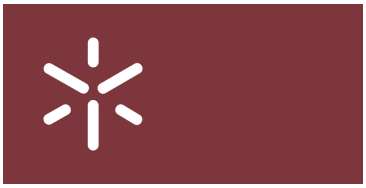

Universidade do Minho
Escola de Engenharia
Licenciatura em Engenharia Informática
Licenciatura em Engenharia Informática

Unidade Curricular - Processamento de Linguagens

Ano Letivo de 2024/2025

Construção de um Compilador para Pascal Standard

Grupo 11

Tomás Henrique Alves Melo a104529
João Gustavo da Silva Couto Mendes Serrão a104444
José Pedro Torres Vasconcelos a100763

PL

Junho, 2025

Resumo

Este relatório tem com objetivo mostrar todo o planeamento, e as fases de desenvolvimento que foram pensadas pelo grupo, para que o trabalho prático da unidade curricular de Processamento de Linguagens, no ano letivo de 2024/25, cujo objetivo consiste na construção de um compilador para uma especificação da linguagem de programação Pascal, o Pascal Standard, fosse bem sucedido.

Área de Aplicação: Engenharia de Software, Desenvolvimento de Compiladores, Análise e Processamento de Linguagens Formais, Automatização de Análise de Código, Sistemas baseados em linguagens de domínio específico (DSLs)

Palavras-Chave: Compilador, Análise léxica, Análise sintática, Análise semântica, Geração de código VM, Otimização de código, Pascal Standard, PLY (Python Lex-Yacc), Tokenização, Parser, Testes de compilador

Índice

Lista de Figuras

1. Introdução

Este relatório descreve o desenvolvimento de um compilador para a linguagem Pascal, realizado no âmbito da unidade curricular de Processamento de Linguagens (2024/25). O objetivo principal foi construir um compilador funcional capaz de analisar, validar e traduzir programas em Pascal linguagem `VM` fornecida pela equipa docente.

O projeto foi dividido em fases clássicas da construção de compiladores: análise léxica, sintática, semântica e geração de código `VM`. Foram também implementados recursos como gestão de escopos, verificação de tipos e suporte a estruturas de controlo, garantindo a correção do código gerado.

O relatório detalha a implementação das principais componentes, as decisões técnicas adotadas, os desafios enfrentados e exemplos de testes que demonstram o funcionamento do compilador.

