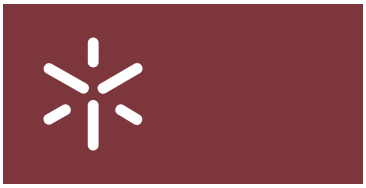

Universidade do Minho
Escola de Engenharia
Licenciatura em Engenharia Informática

Unidade Curricular de Processamento de Linguagens

Ano Letivo de 2024/2025

Processamento de Linguagens

Gonçalo Alves a104079
João Cunha a104611
João Sá a104612

Junho, 2025

PL

Índice

1. Introdução

O presente relatório surge no âmbito da Unidade Curricular **Processamento de Linguagens (PL)**, integrada no plano de estudos da Licenciatura em **Engenharia Informática** da Universidade do Minho. O trabalho prático proposto tem como objetivo principal **o desenvolvimento de um compilador para a linguagem Pascal standard**. Este projeto visa sensibilizar e motivar os estudantes para a implementação de compiladores, de forma a abranger todo o processo de compilação, desde a análise léxica até à geração de código para uma máquina virtual (VM).

A linguagem Pascal é reconhecida pela sua simplicidade e coerência, sendo amplamente utilizada no ensino da programação estruturada. Neste projeto, é utilizada como base para a implementação de um compilador capaz de traduzir programas escritos em Pascal para uma representação intermédia — o código da máquina virtual EWVM, disponibilizada aos alunos. Ao longo do desenvolvimento, são abordadas etapas fundamentais como a análise léxica e sintática, a verificação semântica, a geração e otimização de código, bem como a realização de testes de validação. Para tal, são utilizadas ferramentas como o PLY (Python Lex-Yacc), essenciais para a construção de compiladores modernos.

Ao longo deste relatório serão abordadas todas as componentes do projeto, desde a análise léxica até à geração do código da EWVM equivalente ao código Pascal.

2. Pascal

Numa fase inicial, começamos por fazer um pequeno estudo relativo à linguagem **Pascal**.

Pascal é uma linguagem procedimental, fortemente tipada e estruturada, muito utilizada no ensino de lógica de programação devido à sua clareza e organização. Esta possui uma sintaxe semelhante à linguagem natural e uma estrutura rígida, que obriga o programador a ter uma boa perceção de variáveis, tipos de dados, procedimentos, funções, estruturas de controlo e escopo.

Esta linguagem é excelente para iniciantes no mundo da programação, visto que possui vários conceitos fundamentais de forma acessível, permitindo que o estudante se foque na lógica e na organização do código.

Exemplo de código **Pascal**:

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

3. Máquina Virtual EWVM

O objetivo principal deste compilador é traduzir o código Pascal para código da máquina virtual EWVM, portanto é importante fazer um pequeno estudo da linguagem utilizada pela mesma. Esta máquina possui uma sintaxe semelhante ao *Assembly* e funciona à volta de uma stack. Desta forma, como esta é um pouco mais complexa, o seu entendimento não foi tão simples quanto o de Pascal.

Exemplo de código na linguagem da **EWVM**:

```
pushs "Ola, Mundo!"
writes
pushs "\n"
writes
stop
```

4. Arquitetura

4.1. Gramática

Para validar a estrutura gramatical da linguagem Pascal standard, decidimos adotar uma gramática *Bottom-up*. A estrutura geral da mesma é composta por declarações e um bloco de execução. A gramática desenvolvida é a seguinte:

```
Programa : PROGRAM ID ';' Bloco '.'

Bloco : VarSec Exec

VarSec : VAR ListaDeclaracaoVars
       | empty

ListaDeclaracaoVars : DeclVar ListaDeclaracaoVars
                    | empty

DeclVar : ListaVariaveis ':' Tipo ';'

Variavel : ID '[' Expr ']'
         | ID

ListaVariaveis : Variavel ListaVariaveisTail

ListaVariaveisTail : ',' Variavel ListaVariaveisTail
                   | empty

Tipo : INTEGER
     | REAL
     | CHAR_TYPE
     | BOOLEAN
     | STRING_TYPE
     | ARRAY '[' NUMBER '..' NUMBER ']' OF Tipo

Exec : BEGIN ListaInstrucao END

ListaInstrucao : Instrucao ';' ListaInstrucao
               | Instrucao
               | empty

Instrucao : Exec
          | IF Expr THEN Instrucao
```

```
          | IF Expr THEN Instrucao ELSE Instrucao
          | Loop
          | Atr
          | WRITE '(' ListaExpr ')'
          | WRITELN '(' ListaExpr ')'
          | WRITELN
          | READ '(' ListaVariaveis ')'
          | READLN '(' ListaVariaveis ')'

Loop : WHILE Expr DO Instrucao
     | FOR ID ':=' Expr TO Expr DO Instrucao
     | FOR ID ':=' Expr DOWNTO Expr DO Instrucao

Atr : ID ':=' Expr
    | ID '[' Expr ']' ':=' Expr

Expr : Expr OR TermoAnd
     | TermoAnd

ListaExpr : Expr ListaExprTail

ListaExprTail : ',' Expr ListaExprTail
              | empty

TermoAnd : TermoAnd AND TermoIgualdade
         | TermoIgualdade

TermoIgualdade : TermoIgualdade '=' TermoRelacional
               | TermoIgualdade '<>' TermoRelacional
               | TermoRelacional

TermoRelacional : TermoRelacional '>' TermoAditivo
                | TermoRelacional '<' TermoAditivo
                | TermoRelacional '>=' TermoAditivo
                | TermoRelacional '<=' TermoAditivo
                | TermoAditivo

TermoAditivo : TermoAditivo '+' TermoMultiplicativo
             | TermoAditivo '-' TermoMultiplicativo
             | TermoMultiplicativo

TermoMultiplicativo : TermoMultiplicativo '*' Fator
                    | TermoMultiplicativo '/' Fator
                    | TermoMultiplicativo MOD Fator
                    | Fator

Fator : ID
      | NUMBER
      | STRING
      | TRUE
      | FALSE
      | '(' Expr ')'
```

```
      | ID '[' Expr ']'
      | ID '.' ID
      | ID '(' ID ')'
      | CHAR

ListaArgumentos : Expr ListaArgumentosTail
                | empty

ListaArgumentosTail : ',' Expr ListaArgumentosTail
                    | empty

empty :
```

