

# Processamento de Linguagens e Compiladores - Trabalho Prático

**Autores.** - *Cláudio Rafael Oliveira Ferreira* (A108577) - *Marco António Ferreira Abreu* (A108578)  
- *Nelson Daniel Araújo Sousa* (A109068)

**Linguagem:** Python

**Estrutura:** 1) Introdução e Arquitetura do Sistema 2) Análise Léxica 3) Análise Sintática 4) Análise Semântica e Gestão de Contexto 5) Geração de Código 6) Testes e Resultados 7) Conclusão e Dificuldades Superadas

## 1 Introdução e Arquitetura do Sistema

O objetivo deste projeto é o desenvolvimento de um compilador robusto para a linguagem **Pascal Standard**. O sistema foi desenhado para realizar a tradução completa de programas fonte para uma representação de baixo nível, especificamente para o conjunto de instruções da **Máquina Virtual (VM)** de stack disponibilizada.

### 1.1 Metodologia de Desenvolvimento

A implementação baseia-se num pipeline de múltiplas passagens (multi-pass compiler), o que garante uma separação clara entre a análise linguística e a síntese de código. Esta abordagem facilita a depuração e permite que cada fase do compilador valide a integridade dos dados antes de os passar à etapa seguinte.

### 1.2 Estrutura Modular e Organização

O projeto foi organizado de forma estrita para garantir escalabilidade e permitir a automação de testes. A estrutura de pastas reflete esta organização:

- **main.py:** Interface de linha de comando (CLI) que serve como ponto de entrada para a compilação de ficheiros individuais.
- **/src:** Núcleo do compilador, onde reside a lógica dividida por módulos:
  - **pascal\_analex.py & parser.py:** Responsáveis pelo **Front-end** (Análise Léxica e Sintática).
  - **sem.py & context.py:** Responsáveis pelo **Middle-end** (Análise Semântica, Gestão de Escopos e Tabelas de Símbolos).
  - **codegen.py:** Responsável pelo **Back-end** (Geração de código para a VM).
- **/tests:** Infraestrutura avançada de testes automatizados:
  - **run\_tests.py:** Motor de execução que automatiza a validação de casos de sucesso e de erro.
  - **/cases:** Subdividida em **ok/** (programas válidos) e **error/** (programas com erros propositados para teste de robustez).
  - **/manifests:** Contém o ficheiro **error\_cases.json**, que define as mensagens de erro esperadas para cada teste negativo. **/out\_vm:** Diretoria de destino para os ficheiros de assembly (**.vm**) resultantes da compilação.

### 1.3 Fluxo de Trabalho (Pipeline)

O processo de compilação segue um fluxo sequencial e rigoroso: 1) **Leitura e Tokenização:** O ficheiro `.pas` é lido e o Lexer converte o texto num fluxo de tokens. 2) **Análise e Validação:** O Parser valida a estrutura gramatical e invoca o Analisador Semântico para verificar tipos e declarações. 3) **Gestão de Contexto:** Durante a análise, o `CompilerContext` gere a alocação de memória (endereços globais e locais) e a visibilidade de variáveis. 4) **Emissão de Código:** Em caso de sucesso, o código assembly é gerado, otimizando a stack discipline (especialmente em subprogramas). 5) **Interrupção por Erro:** Caso ocorra uma falha em qualquer fase, o compilador interrompe imediatamente o processo, emitindo uma mensagem de erro informativa (ex: `SemanticError` ou `SyntaxParseError`) que indica a linha e a causa do problema.

## 2 Análise Léxica (`src/pascal_analex.py`)

A análise léxica é a primeira etapa do processo de compilação. O seu objetivo é ler o fluxo de caracteres do código-fonte e agrupá-los em unidades significativas chamadas **tokens**.

### 2.1 Ferramentas e Configuração

Para esta fase, utilizámos a biblioteca `ply.lex`. A configuração baseia-se na definição de uma lista de tokens e na utilização de expressões regulares para descrever o padrão de cada um deles. O analisador foi desenhado para ser **case-insensitive** no que toca a identificadores e palavras reservadas, respeitando a especificação do Pascal Standard (Ver: `src/pascal_analex.py`).

