Universidade do Minho
Escola de Engenharia
Licenciatura em Engenharia Informática

Unidade Curricular de Processamento de Linguagens

Ano Letivo de 2024/2025

Construção de um Compilador para Pascal Standard

António Fonseca a93167
Beatriz Almeida a93210
João Moreira a93326

1 de junho de 2025

PL

Resumo

Este relatório apresenta uma visão detalhada do projeto final da Unidade Curricular de Processamento de Linguagens, concebido para converter código da linguagem de programação Pascal num conjunto de instruções de código máquina que irá ser testado numa máquina virtual disponibilizada pelos docentes da cadeira. Ao longo deste relatório, serão abordados inúmeros conceitos e implementados diferentes conhecimentos que foram adquiridos durante o decorrer das aulas. Este projeto foi desenvolvido em Python, recorrendo à biblioteca PLY (Python Lex-Yacc) para realizar a análise léxica e sintática, com o intuito de obter um compilador da linguagem Pascal de alto nível para instruções da máquina virtual.

Palavras-chave compilador, análise léxica, análise sintática, gramática, máquina virtual, linguagem pascal, código máquina, PLY-LEX, PLY-YACC

Índice

Lista de Figuras

1. Introdução

Este projeto teve como objetivo desenvolver um compilador para a linguagem Pascal, capaz de transformar código-fonte em instruções executáveis para uma máquina virtual. O trabalho seguiu um fluxo com diferentes etapas, começando pela análise léxica onde o código é decomposto em tokens como identificadores, palavras-chave e operadores. De seguida, na análise sintática, estes tokens são organizados numa estrutura gramatical que valida a conformidade com as regras da linguagem. Por fim, o sistema gera o código otimizado para a execução na máquina virtual, completando assim o processo de compilação.

1.1. Motivação

O desenvolvimento deste projeto surge da necessidade de aprofundar e dominar diversos conceitos lecionados na Unidade Curricular de Processamento de Linguagens. De forma a ganhar experiência em engenharia de linguagens e programação gramatical, o projeto permite reforçar a capacidade de escrever gramáticas, bem como aplicar, de forma prática, os termos da linguagem de programação utilizada (Python). Desta forma, é possível solidificar a matéria teórica através de uma abordagem mais prática, promovendo uma aprendizagem mais eficiente e integrada.

1.2. Objetivos

O principal objetivo deste projeto, desenvolvido no âmbito da Unidade Curricular de Processamento de Linguagens, consiste na criação de um compilador com o propósito de transformar código da linguagem Pascal em código máquina. Para alcançar este fim, o trabalho foi estruturado de forma a cumprir também os seguintes objetivos:

1. Disponibilizar uma ferramente capaz de analisar, interpretar e traduzir código Pascal;
2. O compilador deve ser capaz de processar a declaração de variáveis, expressões aritméticas, comandos de controlo de fluxo (if, while, for, etc...);
3. A ferramenta deve devolver o resultante código máquina de forma a correr na máquina virtual disponibilizada;

4. Aprofundar e dominar conceitos lecionadas na unidade curricular.

1.3. Estrutura do Documento

O primeiro capítulo deste relatório apresenta o propósito do projeto, fornecendo contexto sobre este, a motivação por trás do seu desenvolvimento, os objetivos e, finalmente, a estrutura do documento.

No segundo capítulo, abordamos a concepção e implementação do projeto. Aprofundamos as diferentes etapas pelo qual o projeto avançou e também as funcionalidades implementadas.

O capítulo três apresenta os testes feitos e os resultados obtidos, a partir de vários ficheiros exemplo de programas escritos em Pascal.

O capítulo final consiste na síntese das principais conclusões obtidas, bem como na exploração do trabalho futuro.

Por fim, existe também uma última seção onde está presente o código do programa.

2. Concepção e Implementação

2.1. Linguagem de Programação Pascal

Para que este projeto possa cumprir o seu objetivo de analisar, interpretar e traduzir código Pascal para instruções máquina, é fundamental compreendermos a sintaxe característica desta linguagem. A sintaxe de Pascal apresenta várias particularidades que influenciam diretamente o desenvolvimento do compilador:

- **Estrutura Hierárquica:** Divisão clara entre as diferentes seções do programa:
  - Definição do programa principal;
  - Declaração de subprogramas;
  - Seção de declaração de variáveis;
  - Bloco de instruções executáveis;
- **Delimitadores Específicos:**
  - Blocos principais delimitados por *begin* e *end*;
  - Blocos secundários em estruturas de controlo podem omitir os delimitadores quando contêm apenas uma instrução;
  - Ponto e vírgula como terminador de instruções;
  - Dois pontos para declarações de tipo;
- **Palavras-chave:** Termos reservados para instruções específicas:
  - Controlo de fluxo: *while*, *for*, *if*, *then*, *else*
  - Declarações: *var*, *function*;
  - Tipos de dados: *integer*, *real*, *boolean*, *string*;
  - Entre outros.

Com uma melhor compreensão da sintaxe e semântica desta linguagem, podemos proceder à

concepção e implementação do nosso compilador.

2.2. Lex

A biblioteca PLY é composta por dois módulos fundamentais, que irão formar o nosso compilador. Inicialmente, o módulo *Lex* realiza a análise léxica: ao receber um texto de entrada, divide-o em *tokens*, que representam as unidades significativas menores do programa. O *lexer* utiliza a biblioteca Python Lex-Yacc (PLY) para definir os *tokens* a serem reconhecidos, associando a cada um deles uma regra baseada numa expressão regular. Quando o *lexer* encontra uma *string* que corresponde a uma destas regras, gera um *token* do tipo correspondente.

2.2.1. *Tokens*

No ficheiro *lexer.py* foram definidos diferentes tipos de *tokens*. Este ficheiro contém *tokens* que representam **palavras-chave**, ou seja, termos reservados que indicam instruções ou comportamentos específicos da linguagem. Palavras como **PROGRAM**, **VAR**, **BEGIN**, entre outras, indicam a estrutura e o funcionamento do programa.

