Processamento de Linguagens (3º ano de LEI)

Trabalho Prático

Relatório de Desenvolvimento

Diogo Neto	Guilherme Oliveira	Pedro Pacheco
(a98197)	(A95021)	(A61042)

1 de junho de 2025

Resumo

O presente relatório técnico tem como objetivo detalhar o processo de desenvolvimento de um compilador de linguagem pascal standard no âmbito do projeto semestral da unidade curricular. O compilador desenvolvido recorre aos módulos lex e yacc da biblioteca ply do python e, é expectável que a gramática construída gere código assembly para a máquina virtual disponibilizada.

Conteúdo

Capítulo 1

Introdução

A construção de compiladores é uma das áreas fundamentais no estudo do Processamento de Linguagens e fornece uma aplicação prática das técnicas formais de análise léxica, sintática e semântica. No âmbito da unidade curricular de Processamento de Linguagens, foi proposto o desenvolvimento de um compilador da linguagem Pascal, com vista à consolidação dos conhecimentos teóricos e práticos adquiridos ao longo do semestre.

O projeto visa implementar um compilador capaz de reconhecer programas válidos escritos nessa linguagem e gerar código assembly para a máquina virtual EWVM, disponibilizada pelos docentes da unidade.

O desenvolvimento do projeto foi realizado em Python, com recurso à biblioteca PLY, que permite a criação de analisadores léxicos e sintáticos com uma sintaxe própria utilizando as ferramentas lex e yacc.

As secções seguintes detalham a implementação técnica do compilador, desde a definição da gramática até à produção de código, incluindo as metodologias adotadas e as decisões de projeto mais relevantes.

Capítulo 2

Problema Proposto

O principal objetivo do projeto proposto é permitir a aplicação prática dos conceitos teóricos relacionados com as várias fases de compilação: análise léxica, sintática, semântica e geração de código. Pretende-se que a linguagem gerada para a VM suporte os seguintes elementos fundamentais:

- Declaração de variáveis atómicas do tipo `integer`, `boolean` e `string`, com suporte para operações aritméticas (`+`, `-`, `*`, `div`, `mod`), relacionais (`=`, `<>`, `<`, `<=`, `>`, `>=`) e lógicas (`and`, `or`).
- Instruções algorítmicas básicas, nomeadamente atribuições de expressões a variáveis e leitura/escrita através das instruções `readln` e `writeln`, respetivamente.
- Instruções de controlo de fluxo do tipo `if-then[-else]`, permitindo decisões condicionais com base em expressões booleanas.
- Instruções de repetição, nomeadamente o ciclo `for-do`.

O código gerado deve preservar a semântica dos programas originais, assegurando a correta execução das instruções especificadas no destino (EWVM).

Capítulo 3

Concepção da Resolução

3.1 Organização e Estrutura

A implementação do nosso compilador foi organizada de forma modular, sendo dividida em quatro componentes principais de modo a refletir as diferentes fases do processo de compilação, onde cada componente desempenha um papel específico na tradução de código, sendo elas:

- **Construção da Gramática Independente de Contexto**
  A primeira etapa consistiu na definição formal da gramática da linguagem e cobre as principais construções do pacal, aqui foram consideradas produções para declarações de variáveis, funções, expressões aritméticas e lógicas, estruturas de controlo como `if`, `while`, `for`, e chamadas de função. Esta gramática serve como base para o funcionamento do parser.

- **Construção do analisador léxico (lexer)**
  O módulo lexer tem como objetivo identificar os tokens válidos da linguagem, como palavras-chave (`if`, `for`, `function`), identificadores, operadores, literais, símbolos de pontuação e, para isso, foram utilizadas expressões regulares associadas a regras de tratamento de casos especiais, como literais de string e comentários.

- **Construção do analisador sintático (parser)**
  O analisador sintático foi desenvolvido para reconhecer sequências válidas de tokens, conforme a gramática especificada, o parser implementa regras de produção que correspondem a instruções e blocos da linguagem. Além da verificação sintática, cada regra é responsável pela geração de código específico, recorrendo a instruções da VM como `PUSHI`, `STOREG`, `CALL`, entre outras.

- **Conversão para código assembly da VM**
  A última fase consiste na geração do código de saída, que corresponde ao código assembly da máquina virtual onde, este código, é diretamente executável por um interpretador de VM e inclui instruções para as operações descritas anteriormente. O código resultante é construído dinamicamente durante a análise sintática, sendo impresso após a conclusão da análise.

Todas as funcionalidades implementadas e descritas neste capítulo encontram-se detalhadas no **Anexo A** deste documento.

3.2 GIC

A construção do código segue as seguintes produções, com respeito rigoroso às prioridades dos operadores:

```
Programa             : PROGRAM ID SEMICOLON Block DOT
Block                : Declarations CompoundStatement
Declarations         : VarDeclarationPart FunctionDeclarations
VarDeclarationPart   : VAR VarDeclarationList
                     | empty
VarDeclarationList   : VarDeclaration
                     | VarDeclarationList VarDeclaration
VarDeclaration       : IdList COLON Type SEMICOLON
IdList               : ID
                     | ID COMMA IdList
Type                 : BasicType
                     | ArrayType
BasicType            : INTEGER
                     | STRING
                     | BOOLEAN
ArrayType            : ARRAY LBRACKET NUMBER DOTDOT NUMBER RBRACKET OF BasicType
FunctionDeclarations : FunctionDeclaration FunctionDeclarations
                     | empty
FunctionDeclaration  : FUNCTION ID LPAREN FormalParameters RPAREN COLON BasicType SEMICOLO
FormalParameters     : ID COLON Type
                     | ID COLON Type SEMICOLON FormalParameters
                     | empty
CompoundStatement    : BEGIN StatementList END
StatementList        : Statement
                     | StatementList SEMICOLON Statement
Statement            : AssignmentStatement
                     | WriteStatement
                     | ReadStatement
                     | IfStatement
                     | WhileStatement
                     | ForStatement
                     | CompoundStatement
                     | empty
ReadStatement        : READLN LPAREN ArrayAccess RPAREN
                     | READLN LPAREN ID RPAREN
AssignmentStatement  : ID ASSIGN Expression
                     | ArrayAccess ASSIGN Expression
WriteStatement       : WRITELN LPAREN ExpressionList RPAREN
ExpressionList       : Expression
                     | ExpressionList COMMA Expression
```

```
Expression           : ExprBool
ExprBool             : Expr
                     | Expr OpRel Expr
OpRel                : EQ | NEQ | LT | LTE | GT | GTE
Expr                 : Termo
                     | Expr OpAd Termo
OpAd                 : PLUS | MINUS | OR
Termo                : Fator
                     | Termo OpMul Fator
OpMul                : TIMES | DIV | MOD | AND
Fator                : Const
                     | Var
                     | LPAREN ExprBool RPAREN
                     | Expression_function_call
                     | NOT Fator
Const                : NUMBER | STRING_LITERAL | TRUE | FALSE
Var                  : ID
                     | ID LBRACKET ExprBool RBRACKET
Expression_function_call : ID LPAREN ActualParameters RPAREN
                         | LENGTH LPAREN ExprBool RPAREN
ActualParameters     : ExprBool
                     | ExprBool COMMA ActualParameters
                     | empty
IfStatement          : IF Expression THEN Statement
                     | IF Expression THEN Statement ELSE Statement
WhileStatement       : WHILE Expression DO Statement
ForStatement         : FOR ID ASSIGN Expression TO Expression DO Statement
ArrayAccess          : ID LBRACKET Expression RBRACKET
empty                : (vazio)
```

