a,e
   JLES @false2
@true0:
   MOV  $1,bool
   JMP  @exit0
@false2:
```

```
   MOV  $0,bool
@exit0:
```

(podrazumeva se da su brojevi uz labele jedinstveni).

5.2.9 for iskaz

Iskazima

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

(podrazumeva se da su `suma` i `i` globalne, celobrojne i označene promenljive) odgovara kod:

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

(podrazumeva se da su brojevi uz labele jedinstveni).

5.2.10 switch iskaz

Iskazu

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

(podrazumeva se da je `state` globalna celobrojna označena promenljiva) odgovara kod (napomena: realizovani kompajler je jednoprolazni, pa je u skladu sa tim i napravljen sledeći asemblerski kod):

```
@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:
```

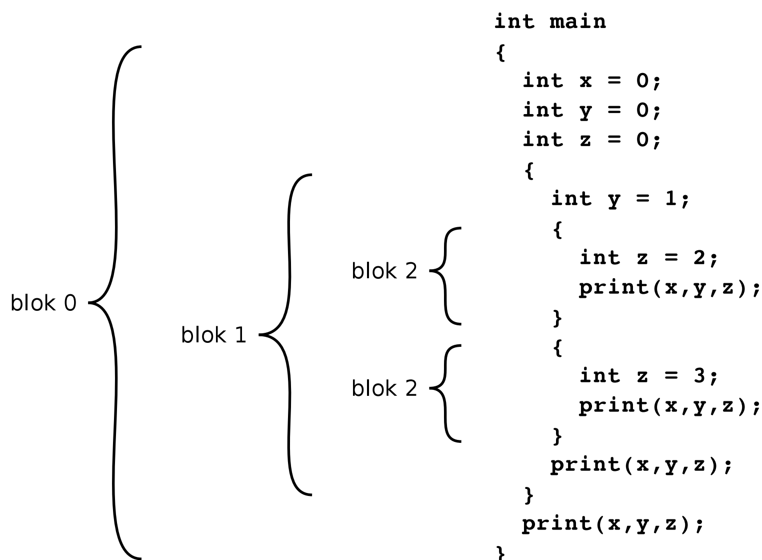
(podrazumeva se da je prvi broj `0` uz labele jedinstven).

Da su izostavljeni `break` iskazi, bile bi izostavljene i prve dve naredbe `JMP @exit0`.

5.2.11 C blokovi

Jedno od rešenja implementacije blokova: Blokovi se međusobno razlikuju po rednom broju koji im dodeljuje kompajler (5.1). U tabeli simbola za svaku lokalnu promenljivu bloka mora biti vezan redni broj bloka (indikator bloka). Kompajler koristi redne brojeve blokova kod određivanja opsega vidljivosti identifikatora (kada u bloku `n` naiđe na upotrebu neke promenljive, kompajler traži tu promenljivu u tabeli simbola za blok `n`, a ako je traženje neuspešno, ono se ponavlja za prethodni blok).

Rezultat štampanja navedenih `print` funkcija će biti:



Slika 5.1: C blokovi

```
0 1 2
0 1 3
0 1 0
0 0 0
```

Lokalne promenljive blokova se čuvaju u frejmu bloka na steku (za razliku od frejma poziva funkcije, frejm bloka ne sadrži argumente kao ni povratnu vrednost). Frejm bloka se stvara na ulazu u blok, a uništava na izlasku iz bloka.

5.2.12 Slogovi

Slogovi (C `struct`, Pascal `record`) - za svaki slog je potrebna posebna tabela simbola, koja sadrži njegova polja sa relativnom pozicijom u slogu kao dodatnim atributom. Za slog

```
struct {
   int x;
   int y;
}z
```

je potrebno zauzeti 2 lokacije. Iskazu

```
z.y = z.x;
```

odgovara kod:

```
MOV  $z,%0
MOV  0(%0),4(%0)
```

(podrazumeva se da je radni registar `%0` slobodan i da je u njega smeštena početna adresa sloga `z`).

5.2.13 Nizovi

Za svaki niz u tabeli simbola treba registrovati i broj njegovih elemenata. Za niz

```
int n[10];
```

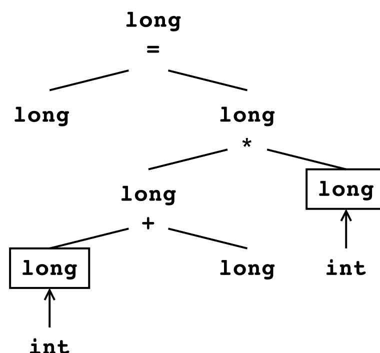
je potrebno zauzeti 10 lokacija. Iskazu