2. Arquitetura Desenvolvida

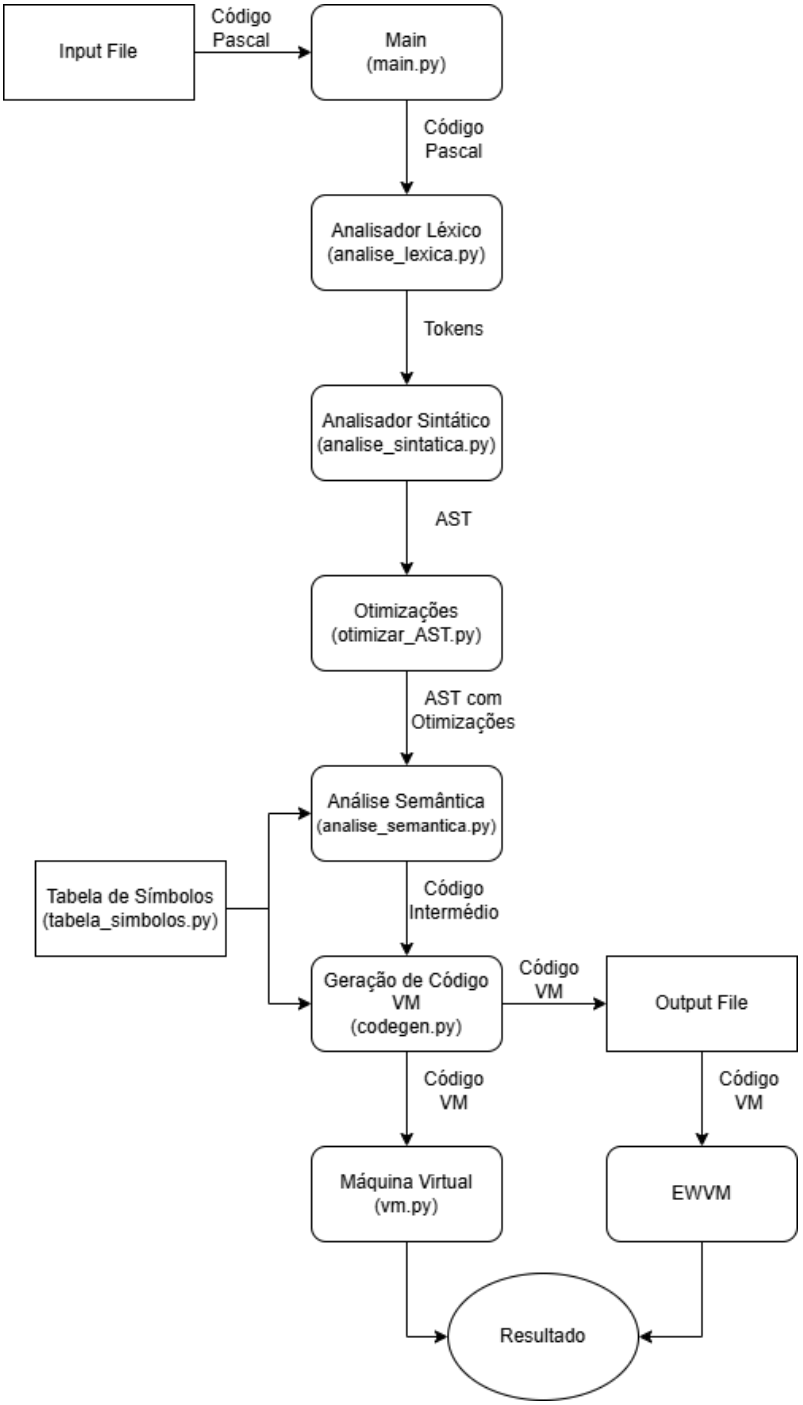

Figura 1: Arquitetura do Projeto

3. Organização do Código

Figura 2: Organização do código

O projeto está estruturado de forma a facilitar a manutenção, a extensibilidade e a compreensão do fluxo do compilador. Dividimos em dois principais módulos.

O módulo `'examples'` que possui uma pasta `'pas'` constituída por programas pascal de extensão `.pas` que servem como programas de teste para testar a correção do compilador e verificar possíveis erros que pudessem surgir ao longo do desenvolvimento do compilador, e ainda uma pasta `'vm'` onde são criados os ficheiros de extensão `.vm` a partir dos programas de teste com instruções `VM`.

O módulo `'src'` possui os seguintes programas necessários para a verdadeira implementação do compilador:

- **ANALISE_LEXICA.PY**
  Contém o analisador léxico, implementado a partir da biblioteca `PLY` *(Python Lex-Yacc)*. Este módulo é responsável por transformar o código-fonte Pascal numa sequência de `tokens`, reconhecendo as palavras-chave, identificadores, números, strings, operadores e símbolos. Ignora ainda comentários e espaços em branco, mantendo o foco apenas nos elementos sintáticos relevantes.
- **ANALISE_SINTATICA.PY**
  Define a gramática da linguagem e implementa o analisador sintático, também com recurso à biblioteca `PLY`. Através das regras gramaticais é construída uma Árvore Sintática Abstrata `(AST)` que representa estruturalmente o programa Pascal. Inclui ainda a inserção dos identificadores na Tabela de Símbolos durante a análise de declarações.
- **OTIMIZAR_AST**
  Este programa é responsável por aplicar otimizações sobre a `AST`. A principal otimização implementada consiste na remoção de variáveis declaradas que nunca são utilizadas no decorrer do programa. Isto é feito em duas passagens: a primeira percorre a `AST` para identificar todas as variáveis efetivamente usadas, e a segunda elimina as declarações de variáveis que não foram encontradas.
- **ANALISE_SEMANTICA.PY**
  Responsável pela análise semântica. Este módulo percorre a `AST` e realiza verificações semânticas, como:
  - validação de tipos em expressões e atribuições;

- existência e escopo de variáveis;
- verificação de tipos compatíveis em condições (`if, while, for, etc`).

- **TABELA_SIMBOLOS.PY**
  Implementa a tabela de símbolos, com suporte a escopos aninhados, `arrays` e tipos básicos. Esta tabela é usada tanto pela análise semântica como na geração de código para manter informação sobre os identificadores declarados, como tipo, endereço, dimensão, entre outros.
- **CODEGEN.PY**
  Este programa é responsável por, finalmente, gerar o código `VM`, ao converter a `AST` para linguagem `VM`.

