

DIM0661-PB1

Grupo 3

9 de abril de 2018

Sumário

1	Introdução	2
2	Regras sintáticas	3
2.1	Gramática	3
3	Termos léxicos	9
3.1	Operadores aritméticos (numericop)	9
3.2	Operadores de conjuntos	9
3.3	Operadores de declaração	9
3.4	Operadores de atribuição	9
3.5	Operadores de comparação (boolop)	9
3.6	Operadores lógicos (boolop)	9
3.7	Literais booleanos	10
4	Regras	11
4.1	id	11
4.2	label	11
4.3	char	11
4.4	string	11
4.5	int	12
4.6	real	12
4.7	Precedência	12
4.8	Regra para comentários	12
5	Forma de uso	13
5.1	Compilação	13
5.2	Execução	13
6	Exemplos	14
6.1	MergeSort	15
6.2	Quicksort	16
6.3	Fatorial	18

Introdução

Este relatório apresenta o manual da linguagem de programação que está sendo desenvolvida na disciplina de Compiladores (DIM0661). A linguagem deve obedecer as seguintes restrições:

- deve ser parecida com Pascal (Pascal-like);
- deve ser em inglês;
- não deve possuir `while` e nem `repeat until`;
- deve possuir um loop geral e permitir uma saída do loop (`exitwhen`);
- deve ter tipagem fraca.

O nome escolhido para a linguagem foi PIE, um acrônimo para Pascal-like (**P**ascal-**lI**ke**E**).

Regras sintáticas

As regras sintáticas da linguagem foram construídas utilizando uma gramática livre de contexto que utiliza o formalismo de Backus-Naur (BNF).

2.1 Gramática

```
1 <prog> ::= program id ';' <decl> <block> '.'
```

```
1 <decl> ::= <consts> <usertypes> <vars> <subprograms>
```

```
1 <consts> ::= ' ' |  
2          const <listconst> ';' ;
```

```
1 <listconst> ::= <constdecl> <listconstprime>]
```

```
1 <listconstprime> ::= ' ' |  
2                  <listconst>
```

```
1 <constdecl> ::= id '=' <expr> ';' ;
```

```
1 <types> ::= id <typesprime> |  
2          <primitives>
```

```
1 <typesprime> ::= ' ' |  
2              '..' <subrangetype>
```

```
1 <primitives> ::= int <primitivesprime> |  
2                real |  
3                bool |  
4                char <primitivesprime> |  
5                string |  
6                <arraytype> |  
7                <settype> |  
8                <enumtype> |  
9                <recordtype>
```

```

1 <primitypesprime> ::= ' ' |
2                       '..' <subrangetype>

```

```

1 <arraytype> ::= array '[' <subrangelist> ']' of <types>

```

```

1 <subrangelist> ::= id '..' <subrangetype> <subrangelistprime> |
2                  int '..' <subrangetype> <subrangelistprime> |
3                  char '..' <subrangetype> <subrangelistprime>

```

```

1 <subrangelistprime> ::= ' ' |
2                      ', ' <subrangelist>

```

```

1 <subrangetype> ::= id      |
2                  int      |
3                  char

```

```

1 <settype> ::= set of <types>

```

```

1 <enumtype> ::= '(' <idlist> ')'

```

```

1 <recordtype> ::= record <varlistlist> end

```

```

1 <usertypes> ::= ' ' |
2              type <listusertypes>

```

```

1 <listusertypes> ::= <usertype> <listusertypesprime>

```

```

1 <listusertypesprime> ::= ' ' |
2                      <listusertypes>

```

```

1 <usertype> ::= id '=' <types> ';'

```

```

1 <vars> ::= ' ' |
2          var <varlistlist>

```

```

1 <varlistlist> ::= <varlist> <varlistlistprime>

```

```

1 <varlistlistprime> ::= ' ' |
2                   <varlistlist>

```

```

1 <varlist> ::= <types> <idlist> ';'

```

```

1 <idlist> ::= id <idattr> <idlistprime>

```

```

1 <idlistprime> ::= ‘ ’ |
2                   ‘,’ <idlist>

```

```

1 <idattr> ::= ‘ ’ |
2            ‘=’ <expr>

```

```

1 <variable> ::= ‘-’ id <variableprime> |
2            ‘[’ <exprlistplus> ‘]’ <variableprime>

