

**POLITECHNIKA ŁÓDZKA**

Katedra Mikroelektroniki i Technik Informatycznych

**Techniki kompilacji**

Rok akademicki 2021/2022

Projekt końcowy

Kompilator języka Pascal – objaśnienie wybranych  
elementów gramatyki

Mateusz Domalązek

239517

# Operacja na tablicach

variable: IDENTIFIER { \$\$ = \$1 }  
| IDENTIFIER [ expresion ] { \$\$ = \$1 [\$3] }

Table 1: Przykład operacji na tablicach

<pre>program sort(input,output); var p :array [1..10] of integer; begin p[8] := 1 end.</pre>	<ul style="list-style-type: none"><li>0 Global procedure read</li><li>1 Global procedure write</li><li>2 Global label lab0</li><li>3 Global variable p array [1..10] of integer offset=0</li><li>4 Global number 1 integer</li><li>5 Global number 10 integer</li><li>6 Global number 8 integer</li><li>7 Global variable \$t0 integer offset=40</li><li>8 Global number 1 integer</li><li>9 Global number 4 integer</li><li>10 Global reference variable \$t1 integer offset=44</li><li>11 Global number 1 integer</li></ul>
<pre>jump.i #lab0      ;jump.i lab0 lab0: sub.i #8, #1, 40   ;sub.i 8, 1, \$t0 mul.i 40, #4, 40   ;mul.i \$t0, 4, \$t0 add.i #0, 40, 44   ;add.i &amp;p, \$t0, \$t1 mov.i #1, *44      ;mov.i 1, \$t1 exit              ;exit</pre>	

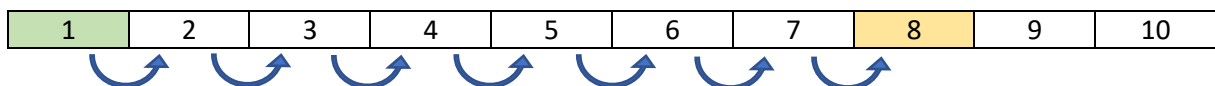
Dlaczego wykonujemy operację odejmowania (SUB)?

sub.i 8, 1, \$t0

$\$t0 = 8 - 1 = 7$

Odejmujemy, aby dowiedzieć się, ile n-następnych elementów występuje po pierwszym elemencie tablicy do elementu docelowego (\$3).

Tablica p (indeksy)




Ósmy element tablicy jest 7 następującym po pierwszym elemencie. Tę informację następnie zapisujemy do zmiennej tymczasowej (\$t0).

Dlaczego wykonujemy operację mnożenia (MUL)?

Musimy wiedzieć, jak to przesunięcie będzie wyrażone w pamięci.

Tablica p

1	2	3	4	5	6	7	8	9	10
4 bajty	4 bajty	4 bajty	4 bajty	4 bajty	4 bajty	4 bajty	P[8]		



Przesunięcie obliczymy poprzez przemnożenie  $\$t0 * 4$  dla tablicy integer lub  $\$t0 * 8$  dla tablicy real

Dlaczego wykonujemy operację dodawania (ADD)?

W celu ustalenia, gdzie w pamięci leży element \$3 potrzebujemy przesunięcia względem początku tablicy. Pierwszy element znajduje się pod indeksem 0.

Takim oto sposobem otrzymaliśmy informację, gdzie jak dostać się do elementu p[8] w tablicy.

Gdybyśmy teraz chcieli wpisać coś do p[8] to przy użyciu operatora assign wiemy, gdzie mamy umieścić nową wartość.

Na przykład:

mov.i 1, \$t1 gdzie zmienna \$t1 przechowuje informację o przesunięciu w pamięci względem początku tablicy a elementem docelowym p[8]

p[8] := 1

Implementacja w kodzie a powyższy przykład:

