Programski prevodioci: Vežbe 4

Sadržaj

1. Uvod

Na ovim vežbama biće prikazana implementacija detekcije semantičkih grešaka.

2. Dodatne semantičke vrednosti tokena

```
"int"               { yylval.i = INT;  return _TYPE; }
"unsigned"          { yylval.i = UINT; return _TYPE; }
```

Kao što je rečeno na prethodnim vežbama, u cilju jednostavnije implementacije parsera, skener za određene regularne izraze vraća isti token. Na primer, bilo da skener prepozna tip `int` ili tip `unsigned`, parseru će vratiti isti token `_TYPE`. Po gramatici samog jezika, na svakoj poziciji na kojoj može da se pojavi tip `int`, sigurno može i da se pojavi tip `unsigned`, pa nema potrebe praviti razliku u vraćenim tokenima.

Međutim, što se tiče **semantike** (značenja) programa, jako je bitno kog tipa je neka promenljiva. Prema tome, s obzirom da od ove nedelje uvodimo **semantičku** analizu, potrebno je na neki način razlikovati tip `int` od tipa `unsigned`.

Kako funkcije u C jeziku mogu imati samo jednu povratnu vrednost, a ta povratna vrednost je već iskorišćena za potrebe vraćanja tokena, dodatnu semantičku vrednost skener i parser razmenjivaće preko globalne promenljive. Preciznije, koristiće se specijalna globalna promenljiva sa nazivom

`yylval`.

Ova globalna promenljiva je unija i njena deklaracija se nalazi u `.y` datoteci:

```
%union {
  int i;
  char *s;
}
```

Unije u C jeziku su slične strukturama, s tim što članovi unije dele isti memorijski prostor, pa instanca unije zauzima onoliko prostora koliko zauzima njen najveći član. Za potrebe ovog predmeta, ova razlika nije od posebnog značaja, pa možete jednostavno zamisliti da se radi o običnoj strukturi.

Prikazana deklaracija govori da dodatna semantička vrednost po potrebi može biti ili tipa `int` ili tipa `char*`.

Ako se vratimo na prethodni primer, može se videti da se u član `i` promenljive `yylval` dodeljuje vrednost `INT` ili `UINT` u zavisnosti od prepoznatog tipa.

Vrednosti `INT` i `UINT` su vrednosti enumeracije deklarisane, na sledeći način:

```
enum types { NO_TYPE, INT, UINT };
```

Kako se enumeracije u C jeziku implicitno konvertuju u `int`, dozvoljeno je ove vrednosti dodeliti u `yylval.i` koji je tipa `int`.

Zbog kompletnosti, u listingu ispod prikazana je i implementacija za aritmetičke i relacione operatore, ali važi ista priča kao i za tipove `int` i `unsigned`.

```
"+"                 { 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; }
```

Uz tokene `_ID`, `_INT_NUMBER` i `_UINT_NUMBER` se takođe vezuje dodatna vrednost:

```
[a-zA-Z][a-zA-Z0-9]* { yylval.s = strdup(yytext); ①
                       return _ID; }
[+-]?[0-9]{1,10}     { yylval.s = strdup(yytext); ②
                       return _INT_NUMBER;}
[0-9]{1,10}[uU]      { yylval.s = strdup(yytext); ②
                       yylval.s[yyleng-1] = 0;
                       return _UINT_NUMBER;}
```

① Dodatna vrednost je sam naziv promenljive koji je prepoznat.

② Dodatna vrednost je sam broj koji je prepoznat.

U odnosu na prethodne tokene, razlika je u tome što su sada dodatne vrednosti tipa `char*` i dodeljuju se u `yylval.s`.

3. Deklaracije tokena

Prilikom deklaracije tokena potrebno je navesti da li oni imaju uz sebe vezanu neku dodatnu semantičku vrednost i kog tipa je ta vrednost:

```
%token <i> _TYPE ①
%token _IF
%token _ELSE
%token _RETURN
%token <s> _ID ②
%token <s> _INT_NUMBER ②
%token <s> _UINT_NUMBER ②
%token _LPAREN
%token _RPAREN
%token _LBRACKET
%token _RBRACKET
%token _ASSIGN
%token _SEMICOLON
%token <i> _AROP ①
%token <i> _RELOP ①
```

① Tokeni koji imaju semantičku vrednost tipa `int`, u deklaraciji imaju navedeno `<i>`.

② Tokeni koji imaju semantičku vrednost tipa `char*`, u deklaraciji imaju navedeno `<s>`.

4. Pristup semantičkim vrednostima iz akcija u `.y` datoteci

Nećemo koristiti neki poseban alat pomoću kojeg će se vršiti semantička analiza, već će se ova funkcionalnost implementirati kroz ručno pisani C kod u okviru akcija u `.y` datoteci.

U okviru akcija u `.y` datoteci moguće je pristupiti semantičkoj vrednosti vezanoj uz token preko

specijalnih "dolar" promenljivih.

