Filip Miletić

JAFFA: Prenosivi kompajler

„Jaffa" je prenosivi kompajler za podskup Paskala. Prenosivost se ogleda u tome što je jasno razdvojen prednji deo (front end) prevodioca koji čita izvorni kod i zadnji deo (back end), koji od međukoda što ga prethodni deo generiše pravi asemblerski listing. Ovaj se listing kasnije može propustiti kroz simbolički asembler konkretne mašine i tako dobiti pravi binarni kod. Konkretna mašina mora biti računar sa registrima i kontinualnom memorijom kojoj se može pristupati u rezoluciji jednog bajta. Relativno jednostavnom izmenom zadnjeg dela može se izvesti da Jaffa daje izlaz koji se može izvršiti na različitim računarima.
Većina Jaffe svoj posao obavlja nad simbolički predstavljenim međukodom koji se konstruiše od izvornog teksta programa. Međukod je sastavljen od instrukcija koje su zajedničke najvećem broju danas raspoloživih računara. Ideja Jaffe je da se generiše međukod koga je moguće prevesti na što veći broj mašinskih jezika, a da se tek tokom generacije asemblerskog listinga pokušaju primetiti idiomatski oblici instrukcija, koji se u skupu instrukcija ciljne mašine mogu kraće zapisati.

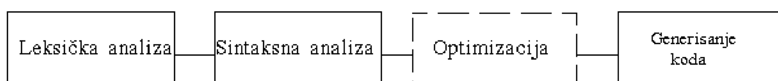
Uvod

Problem sastavljanja prevodilaca za programske jezike je do danas dobro poznat. Algoritmi koji se koriste u njihovoj konstrukciji su tako uklopljeni da predstavljaju elegantnu celinu i uopšte uzev nisu se značajno menjali od vremena kada je napravljen prvi prevodilac. Moderan prevodilac će prema opštem modelu biti sačinjen od sledećih delova: leksičkog analizatora, sintaksnog analizatora i generatora koda (slika 1).

Leksički analizator prolazi kroz izvorni kod (tekst) programa, prepoznajući u tome tekstu ključne reči, promenljive, relacione operatore, komentare itd. Svakoj od ovih kategorija pridružuje se jednoznačno identifikator koji se kasnije koristi u obradi njene sintaktičke vrednosti.

Za ovu obradu zadužen je drugi deo programa, *sintaksni analizator*, koji na osnovu skupa pravila kog nazivamo *kontekstno slobodna gramatika*, prepoznaje konstrukcije jezika. Za kontekstno slobodnu gramatiku koristi se i samo termin „gramatika".

Filip Miletić (1978), Kruševac, Ratka Šakotića 39, učenik 4. razreda Gimnazije u Kruševcu
filmil@afrodita.rcub.bg.ac.yu



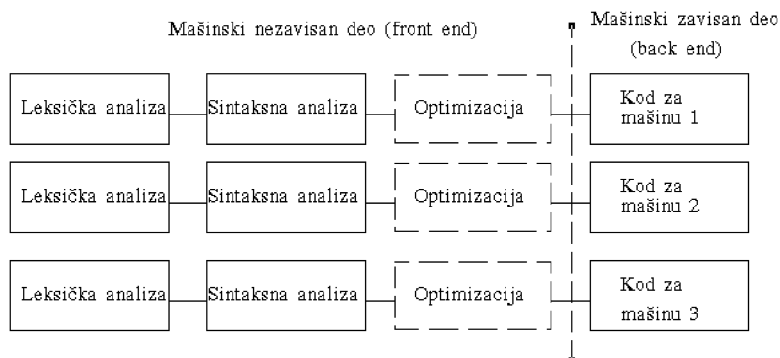
Slika 1. Dijagram toka prevođenja.

Figure 1. Phases of the Compiler.

Treći deo prevodioca je *generator koda* koji prema skupu semantičkih akcija koje su pridružene sintaktičkim konstrukcijama daje konačni oblik prevedenog programa.

Posebna klasa prevodilaca pre poslednjeg dela ima umetnut i *optimizator koda*. To je deo koji koristi određen skup tehnika za analizu koda i toka njegovog izvršavanja, da bi pokušao da stvori ekvivalentan kod koji je po nekome kriterijumu bolji od polaznog. Za te kriterijume uzimaju se npr. dužina zapisa ili brzina izvršavanja. Zavisno od toga kako je prevodilac osmišljen i koji su ciljevi razmatrani pri njegovom projektovanju, ovi delovi mogu biti manje ili više prepleteni. Potpuna rasprava o fazama u prevodiocima može se naći kod Ahoa i saradnika (Aho *et al*. 1986).

Primetimo da je osnovni plan svih prevodilaca veoma sličan. Pogled na sliku 2 otkriva suštinu problema koji se razmatra u ovom radu. Većina faza prevodilaca obavlja identičnu funkciju tako da je prava razlika isključivo u implementaciji. Ovaj deo se zato razumljivo zove *mašinski nezavisnim*. Manji deo prevodioca u koji po pravilu spada generator koda ne može biti isti za sve mašine usled njihove velike različitosti. Razlike mogu biti u obliku i ustrojstvu mašinskog koda, raspoloživosti registara, postojanju steka itd. Ovaj deo je izrazito mašinski zavisan. U literaturi se mašinski nezavisan i mašinski zavisan deo redom označavaju kao „prednji (front end) i „zadnji (back end) deo.



Slika 2. Podela faza prevođenja na prednji i zadnji deo.

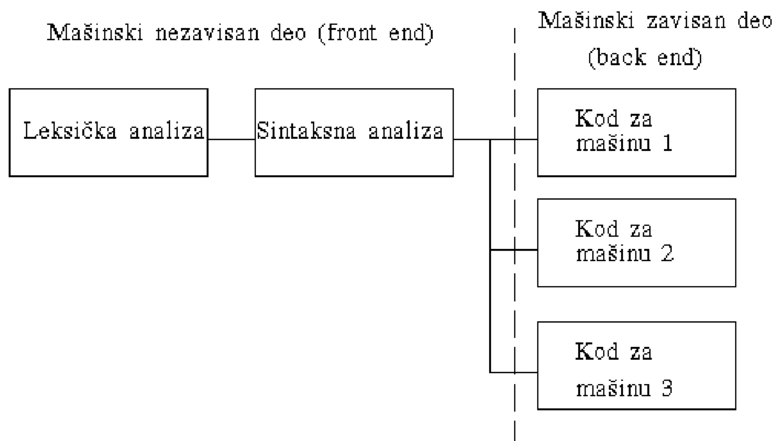
Figure 2. The Front and Back End of the Compiler.

