Projektna dokumentacija

Tema projekta bila je implementacija jednostavnog kompajlera. Implementiran je korišćenjem flex-a i bison-a. Za potrebe projekta kao izvorni jezik korišćen je miniC(uproštena verzija C programskog jezika) a kao ciljni jezik hipotetski asemblerski jezik(uproštena verzija x86 asemblera). Implementacija miniC kompajlera je podeljena na uobičajene faze kompajliranja: leksičku, sintaksnu, semantičku analizu i generisanje koda.

Leksička analiza

Leksički analizator (skener) je program namenjen za leksičku analizu (skeniranje) teksta. Zadatak skenera je da pročita ulazni tekst i da ga podeli na simbole. Skener radi tako što čita tekst, karakter po karakter, i pokušava da prepozna neku od zadatih reči. Ukoliko naiđe na simbol čija pojava nije predviđena, treba da prijavi leksičku grešku. Beline (u koje spadaju prazno mesto, tabulator i karakter za novi red) se preskaču, odnosno prepoznaju i ignorišu, jer imaju ulogu separatora simbola. Za izradu ovog projekta za generisanje skenera koristi se program flex. Flex generiše skener na osnovu pravila zadatih u flex specikaciji. Specikaciju kreira korisnik u posebnoj datoteci koja po konvenciji ima ekstenziju **.l**. Ova datoteka sadrži pravila koja opisuju reči koje skener treba da prepozna. Pravila se zadaju u obliku regularnih izraza. Svakom regularnom izrazu moguće je pridružiti akciju (u obliku C koda) koja će se izvršiti kada skener prepozna dati regularni izraz. Uobičajena akcija je vraćanje oznake simbola, odnosno tokena za taj simbol. Token je numerička oznaka klase (vrste) simbola. Na primer: token **NUMBER** označava bilo koji broj (i broj 2 i broj 359), dok token **IF** označava jedino ključnu reč if. 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. Skener za miniC jezik je implementiran u datoteci **micko.l**. Prvi deo specikacije započinje uključivanjem dve opcije koje utiču na to kako će flex napraviti skener:

```
% option noyywrap yylineno
```

noyywrap opcija znači da će skener skenirati samo jednu datoteku a **yylineno** opcija znači da će skener koristiti globalnu promenljivu flex-a **yylineno** koja sadrži broj trenutno skenirane linije. Za neke simbole se, osim

tokena, prosleđuje još neka vrednost koja bliže opisuje simbol (npr. za simbol broj se uz token **NUMBER** šalje i vrednost konkretnog broja, recimo 123). U ovakvim situacijama, za smeštanje vrednosti, koja može biti različitog tipa - za različite simbole, se koristi unija. Za miniC skener, potrebna je unija koja sadrži jedan celobrojni tip (**int i**) i jedan pokazivač na karakter (**char *s**). Promenljiva tipa unije, preko koje se, uz token, prosleđuje vrednost, se u flex-u zove **yylval**. Niz pravila koja su definisana u skeneru:

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

"int"                   { yylval.i = INT;  return _TYPE; }
"unsigned"              { yylval.i = UINT; return _TYPE; }
"if"                    { return _IF; }
"else"                  { return _ELSE; }
"return"                { return _RETURN; }


"for"                   { return _FOR; }
"++"                    { return _INC; }
"while"                 { return _WHILE; }
"?"                     { return _QMARK; }
":"                     { return _COLON; }

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

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

"<"                     { yylval.i = LT; return _RELOP; }
">"                     { yylval.i = GT; return _RELOP; }
"<="                    { yylval.i = LE; return _RELOP; }
">="                    { yylval.i = GE; return _RELOP; }
"=="                    { yylval.i = EQ; return _RELOP; }
"!="                    { yylval.i = NE; return _RELOP; }

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

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

Prvo pravilo preskače beline. Pravila koja se nalaze u crvenom kvadratu prepozanju ključne reči **if**, **else**, **return**, **for**, **while**, delimitere **(**, **)**, **{**, **}**,

separatore **;**, operator dodele **=** i simbole ternarnog operatora **?** i **:** i za sve njih je dovoljno vezati samo tokene(nisu potrebne dodatne vrednosti koje bi ih detaljnije denisale). Dva pravila iznad kvadrata prepoznaju označene i neoznačene celobrojne tipove. Za oba simbola se vezuje isti token (**_TYPE**), ali različite vrednosti uz token. Vrednosti su konstante (**INT** i **UINT**) koje opisuju prepoznate tipove. Vrednost se prosleđuje kroz promenljivu **yylval**. Kako je konstanta celobrojnog tipa, njena vrednost se smešta u polje **i** promenljive **yylval**. Zatim slede pravila za prepoznavanje aritmetičkih i relacionih operatora. Za ove operatore se, osim tokena, prosleđuje još i odgovarajuća konstanta kao vrednost simbola. Konstanta je potrebna da bi se operatori razlikovali međusobno, jer imaju isti token (na primer: konstanta **ADD** za operator **+**). Zatim slede pravila za prepoznavanje identikatora i literala. U ovim slučajevima se, pored tokena, prosleđuje još i string simbola, koji se smešta u polje **s** (tipa **char***) promenljive **yylval**. String prepoznatog simbola flex čuva u promenljivoj **yytext**. Regularni izraz za identikator opisuje string koji počinje slovom (malim ili velikim: [a-zA-Z], a iza njega se može naći slovo (malo ili veliko) ili cifra, 0 ili viće puta: [a-zA-Z0-9]*. Minimalni identikator se, znači, sastoji od jednog slova. Regularni izraz za označeni celobrojni literal opisuje string od minimalno jedne do maksimalno deset cifara: [0-9]{1,10}, ispred kojih se može, a ne mora, naći predznak plus ili minus: [+-]?. Regularni izraz za neoznačeni celobrojni literal ne prepoznaje predznak, već dozvoljava da se iza broja navede malo ili veliko slovo **u**: [uU]. Linijski komentari se prepoznaju pomoću regularnog izraza koji prepoznaje dva znaka slash na početku stringa: \/\/ (može i: //), a zatim bilo koji znak osim \n znaka, 0 ili više puta: **.***. Poslednje pravilo koje sadrži operator **.** (bilo koji znak osim \n znaka) služi da prepozna leksičku grešku. Bilo koji znak koji ne pripada nijednom prethodnom pravilu predstavlja grešku, koju treba saopštiti korisniku.

