Trabalho Prático | Processamento de Linguagens
Grupo 25 | 2024/2025

Afonso Santos
(A104276)

João Lobo
(A104356)

Rita Camacho
(A104439)

Índice

1. Introdução

O presente documento apresenta informações relativas ao **Trabalho Prático** da Unidade Curricular **Processamento de Linguagens**, pertencente ao 2.º Semestre do 3.º Ano da Licenciatura em Engenharia Informática, realizada no ano letivo 2024/2025, na Universidade do Minho.
O projeto focou-se na criação de um compilador para a linguagem `Pascal Standard`, com suporte à declaração de variáveis, avaliação de expressões aritméticas, comandos de controlo de fluxo (`if`, `while` e `for`) e, opcionalmente, à implementação de subprogramas (`procedures` e `functions`).

O principal objetivo consistiu em desenvolver uma pipeline de compilação composta pelas seguintes fases:

- Análise Léxica;
- Análise Sintática;
- Análise Semântica;
- Geração de código máquina direcionado para uma máquina virtual disponibilizada aos alunos.

Ao longo deste relatório serão descritas detalhadamente as funcionalidades implementadas pelo grupo, bem como as decisões técnicas tomadas, os desafios encontrados e as soluções adotadas. Sempre que pertinente, serão apresentados exemplos de código, diagramas e testes que ilustram o funcionamento e a validação do compilador desenvolvido.

2. Pascal Standard

Após este primeiro contacto, o grupo avançou para o estudo da linguagem Pascal, linguagem alvo do compilador a ser desenvolvido.

Como Pascal é uma linguagem do paradigma imperativo, com o qual o grupo já possuía experiência prévia, a adaptação foi relativamente fluída. Este paradigma assenta na execução sequencial de instruções, no uso explícito de variáveis, estruturas de controlo de fluxo (como `if`, `while` e `for`), bem como na definição de subprogramas (`procedures` e `functions`).

Inicialmente, concentramo-nos em compreender a sintaxe e semântica básica de Pascal, através da implementação de programas simples que incluíam declarações de variáveis, operações aritméticas e estruturas de controlo. Um dos primeiros exercícios consistiu na criação de um programa que somasse dois números inteiros fornecidos pelo utilizador e imprimisse o resultado no ecrã.

Exemplo de código Pascal:

```
program SomaSimples;
var
  a, b, resultado: integer;
begin
  a := 10;
  b := 20;
  resultado := a + b;
  writeln('Resultado: ', resultado);
end.
```

Este tipo de exercício permitiu não só consolidar a compreensão da linguagem, como também delinear de forma prática os componentes necessários para o futuro compilador: o reconhecimento de *tokens* léxicos (como a, :=, +), a estrutura gramatical das expressões e a geração do código equivalente na máquina virtual.

3. Tokenizer

Após a análise da linguagem Pascal e a familiarização com a sua sintaxe e estrutura, iniciou-se a construção do analisador léxico (*tokenizer*), utilizamos a biblioteca `ply.lex` (Python Lex-Yacc). Esta fase transforma o código-fonte, composto por uma sequência de caracteres, numa sequência de *tokens* — unidades léxicas que serão posteriormente interpretadas pelo analisador sintático.

A construção do *tokenizer* foi baseada em expressões regulares que permitem reconhecer os diferentes elementos da linguagem Pascal, tais como:

- Identificadores (ex.: nomes de variáveis e funções);
- Números inteiros e reais;

- Literais de *string*;
- Operadores aritméticos e relacionais;
- Delimitadores (parênteses, vírgulas, pontos e vírgulas, etc.);
- Palavras-chave reservadas da linguagem (ex.: `begin`, `end`, `if`, `while`, `procedure`, entre outros).