Ako bismo objedinili delove zajedničke svim prevodiocima a zatim samo dopisivali one koji su različiti, efektivno bi se smanjio obim prevodilaca a i njihovo pisanje bilo bi olakšano. Dve različite razrade ove ideje mogu se upoznati u literaturi (GNU 1993; Yuen 1997). U ma kom slučaju dolazi se do potrebe da se na formalan način opišu arhitekture mašina za koje se prevodi. Pošto su arhitekture računara beznadežno različite, zapis

formalizma postaje jako obiman ukoliko se želi izražajna mogućnost koja dozvoljava podršku velikoj paleti mašina. Primer je upravo GNU C kompajler (GNU 1993a) u kome je jezik načinjen u ovu svrhu toliko narastao da je dokazivanje korektnosti opisa postao obiman matematički problem. Primeri se mogu naći u izvornom kodu za GNU-C 2.7.0 (GNU 1993c).

Ovaj rad opisuje novi pristup problemu prenosivih kompajlera. Intenzivno se koriste već definisani skupovi pravila kako bi se izbeglo ponavljanje konstrukcija. Generacija koda je dinamička: konstruiše se stablo koje predstavlja zavisnost između elemenata međukoda a zatim po njemu traže oni delovi čiji oblik odgovara određenom pravilu. Ukoliko se pronađe takav deo, izvrši se akcija koja je određena za njega a cela konstrukcija zameni ekvivalentom.

U ovom radu se obrađuje nekoliko problema. Predstavljene su u kratkim crtama tabele podataka koje se koriste pri prevođenju. Opisane su karakteristične sintaktičke konstrukcije, odgovarajuće semantičke akcije i dat je pregled međukoda. Dat je predlog formalizma za opis konkretne mašine i konačni automat koji ovaj opis pretvara u generator koda.



Slika 3. Šema JAFFE

Figure 3. JAFFA Layout

Implementacija

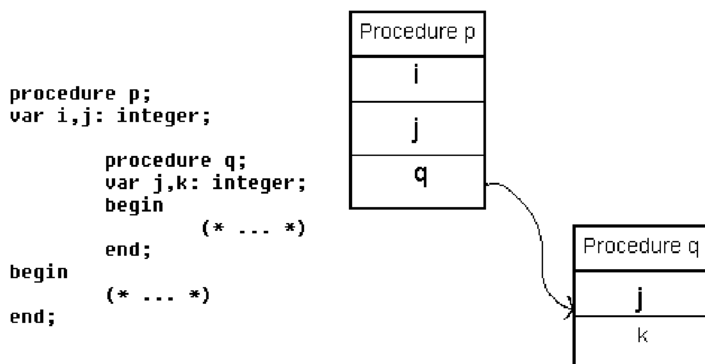
Program koji pokriva izloženu problematiku zove se JAFFA. Urađen je u programskom jeziku C uz korišćenje programskih alata LEX – korišćen je Flex (1995) i GNU Bison, generator parsera kompatibilan sa Berkeley YACC-om (GNU 1993b). JAFFA prihvata korektan tekst programa u podskupu Paskala. Formalna gramatika je data u prilogu ovog rada.

Za implementaciju je odabran C zato što od klasičnih programskih jezika u najvećoj meri ima osobinu prenosivosti a i zbog toga što za njega postoji skup veoma kvalitetnih alata koji su korišćeni pri radu na JAFFI. Za prenosivost C-a odgovorni su uglavnom tradicionalni razlozi kao što je

njegova rasprostranjenost na raznim platformama. Razumljive su prednosti nekih drugih programskih jezika (izraziti primer je Modula-2) koje se ogledaju u čitkosti i visokom nivou izvornog koda. Međutim, ova prednost je u kontrapoziciji sa praktično slabim poštovanjem standarda ovog jezika. Čemu prenosivi prevodilac ako sam njegov kod nije prenosiv?

Rad sa tabelama simbola

Bitna funkcija prevodioca je pamćenje identifikatora koji se koriste u tekstu programa i pamćenje raznih atributa koji u vezi sa njima. Struktura koja služi ovoj svrsi naziva se tabelom simbola. Ona sadrži po jedan slog za svaku od promenljivih koje koristimo u programu. Kod JAFFE je tabela simbola složena tako da je omogućena identifikacija leksičkog opsega života promenljive. Ilustracija koja pokazuje ovo data je na slici 4.



Slika 4. Leksički opseg života promenljive.

Figure 4. Lexical scope.

Svaka tabela je realizovana kao hash tabela. Slika 5 predstavlja spisak funkcija koje se koriste za operisanje sa ovom tabelom. Njihovim korišćenjem se apstrahuje unutrašnja struktura tabele i sva interakcija usmerava na određen skup funkcija. U literaturi (Kingston 1990) je dat opis jednog takvog apstraktnog tipa podataka koji je i korišćen pri implementaciji.

```
typedef bucket elem[MAX_HASH];

typedef struct
{
     int width;
     elem tab;
} sym_tab;

sym_tab  *sym_init( sym_tab **t);
void sym_kill( sym_tab *t);
tab_elem  *sym_insert( sym_tab *t, char *key, void *elem);
tab_elem *sym_delete( sym_tab *t, tab_elem *e);
void sym_add_width( sym_tab *t, int w);
tab_elem *sym_find ( sym_tab *t, char *key);
```

Slika 5. Apstraktni tip tabele simbola (ADT).

Figure 5. The Symbol Table Abstract Data Type (ADT).

Leksička analiza

U JAFFI, leksička analiza je poverena programu Flex. Flex je vrlo efikasan skaner-generator koji uzima specifikaciju skanera kao ulaz dok na izlazu daje izvorni kod konačnog automata koji prepoznaje jezik definisan specifikacijom. Algoritmi na kojima se Flex zasniva su složeni, pa ne bi bilo racionalno kodirati ih ručno (Aho *et al.* 1986). Na slici 6 je u skraćenoj formi dat opis skanera koga koristi JAFFA. To je ujedno i spisak svih ključnih reči koje JAFFA prepoznaje i spisak svih tokena.

Opis međukoda

Kada je namena prevodioca proizvodnja izvornih kodova za više mašina, pokazuje se nepodesno održavati poseban prevodilac za svaki od ciljnih mašinskih jezika. Zajedničko rešenje ovog problema pojavljuje se u različitim formalnim, ali suštinski istim oblicima u vidu međukoda (Aho *et al.* 1986; GNU 1993b; Flex 1995). Međukod je zapis koji se produkuje na osnovu semantičkih akcija (Aho *et al.* 1986) i predstavlja prevod teksta programa na jezik apstraktne mašine. Većina faza prevodioca operiše isključivo nad ovom reprezentacijom programa.