Sintaksna analiza

Sintaksa jezika opisuje pravila po kojima se kombinuju simboli jezika (npr. u deklaraciji promenljive, prvo se piše ime tipa, zatim ime promenljive i na kraju ;). Sintaksa se opisuje gramatikom, obično pomoću BNF notacije. Sintaksna analiza ima zadatak da proveri da li je ulazni tekst sintaksno ispravan. Ona to čini tako što preuzima niz tokena od skenera i proverava da li su tokeni navedeni u ispravnom redosledu. Ako jesu, to znači da je ulazni tekst napisan u skladu sa pravilima gramatike korišćenog jezika, tj. da je sintaksno ispravan. U suprotnom, sintaksna analiza treba da prijavi sintaksnu grešku i da nastavi analizu. Opisani proces se vrši u delu kompajlera koji se zove parser i naziva

se parsiranje. Za implementaciju parsera, često se koristi generator parsera **bison**. Prvi korak u generisanju parsera je priprema njegove specikacije. Specikaciju 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. Prilikom izvršvanja, parser komunicira sa skenerom tako što od njega traži sledeći token iz ulaznog teksta. Kada dobije token, parser proverava da li je njegova pojava na datom mestu dozvoljena (tj. da li je u skladu sa gramatikom). Ako jeste nastaviće parsiranje, a ukoliko nije, prijaviće grešku, pokušaće da se oporavi od greške i da nastavi parsiranje. U momentu kada prepozna celo pravilo, parser će izvršiti akciju koja je pridružena tom pravilu. Parser ya miniC jeyik implementiran je u datoteci **micko.y**. Sve konstante koje su potrebne skeneru i parseru su smeštene u posebnoj datoteci **defs.h** radi lakšeg održavanja koda.

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

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

//konstante arithmetickih operatora
enum arops { ADD, SUB, MUL, DIV, AROP_NUMBER };
//stringovi za generisanje aritmetickih naredbi
static char *ar_instructions[] = { "ADDS", "SUBS", "MULS", "DIVS",
                                   "ADDU", "SUBU", "MULU", "DIVU" };

//konstante relacionih operatora
enum relops { LT, GT, LE, GE, EQ, NE, RELOP_NUMBER };
//stringovi za JMP naredbu
static char* jumps[]={"JLTS", "JGTS", "JLES", "JGES", "JEQ ", "JNE ",
                      "JLTU", "JGTU", "JLEU", "JGEU", "JEQ ", "JNE " };

static char* opp_jumps[]={"JGES", "JLES", "JGTS", "JLTS", "JNE ", "JEQ ",
                          "JGEU", "JLEU", "JGTU", "JLTU", "JNE ", "JEQ "};
```

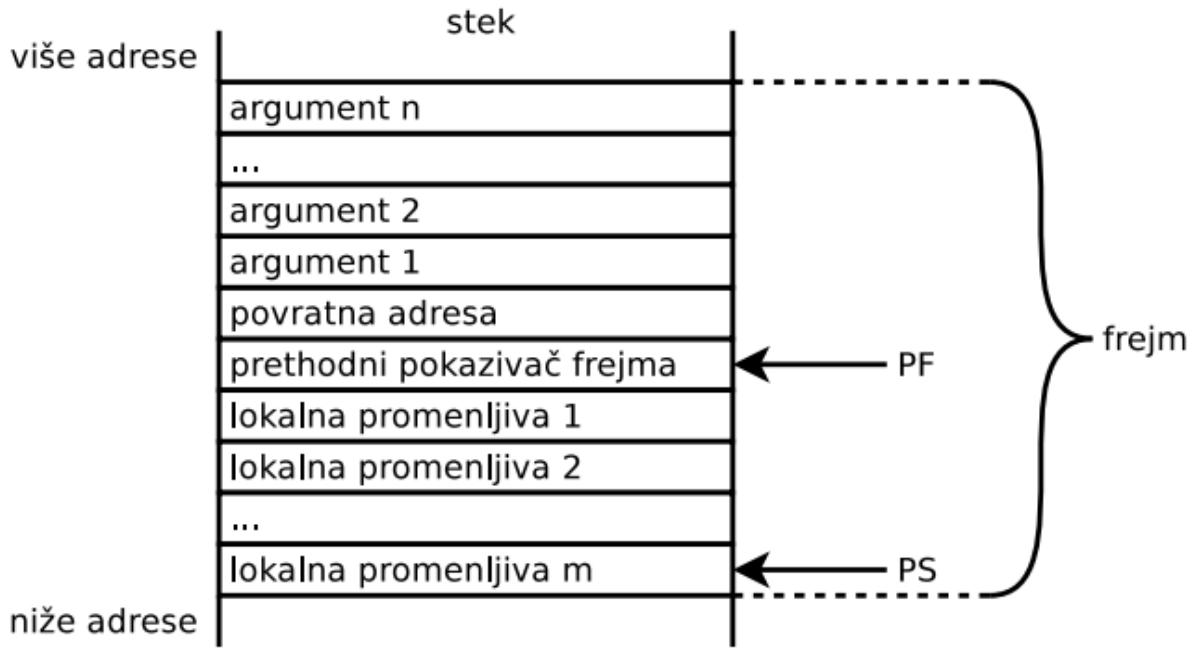
Enumeracije uokvirene crvenom bojom su bitne za implementaciju parsera. Ostale enumeracije su potrebne za generisanje asemblerskog koda osim enumeracije **kinds** koja se koristi u semnatičke svrhe. Enumeracija **types** sadrži konstante koje opisuju tipove. miniC jezik podržava samo **int** i **unsigned** tipove. Enumeracija **arops** sadrži konstante koje opisuju aritmetičke operatore sabiranja, oduzimanja, množenja i deljenja, a enumeracija **relops** sadrži konstante koje opisuju relacione operatore (<, >, <=, >=, == i !=). Tokeni koje šalje skener, a koje prima parser, se moraju

denisati pomoću ključne reči **%token**. Uz svaki token se može denisati tip vrednosti koja se prosleđuje sa njim, između operatora **<i>**. Tokeni koji imaju vrednosti celobrojnog tipa su **_TYPE**, **_AROP** i **_RELOP** (njihova vrednost je konstanta koja opisuje vrstu tipa, aritmetičkog, odnosno relacionog operatora), a tokeni koji imaju vrednost tipa string su **_ID**, **_INT_NUMBER** i **_UINT_NUMBER** (njihova vrednost je string imena ili broja). Funkcija main() jedino poziva parser (tj. funkciju **yyparse()**). Čim naiđe na prvu sintaksnu grešku, parser će završiti parsiranje.

Semantička analiza

Semantička analiza je faza kompajliranja u kojoj se proverava značenje (semantika) programskog teksta (koda). Kod koji je sintaksno ispravan, ne mora biti i semantički ispravan. Na primer, ako se promenljiva tipa int poredi sa promenljivom tipa boolean, kompajler treba da prijavi semantičku grešku. Semantička analiza se, najčešće, implementira u parseru, tako što se pravilima dodaju semantičke provere. Za ovo je potrebno poznavati organizaciju memorije i arhitekturu računara za koju se generiše kod. Globalni identikatori su statični postoje za sve vreme izvršavanja programa i za njih se mogu rezervisati memorijske lokacije u toku kompajliranja. Lokalni identikatori 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 identikatorima dodeljuju memorijske lokacije sa steka. Deo steka koji se zauzima za izvršavanje neke funkcije se zove (stek) frejm. Stek frejm, redom, sadrži:

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



Pokazivač frejma **PF** pokazuje na početak frejma, a pokazivač **PS** pokazuje na vrh steka. **PF** ostaje isti, dok se **PS** menja tokom izvršavanja funkcije. Pošto će generator koda generisati kod na hipotetskom asemblerskom jeziku, potrebno je upoznati se sa arhitekturom naredbi i registara. Ukupno ima 16 registara. Oznaka registra se sastoji od oznake % i rednog broja registra: %0, %1, ..., %15. Registri od %0 do %12 imaju opštu namenu i služe kao radni registri. Po konvenciji, registar %13 je rezervisan za povratnu vrednost funkcije, registar %14 služi kao pokaziva£ frejma (**PF**), a registar %15 služi kao pokazivač steka (**PS**). Tabela simbola (symbol table) je struktura podataka koja čuva identikatore iz kompajliranog programa i sve informacije o tim identikatorima. Za semantičku analizu, ona je neophodna, jer sadrži sve informacije na osnovu kojih se mogu sprovesti semantičke provere. Na primer, u seledećem programu, postoji samo main() funkcija, sa jednim parametrom i jednom lokalnom promenljivom:

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

Tokom kompajliranja ove funkcije, u tabelu simbola treba zapisati da postoji:

- identikator **main** koji predstavlja funkciju, čiji povratni tip je **int** i koja ima jedan parametar tipa **int**
- identikator p tipa **int**, koji predstavlja parametar, prvi po redu
- identikator x tipa **int**, koji predstavlja lokalnu promenljivu, prvu po redu
- literal 100 tipa **int**.

Za ovaj program, sadržaj tabele simbola (na kraju parsiranja funkcije main()) bi bio:

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

I globalni i lokalni identikatori mogu biti smešteni u istu tabelu simbola. Globalni su prisutni u tabeli simbola sve vreme kompajliranja, dok su lokalni prisutni samo u toku kompajliranja funkcije kojoj pripadaju. Izlaskom iz opsega vidljivosti, iz tabele simbola se brišu identikatori vidljivi (samo) u tom opsegu. Na primer, na kraju parsiranja funkcije, brišu se svi lokalni identikatori funkcije. Pošto u tabeli simbola istovremeno mogu postojati identični globalni i lokalni identikatori, radi njihovog razlikovanja, u tabeli mora biti naznačena vrsta identikatora (da li je funkcija, promenljiva, parametar, argument, ...). Za svaki identikator mora postojati i oznaka njegovog tipa (da li je u pitanju tip int, unsigned, char, ...). Za parametre i lokalne promenljive mora postojati njihova relativna pozicija na steku u odnosu na početak frejma, tj. njihov redni broj (ova informacija je potrebna za generisanje koda). Takođe, radi uniformnosti, u tabeli simbola mogu biti smeštene i oznake radnih registara. One se smeštaju u tabelu u vreme njene inicijalizacije. Registar %0 se smešta u element sa indeksom 0, .., a %12 u element sa indeksom 12. Registri su prisutni u tabeli simbola sve vreme kompajliranja. Registri se koriste u fazi generisanja koda, kada se u njih smeštaju međurezultati izraza. Koristi se vrlo jednostavna implementacija tabele simbola: tabela je organizovana kao niz struktura koje opisuju osobine identikatora. U datoteci **defs.h** nalazi se enumeracija **kinds** koja sadrži konstante koje opisuju vrste simbola. Vrednosti

konstanti su stepeni broja 2 (1, 2, 4, 8, ... tj. sadrže samo po 1 setovan bit) da bi se nad njima mogla primeniti bitwise logika. U ovu datoteku smeštena su i dva makroa koja olakšavaju prijavu greške ili upozorenja: **err(args ...)** i **warn(args ...)**, kao i nekoliko konstanti konstante za simulaciju bool tipa (**TRUE, FALSE**), dužina tabele simbola, tj. broj elemenata tabele: **SYMBOL_TABLE_LENGTH**, dužina bafera za smeštanje poruka o greškama: **CHAR_BUFFER_LENGTH**, oznaka da simbol nema atribut **NO_ATR**, redni broj poslednjeg radnog registra **LAST_WORKING_REG**, redni broj registra koji čuva povratnu vrednost funkcije **FUN_REG**.