Além disso, foram definidos **operadores e símbolos**. Estes *tokens* representam expressões lógicas, relacionais e aritméticas, bem como símbolos auxiliares utilizados para estruturar determinadas instruções da linguagem.

Por fim, temos os **identificadores e literais**. Foi definido o *token* **ID**, utilizado como identificador de variáveis ou nomes de funções, o **NUM**, que representa um número inteiro, e o **STRING_LITERAL**, que permite imprimir mensagens de texto na aplicação.

```
tokens = [
    # Palavras-chave
    'PROGRAM', 'FUNCTION', 'VAR','BEGIN', 'END', 'WRITELN', '
       WRITE','READLN', 'INTEGER', 'STRING','BOOLEAN', 'ARRAY',
       'OF',
    'IF', 'THEN', 'ELSE', 'FOR', 'TO', 'DOWNTO', 'DO', 'WHILE',
       'TRUE', 'FALSE',

    # Operadores e símbolos
    'PLUS', 'MINUS', 'TIMES', 'DIVIDE', 'ASSIGN', 'EQUAL', 'LESS
       ', 'LESSEQUAL', 'GREATER', 'GREATEREQUAL',
    'LPAREN', 'RPAREN', 'LBRACKET', 'RBRACKET', 'SEMICOLON', '
       COLON','COMMA', 'DOT', 'MOD', 'DIV', 'AND', 'OR', 'NOT',

    # Identificadores e literais
    'ID', 'NUMBER', 'STRING_LITERAL'
]
```

2.2.2. Especificação de *Tokens*

Cada *token* é especificado através de uma expressão regular. Cada uma destas regras é definida por declarações com o prefixo especial `t_`, que indicam a definição de um *token*. Para *tokens* simples — neste caso, os **operadores e símbolos**, que consistem em um ou dois caracteres — a expressão regular é representada como uma *raw string*, enquanto os restantes *tokens* são especificados através de funções com o mesmo propósito. A especificação dos *tokens* encontra-se nos apêndices.

2.3. Yacc

Este módulo implementa um analisador sintático para validar a estrutura gramatical do código. Cada regra gramatical é definida por uma função em Python, onde a *docstring* dessa função contém a especificação apropriada da gramática livre de contexto. As instruções que compõem o corpo da função implementam as ações semânticas da regra.

No desenvolvimento do compilador, utilizou-se um conjunto de variáveis globais para gerir de forma eficiente o estado do *parser* durante a análise e geração de código.

```
parser.var = dict()
parser.funcs = set()
parser.called_funcs = set()
parser.offset = 0
parser.ifs = 0
parser.cycles = 0
parser.current_type = None
```

As variáveis globais no *parser* foram essenciais, cada uma tendo um papel específico:

- **parser.var:** Armazenar informações sobre variáveis declaradas;
- **parser.funcs:** Registar funções declaradas para verificação de chamadas válidas;
- **parser.called_funcs:** Verificar funções chamadas para validar se foram declaradas;
- **parser.offset:** Calcular posições de memória para variáveis na pilha da VM;
- **parser.ifs:** Criar labels únicos para estruturas de controlo de instruções condicionais aninhadas;
- **parser.cycles:** Gerar labels únicos para estruturas de controlo de instruções cíclicas aninhadas;
- **parser.current_type:** Controlar o tipo atual durante as declarações de variáveis.

2.3.1. Programa

Primeiramente, decidimos dividir o nosso programa em quatro seções principais, a declaração do programa, a declaração de subprogramas, caso se aplique, a declaração das variáveis do programa e o conjunto de instruções, onde são executadas todas as instruções.

```
def p_Program(p):
    '''Program : PROGRAM ID SEMICOLON Function VarsDeclaration
       CodeBlock DOT'''
```

2.3.2. Declaração de Variáveis

Na gramática do projeto, é possível haver múltiplas declarações, podendo estas ser de variáveis do tipo *integer*, *boolean* e *string*, ou ainda de *arrays*. As declarações podem estar agrupadas numa mesma linha, caso tenham o mesmo tipo, ou distribuídas por várias linhas, mesmo que sejam do mesmo tipo ou de tipos diferentes.

Quando uma variável é declarada, é guardado no dicionário de variáveis *parser.var*, juntamente com o seu tamanho, correspondendo a 1, o tipo, sendo 0 para *integer* e *boolean*, e 1 para *string*, e o respetivo *offset* na *stack*, de forma a poder ser posteriormente acedida.

```
def p_Declaration(p):
    '''Declaration : VariablesList COLON Type SEMICOLON'''
    for var in p[1]:
        add_var(var, 1, p.parser.current_type, p)
```

A declaração de *arrays* segue um processo praticamente idêntico. As mesmas informações são armazenadas no dicionário, sendo ainda adicionado o limite inferior do *array*, que será utilizado aquando do seu acesso. Para além disso, o tamanho corresponde à diferença entre o limite superior e o limite inferiro, adicionando um.

```
def p_Declaration_array(p):
    '''Declaration : ID COLON ARRAY LBRACKET NUMBER DOT DOT
       NUMBER RBRACKET OF INTEGER SEMICOLON'''
    size = int(p[8]) - int(p[5]) + 1
    lower_bound = int(p[5])
    add_var(p[1], size, 0, p, lower_bound)
```

2.3.3. Instruções

O compilador é capaz de analisar e transformar diferentes tipos de instruções. O programa escreve no *standart output* todos os tipos de variáveis definidos como também qualquer tipo de *string* e lê do *standard input* para inicializar as variáveis que já tinham sido declaradas anteriormente, guardando o valor lido na posição de memória, que lhe foi atribuída previamente.

Para além disso, é capaz de analisar diversas outras instruções, nomeadamente, instruções condicionais e cíclicas.

