Universidade do Minho
Escola de Ciências

Universidade do Minho

Licenciatura em Ciências da Computação

PLC - Trabalho Prático 2

Grupo nº14

José Bernardo Moniz Fernandes
(A102497)

Pedro Augusto Ennes de Martino Camargo
(A102504)

4 de janeiro de 2025

Conteúdo

Capítulo 1

Introdução

No âmbito da unidade curricular de Processamento de Linguagens e Compiladores foi-nos proposto pelo docente Pedro Rangel Henriques o desenvolvimento de uma Linguagem de Programação Imperativa, ao nosso gosto, e de um compilador para reconhecer programas escritos nessa linguagem, gerando o respetivo código Assembly da Máquina Virtual VM.

Primeiramente, começámos por encontrar um nome para a nossa linguagem e acabou por nos surgir a ideia de colocar o nome "PedroMonizScript"(PMS), inspirámo-nos na linguagem de programação TypeScript, que proporciona segurança no seu desenvolvimento.

Neste documento, está apresentada a gramática e a sintaxe da nossa linguagem, bem como alguns testes com código escrito na nossa linguagem e o respetivo código Assembly gerado.

Capítulo 2

Enunciado

Pretende-se que comece por definir uma linguagem de programação imperativa simples, a seu gosto.
Apenas deve ter em consideração que essa linguagem terá de permitir:

- *declarar* variáveis atómicas do tipo *inteiro*, com os quais se podem realizar as habituais operações aritméticas, relacionais e lógicas.
- *efetuar* instruções algorítmicas básicas como a *atribuição do valor de expressões numéricas a variáveis.*
- *ler* do *standard input* e *escrever* no *standard output.*
- *efetuar* instruções *seleção* para controlo do fluxo de execução.
- *efetuar* instruções de repetição (*cíclicas*) para controlo do fluxo de execução, permitindo o seu aninhamento.
  Note que deve implementar pelo menos o ciclo **while-do**, **repeat-until** ou **for-do**.

Adicionalmente deve ainda suportar, à sua escolha, uma das duas funcionalidades seguintes:

- *declarar e manusear* variáveis estruturadas do tipo *array (a 1 ou 2 dimensões) de inteiros*, em relação aos quais é apenas permitida a operação de indexação (índice inteiro).
- *definir e invocar subprogramas* sem parâmetros mas que possam retornar um resultado do tipo inteiro.

Capítulo 3

Concepção da Solução

Neste capítulo vamos apresentar:

- A sintaxe da linguagem PMS
- Os símbolos
- O desenho da gramática independente de contexto

3.1 Sintaxe da Linguagem PMS

A sintaxe da linguagem é a seguinte:

3.1.1 Declaração de variáveis

```
let a: INT = 1000;
let b: STR = "Ola";
let c: FLOAT = 10.9;
let d: INT;
let f: FLOAT;
let arrayInt: Array<INT> = [0, 1, 2, 3];
let arrayFloat: Array<FLOAT> = [10.9, 199.212, 322.12];
let arrayString: Array<STR> = ["Olá!", "Tudo Bem?"];
const inputtext: INT = console.input("Por favor dê um valor para esta variável: ");
```

3.1.2 Operadores de comparação

```
x > y
x < y
x >= y
x <= y
x == y
x != y
```

3.1.3 Operações numéricas

```
x + y
x - y
x * y
x / y
x % y
x ++
y --
```

3.1.4 Operadores lógicos

```
x && y
x || y
not x
```

3.1.5 Instruções condicionais

3.1.6 If

```
if (Cond) {
    ...
}
```

3.1.7 If-Else

```
if (Cond) {
    ...
} else {
    ...
}
```

3.1.8 Ciclo while

```
while (Cond) {
    ...
}
```

3.1.9 Ciclo do-while

```
do {
    ...
} while (Cond);
```

3.1.10 Função

```
const ID = () => {
    ...
}
```

3.1.11 Input/Output

```
const inputtext: INT = console.input("Por favor dê um valor para esta variável: ");
console.output(inputtext);
```