```
# define bool int
# define TRUE 1
# define FALSE 0
# define SYMBOL_TABLE_LENGTH 64
# define NO_ATR 0
# define LAST_WORKING_REG 12
# define FUN_REG 13
# define CHAR_BUFFER_LENGTH 128
```

Struktura jednog elementa tabele simbola se sastoji od polja(datoteka **symtab.h**):

- **name** - string identikatora
- **kind** - vrsta simbola - sadrži vrednost iz enumeracije **kinds**
- **type** - tip simbola - sadrži vrednost iz enumeracije **types**
- **atr1** - atribut - sadrži različite vrednosti za različite vrste simbola
    - za lokalnu promenljivu, u ovom se smešta redni broj promenljive
    - za parametar, u ovom polju se smešta redni broj parametra
    - za funkciju, u ovom polju se čuva broj parametara
    - za ostale identikatore, ovo polje nije popunjeno.
- **atr2** - atribut - sadrži različite vrednosti za različite vrste simbola
    - za funkciju, u ovom polju se čuva tip parametra (vrednost iz enumeracije **types**

```
// Element tabele simbola
typedef struct sym_entry {
   char *    name;          // ime simbola
   unsigned kind;           // vrsta simbola
   unsigned type;           // tip vrednosti simbola
   unsigned atr1;           // dodatni attribut simbola
   unsigned atr2;           // dodatni attribut simbola
} SYMBOL_ENTRY;
```

Operacije za rad sa tabelom simbola su implementirane u datotekama **symtab.h** i **symtab.c**. Funkcija koja služi za ubacivanje jednog simbola u tabelu simbola se zove **insert_symbol()**. Parametri funkcije su string, vrsta, tip i atributi simbola. Funkcija vraća indeks ubačenog elementa u tabeli simbola.

int insert_symbol (char * name , unsigned kind , unsigned type , unsigned atr1 , unsigned atr2) ;

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

int insert_literal (char * str , unsigned type) ;

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

int lookup_symbol (char * name , unsigned kind) ;
Set i get metode kojima se menjaju i čitaju vrednosti određenih polja elementa tabele simbola su:

void set_name (int index , char * name) ;
char * get_name (int index) ;
void set_kind (int index , unsigned kind) ;
unsigned get_kind (int index) ;
void set_type (int index , unsigned type) ;
unsigned get_type (int index) ;
void set_atr1 (int index , unsigned atr1) ;
unsigned get_atr1 (int index) ;
void set_atr2 (int index , unsigned atr2) ;
unsigned get_atr2 (int index) ;

Funkcija koja briše deo tabele simbola i koja se koristi za brisanje lokalnih simbola funkcije se zove **clear_symbols()**. Parametar ove funkcije je indeks

prvog elementa koji treba da se obriše. Funkcija će obrisati sve simbole koji se nalaze između ovog indeksa i poslednjeg popunjenog elementa tabele simbola.

void clear_symbols (unsigned begin_index);

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

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

miniC parser sa semaničkim proverama koristi prethodno opisanu tabelu simbola a impelementiran je u datoteci **micko.y**. Prvi deo specikacije parsera je proširen denicijama promenljivih koje koristi parser u toku provere semantike. To su broj grašaka **error_count**, broj upozorenja **warning_count**, broj lokalnih promenljivih **var_num**, indeks u tabeli simbola na kom se nalazi simbol trenutno parsirane funkcije **fun_idx** i indeks u tabeli simbola na kom se nalazi ime funkcije čiji poziv se trenutno parsira **fcall_idx**:

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

U nastavku su denisani i tipovi pojmova: iza ključne reči **%type** se navodi tip iz unije (**<i>** ili **<s>**), a zatim imena svih pojmova koji će biti tog tipa. Ako je tip pojma **<i>** to znači da će njegova vrednost biti celobrojnog tipa (**int**), a ako je tipa **<s>** onda će njegova vrednost biti tipa string (**char ***).

%type <i> num_exp exp literal
%type <i> function_call argument rel_exp if_part cond_exp

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

%nonassoc ONLY_IF
%nonassoc _ELSE

Pravila u drugom delu specikacije parsera su proširena akcijama koje sadrže semantičke provere. Semantička provera, koju treba obaviti nakon prepoznavanja celog programa, je da li postoji **main()** funkcija. Provera se vrši pretragom tabele simbola. Ako se u tabeli ne pronađe simbol main koji predstavlja funkciju (**FUN**), prijavi se semantička greška.

