Processamento de Linguagens e Compiladores

Trabalho Prático 2

Carlos Beiramar (a84628) | Jorge Silva (a80931) | José Mendes (a81809)

(16 de janeiro de 2022)

Resumo

O presente trabalho surge com o intuito de explorar a definição de uma linguagem de programação imperativa e a elaboração do seu respetivo compilador. Para o desenvolvimento deste trabalho foi utilizado um *parser YACC* para reconhecer e converter a sintaxe da linguagem imperativa em código **VM**, com o auxílio de uma analisador léxico *Lexer* para identificar as palavras reservadas da mesma.

Conteúdo

1 Introdução **2**

2 Enunciado **3**

3 Análise e Especificação **5**
3.1 Descrição informal do problema 5
3.2 Levantamento de requisitos 5
3.2.1 Requisitos base 5
3.2.2 Requisitos adicionais 5

4 Conceção/desenho da Resolução **6**
4.1 **Lexer** 6
4.2 **Yacc** 9
4.2.1 Variáveis 9
4.2.2 Funções auxiliares 9
4.2.3 Gramática Independente de Contexto 10
4.2.4 Conversão para a linguagem **VM** 12

5 Resultados obtidos **13**
5.1 Problema 1 13
5.2 Problema 2 14
5.3 Problema 3 16
5.4 Problema 4 18
5.5 Problema 5 20
5.6 Problema 6 22

6 Anexos **24**
6.1 Menu 24
6.2 Parser 26
6.3 Analisador Léxico 30

7 Conclusão **42**

1 Introdução

O seguinte projeto encontra-se inserido na unidade curricular de Processamento de Linguagens e compiladores, presente no 3º ano letivo da licenciatura em Ciências da Computação da Universidade do Minho. Este constitui o segundo trabalho prático presente na componente prática da avaliação da unidade curricular, cujo os objetivos são:

- Desenvolver processadores de linguagens segundo o método da tradução dirigida pela sintaxe, a partir de uma gramática tradutora.
- Desenvolver um compilador gerando código para uma máquina de stack virtual.
- Utilizar geradores de compiladores baseados em gramáticas tradutoras, concretamente o *Yacc*, versão *PLY* do **Python**, completado pelo gerador de analisadores léxicos *Lex*, também versão *PLY* do **Python**.

O tema a abordar será o desenvolvimento de um compilador capaz de converter ficheiros de uma linguagem de programação criada pelos alunos, num ficheiro com instruções de **pseudo-código**, *Assembly* da máquina virtual. Este relatório tem como objetivo a descrição detalhada dos requisitos do projeto, assim como detalhar todo o processo de realização deste trabalho. Durante a realização deste trabalho passamos por vários desafios. O mais complicado foi idealizar a estrutura da nossa linguagem. O compilador desenvolvido revela-se eficaz na tarefa proposta, contendo as funcionalidades necessárias para que a nossa linguagem seja interpretada.

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 input*.
- *efetuar* instruções *condicionais* para controlo de fluxo de execução.
- *efetuar* instruções *cíclicas* para controlo de fluxo de execução, permitindo o seu aninhamento.

  Note que deve implementar pelo menos o ciclo **while-do**, **repeat-until**, **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.

Como é da praxe neste tipo de linguagens, as variáveis deverão ser declaradas no início do programa e não pode haver re-declarações, nem utilizações sem declaração prévia. Se nada for explicitado, o valor da variável após a declaração é 0(zero).
Desenvolva, então, um compilador para essa linguagem com base na **GIC** criada acima e com recurso aos módulos **Yacc/Lex** do **PLY/Python**.
O compilador deve gerar **pseudo-código**, Assembly da Máquina Virtual VM.
Muito Importante:
Para a entrega do TP deve preparar um conjunto de testes (programas-fonte escritos na sua linguagem) e mostrar o código **Assembly** gerado bem como o programa a correr na máquina virtual **VM**.

Esse conjunto terá de conter, no mínimo, os 4 primeiros exemplos abaixo e um dos 2 últimos conforme a sua escolha acima:

- Ler 4 números e dizer se podem ser os lados de um quadrado.
- Ler um inteiro N, depois ler N números e escrever o menor deles.
- Ler N (constante do programa) números e calcular e imprimir o seu produtório.
- Contar e imprimir os números impares de um sequência de números naturais.
- Ler e armazenar N números num array; imprimir os valores por ordem inversa.
- invocar e usar num programa seu uma função 'potencia()', que começa por ler do input a base B e o expoente E e retorna o valor B^E.

3 Análise e Especificação

3.1 Descrição informal do problema

Neste projeto deverá ser implementado um compilador de ficheiros gravados em formato *txt*, formato esse escolhido pelo grupo para a realização do trabalho, para ficheiros em formato da *Máquina Virtual VM*. Este compilador será desenvolvido utilizando o módulo *PLY* e o módulo *Yacc* do **Python**. Estes módulos permitirão ao programa reconhecer, por exemplo, a implementação de estruturas cíclicas, arrays unidimensionais, condições com expressões condicionais.

3.2 Levantamento de requisitos

Do enunciado referido anterior é possível levantar um conjunto de requisitos que o projeto deverá cumprir.

3.2.1 Requisitos base

Os requisitos base são:

- *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 input*.
- *efetuar* instruções *condicionais* para controlo de fluxo de execução.
- *efetuar* instruções *cíclicas* para controlo de fluxo de execução, permitindo o seu aninhamento.

  Note que deve implementar pelo menos o ciclo **while-do**, **repeat-until**, **for-do**.