3.3 Lexer

O analisador léxico é responsável por identificar os símbolos terminais (tokens) dos programas escritos em Pascal, recorrendo a expressões regulares associadas diretamente a cada token, como preconizado na abordagem formal lecionada.

Ao invés de uma abordagem baseada em palavras reservadas armazenadas num dicionário (como sugerida por ferramentas como o CoPilot ou abordagens automáticas baseadas em *ReservedKeywords*), optou-se pela separação explícita de cada palavra-chave e operador como uma função individual ou expressão regular. Essa opção evita a sobrecarga desnecessária de lógica no reconhecimento e preserva a clareza e extensibilidade da definição léxica.

As definições de tokens implementadas são as seguintes:

```
PROGRAM        : 'program'
VAR            : 'var'
```

```
BEGIN          : 'begin'
END            : 'end'
IF             : 'if'
THEN           : 'then'
ELSE           : 'else'
WHILE          : 'while'
DO             : 'do'
FOR            : 'for'
TO             : 'to'
FUNCTION       : 'function'
DIV            : 'div'
MOD            : 'mod'
LENGTH         : 'length'
INTEGER        : 'integer'
STRING         : 'string'
BOOLEAN        : 'boolean'
TRUE           : 'true'
FALSE          : 'false'
WRITELN        : 'writeln'
WRITE          : 'write'
READLN         : 'readln'
READ           : 'read'
ARRAY          : 'array'
OF             : 'of'
AND            : 'and'
OR             : 'or'
NOT            : 'not'
PLUS           : '+'
MINUS          : '-'
TIMES          : '*'
ASSIGN         : ':='
EQ             : '='
NEQ            : '<>'
GT             : '>'
LT             : '<'
GTE            : '>='
LTE            : '<='
COMMA          : ','
LPAREN         : '('
RPAREN         : ')'
LBRACKET       : '['
RBRACKET       : ']'
SEMICOLON      : ';'
DOT            : '.'
COLON          : ':'
```

```
DOTDOT         : '..'
STRING_LITERAL : '\'' [^']* '\''
NUMBER         : \d+(\.\d+)?
ID             : [a-zA-Z_][a-zA-Z0-9_]*
```

Todos os comentários, delimitados por `{...}` ou `(**)`, são descartados silenciosamente. O analisador distingue literais, operadores e identificadores sem recorrer a pós-processamento de `ID` para resolução de palavras reservadas, cumprindo rigorosamente o paradigma baseado exclusivamente em expressões regulares.

Cada token reconhecido é posteriormente passado ao analisador sintático (*parser*) para validação das construções gramaticais conforme a linguagem Pascal.

3.4 Parser e conversão para assembly VM

O analisador sintático ou parser, é o responsável por verificar se o código pascal está escrito corretamente, isto é, se o código respeita as regras gramaticais definidas e, não havendo erros sintáticos, o parser converte o código pascal no assembly de acordo com as instruções.

3.4.1 Geração de assembly – Exemplo: Cálculo do Fatorial

Através do exemplo do programa `Fatorial`, ilustramos a geração de código efetuada pelo compilador para a máquina virtual, este programa cobre instruções relacionadas com inicialização de variáveis, ciclos, entrada e saída, entre outras, o que serve para termos uma visão geral de como funcionam as instruções da VM, assim como é feita a tradução e associação das instruções pascal para as da VM.

Início do Programa

```
START
```

Marca o início da execução do programa.

Inicialização de Variáveis Globais

Cada variável declarada globalmente é inicializada a 0:

```
PUSHI 0
STOREG <offset>
```

Por exemplo, para:

```
var n, i, fat: integer;
```

o compilador gera:

```
PUSHI 0   ; n
STOREG 0
PUSHI 0   ; i
STOREG 1
PUSHI 0   ; fat
STOREG 2
```

Escrita de Texto

```
PUSHS "mensagem"
WRITES
```

Corresponde à instrução:

```
writeln('Introduza um número inteiro positivo:');
```

Leitura de Valores (Inteiros)

```
READ
ATOI
STOREG <offset>
```

Para:

```
readln(n);
```

gera-se:

```
READ
ATOI
STOREG 0
```

Atribuições Simples

```
PUSHI <valor>
STOREG <offset>
```

Exemplo:

```
fat := 1;
```

gera:

```
PUSHI 1
STOREG 2
```

Ciclo `for`

O compilador traduz ciclos do tipo:

```
for i := 1 to n do fat := fat * i;
```

da seguinte forma:

```
PUSHI 1
STOREG 1          ; i := 1

L1:
PUSHG 1           ; i
PUSHG 0           ; n
INFEQ
JZ L2
```

```
PUSHG 2          ; fat
PUSHG 1          ; i
MUL
STOREG 2         ; fat := fat * i

PUSHG 1
PUSHI 1
ADD
STOREG 1         ; i := i + 1

JUMP L1
L2:
```

Escrita de Múltiplos Valores

```
PUSHS "texto"
WRITES
PUSHG <offset>
WRITEI
```

Exemplo:

```
writeln('Fatorial de ', n, ': ', fat);
```

gera:

```
PUSHS "Fatorial de "
WRITES
PUSHG 0
WRITEI
PUSHS ": "
WRITES
PUSHG 2
WRITEI
```

Término do Programa

```
STOP
```

Assinala o fim do programa e termina a execução na VM.
Este exemplo demonstra como o compilador lida com as construções da linguagem pascal e as traduz eficientemente em instruções da máquina virtual, respeitando a semântica original do programa.