```
program
  : function_list
      {
        if(lookup_symbol("main", FUN) == NO_INDEX)
          err("undefined reference to 'main'");
      }
  ;
```

Tokom parsiranja denicije 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 tokena **_TYPE** koja nosi konstantu **INT** ili **UINT**) i vrsti identikatora (konstanta **FUN**). Promenljiva **fun_idx** dobija indeks elementa u tabeli simbola u koji je smešteno ime funkcije. Nakon parsiranja celog tela funkcije (nakon prepoznavanja pojma body) iz tabele simbola se brišu svi simboli koji su lokalni za funkciju. Ovo sme da se radi (i potrebno je), jer je opseg vidljivosti lokalnih simbola samo do kraja funkcije u kojoj su deklarisani. Ovim je omogućeno da svaka funkcija ima svoje lokalne promenljive, koje mogu biti (potpuno) iste kao u nekoj drugoj funkciji. Funkcija **clear_symbols()**, koja briše lokalne simbole, se poziva sa argumentom **fun_idx+1**, jer brisanje tabele simbola treba početi nakon imena funkcije. Ime funkcije je globalni identikator i zato treba da bude prisutan u tabeli simbola tokom parsiranja celog programa. Promenljiva **var_num**, koja broji lokalne promenljive jedne funkcije, se resetuje.

```
function
  : _TYPE _ID
      {
        fun_idx = lookup_symbol($2, FUN);
        if(fun_idx == NO_INDEX)
          fun_idx = insert_symbol($2, FUN, $1, NO_ATR, NO_ATR);
        else
          err("redefinition of function '%s'", $2);
      }
    _LPAREN parameter _RPAREN body
      {
        clear_symbols(fun_idx + 1);
        var_num = 0;
      }
  ;
```

Nakon parsiranja pojma parameter, ime parametra (vrednost **$2**) se ubacuje u tabelu simbola kao **PAR** sa tipom koji nosi token **_TYPE** i sa rednim brojem 1 (polje **atr1** parametra čuva redni broj parametra). Tip parametra se zapisuje u polju **atr2** simbola funkcije u tabeli simbola (jer polje **atr2** za funkciju čuva tip njenog parametra). Ako nije bilo parametara u polje **atr1** smešta se podatak o broju parametara funkcije koji je u ovom slučaju 0, a ukoliko je detektovan jedan parametar u polje **atr1** smešta se 1.

```
parameter
  : /* empty */
      { set_atr1(fun_idx, 0); }

  | _TYPE _ID
      {
        insert_symbol($2, PAR, $1, 1, NO_ATR);
        set_atr1(fun_idx, 1);
        set_atr2(fun_idx, $1);
      }
  ;
```

Tokom parsiranja deklaracije promenljive (pojam **variable**), ime promenljive (vrednost **$2**) se ubacuje u tabelu simbola kao **VAR** sa tipom koji nosi token **_TYPE**. Vrednost brojača lokalnih promenljivih se inkrementira i zapisuje se kao atribut **atr1** ovog simbola u tabeli simbola, zato što lokalna promenljiva na mestu prvog atributa u tabeli simbola čuva svoj redni broj. Ukoliko je u tabeli simbola pronađen parametar sa istim imenom, prijavljuje se semantička greška zato što dva lokalna identikatora unutar funkcije ne mogu imati isto ime.

```
variable
  : _TYPE _ID _SEMICOLON
    {
      if(lookup_symbol($2, VAR|PAR) == NO_INDEX)
        insert_symbol($2, VAR, $1, ++var_num, NO_ATR);
      else
        err("redefinition of '%s'", $2);
    }
  ;
```

Nakon parsiranja iskaza dodele(pojam **assignment_statement**), proverava se da li je identikator koji se nalazi sa leve strane jednakosti promenljiva ili parametar. Ako nije, prijavljuje se semantička greška, jer je na tom mestu dozvoljeno samo ime promenljive ili parametra (a ne, recimo, ime funkcije). Dodatno, vrši se i provera tipova leve i desne strane jednakosti, jer bi oni trebali biti isti. Za preuzimanje tipova koristi se funkcija **get_type()**. Njoj treba predati indeks u tabeli simbola onog simbola čiji tip treba da preuzme (u iskazu dodele to su: indeks pronađenog identikatora sa leve strane jednakosti i indeks u tabeli simbola gde se nalazi rezultat numeričkog izraza sa desne strane jednakosti (vrednost **$3**)).

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

Nakon parsiranja prvog dela for petlje(pojam **for_statement**), proverava se da li je identikator promenljiva ili parametar. Ako nije, prijavljuje se semantička greška, jer je na tom mestu dozvoljeno samo ime promenljive ili parametra. Dodatno, vrši se i provera tipova leve i desne strane jednakosti, jer bi oni trebali biti isti. Preuzimanje tipova vrši se na isti način kao i kod **assignment_statement**. U poslednjem delu for petlje, ponovo se proverava da li je identikator promenljiva ili parametar i ako nije prijavljuje se semantička greška.

```
for_statement
  : _FOR _LPAREN _ID _ASSIGN literal
    {
      int i = lookup_symbol($3, VAR|PAR);
      if(i == NO_INDEX)
        err("'%s' undeclared", $3);

      if(get_type(i) != get_type($5))
        err("incompatible types");
    }
  _SEMICOLON rel_exp
  _SEMICOLON _ID _INC _RPAREN statement
    {
      int i = lookup_symbol($11, VAR|PAR);
      if(i == NO_INDEX)
        err("'%s' undeclared", $11);
    }
  ;
```

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 **get_type()** sa argumentima: indeks prvog operanda u tabeli simbola (**$1**) i indeks drugog operanda u tabeli simbola (**$3**).

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

Nakon parsiranja identikatora u izrazima (**exp**), vrši se provera postojanja dotičnog imena, tj. njegovog prisustva u tabeli simbola. Ovu proveru obavlja funkcija **lookup_symbol()**. Ako ga nema u tabeli, znači da nikada nije deklarisan, pa treba prijaviti semantičku gršku, jer se ne može koristiti promenljiva koja nije prethodno deklarisana. Ako se izraz nalazi u zagradama (pretposlednja varijanta pravila **exp**), njegova vrednost treba da postane i vrednost celog izraza. U slučaju proslednjeg pravila **exp** tu se proveravaju tipovi operanada, pa ako nisu isti, prijavljuje se semantička greška. Provera se vrši isto kao i kod **num_exp**.