3.2.2 Requisitos adicionais

Os requisitos adicionais são:

- *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.

4 Conceção/desenho da Resolução

4.1 Lexer

Na criação do **Lexer** foi optado pela criação de uma lista com todas as *tokens* utilizadas para o desenvolvimento do compilador, *tokens* essas que estão representadas na seguinte imagem:

```
tokens = [

    # (Identifier, int)
    'ID','NINT',

    #Operators(+,-,*,/,%, <, ==, &&, or, <, <=, >, >=, !=)
    'PLUS','MINUS','MULT','DIV','MOD', 'EQEQ',
    'AND', 'OR', 'LT', 'LE', 'GT', 'GE', 'NEQ',

    #Assign(=, ++, --, +=, -=, *=, /=, %=)
    'ASSIGN', 'PLUSPLUS', 'MINUSMINUS', 'PLUSEQ','MINUSEQ', 'MULTEQ','DIVEQ','MODEQ',

    #Words
    'FUNCTIONS','DECLARATIONS','BEGIN','WRITE','READ','IF'
    ,'THEN','ELSE','END','PHRASE', 'IntWord','WHILE','UNTIL','ArrayInt','FOR',
    'DEF','FUNC_NAME','RETURN',

    # Delimeters ( ) [ ] { } , ;
    'LROUND','RROUND','LSQUARE','RSQUARE',
    'LCURLY','RCURLY', 'VIR','PONTeVIR', 'DOLLAR'
]
```

Inicialmente foram criados os identificadores:

- **ID** - permite atribuir qualquer nome a uma variável.
- **NINT** - permite representar qualquer número inteiro.

De seguida foram criados os operadores:

- **PLUS** - soma +.
- **MINUS** subtração −.
- **MULT** - multiplicação ∗.
- **DIV** - divisão /.
- **MOD** - módulo %.
- **EQEQ**- comparação ==.
- **AND** - conjunção &&.
- **OR** - disjunção ||.
- **LT** - símbolo menor <.
- **LE** - símbolo menor ou igual <=.
- **GT** - símbolo maior >.
- **GE** - símbolo maior ou igual >=.
- **NEQ** - símbolo diferente ! =.

Posteriormente, foram definidas tokens para representar algumas palavras definidas na linguagem, foi utilizado o *case sensitive* para identificar as palavras:

- **FUNCTIONS** - representa a palavra "FUNCTIONS".
- **DECLARATIONS** - representa a palavra "DECLARATIONS".
- **BEGIN** - representa a palavra "BEGIN".
- **WRITE** - representa a palavra "WRITE".
- **READ** - representa a palavra "READ".
- **IF** - representa a palavra "IF".
- **THEN** - representa a palavra "THEN".
- **ELSE** - representa a palavra "ELSE".
- **END** - representa a palavra "END".
- **PHRASE** - representa um frase.
- **IntWord** - representa a palavra "int".
- **WHILE** - representa a palavra "WHILE".
- **UNTIL** - representa a palavra "UNTIL".
- **ArrayInt** - representa a palavra "ArrayInt".
- **FOR** - representa a palavra "FOR".
- **DEF** - representa a palavra "DEF".
- **FUNC_NAME** - representa o nome de uma função.

Por fim, definiram se alguns critérios para finalizar a sintaxe da nossa linguagem.

- **LROUND** - representa "(".
- **RROUND** - representa ")".
- **LSQUARE** - representa "[".
- **RSQUARE** - representa "]".
- **LCURLY** - representa "{".
- **RCURLY** - representa "}".
- **VIR** - representa ",".
- **PONTeVIR** - representa ";".
- **DOLLAR** - representa "$".

4.2 Yacc

4.2.1 Variáveis

No ficheiro **Yacc** é onde se encontra implementada a gramática criada para a elaboração deste projeto. Inicialmente são implementadas algumas variáveis:

```
variables = {}
functions = []
countWHILE = 0
countFor = 0
countIF = 0
```

Cada variável terá um objetivo diferente:

1. **variables** - será um dicionário onde a *key* é a variáveis e o *value* é valor atribuído a essa variável.
2. **functions** - será uma lista com o nomes das funções implementadas nos ficheiros *txt*.
3. **countWHILE** - é um contador para os *while* com intuito de ajudar a criar as *labels*.
4. **countFor** - é um contador para os *for* com intuito de ajudar a criar as *labels*.
5. **countIF** - é um contador para os *while's* com intuito de ajudar a criar as *labels*.

4.2.2 Funções auxiliares

Foi implementada uma função que permite saber qual é o índice de cada uma das variáveis que se encontram na variável *variables*.

```
def get_indexVar(id):
    count  = 0
    for key in variables.keys():
        if key == id:
            return count
        else:
            count+=1
```