Capítulo 4

Demonstração

4.1 Execução do programa

Para inicializar o compilador, o programa desenvolvimento suporta três modos:

- python yacc.py –example MaiorDe3

  O programa permite a execução dos seguintes programas pascal pré definidos no sistema: MaiorDe3, Fatorial, NumeroPrimo, SomaArray, HelloWorld

- python yacc.py –file tests/LargestOf3.pas

  Neste modo o compilador recebe programas .pas escritos pelo utilizador.

- python yacc.py

  Neste modo o programa lê do STDIN até ao EOF (end of file - ctrl+D)

O resultado dos três modos é sempre apresentado no STDOUT e pode ser validado na VM
Durante o decorrer deste capítulo serão apresentados os porgramas mencionados no primeiro modo apresentado bem como os seus respetivos outputs devidamente testados na VM.

4.2 Teste Hello World

O programa imprime uma string "Ola, Mundo!".

4.2.1 Código pascal - Hello World

```
program HelloWorld;
   begin
   writeln('Ola, Mundo!');
end.
```

4.2.2 Código assembly - Hello World

```
START
PUSHS "Ola, Mundo!"
WRITES
STOP
```

4.3 Teste Maiorde3

O programa calcula o maior de 3 dígitos e imprime o resultado

4.3.1 Código pascal - maiorde3

```
program Maior3;
    var
        num1, num2, num3, maior: integer;
    begin
        { Ler 3 números }
        writeln('Introduza o primeiro número: ');
        readln(num1);
        writeln('Introduza o segundo número: ');
        readln(num2);
        writeln('Introduza o terceiro número: ');
        readln(num3);
        if num1 > num2 then
            if num1 > num3 then
                maior := num1
            else maior := num3
        elseif num2 > num3
            then maior := num2
        else
            maior := num3;
        writeln(maior);
    end.
```

4.3.2 Código assembly gerado - maiorde3

```
START
PUSHI 0
STOREG 0
PUSHI 0
STOREG 1
PUSHI 0
STOREG 2
```

```
PUSHI 0
STOREG 3
PUSHS "Introduza o primeiro número: "
WRITES
READ
ATOI
STOREG 0
PUSHS "Introduza o segundo número: "
WRITES
READ
ATOI
STOREG 1
PUSHS "Introduza o terceiro número: "
WRITES
READ
ATOI
STOREG 2
PUSHG 0
PUSHG 1
SUP
JZ L5
PUSHG 0
PUSHG 2
SUP
JZ L1
PUSHG 0
STOREG 3
JUMP L2
L1:
PUSHG 2
STOREG 3
L2:
JUMP L6
L5:
PUSHG 1
PUSHG 2
SUP
JZ L3
PUSHG 1
STOREG 3
JUMP L4
L3:
PUSHG 2
STOREG 3
L4:
```

```
L6:
PUSHG 3
WRITEI
STOP
```

4.4 Teste fatorial

O programa calcula o fatorial de um número.

4.4.1 Código pascal - fatorial

```
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.
```

4.4.2 Código assembly gerado - fatorial

```
   START
PUSHI 0
STOREG 0
PUSHI 0
STOREG 1
PUSHI 0
STOREG 2
PUSHS "Introduza um número inteiro positivo:"
WRITES
READ
ATOI
STOREG 0
PUSHI 1
STOREG 2
PUSHI 1
STOREG 1
L1:
PUSHG 1
```

```
PUSHG 0
INFEQ
JZ L2
PUSHG 2
PUSHG 1
MUL
STOREG 2
PUSHG 1
PUSHI 1
ADD
STOREG 1
JUMP L1
L2:
PUSHS "Fatorial de "
WRITES
PUSHG 0
WRITEI
PUSHS ": "
WRITES
PUSHG 2
WRITEI
STOP
```

4.5 Teste NumeroPrimo

O programa verifica se um número é par ou ímpar.

4.5.1 Código pascal - NumeroPrimo

```
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.
```

4.5.2 Código assembly gerado - NumerpPrimo

```
 START
PUSHI 0
STOREG 0
PUSHI 0
STOREG 1
PUSHI 0
STOREG 2
PUSHS "Introduza um número inteiro positivo:"
WRITES
READ
ATOI
STOREG 0
PUSHI 1
STOREG 2
PUSHI 2
STOREG 1
L2:
PUSHG 1
PUSHG 0
PUSHI 2
DIV
INFEQ
PUSHG 2
ADD
PUSHI 2
EQUAL
JZ L3
PUSHG 0
PUSHG 1
MOD
PUSHI 0
EQUAL
JZ L1
PUSHI 0
STOREG 2
L1:
PUSHG 1
```

```
PUSHI 1
ADD
STOREG 1
JUMP L2
L3:
PUSHG 2
JZ L4
PUSHG 0
WRITEI
PUSHS " é um número primo"
WRITES
JUMP L5
L4:
PUSHG 0
WRITEI
PUSHS " não é um número primo"
WRITES
L5:
STOP
```

4.6 Teste SomaArray

O programa soma todos os elementos de um array

4.6.1 Código pascal - SomaArray

```
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
            writeln(i);
            readln(numeros[i]);
            soma := soma + numeros[i];
        end;
        writeln('A soma dos números é: ', soma);
    end.
```

4.6.2 Código assembly gerado - SomaArray

```
   START
PUSHI 0
STOREG 0
PUSHI 0
STOREG 1
PUSHI 0
STOREG 2
PUSHI 0
STOREG 3
PUSHI 0
STOREG 4
PUSHI 0
STOREG 5
PUSHI 0
STOREG 6
PUSHI 0
STOREG 6
PUSHS "Introduza 5 números inteiros:"
WRITES
PUSHI 1
STOREG 5
L1:
PUSHG 5
PUSHI 5
INFEQ
JZ L2
PUSHGP
PUSHG 5
PUSHI 1
SUB
PUSHI 0
ADD
PADD
READ
ATOI
STORE 0
PUSHG 6
PUSHGP
PUSHG 5
PUSHI 1
SUB
PUSHI 0
ADD
```

```
PADD
LOAD 0
ADD
STOREG 6
PUSHG 5
PUSHI 1
ADD
STOREG 5
JUMP L1
L2:
PUSHS "A soma dos números é: "
WRITES
PUSHG 6
WRITEI
STOP
```

Capítulo 5

Conclusão

No decorrer do projeto tentamos sempre seguir os métodos de construção de gramáticas aprendidos em sala de aula, o que nos foi bastante útil e permitiu também consolidar o que viemos a aprender durante o semestre. No que diz respeito ao resultado final do projeto, de acordo com o enunciado do mesmo, consideramos que atingimos um bom resultado no que toca à interpretação de funcionalidades simples como operações lógicas, operações aritméticas, variáveis, ciclos e até arrays porém, reconhecemos que para um trabalho mais completo, seria bom termos implementado subprogramas como procedure e function, algo que seria implementado posteriormente como trabalho futuro.