```
exp
  : literal
  | _ID
    {
      $$ = lookup_symbol($1, VAR|PAR);
      if($$ == NO_INDEX)
        err("'%s' undeclared", $1);
    }
  | function_call
  | _LPAREN num_exp _RPAREN
    { $$ = $2; }
  | _LPAREN rel_exp _RPAREN _QMARK cond_exp _COLON cond_exp
    {
      if(get_type($5) != get_type($7))
        err("expressions don't have same types");
    }
  ;
```

Nakon parsiranja identikatora u izrazima (**cond_exp**), vrši se provera postojanja dotičnog imena, tj. njegovog prisustva u tabeli simbola. Ova provera obavlja se isto kao i kod **exp**.

```
cond_exp
  : _ID
    {
      if(($$ = lookup_symbol($1, VAR|PAR)) == NO_INDEX)
        err("'%s' udeclared", $1);
    }
  | literal
  ;
```

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

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

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

Tokom parsiranja poziva funkcije (pojam **function_call**) proverava se da li je navedeni identikator ime neke od postojećih funkcija, pa ako nije, prijavljuje se greška da se poziva nepostojeća funkcija. Kako je usvojeno, povratna vrednost funkcije će biti smeštena u registru **%13**, pa je potrebno postaviti vrednost pojma **function_call** na indeks 13 u tabeli simbola (gde se nalazi

registar **%13**). Registru **%13** se postavlja tip povratne vrednosti funkcije koja se kasnije koristi u izrazima.

```
function_call
  : _ID
    {
      fcall_idx = lookup_symbol($1, FUN);
      if(fcall_idx == NO_INDEX)
        err("'%s' is not a function", $1);
    }
    _LPAREN argument _RPAREN
    {
      if(get_atrl(fcall_idx) != $4)
        err("wrong number of arguments");
      set_type(FUN_REG, get_type(fcall_idx));
      $$ = FUN_REG;
    }
  ;
```

Tokom parsiranja argumenta, treba proveriti da li je njegov tip isti kao tip odgovarajućeg parametra. Ako nije, treba prijaviti grešku. Tip parametra se dobija pozivom funkcije **get_atr2(fcall_idx)**, dok se tip argumenta dobija pozivom funkcije **get_type($1)** (**$1** je indeks elementa u tabeli simbola koji sadrži rezultat numeričkog izraza koji je prosleđen na mestu argumenta). Vrednost pojma **argument** dobija vrednost 1 koja označava da je to prvi po redu (i jedini) argument. Ukoliko argumenta nema, vrednost ovog pojma postaje 0 (oznaka da nema argumenata).

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

  | num_exp
    {
      if(get_atr2(fcall_idx) != get_type($1))
        err("incompatible type for argument");
      $$ = 1;
    }
  ;
```

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

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

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

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

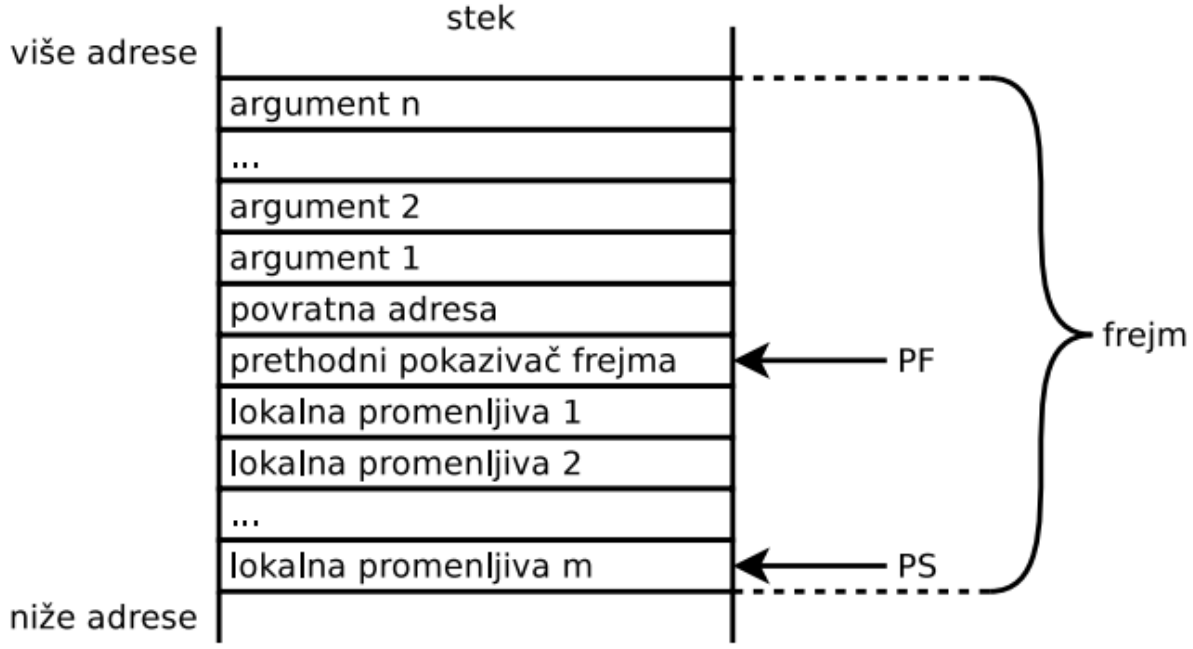
Funkcije **yyerror()** i **warning()** služe za prijavljivanje greške ili upozorenja korisniku.

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

Generisanje koda

Generisanje koda je faza kompajliranja u kojoj se proizvodi datoteka sa ekvivalentnim programom napisanim na ciljnom programskom jeziku. Ciljni jezik je jezik niskog nivoa, i može biti neki mašinski ili asemblerski jezik. Ciljni jezik na koji će u projektu biti preveden miniC programski jezik, je hipotetski asemblerski jezik. Ovaj asemblerski jezik je vrlo jednostavan. 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 zapo£inju malim slovom iza koga mogu da slede mala slova, cifre i

podcrta `_' (alfabet je 7 bitni ASCII). Iza labele se navodi dvota£ka, a ispred sistemskih labela se navodi znak `@'. Potrebno je imati šeme ekvivalencije. To su šabloni koji opisuju kako se određeni iskazi miniC jezika pišu na hipotetskom asemblerskom jeziku. Ovi šabloni ne smeju izmeniti značenje polaznog miniC iskaza. Primeri (mogućih) ekvivalencija između miniC koda i hipoteskog asemblerskog koda, napisani su u dve kolone. U prvoj koloni se nalazi miniC kod koji se prevodi, a u drugoj se nalazi ekvivalentni asemblerski kod. Primeri sadrže i predloge načina generisanja ekvivalentnog koda. U svim navedenim primerima se podrazumeva da su **a**, **b** i **c** celobrojne označene lokalne promenljive, deklarisane tim redom. Na osnovu slike ispod, vidi se da se lokalne promenljive smeštaju ispod pokazivača frejma **%14**. Za pristup ovim lokacijama koristi se indeksno adresiranje pomoću registra **%14**. Tako je lokacija prve promenljive: **-4(%14)** a druge **-8(%14)** (memorijsku lokaciju adresira zbir vrednosti broja i sadržaja registra). Lokacije argumenata počinju na drugoj lokaciji iznad pokazivača frejma (neposredno iznad pokazivača frejma se nalazi povratna adresa), pa se prvom argumentu pristupa na adresi **8(%14)**, drugom na adresi **12(%14)**, itd.