4.2.3 Gramática Independente de Contexto

Foi então elaborada uma gramática que reconhecesse a linguagem criada pelo grupo assim, esta linguagem utilizada várias palavras reservadas que foram definidas no *Lexer* como *tokens*.

```
LstPrograms : Program
            | LstPrograms Program

Program : DECLARATIONS LCURLY LstDecl RCURLY Fs BEGIN LstInst END

# Declarations

LstDecl : Decl
        | Decl LstDecl

Decl : IntWord Variables PONTeVIR
     | ArrayInt Variables PONTeVIR

Variables : Var VIR Variables
          | Var
          | ID
          | ID LSQUARE NINT RSQUARE

# Funtions

Fs  :
    | FUNCTIONS DOLLAR LstFunct DOLLAR

LstFunct : Func
         | LstFunct Func

Func : DEF FUNC_NAME LCURLY LstInst RETURN ID PONTeVIR RCURLY

# Instructions

LstInst : Instruction
        | LstInst Instruction

Instruction : Atrib
            | Function
            | ifStatement
            | Loop

# LOOP
```

```
Loop : WHILE LROUND Condition RROUND LCURLY LstInst RCURLY
     | FOR LROUND Atrib Atrib Condition RROUND LCURLY LstInst RCURLY

# if statement

ifStatement : IF LROUND Condition RROUND THEN LCURLY LstInst RCURLY
            | IF LROUND Condition RROUND THEN LCURLY LstInst RCURLY ELSE
            LCURLY LstInst RCURLY

# atrib

Atrib : ID ASSIGN Expr PONTeVIR
      | ID PLUSEQ Expr PONTeVIR
      | ID MINUSEQ Expr PONTeVIR
      | ID MULTEQ Expr PONTeVIR
      | ID DIVEQ Expr PONTeVIR
      | ID MODEQ Expr PONTeVIR
      | ID PLUSPLUS PONTeVIR
      | ID MINUSMINUS PONTeVIR
      | ID LSQUARE Expr RSQUARE ASSIGN ExprR PONTeVIR

# funções pre definidas

Function : WRITE LROUND PHRASE RROUND PONTeVIR
         | WRITE LROUND ExprR RROUND PONTeVIR

# conditions

Condition : ExprR
          | ExprR AND Condition
          | ExprR UNTIL ExprR
          | ExprR OR Condition

# expressoes

ExprR : Expr
      | Expr EQEQ Expr
      | Expr NEQ Expr
      | Expr LT Expr
      | Expr LE Expr
      | Expr GT Expr
      | Expr GE Expr


Expr : Term
     | Expr PLUS Term
```

```
     | Expr MINUS Term

Term : Factor
     | Term MULT Factor
     | Term DIV Factor
     | Term MOD Factor

Factor : ID
       | NINT
       | LROUND MINUS NINT RROUND
       | READ LROUND RROUND
       | FUNC_NAME
       | ID LSQUARE NINT RSQUARE
       | ID LSQUARE ID RSQUARE
```

4.2.4 Conversão para a linguagem VM

A transição de um programa, escrito com a sintaxe criada pelo grupo, para a linguagem da **VM** é feita através da concatenação dos comandos **VM** resultantes da análise de cada situação específica para, por fim, ser escrito o texto completo, de uma só vez, no ficheiro.

5 Resultados obtidos

5.1 Problema 1

No problema 1 foi pedido que se implementasse um programa que verifica se 4 números podem ser os lados de um quadrado. Assim o código implementado no ficheiro *txt* foi o seguinte:

```
DECLARATIONS {
    int a,b,c,d;
}
BEGIN
    write("Write the square values");
    a = read();
    b = read();
    c = read();
    d = read();
    if ( a == b && a == c && a == d)
    then {write("SQUARE");}
    else {write("NOT SQUARE");}
END
```

Assim, o compilador implementado irá gerar o seguinte código *Assembly*.

```
PUSHI 0
PUSHI 0
PUSHI 0
PUSHI 0
START
JUMP main
main: nop
PUSHS "Write the square values"
WRITES
READ
ATOI
STOREG 0
READ
ATOI
STOREG 1
READ
ATOI
STOREG 2
READ
ATOI
STOREG 3
PUSHG 0
PUSHG 1
EQUAL
PUSHG 0
PUSHG 2
EQUAL
PUSHG 0
PUSHG 3
EQUAL
MUL
MUL
JZ ELSE1
PUSHS "SQUARE"
WRITES
JUMP FIM1
ELSE1:
PUSHS "NOT SQUARE"
WRITES
FIM1:
STOP
```

5.2 Problema 2

No problema 2 foi pedido que se implementasse um programa que lê um inteiro N e, após isso ler N inteiros e imprimir o menos deles. O código implementado no ficheiro *txt* foi o seguinte:

```
DECLARATIONS{
int lido,res,i,j;
}
BEGIN
write("escreva o tamanho da sequencia");
i = read();

write("Insira um inteiro");
res = read();
j = 1;
while( j < i) {
    write("Insira um inteiro");
    lido = read();
    if (lido < res) then {
        res = lido;
    }
    j++;
}
write(res);
END
```

O compilador irá então gerar o seguinte código *Assembly*:

```
PUSHI 0
PUSHI 0
PUSHI 0
PUSHI 0
START
JUMP main
main: nop
PUSHS "escreva o tamanho da sequencia"
WRITES
READ
ATOI
STOREG 2
PUSHS "Insira um inteiro"
WRITES
READ
ATOI
STOREG 1
```

```
PUSHI 1
STOREG 3
WHILE1:
PUSHG 3
PUSHG 2
INF
JZ ENDWHILE1
PUSHS "Insira um inteiro"
WRITES
READ
ATOI
STOREG 0
PUSHG 0
PUSHG 1
INF
JZ FIM1
PUSHG 0
STOREG 1
JUMP FIM1
FIM1:
PUSHG 3
PUSHI 1
ADD
STOREG 3
JUMP WHILE1
ENDWHILE1:
PUSHG 1
WRITEI
STOP
```

5.3 Problema 3

No problema 3 é pedido que o programa faça a leitura de N números e que imprima o seu produtório. O seguinte código foi implementado no ficheiro *txt*:

```
DECLARATIONS{
int quant, aux, i, count;
}
BEGIN
i = 0;
count = 1;
write("Indique o numero de inteiros");
quant = read();

FOR ( i = 0; i++; i < quant) {
    write("Insira o digito:");
    aux = read();
    count = count * aux;
}

write("Produtorio = ");
write(count);
END
```