4. Funcionalidades Implementadas

4.1. Análise Léxica

4.1.1. Reconhecimento dos Tokens

O analisador léxico desenvolvido reconhece os seguintes elementos da linguagem:

- Símbolos e operadores especiais, tais como `'ID'`, `'INTEGER'`, `'REAL'`, `'STRING'`, `'PLUS'`, `'MINUS'`, `'DIVIDE'`, `'TIMES'`, `etc.`
- Palavras reservadas, tais como `'PROGRAM'`, `'BEGIN'`, `'END'`, `'READ'`, `'READLN'`, `'WRITE'`, `'WRITELN'`, `'FOR'`, `'WHILE'`, `'IF'`, `etc.`

Para a listagem dos `tokens` da linguagem, armazenamos o conjunto de símbolos, operadores especiais e palavras-reservadas numa lista `tokens` e implementámos um método regulado por expressões regulares para cada um.
Exemplo:

```
tokens = ['REAL', 'INTEGER', 'ID', 'PROGRAM']
```

```
    def t_REAL(self, t):
        r'\d+\.\d+'
        t.value = float(t.value)
        return t


    def t_INTEGER(self, t):
        r'\d+'
        t.value = int(t.value)
        return t


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

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

Para garantirmos o *case insensitive* nas palavras-reservadas, adicionámos a `flag` *IGNORECASE* no momento de construção do *lexer*:

```
    def build(self, **kwargs):
        self.lexer = lex.lex(module=self, reflags=re.IGNORECASE, **kwargs)
        return self.lexer
```

Desta forma, o compilador reconhece palavras-reservadas como '`Program`', '`PROGRAM`' ou '`program`' de igual forma, característica da linguagem Pascal.

4.1.2. Comentários

O *lexer* implementa suporte aos dois formatos de comentário da linguagem Pascal.

- Comentários entre chavetas `{...}`
- Comentários entre parênteses e asteriscos `(*...*)`

```
def t_COMMENT(self, t):
    r'\{[^}]*\}|\(\*[\s\S]*?\*\)'
    pass
```

Durante a *tokenização*, qualquer conteúdo identificado como comentário é ignorado (usando `pass`), não gerando `tokens` para qualquer conteúdo em comentário.

4.1.3. Deteção de Caracteres Ilegais

Tal como aprendido nas aulas, garantimos o método `t_error` para detetar `tokens` que não foram reconhecidos corretamente no momento de *tokenização*, indicando o caractere inválido e a linha do código onde ocorreu numa mensagem de erro.

```
    def t_error(self, t):
        print(f"Caractere ilegal '{t.value[0]}' na linha {t.lexer.lineno}")
        t.lexer.skip(1)
```

4.2. Análise Sintática

A análise sintática é responsável por verificar se a sequência de *tokens* gerada pelo analisador léxico segue as regras gramaticais corretas. Esta etapa foi implementada utilizando a ferramenta `ply.yacc`, que permite a construção de *parsers* com base numa gramática livre de contexto, escrita em formato semelhante a `BNF(Backus-Naur Form)`.

4.2.1. Implementação da Gramática

A gramática foi implementada com suporte completo às principais construções da linguagem Pascal, incluindo declarações de variáveis com tipos primitivos como **`integer, real, boolean, string e char`**, bem como **`arrays unidimensionais`** com índices inteiros. Foram também suportados blocos compostos delimitados por **`begin...end`**, contendo comandos simples ou aninhados, e **comandos de atribuição** com variáveis simples ou elementos de `arrays`, permitindo **expressões aritméticas e booleanas** com a devida precedência e associatividade. A estrutura contempla ainda comandos de controlo de fluxo, como **`if...then...else, while...do`** e **`for...to/downto...do`**, bem como chamadas a procedimentos padrão como **`writeln, readln e write`**, com múltiplos argumentos e diferentes tipos de dados.

4.2.2. Construção da `AST`

A construção da **`AST`** é realizada durante a análise sintática. Cada regra da gramática definida no analisador gera **um nó da `AST`** utilizando a classe **`Node`**, que representa estruturas semânticas do programa como **declarações, comandos ou expressões**.

A **`AST`** representa a estrutura lógica do programa de forma hierárquica, ignorando elementos sintáticos supérfluos (como parênteses ou pontuação), e permitindo capturar apenas os elementos essenciais.

No código implementado, a **`AST`** é construída progressivamente à medida que os **`tokens`** são analisados e combinados com as **regras sintáticas**. O **nó raiz da árvore** corresponde ao **programa**, e os seus **ramos** abrangem **blocos, comandos e expressões**.