Primeru iskaza pridruživanja odgovara izgenerisani kod:

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

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

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

U prvoj naredbi **SUBS** se zauzima radni registar **%0**, a u drugoj registar **%1**. U naredbi **ADDS** (linija 3) se oslobađaju radni registri **%0** i **%1**, a ponovo se zauzima radni registar **%0**. U naredbi **SUBS** (linija 4) se oslobađa i ponovo zauzima radni registar **%0**, a u naredbi **MOV** se oslobađa radni registar **%0**. Radni registar se zauzima za smeštanje rezultata svakog aritmetičkog izraza (**num_exp**). Radni registar se oslobađa čim se preuzme njegova vrednost. Pošto se međurezultati izraza koriste u suprotnom redosledu od onog u kome su izračunati, radni registri, koji se koriste za smeštanje međurezultata, se zauzimaju i oslobađaju po principu steka. Kao "pokazivač steka registara" koristi se promenljiva **free_reg_num**, koja sadrži broj prvog slobodnog radnog registra. Zauzimanje radnog registra se sastoji od preuzimanja vrednosti promenljive **free_reg_num** i njenog inkrementiranja, a oslobađanje radnog registra se sastoji od dekrementiranja ove promenljive. Treba napomenuti da je broj registra istovremeno i indeks elementa tabele simbola. Broj zauzetog radnog registra služi kao vrednost sintaksnog pojma (**num_exp**) koji odgovara izrazu čiji rezultat radni registar sadrži. Prekoračenje broja radnih registara (**free_reg_num > 12**) predstavlja fatalnu grešku u radu kompajlera. Za smeštanje povratne vrednosti funkcije se koristi registar **%13**. Funkcija, nakon poziva, lokalne promenljive i argumente čuva na stek frejmu. Primer ekvivalencije za deniciju funkcije:

```
int f(int p) {
    int a;
    return p + a;
}
```

```
f:
    PUSH %14
    MOV  %15,%14
    SUBS %15,$4,%15
@f_body:
    ADDS 8(%14),-4(%14),%0
    MOV  %0,%13
    JMP  @f_exit
@f_exit:
    MOV  %14,%15
    POP  %14
    RET
```

Ekvivalentni asemblerski kod za funkciju započinje labelom koja ima isti identikator kao funkcija, u ovom slučaju **f:**. Naredbom **PUSH** na stek se smešta stari pokazivač frejma, dok se pomoću naredbe **MOV** postavlja novi pokazivač frejma (registar **%14** sada pokazuje na vrh steka, kao i **%15**). Pomoću naredbe **SUBS** se zauzima prostor na steku za lokalnu promenljivu a (pokazivač steka se pomera jednu lokaciju, tj. 4 bajta niže). Promenljiva **var_num** služi kao brojač lokalnih promenljivih. Veličina prostora za lokalne promenljive se određuje kao **var_num * 4**. Prostor se zauzima samo ako funkcija ima lokalnih promenljivih (**var_num > 0**). Ekvivalentni asemblerski kod tela funkcije započinje labelom koja sadrži ime funkcije sa postfiksom **_body**, u ovom slučaju **@f_body:**. U naredbi **ADDS** se računa povratna vrednost funkcije, a naredbom **MOV** se ta vrednost smešta u registar **%13**. Ako funkcija ne sadrži return iskaz, kao povratna vrednost funkcije služi zatečeni sadržaj registra **%13**, koji je nepoznat u vreme denisanja funkcije. Ekvivalentni asemblerski kod kraja funkcije započinje labelom koja sadrži ime funkcije sa postfiksom **_exit**, u ovom slučaju **@f_exit:**. Naredbom **MOV** se oslobađa prostor za lokalne promenljive (pokazivač steka se vraća u poziciju u kojoj je bio pre zauzimanja lokalnih promenljivih), a u naredbi **POP** se u pokazivač frejma vraća njegova prethodna vrednost (stari pokazivač frejma). Naredbom **RET** će doći do preusmeravanja toka izvršavanja na mesto odakle je funkcija pozvana. Primeru poziva funkcije (u okviru iskaza dodele) odgovara seldeći kod:

```
a = f(a + b);
```

```
ADDS -4(%14),-8(%14),%0
PUSH %0
CALL f
ADDS %15,$4,%15
MOV  %13,-4(%14)
```

U prvoj liniji (u naredbi **ADDS**) se računa vrednost argumenta poziva funkcije i smešta u radni registar (pretpostavka je da su svi radni registri slobodni). Naredbom **PUSH** se vrednost argumenta smešta na stek. Naredbom **CALL** se poziva funkcija, a pomoću naredbe **ADDS** se oslobađa prostor koji je zauzimao argument (pokazivač steka se pomera 1 lokaciju naviže). Naredbom **MOV** se isporučuje povratna vrednost funkcije. Ako se na mestu argumenta pojave literali ili promenljive, kao u slučaju poziva u sledećem primeru, onda nisu potrebni radni registri, jer se vrednosti literala i promenljivih direktno mogu smeštati na stek.

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

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

Primeru **if** iskaza odgovara sledeći kod:

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

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

U ekvivalentnom asemblerskom kodu se pojavljuju labele **@if**, **@true**, **@false** i **@exit**. Neke od njih su potrebne, jer su ciljevi naredbi skokova (to su **@false** i **@exit**), dok su labele **@if** i **@true** nepotrebne, ali zgodne za praćenje i proveru izgenerisanog koda. Labela **@if** označava početak **if** iskaza, a labela **@exit** njegov kraj. Izvršavanje **if** iskaza započinje proverom uslova (naredba **CMP**) i naredbom skoka (na **@true** ili **@false** labelu) u zavisnosti od ispunjenosti uslova. Neposredno iza labele **@true** se nalazi kod koji će se izvršiti ukoliko je uslov tačan i iza toga bezuslovan skok na kraj if iskaza (odnosno na **@exit** labelu), da izvršavanje ne bi “propalo” na **@false** labelu. Neposredno iza **@false** labele se nalazi kod koji će se izvršiti ukoliko uslov nije tačan. Labele u ekvivalentnom (i izgenerisanom) kodu moraju biti jedinstvene. Svaka labela se završava brojem koji sadrže promenljive **for_num, if_num, while_num** i **ternary_num**(u zavisnosti od toga da li se

koristi **for, if, while** ili **ternarni operator**) (aktuelni broj labele). Jednoznačni brojevi se dobijaju inkrementiranjem datih promenljivih za svaki sledeći iskaz. Ukoliko bi se u prethodnom primeru izostavio **else** deo, u ekvivalentnom kodu bi nestao deo koda iza **@false** labele, kao na u sledećem kodu:

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

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

Datoteke **codegen** (**.c** i **.h**) sarže funkcije za generisanje koda. Deo generisanja je smešten u ovim funkcijama, a deo u parseru (**micko.y**). Funkcija **take_reg()** zauzima prvi slobodni registar i vraća njegov indeks u tabeli simbola.