O processo de desenvolvimento envolveu a criação de uma lista de *tokens* e a definição das suas respetivas regras léxicas, organizadas da seguinte forma:

- **Tokens simples**, como `PLUS`, `MINUS`, `EQUAL`, foram definidos diretamente através de expressões regulares atribuídas a variáveis iniciadas por `t_`;
- **Palavras-chave reservadas** foram implementadas como funções `t_NOME`, cada uma contendo a expressão regular correspondente. Estas funções devolvem o *token* apropriado com o seu nome respetivo.
- O tratamento de números incluiu dois tipos distintos:
  - **NUM_INT** para inteiros simples;
  - **NUM_REAL** para valores com parte decimal e notação científica.
- **Comentários**, tanto no estilo `{...}` como `(*...*)`, foram ignorados no processamento e não geram *tokens*. Contudo, o contador de linhas (`t.lexer.lineno`) é atualizado para manter a rastreabilidade correta.

Também foram tratadas situações de erro léxico, com a função **`t_error`,** que imprime uma mensagem informativa e ignora o caractere inválido.

O resultado desta parte do trabalho foi um *lexer*, capaz de reconhecer a grande maioria dos elementos da linguagem `Pascal Standard`, servindo de base sólida para o desenvolvimento das próximas fases do compilador, nomeadamente a análise sintática com `Yacc`.

4. Yaccer

Para interpretar os *tokens* gerados na fase anterior (análise léxica), desenvolvemos uma gramática com o auxílio da biblioteca `ply.yacc` (Python Yacc). Esta etapa corresponde à análise sintática do compilador, cuja principal função é verificar se a sequência de *tokens* respeita as regras gramaticais da linguagem Pascal.

Na definição da gramática, procurámos realizá-la da forma mais específica e rigorosa possível, de modo a garantir que apenas programas válidos, segundo a sintaxe da linguagem Pascal, fossem aceites. Uma gramática bem definida permite:

- Detetar erros de estrutura no código-fonte;
- Organizar os *tokens* numa árvore sintática abstrata (AST);
- Facilitar as fases seguintes do compilador, nomeadamente a análise semântica e a geração de código.

4.1. Representação da Estrutura Intermédia (AST)

Para representar as estruturas intermédias da linguagem (isto é, a árvore sintática abstrata, ou AST), optámos por criar um conjunto de classes específicas, todas elas subclasses de uma classe abstrata comum denominada Node. Esta abordagem impõe que cada subclasse implemente obrigatoriamente um conjunto de métodos essenciais, tais como:

- Validação semântica dos seus conteúdos;
- Comparação de igualdade entre nós;
- Conversão para código de máquina, através da nossa linguagem-alvo (máquina virtual);
- (Opcionalmente) Conversão para outras linguagens, como Python ou pseudocódigo, tornando o sistema altamente extensível.

Cada nó da árvore (por exemplo, uma operação aritmética, uma declaração de variável, uma chamada de procedimento, etc.) é assim representado por uma classe autónoma, contendo os dados relevantes e a lógica associada ao seu comportamento. Esta organização modular permite manter o código limpo, testável e escalável.

Embora exista uma correspondência natural entre algumas destas classes e regras da gramática definida no *parser*, procurámos evitar um acoplamento direto. Ou seja, a AST foi concebida como uma estrutura mais abstrata, que representa o significado do programa, e não apenas a sua forma textual.

As classes encontram-se num módulo próprio (`language/`), promovendo a separação de responsabilidades no projeto e facilitando a reutilização de componentes. Esta arquitetura, orientada a objetos, oferece também a flexibilidade de adaptar o compilador a outras linguagens de destino, bastando implementar novas variantes dos métodos de conversão em cada nó da árvore.

4.2. Contexto de Execução (`Context`)

Para assegurar uma gestão fluída e consistente dos escopos (blocos de código onde variáveis e funções são visíveis), e para lidar de forma eficaz com a tradução de variáveis e funções na árvore sintática, desenvolvemos uma classe estratégica denominada `Context`. Esta classe atua como um "ambiente de execução" dinâmico, centralizando todas as informações contextuais cruciais durante as fases de análise e geração de código.

Cada objeto `Context` encapsula os seguintes elementos essenciais:

- **Identificador da Função Corrente**: Regista o nome da função em que a análise se encontra, se aplicável;
- **Parâmetro da Função Ativa**: Armazena o nome do argumento que está a ser processado, caso exista;
- **Tabela de Símbolos:** Uma estrutura de mapeamento vital que associa as variáveis já declaradas aos seus nomes correspondentes na linguagem de saída (por exemplo, na máquina virtual ou no código Python gerado);
- **Gerador de Nomes Temporários:** Um contador que facilita a criação de nomes únicos para variáveis auxiliares internas, prevenindo conflitos de nomenclatura;
- **Endereços de Memória:** Informação sobre a alocação de memória para as entidades.

