Chapter 1

Generisanje koda

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

1.1 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 '`@`'.

1.1.1 Operandi

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

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

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

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

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

Operandi se dele na:

ulazne neposredni, registarski, direktni, indirektni i indeksni i

izlazne registarski, direktni, indirektni i indeksni.

1.1.2 Naredbe

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

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

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

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

```
JMP ulazni operand
```

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

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

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

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

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

```
CALL ulazni operand
```

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

```
RET
```

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

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

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

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

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

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

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

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

1.1.3 Direktive

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

```
WORD broj
```

1.2 Primeri ekvivalencije

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

Primeri (mogućih) ekvivalencija između miniC koda i hipoteskog asemblerskog koda, koji slede na listinzima, napisani su u dve kolone. U prvoj koloni se nalazi miniC kod koji se prevodi, a u drugoj se nalazi ekvivalentni asemblerski kod. Primeri sadrže i predloge načina generisanja ekvivalentnog koda.

U svim navedenim primerima se podrazumeva da su `a`, `b` i `c` celobrojne označene lokalne promenljive, deklarisane tim redom.

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

1.2.1 Iskaz pridruživanja

Primeru iskaza pridruživanja odgovara izgenerisani kod:

Listing 1.1: primer ekvivalencije za jednostavan iskaz pridruživanja

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

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

Listing 1.2: primer ekvivalencije za iskaz pridruživanja

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

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

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

Pošto se međurezultati izraza koriste u suprotnom redosledu od onog u kome su izračunati, radni registri, koji se koriste za smeštanje međurezultata, se zauzimaju i oslobađaju po principu steka. Kao "pokazivač steka

registara" koristi se promenljiva `free_reg_num`, koja sadrži broj prvog slobodnog radnog registra. Zauzimanje radnog registra se sastoji od preuzimanja vrednosti promenljive `free_reg_num` i njenog inkrementiranja, a oslobađanje radnog registra se sastoji od dekrementiranja ove promenljive. Treba napomenuti da je broj registra istovremeno i indeks elementa tabele simbola.

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

1.2.2 Funkcija

1.2.2.1 Definicija funkcije

Za smeštanje povratne vrednosti funkcije (po dogovoru) se koristi registar `%13`. Funkcija, nakon poziva, lokalne promenljive i argumente čuva na stek frejmu (vidi sliku **??**). Listing 1.3 sadrži primer ekvivalencije za definiciju funkcije.

Listing 1.3: primer ekvivalencije za definiciju 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 identifikator kao funkcija, u ovom slučaju `f:`. Naredbom `PUSH` na stek se smešta "stari" pokazivač frejma, dok se pomoću naredbe `MOV` postavlja "novi" pokazivač frejma (registar `%14` sada pokazuje na vrh steka, kao i `%15`). Pomoću naredbe `SUBS` se zauzima prostor na steku za lokalnu promenljivu `a` (pokazivač steka se pomera jednu lokaciju, tj. 4 bajta niže). Promenljiva `var_num` služi kao brojač lokalnih promenljivih. Veličina prostora za lokalne promenljive se određuje kao `var_num * 4`. Prostor se zauzima samo ako funkcija ima lokalnih promenljivih (`var_num > 0`).

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

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

1.2.2.2 Poziv funkcije

Primeru poziva funkcije (u okviru iskaza dodele) odgovara kod sa listinga 1.4.

Listing 1.4: primer ekvivalencije za poziv funkcije (a)

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

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

Ako se na mestu argumenta pojave literali ili promenljive, kao u slučaju poziva u primeru sa listinga 1.5, onda nisu potrebni radni registri, jer se vrednosti literala i promenljivih direktno mogu smeštati na stek.

Listing 1.5: primer ekvivalencije za poziv funkcije (b)

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

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

1.2.3 `if` iskaz

Primeru `if` iskaza odgovara kod sa listinga 1.6.

Listing 1.6: primer ekvivalencije za `if` iskaz

```
if(a < b)                          @if0:
  a = 1;                              CMPS -4(%14),-8(%14)
else                                  JGES @false0
  a = 2;                           @true0:
                                      MOV  $1,-4(%14)
                                      JMP  @exit0
                                   @false0:
                                      MOV  $2,-4(%14)
                                   @exit0:
```

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

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

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

Ukoliko bi se u prethodnom primeru izostavio `else` deo, u ekvivalentnom kodu bi nestao deo koda iza `@false` labele, kao na listingu 1.7.