Za potrebe JAFFA prevodioca koncipiran je međukod čije su instrukcije tako odabrane da odgovaraju velikom skupu registarski orijentisanih mašina. Međukod je sastavljen na osnovu podataka iznetih u literaturi (Donovan 1986; Lyu 1987; Beck 1988; Flex 1995). To je skup elementarnih instrukcija za operacije sa memorijom, registrima, Bulovom i celobrojnom aritmetikom. Klase mašina na koje se može direktno primeniti su računari sa linearnom memorijom, posebnim memorijskim elementima – registrima koji su operandi većine aritmetičkih operacija. Kompozicije ovih osnovnih instrukcija ne razmatramo pošto će one biti otkrivene u fazi generisanja koda.

Jedan od načina zapisivanja međukoda je troadresni kod (Aho *et al.* 1986; Donovan 1986; Beck 1988). To je apstraktan oblik međukoda koji je u JAFFI interno predstavljen kao niz uređenih četvorki. Polja uređene četvorke zovu se redom: *opcode*, *result*, op_1 i op_2. Tako se operacija

$$x := y \text{ op } z$$

zapisuje tako što se kao *opcode* stavlja vrednost op , kao op_1 vrednost promenljive y i kao op_2 vrednost z.

Neki kodovi su nepotpuni u tom smislu da ne koriste sve tri raspoložive adrese. Tako na primer instrukcija *.param* koja opisuje parametar koji se prosleđuje proceduri ima samo jedan argument i to je promenljiva op_1, dok se preostala dva polja ne koriste. Sve naredbe međukoda mogu se podeliti prema nameni u nekoliko grupa i to:

Slika 6 (naspramna strana). Ključne reči.

Figure 6 (opposite page). Keywords.

```
LETTER      [a-zA-Z]
WS    [ \t]
NLINE   [\n]
SPACE   {WS}+
DIGIT [0-9]
XDIGIT      [0-9a-fA-F]
XNUM  {XDIGIT}+
INTEGER     {DIGIT}+
ID    {LETTER}({LETTER}|{DIGIT})*
REAL  {INTEGER}"."{INTEGER}
EXPON ({INTEGER}|{REAL})("e"|"E")("+"|"-")?{INTEGER}
STRCONST ({LETTER}|{DIGIT}|{WS})*


%%


{SPACE}
{NLINE}     line_count ++;
"(*"  comment = 1;
"*)"  comment = 0;
";"   if ( ! comment) return ( ';' );
","   if ( ! comment) return ( ',' );
":"   if ( ! comment) return ( ':' );
"("   if ( ! comment )return ( '(' );
")"   if ( ! comment ) return ( ')' ) ;
"["   if ( ! comment ) return ( '[') ;
"]"   if ( ! comment ) return ( ']') ;
"="   if ( ! comment ) return ( '=');
"<"   if ( ! comment ) return ( '<');
">"   if ( ! comment) return ( '>');
"<>"  if ( ! comment) return ( L_NE);
"<="  if ( ! comment) return ( L_LE);
">="  if ( ! comment)   return ( L_GE);
"in"  if ( ! comment) return ( IN);
"+"   if ( ! comment)   return ( '+' );
"-"   if ( ! comment)   return ( '-' ) ;
"or"  if ( ! comment) return ( L_OR);
":="  if ( ! comment ) return ( L_ASSIGN);
"^"   if ( ! comment ) return ( '^');
".."  if ( ! comment ) return ( L_DDOT);
"."   if (  ! comment )return ( '.' );
"not" if ( ! comment ) return ( L_NOT);
"*"   if ( ! comment ) return ( L_MUL);
"/"   if ( ! comment ) return ( '/' );
"div" if ( ! comment ) return ( L_DIV);
"mod" if ( ! comment ) return ( L_MOD);
"and" if ( ! comment ) return ( L_AND);
```

(nastavak na sledećoj strani)

```
"program"   if ( ! comment ) return PROGRAM;
"label"     if ( ! comment ) return L_LABEL;
"const"     if ( ! comment ) return CONST;
"type"      if ( ! comment ) return TYPE;
"packed"    if ( ! comment ) return PACKED;
"file"      if ( ! comment ) return L_FILE;
"of"        if ( ! comment ) return OF;
"array"     if ( ! comment ) return ARRAY;
"record"    if ( ! comment ) return RECORD;
"begin"     if ( ! comment ) return L_BEGIN;
"end"       if ( ! comment ) return END;
"case"      if ( ! comment ) return CASE;
"set"       if ( ! comment ) return SET;
"var"       if ( ! comment ) return VAR;
"procedure" if ( ! comment ) return PROCEDURE;
"forward"   if ( ! comment ) return FORWARD;
"function"  if ( ! comment ) return FUNCTION;
"nil"       if ( ! comment ) return NIL;
"if"        if ( ! comment ) return IF;
"then"      if ( ! comment ) return THEN;
"else"      if ( ! comment ) return ELSE;
"while"     if ( ! comment ) return WHILE;
"do"        if ( ! comment ) return DO;
"repeat"    if ( ! comment ) return REPEAT;
"until"     if ( ! comment ) return UNTIL;
"to"        if ( ! comment ) return TO;
"downto"    if ( ! comment ) return DOWNTO;
"true"      if ( ! comment ) return TRUE;
"false"     if ( ! comment ) return FALSE;
"byte"      if ( ! comment ) return BYTE;
"char"      if ( ! comment ) return CHAR;
"integer"   if ( ! comment ) return INTEGER;
"word"      if ( ! comment ) return WORD;
"longint"   if ( ! comment ) return LONGINT;
"Boolean"   if ( ! comment) return BOOLEAN;
"real"      if ( ! comment ) return REAL;
"for"       if ( ! comment ) return FOR;
"write"     if ( ! comment ) return WRITE;
"writeln"   if ( ! comment ) return WRITELN;
"read"      if ( ! comment ) return READ;
"readln"    if ( ! comment ) return READLN;
"'"{STRCONST}"'" if ( ! comment ) {sscanf( yytext, "'%[^']", &buf);
yylval.id = strdup(buf);return STRCONST; }
{INTEGER}   if ( ! comment ) {sscanf( yytext, "%d", &(yylval.c));
return INTCONST;}
{REAL}      if ( ! comment ) {sscanf( yytext, "%g",
&(yylval.d)); return REALCONST; }
{ID}        if ( ! comment ) {sscanf( yytext, "%s", &buf);
yylval.id = strdup( buf);return ID; }
.           raise("unrecognized character: '%s'", yytext);
```

1. *Bulove funkcije* su standardno operacije: *or*, *and*, *not*. Operacije *or* i *and* su binarne i uzimaju operande tipa *Boolean*. Nad ovim promenljivima se vrši provera tipa. Rezultat je takođe vrednost tipa *Boolean* i predstavlja vrednost respektivne Bulove funkcije. *Not* je unarna operacija i njen argument je tipa *Boolean*. Rezultat je takođe tipa *Boolean* i predstavlja logičku negaciju izraza.