Um programa em PASCAL, tal como referido anteriormente, começa com a palavra reservada `PROGRAM`, seguida de um identificador (ID), de um **Bloco** e termina com um ponto final (.)

O símbolo não terminal, **Bloco** tem duas componentes. A primeira, **VarSec** corresponde à secção de declaração das variáveis e a segunda corresponde a **Exec** e é aqui onde estarão as secções de comandos/instruções (começa com a palavra reservada `BEGIN`).

No segundo símbolo, **VarSec**, tal como referido, as variáveis são declaradas após a palavra reservada `VAR`. É importante referir que é possível declarar múltiplas variáveis do mesmo tipo de uma só vez. O símbolo não terminal **Tipo**, tal como é possível deduzir, representa os tipos básicos (INTEGER, REAL , CHAR_TYPE, BOOLEAN, STRING_TYPE) e os tipos compostos (ARRAY) podendo ser de qualquer outro tipo.

Por sua vez, **Exec** contém uma lista de instruções (ListaInstrucao) entre as palavras reservadas `BEGIN` e `END`.

Na lista de instrução, surge o símbolo não terminal **Instrucao**. Tal como o nome indica, tem com objetivo representar as possíveis instruções do programa e aborda instruções condicionais, *loops*, atribuições e instruções IO.

Os *loops* têm um simbolo não terminal para representar as possíveis declarações. Para representar cada uma, recorremos à definição das palavras reservadas `WHILE` e `FOR`. O ciclo for permite a iteração cresecente (`TO`) ou decresente (`DOWNTO`).

As expressões (**Expr**) são compostas por vários níveis. A gramática desenvolvida respeita a precedência de operadores (podem ser símbolos como “*” e “+” ou palavras reservadas como `OR` e `AND`).

O símbolo não terminal **Fator**, origina do **TermoMultiplicativo** e poderá ser uma variável, constante, uma expressão (entre parênteses), entre outros. A gramática `ID '.' ID` seria utilizada para posteriormente implementar o tipo de dados **Record** presente na linguagem Pascal.

As operações input/output (IO) suportam a entrada e saída com listas de expressões ou variáveis. `WRITELN` poderá ser usada sem argumentos com o intuito de quebrar a linha.

Existem várias produções para listas separadas através de vírgulas e todas seguem o mesmo padrão:

```
Item ListaItemTail
ListaItemTail : ',' Item ListaItemTail | empty
```

Por fim, produções vazias são possíveis através `empty` , permitindo que várias regras possuam partes opcionais.

4.2. Ficheiros

4.2.1. Lexer

O **lexer** é o módulo responsável por dividir o código-fonte da linguagem Pascal em tokens. Para isso, foi utilizada a biblioteca PLY, que permite a definição de expressões regulares associadas a cada token da linguagem.

Neste projeto, foram definidos dois estados especiais:

```
states = (
    ('string', 'exclusive'),
    ('comment', 'exclusive'),
)
```

- O estado **string** permite reconhecer corretamente strings entre apóstrofos, inclusive com apóstrofos duplos (`''`) dentro do texto.
- O estado **comment** é usado para ignorar o conteúdo entre `{ ... }` e `(* ... *)`, visto que estes representam os comentários da linguagem.

Foram definidos tokens para:

- **Identificadores** e **números**;
- **Operadores aritméticos** e **lógicos** `(+, -, *, div, mod, and, or, not, =, <>, <, <=, >, >=)`;
- **Delimitadores** `(;, :, ,, ., .., (, ), [, ])`;
- **Palavras reservadas** (ex: *program*, *var*, *begin*, *end*, *if*, *then*, *else*, *while*, *for*, *procedure*, *function*, *read*, *write*, etc.).

4.2.2. Yaccer

O **yaccer** é o módulo onde definimos e construimos a gramática e onde gerámos o código da EWVM. Para tratar da parte do armazenamento e verificar o sucesso da tradução, foram definidas algumas variáveis.