Esta tabela de símbolos é particularmente importante, pois permite ao compilador manter um registo preciso dos identificadores usados internamente, evitando sobreposições de nomes, especialmente para variáveis intermédias ou aquelas geradas automaticamente durante o processo de compilação.

A operação da classe `Context` mimetiza a funcionalidade de uma estrutura de pilha (`stack`). Sempre que o compilador transita para um novo escopo – seja uma função, um laço de repetição ou uma instrução condicional – uma nova instância de `Context` é instanciada. Esta nova instância herda o estado da instância anterior, mas permite que as definições e alterações subsequentes sejam locais a esse novo escopo. Ao concluir a análise desse escopo, a instância de `Context` associada é simplesmente descartada, e o compilador retoma automaticamente o contexto que estava ativo previamente.

Esta metodologia foi escolhida pela sua simplicidade e robustez, eliminando a necessidade de construir mecanismos complexos para "retroceder" entre escopos. Em vez disso, basta criar um novo contexto quando necessário e abandoná-lo quando a sua utilidade termina. Esta abordagem contribui significativamente para a modularidade do compilador e simplifica consideravelmente o processo de geração de código.

5. Implementação

5.1. Declaração de Variáveis

A linguagem Pascal, para além de ser fortemente estruturada por blocos, apresenta uma natureza declarativa bastante expressiva. Isso implica que, sempre que um novo bloco de declarações é introduzido, as variáveis nele definidas — juntamente com os seus respetivos tipos — são imediatamente registadas no contexto ativo, conforme descrito anteriormente. Esta abordagem garante que todas as entidades declaradas estejam devidamente acessíveis dentro do seu escopo.

```
variableDeclarationBlock : VAR variableDeclarationList SEMI
```

A regra acima é responsável por reconhecer blocos de declaração iniciados pela palavra-chave VAR, seguidos por uma lista de declarações. Quando esse padrão é identificado, um novo objeto `VariableDeclarationBlock` é criado com base na lista de declarações analisadas, que é construída recursivamente pela regra seguinte:

```
variableDeclarationList : variableDeclarationList SEMI
variableDeclaration

                        |variableDeclaration
```

Esta regra permite acumular múltiplas declarações separadas por ponto e vírgula, tratando cada declaração individual de forma incremental. A própria declaração de variáveis é então interpretada pela regra:

```
variableDeclaration : identifierList COLON type\_
```

Esta regra permite identificar as várias variáveis e os respetivos tipos. Ou seja, todas as variáveis listadas serão de um tipo específico. Com esta regra, é possível definir essa característica de forma clara e armazená-la no contexto da aplicação, garantindo coerência e consistência no tratamento dos dados.

De forma semelhante, quando são encontradas declarações dentro de estruturas de controlo, como laços de repetição (`for`, `while`, etc.), ou condicionais, o compilador cria automaticamente um novo objeto `Context`. Este novo contexto herda integralmente o estado do contexto pai, assegurando a visibilidade das definições anteriores, mas permite a introdução de novas variáveis locais sem interferir com o escopo global. As variáveis declaradas nesses blocos são adicionadas à nova instância de `Context` de forma transparente, mantendo a organização e isolamento típicos de uma arquitetura em pilha.