JAFFA proširuje skup Bulovih funkcija sa: *jg*, *jl*, *je*, *jne*, *jge* i *jle*. Ove funkcije porede dva argumenta istog tipa i daju izlaz tipa *Boolean*.

2. *Aritmetičke operacije* su: *plus*, *minus*, *imod*, *idiv*, i*mul, fmul*, *fdiv*, *uminus*. Imena instrukcija su davana prema uobičajenim konvencijama (GNU 1994). Funkcije koje imaju f u prefiksu operišu sa realnim brojevima i njihov rezultat je takođe realan broj. Funkcije sa i u prefiksu operišu sa celim brojevima. Njihov rezultat je takođe ceo broj. Funkcije bez prefiksa (*plus*, *minus*, *uminus*) mogu da operišu sa bilo kojim tipom brojeva.

3. *Operacije dodele*: *assign*, *rmove*, *cmove*. *Assign* služi za kopiranje vrednosti promenljive proizvoljnog tipa na novo mesto. *Rmove* i *cmove* dodeljuju promenljivama realne odnosno celobrojne konstante.

4. *Operacije sa memorijom*: *addr*, *deref* i [ ] služe za indirektan pristup memorijskim lokacijama.

5. *Naredbe skoka*: *goto*, *jtrue* i *jfalse*. *Goto* je uobičajena oznaka za bezuslovan skok. *Jtrue* i *jfalse* označavaju uslovne skokove prema istinosnoj vrednosti argumenta.

6. *Posebne i makro naredbe* spadaju u kategoriju onih koje su se našle u definiciji međukoda iz praktičnih razloga. To su redom: *.param, .call, .return*, *.arg*, *.local*, *.proc*, *.endp*, *.note*. Prefiks . (tačka) ističe njihovu posebnu namenu.

.param, *.call*, *.local*, *.arg*, i *.return* se koriste pri pozivu procedura. One se najčešće realizuju kao sekvenca naredbi realne mašine i u tom smislu se izdvajaju od prethodno opisanih.

.proc i *.endp* uokviruju početak i kraj procedure i čuvaju njeno ime.

.note je generalni komentar. Ova se naredba ignoriše pri prevođenju. Služi samo da bi se zapis međukoda učinio jasnijim. Označava početke i ključna mesta kontrolnih struktura.

7. *Labele* označavaju mesta koja su odredište naredbi skoka. Označavaju se znakom dvotačke (:) i jedinstvenim nizom znakova.

Argumenti u međukodu mogu biti registri, memorijske lokacije ili konstante. Registri se označavaju speciajalnim prefiksom i brojem odgovarajućeg registra. Memorijske lokacije su označene odgovarajućom labelom. Konstante su predstavljene njihovom neposrednom vrednošću.

Sintaksna analiza

Već je napomenuto da JAFFA predstavlja podskup programskog jezika Paskal. Uvešćemo notaciju pod nazivom kontekstno slobodna gramatika (skraćeno: gramatika) da bismo opisali sintaksu jezika (Aho *et al*. 1986). Gramatika na prirodan način opisuje hijerarhijsku strukturu mnogih programskih jezika. Na primer, *if-else* struktura u C-u ima oblik:

IF (*expression*) *stmt ELSE stmt*

Znači, struktura je niz koji se sastoji od ključne reči *if*, otvorene zagrade, izraza, zatvorene zagrade, naredbe, ključne reči *else* i još jedne naredbe. Ako iskoristimo promenljivu *expr* da bismo označili izraz, ovo pravilo se može zapisati kao

stmt $\rightarrow$ *IF* (*expr*) *stmt ELSE stmt*

gde se strelica može čitati kao „ima oblik . Ovakvo pravilo se zove produkcija. U produkciji leksički elementi kao što je ključna reč *if* i zagrade nazivaju se tokenima. Promenljive kao *expr* i *stmt* predstavljaju nizove tokena i zovu se neterminalni simboli.

Kontekstno slobodna gramatika ima četiri komponente:

1. skup tokena, koji predstavljaju terminalne simbole
2. skup neterminalnih simbola
3. skup produkcija gde se svaka sastoji od neterminalnog simbola, koji se zove levom stranom produkcije i niza terminalnih i neterminalnih simbola koji odgovaraju levoj strani produkcije; niz se naziva *desnom stranom produkcije*
4. istaknut jedan neterminalni simbol kao početni, startni, simbol.

Po konvenciji navodimo gramatiku tako što redom ispišemo produkcije koje je sačinjavaju i to tako da je startna produkcija prva u tom spisku. Pretpostavljamo da cifre, znaci kao npr. relacioni operator (<=), i reči ispisane velikim slovima predstavljaju terminalne simbole. Dozvoljena je i disjunkcija uz pomoć znaka | (pipe) koja se čita kao „ili (Aho *et al*. 1986).

Shema prevođenja je kontekstno slobodna gramatika sa uvršćenim fragmentima koda koji određuju semantičke akcije. Semantička akcija pridružena određenoj produkciji piše se desno od pripadajuće produkcije. Primer je gramatika na slici 7, koja opisuje aritmetički izraz u kojem se javljaju četiri osnovne računske operacije.

Ovde je *E.val* atribut pridružen produkciji. Iz primera se vidi da se vrednost ovog atributa određuje na osnovu vrednosti istog atributa neterminalnih simbola sa desne strane produkcije. Primećujemo da se prve izvode semantičke akcije spregnute sa onim neterminalnim simbolom koji se

```
E → E1 + E2 {E.val = E1.val + E2.val}
E → E1 - E2 {E.val = E1.val - E2.val}
E → E1 * E2 {E.val = E1.val * E2.val}
E → E1 / E2 {E.val = E1.val * E2.val}
E → ( E1 )  {E.val = E1.val}

E → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 9

    {E.val = $1.val
     1 → {"1".val = 1}
          ...
```

Slika 7. Aritmetički izraz u obliku formalne gramatike.

Slika 7. Arithmetic expression presented as formal grammar.

prvi pojavljuje sa desne strane produkcije. Ovo je naročito važno pri prenošenju nasleđenih atributa (Aho *et al*. 1986) o čemu će posebno biti reči kada se govori o prevođenju izraza koji opisuju pristup nizovima. Cela gramatika JAFFE je data u prilogu. Detaljnije se diskutuju samo semantičke akcije koje su od posebnog interesa za proces prevođenja.