Apêndice A

Código do Programa

Listing A.1: Ficheiro yacc.py

```
import ply.lex as lex

tokens = [
    'PROGRAM', 'VAR', 'BEGIN', 'END',
    # ciclos e condicionais
    'IF', 'THEN', 'ELSE', 'WHILE', 'DO', 'FOR', 'TO',
    # fun   es
    'FUNCTION', 'DIV', 'MOD', 'LENGTH',
    # tipos
    'INTEGER', 'STRING', 'BOOLEAN',
    # bools
    'TRUE', 'FALSE',
    # in&out
    'WRITELN', 'WRITE', 'READLN', 'READ',
    # aritmeticos e relacionais
    'PLUS', 'MINUS', 'TIMES', 'ASSIGN',
    'GT', 'LT', 'EQ', 'NEQ', 'GTE', 'LTE',
    # literals
    'SEMICOLON', 'DOT', 'COLON', 'COMMA', 'LPAREN', 'RPAREN', 'LBRACKET',
       'RBRACKET', 'DOTDOT', 'STRING_LITERAL',
    # l gicos
    'AND', 'OR',
    'ARRAY', 'OF',
    'ID', 'NUMBER'
]

t_ignore = ' \r\n\t'
# PR
keywords = {
    'program': 'PROGRAM',
    'var': 'VAR',
    'begin': 'BEGIN',
```

```
    'end': 'END',
    'if': 'IF',
    'then': 'THEN',
    'else': 'ELSE',
    'while': 'WHILE',
    'do': 'DO',
    'for': 'FOR',
    'to': 'TO',
    'function': 'FUNCTION',
    'div': 'DIV',
    'mod': 'MOD',
    'integer': 'INTEGER',
    'string': 'STRING',
    'boolean': 'BOOLEAN',
    'true': 'TRUE',
    'false': 'FALSE',
    'writeln': 'WRITELN',
    'write': 'WRITE',
    'readln': 'READLN',
    'read': 'READ',
    'array': 'ARRAY',
    'of': 'OF',
    'and': 'AND',
    'length': 'LENGTH'

}
t_PLUS = r'\+'
t_MINUS = r'-'
t_TIMES = r'\*'
t_ASSIGN = r':='
t_EQ = r'='
t_NEQ = r'<>'
t_GT = r'>'
t_LT = r'<'
t_GTE = r'>='
t_LTE = r'<='
t_COMMA = r','
t_LPAREN = r'\('
t_RPAREN = r'\)'
t_LBRACKET = r'\['
t_RBRACKET = r'\]'
t_SEMICOLON = r';'
t_DOT = r'\.'
t_COLON = r':'
t_DOTDOT = r'\.\.'
t_OR = r'or'
t_AND = r'and'

```

```
80 def t_STRING_LITERAL(t):
81     r"'[^']*'"
82     t.value = t.value[1:-1]
83     return t
84 
85 def t_MOD(t):
86     r'mod'
87     t.type="MOD"
88     return t
89 
90 def t_DIV(t):
91     r'div'
92     t.type="DIV"
93     return t
94 
95 def t_COMMENT(t):
96     r'(\{.*?\})|(\(\*.*?\*\))'
97     pass
98 
99 def t_NUMBER(t):
100     r'\d+\.\d+|\d+'
101     if '.' in t.value:
102         t.value = float(t.value)
103     else:
104         t.value = int(t.value)
105     return t
106 
107 def t_ARRAY(t):
108     r'array'
109     t.type="ARRAY"
110     return t
111 
112 def t_OF(t):
113     r'of'
114     t.type="OF"
115     return t
116 
117 def t_PROGRAM(t):
118     r'program'
119     t.type = 'PROGRAM'
120     return t
121 
122 def t_VAR(t):
123     r'var'
124     t.type = 'VAR'
125     return t
126 
127 def t_BEGIN(t):
```

```
128     r'begin'
129     t.type = 'BEGIN'
130     return t
131
132 def t_END(t):
133     r'end'
134     t.type = 'END'
135     return t
136
137 def t_IF(t):
138     r'if'
139     t.type = 'IF'
140     return t
141
142 def t_THEN(t):
143     r'then'
144     t.type = 'THEN'
145     return t
146
147 def t_ELSE(t):
148     r'else'
149     t.type = 'ELSE'
150     return t
151
152 def t_WHILE(t):
153     r'while'
154     t.type = 'WHILE'
155     return t
156
157 def t_DO(t):
158     r'do'
159     t.type = 'DO'
160     return t
161
162 def t_FOR(t):
163     r'for'
164     t.type = 'FOR'
165     return t
166
167 def t_TO(t):
168     r'to'
169     t.type = 'TO'
170     return t
171
172 def t_FUNCTION(t):
173     r'function'
174     t.type = 'FUNCTION'
175     return t
```

```
176
177 def t_INTEGER(t):
178     r'integer'
179     t.type = 'INTEGER'
180     return t
181
182 def t_STRING(t):
183     r'string'
184     t.type = 'STRING'
185     return t
186
187 def t_BOOLEAN(t):
188     r'boolean'
189     t.type = 'BOOLEAN'
190     return t
191
192 def t_TRUE(t):
193     r'true'
194     t.type = 'TRUE'
195     return t
196
197 def t_FALSE(t):
198     r'false'
199     t.type = 'FALSE'
200     return t
201
202 def t_WRITELN(t):
203     r'writeln'
204     t.type = 'WRITELN'
205     return t
206
207 def t_WRITE(t):
208     r'write'
209     t.type = 'WRITE'
210     return t
211
212 def t_READLN(t):
213     r'readln'
214     t.type = 'READLN'
215     return t
216
217 def t_READ(t):
218     r'read'
219     t.type = 'READ'
220     return t
221
222 def t_ID(t):
223     r'[a-zA-Z_][a-zA-Z0-9_]*'
```

```
224     t.value = t.value.lower()
225     t.type = keywords.get(t.value, 'ID')
226     return t
227 
228 def t_error(t):
229     print(f'Illegal character: {t.value[0]}')
230     t.lexer.skip(1)
231 
232 lexer = lex.lex()
```