O código *Assembly* gerado pelo compilador foi o seguinte:

```
PUSHI 0
PUSHI 0
PUSHI 0
PUSHI 0
START
JUMP main
main: nop
PUSHI 0
STOREG 2
PUSHI 1
STOREG 3
PUSHS "Indique o numero de inteiros"
WRITES
READ
ATOI
STOREG 0
PUSHI 0
STOREG 2
FOR1:
PUSHG 2
PUSHG 0
INF
JZ ENDFOR1
PUSHS "Insira o digito:"
WRITES
READ
ATOI
STOREG 1
PUSHG 3
PUSHG 1
MUL
STOREG 3
PUSHG 2
PUSHI 1
ADD
STOREG 2
JUMP FOR1
ENDFOR1:
PUSHS "Produtorio = "
WRITES
PUSHG 3
WRITEI
STOP
```

5.4 Problema 4

No problema 4 é pedido que o programa conte e imprima os números ímpares de uma sequência de números naturais. O seguinte código foi implementado no ficheiro *txt*:

```
DECLARATIONS {
int i, quant, store, count;
}
BEGIN
i = 0;
count = 0;
write("Indique quantos numeros vai inserir");
quant = read();
while (i < quant){
    write("Insira um digito");
    store = read();
    if (store / 2 == 1) then {
        count = count + 1;
        write(store);
    }
    i = i + 1;
}
write(count);
END
```

O código *Assembly* gerado foi o seguinte:

```
PUSHI 0
PUSHI 0
PUSHI 0
PUSHI 0
START
JUMP main
main: nop
PUSHI 0
STOREG 0
PUSHI 0
STOREG 3
PUSHS "Indique quantos numeros vai inserir"
WRITES
READ
ATOI
STOREG 1
WHILE1:
PUSHG 0
PUSHG 1
INF
JZ ENDWHILE1
PUSHS "Insira um digito"
WRITES
READ
ATOI
STOREG 2
PUSHG 2
PUSHI 2
DIV
PUSHI 1
EQUAL
JZ FIM1
PUSHG 3
PUSHI 1
ADD
STOREG 3
PUSHG 2
WRITEI
JUMP FIM1
FIM1:
PUSHG 0
PUSHI 1
ADD
STOREG 0
JUMP WHILE1
ENDWHILE1:
PUSHG 3
WRITEI
STOP
```

5.5 Problema 5

No problema 5 é pedido para ler e armazenar N números num array e, após isso imprimir o array por ordem inversa. Assim, o código implementado no ficheiro *txt* foi:

```
DECLARATIONS{
    ArrayInt a[10];
    int i, j, store;
}
BEGIN
    i = 0;

    WHILE(i < 10) {
        write("Insira um numero");
        store = read();
        a[i] = store;
        i = i + 1;
    }

    i = i - 1;
    WHILE( i >= 0 ){
        write(a[i]);
        i = i - 1;
    }
END
```

O compilador irá gerar o seguinte código *Assembly*:

```
PUSHN 10
PUSHI 0
PUSHI 0
PUSHI 0
START
JUMP main
main: nop
PUSHI 0
STOREG 1
WHILE1:
PUSHG 1
PUSHI 10
INF
JZ ENDWHILE1
PUSHS "Insira um numero"
WRITES
READ
ATOI
STOREG 3
PUSHGP
PUSHI 0
PUSHG 1
ADD
PUSHG 3
STOREN
PUSHG 1
PUSHI 1
ADD
STOREG 1
JUMP WHILE1
ENDWHILE1:
PUSHG 1
PUSHI 1
SUB
STOREG 1
WHILE2:
PUSHG 1
PUSHI 0
SUPEQ
JZ ENDWHILE2
PUSHG 10
WRITEI
PUSHG 1
PUSHI 1
SUB
STOREG 1
JUMP WHILE2
ENDWHILE2:
STOP
```

5.6 Problema 6

No problema 6 é pedido que seja possível ao compilador invocar uma função *potencia()* que lê do input a base e o expoente e que calcula a potência. Assim, o código implementado no ficheiro *txt* foi o seguinte:

```
DECLARATIONS{
    int res, base, expoente, i;
}

FUNCTIONS $
    def potencia(){
        base = 0;
        res = 1;
        write("Insira a base");
        base = read();
        write("Insira o expoente");
        expoente = read();

        for(i = 0; i++; i < expoente){
            res *= base;
        }

        return res;
    }
$

BEGIN
res = potencia();
END
```

O compilador gerou o seguinte código *Assembly*.

```
PUSHI 0
PUSHI 0
PUSHI 0
PUSHI 0
START
JUMP main
potencia: nop
PUSHI 0
STOREG 1
PUSHI 1
STOREG 0
PUSHS "Insira a base"
WRITES
READ
ATOI
STOREG 1
PUSHS "Insira o expoente"
WRITES
READ
ATOI
STOREG 2
PUSHI 0
STOREG 3
FOR1:
PUSHG 3
PUSHG 2
INF
JZ ENDFOR1
PUSHG 0
PUSHG 1
MUL
STOREG 0
PUSHG 3
PUSHI 1
ADD
STOREG 3
JUMP FOR1
ENDFOR1:
RETURN
ret: nop
PUSHG 0
JUMP endret
main: nop
PUSHA potencia
CALL
nop
JUMP ret
endret: nop
STOREG 0
STOP
```