Referisanje promenljivih

Sve promenljive, bez obzira kojeg tipa bile, mogu se podeliti u dve kategorije: *jednostavne promenljive*, za pristup kojima je dovoljno znati samo identifikator i *nizove*, za koje je osim imena (koje čuva i informacije o baznoj adresi na kojoj se nalazi niz) potrebno znati i indeks elementa sa kojim želimo da operišemo. Pogledajmo izraz kojim se ostvaruje referisanje promenljive samo po njenom imenu:

$$VariableReference \rightarrow ID$$

Promenljivoj se, uz konsultovanje tabele simbola, pristupa preko njene lokacije.

Elementima niza se može lako pristupati ukoliko su smeštene u uzastopnim memorijskim lokacijama. Ako je veličina jednog elementa niza jednaka w, onda se i-ti element nalazi na lokaciji:

$$base + (i - low) \cdot w$$

gde je *low* donja granica elementa niza, a *base* bazna adresa toga niza. Ukoliko ovaj izraz prepišemo na drugačiji način, dobijamo:

$$i \cdot w + (base - low \cdot w).$$

Izraz

$$c_1 = base - low \cdot w$$

je konstantan za jedan niz pa se može izračunati u vreme kompilacije. Vrednost izraza c_1 se smešta u tabelu simbola kao atribut niza. Da bi se dobila adresa elementa kog tražimo, dovoljno je izračunati $i \cdot w + c_1$ u

vreme izvršavanja programa. Ukoliko je niz dvodimenzionalan, oblika $A[i_1, i_2]$, pristup elementu $A[I, J]$ se može napisati kao:

$$base + \Big((i_1 - low_1) n_2 + i_2 - low_2 \Big) \cdot w$$

gde su low_1 i low_2 donje granice mogućih vrednosti i_1 i i_2 a n_1 i n_2 brojevi vrednosti koje i_1 i i_2 mogu uzeti. Ova se forma lako generalizuje za nizove $A[i_1, i_2, i_3, \ldots, i_k]$ na sledeći način:

$$\Bigg(\Big(\ldots\Big(\Big(i_1 n_2 + i_2 \Big) n_3 + i_3 \Big) \ldots \Big) n_k + i_k \Bigg) \cdot w +$$
$$+ \, base - \Bigg(\Big(\ldots\Big(\Big(low_1 n_2 + low_2 \Big) n_3 + low_3 \Big) \ldots \Big) n_k + low_k \Bigg) \cdot w$$

Ukoliko se želi pristup nizovskim promenljivama, produkcija koja to ostvaruje mogla bi se zapisati kao:

```
VariableReference → ArrayReference
ArrayReference → ID [ DimList ]
DimList → DimList1, Dim { DimList.ndim := ndim + 1;
                          DimList.dim  := DimList1.dim + newdim( Dim.dim);
                    }
DimList → Dim               { DimList.ndim := 1;
                          DimList.dim  := Dim.dim
                    }
```

Korišćeni atributi su *ndim*, koji označava broj dimenzija niza, i *dim*, koji može predstavljati uređenu *n*-torku tipova dimenzije niza. Primetimo da se pri svakoj produkciji, vrednosti atributa produkcije izvode iz atributa neterminala s desne strane produkcije. Ovakvi atributi nazivaju se *sintetisanim*.

Glavni problem kod generisanja koda za obraćanje elementima niza je povezati izračunavanje formule i odgovarajuću gramatičku konstrukciju. Da bi granice nizova bile na raspolaganju pri izračunavanju izraza, korisno je prepisati odgovarajuću grupu produkcija kao:

```
VariableExpression → ArrayReference ]
ArrayReference → ArrayReference, Expr
ArrayReference → ID [ Expr
```

Produkcija *ArrayReference* za prvih *m* indeksa *k*-dimenzionog niza

$$A\,[\, i_1, i_2, \ldots, i_k \,]$$

računaće se prema izrazu (Aho *et al*. 1986)

$$e_m = \Big(\ldots\Big(\Big(i_1 n_2 + i_2 \Big) n_3 + i_3 \Big) \ldots \Big) n_m + i_m$$

uz pomoć rekurzivnog izraza:

$$e_1 = i_1 \;, \quad e_m = e_{m-1}\, n_m + i_m$$

Rešavanje leksičkog opsega života promenljivih

Paskal, te prema njemu i JAFFA, ima tzv. leksički opseg života promenljivih. Pogledajmo sledeći fragment programa:

```
procedure a;
var i, j: integer;

      procedure b;
      var i, k: integer;

      begin
            i := 1;
            k := 2;
            j := i + 2*k;
      end;
end;
```

Primetićemo da se unutar procedure *b* operiše sa promenljivom j koja je deklarisana van opsega procedure *b* i to unutar *a*. Po pravilu, procedura koja je ugnežđena unutar *k* procedura može pristupati promenljivama koje su deklarisane unutar svih njih, dok obrnuto ne važi. To se naziva *leksičkim opsegom života* promenljive. Skupljanje informacija o opsegu života ilustrovano je u primeru koji sledi. Unutar semantičkih akcija koriste se sledeće procedure (Aho *et al*. 1986):

1. *mktable* (*previous*) kreira novu tabelu simbola i vraća pointer na nju. Argument *previous* pokazuje na prethodno kreiranu tabelu. Ovaj pointer se čuva unutar nove tabele i time efektivno kreira hijerarhijsko uređenje ovih dveju tabela.
2. *enter* (*table*, *name*, *type*, *offset*) kreira novi ulaz za ključ *name* u tabeli simbola na koju pokazuje *table*, *type* i *offset* su atributi koji redom određuju tip i adresu labele određene ključem.
3. *addwidth* (*table*, *width*) pamti zbirnu dužinu svih ulaza u tabelu u njenom zaglavlju.
4. *enterproc* (*table*, *name*, *newtable*) kreira novi ulaz za proceduru s imenom name u tabelu simbola koju određuje table. Argument *newtable* pokazuje na tabelu simbola pridruženu toj proceduri.