Condicionais

Para a escrita de expressões condicionais, considerou-se duas hipóteses. Utilização de um *if* e de um *then* e utilização de um *if then else*. Para se realizar os saltos necessários, guarda-se um contador do número de *ifs* no programa, ficando cada instrução condicional com um "id" associado. No primeiro caso, no momento da entrada na instrução é testada uma condição. Se esta for falsa, é executado o comando *jz* para saltar para o fim do *if*. Caso contrário, são executadas as instruções dentro do bloco de código.

No caso de existir um *else*, os saltos funcionam de maneira diferente. Após a condição, é realizado um *jz* para o inicio do código do *else*, caso esta seja falsa, e são executadas as instruções do *if* caso contrário. No fim do código do if, é utilizado o comando *jump* para o fim de toda a instrução condicinal de modo a não ser executado o código do *else*.

Cíclicas

A escrita de expressões cíclicas possui três métodos principais: quando é utilizado um *while* seguido de um *do*; um *for-to* seguido de um *do*; ou um *for-downto*, acabando num *do*. No ciclo *while* a condição é testada no início. Caso esta seja verdadeira entra no ciclo, caso contrário é realizado um *jz* para fora deste. Além disso, ainda é realizado um *jump* no fim de cada iteração para o início para a condição voltar a ser testada.

O segundo método segue os mesmos princípios que o ciclo *while*, mudando apenas o facto de existir uma atribuição antes do início do ciclo e outra no fim de cada iteração, correspondendo ao incremento por uma unidade à variável condicional.

Por fim, o terceiro método apenas difere do segundo que no final da iteração ao invés de incrementar, diminui a variável condicional.

2.3.4. Operações aritméticas, relacionais e lógicas

De modo a realizar operações aritméticas, relacionais e lógicas, utilizou-se a seguinte precedência de operações:

- **Multiplicação:** * / div mod
- **Adição:** + -
- **Relacional:** < <= > >= ==
- **Lógico:** and or not

2.3.5. Outros métodos

De modo a controlar os erros do nosso compilador teve-se em atenção alguns pormenores. Primeiramente, de modo a não haver chamadas a funções inexistentes, verifica-se se o *set parser.called_funcs* é um *subset* do *set parser.funcs*. Caso seja falso, é levantada uma exceção.

Do mesmo modo, quando adicionamos uma variável aos registos, é verificado se esta já existe. Caso exista, é também levantada uma exceção.

Todas estas exceções são depois apanhadas juntamente com alguns erros de sintaxe quando se está a processar a escrita do ficheiro *.vm*. Em caso de erro, o ficheiro de output é removido.

```
try:
    result = parser.parse(data)
    fileOut.write(result)
except (TypeError, Exception):
    print("\nCompilation error, aborting!")
    os.remove(outFilePath)
```

3. Resultados Obtidos

Neste capítulo são apresentados os testes realizados durante o desenvolvimento do projeto, bem como a análise dos resultados obtidos. Inicialmente, é apresentado um exemplo de um programa escrito em Pascal e, de seguida, o correspondente código máquina gerado pelo compilador desenvolvido.

3.1. Exemplo 1: Olá, Mundo!

O seguinte bloco de código consiste num programa simples em Pascal que imprime "Ola, Mundo!"no *standard output*, utilizando o comando *writeln*.

```
program HelloWorld;
begin
  writeln('Olá, mundo!');
end.
```

Figura 3.1.: VM ex1

3.2. Exemplo 2: Maior de 3

Este exemplo lê três números inteiros, introduzidos pelo utilizador, e determina qual deles é o maior, através de várias comparações condicionais. Por fim, imprime o maior valor encontrado.

```
program Maior3;
var
    num1, num2, num3, maior: Integer;
begin
    { Ler 3 números }
    Write('Introduza o primeiro número: ');
    ReadLn(num1);
    Write('Introduza o segundo número: ');
    ReadLn(num2);
    Write('Introduza o terceiro número: ');
    ReadLn(num3);

    { Calcular o maior }
    if num1 > num2 then
        if num1 > num3 then
            maior := num1
        else
            maior := num3
    else
        if num2 > num3 then
            maior := num2
        else
            maior := num3;

    { Escrever o resultado }
    WriteLn('O maior é: ', maior);
end.
```

Figura 3.2.: VM ex2

3.3. Exemplo 3: Fatorial

Este programa calcula o fatorial de um número inteiro positivo, fornecido pelo utilizador. Utiliza um ciclo *for* para multiplicar sequencialmente os números de 1 até n e, no final, imprime o resultado.

```
program Fatorial;
var
    n, i, fat: integer;
begin
    writeln('Introduza um número inteiro positivo:');
    readln(n);
    fat := 1;
    for i := 1 to n do
        fat := fat * i;
    writeln('Fatorial de ', n, ': ', fat);
end.
```

