DIM0661-PB1

Grupo 3

9 de abril de 2018

Sumário

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-l**I**k**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

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

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

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

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

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

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

```
<types> ::= id <typesprime> |
            <primtypes>
```

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

```
<primtypes> ::= int <primtypesprime>    |
                real                    |
                bool                    |
                char   <primtypesprime> |
                string                  |
                <arraytype>             |
                <settype>               |
                <enumtype>              |
                <recordtype>
```

```
        <primtypesprime> ::= ''  |
                             '..' <subrangetype>
```

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

```
<subrangelist> ::= id '..' <subrangetype> <subrangelistprime>  |
                   int '..' <subrangetype> <subrangelistprime> |
                   char '..' <subrangetype> <subrangelistprime>
```

```
<subrangelistprime> ::=  '' |
                         ',' <subrangelist>
```

```
<subrangetype> ::= id     |
                   int    |
                   char
```

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

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

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

```
<usertypes> ::= ''                      |
                type <listusertypes>
```

```
<listusertypes> ::= <usertype> <listusertypesprime>
```

```
<listusertypesprime> ::= ''                |
                         <listusertypes>
```

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

```
<vars> ::= ''                 |
           var <varlistlist>
```

```
<varlistlist> ::= <varlist> <varlistlistprime>
```

```
<varlistlistprime> ::= ''            |
                       <varlistlist>
```

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

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

```
<idlistprime> ::= ‘’          |
                  ‘,’ <idlist>
```

```
<idattr> ::= ‘’          |
             ‘=’ <expr>
```

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

```
<variableprime> ::= ‘’          |
                    <variable>
```

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

```
<stmts> ::=  <stmt> <stmtlistprime>
```

```
<stmtlistprime> ::= ‘’ |
                    ‘;’ <stmts>
```

```
<stmt> ::= ‘’              |
           label <stmt>    |
           <block>         |
           <writestmt>     |
           <writelnstmt>   |
           <readstmt>      |
           <readlnstmt>    |
           <loopblock>     |
           <ifblock>       |
           <forblock>      |
           <caseblock>     |
           <gotostmt>      |
           <exitstmt>      |
           <returnstmt>    |
           id <stmtprime>
```

```
<stmtprime> ::= <attrstmt> |
                <subprogcall>
```

```
<subprogcall> ::= ‘(’ <exprlist> ‘)’
```

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

```
<returnstmt> ::= return <expr>
```

```
<attrstmt> ::= id <attrstmtprime>
```

```
<attrstmt> ::= <variable> ':=' <expr> |
               ':=' <expr>
```

```
<ifblock> ::= if '(' <expr> ')' <stmt> <elseblock>
```

```
<elseblock> ::= ''          |
                else <stmt>
```

```
<loopblock> ::= loop <stmt>
```

```
<caseblock> ::= case <expr> of <caselist> <caseblockprime>
```

```
<caseblockprime> ::= end |
                     else <stmt> end
```

```
<caselist> ::= <literallist> ':' <stmt> ';'
```

```
<literallist> ::= <literal> <literallistprime>
```

```
<literallistprime> ::= '' |
                       ',' <literallist>
```

```
<gotostmt> ::= goto label
```

```
<forblock> ::= for id <forblockprime>
```

```
<forblockprime> ::= <variable> ':=' <expr> to <expr> step <expr> do <stmt> |
                    ':=' <expr> to <expr> step <expr> do <stmt>
```

```
<expr> ::= <conj>  <disj>
```

```
<disj> ::= '' |
           '||' <conj>
```

```
<conj> ::= <comp> <conjprime>
```

```
<conjprime> ::=  ''          |
                 '&&' <comp>
```

```
<comp> ::= <relational> <comprime>
```

```
<relational> ::= <sum> <relationalprime>
```

```
<sum> ::= <neg> <sumprime>
```

```
<sumprime> ::= ‘’                          |
               <add-op> <neg> <sumprime>
```

```
<neg> ::= <mul>          |
          ‘!’ <mul>
```

```
<mul> ::= <final-term> <mulprime>
```

```
<mulprime> ::= ‘’                                |
                 <mul-op> <final-term> <mulprime>
```

```
<relationalprime> ::= ‘’                       |
                      <relational-op> <sum>
```

```
<comprime> ::= ‘’ |
               <equality-op> <relational>
```

```
<final-term> ::= id <final-termprime> |
                 <literal>            |
                 ‘(’ <expr> ‘)’
```

```
<final-termprime> ::= ‘’          |
                      <variable>  |
                      <subprogcall>
```

```
<add-op> ::= ‘+’ |
             ‘-’
```

```
<mul-op> ::= ‘*’ |
             ‘/’ |
             ‘%’
```

```
<equality-op> ::= ‘==’ |
                  ‘!=’
```

```
<relational-op> ::= ‘<’ |
                    ‘<=’ |
                    ‘>’  |
                    ‘>=’
```

```
<literal> ::= intliteral         |
              realiteral         |
              charliteral        |
              stringliteral      |
              <subrangeliteral>
```

