Chapter 1

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.

1.1 Implementacija parsera

Za implementaciju parsera, često se koristi generator parsera `bison`. Korišćenje `bison`-a je prikazano na slici 1.1.

Figure 1.1: Korišćenje `bison`-a

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

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

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

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

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

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

1.2 miniC parser

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

Sve konstante koje su potrebne skeneru i parseru su smeštene u posebnoj datoteci (`defs.h`, listing 1.1) radi lakšeg održavanja koda.

Listing 1.1: `defs.h`

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

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

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

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

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

Specifikacija miniC jezika pripremljena za `bison` je data na listingu 1.2.

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

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

U drugom delu `bison` specifikacije navedena je gramatika miniC programskog jezika, kao što je navedeno u delu ??. To znači da će parser proveravati sintaksnu ispravnost miniC programa (listing 1.2).

Listing 1.2: `syntax.y`

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

  int yyparse(void);
```

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

%union {
  int i;
  char *s;
}

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

%nonassoc ONLY_IF
%nonassoc _ELSE

%%

program
  : function_list
  ;

function_list
  : function
  | function_list function
  ;

function
  : type _ID _LPAREN parameter _RPAREN body
  ;

type
  : _TYPE
  ;

parameter
  : /* empty */
  | type _ID
  ;

body
  : _LBRACKET variable_list statement_list _RBRACKET
  ;

variable_list
  : /* empty */
  | variable_list variable
  ;

variable
  : type _ID _SEMICOLON
  ;

statement_list
  : /* empty */
  | statement_list statement
  ;
```

```
statement
  : compound_statement
  | assignment_statement
  | if_statement
  | return_statement
  ;

compound_statement
  : _LBRACKET statement_list _RBRACKET
  ;

assignment_statement
  : _ID _ASSIGN num_exp _SEMICOLON
  ;

num_exp
  : exp
  | num_exp _AROP exp
  ;

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

literal
  : _INT_NUMBER
  | _UINT_NUMBER
  ;

function_call
  : _ID _LPAREN argument _RPAREN
  ;

argument
  : /* empty */
  | num_exp
  ;

if_statement
  : if_part %prec ONLY_IF
  | if_part _ELSE statement
  ;

if_part
  : _IF _LPAREN rel_exp _RPAREN statement
  ;

rel_exp
  : num_exp _RELOP num_exp
  ;

return_statement
  : _RETURN num_exp _SEMICOLON
  ;

%%

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

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

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

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

1.2.1 Primer upotrebe parsera

Ako parseru sa listinga 1.2 prosledimo datoteku `abs.mc` (listing ??), parser neće ispisati ništa, jer ova datoteka sadrži sintaksno ispravan miniC program:

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

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

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

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

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

1.2.2 IF-ELSE konflikt

Nakon kompajliranja primera sa listinga 1.2, `bison` će prijaviti postojanje jednog konflikta, koji se javlja zbog `if` pravila:

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

`Bison` je detektovao stanje u kom može da uradi i *shift* i *reduce* akciju [?] (da preuzme token ili da uradi redukciju po nekom pravilu), a nema na osnovu čega da odluči koju akciju da izvrši. Ova situacija se desi kada se na steku zatekne sledeće stanje `if rel_exp1 if rel_exp2 statement1` (slika 1.2.a) i ukoliko parser kao sledeći token dobija `else` (i posle toga njemu pripadajući iskaz `statement2`).

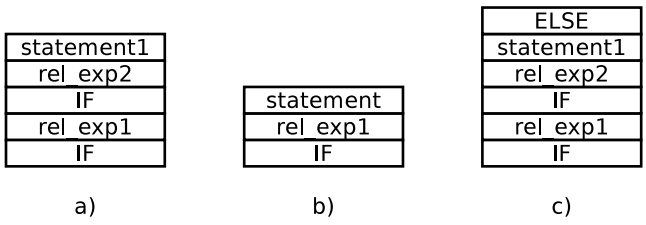

Figure 1.2: IF-ELSE konflikt (stanje na steku stanja)

U tom trenutku se može obaviti:

1. *reduce* akcija nad poslednja 3 elementa steka (slika 1.2.b), nakon čega će se `else` token i njemu pripadajući iskaz `statement2` priključiti prvom `if` iskazu, ili se može obaviti
2. *shift* akcija, u kojoj se na stek prebacuje `else` token (slika 1.2.c), nakon čega će uslediti iskaz `statement2` na steku, a tada je moguća redukcija poslednja 4 elementa sa steka. Time će se `else` token i njemu pripadajući iskaz `statement2` priključiti drugom `if` iskazu.

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

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

1.2.3 Oporavak od greške na primeru miniC parsera

Kada `bison`-ov parser detektuje grešku, ispiše string **"`syntax error`"** i završi parsiranje (završi program). Ukoliko korisnik želi da parser detektuje više od jedne greške, mora implementirati detekciju i oporavak greške. `Bison` nudi dve vrste opravka od greške. Obe će biti pokazane na primeru sa listinga 1.2.

Prvi primer oporavka koristi `error` token (listing 1.3) i biće primenjen na `if` pravilo. Ovaj token je postavljen između zagrada, tako da će parser moći da obavi redukciju po ovom pravilu, ukoliko dođe do sintaksne greške unutar uslova (tj. unutar relacionog izraza). Nakon toga može da izvrši pridruženu akciju u kojoj se ispisuje adekvatna poruka o lokaciji greške. Akcija sadrži i poziv makroa `yyerrok` koji signalizira parseru da je oporavak završen i da može da nastavi parsiranje.

Listing 1.3: Oporavak od greške pomoću `error` tokena

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

Drugi primer oporavka je primenjen na `return` pravilo i ima oblik pravila (*production rule*) (listing 1.4). Pravilu za `return` iskaz je dodato još jedno pravilo, u kom nedostaje karakter tačka-zarez na kraju iskaza. Ako se u ulaznom miniC programu pojavi `return` iskaz bez tačke-zarez na kraju, parser će obaviti redukciju po ovom pravilu i izvršiti pridruženu akciju u kojoj ispisuje poruku sa preciznim opisom greške. Ako se u ulaznom kodu pojavi kompletan `return` iskaz (koji sadrži karakter tačka-zarez na kraju iskaza) parser će proći kroz pravilo koje opisuje ispravan `return` iskaz. U ovoj varijanti oporavka od greške nema potrebe pozivati makro `yyerrok`, jer parser nije u vanrednom stanju zbog greške - on ovakvu situaciju ne vidi kao grešku, već kao najobičnije pravilo koje treba redukovati.

Listing 1.4: Oporavak od greške pomoću pravila

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

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

1.2.3.1 Primeri oporavka od greške

Ako malo izmenimo `abs` program (listing 1.5), tako da napravimo dve sintaksne greške - u dva iskaza izostavimo separator "`;`":

Listing 1.5: `abs2.mc`

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

```
    return i    //error
}

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

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

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

1.3 Vežbe

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

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

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

   ```
   break;
   ```

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

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

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

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

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

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

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

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

7. Proširiti deklaraciju promenljive njenom inicijalizacijom, tako da parser prihvata i
   `int a = 0;`

8. Izmeniti miniC parser tako da u telu funkcije bude moguće naizmenično pisati deklaracije promenljivih i iskaze, na primer:

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

9. Proširiti miniC jezik logičkim izrazima `&&` i `||` i proširiti miniC parser.

10. Proširiti miniC jezik i miniC parser definicijama funkcija sa više parametara, kao i pozivima funkcija sa više argumenata.