Listing A.2: Ficheiro lex.py

```
1  import ply.yacc as yacc
2  from lex import tokens
3  import sys
4  import argparse
5  
6  symbol_table_stack = [{}] # stack vars
7  current_scope_level = 0 # distingue global e local
8  stack_pointer = 0
9  local_offset = 0
10 label_counter = 0
11 current_function = None
12 function_table = {} # simbols de fun   es
13 
14 def new_label():
15     global label_counter
16     label_counter += 1
17     return f"L{label_counter}"
18 
19 def add_symbol(name, sym_type, is_array=False, array_range=None, is_local=
       False, offset=None):
20     global stack_pointer, local_offset
21     name = name.lower()
22     if name in symbol_table_stack[current_scope_level]:
23         return None
24     symbol = {'type': sym_type, 'is_array': is_array}
25     if is_local:
26         if is_array:
27             print("Error: Arrays not supported as local variables")
28             return None
29         symbol['offset'] = local_offset
30         symbol['is_local'] = True
31         local_offset += 1
32     else:
33         if is_array:
34             low, high = array_range
35             size = high - low + 1
```

```
36              symbol['range'] = array_range
37              symbol['size'] = size
38              symbol['offset'] = stack_pointer
39              stack_pointer += size
40          else:
41              symbol['offset'] = stack_pointer if offset is None else offset
42              if offset is None:
43                  stack_pointer += 1
44      symbol_table_stack[current_scope_level][name] = symbol
45      return symbol
46
47  def get_symbol(name):
48      name = name.lower()
49      for scope in reversed(symbol_table_stack):
50          if name in scope:
51              return scope[name]
52      return None
53
54  def p_Program(p):
55      '''Program : PROGRAM ID SEMICOLON Block DOT'''
56      p[0] = f"START\n{p[4]}STOP\n"
57
58  def p_Block(p):
59      '''Block : Declarations CompoundStatement'''
60      p[0] = f"{p[1]}{p[2]}"
61
62  def p_Declarations(p):
63      '''Declarations : VarDeclarationPart FunctionDeclarations'''
64      p[0] = f"{p[1]}{p[2]}"
65
66  def p_VarDeclarationPart(p):
67      '''VarDeclarationPart : VAR VarDeclarationList
68                            | empty'''
69      p[0] = p[2] if len(p) == 3 else ""
70
71  def p_VarDeclarationList(p):
72      '''VarDeclarationList : VarDeclaration
73                            | VarDeclarationList VarDeclaration'''
74      p[0] = p[1] if len(p) == 2 else f"{p[1]}{p[2]}"
75
76  def p_VarDeclaration(p):
77      '''VarDeclaration : IdList COLON Type SEMICOLON'''
78      ids = p[1]
79      var_type = p[3]
80      code = ""
81      for var_name in ids:
82          if current_function is not None:
83              if isinstance(var_type, tuple) and var_type[0] == 'array':
```

```
84                  print("Error: Arrays not supported as local variables")
85                  p[0] = ""
86                  return
87              add_symbol(var_name, var_type, is_local=True)
88          else:
89              if isinstance(var_type, tuple) and var_type[0] == 'array':
90                  base_type = var_type[1]
91                  array_range = var_type[2]
92                  symbol = add_symbol(var_name, base_type, is_array=True,
                        array_range=array_range)
93                  if symbol:
94                      for i in range(array_range[1] - array_range[0] + 1):
95                          code += f"PUSHI 0\nSTOREG {symbol['offset'] + i}\n
                                "
96              else:
97                  symbol = add_symbol(var_name, var_type)
98                  if symbol:
99                      code += f"PUSHI 0\nSTOREG {symbol['offset']}\n"
100     p[0] = code
101
102 def p_IdList(p):
103     '''IdList : ID
104               | ID COMMA IdList'''
105     p[0] = [p[1]] if len(p) == 2 else [p[1]] + p[3]
106
107 def p_Type(p):
108     '''Type : BasicType
109             | ArrayType'''
110     p[0] = p[1]
111
112 def p_BasicType(p):
113     '''BasicType : INTEGER
114                  | STRING
115                  | BOOLEAN'''
116     p[0] = p[1].lower()
117
118 def p_ArrayType(p):
119     '''ArrayType : ARRAY LBRACKET NUMBER DOTDOT NUMBER RBRACKET OF
            BasicType'''
120     low = int(p[3])
121     high = int(p[5])
122     base_type = p[8].lower()
123     p[0] = ('array', base_type, (low, high))
124
125 def p_FunctionDeclarations(p):
126     '''FunctionDeclarations : FunctionDeclaration FunctionDeclarations
127                             | empty'''
128     p[0] = p[1] + p[2] if len(p) == 3 else ""
```

```
129
130 def p_FunctionDeclaration(p):
131     '''FunctionDeclaration : FUNCTION ID LPAREN FormalParameters RPAREN
        COLON BasicType SEMICOLON Declarations CompoundStatement'''
132     global current_function, local_offset
133     func_name = p[2].lower()
134     params = p[4]
135     return_type = p[7].lower()
136     local_decls = p[9]
137     body = p[10]
138     func_label = new_label()
139     function_table[func_name] = {'label': func_label, 'params': params, '
        return_type': return_type}
140     old_offset = local_offset
141     local_offset = 1
142     symbol_table_stack.append({})
143     current_scope_level += 1
144     current_function = func_name
145     number_of_params = len(params)
146     return_offset = -(number_of_params + 1)
147     add_symbol(func_name, return_type, is_local=False, offset=
        return_offset)
148     for i, param in enumerate(params):
149         param_name, param_type = param
150         offset = -number_of_params + i
151         add_symbol(param_name, param_type, is_local=False, offset=offset)
152     local_decls_code = local_decls
153     number_of_locals = local_offset - 1
154     func_code = f"{func_label}:\nPUSHN {number_of_locals}\n{
        local_decls_code}{body}RETURN\n"
155     symbol_table_stack.pop()
156     current_scope_level -= 1
157     local_offset = old_offset
158     current_function = None
159     p[0] = func_code
160
161 def p_FormalParameters(p):
162     '''FormalParameters : ID COLON Type
163                         | ID COLON Type SEMICOLON FormalParameters
164                         | empty'''
165     if len(p) == 2:
166         p[0] = []
167     elif len(p) == 4:
168         p[0] = [(p[1], p[3])]
169     else:
170         p[0] = [(p[1], p[3])] + p[5]
171
172 def p_CompoundStatement(p):
```

```
    '''CompoundStatement : BEGIN StatementList END'''
    p[0] = p[2]

def p_StatementList(p):
    '''StatementList : Statement
                     | StatementList SEMICOLON Statement'''
    p[0] = p[1] if len(p) == 2 else f"{p[1]}{p[3]}"

def p_Statement(p):