```
n[0] = n[1];
```

odgovara kod:

```
MOV  $n,%0
MOV  4(%0),0(%0)
```

(podrazumeva se da su radni registar `%0` slobodan i da je u njega smeštena početna adresa niza `n`).



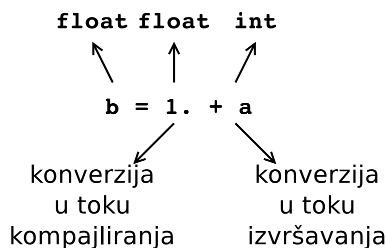
Slika 5.2: Konverzija tipova

5.2.14 Konverzija tipova

U izrazima sa promenljivim i konstantama raznih ali kompatibilnih tipova (`int` i `long` ili `int` i `float`) potrebno je obaviti podrazumevajuće konverzije tipova (`int` u `long` ili `int` u `float`) pre korišćenja vrednosti promenljivih ili konstanti.

```
long = (int + long) * int
```

Konverzije tipova se zasnivaju na određivanju tipova izraza i iskaza. Za određivanje tipova izraza (iskaza) potrebno je izraze (iskaze) predstaviti u obliku grafa čiji čvorovi sadrže tipove podizraza (podiskaza). Konverzija se obavlja (ako je moguća) kada se prepozna binarni izraz sa operandima raznih ali kompatibilnih tipova, ili kada se prepozna dodela vrednosti jednog tipa promenljivoj različitog kompatibilnog tipa. Radi konverzije kompajler generiše odgovarajuću naredbu za konverziju.



Slika 5.3: Različite konverzije tipova

5.3 MICKO

MICKO je akronim reči mC kompajler (MIkro C KOmpajler).

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_str_lab(char *str1, char *str2);
void gen_num_lab(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 čiji su parametri indeksi operanada u tabeli simbola:

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

Sledeća funkcija generiše MOV naredbu, a parametri su joj indeksi operanada 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`), dok su drugi i treći parametar indeksi operanada u tabeli simbola:

```
int gen_arith(int statement,
              int operand1_index, int operand2_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 pruhvata indeks elementa tabele simbola na kom se nalazi funkcija čiji poziv treba izgenerisati:

```
void gen_function_call(int name_index);
```

Generisanje koda se realizuje u parseru, tako da se faza generisanja koda, praktično, odvija paralelno sa sintaksnom i semantičkom analizom. Sledi `micko.y` datoteka: parser sa semantičkim proverama i generisanjem hipotetskog asemblerskog koda. Parser pre početka parsiranja napravi datoteku `output.asm` 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 (vidi `lab_num` i `false_lab_num`: 5.2.4.1, stek labela `label_stack`: 5.2.4.4 i `output`: 5.3):

Listing 5.1: micko.y-1

```
int lab_num;
int false_lab_num = -1;
stack label_stack;
FILE *output;
```

Generisanje koda za definiciju funkcije podrazumeva generisanje:

1. labele koja se isto zove kao funkcija
2. smeštanja starog pokazivača frejma (registar) 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 (`RET`).

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

Listing 5.2: micko.y-2

```
function
  : type _ID
      {
        function_index = try_to_insert_id($2,FUNCTION,$1);
        var_num = 0;

        code("\n%s:", $2);
        code("\n\t\t\tPUSH\t%%14");
        code("\n\t\t\tMOV␣\t%%15,%%14");
      }
    _LPAREN _RPAREN body
      {
        if (get_last_element() > function_index)
          clear_symbols(function_index + 1);

        gen_str_lab($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 (`body`) nalaze lokalne promenljive (`variable_list`) potrebno je izgenerisati zauzimanje prostora za njih na steku. 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` (za `var_num` vidi 4.3).

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

Listing 5.3: micko.y-3

```
body
  : _LBRACKET variable_list
      {
        code("\n\t\t\tSUBU\t%%15,$%d,%%15", 4*var_num);
        gen_str_lab(get_name(function_index), "_body");
      }
    statement_list _RBRACKET

  | _LBRACKET
      { gen_str_lab(get_name(function_index), "_body"); }
    statement_list _RBRACKET
  ;
```

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

Listing 5.4: micko.y-4