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

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

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

```
<subprograms> ::= ''                                |
                  <procedure> <subprogramsprime>    |
                  <function>  <subprogramsprime>
```

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

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

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

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

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

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

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

```
<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 (numpericop)

```
+  -  *  /  %
```

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

```
##
# PIE Merge Sort
# Pascal version: http://rextester.com/GHRH16649.
##
program merge_sort;

const
    FIRST = 0;
    LAST = 9;

type
    TRange = FIRST..LAST;
    TVector = array [TRange] of int;

var
    TVector vector;
    TRange index;

proc read_arr(TVector vec);
    var
        int i;
    begin
        writeln("Please give 10 integers as input for the array");
        for i := FIRST to LAST step 1 do
            read(vec[i])
    end;

proc merge(TVector helper; ref TVector vec; int first, last, center);
    var
        int i, j, k;
    begin
        i := first;
        j := center + 1;
        k := first;

        loop
            begin
                exitwhen (i > center) || (j > last);
                if helper[i] < helper[j]
                    begin
                       vec[k] := helper[i];
                       i := i + 1
                    end
                else
                    begin
                       vec[k] := helper[j];
                       j := j + 1
                    end;
```

```
                k := k + 1
            end;

        for j := i to center step 1 do
            begin
              vec[k] := helper[j];
              k := k + 1
            end
    end;

proc split_merge(TVector vec; int first, last);
    var
        int center;
    begin
        if last > first
            begin
                center := (first + last) / 2;
                split_merge(vec, first, center);
                split_merge(vec, center + 1, last);
                merge(vec, vec, first, last, center)
            end
    end;

begin
    read_arr(vector);
    split_merge(vector, FIRST, LAST);
    writeln('Sorted vector: ');
    for index := FIRST to LAST step 1 do
        begin
           write(vector[index], ' ')
        end
end.
```

6.2 Quicksort

```
##
# PIE Quick Sort
# Pascal version: http://sandbox.mc.edu/~bennet/cs404/doc/qsort_pas.html.
##
program quick_sort;

const
    FIRST = 0;
    LAST = 9;

type
    TRange = FIRST..LAST;
    TVector = vecay [TRange] of int;

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
```

```
                            end;
                        if left < right
                            swap(vec[left], vec[right]);
                    end;
                # Put the pivot between the halves.
                swap(vec[start], vec[right]);
                return right
            end
        begin
            if start < stop
                begin
                    splitpt := Split(start, stop);
                    quick_sort_recur(start, splitpt - 1);
                    quick_sort_recur(splitpt + 1, stop);
                end
        end
    begin
        quick_sort_recur(FIRST, LAST)
    end

begin
    read_vec(vector);
    quick_sort(vector);
    for i := 1 to size step 1 do
        writeln(vector[i])
end.
```

6.3 Fatorial

```
##
# PIE Factorial
# Pascal version: http://rextester.com/UXP1971.
##

program factorial;

var
    int num, sum, i;

begin
    writeln("Please Input an Integer");
    readln(num);
    sum := 1;
    for i := 1 to num step 1 do
        sum := sum * i;
    write('Result: ');
    write(sum)
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.