3.2 Símbolos

Os simbolos da linguagem são os seguintes:

```
'INTVALUE',
'FLOATVALUE',
'STRINGVALUE',
'INC',
'DEC',
'ID',
'GEQUAL',
'LEQUAL',
'EQUAL',
'DIFF',
'INT',
'FLOAT',
'STR',
'CONST',
'LET',
'WHILE',
'ARRAY',
'IF',
'ELSE',
'DO',
'WHILE',
'PRINT',
'INPUT',
'SEMICOLON',
'OR',
'AND',
'NOT',
'CALL',
'RETURN',
':',
```

```
'=',
'[',
']',
'<',
'>',
',',
'(',
')',
'<',
'>',
',',
'+',
'-',
'*',
'/',
'%',
'{',
'}'
```

3.3 Desenho da GIC

A nossa linguagem é gerada pela seguinte grámatica independente de contexto:

```
ProgramInit : Declarations Instructions
            | Declarations
            | Instructions

Declarations  : Declarations IntDeclaration
              | Declarations IntDeclarationInput
              | Declarations StringDeclaration
              | Declarations StringDeclarationInput
              | Declarations FloatDeclaration
              | Declarations FloatDeclarationInput
              | Declarations ArrayDeclaration
              | Declarations FunctionDeclaration
              | Empty

Instructions : Instructions Instruction
             | Instruction

MutationType  : CONST
              | LET

IntDeclaration : MutationType ID ':' INT '=' INTVALUE SEMICOLON
               | MutationType ID ':' INT SEMICOLON

IntDeclarationInput : MutationType ID ':' INT '=' Input SEMICOLON
```

```
25
26    StringDeclaration : MutationType ID ':' STR '=' STRINGVALUE SEMICOLON
27                      | MutationType ID ':' STR SEMICOLON
28
29    StringDeclarationInput : MutationType ID ':' STR '=' Input SEMICOLON
30
31    FloatDeclaration : MutationType ID ':' FLOAT '=' FLOATVALUE SEMICOLON
32                     | MutationType ID ':' FLOAT SEMICOLON
33
34    FloatDeclarationInput : MutationType ID ':' FLOAT '=' Input SEMICOLON
35
36    ArrayDeclaration : MutationType ID ':' ARRAY '<' INT '>' '=' '['
      ArrayIntDeclaration ']' SEMICOLON
37                     | MutationType ID ':' ARRAY '<' FLOAT '>' '=' '['
      ArrayFloatDeclaration ']' SEMICOLON
38                     | MutationType ID ':' ARRAY '<' STR '>' '=' '['
      ArrayStringDeclaration ']' SEMICOLON
39
40    ArrayIntDeclaration : ArrayIntDeclaration ',' INTVALUE
41                        | INTVALUE
42                        | Empty
43
44    ArrayFloatDeclaration : ArrayFloatDeclaration ',' FLOATVALUE
45                          | FLOATVALUE
46                          | Empty
47
48    ArrayStringDeclaration : ArrayStringDeclaration ',' STRINGVALUE
49                           | STRINGVALUE
50                           | Empty
51
52    FunctionDeclaration : CONST ID '=' '(' ')' '=' '>' '{' Instructions '}'
53
54    Instruction : Attributions
55                | Output
56                | Call
57                | Return
58                | If
59                | Loop
60
61    Attributions : Attributions NormalAttribution SEMICOLON
62                 | Attributions IncDecAttribution SEMICOLON
63                 | Attributions Expression SEMICOLON
64                 | Empty
65
66    NormalAttribution : ID '=' INTVALUE
67                      | ID '=' STRINGVALUE
68                      | ID '=' FLOATVALUE
69
70    IncDecAttribution : ID DEC
71                      | ID INC
72
73    Expression : ID '=' Expr
74
75    Expr : IncDecAttribution
```

```
        | ID
        | INTVALUE
        | FLOATVALUE
        | Expr '+' Expr
        | Expr '-' Expr
        | Expr '*' Expr
        | Expr '/' Expr
        | Expr '%' Expr

    Cond : Expr '<' Expr
         | Expr '>' Expr
         | Expr GEQUAL Expr
         | Expr LEQUAL Expr
         | Expr EQUAL Expr
         | Expr DIFF Expr
         | Cond OR Cond
         | Cond AND Cond
         | NOT '(' Cond ')'

    If : IF '(' Cond ')' '{' Instructions '}'
       | IF '(' Cond ')' '{' Instructions '}' ELSE '{' Instructions '}'

    Loop : While
         | DoWhile

    While : WHILE '(' Cond ')' '{' Instructions '}'

    DoWhile : DO '{' Instructions '}' WHILE '(' Cond ')' SEMICOLON

    Call : CALL SEMICOLON
         | RETURN CALL SEMICOLON

    Return : RETURN Expr SEMICOLON
           | RETURN Call

    Input : INPUT '(' STRINGVALUE ')'

    Output : PRINT '(' ID ')' SEMICOLON
           | PRINT '(' STRINGVALUE ')' SEMICOLON

    Empty :
```