```
assignment_statement
  : _ID _ASSIGN num_exp _SEMICOLON
      {
        int i = -1;
        if( (i = lookup_id($1, LOCAL_VAR)) == -1 )
          printerror("invalid lvalue in assignment");
        if(!check_types(i, $3))
          printerror("incompatible types in assignment");
        gen_mov($3, i);
      }
  ;
```

Za generisanje aritmetičkih operacija koristi se `funkcija gen_arith()` koja generiše `ADDS` ili `SUBS` naredbu. Kao prvi argument prosleđuje se oznaka aritmetičke operacije (konstanta `ADD` ili `SUB`), kao prvi argument se prosleđuje vrednost `$1` koja sadrži indeks elementa u tabeli simbola koji opisuje prvi operand, a kao drugi argument se prosleđuje vrednost `$3` koja sadrži indeks elementa u tabeli simbola koji opisuje drugi operand.

Listing 5.5: micko.y-5

```
num_exp
  : exp

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

U pravilu za izraze (`exp`), samo pravilo koje opisuje poziv funkcije sadrži generisanje koda. Kada se poziv funkcije nađe negde u izrazu, njenu vrednost treba preuzeti 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`. Zato povratne vrednosti funkcija u izrazima treba čuvati u registrima. Vrednost pojma `exp` postaje indeks u tabeli simbola na kom se nalazi rezultat izvršavanja funkcije.

Listing 5.6: micko.y-6

```
exp
  : constant

  | _ID
      {
        if( ($$ = lookup_id($1, LOCAL_VAR)) == -1)
          printerror("'%s' undeclared", $1);
      }

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

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

Za generisanje poziva funkcije koristi se funkcija `gen_function_call()` koja generiše `CALL` naredbu. Vrednost pojma `function_call` se postavlja na registar `%13` a njegov tip se setuje na povratni tip funkcije.

Da su postojali argumenti poziva funkcije u gramatici, njihovo smeštanje na stek bi usledilo neposredno pre generisanja `CALL` naredbe.

Listing 5.7: micko.y-7

```
function_call
  : _ID _LPAREN _RPAREN
      {
        if((function_call_index = lookup_id($1, FUNCTION))==-1)
          printerror("'%s' is not a function", $1);

        gen_function_call(function_call_index);
        set_register_type(FUNCTION_REG,
          get_type(function_call_index));
        $$ = FUNCTION_REG;
      }
  ;
```

U okviru `if` iskaza treba izgenerisati više asemblerskih naredbi. Na početku `if` iskaza generiše se labela `@ifX`, gde je `X` redni broj iskaza u programu. Brojač (`lab_num` (vidi 5.2.4.1)) sa ovom vrednošću se na početku generisanja inkrementira, da bi označio sledeći iskaz. Na mestu relacionog izraza će se izgenerisati odgovarajuća `CMP` naredba (vidi listing 5.9). Nakon toga se

generiše skok sa kontra-uslovom. Ako je u relaciji bio naveden operator ==, izgenerisaće se skok JNE na false labelu if iskaza. Sve false labele (i jedino one) sadrže brojač false_lab_num (vidi 5.2.4.1). 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 (na kraju parsiranja pojma if_part), generiše se exit labela, kao oznaka kraja ekvivalentnog asemblerskog koda if iskaza.

Listing 5.8: micko.y-8

```
if_statement
  : if_part
      { gen_num_lab("exit", lab_num); }

  | if_part _ELSE
      { push(&label_stack, lab_num); }
    statement
      { gen_num_lab("exit", pop(&label_stack)); }
  ;

if_part
  : _IF _LPAREN
      {
        lab_num = ++false_lab_num;
        gen_num_lab("if", lab_num);
      }
    rel_exp
      {
        const char* opposite_jumps[2]={"JNE␣", "JEQ␣"};
        code("\n\t\t\t%s\t@false%d",
          opposite_jumps[$4], false_lab_num);
        gen_num_lab("true", lab_num);
        push(&label_stack, false_lab_num);
        push(&label_stack, lab_num);
      }
    _RPAREN statement
      {
        lab_num = pop(&label_stack);
        code("\n\t\t\tJMP\t@exit%d", lab_num);
        gen_num_lab("false", pop(&label_stack));
      }
  ;
```

Može da se desi da then ili else telo if-a sadrže drugi if ili neki drugi iskaz koji u svom generisanju koda takođe koristi brojače lab_num i false_lab_num.

U tom slučaju, potrebno je sačuvati vrednosti ova dva brojača pre ulaska u novi iskaz, i vratiti njihove vrednosti posle parsiranja novog iskaza (zbog nastavka parsiranja spoljašnjeg `if`-a). Ove vrednosti se čuvaju na steku labela (`label_stack` (vidi 5.2.4.4)).

Ekvivalent relacionog izraza (`rel_exp`) u asemblerskom jeziku je naredba poređenja (`CMP`). Za generisanje ove naredbe poziva se funkcija `gen_cmp()` sa argumentima koji sadrže indekse elemenata u tabeli simbola na kojima se nalaze operandi operacije poređenja. Kao vrednost pojma `rel_exp` definiše se konstanta koja predstavlja vrstu relacionog operatora (`EQ`, `NE`). Ova vrednost se koristi u `if` iskazu za generisanje koda (vidi listing 5.8).

Listing 5.9: micko.y-9