U semantičkim akcijama koriste se još i stekovi s imenom *tblptr* i *offset*. Skup operacija nad njima je standardan za ovaj apstraktni tip i dat je u literaturi (Aho *et al*. 1986; Donovan 1986; Kingston 1990). U nekim produkcijama se javlja terminal ε (epsilon). To je nemi terminalni simbol koji ne odgovara direktno nijednom tokenu, ali se za njega može vezati semantička akcija. Kada se naiđe na neterminal koji ga sadrži, odgovarajuća semantička akcija se odmah izvršava. Atributi koji se javljaju su redom:

a) *name*: označava ime vezano za dati atribut;
b) *type*: označava tip vezan za taj atribut;
c) *width*: označava veličinu promenljive.

```
P → M D     { addwidth( tblptr), top(offset));
              pop( tblptr); pop(offset) }
M → ε             { t := mktable( nil);
              push( t, tblptr); push ( 0, offset);
            }
D → D1 ; D2       {
D → PROC ID ; N D1 ; S { t := top(tblptr);
                     addwidth( t, top(offset));
                     pop(tblptr); pop( offset);
                     enterproc(top(tblptr), ID.name, t) }
D → ID : T              { enter(top(tblptr), id.name, T.type,
top(offset));
                    top(offset) := top( offset) + T.width; }
N → ε                   { t := mktable( top(tblptr));
                    push(t, tblptr); push( 0, offset); }
```

Prema definiciji semantičkih akcija, ako je data produkcija

$$A \to B\ C\ \{a(A)\}$$

gde je $a(X)$ oznaka za semantičku akciju koja odgovara neterminalu X. Redosled izvršavanja produkcija dat je sa: $(a(B), a(C), a(A))$ (Aho *et al.* 1986).

Generisanje konačnog koda

Generacija koda je proces pri kome se zapis međukoda preslikava u zapis koji predstavlja konačni izlaz prevođenja. Međukod se kod JAFFE predstavlja u obliku niza uređenih četvorki oblika: (*opcode*, op_1, op_2, *result*). Ovaj oblik je veoma pogodan pri stvaranju međukoda i većini manipulacija sa njime zbog svoje pravilnosti. Pri generaciji koda ćemo zadržati ovaj oblik prezentacije ali ćemo informacije koje on nudi interpretirati na donekle izmenjen način.

Kažemo da, ako ni $opcode_1$ ni $opcode_2$ nisu instrukcije skoka, uređena četvorka

$$A = \left(opcode_1, op_{11}, op_{12}, result_1\right)$$

zavisi od uređene četvorke

$$B = \left(opcode_2, op_{21}, op_{22}, result_2\right)$$

ako i samo ako važi

$$op_{11} = result_2 \lor op_{12} = result_2$$

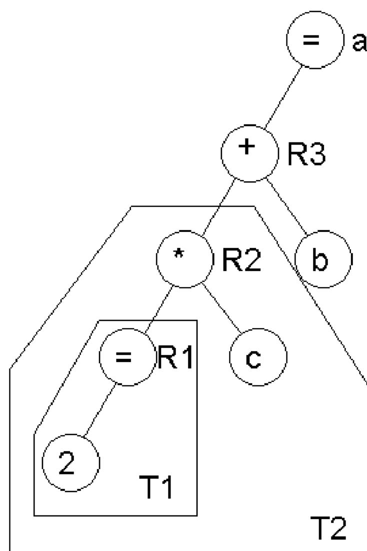
Stablo zavisnosti T je opisano kao uređena trojka $T = (A, T2, T3)$ gde je A uređena četvorka međukoda, a $T2$ i $T3$ su stabla zavisnosti. Ukoliko ne postoje takva stabla $T2$ i $T3$, stablo pišemo kao $(A, *, *)$. Posmatrajmo kako izgleda zapis izraza

$$a := b + 2 * c$$

u međukodu:

(1) (*cmove*, 2, *, $R1$)
(2) (*imul*, $R1$, c, $R2$)
(3) (*plus*, $R2$, b, $R3$)
(4) (*assign*, $R3$, *, a)

Prema gore izloženom, od ovog niza uređenih četvorki može se načiniti stablo zavisnosti. Stablo je grafički prikazano na slici 8.



Slika 8. Grafički prikaz stabla

Figure 8. The tree.

U svakom čvoru stabla zapisana je operacija koja se koristi za dobijanje vrednosti čvora. Pored svakog čvora zapisano je ime promenljive ili registra koji se koristi za smeštanje dobijenog rezultata. Pročitamo li ovo stablo u pre-orderu (Kingston 1990), dobijamo:

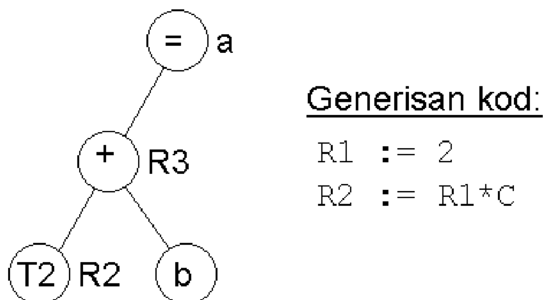
$$T \rightarrow =(a)\ +(R3)\ *(R2)\ =(R1)\ C(2)\ V(c)$$

gde smo celo stablo imenovali sa T. Na sličan način možemo zadati $T1$ i $T2$ kao:

$$T1(R1) \rightarrow =(R1)\ C(2)$$
$$T2(R2) \rightarrow *(R2)\ T1(R1)\ V(c)$$

Ovde je sa $T1(R1)$ i $T2(R2)$ naznačeno gde će se naći rezultat operacije. Primetimo i da je u ovom primeru $T2$ izraženo preko već definisanog podstabla $T1$. Možemo videti produkcije gramatike koja je proistekla iz ove akcije. Ako napišemo skup ovako zadatih pravila i njima pridružimo odgovarajuće semantičke akcije, dobijamo formu za opis konkretne mašine. Proces generisanja koda svodi se na konstruisanje stabla zavisnosti i uzastopno pronalaženje podstabala čiji se preorder zapisi poklapaju sa ne-

kom od produkcija u formi. Kada se pronađe podudaran deo izvršava se semantička akcija dodeljena produkciji, a celo podstablo zamenjuje čvorom koji nosi oznaku jednaku imenu produkcije. Na primer, posle ove zamene podstabla *T*2, novo stablo i generisan kod izgledaju kao na slici 9.



Slika 9. Novo stablo i generisan kod.

Figure 9. The view of new tree and the code.

Kod određivanja produkcije koja odgovara podstablu osim poznavanja stabla treba znati i dopunske uslove. Ovi dopunski uslovi se zapisuju unutar zagrada. Tako, produkcija *T*2(*R*2) znači: produkcija *T*2 koja smešta rezultat u registar *R*2. Efektivno, ovim je uvedena nova operacija *T*2 koja je ravnopravna sa ostalim elementarnim operacijama poput sabiranja i oduzimanja. Možemo uvesti i uslov *C*(*n*), gde je *n* neka konstanta. Tako bi se *T*1(*C*(2)) čitalo: operacija *T*1 vraća vrednost konstante 2. *C*(*) bi značilo ma koju konstantnu vrednost, a *V*(*x*) promenljivu. Produkcije *T*1 i *T*2 su u ovom slučaju neterminalni simboli, dok su operacije (+, –, *), konstante i promenljive terminalni simboli.