    '''Statement : AssignmentStatement
                 | WriteStatement
                 | ReadStatement
                 | IfStatement
                 | WhileStatement
                 | ForStatement
                 | CompoundStatement
                 | empty'''
    p[0] = p[1]

def p_ReadStatement(p):
    '''ReadStatement : READLN LPAREN ArrayAccess RPAREN
                     | READLN LPAREN ID RPAREN'''
    if p[3][0] == 'array_access':
        var_name, index_code, index_type = p[3][1:]
        symbol = get_symbol(var_name)
        if not symbol or not symbol.get('is_array'):
            print(f"Error: '{var_name}' is not an array")
            p[0] = ""
            return
        if index_type != 'integer':
            print(f"Error: Array index must be integer")
            p[0] = ""
            return
        if symbol['type'] != 'integer':
            print(f"Error: Array elements must be integer for READLN")
            p[0] = ""
            return
        low = symbol['range'][0]
        offset = symbol['offset']
        p[0] = f"PUSHGP\n{index_code}PUSHI {low}\nSUB\nPUSHI {offset}\nADD
           \nPADD\nREAD\nATOI\nSTORE 0\n"
    else:
        var_name = p[3].lower()
        symbol = get_symbol(var_name)
        if not symbol:
            print(f"Error: Variable '{var_name}' not declared")
            p[0] = ""
            return
```

```
220         if symbol.get('is_array'):
221             print(f"Error: '{var_name}' is an array; use array indexing")
222             p[0] = ""
223             return
224         if symbol['type'] != 'integer':
225             print(f"Error: Variable must be integer for READLN")
226             p[0] = ""
227             return
228         offset = symbol['offset']
229         p[0] = f"READ\nATOI\nSTOREG {offset}\n"
230 
231 def p_AssignmentStatement(p):
232     '''AssignmentStatement : ID ASSIGN Expression
233                            | ArrayAccess ASSIGN Expression'''
234     global current_function
235     if p[1][0] == 'array_access':
236         var_name, index_code, index_type = p[1][1:]
237         expr_code, expr_type = p[3]
238         symbol = get_symbol(var_name)
239         if not symbol or not symbol.get('is_array'):
240             print(f"Error: '{var_name}' is not an array")
241             p[0] = ""
242             return
243         if index_type != 'integer':
244             print(f"Error: Array index must be integer")
245             p[0] = ""
246             return
247         if symbol['type'] != expr_type:
248             print(f"Error: Type mismatch in array assignment")
249             p[0] = ""
250             return
251         low = symbol['range'][0]
252         offset = symbol['offset']
253         p[0] = f"PUSHGP\n{index_code}PUSHI {low}\nSUB\nPUSHI {offset}\nADD
              \nPADD\n{expr_code}STORE 0\n"
254     else:
255         var_name = p[1].lower()
256         expr_code, expr_type = p[3]
257         symbol = get_symbol(var_name)
258         if not symbol:
259             print(f"Error: Variable '{var_name}' not declared")
260             p[0] = ""
261             return
262         if symbol.get('is_array'):
263             print(f"Error: '{var_name}' is an array; use array indexing")
264             p[0] = ""
265             return
266         if symbol['type'] != expr_type:
```

```
267             print(f"Error: Type mismatch in assignment")
268             p[0] = ""
269             return
270         offset = symbol['offset']
271         if var_name == current_function:
272             number_of_params = len(function_table[current_function]['
                    params'])
273             p[0] = f"{expr_code}STOREL -{number_of_params + 1}\n"
274         elif symbol.get('is_local'):
275             p[0] = f"{expr_code}STOREL {offset}\n"
276         else:
277             p[0] = f"{expr_code}STOREG {offset}\n"
278 
279 def p_WriteStatement(p):
280     '''WriteStatement : WRITELN LPAREN ExpressionList RPAREN'''
281     code = ""
282     for expr in p[3]:
283         expr_code, expr_type = expr
284         if expr_type == 'integer':
285             code += f"{expr_code}WRITEI\n"
286         elif expr_type == 'string':
287             code += f"{expr_code}WRITES\n"
288         else:
289             print(f"Error: Unsupported type '{expr_type}' for WRITELN")
290             p[0] = ""
291             return
292     p[0] = code
293 
294 def p_ExpressionList(p):
295     '''ExpressionList : Expression
296                       | ExpressionList COMMA Expression'''
297     if len(p) == 2:
298         p[0] = [p[1]]
299     else:
300         p[0] = p[1] + [p[3]]
301 
302 def p_Expression_length(p):
303     '''Expression : LENGTH LPAREN Expression RPAREN'''
304     expr_code, expr_type = p[3]
305     if expr_type != 'string':
306         print("Error: length() argument must be a string")
307         p[0] = ("", 'error')
308         return
309     p[0] = (f"{expr_code}STRLEN\n", 'integer')
310 
311 def p_Expression_function_call(p):
312     '''Expression : ID LPAREN ActualParameters RPAREN'''
313     func_name = p[1].lower()
```

```
314     if func_name == 'length':
315         print("Error: 'length' is a reserved function name")
316         p[0] = ("", 'error')
317         return
318     if func_name not in function_table:
319         print(f"Error: Undeclared function '{func_name}'")
320         p[0] = ("", 'error')
321         return
322     func_info = function_table[func_name]
323     args = p[3]
324     if len(args) != len(func_info['params']):
325         print(f"Error: Argument count mismatch for '{func_name}'")
326         p[0] = ("", 'error')
327         return
328     arg_code = ""
329     for arg, param in zip(args, func_info['params']):
330         arg_code += arg[0]
331         if arg[1] != param[1]:
332             print(f"Error: Type mismatch in argument for '{func_name}'")
333             p[0] = ("", 'error')
334             return
335     call_code = f"PUSHI 0\n{arg_code}PUSHA {func_info['label']}\nCALL\n"
336     p[0] = (call_code, func_info['return_type'])
337 
338 def p_IfStatement_then(p):
339     '''IfStatement : IF Expression THEN Statement'''
340     cond_code, cond_type = p[2]
341     if cond_type != 'boolean':
342         print("Error: Condition must be boolean")
343         p[0] = ""
344         return
345     then_code = p[4]
346     end_label = new_label()
347     p[0] = f"{cond_code}JZ {end_label}\n{then_code}{end_label}:\n"
348 
349 def p_IfStatement_else(p):
350     '''IfStatement : IF Expression THEN Statement ELSE Statement'''
351     cond_code, cond_type = p[2]
352     if cond_type != 'boolean':
353         print("Error: Condition must be boolean")