p odpowiada \$1, natomiast \$3 odpowiada expression czyli 8

```
575 IDENTIFIER '[' expression ']'
576
577
578 int id = 0;
579 if(arrayOfSymbols.getType($1) != ARRAY){
580     yyerror((arrayOfSymbols.getSymbolName($1) + " - expected ARRAY").c_str());
581     YYERROR;
582 }
583
584 if(arrayOfSymbols.getType($3) == ARRAY){
585     yyerror((arrayOfSymbols.getSymbolName($3) + " - expected INT or REAL elements inside []").c_str());
586     YYERROR;
587 }
588
589 //convert from real to int
590 if(arrayOfSymbols.getType($3) == REAL){
591     int tmp_id = arrayOfSymbols.newTemp(REAL);
592     emitter.realToInt(arrayOfSymbols.syntable[$3], arrayOfSymbols.syntable[tmp_id]);
593
594     id = emitter.addop(arrayOfSymbols.syntable[tmp_id], SUB,
595         arrayOfSymbols.syntable[arrayOfSymbols.syntable[$1].blockDescription.firstID]);
596 }else{
597     id = emitter.addop(arrayOfSymbols.syntable[$3], SUB, arrayOfSymbols.syntable[arrayOfSymbols.syntable[$1].blockDescription.firstID]);
598 }
599
600 string t = (arrayOfSymbols.syntable[$1].blockDescription.dataType== REAL ? "#8" : "#4");
601 cout << "Typ " << t << endl;
602 emitter.write("mul.i\t" + emitter.addressFormat(arrayOfSymbols.syntable[id]) + ", " + t + ", " + emitter.addressFormat(arrayOfSymbols.syntable[id]),
603 "mul.i\t" + emitter.addressFormat(arrayOfSymbols.syntable[id], true) + ", " + t + ", " + emitter.addressFormat(arrayOfSymbols.syntable[id], true));
604
605 int element = emitter.addop(arrayOfSymbols.syntable[$1], ADD, arrayOfSymbols.syntable[id],true);
606 arrayOfSymbols.syntable[element].isReference = true;
607 $$ = element;
608
609
610 ;
611
```

Sprawdzenie czy p jest tablicą

Sprawdzenie indeksu - real lub integer

Jeśli jest real to dokonujemy konwersji na int

Wykonujemy operację SUB  
\$t0 = \$3 - firstID

Sprawdźmy typ elementów tablicy (real czy integer)

Wykonujemy operację MUL  
\$t0 = \$t0 \* t

Wykonujemy operację ADD  
\$t1 = &p + \$t0

Teraz możemy odwołać się do elementu docelowego p[8]. Gdybyśmy chcieli użyć operatora przypisania, to wystarczy:

```
mov.i #value, $t1
```

# Instrukcje warunkowe (IF)

statement: ... | IF expression

THEN statement

ELSE statement ;

expression: simpler\_expression

| simpler\_expression RELOP simpler\_expression ;

simpler\_expression to najczęściej liczba lub zmienna

Table 2: Przykład prostej instrukcji warunkowej

<pre>program example(input,output);  var x: integer; begin   x := 2;   if x &gt; 3 then     x := 4   else     x := 5   end.</pre>	<pre>Symbol table dump for main program 0 Global procedure read 1 Global procedure write 2 Global label lab0 3 Global variable x integer offset=0 4 Global number 2 integer 5 Global number 3 integer 6 Global label lab1 7 Global label lab2 8 Global variable \$t0 integer offset=4 9 Global number 0 integer 10 Global number 1 integer 11 Global label lab3 12 Global label lab4 13 Global number 0 real 14 Global number 4 integer 15 Global number 5 integer</pre>
<pre>      jump.i #lab0      ;jump.i lab0 lab0:       mov.i #2,0        ;mov.i 2,x       jg.i 0,#3,#lab1   ;jg.i x,3,lab1       mov.i #0,4        ;mov.i 0,\$t0       jump.i #lab2      ;jump.i lab2 lab1:       mov.i #1,4        ;mov.i 1,\$t0 lab2:       je.i 4,#0,#lab3   ;je.i \$t0,0,lab3       mov.i #4,0        ;mov.i 4,x       jump.i #lab4      ;jump.i lab4 lab3:       mov.i #5,0        ;mov.i 5,x lab4:       exit              ;exit</pre>	

Gdy natrafiamy na token IF to idziemy do expression.

Tam będziemy potrzebowali minimum jednej zmiennej pomocniczej, która przechowa informację czy warunek został spełniony czy nie, a następnie ta wartość zostanie przekazana dalej do \$\$ (expression).

Implementacja w kodzie:

REALOP {\$2}

{\$\$ = \$1 \$2 \$3}

```
385 expression:
386     simpler_expression
387     | simpler_expression REALOP simpler_expression
388     {
389         int boolean_0_1 = arrayOfSymbols.newTemp(INT);
390
391         int jump_if_true = arrayOfSymbols.newLabel();
392
393         emitter.jump(arrayOfSymbols.syntable[$1], $2, arrayOfSymbols.syntable[$3],
394             arrayOfSymbols.syntable[jump_if_true]);
395     }
396
397     int id_false = arrayOfSymbols.addSymbol(Symbol("0", NUMBER, INT, arrayOfSymbols.isGlobalContext()));
398     int finish_label = arrayOfSymbols.newLabel();
399
400     emitter.assign(arrayOfSymbols.syntable[boolean_0_1], arrayOfSymbols.syntable[id_false]);
401     emitter.jump(Symbol(), 0, Symbol(), arrayOfSymbols.syntable[finish_label], true);
402
403     emitter.write(arrayOfSymbols.syntable[jump_if_true].name);
404
405     int id_true = arrayOfSymbols.addSymbol(Symbol("1", NUMBER, INT, arrayOfSymbols.isGlobalContext()));
406
407     emitter.assign(arrayOfSymbols.syntable[boolean_0_1], arrayOfSymbols.syntable[id_true]);
408     emitter.write(arrayOfSymbols.syntable[finish_label].name);
409     $$ = boolean_0_1;
410 }
411 ;
412
413 expression = $t0
```

