Chapter 1

Semantička analiza

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

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

1.1 Organizacija memorije za miniC

Globalni identifikatori su statični – postoje za sve vreme izvršavanja programa i za njih se mogu rezervisati memorijske lokacije u toku kompajliranja. Lokalni identifikatori su dinamični – postoje samo za vreme izvršavanja funkcija i za njih se zauzimaju memorijske lokacije na početku izvršavanja funkcija. Ove lokacije se oslobađaju na kraju izvšavanja funkcija, pa se zato lokalnim identifikatorima dodeljuju memorijske lokacije sa steka. Deo steka koji se zauzima za izvršavanje neke funkcije se zove (stek) frejm. Tipični frejm izgleda kao na slici 1.1.

Stek frejm, redom, sadrži:

- ⋆ argumente poziva funkcije (od poslednjeg ka prvom),

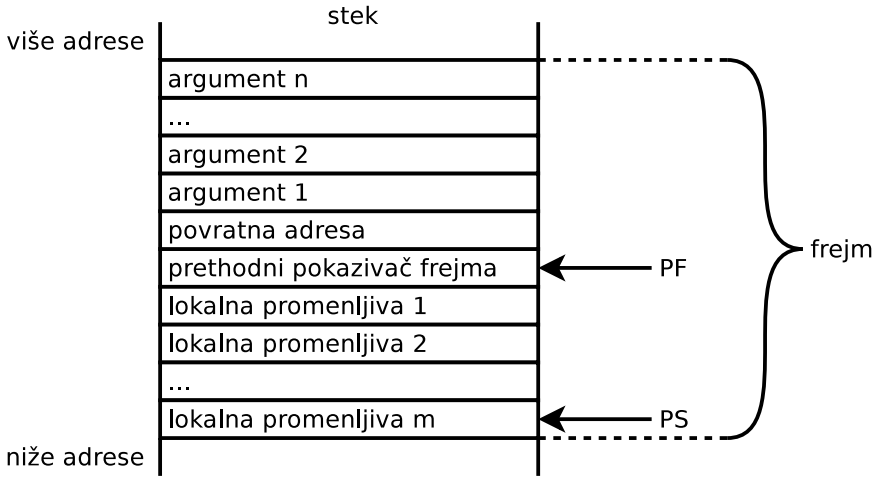

Figure 1.1: Stek frejm

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

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

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

1.2 Tabela simbola

Tabela simbola (*symbol table*) je struktura podataka koja čuva identifikatore iz kompajliranog programa i sve informacije o tim identifikatorima. Za

semantičku analizu, ona je neophodna, jer sadrži sve informacije na osnovu kojih se mogu sprovesti semantičke provere.

Na primer, u programu sa listinga 1.1, postoji samo `main()` funkcija, sa jednim parametrom i jednom lokalnom promenljivom:

Listing 1.1: `add.mc`

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

Tokom kompajliranja ove funkcije, u tabelu simbola treba zapisati da postoji (slika 1.2):

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

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

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

- literal 100 tipa `int`.

Za program sa listinga 1.1, 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	-	-

Figure 1.2: Primer tabele simbola

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

Pošto u tabeli simbola istovremeno mogu postojati identični globalni i lokalni identifikatori, radi njihovog razlikovanja, u tabeli mora biti naznačena vrsta identifikatora (da li je funkcija, promenljiva, parametar, argument, ...). Za svaki identifikator mora postojati i oznaka njegovog tipa (da li je 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.

1.2.1 Implementacija tabele simbola

Za potrebe kursa koristi se vrlo jednostavna implementacija tabele simbola: tabela je organizovana kao niz struktura koje opisuju osobine identifikatora (slika 1.2).

Datoteka `defs.h` (listing 1.2) koja je uvedena u poglavlju sintaksne analize (listing `??`) je proširena enumeracijom `kinds` koja sadrži konstante koje opisuju vrste simbola. Vrednosti konstanti su stepeni broja 2 (1, 2, 4, 8, ... tj. sadrže samo po 1 setovan bit) da bi se nad njima mogla primeniti *bitwise* logika. Ovakav zapis klase simbola olakšava pretragu tabele simbola.

Listing 1.2: `defs.h` - enumeracija `kinds`

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

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

Listing 1.3: `defs.h` - dodate konstante

```
#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
```

1.2.1.1 Struktura jednog elementa tabele simbola

Struktura jednog elementa tabele simbola se sastoji od polja (listing 1.4):

`name` string identifikatora

`kind` vrsta simbola - sadrži vrednost iz enumeracije `kinds` (listing 1.2)

`type` tip simbola - sadrži vrednost iz enumeracije `types` (listing 1.2)

`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[1]
- ⋆ za ostale identifikatore, 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` (listing 1.2))