```
class Node:
    def __init__(self, type, children=None, leaf=None):
```

```
        self.type = type
        self.children = children if children else []
        self.leaf = leaf
```

A regra **p_program**, por exemplo, define a regra para a unidade de programa completa de um programa Pascal, sendo o ponto de entrada da gramática, reconhecendo o padrão geral:

```
program NomeDoPrograma;
<bloco>
.
```

```
def p_program(self, p):
    '''program : PROGRAM ID SEMICOLON block PERIOD'''
    p[0] = Node('program', [p[4]], p[2])
```

4.3. Análise Semântica

A análise semântica é a etapa do compilador responsável por garantir que o código fonte seja coerente do ponto de vista lógico e de tipos, respeitando as regras semânticas da linguagem. Nesta fase, são identificados erros que não podem ser detetados apenas pela análise léxica ou sintática.

4.3.1. Tabela de Símbolos e Escopos

```
class Symbol:
    def __init__(self, name, type=None, value=None, kind=None, params=None, scope=None,
address=None):
        self.name = name
        self.type = type
        self.value = value
        self.kind = kind
        self.params = params
        self.scope = scope
        self.address = address
        self.size = 1
        self.dimensions = None
        self.element_type = None

class SymbolTable:
    def __init__(self):
        # Escopo global
        self.scopes = [{}]
        self.current_scope = 0
        self.scope_names = ["global"]
```

Foi implementado um sistema de **tabela de símbolos** com suporte a **escopos aninhados** para controlar a **visibilidade** e a **validade de identificadores** ao longo do programa.

Cada **escopo** mantém sua **própria tabela de símbolos**, permitindo a inserção e remoção de identificadores conforme a entrada e saída de blocos, deteção de redefinições indevidas e resolução correta de nomes conforme o escopo atual.

4.3.2. Verificações Semânticas Realizadas

Durante a travessia da **Árvore de Sintaxe Abstrata** (AST), são efetuadas diversas **validações semânticas** com o apoio da tabela de símbolos, de forma a garantir a consistência lógica do programa. Uma das principais verificações consiste em garantir que todas as variáveis utilizadas tenham sido previamente declaradas. O uso de identificadores não definidos resulta num erro semântico.

Adicionalmente, são analisadas as expressões aritméticas e booleanas para assegurar a compatibilidade de tipos entre operandos, assim como a correspondência entre o tipo de uma variável e o valor atribuído à mesma.

No que respeita ao controlo de fluxo, as estruturas condicionais e de repetição, como `if` e `while`, são analisadas para garantir que a condição especificada é uma expressão booleana válida.

Estas verificações são fundamentais para assegurar que o programa analisado está semanticamente correto, prevenindo comportamentos inesperados em tempo de execução e garantindo uma base sólida para a geração de código posterior.

4.4. Geração de Código Intermediário

A geração de código intermediário consiste na transformação da representação interna do programa (a **Árvore de Sintaxe Abstrata** - AST) numa **sequência de instruções** adequadas para uma **máquina virtual** simples (VM), fornecida no contexto deste projeto. Essa etapa tem como objetivo produzir um código executável pela **VM**, mantendo a lógica e semântica do programa Pascal original.

4.4.1. Estrutura do Código Gerado

O código gerado é composto por uma série de instruções intermediárias que simulam o comportamento de uma arquitetura de máquina abstrata. Estas instruções são de baixo nível, porém suficientemente expressivas para representar as principais construções da linguagem.

4.4.2. Suporte a Operações

A geração de código contempla suporte completo às seguintes categorias de instruções:

- **Operações aritméticas** (ADD, SUB, MUL, DIV) e **relacionais** (EQ, NE, LT, LE, GT, GE) entre operandos, com base na análise semântica de tipos;
- Controle de fluxo, incluindo:
  - **Instruções de salto incondicional** (JUMP);
  - **Saltos condicionais** (JZ – jump if zero, JNZ – jump if not zero);
  - **Geração e utilização de *labels*** para representar destinos de saltos em estruturas como **`if`, `while` e `for`**;
- **Gestão de variáveis e `arrays`** na memória da VM:
  - **Atribuições** (STORE, LOAD);
  - **Acesso a elementos de arrays** por índice;
  - **Endereçamento relativo** em blocos e escopos;
- Entrada e saída de dados, por meio de instruções específicas como:
  - **`READLN`** para leitura de valores da entrada padrão;
  - **`WRITELN`** para escrita de valores na saída padrão, com suporte a diferentes tipos;
- Estrutura do programa, com:
  - **Instrução `START`** para inicializar a execução do programa;
  - **Instrução `STOP`** para finalizar a execução corretamente.

4.4.3. Tradução Dirigida pela Sintaxe

A **geração das instruções** é realizada de forma dirigida pela **sintaxe**, com base na travessia da **AST** construída durante a **análise sintática**. Cada tipo de nó da árvore é traduzido diretamente para uma ou mais instruções equivalentes da máquina virtual, permitindo uma geração modular, clara e extensível.

4.5. Suporte a Estruturas de Controlo

O compilador implementado oferece **suporte** completo às **principais estruturas de controlo de fluxo** da linguagem ***Pascal***, permitindo a execução condicional e repetitiva de blocos de código. Essas estruturas são fundamentais para a expressividade da linguagem e são traduzidas para instruções adequadas da máquina virtual (`VM`).