Sukcesivnim primenama ovih pravila stablo se može redukovati do jednog jedinog čvora. Uslov za to je očigledno da postoji makar jedna produkcija kojom se instrukcija međukoda može prevesti. Ukoliko načinimo produkcije u kojima se kao argumenti javljaju druga podstabla, kao što je to učinjeno sa podstablima *T*1 i *T*2, dolazimo do mogućnosti da deo stabla većeg od elementarnog obradimo jednom produkcijom što daje mogućnost prepoznavanja idiomatskih konstrukcija. U formalnom smislu, gramatike koje nastaju na ovaj način su nejednoznačne. Zato je, da bi se ovaj problem razrešio, uvedeno jedno ograničenje: produkcije ne mogu biti duže od 3. To znači da se jedna produkcija ne može razviti na više od tri neterminalna ili terminalna simbola. Ako se želi složenija produkcija, ona se mora razbiti na određen broj jednostavnijih. U primeru je produkcija *T*2 zadata na ovaj način.

Prema ovde izloženom može se konstruisati konačni automat koji prepoznaje podstabla. Sledi algoritam po kome se na osnovu definisanih produkcija izgrađuje konačni automat. U opisu algoritma koriste se sledeće procedure:

1. *Firstprod*(p) i *nextprod*(p) učitavaju produkciono pravilo iz ulazne datoteke.
2. *Newstate* je funkcija koja vraća novo stanje konačnog automata.
3. *Firstterm*(p) vraća vrednost prvog tokena u produkciji p. Sledeći token dobija se pozivom funkcije *nextterm*(p). Ove funkcije vraćaju ili vrednost traženog tokena ili *nil* ako ona ne postoji.
4. *Move*(m, *current*, *tok*) vraća ono stanje automata m u koje se preslikava tranzicija m(*current*, *tok*) ili *nil* ako takvo stanje ne postoji.
5. *Markterm*(s) označava stanje s kao prihvatljivo (accepting). Ova funkcija smešta i semantičku akciju u tabelu stanja.
6. *Isterm*(s) vraća vrednost *True* ako je stanje s jedno od prihvatljivih krajnjih stanja.

Algoritam 1. Izgradnja tabele tranzicija na osnovu zadate gramatike. Ulaz: Gramatika koja opisuje podstabla koja prepoznaje automat. Izlaz: Skup stanja i tabela funkcije m: $(state, token) \rightarrow state$ koja preslikava uređene parove (*state*, *token*) u skup stanja.

Tekst algoritma 1:

1. Inicijalizacija: m = $\varnothing$; *start* = nil; p = nil.
2. Ucitaj prvu produkciju: *Firstprod*(p).
3. Mora postojati pocetno stanje: *start* = *newstate*.
4. Dok važi p $\neq$ nil ponavljaj do koraka 11.
5. Postavi pocetno stanje automata: s = *start*; t = *Firstterm*(p).
6. Dok važi $t \neq$ nil ponavljaj do koraka 9.
7. Ako važi *Move*(m, s, t) = nil, tada ne postoji odgovarajuća tranzicija i zato je dodaj u skup m: $m = m \cup \left(\left(s, tright\right), newstate\right)$. Idi na 9.
8. Ako važi *Isterm*(s) onda javi grešku, jer terminalno stanje ne može imati tranzicije.
9. Predi u novo stanje i uzmi sledeci token: s = *Move*(m, s, t); t = *nextterm*(p).
10. Označi trenutno stanje kao terminalno: *Markterm*(s).
11. Uzmi sledeću produkciju: *nextprod*(p).

Produkcije za generisanje koda se primenjuju pri analizi stabla zavisnosti. U tu svrhu definišemo automat (Aho *et al*. 1986) Q koji ima sledeće karekteristike:

1. Stek S koji čuva trenutno stanje mašine
2. Stek H koji služi za odlaganje hendlova (handle)
3. Podatak L koji čuva sledeći simbol (lookahead)
4. Funkcija m(*state*, *tok*) koja vraća stanje koje odgovara prelazu definisanim sa (*state*, *tok*), gde je *state* stanje a *tok* vrednost ulaznog simbola

Ulaz u algoritam je stablo zavisnosti koje se čita u preorder režimu, čvor po čvor. Operacija *shift* znači postavljanje simbola L na stek i učitavanje sledećeg preorder čvora u L. Operacija *reduce* po nekoj produkciji znači skidanje hendla koji odgovara toj produkciji. Neterminalni simbol pridružen produkciji postavlja se u L. Izvršava se pridružena semantička akcija ako postoji.

Algoritam 2. Prepoznavanje podstabla po metodu bottom-up.

Ulaz: Stablo zavisnosti T načinjeno prema međukodu.

Izlaz: Redosled semantičkih akcija (instrukcija) koje izračunavaju datu sekvencu međukoda na jeziku ciljne mašine.

1. Inicijalizacija L na prvi preorder čvor stabla T. Inicijalizacija stekova S i H. Inicijalizacija stanja na početnu vrednost.
2. *Shift*: simbol L se prebacuje na stek H i učitava novi. Ako nema više simbola program je završen.
3. Ako je trenutno stanje terminalno, izvrši *reduce*, a zatim *pop*(S). Idi na 2.
4. Ako je *top*(H) terminal koji znači operaciju, pređi u stanje m(S, *top*(H)), izvrši *push*(S), pređi u početno stanje i idi na 2.

Zaključak

U ovom radu opisali smo ključne aspekte u razradi prevodioca. Poseban akcenat je bio stavljen na algoritme koji se koriste za generisanje koda. Može se proveriti da generator koda koji je ovde izložen koristi algoritam niske složenosti (Kingston 1990). Koncept po kome se podstablima pridružuju semantičke akcije je dovoljno opšti da čini proširivanje skupa mašina na koje ovaj algoritam može da se primeni mogućim. Korišćenjem mogućnosti da se produkcije koje se javljaju kao podskup većeg broja drugih složenijih produkcija navode u opisu mašine samo jedanput, za razliku od modela u literaturi (GNU 1993a; 1993c).

Literatura

Aho, A.V., Sethi, R., Ullman, J.D. 1986. *Compilers: Principles, Techniques and Tools*. Addison-Wesley.

Beck, L.L. 1988. *Vvedenie v sistemnoe programirovanie*. Moskva: Mir

Donovan, J. 1986. *Systems Programming*. Addison-Wesley.

Flex 1995. *The Fast Lexical Scanner Generator*. Berkeley: Lawrence Berkeley Labs.