Utworzenie zmiennej \$t0 do ustawiania

Utworzenie etykiety lbl1, gdzie ustawimy \$t0 na 1

Jeśli \$1 REALOP (\$2) \$3 to skok do lbl1

Utworzenie etykiety lbl2

Dodanie symbolu 0 dla \$t0

Skok z main do lbl2

Wypisanie lbl1:

Dodanie symbolu 1 dla \$t1

Ustawienie \$t0 = 1 (warunek spełniony)

Wypisanie lbl2:

expression = \$t0

```
329 IF expression
330 {
331     int then = arrayOfSymbols.newLabel();
332     int id_0_1 = arrayOfSymbols.addSymbol(Symbol("0", NUMBER, arrayOfSymbols.syntable[$2].type, arrayOfSymbols.isGlobalContext()));
333     emitter.jump(arrayOfSymbols.syntable[$2], EQ, arrayOfSymbols.syntable[id_0_1], arrayOfSymbols.syntable[then]);
334     $2 = then;
335 }
336 THEN statement
337 {
338     int else_ = arrayOfSymbols.newLabel();
339     emitter.jump(Symbol(), 0, Symbol(), arrayOfSymbols.syntable[else_], true);
340
341     emitter.write(arrayOfSymbols.syntable[$2].name);
342     $4 = else_;
343 }
344 ELSE statement
345 {
346     emitter.write(arrayOfSymbols.syntable[$4].name);
347 }
348 ;
349
```

Utworzenie etykiety lbl3

Utworzenie symbolu 0 w celu porównania z \$t0 (expression)

\$2 = id w tablicy symboli gdzie jest lbl3

Jeżeli \$t0 == 0 to skocz do lbl3

Utworzenie etykiety lbl4

Wypisanie lbl3:

Wypisanie skoku do lbl4

\$4 = id w tablicy symboli gdzie jest lbl4

Wypisanie lbl4:

Uwaga: zmienne pomocnicze \$t oraz lbl mogą mieć inną numerację. Tutaj omawiany jest prosty przykład instrukcji warunkowej.

# Pętla WHILE

statement: ... | WHILE

expression DO

statement

expression: simpler\_expression

| simpler\_expression RELOP simpler\_expression ;

simpler\_expression to najczęściej liczba lub zmienna

<pre>program example(input,output);  var x: integer; begin x := 2; while x &lt; 5 do x := x + 1 end.</pre>	<p>Symbol table dump for main program</p> <ul style="list-style-type: none"><li>0 Global procedure read</li><li>1 Global procedure write</li><li>2 Global label lab0</li><li>3 Global variable x integer offset=0</li><li>4 Global number 2 integer</li><li>5 Global label lab1</li><li>6 Global label lab2</li><li>7 Global number 5 integer</li><li>8 Global label lab3</li><li>9 Global label lab4</li><li>10 Global variable \$t0 integer offset=4</li><li>11 Global number 0 integer</li><li>12 Global number 1 integer</li><li>13 Global number 0 real</li><li>14 Global number 1 integer</li><li>15 Global variable \$t1 integer offset=8</li></ul>
<pre>    jump.i #lab0      ;jump.i lab0 lab0:     mov.i #2,0        ;mov.i 2,x lab1:     jl.i 0,#5,#lab3   ;jl.i x,5,lab3     mov.i #0,4        ;mov.i 0,\$t0     jump.i #lab4      ;jump.i lab4 lab3:     mov.i #1,4        ;mov.i 1,\$t0 lab4:     je.i 4,#0,#lab2   ;je.i \$t0,0,lab2     add.i 0,#1,8      ;add.i x,1,\$t1     mov.i 8,0         ;mov.i \$t1,x     jump.i #lab1      ;jump.i lab1 lab2:     exit              ;exit</pre>	