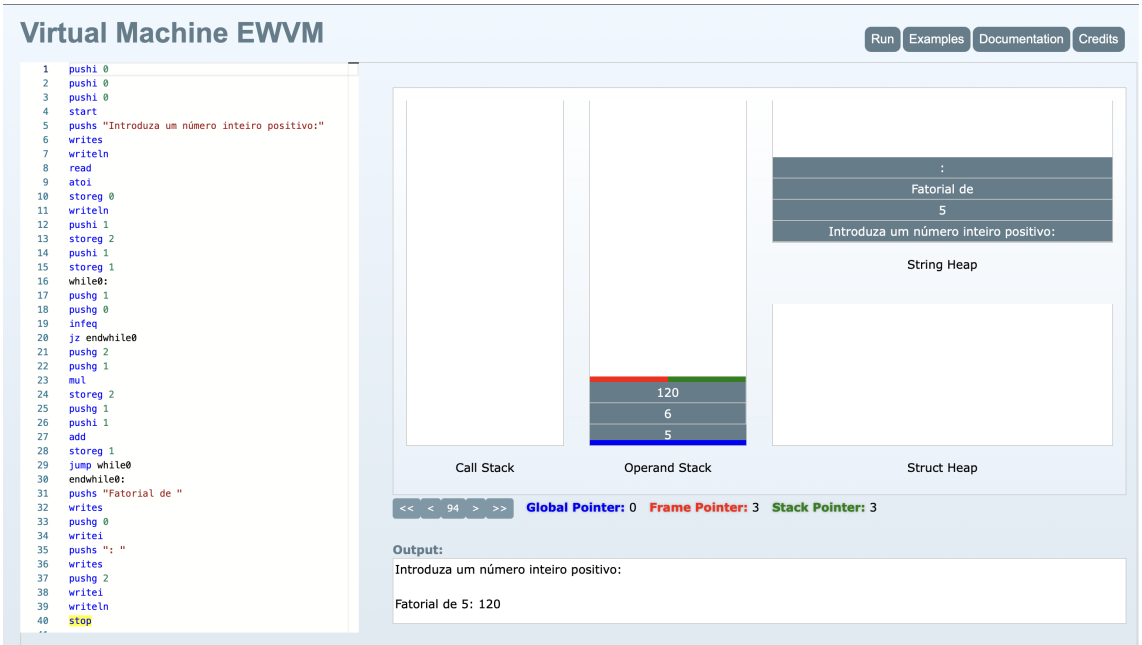

Figura 3.3.: VM ex3

3.4. Exemplo 4: Verificação de Número Primo

O seguinte exemplo verifica se um número inteiro positivo, fornecido pelo utilizador, é primo, exibindo o resultado final.

```
program NumeroPrimo;
var
    num, i: integer;
    primo: boolean;
begin
    writeln('Introduza um número inteiro positivo:');
    readln(num);
    primo := true;
    i := 2;
    while (i <= (num div 2)) and primo do
    begin
        if (num mod i) = 0 then
            primo := false;
        i := i + 1;
    end;
    if primo then
        writeln(num, ' é um número primo')
    else
```

```
        writeln(num, ' não é um número primo');
end.
```

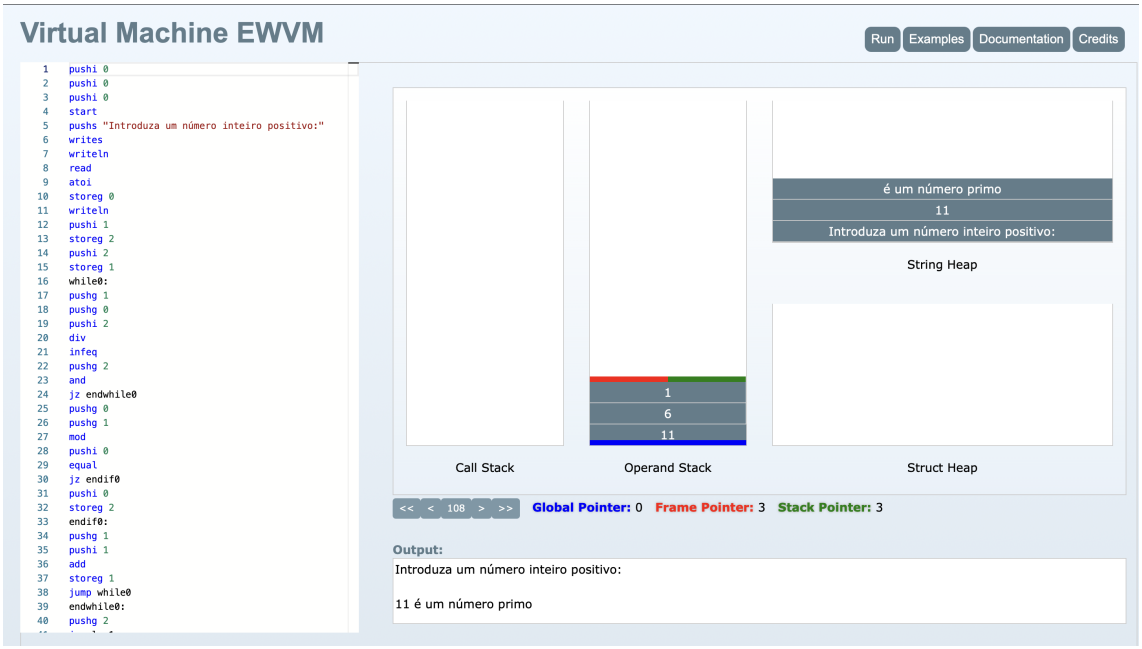

Figura 3.4.: VM ex4

3.5. Exemplo 5: Soma de uma lista de inteiros

Este programa lê cinco números inteiros introduzidos pelo utilizador, armazena-os num array e calcula a soma de todos os valores, devolvendo o resultado no final.

```
program SomaArray;
var
    numeros: array[1..5] of integer;
    i, soma: integer;
begin
    soma := 0;
    writeln('Introduza 5 números inteiros:');
    for i := 1 to 5 do
    begin
        readln(numeros[i]);
        soma := soma + numeros[i];
    end;
    writeln('A soma dos números é: ', soma);
end.
```

Figura 3.5.: VM ex5

3.6. Exemplo 6: Conversão binário-decimal

Este programa converte um número binário no seu valor inteiro correspondente. A função percorre a string do último para o primeiro carácter, somando potências de 2 sempre que encontra um '1', e devolve o resultado final.

```
program BinarioParaInteiro;
var
    bin: string;
    i, valor, potencia: integer;
begin
    writeln('Introduza uma string binária:');
    readln(bin);
    valor := 0;
    potencia := 1;
    for i := length(bin) downto 1 do
    begin
        if bin[i] = '1' then
            valor := valor + potencia;
        potencia := potencia * 2;
    end;
    writeln('O valor inteiro correspondente é: ', valor);
end.
```