int take_reg(void);

Funkcija **free_reg()** oslobađa poslednji zauzeti registar.

void free_reg(void);

Funkcija **free_if_reg()** oslobađa registar čiji je indeks prosleđen.

void free_if_reg(int reg_index);

Funkcija **gen_sym_name()** ispisuje simbol koji se nalazi na prosleđenom indeksu u tabeli simbola. Funkcija ispisuje simbol u odgovarajućem obliku, u zaivnosti od njegove vrste.

void gen_sym_name(int index);

Za generisanje naredbe poređenja - **CMP** naredba, predviđena je funkcija **gen_cmp()** čiji su parametri indeksi operanada u tabeli simbola:

void gen_cmp (int op1_index , int op2_index) ;

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

void gen_mov (int input_index , int output_index) ;

Generisanje koda se realizuje u parseru, tako da se faza generisanja koda, praktično, odvija zajedno sa sintaksnom i semantičkom analizom. Sledi **micko.y** datoteka: parser sa semantičkim proverama i generisanjem hipotetskog asemblerskog koda. Parser pre početka parsiranja napravi **.asm** datoteku i zatim u nju generiše kod. Ukoliko parsiranje prođe bez greške, ova datoteka će biti validna. Ako se pojave greške u toku parsiranja, parser će obrisati ovu datoteku, jer sigurno nije validna. U prvom delu bison specikacije dodate su promenljive: **for_num, if_num, while_num** i **ternary_num** - brojači labela i **output** - izlazni fajl u koji će biti izgenerisan hipotetski asemblerski program:

```
FILE *output;
int for_num = -1;
int if_num = -1;
int while_num = -1;
int ternary_num = -1;
```

Generisanje koda za deniciju funkcije podrazumeva generisanje:

1. labele koja se zove isto kao funkcija

2. smeštanja starog pokazivača frejma (registar **%14**) na stek:

   PUSH %14

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

   MOV %15 ,%14

4. tela funkcije (na čijem početku će se izgenerisati zauzimanje prostora za lokalne promenljive)

5. **exit** labele funkcije

6. oslobađanja prostora zauzetog za lokalne promenljive: pomeranjem stek pokazivača na frejm pokazivač:

MOV %14 ,%15

7. vraćanja starog pokazivača frejma:

   POP %14

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

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

```
function
  : _TYPE _ID
      {
        fun_idx = lookup_symbol($2, FUN);
        if(fun_idx == NO_INDEX)
          fun_idx = insert_symbol($2, FUN, $1, NO_ATR, NO_ATR);
        else
          err("redefinition of function '%s'", $2);

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

        code("\n@%s_exit:", $2);
        code("\n\t\tMOV \t%%14,%%15");
        code("\n\t\tPOP \t%%14");
        code("\n\t\tRET");
      }
  ;
```

Ukoliko se u telu funkcije (pojam **body**) nalaze lokalne promenljive, potrebno je izgenerisati zauzimanje prostora za njih na steku. Ova akcija se može sproversti tek kada se zna ukupan broj lokalnih promenljivih, a to je nakon parsiranja pojma **variable_list**. 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**. Zbog preglednosti izgenerisanog koda, na početku tela funkcije, generiše se **body** labela funkcije.

```
body
  : _LBRACKET variable_list
    {
      if(var_num)
        code("\n\t\tSUBS\t%%15,$%d,%%15", 4*var_num);
      code("\n@%s_body:", get_name(fun_idx));
    }
    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**, tj. rezultat izraza sa desne strane znaka jednakosti. Drugi argument je indeks elementa tabele simbola na kom se nalazi identikator sa leve strane znaka jednakosti, koji predsavlja izlazni operand naredbe **MOV**.

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

Za generisanje aritmetiških operacija koriste se nizovi stringova **ar_instructions[]** (u **defs.h**) koji sadrže varijante stringova (imena naredbi) za označene i neoznačene tipove podataka. Za indeksiranje ovog niza potrebna je oznaka aritmetičke operacije (konstanta **ADD** ili **SUB**, koja se nalazi u metapromenljivoj **$2** a koja je denisana u enumeraciji **arops**) i oznaka tipa (bilo kog) operanda. Kada se ove dve informacije iskoriste, ona će dobiti string odgovarajuče aritmetičke naredbe. Prvi i drugi (ulazni) operand naredbe se generišu pomoću funkcije **gen_sym_name()** jer imamo njihove indekse u tabeli simbola. Ukoliko je neki od ulaznih operanada bio smešten u registru, nakon preuzimanja vrednosti on se može osloboditi. U ovu svrhu se koristi funkcija **free_if_reg()**. Prilikom generisanja izlaznog operanda aritmetičke naredbe, zauzima se prvi slobodan registar pomoću funkcije **take_reg()** i u

njega smešta rezultat operacije. Vrednost **num_exp** pojma biće indeks tog registra u tabeli simbola (kao mesto gde se nalazi rezultat parsiranog numeričkog izraza).

```
num_exp
  : exp
  | num_exp _AROP exp
      {
        if(get_type($1) != get_type($3))
          err("invalid operands: arithmetic operation");
        int t1 = get_type($1);
        code("\n\t\t%s\t", ar_instructions[$2 + (t1 - 1) * AROP_NUMBER]);
        gen_sym_name($1);
        code(",");
        gen_sym_name($3);
        code(",");
        free_if_reg($3);
        free_if_reg($1);
        $$ = take_reg();
        gen_sym_name($$);
        set_type($$, t1);
      }
  ;
```

U pravilu za izraze (**exp**), pravilo, koje opisuje poziv funkcije i pravilo za ternarni operator, sadrži generisanje koda. Kada se poziv funkcije nađe negde u izrazu, njenu povratnu vrednost treba kopirati u prvi slobodan registar. Ovo je neophodno, jer je moguće da se u izrazu ponovo zatekne poziv neke funkcije koji će njenu povratnu vrednost ponovo smestiti u registar **%13** (i time poništiti prethodnu vrednost). Zato povratne vrednosti funkcija u izrazima treba čuvati u registrima. Generisanje koda za ternarni izraz podrazumeva: zauzimanje prvog slobodnog registra, skok na **false** labelu ukoliko relacioni izraz nije zadovoljen gde se pomera drugi **cond_exp** u registar, generisanje **true** labele i ukoliko je taj uslov ispunje onda se prvi **cond_exp** premešta u taj registar i skače se na **exit** labelu. Vrednost pojma **exp** postaje indeks elementa u tabeli simbola, u kom se nalazi rezultat izvršavanja funkcije.