- **parser.success**: Controla se o parsing foi bem-sucedido ou se ocorreram erros, devolvendo `false` caso ocorra algum erro.

- **parser.var_counter**: Contador de endereços de memória para variáveis utilizado para alocar espaço para as mesmas.
- **parser.symbol_table**: Dicionário que armazena informações sobre variáveis declaradas.
- **parser.label_counter**: Contador responsável por criar pontos de salto únicos para estruturas de controlo (if, while, for).
- **parser.used_variables**: Set responsável por armazenar as variáveis utilizadas.

4.2.3. Anasem

O **anasem** é uma componente fundamental do compilador responsável por realizar a análise semântica do código. Esta classe implementa um conjunto abrangente de verificações que garantem a correção semântica do programa, sendo estas:

- **Declaração de Variáveis**: Confirma que todas as variáveis utilizadas foram previamente declaradas.
- **Compatibilidade de Tipos**: Verifica atribuições, operações aritméticas (integer/real), lógicas (boolean) e de comparação entre tipos compatíveis.
- **Acesso a Estruturas**: Valida o acesso correto a arrays e strings, incluindo verificação de tipos de índices e limites quando conhecidos em tempo de compilação.
- **Estruturas de Controlo**: Garante que condições são do tipo boolean e que variáveis de controlo em loops FOR são do tipo integer.

O sistema mantém listas separadas de erros e avisos (como variáveis não utilizadas), associando cada problema ao número da linha onde este ocorre. Esta componente assegura que apenas programas semanticamente corretos prossigam para as fases seguintes da compilação.

5. Implementação

5.1. Estruturas Condicionais

As estruturas condicionais em Pascal foram implementadas de forma direta, suportando duas variantes:

- `IF Expr THEN Instrucao`
- `IF Expr THEN Instrucao ELSE Instrucao`

O compilador utiliza um sistema de labels dinâmicas geradas pela função `new_label()` para controlar o fluxo de execução. Cada estrutura condicional recebe labels únicos (`else_label` e `end_label`) que garantem o funcionamento correto de condicionais.

5.2. Estruturas de Repetição

Implementados três tipos de loops:

- **WHILE-DO**: Verifica condição inicial
- **FOR-TO**: Incrementa de forma crescente com gestão automática da variável de controlo
- **FOR-DOWNTO**: Incrementa de forma decrescente

O compilador gere automaticamente a inicialização, a verificação da condição e a atualização da variável de controlo.

5.3. Variáveis

Sistema baseado em tabela de símbolos que armazena nome, endereço e tipo de cada variável. Suporta:

- Tipos básicos: `INTEGER`, `REAL`, `CHAR_TYPE`, `BOOLEAN`
- Arrays: `ARRAY[inicio..fim] OF tipo`
- Strings: `STRING[n]` ou `STRING` (255 caracteres)

Todas as variáveis são verificadas de acordo com a declaração previamente feita, com mensagens de erro detalhadas.

5.4. Expressões

Implementação hierárquica respeitando a precedência de operadores:

1. OR (menor precedência)
2. AND
3. Igualdade (=, <>)
4. Relacionais (<, >, <=, >=)
5. Adição (+, -)
6. Multiplicação `(*, DIV, MOD)`

Cada operação é traduzida para as correspondentes instruções na linguagem da máquina virtual.

5.5. Input/Output

- **WRITE/WRITELN**: Suporta números, strings e variáveis
- **READ/READLN**: Input com conversão automática para inteiros utilizando `atoi`

5.6. Gestão de Memória

Sistema automático que:

- Conta variáveis declaradas (**`parser.var_counter`**)
- Reserva espaço inicial com **`pushn n`**
- Aloca arrays e strings dinamicamente

5.7. Sistema de Labels e Erros

- Geração automática de labels únicas para estruturas de controlo
- Verificação de erros em tempo de compilação
- Flag **`parser.success`** verifica o sucesso da geração do código final

6. Conclusão

O desenvolvimento deste projeto permitiu uma compreensão prática da matéria lecionada e dos principais componentes de um compilador, incluindo a análise léxica, sintática e semântica. Para além disso, abordou a geração de código para máquina virtual, bastante semelhante a *Assembly*. Através da implementação de um compilador de Pascal, foi possível aplicar conceitos teóricos, que sem este projeto não teriam sido consolidados como suposto. O projeto demonstrou a importância de estruturação e da validação para garantir o funcionamento e eficiência de um compilador. Tendo isto em conta, o grupo sente-se bastante satisfeito com o trabalho desenvolvido.

Pascal foi a primeira linguagem de programação para alguns dos elementos do grupo, no entanto, foi necessário um estudo da linguagem para entender como a mesma funcionava de maneira a facilitar a implementação do compilador.

No que toca a trabalho futuro, reconhecemos que existem certas nuances que podem ser melhoradas. Uma dessas seria “polir” melhor a gramática. Esta pode ser melhorada, tornando a mesma mais abrangente.

De uma forma geral, consideramos que o trabalho foi bem conseguido e ajudou-nos a consolidar componentes teóricas cruciais para o nosso percurso na área da programação.