Capítulo 4

Exemplos de funcionamento

4.0.1 While

```
let a: INT = 3;

while (a > 0) {
    a--;
    console.output(a);
    console.output("\n");
}

```

Código Assembly gerado:

```
pushi 3
start
label0c: NOP
PUSHG 0
PUSHI 0
SUP
JZ label0f
PUSHG 0
PUSHI 1
SUB
STOREG 0
PUSHG 0
WRITEI
PUSHS "\n"
WRITES
JUMP label0c
label0f: NOP
stop

```

4.0.2 If-Else com operadores de comparação e lógicos

```
let a: INT = console.input("Coloque um valor inteiro para a: \n");
let b: INT = console.input("Coloque um valor inteiro para b: \n");
let c: INT = console.input("Coloque um valor inteiro para c: \n");

if (not(a == b && b == c)) {
    console.output("triangulo nao e equilatero\n");
} else {
    console.output("triangulo e equilatero\n");
}

if (not(a == b || b == c || a == c)) {
    console.output("triangulo nao e isosceles\n");
} else {
    console.output("triangulo e isosceles\n");
}

if (a != b && b != c && a != c) {
    console.output("triangulo e escaleno\n");
} else {
    console.output("triangulo nao e escaleno\n");
}

```

Código Assembly gerado:

```
PUSHS "Coloque um valor inteiro para a: \n"
WRITES
READ
ATOI
STOREG 0
PUSHS "Coloque um valor inteiro para b: \n"
WRITES
READ
ATOI
STOREG 1
PUSHS "Coloque um valor inteiro para c: \n"
WRITES
READ
ATOI
STOREG 2
start
PUSHG 0
PUSHG 1
EQUAL
PUSHG 1
PUSHG 2
EQUAL
MUL
NOT
JZ label0
PUSHS "triangulo nao e equilatero\n"
WRITES
JUMP label0f
```

```
label0: NOP
PUSHS "triangulo e equilatero\n"
WRITES
label0f: NOP
PUSHG 0
PUSHG 1
EQUAL
PUSHG 1
PUSHG 2
EQUAL
PUSHG 0
PUSHG 2
EQUAL
ADD
ADD
NOT
JZ label1
PUSHS "triangulo nao e isosceles\n"
WRITES
JUMP label1f
label1: NOP
PUSHS "triangulo e isosceles\n"
WRITES
label1f: NOP
PUSHG 0
PUSHG 1
EQUAL
NOT
PUSHG 1
PUSHG 2
EQUAL
NOT
PUSHG 0
PUSHG 2
EQUAL
NOT
MUL
MUL
JZ label2
PUSHS "triangulo e escaleno\n"
WRITES
JUMP label2f
label2: NOP
PUSHS "triangulo nao e escaleno\n"
WRITES
label2f: NOP
stop