Listing 1.4: `symtab.h` - struktura elementa 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;
```

[1] miniC funkcija podržava maksimalno jedan parametar

1.2.1.2 Operacije za rad sa tabelom simbola

Operacije za rad sa tabelom simbola su implementirane u datotekama `symtab.h` i `symtab.c`.

Funkcije za ubacivanje u tabelu simbola

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);
```

Funkcije za pretraživanje tabele simbola

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);
```

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

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

Ažuriranje elementa tabele simbola

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

```
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);
```

Funkcije za brisanje simbola iz tabele simbola

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

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

Funkcije za rad sa tabelom simbola u celini

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

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

1.3 miniC parser sa semantičkim proverama

miniC parser sa semaničkim proverama koristi prethodno opisanu tabelu simbola (listinzi 1.2, 1.4) a impelementiran je u datoteci `semantic.y`.

Prvi deo specifikacije parsera (dat na listingu **??**), je proširen definicijama promenljivih koje koristi parser u toku provere semantike (listing 1.5). 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`:

Listing 1.5: `semantic.y` - promenljive

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

U nastavku su definisani i tipovi pojmova: iza ključne reči `%type` se navodi tip iz unije (`<i>` ili `<s>`), a zatim imena svih pojmova koji će biti tog tipa (listing 1.6). 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 *`).

Listing 1.6: `semantic.y` - tipovi pojmova

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

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

Listing 1.7: `semantic.y` - tokeni za razrešavanje dvosmislenosti

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

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

Semantička provera, koju treba obaviti nakon prepoznavanja celog programa, je da li postoji `main()` funkcija i da li je njen tip `int`. Provera se vrši pretragom tabele simbola. Ako se u tabeli ne pronađe simbol `main` koji predstavlja funkciju (`FUN`), prijavi se semantička greška. Drugi deo provere se radi samo ako simbol `main` postoji u tabeli: proveri se njegov tip i prijavi semantička greška ako nije `int` (listing 1.8).

Listing 1.8: `semantic.y` - `program`

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

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

Listing 1.9: `semantic.y` - tip

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

Tokom parsiranja definicije funkcije vrši se nekoliko semantičkih akcija (listing 1.10). Čim pristigne ime funkcije, ono se smešta u tabelu simbola zajedno sa informacijama o tipu povratne vrednosti funkcije (vrednost pojma `type`, tj. vrednost meta-promenljive `$1`, koja nosi konstantu `INT` ili `UINT`) i vrsti identifikatora (konstanta `FUN`). Promenljiva `fun_idx` dobija indeks elementa u tabeli simbola u koji je smešteno ime funkcije.

Nakon parisranja definicije parametra u element tabele simbola sa indeksom `fun_idx` koji čuva informacije o funkciji, u polje `atr1`, smešta se podatak o broju parametara funkcije. Ovaj podatak se čita kao vrednost meta-promenljive `$5`, tj. kao vrednost pojma `parameter`. Promenljiva `var_num`, koja broji lokalne promenljive jedne funkcije, se resetuje, jer sada sledi ulazak u telo (nove) funkcije.

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

imena funkcije. Ime funkcije je globalni identifikator i zato treba da bude prisutan u tabeli simbola tokom parsiranja celog programa.

Listing 1.10: `semantic.y` - definicija funkcije