6 Anexos

6.1 Menu

```
from os import close
import tp1_yacc as yacc

def menu():
    print(" _____MENU", end= "")
    print("_____")
    print("|                                                    ", end = "")
    print("                                                  |")
    print("|\t[1] - Ler 4 números e verificar se podem ", end = "")
    print("ser uma quadrado ou não.                          |")
    print("|\t[2] - Ler um inteiro N e depois ler ", end = "")
    print("N inteiros e imprimir o menor.                        |")
    print("|\t[3] - Ler N numeros e imprimir o seu produtório.", end = "")
    print("                                                |")
    print("|\t[4] - Contar e imprimir os numeros impares ", end = "")
    print("de uma sequencia de numeros naturais.", end = "")
    print("             |")
    print("|\t[5] - Ler e armazenar N numeros num array.", end = "")
    print(" Imprimir por ordem inversa.", end = "")
    print("                        |")
    print("|\t[6] - Invocar uma funcao potencia() que le do input", end = "")
    print(" a base e o expoente e calcula a potencia.|")
    print("|\t[0] - Sair.                                          ", end = "")
    print("                                        |")
    print("|                                                      ", end = "")
    print("                                                  |")
    print("|_____", end = "")
    print("_____|")
    print("")

def init():

    menu()
    opcao = input("Escolha uma opcao:")
    while(opcao != 0):

        if opcao == '1':

            yacc.parser.file = open("vmFiles/ex1.vm", "w")
            with open('funcFiles/ex1.txt', 'r') as file:
                program = file.read().rstrip()
            yacc.parser.parse(program)
```

```
            opcao = 0

        elif opcao == '2':

            yacc.parser.file = open("vmFiles/ex2.vm", "w")
            with open('funcFiles/ex2.txt', 'r') as file:
                program = file.read().rstrip()
            yacc.parser.parse(program)
            opcao = 0

        elif opcao == '3':

            yacc.parser.file = open("vmFiles/ex3.vm", "w")
            with open('funcFiles/ex3.txt', 'r') as file:
                program = file.read().rstrip()
            yacc.parser.parse(program)
            opcao = 0

        elif opcao == '4':
            yacc.parser.file = open("vmFiles/ex4.vm", "w")
            with open('funcFiles/ex4.txt', 'r') as file:
                program = file.read().rstrip()
            yacc.parser.parse(program)
            opcao = 0

        elif opcao == '5':
            yacc.parser.file = open("vmFiles/ex5.vm", "w")
            with open('funcFiles/ex5.txt', 'r') as file:
                program = file.read().rstrip()
            yacc.parser.parse(program)
            opcao = 0
        elif opcao == '6':
            yacc.parser.file = open("vmFiles/ex6.vm", "w")
            with open('funcFiles/ex6.txt', 'r') as file:
                program = file.read().rstrip()
            yacc.parser.parse(program)
            opcao = 0
        elif opcao == '0':
            print("Programa terminado.\n")
            opcao = 0
        else:
            print("Valor invalido")
            opcao = input("Escolha uma opcao:")
init()
```

6.2 Parser

```
import ply.lex as lex
import sys

tokens = [

    'ID','NINT',

    'PLUS','MINUS','MULT','DIV','MOD', 'EQEQ',
    'AND', 'OR', 'LT', 'LE', 'GT', 'GE', 'NEQ',

    'ASSIGN', 'PLUSPLUS', 'MINUSMINUS',
    'PLUSEQ','MINUSEQ', 'MULTEQ','DIVEQ','MODEQ',

    'FUNCTIONS','DECLARATIONS','BEGIN','WRITE','READ','IF'
    ,'THEN','ELSE','END','PHRASE', 'IntWord','WHILE','UNTIL','ArrayInt','FOR',
    'DEF','FUNC_NAME','RETURN',

    'LROUND','RROUND','LSQUARE','RSQUARE',
    'LCURLY','RCURLY', 'VIR','PONTeVIR', 'DOLLAR'
]

#Operators(+,-,*,/,%, <, ==, &&, ||, <, <=, >, >=, !=)
t_PLUS = r'\+'
t_MINUS = r'-'
t_MULT = r'\*'
t_DIV = r'/'
t_MOD = r'%'
t_EQEQ = r'=='
t_AND = r'&&'
t_OR = r'\|\|'
t_LT = r'<'
t_LE = r'<='
t_GT = r'>'
t_GE = r'>='
t_NEQ = r'!='

#Assign(=, ++, +=, -=, *=, /=, %=)
t_ASSIGN = r'='
t_PLUSPLUS = r'\+\+'
t_MINUSMINUS = r'\-\-'
t_PLUSEQ = r'\+\='
t_MINUSEQ = r'\-\='
t_MULTEQ = r'\*\='
t_DIVEQ = r'\/\='
```

```
t_MODEQ = r'\%\='

t_LROUND = r'\('
t_RROUND = r'\)'
t_LSQUARE = r'\['
t_RSQUARE = r'\]'
t_LCURLY = r'\{'
t_RCURLY = r'\}'
t_PONTeVIR = r';'
t_VIR = r','
t_DOLLAR = r'\$'


def t_BEGIN(t):
    r'(?i:begin)'
    return t

def t_WRITE(t):
    r'(?i:write)'
    return t

def t_READ(t):
    r'(?i:read)'
    return t

def t_DECLARATIONS(t):
    r'(?i:declarations)'
    return t

def t_IntWord(t):
    r'(?i:int)'
    return t

def t_ArrayInt(t):
    r'(?i:arrayint)'
    return t

def t_IF(t):
    r'(?i:if)'
    return t

def t_THEN(t):
    r'(?i:then)'
    return t

def t_ELSE(t):
```

```
    r'(?i:else)'
    return t

def t_END(t):
    r'(?i:end)'
    return t

def t_PHRASE(t):
    r'\"[a-zA-Z0-9 =:,\"\n]+'
    return t

def t_FUNC_NAME(t):
    r'[a-zA-Z_][a-zA-Z0-9_]*\(\)'
    return t

def t_WHILE(t):
    r'(?i:while)'
    return t

def t_FOR(t):
    r'(?i:for)'
    return t

def t_FUNCTIONS(t):
    r'(?i:functions)'
    return t

def t_RETURN(t):
    r'(?i:return)'
    return t

def t_DEF(t):
    r'(?i:def)'
    return t

def t_ID(t):
    r'[a-zA-Z_][a-zA-Z0-9_]*'
    return t

def t_NINT(t):
    r'[0-9]+'
    t.value = int(t.value)
    return t

def t_COMMENT(t):
    r'\#.*'
```

```
    pass


t_ignore = ' \r\n\t'

def t_error(t):
    print('Illegal character: ' + t.value[0])
    t.lexer.skip(1)
    return

lexer = lex.lex()
```