### 2.2 Gestão de Palavras Reservadas

Para evitar que o compilador confunda identificadores (nomes de variáveis) com comandos da linguagem, utilizámos um dicionário de mapeamento chamado `reserved`. - **Exemplos de Palavras-Chave:** PROGRAM, BEGIN, END, VAR, IF, THEN, ELSE, WHILE, FOR, FUNCTION, PROCEDURE, ARRAY, OF, entre outras. - **Lógica de Identificação:** Quando o lexer encontra uma sequência de letras, verifica primeiro se ela existe no dicionário `reserved`. Se existir, atribui o token da palavra-chave; caso contrário, classifica-o como um identificador (ID).

### 2.3 Expressões Regulares e Tokens

Os tokens foram divididos em duas categorias de definição: 1) **Tokens Simples:** Definidos como strings diretas para operadores e delimitadores. Exemplos: `t_PLUS = r'+'`, `t_ASSIGN = r':='`, `t_SEMICOLON = r';'`. 2) **Tokens Complexos:** Definidos através de funções para permitir lógica adicional ou capturar valores. - **Números:** Distinguimos entre `NUMBER_INT` e `NUMBER_REAL` através de padrões que procuram a presença do ponto decimal. - **Strings:** O token `STRING_LITERAL` captura texto entre plicas (`'...'`), tratando corretamente sequências de caracteres.

### 2.4 Tratamento de “Ruído” e Erros

- **Espaços e Comentários:** O compilador ignora espaços em branco e tabulações através da variável `t_ignore`. Além disso, implementámos o tratamento de comentários delimitados por chavetas `{ ... }`, garantindo que estas informações não cheguem ao analisador sintático.

- **Rastreamento de Linhas:** A função `t_newline` incrementa o contador de linhas sempre que encontra um caractere de nova linha (`***`), o que permite emitir mensagens de erro precisas com a localização exata no código.
- **Erros Léxicos:** A função `t_error` captura qualquer caractere que não corresponda a nenhuma regra definida, interrompendo a compilação e informando o utilizador sobre o caractere ilegal detetado.

### 3 Análise Sintática (src/parser.py)

A análise sintática é a fase onde o compilador verifica se a sequência de tokens fornecida pelo analisador léxico respeita as regras gramaticais da linguagem Pascal Standard.

#### 3.1 Metodologia e Ferramentas

Utilizamos o **ply.yacc**, que implementa um algoritmo de análise **LALR(1)**. A gramática foi definida através de uma série de funções Python cujas docstrings contêm as regras de produção na forma de **BNF**.

#### 3.2 Estrutura da Gramática

A gramática foi desenhada para cobrir a estrutura hierárquica do Pascal: - **Raiz do Programa:** A regra `p_program` define a estrutura global: o cabeçalho (`program ID;`), as declarações e o bloco principal de comandos `BEGIN ... END.` - **Declarações:** Implementámos regras para processar a secção `VAR`, permitindo a declaração de tipos simples (`integer`, `real`, `boolean`) e tipos estruturados (`array`). - **Subprogramas:** O parser suporta a definição de `procedure` e `function`, gerindo corretamente a assinatura (parâmetros) e o bloco local de cada subprograma.

#### 3.3 Tratamento de Expressões e Precedência

Para garantir que operações como `a + b * c` são calculadas corretamente (a multiplicação antes da soma), definimos níveis de precedência explicitamente no objeto **precedence**. Isto resolve ambiguidades gramaticais sem a necessidade de criar múltiplas sub-regras complexas para cada operador.