```
exp
  : literal
  | _ID
      {
        $$ = lookup_symbol($1, VAR|PAR);
        if($$ == NO_INDEX)
          err("'%s' undeclared", $1);
      }
  | function_call
      {
        $$ = take_reg();
        gen_mov(FUN_REG, $$);
      }
  | _LPAREN num_exp _RPAREN
      { $$ = $2; }
  | _LPAREN rel_exp _RPAREN _QMARK cond_exp _COLON cond_exp
    {
      int out = take_reg();
      ++ternary_num;

      if(get_type($5) != get_type($7))
        err("expressions don't have same types in ternary exp");

      code("\n\t\t%s\t@false_ternary%d", opp_jumps[$2], ternary_num);
      code("\n@true_ternary%d:", ternary_num);
      gen_mov($5, out);

      code("\n\t\tJMP\t\t@exit_ternary%d", ternary_num);
      code("\n@false_ternary%d:", ternary_num);
      gen_mov($7, out);
      code("\n@exit_ternary%d:", ternary_num);
      $$ = out;
    }
  ;
```

Za generisanje poziva funkcije koristi se naredba **CALL**. Posle poziva funkcije treba obrisati deo steka na kom su bili argumenti funkcije. U tu svrhu generiše se **ADDS** naredba kojom se pomeri pokazivač steka onoliko lokacija naviše koliko je bilo argumenata (svaka lokacija je velika 4 bajta). Vrednost pojma **function_call** postaje registar **%13**, a njegov tip se postavlja na povratni tip funkcije.

```
function_call
  : _ID
    {
      fcall_idx = lookup_symbol($1, FUN);
      if(fcall_idx == NO_INDEX)
        err("'%s' is not a function", $1);
    }
    _LPAREN argument _RPAREN
    {
      if(get_atr1(fcall_idx) != $4)
        err("wrong number of arguments");
      code("\n\t\tCALL\t%s", get_name(fcall_idx));
      if($4 > 0)
        code("\n\t\tADDS\t%%15,$%d,%%15", $4 * 4);
      set_type(FUN_REG, get_type(fcall_idx));
      $$ = FUN_REG;
    }
  ;
```

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

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

  | num_exp
    {
      if(get_atr2(fcall_idx) != get_type($1))
        err("incompatible type for argument");
      free_if_reg($1);
      code("\n\t\tPUSH\t");
      gen_sym_name($1);
      $$ = 1;
    }
  ;
```

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 (**if_num**) sa ovom vrednošću se na početku generisanja if iskaza inkrementira, da bi označio sledeći iskaz. Na mestu relacionog izraza će se izgenerisati odgovarajuća **CMP** naredba. Nakon toga se generiše skok sa suprotnim uslovom na odgovarajuću **false** labelu. Ako je, na primer, u relaciji naveden operator ==, izgenerisaće se skok **JNE**. U ovu svrhu se koristi enumeracija **opp_jumps[]** koja sadrži imena naredbi skokova ali u kontra redosledu od konstanti relacionih operatora(denisano u **defs.h**). U nastavku se generiše true labela sa brojem **if_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. Zatim sledi generisanje **false** labele, iza koje će uslediti generisanje **else** tela (ako postoji).

```
if_part
  : _IF _LPAREN
      {
        $<i>$ = ++if_num;
        code("\n@if%d:", if_num);
      }
    rel_exp
      {
        code("\n\t\t%s\t@false_if%d", opp_jumps[$4], $<i>3);
        code("\n@true_if%d:", $<i>3);
      }
    _RPAREN statement
      {
        code("\n\t\tJMP\t@exit_if%d", $<i>3);
        code("\n@false_if%d:", $<i>3);
        $$ = $<i>3;
      }
  ;
```

Ako bi se u izvornom kodu pojavila situacija da **then** ili **else** telo if-a sadrže drugi if ili neki drugi iskaz, koji u svom generisanju koda takođe koristi brojač **if_num**, bilo bi potrebno sačuvati vrednost ovog brojača pre ulaska u novi iskaz, i vratiti vrednost posle parsiranja novog iskaza (zbog nastavka parsiranja spoljašnjeg if-a). Ove vrednosti se mogu čuvati na nekom steku labela ili ih (kao što je u ovom primeru rešeno) čuvati kao vrednost akcije. Vrednost broja tekuće labele se čuva kao vrednost prve akcije u if pravilu (tj. **$3**). Na kraju if iskaza, generiše se **exit** labela, kao oznaka kraja ekvivalentnog asemblerskog koda if iskaza.

```
if_statement
  : if_part %prec ONLY_IF
      { code("\n@exit_if%d:", $1); }

  | if_part _ELSE statement
      { code("\n@exit_if%d:", $1); }
  ;
```

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

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

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

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

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

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

  synerr = yyparse();

  clear_symtab();
  fclose(output);

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

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

  if(synerr)
    return -1;  //syntax error
  else if(error_count)
    return error_count & 127; //semantic errors
  else if(warning_count)
    return (warning_count & 127) + 127; //warnings
  else
    return 0; //OK
}
```

Za generisanje **for** iskaza radi se sledeće: literal se pomeri u promenljivu koja se koristi u **for** iskazu pomoću naredbe **gen_mov()** pa se zatim izgeneriše labela **for**. Onda će se zobg prisustva **rel_exp** izgenerisati **CMP** naredba, a posle nje i skok na **exit** labelu u slučaju da **rel_exp** nije ispunjen. Zatim se generiše kod pojma **statement** koji je već objašnjen. Nakon njega će se inkrement iskaz za koji se koriste **ADDS** ili **ADDU** u zavisnosti od tipa int-a i pomoću funkcije **gen_sym_name()**. Nakon njih će se izgenerisati skok na **for** labelu a zatim i sama **exit** labela.

```
for_statement
  : _FOR _LPAREN _ID _ASSIGN literal
    {
      $<i>$ = ++for_num;

      int i = lookup_symbol($3, VAR|PAR);
      if(i == NO_INDEX)
        err("'%s' undeclared", $3);

      if(get_type(i) != get_type($5))
        err("incompatible types");

      gen_mov($5, i);
      code("\n@for%d:", for_num);
    }
  _SEMICOLON rel_exp
    {
      code("\n\t\t%s\t@exit_for%d", opp_jumps[$8], $<i>6);
    }
  _SEMICOLON _ID _INC _RPAREN statement
    {
      int i = lookup_symbol($11, VAR|PAR);
      if(i == NO_INDEX)
        err("'%s' undeclared", $11);

      if(get_type(i) == INT)
        code("\n\t\tADDS\t");
      else
        code("\n\t\tADDU\t");

      gen_sym_name(i);
      code(",$1,");
      gen_sym_name(i);

      code("\n\t\tJMP\t\t@for%d", $<i>6);
      code("\n@exit_for%d:", $<i>6);
    }
  ;
```

Generisanje koda za **while** petlju je slično kao i za **for** petlju pa čak i jednostavnije. Prvo će se izgenerisati **while** labela nakon koje će zbog prisustva **rel_exp** biti izgenerisana **CMP** naredba. Onda će se izgenerisati skok na **exit** labelu ukoliko **rel_exp** nije ispunjen. Nakon toga će se izgenerisati kod za pojam **statement**. I na kraju biće izgenerisan skok na **while** labelu nakon koje će biti izgenerisana sama **exit** labela.

```
while_statement
  : _WHILE
    {
      $<i>$ = ++while_num;

      code("\n@while%d:", while_num);
    }
  _LPAREN rel_exp
    {
      code("\n\t\t%s\t@exit_while%d", opp_jumps[$4], $<i>2);
    }
  _RPAREN statement
    {
      code("\n\t\tJMP\t\t@while%d", $<i>2);
      code("\n@exit_while%d:", $<i>2);
    }
  ;
```