6.3 Analisador Léxico

```
import ply.yacc as yacc

from tp1_lexer import tokens

# guardas as variaveis num dicionario
variables = {}
functions = []
countWHILE = 0
countFor = 0
countIF = 0


def p_LstPrograms_Program(p):
    '''
    LstPrograms : Program
    '''
    parser.file.write(p[1])

def p_LstPrograms_LstPrograms(p):
    '''
    LstPrograms : LstPrograms Program
    '''
    p[0] = p[1] + p[2]

def p_Program(p):
    '''
    Program : DECLARATIONS LCURLY LstDecl RCURLY Fs BEGIN LstInst END
    '''
    p[0] = p[3] + "JUMP main\n" + p[5] + "main: nop\n" + p[7] + "STOP"

#
##
## DECLARATIONS ------------------------------------
##
#

def p_LstDecl_Decl(p):
    '''
    LstDecl : Decl
    '''
    p[0] = p[1] + "START\n"

def p_LstDecl_LstDecl(p):
    '''
```

```
    LstDecl : Decl LstDecl
    '''
    p[0] = p[1] + p[2]


def p_Decl_Int(p):
    '''
    Decl : IntWord Variables PONTeVIR
    '''
    p[0] = p[2]

def p_Decl_ArrayInt(p):
    '''
    Decl : ArrayInt Variables PONTeVIR
    '''
    p[0] = p[2]


def p_Variables_LstVar(p):
    '''
    Variables : Var VIR Variables
    '''
    p[0] = p[1] + p[3]


def p_Variables_Var(p):
    '''
    Variables : Var
    '''
    p[0] = p[1]


def p_Var_ID(p):
    '''
    Var : ID
    '''
    p[0] = "PUSHI 0\n"
    variables[p[1]] = -1

def p_Var_Array(p):
    '''
    Var : ID LSQUARE NINT RSQUARE
    '''
    p[0] = "PUSHN " + str(p[3]) + "\n"
    variables[p[1]] = [-1] * p[3]
```

```
#
##---------------------------------------

def p_Fs_Empty(p):
    '''
    Fs :
    '''
    p[0] = ""

def p_Fs_LstFuntc(p):
    '''
    Fs : FUNCTIONS DOLLAR LstFunct DOLLAR
    '''
    p[0] = p[3]

def p_LstFunct_Func(p):
    '''
    LstFunct : Func
    '''
    p[0] = p[1]

def p_LstFunct_Recursive(p):
    '''
    LstFunct : LstFunct Func
    '''
    p[0] = p[1] + p[2]

def p_Func(p):
    '''
    Func : DEF FUNC_NAME LCURLY LstInst RETURN ID PONTeVIR RCURLY
    '''
    x = p[2]
    x = x[:len(x) - 2]

    if x not in functions:
        functions.append(x)
        p[0] = x + ": nop\n" + p[4] + p[5].upper() + "\n" + "ret: nop" +

        "\n" + "PUSHG " + f"{get_indexVar(p[6])}\n" + "JUMP endret\n"
    else:
        p[0] = f"ERR 'A funcao {x} ja esta definida.'\nSTOP\n"

##---------------------------------------

#
# Lista de instruções de cada programa
```

```
#

def p_LstInst_Instruction(p):
    '''
    LstInst : Instruction
    '''
    p[0] = p[1]

def p_LstInst_Recursive(p):
    '''
    LstInst : LstInst Instruction
    '''
    p[0] = p[1] + p[2]


#
# Instruction - atribuição, função ou if
#
def p_Instruction(p):
   '''
   Instruction : Atrib
               | Function
               | ifStatement
               | Loop
   '''
   p[0] = p[1]
```

```
#
# Loop - Ciclo for, while, while do, repeat-until
#

def p_Loop(p):
    '''
    Loop : WHILE LROUND Condition RROUND LCURLY LstInst RCURLY
         | FOR LROUND Atrib Atrib Condition RROUND LCURLY LstInst RCURLY

    '''
    if p[1].upper() == "WHILE":
        parser.countWHILE += 1
        p[0] = "WHILE" + str(parser.countWHILE) + ":\n" + p[3] + "JZ ENDWHILE"

        + str(parser.countWHILE) +

        "\n" + p[6] +

        "JUMP WHILE" + str(parser.countWHILE)

        + "\nENDWHILE"

        + str(parser.countWHILE) + ":\n"

    elif p[1].upper() == "FOR":
        parser.countFor += 1

        p[0] =  p[3] + "FOR" + str(parser.countFor)

        + ":\n" + p[5] + "JZ ENDFOR"

        + str(parser.countFor) + "\n"

        + p[8] + p[4]+ "JUMP FOR"

        + str(parser.countFor)


        + "\nENDFOR" + str(parser.countFor) + ":\n"


#
# ifStatement - if then else, if then
#

def p_ifStatementThen(p):
```

```
    '''
    ifStatement : IF LROUND Condition RROUND THEN LCURLY LstInst RCURLY
    '''