GNU 1993a. *The GNU C Compiler Documentation*. GNU Free Software Foundation Info Pages

GNU 1993b. *The GNU Bison Parser Generator Documentation*. GNU Free Software Foundation Info Pages

GNU 1993c. Izvorni kod GNU C 2.7.0. GNU Free Software Foundation

GNU 1994. *The GNU Assembler Info Pages*. GNU Free Software Foundation

Lyu, Y.C. 1987. *Mikroprocesory semeistva 8086/8088*. Moskva: Radio i svyaz.

Kingston, J.H. 1990. *Algorithms And Data Structures: Design, Correctness and Analysis*. Sydney: University of Sydney.

Yuen, A. 1997. Retargetable Concurrent Small C, *Dr. Dobb s Journal*, (August 1997): 58-70

Filip Miletić

JAFFA: A Portable Compiler

JAFFA is a compiler model which has increased portability as its primary aim. It is set up consisting of two distinct parts, the so called front and the back end. The front end accepts correct Pascal subset source code input, transforming it into an intermediate code representation. During this first phase the symbol table, scope and type information tables are built and prepared for further use. The second phase utilizes the information provided by the tables, the intermediate code and a pattern matcher driven by a set of semantic rules to output final code. The semantic rules are easily modifiable, enabling the compiler to output code for a diverse set of computers. These computers are assumed to be register machines with byte-addressable memory.

The implementation done in C, using the LEX and GNU Bison scanner and parser generators. Portability is achieved by clearly distinguishing the front and the back end of the compiler, the first reading the source text and the second one producing assembly output. This output can be submitted to a symbolic assembler of a real machine to produce binary code.

Issues of compiler construction are well known today. Algorithms employed in their construction are fitted in such a way to form an elegant compound and, in general, have not changed since the days the first compilers were built. A modern compiler consists of the following distinct parts: a lexical analyzer, a syntax analyzer and a code generator; a special class of compilers have an extra module fitted between the latter two: a code optimization engine.

The basic layout of all the compilers is rather similar. If we could retain the parts of the compilers that are common for all of them, later

adding only the parts that effectively differ, we would reduce the size of compiler sources, and ease the implementation. That way it becomes necessary to devise a formal way of describing machine characteristics by means of rulesets.

This paper describes a new approach to portable compilers issue. The constructed rulesets are intensely used to avoid repetitions in code generation. Code generation is dynamic: a dependency tree, showing interconnections between elements of intermediate code, is constructed and afterwards subjected to exaustive search to find a subtree that matches a rule. If such a subtree is found, a semantic action attached to the rule is executed, and the whole subtree is substituted with the equivalent.

Most of JAFFA manipulates with symbolic intermediate code which is constructed from the source program text. Intermediate code consists of instructions which are common to most computers available today. The leading idea behind JAFFA is to generate intermediate code that is easily translated to a variety of machine languages, and only during generation of final code to try noticing idiomatic constructs that can be written concisely in target machine instruction set.

There are several issues discussed in this paper. There is a layout of the data tables used in the compiler. Semantic actions for the parser and the intermediate code proposal are presented. Proposals for the machine description, and the finite automaton which produces the code generator are given.

Prilog

```
program --> PROGRAM ID InputOutputList ; ProceduralBody .
InputOutputList --> : ( ID ) | ( ID , ID )
ProceduralBody --> TypeDeclaration VarDeclaration ProcOrFunc StatementBlock
ProcOrFunc --> /* empty */ | ProcOrFunc ; | ProcOrFunc ; | ProcDecl | FunDecl
TypeDeclaration --> /* empty */ | TYPE TypeList ;
TypeList --> TypeList ; TypeAssign | TypeAssign ;
TypeAssign --> ID '=' VariableType
VariableType -->  SimpleType | ArrayType
SimpleType --> ID | BOOOLEAN | INTEGER | LONGINT | WORD     | REAL | CHAR | BYTE
IdList --> IdList , ID | ID
ArrayType --> ARRAY [ SimpleTypeList ] OF VariableType
SimpleTypeList --> SimpleTypeList , SimpleTypeP | SimpleTypeP
SimpleTypeP --> Subrange | SimpleType
Subrange --> s_or_ui L_DDOT s_or_ui
s_or_ui --> INTCONST | '-' INTCONST
VarDecSeq --> VarDecSeq ; VarAssignment | VarAssignment
VarAssignment --> IdList : VariableType
VarDeclaration --> /* empty */ | VAR VarDecSeq ;
ProcDecl --> PROCEDURE ID ParDecList
ParDecList --> /*empty */ | ( ParDecList1 )
ParDecList1 --> ParDecList1 ; Pard |
Pard --> VAR ProcVar | ProcVar
ProcVar --> ProcVar , ProcVarAs      | ProcVarAs
ProcVarAs --> IdList : VariableType
FunDecl --> FUNCTION ID ParDecList : SimpleType ; ProceduralBody
StatementBlock --> L_BEGIN StSeq END
StSeq -->  StSeq ';' Statement | Statement
Statement --> AssignStat | ProcCall | StatementBlock | CompoundStat
AssignStat --> VariableReference L_ASSIGN Expr
VariableReference --> ArrayReference ]       | ID
ArrayReference --> ArrayReference ',' Expr | ID '[' Expr
Expr --> SimpleExpr | SimpleExpr = SimpleExpr
        | SimpleExpr L_NE SimpleExpr
        | SimpleExpr L_GE SimpleExpr
        | SimpleExpr L_LE SimpleExpr
        | SimpleExpr '<'  SimpleExpr
        | SimpleExpr '>'  SimpleExpr
SimpleExpr --> Terms | Terms '+' Terms
        | Terms '-' Terms
        | Terms L_OR Terms
Terms --> Factor | Factor '*' Factor
        | Factor '/' Factor
        | Factor L_DIV Factor
        | Factor L_MUL Factor
        | Factor L_MOD Factor
        | Factor L_AND Factor
        | '-' Factor %prec L_UMINUS
Factor --> UnsignedConstant | VariableReference     | ProcCall
        | '(' Expr ')'        | L_NOT Factor
UnsignedConstant --> REALCONST | INTCONST | TRUE | FALSE
ProcCall --> ID ( ParList )

ParList --> ParList , Expr | Expr
CompoundStat -->  IfStatement|WhileStatement|RepeatStatement|ForStatement
IfStatement --> IF Expr THEN Statement | IF Expr THEN Statement ELSE Statement
WhileStatement -->  WHILE Expr DO Statement
RepeatStatement --> REPEAT StSeq UNTIL Expr
ForStatement --> FOR ID L_ASSIGN Expr TO  Expr DO Statement
        | FOR ID L_ASSIGN Expr DOWNTO Expr DO Statement
```