4.5.1. Condicional: `if-then[-else]`

A estrutura condicional **`if-then[-else]`** permite que determinados blocos de código sejam executados com base na avaliação de uma expressão **`booleana`**. A implementação garante:

- **Verificação semântica** de que a condição seja do tipo **`boolean`**;
- **Tradução** para a `VM` utilizando:
  - **Geração de `labels`** para ramificações;
  - **Instruções de salto condicional** (`JZ ou JNZ`);
  - **Suporte opcional** à cláusula **`else`**, com salto incondicional para ignorar o bloco alternativo quando necessário.

4.5.2. Ciclo `while-do`

O ciclo **`while-do`** permite a execução repetitiva de um bloco enquanto uma condição **`booleana`** for verdadeira. O compilador implementa:

- **Avaliação da condição** antes de cada iteração;
- **Geração de instruções de salto** (`JZ`) para sair do ciclo caso a condição seja falsa;
- **Reutilização de `labels`** para o início e fim do bloco de repetição, garantindo controle correto do fluxo.

4.5.3. Ciclo `for ... to/downto ... do`

O ciclo **`for`** permite a repetição controlada através de um contador. A implementação suporta as duas variantes da linguagem:

- **`for i := a to b do`**: incremento automático do contador;
- **`for i := a downto b do`**: decremento automático do contador.
- Durante a geração de código, são realizadas:
  - **Inicialização do contador** com o valor inicial;
  - **Comparação com o valor final** antes de cada iteração;
  - **Incremento ou decremento do valor** conforme a direção do ciclo;
  - **Tradução para instruções de controle** (`LOAD, STORE, CMP, JUMP, JZ/JNZ`).

4.5.4. Instrução `halt`

A instrução especial **`halt`** não era requerida, porém surgiu a necessidade de a criar para testarmos **programas mais complexos** e tornarmos o nosso compilador mais **robusto e completo**. Esta é utilizada

para **terminar imediatamente** a execução do programa. A sua presença no código-fonte gera a instrução **`STOP`** na saída da **`VM`**, sinalizando a interrupção definitiva da execução.

4.6. Instruções de Entrada e Saída

O compilador desenvolvido inclui suporte às **operações de entrada e saída de dados**, elementos essenciais para a interação entre o utilizador e o programa em execução. As instruções correspondentes são traduzidas para chamadas específicas da máquina virtual (`VM`), permitindo o correto processamento de comandos como o **`readln`** e **`writeln`** no código Pascal.

4.6.1. `writeln`

A instrução **`writeln`** foi implementada com suporte à impressão de vários tipos de dados:

- **Inteiros** (`integer`);
- **Reais** (`real`);
- **Strings** (`string`);
- **Booleanos** (`boolean`).

Durante a geração de código, é realizada a **identificação do tipo** de cada valor passado à função, sendo emitida a instrução de saída correspondente. A **`VM`** trata a **formatação e apresentação dos dados**, garantindo que os valores sejam corretamente exibidos na consola.

A instrução suporta ainda a impressão de expressões compostas, avaliando os seus operandos antes de os enviar para saída.

4.6.2. `readln`

A função **`readln`** permite a leitura de dados introduzidos pelo utilizador a partir da entrada padrão. A implementação inclui um **mecanismo de conversão automática de tipos**, consoante o tipo da variável de destino:

- **Conversão de string para inteiro** (`atoi`) quando o destino é do tipo `integer`;
- **Conversão de string para real** (`atof`) quando o destino é do tipo `real`.

Esta conversão garante que os valores introduzidos sejam corretamente armazenados na memória virtual com o tipo apropriado. A leitura é feita diretamente pela `VM`, sendo o valor lido convertido e atribuído à variável correspondente.

A abordagem adotada assegura que a interação com o utilizador seja intuitiva, precisa e compatível com os requisitos da linguagem Pascal standard.

4.7. Otimizações

Foram aplicadas algumas **otimizações** ao longo do processo de compilação, com o objetivo de **melhorar a eficiência e a qualidade do código gerado**, bem como de **evitar erros** comuns em tempo de compilação.

4.7.1. Eliminação de Comentários

Os **comentários** no código-fonte Pascal, quer no formato **`{...}`** quer **`(* ... *)`**, são **ignorados** já na **fase léxica**. Isso significa que não são incluídos na lista de `tokens` nem propagados para as fases seguintes do compilador. Desta forma, não influenciam a geração de código nem o comportamento do programa final, o que **reduz o ruído e o volume de informação processada**.

4.7.2. Remoção de variáveis declaradas não usadas

Depois de criada a **AST** existe uma travessia pela mesma para tentar encontrar e remover **variáveis não usadas** pelo programa, assim **evitando** a necessidade de as **declarar**, **guardando o espaço de memória**. Este processo ocorre em **duas fases**, uma travessia que encontra todos os usos de variáveis, guarda os nomes delas e uma outra travessia, que vai pelas declarações e, caso a variável não esteja incluída nas previamente guardadas, a declaração da mesma é removida.

4.8. Programa de Testes para Validação - TEST.PY

Para validar o funcionamento do compilador, foi desenvolvido o **`script test.py`**, que testa programas Pascal nas **fases léxica, sintática e semântica**. O **`script`** imprime os **`tokens`** detetados, a **AST** gerada (caso válida) e os **erros sintáticos** ou **semânticos** encontrados, ou declara o **sucesso da análise**. Pode ser usado com diferentes modos de execução e facilita a verificação da correção do compilador.

Testar análise léxica:

```
python3 test.py examples/pas/ex1.pas tokens
```

Testar análise sintática:

```
python3 test.py examples/pas/ex1.pas ast
```

Testar análise semântica:

```
python3 test.py examples/pas/ex1.pas semantic
```

Testar análise léxica, análise sintática (com impressão da AST) e análise semântica:

```
python3 test.py examples/pas/ex1.pas all
```

4.9. Gestão de Erros e Mensagens Informativas

Um dos aspetos essenciais na construção deste compilador foi **garantir mensagens de erro claras e informativas**, que ajudem o utilizador a identificar e corrigir facilmente problemas ao longo de todas as fases da compilação.

4.9.1. Mensagens de erro

As **mensagens de erro** emitidas pelo compilador incluem **informações claras** sobre o **tipo de erro** e a **linha** onde ocorreu, permitindo ao utilizador identificar facilmente os **problemas** no seu código e **corrigi-los de forma eficaz**. Esta abordagem melhora a usabilidade da ferramenta e apoia o processo de desenvolvimento de programas corretos.

4.9.1.1. Erros Léxicos

Durante a análise léxica, são detetados caracteres ilegais que não pertencem ao alfabeto da linguagem Pascal. Quando tal acontece, é gerada uma mensagem indicando o carácter inválido e a linha correspondente, permitindo uma correção imediata.

Erro léxico ao adicionar $ ao em vez de ;

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

Mensagem de erro:

```
Caractere ilegal '$' na linha 1
Erro de sintaxe na linha 2, token 'begin'
Erros de parsing:
 - Erro de sintaxe na linha 2, token 'begin'
```

4.9.1.2. Erros Sintáticos

Na fase de análise sintática, são detetadas **violações das regras gramaticais** da linguagem. O **`parser`** foi configurado para indicar, sempre que possível, a produção esperada ou o símbolo que causou a falha.

Adicionar um LBRACKET extra na instrução writeln

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

Mensagem de erro:

```
Erro de sintaxe na linha 3, token ';'
Erros de parsing:
 - Erro de sintaxe na linha 3, token ';'
```

4.9.1.3. Erros Semânticos

Durante a análise semântica, o compilador garante a **coerência lógica** do programa, verificando, por exemplo, se as variáveis foram declaradas antes de serem usadas ou se os tipos nas atribuições são compatíveis.

Tentativa de somar um valor inteiro com uma string

```
program Soma;
var
  a, c: integer;
  b: string;
begin
  readln(a);
  readln(b);
  c := a + b;
  writeln(c);
end.
```

Mensagem de erro:

```
Erros semânticos encontrados:
 - Erro: operação aritmética requer operandos do tipo inteiro ou real
```

5. Programas Exemplo

5.1. Programa que verifica se o número introduzido pelo utilizador é primo

```
program NumeroPrimo;
var
    num, i: integer;
    primo: boolean;
begin
    writeln('Introduza um número inteiro positivo:');
    readln(num);

    if num < 0 then
      begin
      writeln('Não existem números primos negativos.');
      halt;
      end;

    if num < 2 then
        primo := false
    else
    begin
        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;
    end;


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

INPUT:

```
3
```

OUTPUT:

```
3 é um número primo
```

5.2. Programa que soma os valores de um array que estejam em índices ímpares

```
program SomaIndicesImpares;
var
  v: array[1..6] of integer;
  i, soma: integer;
begin
  soma := 0;
  writeln('Introduza 6 números:');
  for i := 1 to 6 do
    readln(v[i]);

  for i := 1 to 6 do
    if i mod 2 = 1 then
      soma := soma + v[i];

  writeln('Soma dos índices ímpares: ', soma);
end.
```