Implementacja w kodzie:

```
309         | WHILE
310         {
311
312             $1 = arrayOfSymbols.newLabel(); //utworzenie etykiety lbl1 i zapisanie jej id do $1
313             $$ = arrayOfSymbols.newLabel(); //utworzenie etykiety lbl2 i zapisanie jej id do $$ Bedzie to ostatnia etykieta dla petli
314
315             emitter.write(arrayOfSymbols.getSymbolName($1)); //wypisanie lbl1:
316
317         }
318         expression DO
319         {
320
321
322             int id_0_1 = arrayOfSymbols.addSymbol(Symbol("0", NUMBER, arrayOfSymbols.syntable[$2].type, arrayOfSymbols.isGlobalContext()));
323             //dodanie symbolu 0 w celu porownania z $t0
324
325             emitter.jump(arrayOfSymbols.syntable[$3], EQ, arrayOfSymbols.syntable[id_0_1], arrayOfSymbols.syntable[$2]); //Jezeli $t0 == 0 to skocz do lbl2
326
327         }
328         statement
329         {
330
331
332             emitter.jump(Symbol(), 0, Symbol(), arrayOfSymbols.syntable[$1], true); //wypisanie skoku do lbl1
333
334             emitter.write(arrayOfSymbols.syntable[$2].name); //wypisanie lbl2:
335
336         }
337     }
```

Spora część kodu została już opisana przy objaśnianiu instrukcji warunkowych. Na niebiesko wygenerowany kod assemblera, który opisano powyżej przy instrukcji if. Na pomarańczowo zaznaczono nowy kod opisany w obecnej sekcji.

jump.i #lab0	;jump.i lab0
lab0:	
mov.i #2,0	;mov.i 2,x
lab1:	
jl.i 0,#5,#lab3	;jl.i x,5,lab3
mov.i #0,4	;mov.i 0,\$t0
jump.i #lab4	;jump.i lab4
lab3:	
mov.i #1,4	;mov.i 1,\$t0
lab4:	
je.i 4,#0,#lab2	;je.i \$t0,0,lab2
add.i 0,#1,8	;add.i x,1,\$t1
mov.i 8,0	;mov.i \$t1,x
jump.i #lab1	;jump.i lab1
lab2:	
exit	;exit



```
%{
    /* definitions of manifest constants
       LT, LE, EQ, NE, GT, GE,
       IF, THEN, ELSE, ID, NUMBER, RELOP */
}%

/* regular definitions */
delim      [ \t\n]
ws         {delim}+
letter     [A-Za-z]
digit      [0-9]
id         {letter}({letter}|{digit})*
number     {digit}+(\.{digit}+)?(E[+-]?{digit}+)?

%%

{ws}       { /* no action and no return */ }
if         {return(IF);}
then       {return(THEN);}
else       {return(ELSE);}
{id}       {yylval = (int) installID(); return(ID);}
{number}   {yylval = (int) installNum(); return(NUMBER);}
"<"       {yylval = LT; return(RELOP);}
"<="     {yylval = LE; return(RELOP);}
"="       {yylval = EQ; return(RELOP);}
"<>"     {yylval = NE; return(RELOP);}
">"       {yylval = GT; return(RELOP);}
">="     {yylval = GE; return(RELOP);}

%%

int installID() { /* function to install the lexeme, whose
                  first character is pointed to by yytext,
                  and whose length is yyleng, into the
                  symbol table and return a pointer
                  thereto */
}

int installNum() { /* similar to installID, but puts numer-
                   ical constants into a separate table */
}
```

Figure 3.23: Lex program for the tokens of Fig. 3.12

**Algorithm 4.19:** Eliminating left recursion.

**INPUT:** Grammar  $G$  with no cycles or  $\epsilon$ -productions.

**OUTPUT:** An equivalent grammar with no left recursion.

**METHOD:** Apply the algorithm in Fig. 4.11 to  $G$ . Note that the resulting non-left-recursive grammar may have  $\epsilon$ -productions.  $\square$

- 1) arrange the nonterminals in some order  $A_1, A_2, \dots, A_n$ .
- 2) **for** ( each  $i$  from 1 to  $n$  ) {
- 3)     **for** ( each  $j$  from 1 to  $i - 1$  ) {
- 4)         replace each production of the form  $A_i \rightarrow A_j \gamma$  by the  
               productions  $A_i \rightarrow \delta_1 \gamma \mid \delta_2 \gamma \mid \dots \mid \delta_k \gamma$ , where  
                $A_j \rightarrow \delta_1 \mid \delta_2 \mid \dots \mid \delta_k$  are all current  $A_j$ -productions
- 5)     }
- 6)     eliminate the immediate left recursion among the  $A_i$ -productions
- 7) }

Figure 4.11: Algorithm to eliminate left recursion from a grammar

Symbol \$

$$E \rightarrow E + T \mid T$$

and their associated semantic actions as:

```

expr : expr '+' term    { $$ = $1 + $3; }
    | term
    ;

```

Note that the nonterminal **term** in the first production is the third grammar symbol of the body, while  $+$  is the second. The semantic action associated with the first production adds the value of the **expr** and the **term** of the body and assigns the result as the value for the nonterminal **expr** of the head. We have omitted the semantic action for the second production altogether, since copying the value is the default action for productions with a single grammar symbol in the body. In general,  $\{ \$\$ = \$1; \}$  is the default semantic action.