    parser.countIF += 1
    p[0] = p[3] + "JZ FIM"

    + str(parser.countIF) + "\n"


    + p[7] + "JUMP FIM"


    + str(parser.countIF) + "\n"


    + "FIM" + str(parser.countIF) + ":\n"


def p_ifStatement(p):
    '''
    ifStatement : IF LROUND Condition RROUND

    THEN LCURLY LstInst RCURLY

    ELSE LCURLY LstInst RCURLY
    '''
    parser.countIF += 1

    p[0] = p[3] + "JZ ELSE" + str(parser.countIF) + "\n"

    + p[7] + "JUMP FIM" + str(parser.countIF)

    + "\n" + "ELSE" + str(parser.countIF)

    + ":\n" + p[11] + "FIM" + str(parser.countIF) + ":\n"


def p_Atrib(p):
    '''
    Atrib : ID ASSIGN Expr PONTeVIR
          | ID PLUSEQ Expr PONTeVIR
          | ID MINUSEQ Expr PONTeVIR
          | ID MULTEQ Expr PONTeVIR
          | ID DIVEQ Expr PONTeVIR
          | ID MODEQ Expr PONTeVIR
          | ID PLUSPLUS PONTeVIR
```

```
      | ID MINUSMINUS PONTeVIR
'''
if p[1] in variables:
    if p[2] == "=":
        p[0] = p[3] + "STOREG " + f"{get_indexVar(p[1])}\n"

    elif p[2] == "+=":
        p[0] = "PUSHG " + f"{get_indexVar(p[1])}\n"

        + p[3] + "ADD\n" + "STOREG "

        + f"{get_indexVar(p[1])}\n"

    elif p[2] == "-=":
        p[0] = "PUSHG " + f"{get_indexVar(p[1])}\n"

        + p[3] + "SUB\n" + "STOREG "

        + f"{get_indexVar(p[1])}\n"

    elif p[2] == "*=":
        p[0] = "PUSHG " + f"{get_indexVar(p[1])}\n"

        + p[3] + "MUL\n" + "STOREG "

        + f"{get_indexVar(p[1])}\n"

    elif p[2] == "/=":
        p[0] = "PUSHG " + f"{get_indexVar(p[1])}\n" +

        p[3] + "DIV\n" + "STOREG " +

        f"{get_indexVar(p[1])}\n"

    elif p[2] == "%=":
        p[0] = "PUSHG " + f"{get_indexVar(p[1])}\n"

        + p[3] + "MOD\n" + "STOREG "

        + f"{get_indexVar(p[1])}\n"

    elif p[2] == "++":
        p[0] = "PUSHG " + f"{get_indexVar(p[1])}\n"

        + "PUSHI 1\n" + "ADD\n" + "STOREG "
```

```
            + f"{get_indexVar(p[1])}\n"

        elif p[2] == '--':
            p[0] = "PUSHG " + f"{get_indexVar(p[1])}\n"

            + "PUSHI 1\n" + "SUB\n" + "STOREG "

            + f"{get_indexVar(p[1])}\n"
    else:
        p[0] = f"ERR 'A variável {p[1]} não existe'\nSTOP\n"


def p_Atrib_Array(p):
    '''
    Atrib : ID LSQUARE Expr RSQUARE ASSIGN ExprR PONTeVIR
    '''
    arrayindex = get_indexVar(p[1])
    p[0] = f"PUSHGP\nPUSHI {arrayindex}\n{p[3]}ADD\n{p[6]}STOREN\n"
#
# Function - write and read function
#

def p_Function_WritePHRASE(p):
    '''
    Function : WRITE LROUND PHRASE RROUND PONTeVIR
    '''
    p[0] = "PUSHS " + p[3] + "\n" + "WRITES\n"


def p_Function_WriteExprR(p):
    '''
    Function : WRITE LROUND ExprR RROUND PONTeVIR
    '''
    p[0] = p[3] + "WRITEI\n"


#
# Condition - AND e OR
#

def p_Condition(p):
    '''Condition : ExprR'''
    p[0] = p[1]

def p_Condition_AND(p):
    '''Condition : ExprR AND Condition'''
```

```
    p[0] = p[1] + p[3] + "MUL\n"

def p_Condition_UNTIL(p):
    '''Condition : ExprR UNTIL ExprR'''
    p[0] = p[1] + p[3] + "INFEQ\n"

def p_Condition_OR(p):
    '''Condition : ExprR OR Condition'''
    p[0] = p[1] + p[3] + "ADD\n"

#
# ExprR - operadores para comparacoes
#

def p_ExprR(p):
    '''ExprR : Expr'''
    p[0] = p[1]

# operador  ==
def p_ExprR_EQEQ(p):
    '''ExprR : Expr EQEQ Expr'''
    p[0] = p[1] + p[3] + "EQUAL\n"

def p_ExprR_NOTEQ(p):
    '''ExprR : Expr NEQ Expr'''
    p[0] = p[1] + p[3] + "EQUAL\n"

# operador  <
def p_ExprR_LT(p):
    '''ExprR : Expr LT Expr'''
    p[0] = p[1] + p[3] + "INF\n"


# operador  <=
def p_ExprR_LE(p):
    '''ExprR : Expr LE Expr'''
    p[0] = p[1] + p[3] + "INFEQ\n"


# operador  >
def p_ExprR_GT(p):
    '''ExprR : Expr GT Expr'''
    p[0] = p[1] + p[3] + "SUP\n"

# operador  >=
def p_ExprR_GE(p):
```

```
    '''ExprR : Expr GE Expr'''