354         p[0] = ""
355         return
356     then_code = p[4]
357     else_code = p[6]
358     else_label = new_label()
359     end_label = new_label()
360     p[0] = f"{cond_code}JZ {else_label}\n{then_code}JUMP {end_label}\n{
        else_label}:\n{else_code}{end_label}:\n"
```

```
361
362 def p_WhileStatement(p):
363     '''WhileStatement : WHILE Expression DO Statement'''
364     start_label = new_label()
365     end_label = new_label()
366     print(p[2])
367     cond_code, cond_type = p[2]
368     if cond_type != 'boolean':
369         print("Error: While condition must be boolean")
370         p[0] = ""
371         return
372     body_code = p[4]
373     p[0] = f"{start_label}:\n{cond_code}JZ {end_label}\n{body_code}JUMP {
        start_label}\n{end_label}:\n"
374
375 def p_ForStatement(p):
376     '''ForStatement : FOR ID ASSIGN Expression TO Expression DO Statement
        '''
377     var_name = p[2].lower()
378     init_code, init_type = p[4]
379     end_code, end_type = p[6]
380     body_code = p[8]
381     symbol = get_symbol(var_name)
382     if not symbol:
383         print(f"Error: Variable '{var_name}' not declared")
384         p[0] = ""
385         return
386     if symbol.get('is_array'):
387         print(f"Error: Loop variable '{var_name}' cannot be an array")
388         p[0] = ""
389         return
390     if init_type != 'integer' or end_type != 'integer':
391         print(f"Error: FOR loop bounds must be integers")
392         p[0] = ""
393         return
394     if symbol['type'] != 'integer':
395         print(f"Error: Loop variable must be integer")
396         p[0] = ""
397         return
398     offset = symbol['offset']
399     start_label = new_label()
400     end_label = new_label()
401     load_op = 'PUSHL' if symbol.get('is_local') else 'PUSHG'
402     store_op = 'STOREL' if symbol.get('is_local') else 'STOREG'
403     p[0] = (
404         f"{init_code}{store_op} {offset}\n"
405         f"{start_label}:\n"
406         f"{load_op} {offset}\n"
```

```
407         f"{end_code}"
408         f"INFEQ\n"
409         f"JZ {end_label}\n"
410         f"{body_code}"
411         f"{load_op} {offset}\n"
412         f"PUSHI 1\n"
413         f"ADD\n"
414         f"{store_op} {offset}\n"
415         f"JUMP {start_label}\n"
416         f"{end_label}:\n"
417     )
418
419 def p_Expression_number(p):
420     '''Expression : NUMBER'''
421     p[0] = (f"PUSHI {p[1]}\n", 'integer')
422
423 def p_Expression_group(p):
424     '''Expression : LPAREN Expression RPAREN'''
425     p[0] = p[2]
426
427 def p_Expression_boolean(p):
428     '''Expression : TRUE
429                   | FALSE'''
430     p[0] = ("PUSHI 1\n", 'boolean') if p[1].lower() == 'true' else ("PUSHI
        0\n", 'boolean')
431
432 def p_Expression_string(p):
433     '''Expression : STRING_LITERAL'''
434     p[0] = (f"PUSHS \"{p[1]}\"\n", "string")
435
436 def p_Expression_id(p):
437     '''Expression : ID'''
438     var_name = p[1].lower()
439     symbol = get_symbol(var_name)
440     if not symbol:
441         print(f"Error: Undeclared variable '{var_name}'")
442         p[0] = ("", 'error')
443         return
444     if symbol.get('is_array'):
445         print(f"Error: '{var_name}' is an array; use array indexing")
446         p[0] = ("", 'error')
447         return
448     op = 'PUSHL' if symbol.get('is_local') else 'PUSHG'
449     p[0] = (f"{op} {symbol['offset']}\n", symbol['type'])
450
451 def p_Expression_array_access(p):
452     '''Expression : ID LBRACKET Expression RBRACKET'''
453     var_name = p[1].lower()
```

```
454     index_code, index_type = p[3]
455     symbol = get_symbol(var_name)
456     if not symbol:
457         print(f"Error: Undeclared variable '{var_name}'")
458         p[0] = ("", 'error')
459         return
460     if symbol.get('is_array'):
461         if index_type != 'integer':
462             print(f"Error: Array index must be integer")
463             p[0] = ("", 'error')
464             return
465         low = symbol['range'][0]
466         offset = symbol['offset']
467         code = f"PUSHGP\n{index_code}PUSHI {low}\nSUB\nPUSHI {offset}\nADD
              \nPADD\nLOAD 0\n"
468         p[0] = (code, symbol['type'])
469     elif symbol['type'] == 'string':
470         if index_type != 'integer':
471             print(f"Error: String index must be integer")
472             p[0] = ("", 'error')
473             return
474         load_op = 'PUSHL' if symbol.get('is_local') else 'PUSHG'
475         code = f"{load_op} {symbol['offset']}\n{index_code}PUSHI 1\nSUB\
              nCHARAT\n"
476         p[0] = (code, 'integer')
477     else:
478         print(f"Error: '{var_name}' is not an array or string")
479         p[0] = ("", 'error')
480
481 def p_Expression_function_call(p):
482     '''Expression : ID LPAREN ActualParameters RPAREN'''
483     func_name = p[1].lower()
484     if func_name not in function_table:
485         print(f"Error: Undeclared function '{func_name}'")
486         p[0] = ("", 'error')
487         return
488     func_info = function_table[func_name]
489     args = p[3]
490     if len(args) != len(func_info['params']):
491         print(f"Error: Argument count mismatch for '{func_name}'")
492         p[0] = ("", 'error')
493         return
494     arg_code = ""
495     for arg, param in zip(args, func_info['params']):
496         arg_code += arg[0]
497         if arg[1] != param[1]:
498             print(f"Error: Type mismatch in argument for '{func_name}'")
499             p[0] = ("", 'error')
```

```
500             return
501     call_code = f"PUSHI 0\n{arg_code}PUSHA {func_info['label']}\nCALL\n"
502     p[0] = (call_code, func_info['return_type'])
503 
504 def p_ActualParameters(p):
505     '''ActualParameters : Expression
506                         | Expression COMMA ActualParameters
507                         | empty'''
508     if len(p) == 2:
509         p[0] = [] if p[1] == '' else [p[1]]
510     else:
511         p[0] = [p[1]] + p[3]
512 
513 def p_Expression_binop(p):
514     '''Expression : Expression PLUS Expression
515                   | Expression MINUS Expression
516                   | Expression TIMES Expression'''
517     code1, type1 = p[1]
518     op = p[2]
519     code3, type3 = p[3]
520     if type1 == 'integer' and type3 == 'integer':
521         op_code = {'+': 'ADD', '-': 'SUB', '*': 'MUL'}[op]
522         p[0] = (f"{code1}{code3}{op_code}\n", 'integer')
523     else:
524         print("Error: Type mismatch in expression")
525         p[0] = ("", 'error')
526 
527 def p_Expression_moddiv(p):
528     '''Expression : Expression MOD Expression
529                   | Expression DIV Expression'''