```
rel_exp
  : num_exp _RELOP num_exp
      {
        if(!check_types($1, $3))
          printerror("invalid operands: relational operator");
        gen_cmp($1, $3);
        $$ = $2;
      }
  ;
```

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

Listing 5.10: micko.y-10

```
return_statement
  : _RETURN num_exp _SEMICOLON
      {
        if(!check_types(function_index, $2))
          printerror("incompatible types in return");

        // vrednost return izraza se prebaci u registar %13
        gen_mov($2, FUNCTION_REG);
        code("\n\t\t\tJMP\t@%s_exit",get_name(function_index));
      }
  ;
```

Funkcija `main` pre početka parsiranja obavi potrebne inicijalizacije: incijalizaciju tabele simbola i steka labela i kreira i otvori datoteku `output.asm` za generisanje izlaznog koda. Zatim poziva parser, pa nakon parsiranja briše

tabelu simbola, zatvara datoteku i ako je bilo grešaka u toku parsiranja briše izlaznu datoteku jer nije validna.

Listing 5.11: micko.y-11

```
int main() {
  printf("\nSTART\n");
  init_symtab();
  init_stack(&label_stack);
  output = fopen("output.asm", "w+");

  yyparse();

  clear_symtab();
  fclose(output);
  printf("\nSTOP\n");