#### 3.4 Atributos Sintáticos e Ações

Em cada regra gramatical, o analisador executa uma “ação”. No nosso projeto: - **Construção de Expressões:** Cada expressão (`expr`) retorna um dicionário contendo o seu **tipo**, se é uma **constante** e o **código gerado** até ao momento. - **Comandos de Controlo:** Regras como `p_statement_if` ou `p_statement_while` não apenas validam a sintaxe, mas coordenam a criação de etiquetas de salto (`labels`) necessárias para o fluxo de execução na Máquina Virtual.

#### 3.5 Recuperação de Erros Sintáticos

Implementámos a função `p_error`, que é disparada quando o parser encontra um token inesperado. Em vez de simplesmente terminar, o compilador levanta uma exceção **SyntaxParseError**, indicando ao utilizador a linha e o token onde a gramática foi violada, facilitando a correção do código-fonte.

### 3.6 Gramática

```
Rule 0      S' -> programa
Rule 1      programa -> PROGRAM ID SEMICOLON bloco DOT
Rule 2      bloco -> decls compound_stmt
Rule 3      decls -> decl decls
Rule 4      decls -> <empty>
Rule 5      decl -> var_section
Rule 6      decl -> subprog_decl
Rule 7      var_section -> VAR var_decl_list
Rule 8      var_decl_list -> var_decl var_decl_list_tail
Rule 9      var_decl_list_tail -> var_decl var_decl_list_tail
Rule 10     var_decl_list_tail -> <empty>
Rule 11     var_decl -> id_list COLON tipo SEMICOLON
Rule 12     id_list -> ID id_list_tail
Rule 13     id_list_tail -> COMMA ID id_list_tail
Rule 14     id_list_tail -> <empty>
Rule 15     tipo -> INTEGER
Rule 16     tipo -> REAL
Rule 17     tipo -> BOOLEAN
Rule 18     tipo -> CHAR
Rule 19     tipo -> STRING
Rule 20     tipo -> array_type
Rule 21     array_type -> ARRAY LBRACKET range RBRACKET OF tipo
Rule 22     range -> NUMBER_INT RANGE NUMBER_INT
Rule 23     subprog_decl -> function_decl
Rule 24     subprog_decl -> procedure_decl
Rule 25     function_header -> FUNCTION ID LPAREN param_list_opt RPAREN COLON tipo SEMICOLON
Rule 26     func_enter -> <empty>
Rule 27     function_decl -> function_header func_enter bloco SEMICOLON
Rule 28     procedure_header -> PROCEDURE ID LPAREN param_list_opt RPAREN SEMICOLON
Rule 29     proc_enter -> <empty>
Rule 30     procedure_decl -> procedure_header proc_enter bloco SEMICOLON
Rule 31     param_list_opt -> param_list
Rule 32     param_list_opt -> <empty>
Rule 33     param_list -> param param_list_tail
Rule 34     param_list_tail -> SEMICOLON param param_list_tail
Rule 35     param_list_tail -> <empty>
Rule 36     param -> id_list COLON tipo
Rule 37     compound_stmt -> BEGIN stmt_list_opt END
Rule 38     stmt_list_opt -> stmt_list
Rule 39     stmt_list_opt -> <empty>
Rule 40     stmt_list -> stmt stmt_list_tail
Rule 41     stmt_list_tail -> SEMICOLON stmt stmt_list_tail
Rule 42     stmt_list_tail -> SEMICOLON
Rule 43     stmt_list_tail -> <empty>
Rule 44     stmt -> assign_stmt
Rule 45     stmt -> if_stmt
```