```
function
  : type _ID
      {
        fun_idx = lookup_symbol($2, FUN);
        if(fun_idx == -1)
          fun_idx = insert_symbol($2, FUN, $1, NO_ATR, NO_ATR);
        else
          err("redefinition␣of␣function␣'%s'", $2);
      }
    _LPAREN parameter _RPAREN
      {
        set_atr1(fun_idx, $5);
        var_num = 0;
      }
    body
      {
        clear_symbols(fun_idx + 1);
      }
  ;
```

Nakon parsiranja pojma `parameter`, ime parametra (vrednost `$2`) se ubacuje u tabelu simbola kao `PAR` sa tipom koji nosi pojam `type` (vrednost `$1`) i sa rednim brojem `1` (polje `atr1` parametra čuva redni broj parametra) (listing 1.11). Tip parametra (vrednost `$1`) se zapisuje u polju `atr2` simbola funkcije u tabeli simbola (jer polje `atr2` za funkciju čuva tip njenog parametra). Vrednost pojma `parameter` se postavlja ili na `0` (ako nije bilo parametara) ili na `1` ukoliko je detektovan (jedan) parametar.

Listing 1.11: `semantic.y` - parametar

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

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

Tokom parsiranja deklaracije promenljive (pojam `variable`), ime promenljive (vrednost `$2`) se ubacuje u tabelu simbola kao `VAR` sa tipom koji nosi pojam

`type` (vrednost `$1`). Vrednost brojača lokalnih promenljivih se inkrementira i zapisuje se kao atribut `atr1` ovog simbola u tabeli simbola, zato što lokalna promenljiva na mestu prvog atributa u tabeli simbola čuva svoj redni broj (listing 1.11).

Ukoliko je u tabeli simbola pronađen parametar sa istim imenom, prijavljuje se semantička greška zato što dva lokalna identifikatora unutar funkcije ne mogu imati isto ime.

Listing 1.12: `semantic.y` - promenljiva

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

Nakon parsiranja iskaza dodele, proverava se da li je identifikator koji se nalazi sa leve strane jednakosti promenljiva ili parametar (listing 1.13). Ako nije, prijavljuje se semantička greška, jer je na tom mestu dozvoljeno samo ime promenljive ili parametra (a ne, recimo, ime funkcije). Dodatno, vrši se i provera tipova leve i desne strane jednakosti, jer bi oni trebali biti isti. 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 identifikatora sa leve strane jednakosti i indeks u tabeli simbola gde se nalazi rezultat numeričkog izraza sa desne strane jednakosti (vrednost `$3`)).

Listing 1.13: `semantic.y` - iskaz dodele

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

Nakon parsiranja aritmetičkog izraza (pojam `num_exp`), vrši se provera tipova operanada, pa ako nisu isti, prijavljuje se semantička greška (listing 1.14).

Provera se vrši pozivom funkcije `get_type()` sa argumentima: indeks prvog operanda u tabeli simbola (`$1`) i indeks drugog operanda u tabeli simbola (`$3`).

Listing 1.14: `semantic.y` - numerički izraz

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

Nakon parsiranja identifikatora u izrazima (`exp`), vrši se provera postojanja dotičnog imena, tj. njegovog prisustva u tabeli simbola (listing 1.15). Ovu proveru obavlja funkcija `lookup_symbol()`. Ako ga nema u tabeli, znači da nikada nije deklarisan, pa treba prijaviti semantičku grešku, jer se ne može koristiti promenljiva koja nije prethodno deklarisana.

Ako se izraz nalazi u zagradama (poslednja varijanta pravila `exp`), njegova vrednost treba da postane i vrednost celog izraza.

Listing 1.15: `semantic.y` - izraz

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

Kada se isparsira literal, treba ga ubaciti u tabelu simbola kao lokalni simbol, a pojmu `literal` dodeliti vrednost: indeks u tabeli simbola na kom je smeštena konstanta (listing 1.16).

Listing 1.16: `semantic.y` - literal

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

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

Tokom parsiranja poziva funkcije (pojam `function_call`, listing 1.17) proverava se da li je navedeni identifikator ime neke od postojećih funkcija, pa ako nije, prijavljuje se greška da se poziva nepostojeća funkcija. Kako je usvojeno, povratna vrednost funkcije će biti smeštena u registru `%13`, pa je potrebno postaviti vrednost pojma `function_call` na indeks `13` u tabeli simbola (gde se nalazi registar `%13`). Registru `%13` se postavlja tip povratne vrednosti funkcije koja se kasnije koristi u izrazima.

Listing 1.17: `semantic.y` - poziv funkcije

```
function_call
  : _ID
      {
        fcall_idx = lookup_symbol($1, FUN);
        if(fcall_idx == -1)
          err("'%s' is not a function", $1);
      }
    _LPAREN argument _RPAREN
      {
        if(get_atr1(fcall_idx) != $4)
          err("wrong number of args to function '%s'",
              get_name(fcall_idx));
        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 (listing 1.18). 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).

Listing 1.18: `semantic.y` - argument

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

```
  | num_exp
    {
      if(get_atr2(fcall_idx) != get_type($1))
        err("incompatible␣type␣for␣argument␣in␣'%s'",
            get_name(fcall_idx));
      $$ = 1;
    }
  ;
```

Tokom parsiranja relacionog izraza odnosno pojma `rel_exp` (vidi listing 1.19) 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`).

Listing 1.19: `semantic.y` - relacioni izraz

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

Nakon prepoznavanja `return` iskaza (listing 1.20), potrebno je proveriti tip izraza koji se vraća iz funkcije: semantika nalaže da ovaj tip mora biti isti kao povratni tip funkcije. Funkcija koja 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.

Listing 1.20: `semantic.y` - `return` iskaz

```
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.

Listing 1.21: `semantic.y` - korisničke funkcije parsera

```
int yyerror(char *s) {
```

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

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

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

Listing 1.22: `semantic.y` - `main` funkcija

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

  synerr = yyparse();

  clear_symtab();

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

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

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

1.3.1 Primer upotrebe parsera sa semantičkim proverama

Ako parseru sa semantičkim proverama prosledimo datoteku `abs.mc`:

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

```
  return abs(-5);
}
```

parser će ispisati da nema (semantičkih) grešaka, jer ova datoteka sadrži sintaksno i semantički ispravan miniC program:

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

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

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

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

```
$./semantic <abs.mc
line 11: ERROR: incompatible type for argument in 'abs'
error(s).
$
```

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

```
$./semantic <abs.mc
line 13: ERROR: undefined reference to 'main'
1 error(s).
$
```

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

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

1.4 Vežbe

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

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