530     code1, t1 = p[1]
531     code3, t2 = p[3]
532     if t1 == t2 == 'integer':
533         p[0] = (f"{code1}{code3}{p[2]}\n", 'integer')
534     else:
535         print("Error: Type mismatch in mod/div")
536         p[0] = ("", 'error')
537 
538 def p_Expression_rel(p):
539     '''Expression : Expression LT Expression
540                   | Expression GT Expression
541                   | Expression EQ Expression
542                   | Expression NEQ Expression
543                   | Expression LTE Expression
544                   | Expression GTE Expression'''
545     code1, type1 = p[1]
546     op = p[2]
547     code3, type3 = p[3]
```

```
548     if type1 == type3 and type1 in ('integer', 'boolean'):
549         op_code = {'<': 'INF', '>': 'SUP', '=': 'EQUAL', '<>': 'NOTEQUAL',
                '<=': 'INFEQ', '>=': 'SUPEQ'}[op]
550         p[0] = (f"{code1}{code3}{op_code}\n", 'boolean')
551     else:
552         print("Error: Type mismatch in relational expression")
553         p[0] = ("", 'error')
554 
555 def p_Expression_log(p):
556     '''Expression : Expression AND Expression
557                   | Expression OR Expression'''
558     code1, type1 = p[1]
559     op = p[2]
560     code3, type3 = p[3]
561     if type1 == 'boolean' and type3 == 'boolean':
562         if op == "and":
563             p[0] = (f"{code1}{code3}ADD\nPUSHI 2\nEQUAL\n", 'boolean')
564         elif op == "or":
565             p[0] = (f"{code1}{code3}ADD\nPUSHI 1\nSUPEQ\n", 'boolean')
566     else:
567         print("Error: Type mismatch in logical expression")
568         p[0] = ("", 'error')
569 
570 def p_ArrayAccess(p):
571     '''ArrayAccess : ID LBRACKET Expression RBRACKET'''
572     var_name = p[1].lower()
573     index_code, index_type = p[3]
574     p[0] = ('array_access', var_name, index_code, index_type)
575 
576 def p_empty(p):
577     '''empty :'''
578     p[0] = ""
579 
580 def p_error(p):
581     print("Syntax error")
582 
583 parser = yacc.yacc()
584 parser.exito = True
585 
586 PREMADE_EXAMPLES = {
587     "MaiorDe3": """
588         program Maior3;
589         var
590             num1, num2, num3, maior: integer;
591         begin
592             { Ler 3 n meros }
593             writeln('Introduza o primeiro n mero: ');
594             readln(num1);
```

```
595             writeln('Introduza o segundo n mero: ');
596             readln(num2);
597             writeln('Introduza o terceiro n mero: ');
598             readln(num3);
599             if num1 > num2 then
600                 if num1 > num3 then
601                     maior := num1
602                 else maior := num3
603             elseif num2 > num3
604                 then maior := num2
605             else
606                 maior := num3;
607             writeln(maior);
608         end.
609     """,
610
611     "Fatorial": """
612         program Fatorial;
613         var
614             n, i, fat: integer;
615         begin
616             writeln('Introduza um n mero inteiro positivo:');
617             readln(n);
618             fat := 1;
619             for i := 1 to n do
620                 fat := fat * i;
621             writeln('Fatorial de ', n, ': ', fat);
622         end.
623     """,
624
625     "NumeroPrimo": """
626         program NumeroPrimo;
627         var
628             num, i: integer;
629             primo: boolean;
630         begin
631             writeln('Introduza um n mero inteiro positivo:');
632             readln(num);
633             primo := true;
634             i := 2;
635             while (i <= (num div 2)) and primo do
636                 begin
637                     if (num mod i) = 0 then
638                         primo := false;
639                     i := i + 1;
640                 end;
641             if primo then
642                 writeln(num, '    um n mero primo')
```

```
643             else
644                 writeln(num, ' n o    um n mero primo');
645         end.
646     """,
647
648     "SomaArray": """
649         program SomaArray;
650         var
651             numeros: array[1..5] of integer;
652             i, soma: integer;
653         begin
654             soma := 0;
655             writeln('Introduza 5 n meros inteiros:');
656             for i := 1 to 5 do
657             begin
658                 writeln(i);
659                 readln(numeros[i]);
660                 soma := soma + numeros[i];
661             end;
662             writeln('A soma dos n meros   : ', soma);
663         end.
664     """,
665     "HelloWorld": """
666     program HelloWorld;
667         begin
668         writeln('Ola, Mundo!');
669     end.
670     """
671
672 }
673
674 def main():
675     parser_args = argparse.ArgumentParser(
676         description="Compile  P a s c a l like  source code into intermediate
            instructions."
677     )
678
679     group = parser_args.add_mutually_exclusive_group()
680     group.add_argument(
681         "-e", "--example",
682         choices=list(PREMADE_EXAMPLES.keys()),
683         help="Name of a premade example to compile"
684     )
685     group.add_argument(
686         "-f", "--file",
687         metavar="PATH",
688         help="Path to a  s o u r c e file  to compile"
689     )
```

```
690
691     args = parser_args.parse_args()
692
693     if args.example:
694         source_code = PREMADE_EXAMPLES[args.example]
695         print(f"*** Compiling example: {args.example} ***\n")
696     elif args.file:
697         try:
698             with open(args.file, 'r', encoding='utf-8') as fp:
699                 source_code = fp.read()
700             print(f"*** Compiling file: {args.file} ***\n")
701         except IOError as e:
702             print(f"Error: could not open file '{args.file}': {e}")
703             sys.exit(1)
704     else:
705         print("*** Reading source code from stdin (terminate with EOF)
               ***\n")
706         source_code = sys.stdin.read()
707
708     global symbol_table_stack, current_scope_level, stack_pointer
709     global local_offset, label_counter, current_function, function_table
710
711     symbol_table_stack = [{}]
712     current_scope_level = 0
713     stack_pointer = 0
714     local_offset = 0
715     label_counter = 0
716     current_function = None
717     function_table = {}
718
719     result = parser.parse(source_code, debug=False)
720
721     if result is None:
722         print("Parsing returned None (likely a syntax error).")
723         sys.exit(1)
724     else:
725         print("=== generated intermediate code ===\n")
726         print(result)
727
728 if __name__ == "__main__":
729     main()
```