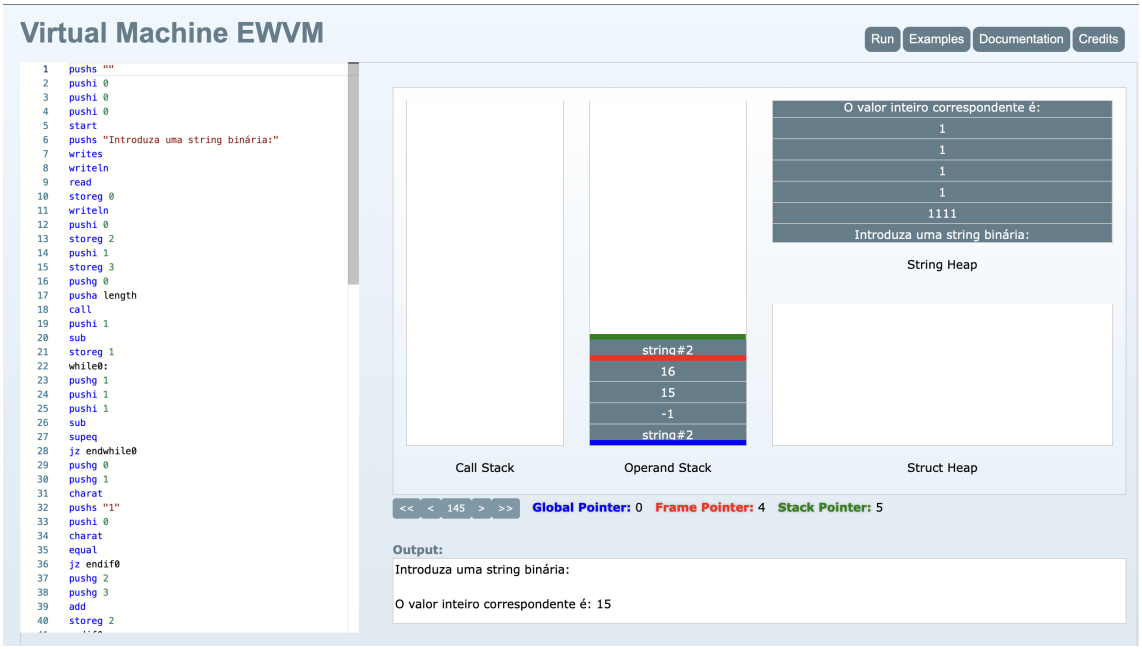

Figura 3.6.: VM ex6

4. Conclusões e Trabalho Futuro

Consideramos que este trabalho prático permitiu-nos consolidar melhor a matéria dada nas aulas da cadeira, podendo assim medir, de forma mais reslista, o aproveitamento que derivámos da UC. Ao longo deste projeto, conseguimos superar várias dificuldades, que se verificaram na realização deste projeto. O que nos fez ganhar um maior à vontade para com os conceitos lecionados, aprendendo muito sobre a interpretação de linguagens de programação e a utilização de analisadores léxicos e sintáticos. Concluindo, o resumo que o grupo retira deste projeto é que foi bem-sucedido. Para além disso, foi adquirida mais experiência no tema em questão, que se poderá tornar útil no futuro.

4.1. Trabalho Futuro

Como continuação deste projeto, poderia-se expandir a gramática e o compilador para suportar um conjunto mais amplo de estruturas da linguagem Pascal. Em particular, adicionar regras para a representação e análise de funções auxiliares e subprogramas, incluindo procedimentos e funções com parâmetros, *scope* local e retorno de valores. Esta extensão permitirá que o compilador analise programas mais complexos e reais, aumentando a sua utilidade e robustez. Além disso, será importante implementar mecanismos para verificar a correção semântica destas novas construções e a implementação de mais métodos de controlo da declaração de variáveis, bem como otimizar o processamento e geração do código correspondente.

A. Lex

```
import ply.lex as lex


tokens = [
    # Palavras-chave
    'PROGRAM', 'FUNCTION', 'VAR','BEGIN', 'END', 'WRITELN', '
       WRITE','READLN', 'INTEGER', 'STRING','BOOLEAN', 'ARRAY',
       'OF',
    'IF', 'THEN', 'ELSE', 'FOR', 'TO', 'DOWNTO', 'DO', 'WHILE',
       'TRUE', 'FALSE',

    # Operadores e símbolos
    'PLUS', 'MINUS', 'TIMES', 'DIVIDE', 'ASSIGN', 'EQUAL', 'LESS
       ', 'LESSEQUAL', 'GREATER', 'GREATEREQUAL',
    'LPAREN', 'RPAREN', 'LBRACKET', 'RBRACKET', 'SEMICOLON', '
       COLON','COMMA', 'DOT', 'MOD', 'DIV', 'AND', 'OR', 'NOT',

    # Identificadores e literais
    'ID', 'NUMBER', 'STRING_LITERAL'
]

t_LESSEQUAL     = r'<='
t_GREATEREQUAL  = r'>='
t_LESS          = r'<'
t_GREATER       = r'>'
t_PLUS          = r'\+'
t_MINUS         = r'-'
t_TIMES         = r'\*'
t_DIVIDE        = r'/'
t_ASSIGN        = r':='
t_EQUAL         = r'='
t_LPAREN        = r'\('
t_RPAREN        = r'\)'
t_LBRACKET      = r'\['
t_RBRACKET      = r'\]'
t_COMMA         = r','
t_COLON         = r':'
t_SEMICOLON     = r';'
t_DOT           = r'\.'
```

```
def t_PROGRAM(t):
    r'\bprogram\b'
    return t

def t_FUNCTION(t):
    r'\bfunction\b'
    return t

def t_VAR(t):
    r'\bvar\b'
    return t

def t_BEGIN(t):
    r'\bbegin\b'
    return t

def t_END(t):
    r'\bend\b'
    return t

def t_WRITELN(t):
    r'\bwriteln\b|\bWriteLn\b'
    return t

def t_WRITE(t):
    r'\b[wW]rite\b'
    return t

def t_READLN(t):
    r'\breadln\b|\bReadLn\b'
    return t

def t_INTEGER(t):
    r'\b[iI]nteger\b|\bINTERGER\b'
    return t

def t_STRING(t):
    r'\b[sS]tring\b|\bSTRING\b'
    return t

def t_BOOLEAN(t):
    r'\b[bB]oolean\b|\bBOOLEAN\b'
    return t

def t_ARRAY(t):
```

```
    r'\barray\b'
    return t

def t_OF(t):
    r'\bof\b'
    return t

def t_IF(t):
    r'\bif\b'
    return t

def t_THEN(t):
    r'\bthen\b'
    return t

def t_ELSE(t):
    r'\belse\b'
    return t

def t_FOR(t):
    r'\bfor\b'
    return t

def t_TO(t):
    r'\bto\b'
    return t

def t_DOWNTO(t):
    r'\bdownto\b'
    return t

def t_DO(t):
    r'\bdo\b'
    return t

def t_WHILE(t):
    r'\bwhile\b'
    return t

def t_TRUE(t):
    r'\btrue\b'
    return t

def t_FALSE(t):
    r'\bfalse\b'
    return t
```

```
def t_MOD(t):
    r'\bmod\b'
    return t

def t_DIV(t):
    r'\bdiv\b'
    return t

def t_AND(t):
    r'\band\b'
    return t

def t_OR(t):
    r'\bor\b'
    return t

def t_NOT(t):
    r'\bnot\b'
    return t


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

def t_NUMBER(t):
    r'\d+'
    t.value = int(t.value)
    return t

def t_STRING_LITERAL(t):
    r"'(.*?)'"
    t.value = t.value[1:-1]
    return t

def t_COMMENT(t):
    r'\{.*?\}'
    pass

t_ignore = ' \t'

def t_newline(t):
    r'\n+'
    t.lexer.lineno += len(t.value)


def t_error(t):
```

```
    print(f"Illegal character '{t.value[0]}' at line {t.lineno},
        column {t.lexpos}")
    t.lexer.skip(1)

# ---------------- Lexer ----------------
lexer = lex.lex()
```