INPUT:

```
program ExemploVariaveis;

var
  nome: string;
  idade: integer;
  altura: real;

begin
  { Atribuindo valores às variáveis }
  nome := 'Ana';
  idade := 25;
  altura:= 1.70;


  { Exibindo os valores das variáveis }
  writeln('Nome: ', nome);
  writeln('Idade: ', idade);
  writeln('Altura:', altura);

end.
```

OUTPUT:

```
      START
PUSHS "Ana"
STOREG 0
PUSHI 25
STOREG 1
PUSHF 1.7
STOREG 2
PUSHS "Nome: "
WRITES
PUSHG 0  // Load nome
WRITES
WRITELN
PUSHS "Idade: "
WRITES
PUSHG 1  // Load idade
WRITEI
WRITELN
PUSHS "Altura:"
WRITES
PUSHG 2  // Load altura
WRITEF
WRITELN
STOP
```

5.2. Condicionais

As estruturas condicionais em Pascal são notoriamente simples e diretas, sendo reconhecidas já na fase de análise sintática através de regras específicas. A linguagem permite duas formas principais para a construção de instruções `if`: com ou sem a cláusula `else`. Ambas são tratadas na gramática por meio da seguinte regra:

```
ifStatement : IF expression THEN statement
            | IF expression THEN statement ELSE statement
```

Esta definição contempla as duas variantes sintáticas possíveis:

1. Condicional simples (`if ... then ...`)
2. Condicional composta (`if ... then ... else ...`)

Durante o processo de *parsing*, ao reconhecer a palavra-chave `IF`, a análise prossegue com a leitura de uma expressão booleana (avaliada como condição) seguida por um bloco de instrução associado à cláusula `THEN`. Opcionalmente, um segundo bloco pode ser associado à cláusula `ELSE`, indicando a execução alternativa caso a condição seja falsa.

A representação interna dessas estruturas é normalmente encapsulada em objetos como `IfStatement`, que armazenam:

- A expressão condicional (campo `expression`),
- A instrução ou bloco a ser executado quando a condição for verdadeira (`then_body`),
- E, caso aplicável, a instrução alternativa para o ramo *else* (`else_body`).

INPUT:

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

 Write('Introduza o segundo
número: ');
 ReadLn(num2);

 Write('Introduza o terceiro
número: ');
 ReadLn(num3);

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

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

OUTPUT:

```
      START
PUSHS "Introduza o primeiro
número: "
WRITES
WRITELN
READ
ATOI
STOREG 0
PUSHS "Introduza o segundo número:
"
WRITES
WRITELN
READ
ATOI
STOREG 1
PUSHS "Introduza o terceiro
número: "
WRITES
WRITELN
READ
ATOI
STOREG 2
PUSHG 0  // Load num1
PUSHG 1  // Load num2
SUP
JZ ELSE1
PUSHG 0  // Load num1
PUSHG 2  // Load num3
SUP
JZ ELSE2
PUSHG 0  // Load num1
STOREG 3
JUMP ENDIF2
ELSE2:
PUSHG 2  // Load num3
STOREG 3
ENDIF2:
JUMP ENDIF1
ELSE1:
PUSHG 1  // Load num2
PUSHG 2  // Load num3
SUP
```

INPUT: **OUTPUT:**

```
JZ ELSE3
PUSHG 1  // Load num2
STOREG 3
JUMP ENDIF3
ELSE3:
PUSHG 2  // Load num3
STOREG 3
ENDIF3:
ENDIF1:
PUSHS "O maior é: "
WRITES
PUSHG 3  // Load maior
WRITEI
WRITELN
STOP
```

5.3. Ciclos

No desenvolvimento do analisador sintático para a linguagem Pascal, tivemos o cuidado de implementar suporte completo para os três principais tipos de ciclos previstos pela linguagem: `for`, `while` e `repeat ... until`. Cada um desses ciclos foi devidamente identificado na gramática através da seguinte regra geral:

```
loopStatement : forStatement
              | whileStatement
              | repeatStatement
```

Esta regra atua como um ponto de entrada comum para instruções de repetição, delegando a análise específica de cada estrutura a sub-regras dedicadas.

5.3.1. Ciclo `for`

A estrutura do ciclo `for` é reconhecida por:

```
forStatement : FOR identifier ASSIGN Init_Final TO Init_Final DO statement
        | FOR identifier ASSIGN Init_Final DOWNTO Init_Final DO statement
```

Esta regra cobre as duas formas sintáticas do `for` em Pascal: com incremento (`TO`) e com decremento (`DOWNTO`). O nó `ForStatementNode` armazena o identificador de controlo, os valores inicial e final, o corpo do laço e a direção da iteração.