    p[0] = p[1] + p[3] + "SUPEQ\n"


#
# Expr - Soma e subtracao
#

def p_Expr_Term(p):
    '''Expr : Term'''
    p[0] = p[1]


def p_Expr_PLUS(p):
    '''Expr : Expr PLUS Term'''
    p[0] = p[1] + p[3] + "ADD\n"


def p_Expr_MINUS(p):
    '''Expr : Expr MINUS Term'''
    p[0] = p[1] + p[3] + "SUB\n"


#
# Term - faz multiplcacoes, divisoes e modulos
#

def p_Term(p):
    '''Term : Factor'''
    p[0] = p[1]

def p_Term_Mult(p):
    '''Term : Term MULT Factor'''
    p[0] = p[1] + p[3] + "MUL\n"


def p_Term_Div(p):
    '''Term : Term DIV Factor'''
    p[0] = p[1] + p[3] + "DIV\n"


def p_Term_MOD(p):
    '''Term : Term MOD Factor'''
    p[0] = p[1] + p[3] + "MOD\n"
```

```
#
# Factor - deteta um inteiro, uma variavel
#

# PUSHG "index da variavel no dic variables"
def p_Factor_ID(p):
    '''Factor : ID'''
    if p[1] not in variables:
        p[0] = f"ERR 'Variavel {p[1]} não existe'\nSTOP\n"
    else:
        p[0] = f"PUSHG {get_indexVar(p[1])}\n"

def p_Factor_NINT(p):
    '''Factor : NINT'''
    p[0] = "PUSHI " + str(p[1]) + '\n'

def p_Factor_MinusNint(p):
    '''Factor : LROUND MINUS NINT RROUND'''
    x = -1 * p[3]
    p[0] = "PUSHI " + str(x) + '\n'

def p_FactorRead(p):
    '''
    Factor : READ LROUND RROUND
    '''
    p[0] = "READ\n" + "ATOI\n"

def p_FactorFunc(p):
    '''
    Factor : FUNC_NAME
    '''
    x = p[1]
    x = x[:len(x) - 2]
    p[0] = "PUSHA " + x + "\n" + "CALL\n" + "nop\n" + "JUMP ret\n" + "endret: nop\n"

def p_FactorArray(p):
    '''
    Factor : ID LSQUARE NINT RSQUARE
    '''
    if p[1] not in variables.keys():
        p[0] = f"ERR 'Variavel {p[1]} não existe'\nSTOP\n"
    else:
        x = p[3]
        if len(variables[p[1]]) > int(x) and int(x) >= 0:
            p[0] = "PUSHG " + p[3] + f"{get_indexVar(p[1]) + int(x)}\n"
        elif len(variables[p[1]]) <= int(x):
```

```
            p[0] = f"ERR 'O valor {p[3]} maior que o tamanho do array {p[1]}. '\nSTOP\n"
        elif int(x) < 0:
            p[0] = f"ERR 'O valor {p[3]} nao pode ser negativo. '\nSTOP\n"

def p_FactorArrayID(p):
    '''
    Factor : ID LSQUARE ID RSQUARE
    '''
    if p[1] not in variables.keys():
        p[0] = f"ERR 'Variavel {p[1]} não existe'\nSTOP\n"
    else:
        x = get_indexVar(p[3])
        if (get_indexVar(p[1]) < x):
            x += len(variables[p[1]]) - 1

        if len(variables[p[1]]) >= int(x) and int(x) >= 0:
            p[0] = "PUSHG " + f"{get_indexVar(p[1]) + int(x)}\n"
        elif len(variables[p[1]]) < int(x):
            p[0] = f"ERR 'O valor {p[3]} maior que o tamanho do array {p[1]}. '\nSTOP\n"
        elif int(x) < 0:
            p[0] = f"ERR 'O valor {p[3]} nao pode ser negativo. '\nSTOP\n"

def p_error(p):
    parser.success = False
    print('Syntax error!')


def get_indexVar(id):
    count  = 0
    for key in variables.keys():
        if key == id:
            return count
        else:
            count+=1


###inicio do parsing
parser = yacc.yacc()
parser.countWHILE = 0
parser.countFor = 0
parser.countIF = 0
parser.success = True
```

7 Conclusão

Neste trabalho prático fomos capazes de criar uma linguagem de programação imperativa simples, onde era possível declarar variáveis atómicas do tipo inteiro, efetuar atribuições algoritmicas, como por exemplo atribuições do valor de expressões numéricas a variáveis, ler do *standard input* e escrever no *standard output*, efetuar instruções condicionais e instruções cíclicas, como por exemplo *if* e *while* para controlo do fluxo de execução. Desenvolvemos também um compilador para essa linguagem com base na *GIC* com recurso ao módulo *YACC* e *LEX*. O compilador gera **pseudo-código** na máquina virtual. Posto isto, conclui-se o presente relatório com uma apreciação positiva do trabalho realizado.