  if(error_count)
    remove("output.asm");
  return error_count;
}
```

Listings

2.1 klot.l 13

2.2 njam.l 15

2.3 num.l 15

2.4 ws.l 16

2.5 union.l 17

2.6 wclc.l 18

2.7 wsdot.l 20

2.8 comments.l 21

2.9 2to10.l 22

2.10 cmd.l 24

2.11 date.l 25

2.12 text.l 26

2.13 mcscanner.l 27

3.1 calc1.l 38

3.2 calc1.y 39

3.3 calc2.y 40

3.4 calc3.y 44

3.5 calc4.l 49

3.6 calc4.y 49

3.7 count2.l 52

3.8 count2.y 52

3.9 count3.y 54
3.10 count4.l 56
3.11 count4.y 56
3.12 date.l 59
3.13 date.y 59
3.14 meteo.l 61
3.15 meteo.y 62
3.16 defs.h 64
3.17 syntax.l 64
3.18 syntax.y 65
4.1 defs.h 76
4.2 symtab.h 77
4.3 sem.c 79
4.4 semantic.y-1 80
4.5 semantic.y-2 81
4.6 semantic.y-3 81
4.7 semantic.y-4 82
4.8 semantic.y-5 82
4.9 semantic.y-6 83
4.10 semantic.y-7 84
4.11 semantic.y-8 84
4.12 semantic.y-9 84
4.13 semantic.y-10 85
4.14 semantic.y-11 85
4.15 semantic.y-12 86
5.1 micko.y-1 109
5.2 micko.y-2 110
5.3 micko.y-3 110

5.4 micko.y-4 111
5.5 micko.y-5 111
5.6 micko.y-6 112
5.7 micko.y-7 112
5.8 micko.y-8 113
5.9 micko.y-9 114
5.10 micko.y-10 114
5.11 micko.y-11 115

Slike

1.1 Faze kompajliranja 5

3.1 Stablo parsiranja za izraz `2+3` 36
3.2 Stablo parsiranja za ulaz `2+3\n` 37
3.3 Deo stabla parsiranja za izraz `2+3-4+5` 38
3.4 Primer rada parsera za ulaz `1+2\n` 43
3.5 Dva moguća stabla parsiranja za primer `2+3*4` 46
3.6 Primer konflikta za izraz `2+3*4` 47

4.1 Stek frejm 74
4.2 Tabela simbola 75

5.1 C blokovi 105
5.2 Konverzija tipova 107
5.3 Različite konverzije tipova 107

Indeks

.l, 9
.output, 46
.y, 33

arg_num, 93

back end, 5
bison, 7, 33
  $$, 41
  $1, 41
  .output, 46
  .y, 33
  asocijativnost, 47
  `error`, 44
  opcije
    -d, 40
    -v, 46
  oporavak od greške, 44
  parser, 36
  pravila, 34
  prioritet operatora, 48
  reduce, 42
  shift, 42
  specifikacija, 33, 34
  union, 59
  yyerrok, 45
  yyerror(), 40
BNF notacija, 4

ciljni jezik, 3, 87

false_lab_num, 95, 98, 109, 113
faze kompajliranja, 5
flex, 7, 9
  .l, 9
  funkcije
    noyywrap, 13
    yywrap(), 11
  opcije, 10
    noyywrap, 10, 13
    yylineno, 11
  promenljive
    yylineno, 11
    yylval, 17, 18
    yytext, 18
  specifikacija, 10, 13
free_reg_num, 91
frejm, 73
front end, 5

generatori
  kompajlera, 6
  parsera, 7
  skenera, 6
generisanje koda, 5, 87
generisanje međukoda, 4
gramatika, 4, 31
  dvosmislena, 45, 46
  LL, 31
  LR, 31
greška
  leksička greška, 11, 27
  semantička greška, 71
  sintaksna greška, 31

hipotetski asemblerski jezik, 7

identifikator, 71

dinamični, 73
globalni, 72, 73
lokalni, 72, 73
`main`, 71
statični, 73
izvorni jezik, 3

kompajler
implementacija, 6
kompajliranje, 3
konflikt, 45
razrešenje, 45

lab_num, 95, 98, 109, 113
label_stack, 109, 114
labela, 87
leksička analiza, 4, 9
leksička greška, 11, 27
leksički analizator, 4, 9
lex.yy.c, 9, 13, 40
LR parser
accept, 33
error, 33
reduce, 32
shift, 32

mC
kompajler, 7
opseg vidljivosti, 72
organizacija memorije, 73
programski jezik, 7
semantička provera, 79
MICKO, 108

opseg vidljivosti, 72, 74–76, 82
ugnježden, 72
output, 109
output.asm, 109, 114

parser, 6, 31
bottom-up, 31
konflikt, 45, 46
LALR(1), 45
LL parser, 32
lookahead token, 45
LR parser, 32
oporavak od greške, 44
reduce, 42
shift, 42
top-down, 31
parsiranje, 4, 31
prolaz, 5
prostor imena, 74

radni registar, 87, 91
free_reg_num, 91
oslobađanje, 91
zauzimanje, 91
registar %13, 87, 92, 112, 114
registar %14, 87, 109
registar %15, 87, 89
rekurzivno pravilo, 38, 51
rukovanje greškama, 6

scope, 72, 74
semantička analiza, 4, 71
semantička greška, 71
semantička pravila, 71
semantika, 4, 71
sintaksa, 3, 31
sintaksna analiza, 4, 31
sintaksna greška, 31
skener, 9
skeniranje, 9
stablo parsiranja, 4, 31, 36, 37
stek frejm, 73
stek labela, 98, 109, 114

tabela simbola, 5, 73
efikasnost, 76
implementacija, 75
informacije, 73
operacije, 74

token, 17
token `error`, 44

unija, 18, 59, 62, 65

`var_num`, 81
var_num, 92, 110

yyerrok, 45
yyerror(), 40
yylex(), 9
yyparse(), 68

Bibliografija

[1] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers, principles, techniques, and tools*. Addison-Wesley series in computer science. Addison-Wesley Pub. Co., 1986.

[2] Stephen C. Johnson. Yacc: Yet another compiler-compiler. Technical report, 1979.

[3] M. E. Lesk and E. Schmidt. Unix vol. ii. chapter Lex - a lexical analyzer generator, pages 375–387. W. B. Saunders Company, Philadelphia, PA, USA, 1990.

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

[5] Torben Ægidius Mogensen. *Basics of Compiler Design*. lulu.com, Copenhagen, DK, 2009.

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

[7] M.L. Scott. *Programming Language Pragmatics*. Programming Language Pragmatics. Elsevier Books, Oxford, 2000.