5.3.2. Ciclo `while`

A estrutura condicional de repetição `while` é tratada por:

```
whileStatement : WHILE expression DO statement
```

Este ciclo executa o corpo enquanto a expressão condicional se mantiver verdadeira. A regra cria um nó `WhileStatementNode`, armazenando a condição de continuidade e o bloco a ser executado.

5.3.3. Ciclo `repeat ... until`

Por fim, também incluímos suporte ao ciclo `repeat ... until`, conforme a gramática Pascal tradicional:

```
repeatStatement : REPEAT statements UNTIL expression
```

Este ciclo executa repetidamente o bloco de instruções antes de avaliar a condição (semelhante ao `do ... while` em outras linguagens). O nó gerado armazena tanto o corpo do laço quanto a expressão que determina a sua finalização.

INPUT:

```
 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.
```

OUTPUT:

```
      START
PUSHS "Introduza um número inteiro
positivo:"
WRITES
WRITELN
READ
ATOI
STOREG 0
PUSHI 1
STOREG 1
PUSHI 1
STOREG 2
loop1:
PUSHG 2
PUSHG 0  // Load n
INFEQ
JZ endloop1
PUSHG 1  // Load fat
PUSHG 2  // Load i
MUL
STOREG 1
PUSHG 2
PUSHI 1
ADD
STOREG 2
JUMP loop1
endloop1:
PUSHS "Fatorial de "
WRITES
PUSHG 0  // Load n
WRITEI
PUSHS ": "
WRITES
PUSHG 1  // Load fat
```

INPUT: **OUTPUT:**

```
WRITEI
WRITELN
STOP
```

Embora tenhamos implementado suporte gramatical e estrutural para os três tipos mencionados, apenas os ciclos `for` e `while` foram traduzidos para a linguagem de máquina, conforme exigido no enunciado do projeto. Ainda assim, o modelo seguido para a construção dos nós de AST permite que a tradução do ciclo `repeat ... until` fosse implementada com a mesma lógica dos demais, respeitando o fluxo de controlo e a estrutura do contexto correspondente.

5.4. `Functions` & `Procedures`

No desenvolvimento do analisador sintático para a linguagem Pascal, é fundamental reconhecer e tratar corretamente as declarações de `procedures` e `functions`, que são estruturas essenciais para organizar e modularizar o código.

Em Pascal, as `procedures` e `functions` são declaradas geralmente no início do programa ou dentro de outros blocos, permitindo assim uma organização hierárquica semelhante a uma estrutura de programas aninhados. Isto ocorre porque cada bloco pode conter suas próprias declarações locais, funcionando quase como um programa interno dentro de outro.

A gramática para a declaração de `procedures` e `functions` pode ser definida da seguinte forma:

```
procedureAndFunctionDeclarationBlock : procedureDeclaration SEMI
                                     | functionDeclaration SEMI
```

Ou seja, uma `procedure` inicia com a palavra reservada `PROCEDURE`, seguida do seu nome (identificador), opcionalmente uma lista de parâmetros formais, um ponto e vírgula (;) e, finalmente, o bloco que define o corpo da rotina. Já uma função inicia com `FUNCTION`, seguida do nome, parâmetros opcionais, o tipo de resultado (após os dois pontos :), ponto e vírgula, e o bloco com o seu corpo.

```
procedureDeclaration : PROCEDURE identifier SEMI block
| PROCEDURE identifier formalParameterList_opt SEMI block
      functionDeclaration : FUNCTION identifier
|formalParameterList_opt COLON resultType SEMI block
```

Ao identificar o nome da `procedure` ou `function`, o analisador deve armazenar essa informação no contexto semântico, classificando a entrada como `procedure` ou `function`. O bloco associado — ou seja, o corpo da rotina — é guardado para análise e tradução

posterior, respeitando as regras de escopo e visibilidade das variáveis locais declaradas dentro dele.

Dessa forma, gramaticalmente, podemos imaginar uma estrutura aninhada, onde cada `procedure` ou `function` representa um programa "interno", contendo as suas próprias declarações e instruções. Esta organização facilita a modularização do código, melhora a clareza e permite a reutilização eficiente dos blocos de código.