Listing 1.7: primer ekvivalencije za `if` iskaz bez `else` dela

```
if(a < b)                          @if0:
  a = 1;                              CMPS -4(%14),-8(%14)
                                      JGES @false0
                                   @true0:
                                      MOV  $1,-4(%14)
                                      JMP  @exit0
```

```
@false0:
@exit0:
```

1.3 miniC parser sa generisanjem koda

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

Funkcije koje generišu labele su:

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

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

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

```
void gen_cmp(int 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` specifikacije dodate su promenljive: `lab_num` - brojač labela i `output` - izlazni fajl u koji će biti izgenerisan hipotetski asemblerski program:

Listing 1.8: `micko.y` - promenljive u parseru

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

Generisanje koda za definiciju funkcije podrazumeva generisanje (vidi listing 1.3):

1. labele koja se zove isto kao funkcija
2. smeštanja starog pokazivača frejma (registar `%14`) na stek:
   ```
   PUSH %14
   ```
3. postavljanja novog pokazivača frejma (`%14`): prebacivanjem vrednosti stek pokazivača u pokazivač frejma:
   ```
   MOV %15,%14
   ```
4. tela funkcije (na čijem početku će se izgenerisati zauzimanje prostora za lokalne promenljive)
5. `exit` labele funkcije
6. oslobađanja prostora zauzetog za lokalne promenljive: pomeranjem stek pokazivača na frejm pokazivač:
   ```
   MOV %14,%15
   ```
7. vraćanja starog pokazivača frejma:
   ```
   POP %14
   ```
8. naredbe `RET`, koja će sa steka skinuti povratnu adresu i dovesti do preusmeravanja toka izvršavanja na deo koda iz kog je funkcija pozvana.

Prva tri elementa generisanja koda treba da se izvrše na početku parsiranja funkcije, tj. odmah nakon prepoznavanja imena funkcije (listing 1.9). Telo funkcije se generiše tokom parsiranja pojma `body`. Na kraju parsiranja funkcije se generišu elementi 5-8.

Listing 1.9: `micko.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);

        code("\n%s:", $2);
        code("\n\t\t\tPUSH\t%%14");
        code("\n\t\t\tMOV␣\t%%15,%%14");
```

```
      }
    _LPAREN parameter _RPAREN
      {
        set_atr1(fun_idx, $5);
        var_num = 0;
      }
    body
      {
        clear_symbols(fun_idx + 1);

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

Ukoliko se u telu funkcije (pojam `body`) nalaze lokalne promenljive, potrebno je izgenerisati zauzimanje prostora za njih na steku (listing 1.10). 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` (`var_num`: vidi listing ??).

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

Listing 1.10: `micko.y` - telo funkcije

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

Kao ekvivalent iskaza dodele, generiše se `MOV` naredba (listing 1.11). Funkciji `gen_mov()` se, kao prvi argument, prosleđuje `$3` koji sadrži indeks elementa tabele simbola na kom se nalazi ulazni operand naredbe `MOV`, tj. rezultat izraza sa desne strane znaka jednakosti. Drugi argument je indeks elementa tabele simbola na kom se nalazi identifikator sa leve strane znaka jednakosti, koji predsavlja izlazni operand naredbe `MOV`.

Listing 1.11: `micko.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");
        gen_mov($3, idx);
      }
  ;
```

Za generisanje aritmetičkih operacija koriste se nizovi stringova `arithmetic_operators[]` (vidi *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 meta-promenljivoj `$2` a koja je definisana u enumeraciji `arops`) i oznaka tipa (bilo kog) operanda. Kada se ove dve informacije proslede funkciji `get_arop_stmt()`, ona će vratiti string odgovarajuće aritmetičke naredbe.

Prvi i drugi (ulazni) operand naredbe se generišu pomoću funkcije `print_symbol()` 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_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).

Listing 1.12: `micko.y` - aritmetičke operacije

```
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\t%s\t", get_arop_stmt($2, t1));
        print_symbol($1);
        code(",");
        print_symbol($3);
        code(",");
        if($3 >= 0 && $3 <= LAST_WORKING_REG)
          free_reg();
        if($1 >= 0 && $1 <= LAST_WORKING_REG)
```

```
          free_reg();
        $$ = take_reg();
        print_symbol($$);
        set_type($$, t1);
      }
  ;
```

U pravilu za izraze (`exp`), samo pravilo, koje opisuje poziv funkcije, sadrži generisanje koda (listing 1.13). Kada se poziv funkcije nađe negde u izrazu, njenu povratnu vrednost treba kopirati u prvi slobodan registar. Ovo je neophodno, jer je moguće da se u izrazu ponovo zatekne poziv neke funkcije koji će njenu povratnu vrednost ponovo smestiti u registar `%13` (i time poništiti prethodnu vrednost). Zato povratne vrednosti funkcija u izrazima treba čuvati u registrima. Vrednost pojma `exp` postaje indeks elementa u tabeli simbola, u kom se nalazi rezultat izvršavanja funkcije.