B. Yacc

```
import ply.yacc as yacc
from lexer import tokens
import logging
import os

precedence = (
    ('left', 'OR'),
    ('left', 'AND'),
    ('right', 'NOT'),
    ('nonassoc', 'IFX'),
    ('nonassoc', 'ELSE'),
    )

# ---------------------------------------- Program
    ----------------------------------------

def p_Program(p):
    '''Program : PROGRAM ID SEMICOLON Function VarsDeclaration
       CodeBlock DOT'''
    missing_funcs = [func for func in p.parser.called_funcs if
       func in registered_funcs]
    funcs_code = "\n".join(registered_funcs[func] for func in
       missing_funcs if func in registered_funcs)
    p[0] = (
        f"{p[5]}"
        f"start\n"
        f"{p[6]}"
        f"stop\n\n"
        f"{funcs_code}\n"
    )


# ---------------------------------------- Function
    ----------------------------------------

def p_Function_empty(p):
    '''Function : '''
    p[0] = ""
```

```
# ---------------------------------- Variable Declaration
    ----------------------------------

def p_VarsDeclaration(p):
    '''VarsDeclaration : VAR Declarations
                       | '''
    if len(p) == 3:
        p[0] = p[2]
    else:
        p[0] = ""


def p_Declarations(p):
    '''Declarations : Declaration SubDeclarations'''
    p[0] = p[1] + p[2]


def p_SubDeclarations(p):
    '''SubDeclarations : Declaration
                       | '''
    if len(p) == 2:
        p[0] = p[1]
    else:
        p[0] = ""


def p_Declaration_simple(p):
    '''Declaration : ID COLON Type SEMICOLON'''
    add_var(p[1], 1, p.parser.current_type, p)
    if p.parser.current_type == 0:
        p[0] = f"pushi 0\n"
    else:
        p[0] = f'pushs ""\n'


def p_Declaration(p):
    '''Declaration : VariablesList COLON Type SEMICOLON'''
    code = ""
    for var in p[1]:
        add_var(var, 1, p.parser.current_type, p)
        if p.parser.current_type == 0:
            code += f"pushi 0\n"
        else:
            code += f'pushs ""\n'
    p[0] = code
```

```
def p_Declaration_array(p):
    '''Declaration : ID COLON ARRAY LBRACKET NUMBER DOT DOT
       NUMBER RBRACKET OF INTEGER SEMICOLON'''
    size = int(p[8]) - int(p[5]) + 1
    lower_bound = int(p[5])
    add_var(p[1], size, 0, p, lower_bound)
    p[0] = f"pushn {size}\n"


def p_VariablesList(p):
    '''VariablesList : ID
                     | VariablesList COMMA ID'''
    if len(p) == 2:
        p[0] = [p[1]]
    else:
        p[0] = p[1] + [p[3]]


def p_Type(p):
    '''Type : INTEGER
            | BOOLEAN
            | STRING'''
    p.parser.current_type = 0 if p[1].lower() in ('integer', '
       boolean') else 1


# ---------------------------------------- Code Block
   ----------------------------------------

def p_CodeBlock(p):
    '''CodeBlock : BEGIN Instructions END'''
    p[0] = p[2]


# ---------------------------------------- Instructions
   ----------------------------------------

def p_Instructions(p):
    '''Instructions : Instructions Instruction SEMICOLON
                    | Instruction SEMICOLON
                    | Instruction'''
    if len(p) == 4:
        p[0] = p[1] + p[2]
    else:
        p[0] = p[1]
```

```
def p_Instruction_writeln(p):
    '''Instruction : WRITELN LPAREN Write_Exp STRING_LITERAL
       Write_Exp RPAREN'''
    p[0] = f'{p[3]}pushs "{p[4]}"\nwrites\n{p[5]}writeln\n'


def p_Instruction_write(p):
    '''Instruction : WRITE LPAREN Write_Exp STRING_LITERAL
       Write_Exp RPAREN'''
    p[0] = f'{p[3]}pushs "{p[4]}"\nwrites\n{p[5]}'


def p_Write_Exp_empty(p):
    '''Write_Exp : '''
    p[0] = ""


def p_Write_Exp_string(p):
    '''Write_Exp : COMMA String Write_Exp'''
    p[0] = f'pushs "{p[2]}"\nwrites\n{p[3]}'


def p_Write_id(p):
    '''Write_Exp : COMMA ID Write_Exp'''
    v = get_index(p[2], p)
    (_, _, offset, _) = v
    p[0] = f"pushg {offset}\nwritei\n{p[3]}"


def p_Write_var(p):
    '''Write_Exp : ID COMMA'''
    v = get_index(p[1], p)
    (_, _, offset, _) = v
    p[0] = f"pushg {offset}\nwritei\n"


def p_Instruction_readln(p):
    '''Instruction : READLN LPAREN ID RPAREN'''
    v = get_index(p[3], p)
    (_, type, offset, _) = v
    if type == 0:
        p[0] = f"read\natoi\nstoreg {offset}\nwriteln\n"
    else:
        p[0] = f"read\nstoreg {offset}\nwriteln\n"
```

```
def p_Instruction_read_array(p):
    '''Instruction : READLN LPAREN ID LBRACKET Exp RBRACKET
       RPAREN'''
    v = get_index(p[3], p)
    (_, _, offset, lower_bound) = v
    if lower_bound != 0:
        p[0] = f"pushgp\npushi {offset}\npadd\n{p[5]}pushi {
           lower_bound}\nsub\nread\natoi\nstoren\nwriteln\n"
    else:
        p[0] = f"pushgp\npushi {offset}\npadd\n{p[5]}read\natoi\
           nstoren\nwriteln\n"



def p_Instruction_assign(p):
    '''Instruction : ID ASSIGN Exp'''
    v = get_index(p[1], p)
    (_, _, offset, _) = v
    p[0] = f"{p[3]}storeg {offset}\n"