INPUT:

```
program CafeLoop;

function PrepararParaProgramar:
string;
var
  cafezinhos: integer;
begin
  cafezinhos := 0;

  writeln('Preparando o cérebro
para codar...');

  while cafezinhos < 3 do
  begin
    writeln('Bebendo café número
', cafezinhos + 1, '...');
    cafezinhos := cafezinhos + 1;
  end;

  PrepararParaProgramar :=
'Pronto! Modo programador ativado
com sucesso!';
end;

begin
  writeln('Bom dia,
programador!');
  writeln(PrepararParaProgramar);
  writeln('Agora sim, pode abrir o
compilador.');
end.
```

OUTPUT:

```
      START
PUSHS "Bom dia, programador!"
WRITES
WRITELN
PUSHA funcPrepararParaProgramar
CALL
WRITEI
WRITELN
PUSHS "Agora sim, pode abrir o
compilador."
WRITES
WRITELN
STOP
funcPrepararParaProgramar:
PUSHI 0
STOREG 0
PUSHS "Preparando o cérebro para
codar..."
WRITES
WRITELN
WHILE1:
PUSHG 0  // Load cafezinhos
PUSHI 3
INF
JZ ENDWHILE1
PUSHS "Bebendo café número "
WRITES
PUSHG 0  // Load cafezinhos
PUSHI 1
ADD
PUSHS "..."
WRITES
WRITELN
PUSHG 0  // Load cafezinhos
PUSHI 1
ADD
STOREG 0
JUMP WHILE1
ENDWHILE1:
```

INPUT:

OUTPUT:

```
PUSHS "Pronto! Modo programador
ativado com sucesso!"
STOREG 1
RETURN
```

6. Conclusão

Com o desenvolvimento deste projeto, o grupo sente-se bastante realizado com as estruturas e a solução que conseguiu implementar. Um dos principais pontos fortes do trabalho foi a criação das estruturas intermédias, que se revelaram uma abordagem extremamente versátil e útil. Estas estruturas permitiram-nos modelar de forma eficaz os diferentes comportamentos dos vários tipos de nós da árvore sintática, facilitando não só a adaptação a diferentes construções da linguagem, mas também tornando o sistema mais flexível para suportar diferentes linguagens de programação no futuro.

Este desenho modular teve um impacto particularmente positivo nas fases de análise semântica e geração de código, onde foi possível reutilizar e adaptar partes da estrutura consoante o contexto. A utilização da classe `Context` foi crucial neste processo, pois funcionou como um repositório organizado de informações relevantes ao longo da análise – nomeadamente a tabela de símbolos, escopos, tipos de variáveis e validação de regras semânticas. Esta abstração facilitou o rastreamento de declarações e usos de variáveis, garantindo coerência e integridade semântica durante a compilação.

Foram cumpridos todos os requisitos apresentados no enunciado, nomeadamente:

- A declaração e verificação de variáveis e tipos;
- O suporte a expressões aritméticas e lógicas;
- A implementação de comandos de controlo de fluxo como `if`, `while` e `for`;
- E, como opcional, o suporte a subprogramas (`procedure` e `function`) que, apesar de mais complexos, foram incorporados com sucesso na gramática e no fluxo semântico.

Além do desenvolvimento técnico, este projeto permitiu-nos aprofundar significativamente o nosso entendimento sobre o funcionamento de compiladores, a importância das gramáticas formais e o papel fundamental da análise sintática e semântica no processamento de linguagens de programação.

A escolha da linguagem Pascal também contribuiu positivamente para o projeto. Por ser uma linguagem estruturada, expressiva e com uma sintaxe clara, permitiu-nos identificar padrões e delimitações de blocos com relativa facilidade. Isto refletiu-se na clareza da nossa gramática, desenhada de forma a capturar tais regras de forma natural e intuitiva.

Em suma, este projeto não só reforçou os nossos conhecimentos teóricos sobre compiladores, como também nos deu uma experiência prática valiosa na construção de uma ferramenta funcional, a partir de uma base sólida tanto a nível estrutural como semântico.