INPUT:

```
1
2
3
4
5
6
```

OUTPUT:

```
Soma dos índices ímpares: 9
```

5.3. Programa que verifica se um array está ordenado

```
program VerificarOrdem;

var
  i, n, atual, anterior: integer;
  ordenado: boolean;

begin
  writeln('Quantos números vais inserir?');
  readln(n);

  if n <= 1 then
  begin
    writeln('Sequência está ordenada por definição.');
    halt;
  end;

  ordenado := true;

  writeln('Número 1:');
  readln(anterior);

  for i := 2 to n do
  begin
    writeln('Número ', i, ':');
    readln(atual);
```

```
    if atual < anterior then
      ordenado := false;
    anterior := atual;
  end;

  if ordenado then
    writeln('A sequência está ordenada!')
  else
    writeln('A sequência não está ordenada.');
end.
```

INPUT:

```
4
1
2
3
4
```

OUTPUT:

```
A sequência está ordenada!
```

5.4. Média de 3 valores

```
program MediaTresNumeros;
var
  a, b, c: real;
  media: real;
begin
  writeln('Introduza três números:');
  readln(a);
  readln(b);
  readln(c);
  media := (a + b + c) / 3;
  writeln('A média é: ', media);
end.
```

INPUT:

```
1
4
8
```

OUTPUT:

```
A média é:  4.333333333333333
```

5.5. Ordem das operações e reconhecimento de comentários

```
program CalcularExpressaoComplexa;

var
  resultado: integer;
begin
  { Calcular 2 + 2 * 3 * 2, respeitando a ordem das operações }
  resultado := 2 + 2 * 3 * 2;

  (*Resultado final*)
```

```
  WriteLn('O resultado de 2 + 2 * 3 * 2 é: ', resultado);
end.
```

OUTPUT:

```
O resultado de 2 + 2 * 3 * 2 é:  14
```

VM CODE:

```
pushi 0
storeg 0
start
pushi 2
pushi 2
pushi 3
mul
pushi 2
mul
add
storeg 0
pushs "O resultado de 2 + 2 * 3 * 2 é: "
writes
pushg 0
writei
writeln
stop
```

6. Interface Web

Com o objetivo de tornar a utilização do compilador mais amigável e intuitiva, desenvolvemos uma **interface *web***, que permite ao utilizador **correr o programa** diretamente a partir do ***browser***.

Através desta interface, é possível selecionar facilmente um **ficheiro `.pas`**. Após a seleção, é possível gerar o código `VM` do programa, assim como analisar a sequência de `tokens` gerada, a `AST` gerada e ainda a validação da análise semântica e sintática.