def p_Instruction_code_blocl(p):
    '''Instruction : CodeBlock'''
    p[0] = p[1]


# --- Condicionais e Ciclos ---

def p_Condition(p):
    '''Instruction : IF Log THEN Instruction ELSE Instruction'''
    current_if = p.parser.ifs
    p[0] = (f"{p[2]}"
            f"jz else{current_if}\n"
            f"{p[4]}"
            f"jump endif{current_if}\n"
            f"else{current_if}:\n"
            f"{p[6]}\n"
            f"endif{current_if}:\n"
        )
    p.parser.ifs += 1


def p_Condition_simple(p):
    '''Instruction : IF Log THEN Instruction %prec IFX'''
    current_if = p.parser.ifs
    p[0] = (f"{p[2]}"
            f"jz endif{current_if}\n"
```

```
            f"{p[4]}"
            f"endif{current_if}:\n"
        )
    p.parser.ifs += 1


def p_Cycle_While(p):
    '''Instruction : WHILE Log DO Instruction'''
    p[0] = (
        f"while{p.parser.cycles}:\n"
        f"{p[2]}"
        f"jz endwhile{p.parser.cycles}\n"
        f"{p[4]}"
        f"jump while{p.parser.cycles}\n"
        f"endwhile{p.parser.cycles}:\n"
    )
    p.parser.cycles += 1


def p_Cycle_For_Do(p):
    '''Instruction : FOR ID ASSIGN Exp TO Exp DO Instruction'''
    v = get_index(p[2], p)
    (_, _, offset, _) = v

    p[0] = (
        f"{p[4]}storeg {offset}\n"
        f"while{p.parser.cycles}:\n"
        f"pushg {offset}\n"
        f"{p[6]}infeq\n"
        f"jz endwhile{p.parser.cycles}\n"
        f"{p[8]}"
        f"pushg {offset}\npushi 1\nadd\nstoreg {offset}\n"
        f"jump while{p.parser.cycles}\n"
        f"endwhile{p.parser.cycles}:\n"
    )
    p.parser.cycles += 1


def p_Cycle_For_Downto(p):
    '''Instruction : FOR ID ASSIGN Exp DOWNTO Exp DO Instruction
                   | FOR ID ASSIGN ID LPAREN ID RPAREN DOWNTO
                      NUMBER DO Instruction'''
    v = get_index(p[2], p)
    (_, _, offset, _) = v

    if len(p) == 10:
        p[0] = (
```

```
            f"{p[4]}storeg {offset}\n"
            f"while{p.parser.cycles}:\n"
            f"pushg {offset}\n"
            f"{p[6]}supeq\n"
            f"jz endwhile{p.parser.cycles}\n"
            f"{p[8]}"
            f"pushg {offset}\n"
            f"storeg {offset}\n"
            f"jump while{p.parser.cycles}\n"
            f"endwhile{p.parser.cycles}:\n"
        )
    else:
        p.parser.called_funcs.add(p[4])
        if p[4] in p.parser.funcs:
            res = get_index(p[6], p)
            (_, _, offset1, _) = res
            p[0] = (
                f"pushg {offset1}\n"
                f"pusha {p[4]}\n"
                f"call\n"
                f"pushi 1\n"
                f"sub\n"
                f"storeg {offset}\n"
                f"while{p.parser.cycles}:\n"
                f"pushg {offset}\n"
                f"pushi {p[9]}\n"
                f"pushi 1\n"
                f"sub\n"
                f"supeq\n"
                f"jz endwhile{p.parser.cycles}\n"
                f"{p[11]}"
                f"pushg {offset}\n"
                f"pushi 1\n"
                f"sub\n"
                f"storeg {offset}\n"
                f"jump while{p.parser.cycles}\n"
                f"endwhile{p.parser.cycles}:\n"
            )
    p.parser.cycles += 1


# ---------------------------------------- Expressions
    ----------------------------------------

def p_Log_and(p):
    '''Log : Log AND Rel'''
    p[0] = f"{p[1]}{p[3]}and\n"
```

```
def p_Log_or(p):
    '''Log : Log OR Rel'''
    p[0] = f"{p[1]}{p[3]}or\n"


def p_Log_not(p):
    '''Log : NOT Log'''
    p[0] = f"{p[2]}not\n"


def p_Log_rel(p):
    '''Log : Rel
           | LPAREN Log RPAREN'''
    if len(p) == 2:
        p[0] = p[1]
    else:
        p[0] = p[2]


def p_Rel_less(p):
    '''Rel : Rel LESS Exp'''
    p[0] = f"{p[1]}{p[3]}inf\n"


def p_Rel_lessequal(p):
    '''Rel : Rel LESSEQUAL Exp'''
    p[0] = f"{p[1]}{p[3]}infeq\n"


def p_Rel_greater(p):
    '''Rel : Rel GREATER Exp'''
    p[0] = f"{p[1]}{p[3]}sup\n"


def p_Rel_greaterequal(p):
    '''Rel : Rel GREATEREQUAL Exp'''
    p[0] = f"{p[1]}{p[3]}supeq\n"


def p_Rel_equals(p):
    '''Rel : Rel EQUAL Exp'''
    p[0] = f"{p[1]}{p[3]}equal\n"


def p_Rel_exp(p):
```

```
    '''Rel : Exp
           | LPAREN Rel RPAREN'''
    if len(p) == 2:
        p[0] = p[1]
    else:
        p[0] = p[2]


def p_Exp_add(p):
    '''Exp : Exp PLUS Term'''
    p[0] = f"{p[1]}{p[3]}add\n"


def p_Exp_sub(p):
    '''Exp : Exp MINUS Term'''
    p[0] = f"{p[1]}{p[3]}sub\n"


def p_Exp_term(p):
    '''Exp : Term'''
    p[0] = p[1]


def p_Term_mult(p):
    '''Term : Term TIMES Factor'''
    p[0] = f"{p[1]}{p[3]}mul\n"


def p_Term_div(p):
    '''Term : Term DIVIDE Factor'''
    f"{p[1]}{p[3]}div\n"


def p_Term_mod(p):
    '''Term : Term MOD Factor'''
    p[0] = f"{p[1]}{p[3]}mod\n"


def p_Term_divide(p):
    '''Term : Term DIV Factor'''
    p[0] = f"{p[1]}{p[3]}div\n"


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