Listing 1.13: `micko.y` - izrazi

```
exp
  : literal

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

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

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

Za generisanje poziva funkcije koristi se naredba `CALL` (listing 1.14). 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.

Listing 1.14: `micko.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 arguments");
        code("\n\t\t\tCALL\t%s", get_name(fcall_idx));
        if($4 > 0)
          code("\n\t\t\tADDS\t%%15,$%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 (listing 1.15). Kako se parsiranje ovog pojma odvija pre generisanja poziva funkcije, to će naredba `PUSH` prethoditi naredbi `CALL` u asemblerskom programu.

Listing 1.15: `micko.y` - argument

```
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\t\tPUSH\t");
      print_symbol($1);
      $$ = 1;
    }
  ;
```

U okviru `if` iskaza treba izgenerisati više asemblerskih naredbi (listing 1.17). Na početku `if` iskaza generiše se labela `@ifX`, gde je `X` redni broj iskaza u programu. Brojač (`lab_num` (vidi 1.2.3)) 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 (vidi listing 1.18). Nakon toga se generiše skok sa suprotnim uslovom na odgovarajuću `@false` labelu. Ako je, na primer, u relaciji naveden operator `==`, izgenerisaće se skok `JNE`. U ovu svrhu se koristi niz `opp_jumps[]` koji

sadrži imena naredbi skokova ali u "kontra" redosledu od konstanti relacionih operatora (definisanih u enumeraciji `relops`, vidi `defs.h`). Funkcija `get_jump_stmt()` kojoj se prosledi vrednost pojma `rel_exp` i oznaka da li je potreban "kontra" skok ili ne, će vratiti string potrebne naredbe skoka.

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. Zatim sledi generisanje `false` labele, iza koje će uslediti generisanje `else` tela (ako postoji).

Listing 1.16: `micko.y` - `if` iskaz - 1

```
if_part
  : _IF _LPAREN
      {
        $<i>$ = ++lab_num;
        gen_snlab("if", lab_num);
      }
    rel_exp
      {
        code("\n\t\t\t%s\t@false%d",
          get_jump_stmt($4, TRUE),$<i>3);
        gen_snlab("true", $<i>3);
      }
    _RPAREN statement
      {
        code("\n\t\t\tJMP \t@exit%d", $<i>3);
        gen_snlab("false", $<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 `lab_num`, bilo bi potrebno sačuvati vrednost ovog brojača pre ulaska u novi iskaz, i vratiti vrednost posle parsiranja novog iskaza (zbog nastavka parsiranja spoljašnjeg `if`-a). Ove vrednosti se mogu čuvati na nekom steku labela 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 (listing 1.16).

Listing 1.17: `micko.y` - `if` iskaz - 2

```
  : if_part %prec ONLY_IF
      { gen_snlab("exit", $1); }

  | if_part _ELSE statement
      { gen_snlab("exit", $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 (listing 1.18). Vrednost pojma `rel_exp` postaje redni broj naredbe skoka u nizu `opp_jumps[]` (vidi `defs.h`).

Listing 1.18: `micko.y` - relacioni izraz

```
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 (listing 1.19). Iza toga se generiše bezuslovni skok (`JMP`) na `exit` labelu, koja označava kraj tela funkcije, čime se realizuje semantika `return` iskaza.

Listing 1.19: `micko.y` - `return` iskaz

```
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\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.

Listing 1.20: `micko.y` - `main()` funkcija

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

  synerr = yyparse();

  clear_symtab();
  fclose(output);

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

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

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

1.3.1 Primer upotrebe parsera sa generisanjem koda

Ako se parseru sa generisanjem koda prosledi datoteka `abs.mc` (listing 1.21), kompajler će izgenerisati datoteku `abs.asm` koja sadrži ekvivalentan kod, napisan na hipotetskom asemblerskom jeziku (listing 1.22).

Listing 1.21: `abs.mc`

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

Listing 1.22: `abs.asm`

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

1.4 Vežbe

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

1. globalne promenljive
2. definicija promenljive sa inicijalizacijom `int a = 5;`

3. definicija funkcije sa više parametara i poziv funkcije sa više argumenata

4. `while` iskaz

Listing 1.23: primer ekvivalencije za `while` iskaz

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

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

Listing 1.24: primer ekvivalencije za `continue` iskaz

```
while(a < 5) {                   @while0:
   if(a == b)                        CMPS a,$5
      continue;                      JGES @false0
   a = a + 1;                    @true0:
}                                @if1:
                                     CMPS a,b
                                     JNE  @false1
                                 @true1:
                                     JMP  @while0    //continue
                                     JMP  @exit1
                                 @false1:
                                 @exit1:
                                     ADDS a,$1,%0
                                     MOV  %0,a
                                     JMP  @while0
                                 @false0:
                                 @exit0:
```

6. `for` iskaz

Listing 1.25: primer ekvivalencije za `for` iskaz

```
for(i = 0; i <= 5; i++)              MOV  $0,i
   suma = suma + i;                @for0:
                                       CMPS i,$5
                                       JGTS @exit0
                                       ADDS suma,i,%0
                                       MOV  %0,suma
```

```
    ADDS i,$1,i
    JMP  @for0
@exit0:
```

7. `switch` iskaz

Listing 1.26: primer ekvivalencije za `switch` iskaz

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

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

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