```

4.0.3 Do-While

```
let N: INT = console.input("Digite um numero: ");
let soma: INT = 0;
```

```
let i: INT = 1;

do {
    soma = soma + i;
    i++;
} while (i <= N);

console.output("\n");
console.output("A soma dos primeiros ");
console.output(N);
console.output(" numeros e: ");
console.output(soma);

```

Código Assembly gerado:

```
PUSHS "Digite um numero: "
WRITES
READ
ATOI
STOREG 0
pushi 0
pushi 1
start
label0:
PUSHG 1
PUSHG 2
ADD
STOREG 1
PUSHG 2
PUSHI 1
ADD
STOREG 2
PUSHG 2
PUSHG 0
INFEQ
NOT
JZ label0
PUSHS "\n"
WRITES
PUSHS "A soma dos primeiros "
WRITES
PUSHG 0
WRITEI
PUSHS " numeros e: "
WRITES
PUSHG 1
WRITEI
stop

```

4.0.4 Operações numéricas

```
let a1: INT = 5;
let a2: FLOAT = 9.0;
let a3: INT = 2;
let a4: INT = 20;
let a5: INT = 25;
let b: INT = 5;
let c: INT = 7;

a1 = a1 + b;

a2 = a2 - 5;

a3 = a3 * b--;

a4 = a4 / b;

a5 = a5 % b;

c = a1 + a2 + 6 - 2;

```

Código Assembly gerado:

```
pushi 5
pushf 9.0
pushi 2
pushi 20
pushi 25
pushi 5
pushi 7
start
PUSHG 0
PUSHG 5
ADD
STOREG 0
PUSHG 1
PUSHI 5
SUB
STOREG 1
PUSHG 2
PUSHG 5
PUSHI 1
SUB
STOREG 5
PUSHG 5
MUL
STOREG 2
PUSHG 3
PUSHG 5
DIV
STOREG 3
PUSHG 4
PUSHG 5
```

```
MOD
STOREG 4
PUSHG 0
PUSHG 1
PUSHI 6
PUSHI 2
SUB
ADD
ADD
STOREG 6
stop

```

4.0.5 Função

```
let a: INT = 10;

const test = () => {
    console.output("Resposta de a + 10: ");
    a = a + 10;
    console.output(a);

    return a;
}

test();

```

Código Assembly gerado:

```
pushi 10
function0:
PUSHS "Resposta de a + 10: "
WRITES
PUSHG 0
PUSHI 10
ADD
STOREG 0
PUSHG 0
WRITEI
PUSHG 0
RETURN

start
PUSHA function0
CALL
stop

```

Capítulo 5

Conclusão

A realização deste trabalho prático, proporcionou-nos atingir os objetivos inicialmente propostos, foi verdadeiramente um trabalho desafiador e produtivo.
Desenvolver a nossa própria linguagem de programação foi um processo que nos exigiu criatividade e decisões cuidadosas.

Além disso, chegar ao final do projeto e perceber que conseguimos projetar e implementar a base de uma linguagem é extremamente recompensador.

Este trabalho proporcionou-nos uma sólida compreensão sobre o funcionamento dos módulos Lexer e Yacc, especialmente no reconhecimento de tokens e na implementação da gramática. Além de que, a geração de código Assembly, para Máquina Virtual, revelou-se uma etapa interessante, permitindo-nos aprofundar o nosso entendimento sobre as instruções e a lógica relacionada a esta linguagem.

Para concluir, entendemos que a maior parte dos objetivos foram alcançados, consideramos, ainda que, este trabalho foi desafiante e preparou-nos para enfrentar desafios futuros.