```
def p_Factor_true(p):
    '''Factor : TRUE'''
    p[0] = "pushi 1\n"


def p_Factor_false(p):
    '''Factor : FALSE'''
    p[0] = "pushi 0\n"


def p_Factor_num(p):
    '''Factor : NUMBER'''
    p[0] = f"pushi {p[1]}\n"


def p_Factor_id(p):
    '''Factor : ID'''
    res = get_index(p[1], p)
    (_, _, offset, _) = res
    p[0] = f"pushg {offset}\n"


def p_Factor_paren(p):
    '''Factor : LPAREN Exp RPAREN'''
    p[0] = p[2]


def p_Factor_func(p):
    '''Factor : ID LPAREN ID RPAREN'''
    p.parser.called_funcs.add(p[1])
    if p[1] in p.parser.funcs:
        res = get_index(p[3], p)
        (_, _, offset, _) = res
        p[0] = f"pushg {offset}\npusha {p[1]}\ncall\n\n"
    else:
        raise Exception(f"Function {p[1]} not declared")


def p_Factor_string(p):
    '''Factor : STRING_LITERAL'''
    p[0] = f'pushs "{p[1]}"\npushi 0\ncharat\n'


def p_Factor_array(p):
    '''Factor : ID LBRACKET Exp RBRACKET'''
    res = get_index(p[1], p)
```

```
    (_, type, offset, lower_bound) = res
    if type == 0:
        if lower_bound != 0:
            p[0] = f"pushgp\npushi {offset}\npadd\n{p[3]}pushi {
               lower_bound}\nsub\nloadn\n"
        else:
            p[0] = f"pushgp\npushi {offset}\npadd\n{p[3]}loadn\n
               "
    else:
        p[0] = f"pushg {offset}\n{p[3]}charat\n"


# ----------------------------------------- String
   -----------------------------------------

def p_String(p):
    '''String : STRING_LITERAL'''
    p[0] = p[1]


# ----------------------------------------- Others
   -----------------------------------------

def p_error(p):
    if p:
        print(f"Erro de sintaxe na linha {p.lineno}, token '{p.
           value}' (tipo: {p.type})")
        # Mostra os últimos 5 tokens processados
        print("Contexto:", parser.symstack[-5:])
    else:
        print("Erro de sintaxe no final do arquivo")


def add_var(id, size, type, p, lower_bound = 0):
    if id not in p.parser.var.keys():
        p.parser.var[id] = (size, type, p.parser.offset,
           lower_bound)
        #print(f"Adding variable {id} with size {size} at offset
            {p.parser.offset}")
        p.parser.offset += size
    else:
        raise Exception


def get_index(id, p):
    if id in p.parser.var.keys():
        return p.parser.var[id]
```

```
# ------------------------------------------ Run
    ------------------------------------------

r = 1
while r:
    inFileFolder = "input/"
    inFile = input("Code File >> ")
    inFilePath = os.path.join(inFileFolder, inFile)

    try:
        fileIn = open(inFilePath, "r")
        r = 0
    except (FileNotFoundError, NotADirectoryError):
        print("Inexistent file!\n\nAvailable files:")
        files = os.listdir(inFileFolder)
        files.sort()
        for file in files:
            print(file)


r = 1
while r:
    outFileFolder = "output/"
    outFilePath = os.path.join(outFileFolder, inFile) + ".vm"

    if outFilePath != inFilePath:
        try:
            fileOut = open(outFilePath, "w")
            r = 0
        except (FileNotFoundError, NotADirectoryError):
            print("Wrong File Path\n")
    else:
        print("Wrong File Path\n")

parser = yacc.yacc(debug=True)

parser.var = dict()
parser.funcs = set()
parser.called_funcs = set()
parser.offset = 0
parser.ifs = 0
parser.cycles = 0
parser.current_type = None
```

```
parser.funcs.add("length")
registered_funcs = {
    'length': "length:\npushfp\nload -1\nstrlen\nreturn"
}


data = fileIn.read()

try:
    result = parser.parse(data)
    fileOut.write(result)
except (TypeError, Exception):
    print("\nCompilation error, aborting!")
    os.remove(outFilePath)

fileIn.close()
fileOut.close()
```