```

```

1 <variableprime> ::= ‘ ’ |
2                   <variable>

```

```

1 <block> ::= begin <stmts> end

```

```

1 <stmts> ::= <stmt> <stmtlistprime>

```

```

1 <stmtlistprime> ::= ‘ ’ |
2                   ‘;’ <stmts>

```

```

1 <stmt> ::= ‘ ’ |
2           label <stmt> |
3           <block> |
4           <writestmt> |
5           <writelnstmt> |
6           <readstmt> |
7           <readlnstmt> |
8           <loopblock> |
9           <ifblock> |
10          <forblock> |
11          <caseblock> |
12          <gotostmt> |
13          <exitstmt> |
14          <returnstmt> |
15          id <stmtprime>

```

```

1 <stmtprime> ::= <attrstmt> |
2               <subprogcalls>

```

```

1 <subprogcalls> ::= ‘(’ <exprlist> ‘)’

```

```

1 <exitstmt> ::= exitwhen ‘(’ <expr> ‘)’

```

```

1 <returnstmt> ::= return <expr>

```

```

1 <attrstmt> ::= id <attrstmtprime>

```

1	<code><attrstmt> ::= <variable> ‘:=’ <expr> </code>
2	<code>‘:=’ <expr></code>
1	<code><ifblock> ::= if ‘(’ <expr> ‘)’ <stmt> <elseblock></code>
1	<code><elseblock> ::= ‘ ’ </code>
2	<code>else <stmt></code>
1	<code><loopblock> ::= loop <stmt></code>
1	<code><caseblock> ::= case <expr> of <caselist> <caseblockprime></code>
1	<code><caseblockprime> ::= end </code>
2	<code>else <stmt> end</code>
1	<code><caselist> ::= <literallist> ‘:’ <stmt> ‘;’</code>
1	<code><literallist> ::= <literal> <literallistprime></code>
1	<code><literallistprime> ::= ‘ ’ </code>
2	<code>‘,’ <literallist></code>
1	<code><gotostmt> ::= goto label</code>
1	<code><forblock> ::= for id <forblockprime></code>
1	<code><forblockprime> ::= <variable> ‘:=’ <expr> to <expr> step <expr> do <stmt> </code>
2	<code>‘:=’ <expr> to <expr> step <expr> do <stmt></code>
1	<code><expr> ::= <conj> <disj></code>
1	<code><disj> ::= ‘ ’ </code>
2	<code>‘ ’ <conj></code>
1	<code><conj> ::= <comp> <conjprime></code>
1	<code><conjprime> ::= ‘ ’ </code>
2	<code>‘&&’ <comp></code>
1	<code><comp> ::= <relational> <comprime></code>
1	<code><relational> ::= <sum> <relationalprime></code>
1	<code><sum> ::= <neg> <sumprime></code>

```

1 <sumprime> ::= ' ' |
2           <add_op> <neg> <sumprime>

1 <neg> ::= <mul> |
2       '!' <mul>

1 <mul> ::= <final_term> <mulprime>

1 <mulprime> ::= ' ' |
2             <mul_op> <final_term> <mulprime>

1 <relationalprime> ::= ' ' |
2                  <relational_op> <sum>

1 <comprime> ::= ' ' |
2             <equality_op> <relational>

1 <final_term> ::= id <final_termprime> |
2               <literal> |
3               '(' <expr> ') '

1 <final_termprime> ::= ' ' |
2                   <variable> |
3                   <subprogcall>

1 <add_op> ::= '+' |
2          '-'

1 <mul_op> ::= '*' |
2          '/' |
3          '%'

1 <equality_op> ::= '==' |
2               '!='

1 <relational_op> ::= '<' |
2                 '<=' |
3                 '>' |
4                 '>='

1 <literal> ::= intliteral |
2           realliteral |
3           charliteral |
4           stringliteral |
5           <subrangeliteral>