Figura 3: Interface Web

- **Sequência de tokens gerados**

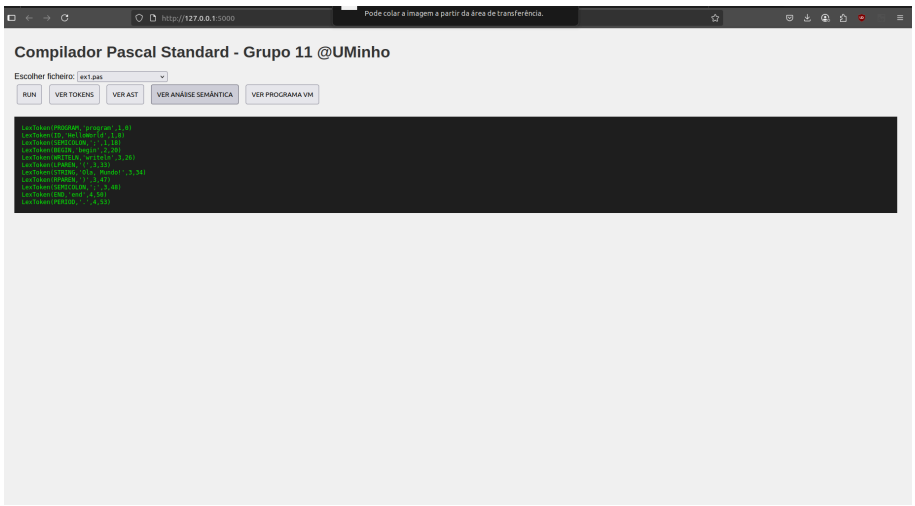

Figura 4: Sequência de tokens gerados pelo programa Pascal

- **Validação da análise sintática e AST gerada**

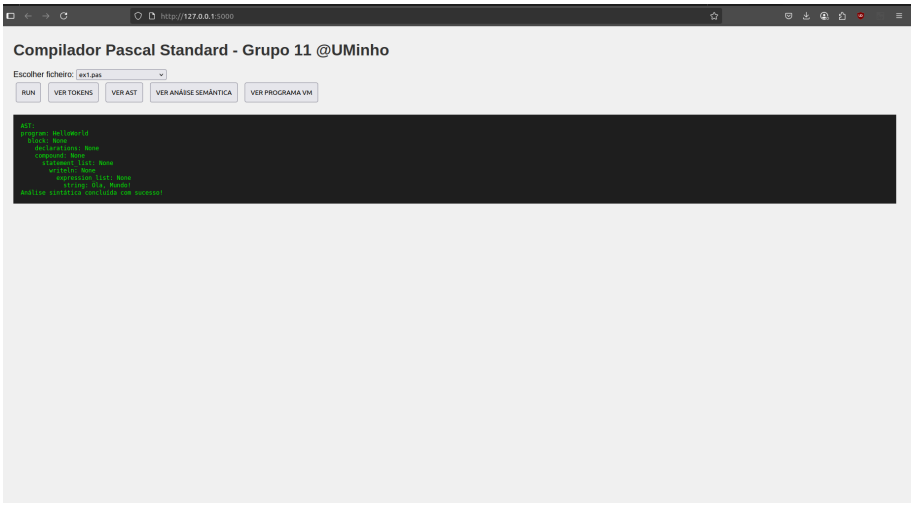

Figura 5: Visualização da AST após análise sintática

- **Validação da análise semântica**

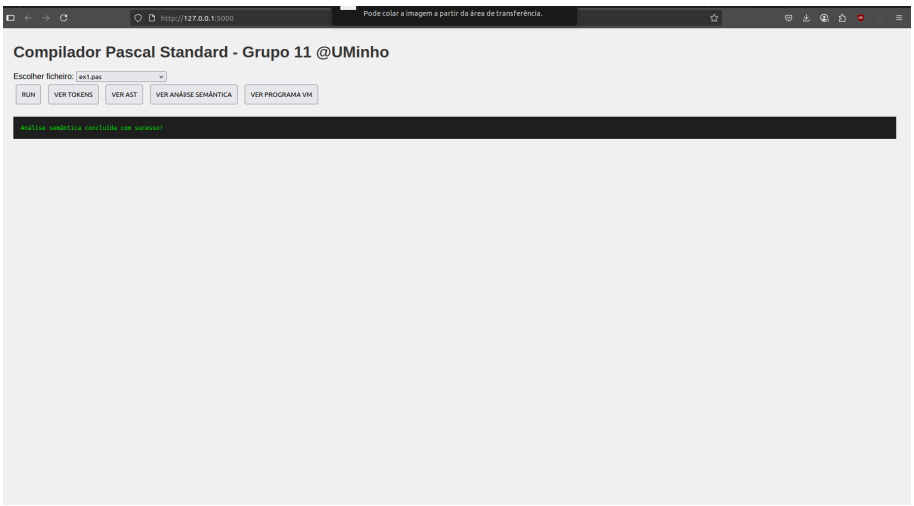

Figura 6: Validação da análise semântica do programa Pascal

7. Conclusão

Dada por concluída a realização deste trabalho prático, consideramos que o desenvolvimento do compilador para Pascal Standard foi essencial para consolidar os conhecimentos adquiridos na unidade curricular de Processamento de Linguagens. Ao longo do projeto, implementámos todas as fases de um compilador, da análise léxica à execução em máquina virtual, enfrentando desafios como a gestão de escopos, a geração de código para `arrays` e ciclos `for e while`, e a adaptação à `VM` proposta.

Apesar das dificuldades, conseguimos superá-las com sucesso, integrando todas as funcionalidades exigidas e extras sugeridos (à exceção do extra 'programas com `functions` e `procedures`'), o que resultou num compilador robusto e funcional. Esta experiência permitiu-nos aprofundar a compreensão dos mecanismos internos das linguagens de programação e reforçou a nossa capacidade de análise e desenvolvimento de soluções complexas. De maneira geral, ficamos satisfeitos com o resultado final obtido.

Referências - APA

1 Pascal Tokens. (s.d.). Free Pascal - Advanced open source Pascal compiler for Pascal and Object Pascal - Home Page. https://www.freepascal.org/docs-html/ref/refch1.html

EWVM. (2025). Uminho.pt. https://ewvm.epl.di.uminho.pt/

EWVM-Manual. (2025). Uminho.pt. https://ewvm.epl.di.uminho.pt/manual

EWVM-Examples. (2025). Uminho.pt. https://ewvm.epl.di.uminho.pt/examples

Standard Pascal (2025). Freepascal.org. https://wiki.freepascal.org/Standard_Pascal