Primer:

`.l` *datoteka:*

```
···         { yylval.i = 14; return _T1; }
···         { yylval.s = "hello world"; return _T2; }
```

`.y` *datoteka:*

```
%token <i> _T1
%token <s> _T2

%%

pojam
  : _T1 _T2
  {
    printf("%d", $1); // ispisuje broj "14"
    printf("%s", $2); // ispisuje se string "hello world"
  }
  ;
```

U okviru akcije postoje specijalne "dolar" promenljive: `$1`, `$2`, `$3` ... `$n`. Broj `n` označava semantičku vrednost `n`-tog tokena, redom kojim su navedeni u pravilu.

5. Semantičke vrednosti pojmova

Ne samo da tokeni mogu imati semantičku vrednost, već to mogu imati i pojmovi.

`.y` *datoteka:*

```
%type <i> pojam2 ①

%%

pojam1
  : pojam2
  {
    printf("%d", $1); // ispisuje se broj "14" ③
  }
  ;
```

```
pojam2
  : _T1 _T2
  {
    $$ = 14; ②
  }
  ;
```

① Pojmu `pojam2` će biti dodeljena semantička vrednost tipa `int`, pa je to potrebno navesti u deklaraciji pojma.

② Pojmu se dodeljuje semantička vrednost, tako što se u poslednjoj akciji dodeli neka vrednost promenljivoj `$$`.

③ Semantičkoj vrednosti pojma se pristupa isto kao i vrednosti tokena, preko "dolar" promenljivih.

6. Semantičke vrednosti akcija

Pored tokena i pojmova, same akcije mogu imati semantičku vrednost.

`.y` datoteka:

```
pojam
//   $1      $2        $3
  : _T1 { $<i>$ = 123; } _T2
  {
    printf("%d", $1); // ispisuje se vrednost tokena _T1
    printf("%d", $<i>2); // ispisuje se vrednost same akcije
    printf("%d", $3); // ispisuje se vrednost tokena _T2
  }
  ;
```

Vrednost akcije se postavlja tako što se u okviru same akcije dodeli neka vrednost u `$$` promenljivu.

Akcije se ne deklarišu na vrhu `.y` datoteke, pa se tip njihove semantičke vrednosti navodi između dva `$` znaka, kao što je prikazano u primeru iznad.

Obratiti pažnju da `$$` promenljiva ima različito značenje kada se koristi u okviru akcije koja se nalazi na samom kraju pravila i svih ostalih akcija.

Akcija na kraju pravila	**Sve ostale akcije**
Promenljiva `$$` označava vrednost **pojma** sa leve strane pravila	Promenljiva `$$` označava vrednost same **akcije**

7. Tabela simbola

Primer jedne semantičke provere koju treba implementirati je provera da li je korišćena promenljiva prethodno deklarisana.

Primer:

```
int main() {
  a = 14; // GRESKA: promenljiva `a` nije deklarisana.
  return 0;
}
```

Da bismo znali da li je promenljiva prethodno deklarisana, potrebno je sve deklarisane promenljive čuvati negde u memoriji.

Za potrebe čuvanja deklarisanih simbola, uvodi se tabela simbola.

Jedan red u tabeli simbola:

```
typedef struct sym_entry {
   char *   name; ①
   unsigned kind; ②
   unsigned type; ③
   unsigned atr1; ④
   unsigned atr2; ④
} SYMBOL_ENTRY;
```

① Naziv simbola.

② Vrsta simbola.

③ Tip vrednosti simbola.

④ Dodatni atributi simbola.

Svaki red u tabeli simbola predstavlja 1 simbol i može da se implementira kao gore navedena struktura ili kao neka druga struktura podataka.

Za dodavanje novog simbola može se koristiti sledeća funkcija:

```
int insert_symbol(char *name, unsigned kind, unsigned type,
                  unsigned atr1, unsigned atr2); ①
```

① Funkcija dodaje simbol na kraj tabele i vraća indeks simbola. Indeks predstavlja red u kojem je simbol dodat.

Pronalaženje simbola u tabeli simbola obavlja se sledećom funkcijom:

```
int lookup_symbol(char *name, unsigned kind); ①
```

① Ova funkcija prihvata naziv i vrstu simbola, po kojem vrši pretragu. Povratna vrednost je indeks pronađenog simbola. Ako simbol uopšte ne postoji u tabeli, povratna vrednost je `-1`.

Po pronalaženju simbola, moguće je pristupiti željenim kolonama preko `get` i `set` funkcija:

```
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);
```

Obratiti pažnju da svi geteri i seteri prihvataju **indeks** u tabeli simbola. Dakle, prvo je potrebno pozvati `lookup_symbol` funkciju, pa dobijeni indeks iskoristiti pri pozivu getera i setera.

8. Primer jedne semantičke provere

U ovom poglavlju dat je primer jedne semantičke provere koja objedinjuje sve prethodno objašnjeno gradivo.