```



```
1 <exprlist> ::= ' ' |  
2           <exprlistplus>
```

```
1 <exprlistplus> ::= <expr> <exprlistplusprime>
```

```
1 <exprlistplusprime> ::= ' ' |  
2                       ' ,' < exprlistplus>
```

```
1 <subprograms> ::= ' ' |  
2                 <procedure> <subprogramsprime> |  
3                 <function>  <subprogramsprime>
```

```
1 <subprogramsprime> ::= ' ' |  
2                   ' ; ' <subprograms>
```

```
1 <procedure> ::= proc id '(' <param> ')' ' ; ' <decl> <block>
```

```
1 <function> ::= func <types> id '(' <param> ')' ' ; ' <decl> <block>
```

```
1 <param> ::= ' ' |  
2          <varlistlist>
```

```
1 <writestmt> ::= write '(' <expr> ')'
```

```
1 <writelnstmt> ::= writeln '(' <expr> ')'
```

```
1 <readstmt> ::= read '(' id ')'
```

```
1 <readlnstmt> ::= readln '(' id ')'
```

Termos léxicos

```
; ( ) [ ] { } nil program proc begin end func const type var if else goto of  
for to do step in loop exitwhen case write writeln read readln return  
int bool real char string array set record enum subrange
```

3.1 Operadores aritméticos (numericop)

+ - * / %

3.2 Operadores de conjuntos

+ - * = != <= in

3.3 Operadores de declaração

=

3.4 Operadores de atribuição

:=

3.5 Operadores de comparação (boolop)

> < >= <= == !=

3.6 Operadores lógicos (boolop)

&& || !

3.7 Literais booleanos

`true false`

Regras

Meta-operadores para definir as expressões regulares:

- [] : para enumerações associadas à -
- * : repetição
- + : para repetições de um vez ou mais
- ? : de zero a uma vez
- . : como um caractere ‘‘joker’’ exceto \n
- ^ : para o complementar de um []
- | : para representar uma alternativa

4.1 id

Identificadores podem começar apenas com letras, podem ter números e underline (“_”) em sua estrutura. A expressão regular que gera um identificador correto é:

```
id : [a-zA-Z][a-zA-Z0-9_]*
```

4.2 label

```
label : "@"[a-zA-Z0-9_]*
```

4.3 char

```
charliteral : \'[^\']*\'
```

4.4 string

```
stringliteral : \"^[^\n]*\"
```

4.5 int

intliteral : (("-" | "+")? [0-9] +)

4.6 real

exponent : ([E|e] ("+" | "-"))? ({DIGIT} +)

real : ([0-9] * [.])? [0-9] +

realexponent : ([0-9] * [.])? [0-9] + {exponent}?

realliteral : (("-" | "+")? {real} | ("-" | "+")? {realexponent})

4.7 Precedência

A ordem de precedência deve valer para os seguintes operadores (), [], {}, *, /, %, !, +, -, <, <=, >, >=, :=, =, ==, !=, &&, ||. A ordem de precedência pode ser visualizada na Tabela 4.1.

Operador	Precedência
()	maior
[]	
{}	
*, /, %	
!	
+, -	
<, <=, >, >=	
==, !=	
&&	
:=, =	menor

Tabela 4.1: Ordem de precedência para os operadores.

4.8 Regra para comentários

Comentários iniciam por “#” e são eliminados no pré-processamento.

linecomment : "#" (.) * \n

Forma de uso

O código se encontra em <https://github.com/raquel-oliveira/PIE>

5.1 Compilação

```
lex pie.l  
cc lex.yy.c -o name -ll
```

5.2 Execução

```
./name
```

ou

```
./name < pathtofile.pie
```

Exemplos

```
lex pie.l  
cc lex.yy.c -o pie -ll  
./pie < codesamples/merge_sort.pie
```

6.1 MergeSort

```
1  ##
2  # PIE Merge Sort
3  # Pascal version: http://rextester.com/GHRH16649.
4  ##
5  program merge_sort;
6
7  const
8      FIRST = 0;
9      LAST = 9;
10
11  type
12      TRange = FIRST..LAST;
13      TVector = array [TRange] of int;
14
15  var
16      TVector vector;
17      TRange index;
18
19  proc read_arr(TVector vec);
20      var
21          int i;
22      begin
23          writeln("Please give 10 integers as input for the array");
24          for i := FIRST to LAST step 1 do
25              read(vec[i])
26          end;
27
28  proc merge(TVector helper; ref TVector vec; int first, last, center);
29      var
30          int i, j, k;
31      begin
32          i := first;
33          j := center + 1;
34          k := first;
35
36          loop
37              begin
38                  exitwhen (i > center) || (j > last);
39                  if helper[i] < helper[j]
40                      begin
41                          vec[k] := helper[i];
42                          i := i + 1
43                      end
44                  else
45                      begin
46                          vec[k] := helper[j];
47                          j := j + 1
48                      end;
49              end;
```



```

49         k := k + 1
50     end;
51
52     for j := i to center step 1 do
53     begin
54         vec[k] := helper[j];
55         k := k + 1
56     end
57 end;
58
59 proc split_merge(TVector vec; int first, last);
60 var
61     int center;
62 begin
63     if last > first
64     begin
65         center := (first + last) / 2;
66         split_merge(vec, first, center);
67         split_merge(vec, center + 1, last);
68         merge(vec, vec, first, last, center)
69     end
70 end;
71
72 begin
73     read_arr(vector);
74     split_merge(vector, FIRST, LAST);
75     writeln('Sorted vector: ');
76     for index := FIRST to LAST step 1 do
77     begin
78         write(vector[index], ' ')
79     end
80 end.

```

6.2 Quicksort

```

1  ##
2  # PIE Quick Sort
3  # Pascal version: http://sandbox.mc.edu/~bennet/cs404/doc/qsort\_pas.html.
4  ##
5  program quick_sort;
6
7  const
8      FIRST = 0;
9      LAST = 9;
10
11  type
12      TRange = FIRST..LAST;
13      TVector = vecay [TRange] of int;
14
15  var

```

```

16     int i, j, tmp;
17     TVector vector;
18
19 proc read_vec(TVector vec);
20     var
21         int i;
22     begin
23         writeln("Please give 10 integers as input for the vecay");
24         for i := FIRST to LAST step 1 do
25             read(vec[i])
26         end;
27
28 proc quick_sort(TVector vec);
29
30     proc quick_sort_recur(int start, stop);
31         var
32             int m;
33             int splitpt; # The location separating the high and low parts
34
35         # Returns the split point
36         func int split(int start, stop);
37             var
38                 int left, right;
39                 int pivot;
40             proc swap(int a, b);
41                 var
42                     int t;
43                 begin
44                     t := a;
45                     a := b;
46                     b := t
47                 end
48             begin
49                 # Set up the pointers for the hight and low sections, and
50                 # get the pivot value
51                 pivot := vec[start];
52                 left := start + 1;
53                 right := stop;
54                 # Look for pairs out of place and swap em
55                 loop
56                     begin
57                         exitwhen (left > right);
58                         loop
59                             begin
60                                 exitwhen (left > stop) || (vec[left] >= pivot);
61                                 left := left + 1
62                             end;
63                         loop
64                             begin
65                                 exitwhen (right <= start) || (vec[right] < pivot);
66                                 right := right - 1

```

```

67         end;
68         if left < right
69             swap(vec[left], vec[right]);
70         end;
71         # Put the pivot between the halves.
72         swap(vec[start], vec[right]);
73         return right
74     end
75 begin
76     if start < stop
77         begin
78             splitpt := Split(start, stop);
79             quick_sort_recur(start, splitpt - 1);
80             quick_sort_recur(splitpt + 1, stop);
81         end
82     end
83 begin
84     quick_sort_recur(FIRST, LAST)
85 end
86
87 begin
88     read_vec(vector);
89     quick_sort(vector);
90     for i := 1 to size step 1 do
91         writeln(vector[i])
92 end.

```

6.3 Fatorial

```

1  ##
2  # PIE Factorial
3  # Pascal version: http://rextester.com/UXP1971.
4  ##
5
6  program factorial;
7
8  var
9      int num, sum, i;
10
11 begin
12     writeln("Please Input an Integer");
13     readln(num);
14     sum := 1;
15     for i := 1 to num step 1 do
16         sum := sum * i;
17     write('Result: ');
18     write(sum)
19 end.

```

Referências Bibliográficas

- [1] Rextester, “Pascal merge sort.” Disponível em: <http://rextester.com/GHRH16649>. Acessado 12 de março de 2018.
- [2] M. College, “Pascal quick sort.” Disponível em: http://sandbox.mc.edu/~bennet/cs404/doc/qsort_pas.html. Acessado 12 de março de 2018.
- [3] Rextester, “Pascal factorial.” Disponível em: <http://rextester.com/UXP1971>. Acessado 12 de março de 2018.
- [4] J. Jain, “Lexical analyzer for c written in lex.” Disponível em: <https://github.com/jinankjain/Lexical-Analyzer-for-C>. Acessado 12 de março de 2018.