Uzmimo za primer pojam `variable`, koji predstavlja deklaraciju jedne lokalne promenljive. Semantička provera za ovaj deo gramatike je to da promenljiva ne sme biti prethodno deklarisana. Drugim rečima, ne možemo imati dve lokalne promenljive sa istim nazivom.

```
variable
  : _TYPE _ID _SEMICOLON
      {
        if(lookup_symbol($2, VAR|PAR) == -1) ①
           insert_symbol($2, VAR, $1, ++var_num, NO_ATR); ②
        else
           err("redefinition of '%s'", $2); ③
      }
  ;
```

① Funkcija `lookup_symbol` prihvata naziv i vrstu simbola za pretragu. Na mestu naziva, prosleđena je vrednost promenljive `$2`. Vrednost promenljive `$2` je semantička vrednost vezana uz token `_ID`, a to je sam naziv promenljive. Vrsta simbola je navedena kao `VAR|PAR` što označava da pretražujemo samo lokalne promenljive i parametre. Ukoliko je dobijena povratna vrednost jednaka broju `-1`, to znači da simbol nije pronađen u tabeli simbola.

② U slučaju da se simbol nije već nalazio u tabeli, to znači da nema greške i da treba tekući simbol dodati u tabelu. Dodavanje radimo pozivom funkcije `insert_symbol`.

③ U slučaju da simbol već postoji u tabeli, došlo je do greške i treba je prijaviti korisniku. Prijavljivanje grešaka radi se pomoću makroa `err`. Ovaj makro se koristi identično kao i `printf`.

9. Kompajliranje, pokretanje i testiranje

Kao i prethodne nedelje, rešenje se kompajlira i pokreće na sledeći način:

```
make
./semantic
```

Kako bismo izbegli da prilikom testiranja svaki put ručno kucamo test primer, moguće je test primer iskucati u nekoj datoteci, a zatim upotrebom redirekcije proslediti tu datoteku na standardni ulaz programa:

```
./semantic < ulazna_datoteka
```

Takođe, `Makefile` je napravljen tako da može automatski pozvati sve napisane testove odjednom. Da bi ovo radilo, test datoteke je potrebno nazvati u sledećem formatu:

Početak naziva datoteke	Značenje
`test-sanity`	Testovi koji proveravaju originalnu gramatiku. Ovi testovi treba uvek da prolaze, čak i ako niste počeli da kucate rešenje zadatka. Ako ovi testovi ne prolaze, to znači da ste rešavanjem zadatka slučajno promenili originalnu gramatiku. Bilo bi lepo da za Vaše projekte imate i ovu vrstu testova.
`test-ok`	Testovi koji sadrže korektan program sa novim konstrukcijama.
`test-warn`	Testovi koji sadrže korektan program u kojem se prijavljuje upozorenje.
`test-synerr`	Testovi koji sadrže nekorektan program sa jednom sintaksnom greškom.
`test-semerr`	Testovi koji sadrže nekorektan program sa jednom semantičkom greškom.

Svi testovi treba da imaju ekstenziju Vašeg programskog jezika.

Nakon što su testovi napravljeni, potrebno je samo ukucati sledeću komandu:

```
make test
```

Navedena komanda će automatski pokrenuti sve testove i prikazati koji testovi prolaze (zeleni), a koji ne prolaze (crveni). Ispravno rešen zadatak treba da sadrži samo zelene testove.

Da bi dobili više informacija o testovima, potrebno je samo ukucati sledeću komandu:

```
make det
```

Navedena komanda će automatski pokrenuti sve testove i prikazati koji testovi prolaze, a koji ne prolaze, uz detalje o prijavljenim greškama ako one postoje.

Da bi obrisali izvršne fajlove, što je poželjno pre testiranja da bi bili sigurni da su izgenerisani novi, potrebno je samo ukucati sledeću komandu:

```
make clean
```

10. Zadaci

10.1. Zadatak 1

Proširiti gramatiku tako da se u jednoj deklaraciji može deklarisati više promenljivih odvojenih zarezom.

Realizovati semantičke provere:

1. Promenljiva ne sme biti prethodno deklarisana,
2. Lokalni identifikatori moraju biti jednoznačni.

10.2. Zadatak 2

Proširiti (samo) miniC izraze postinkrement operatorom.

Realizovati semantičku proveru:

1. Postinkrement operator može da se primeni samo na promenljive i parametre (a ne, recimo, na funkcije).

10.3. Zadatak 3

Proširiti miniC gramatiku `do while` iskazom.

Sintaksa `do while` iskaza ima oblik:

```
"do"
    <statement>
"while" "(" <id> <relop> <lit> ")" ";"
```

- `<statement>` predstavlja jednu naredbu ili blok naredbi
- `<id>` predstavlja identifikator

- `<relop>` predstavlja jedan od relacionih operatora
- `<lit>` predstavlja literal

Realizovati semantičke provere:

1. `<id>` mora biti postojeca promenljiva ili parametar
2. `<id>` i `<lit>` moraju biti istog tipa.

10.4. Zadatak 4

Proširiti miniC gramatiku višestrukim dodelama:

```
a = b = c = d = e + 5;
```

Pri tome, za svaki znak `=`, obe strane jednakosti moraju imati isti tip i svi identifikatori moraju biti definisani.