Rule 46 stmt -> while\_stmt  
 Rule 47 stmt -> for\_stmt  
 Rule 48 stmt -> repeat\_stmt  
 Rule 49 stmt -> compound\_stmt  
 Rule 50 stmt -> proc\_call  
 Rule 51 assign\_stmt -> lvalue ASSIGN expr  
 Rule 52 var\_ref -> ID  
 Rule 53 var\_ref -> ID LBRACKET expr RBRACKET  
 Rule 54 lvalue -> ID  
 Rule 55 lvalue -> ID LBRACKET expr RBRACKET  
 Rule 56 if\_stmt -> IF expr THEN stmt  
 Rule 57 if\_stmt -> IF expr THEN stmt ELSE stmt  
 Rule 58 while\_stmt -> WHILE expr DO stmt  
 Rule 59 for\_dir -> TO  
 Rule 60 for\_dir -> DOWNT0  
 Rule 61 for\_stmt -> FOR ID ASSIGN expr for\_dir expr DO for\_enter stmt for\_exit  
 Rule 62 for\_enter -> <empty>  
 Rule 63 for\_exit -> <empty>  
 Rule 64 repeat\_stmt -> REPEAT stmt\_list\_opt UNTIL expr  
 Rule 65 proc\_call -> ID  
 Rule 66 proc\_call -> ID LPAREN arg\_list\_opt RPAREN  
 Rule 67 proc\_call -> WRITELN args\_opt  
 Rule 68 proc\_call -> READLN read\_args\_opt  
 Rule 69 read\_args\_opt -> LPAREN read\_var\_list RPAREN  
 Rule 70 read\_args\_opt -> <empty>  
 Rule 71 read\_var\_list -> lvalue  
 Rule 72 read\_var\_list -> lvalue COMMA read\_var\_list  
 Rule 73 args\_opt -> LPAREN arg\_list\_opt RPAREN  
 Rule 74 args\_opt -> <empty>  
 Rule 75 arg\_list\_opt -> arg\_list  
 Rule 76 arg\_list\_opt -> <empty>  
 Rule 77 arg\_list -> expr arg\_list\_tail  
 Rule 78 arg\_list\_tail -> COMMA expr arg\_list\_tail  
 Rule 79 arg\_list\_tail -> <empty>  
 Rule 80 expr -> or\_expr  
 Rule 81 or\_expr -> and\_expr  
 Rule 82 or\_expr -> or\_expr OR and\_expr  
 Rule 83 and\_expr -> rel\_expr  
 Rule 84 and\_expr -> and\_expr AND rel\_expr  
 Rule 85 rel\_expr -> add\_expr rel\_opt  
 Rule 86 rel\_opt -> relop add\_expr  
 Rule 87 rel\_opt -> <empty>  
 Rule 88 relop -> EQUAL  
 Rule 89 relop -> NOTEQUAL  
 Rule 90 relop -> LESS  
 Rule 91 relop -> LESSEQUAL  
 Rule 92 relop -> GREATER  
 Rule 93 relop -> GREATEREQUAL

```

Rule 94    add_expr -> mul_expr
Rule 95    add_expr -> add_expr PLUS mul_expr
Rule 96    add_expr -> add_expr MINUS mul_expr
Rule 97    mul_expr -> unary_expr
Rule 98    mul_expr -> mul_expr TIMES unary_expr
Rule 99    mul_expr -> mul_expr DIVIDE unary_expr
Rule 100   mul_expr -> mul_expr DIV unary_expr
Rule 101   mul_expr -> mul_expr MOD unary_expr
Rule 102   unary_expr -> MINUS unary_expr
Rule 103   unary_expr -> NOT unary_expr
Rule 104   unary_expr -> primary
Rule 105   primary -> NUMBER_REAL
Rule 106   primary -> NUMBER_INT
Rule 107   primary -> STRING_LITERAL
Rule 108   primary -> TRUE
Rule 109   primary -> FALSE
Rule 110   primary -> var_ref
Rule 111   primary -> ID LPAREN arg_list_opt RPAREN
Rule 112   primary -> LPAREN expr RPAREN

```

## 4 Análise Semântica e Gestão de Contexto (src/sem.py e src/context.py)

A análise semântica é responsável por verificar se o programa, embora sintaticamente correto, faz sentido do ponto de vista da lógica da linguagem Pascal.

### 4.1 Tabela de Símbolos e Gestão de Escopos

Utilizámos uma estrutura de Tabela de Símbolos implementada em sem.py, que funciona através de uma pilha de dicionários (scopes). - **Escopo Global vs. Local:** Ao entrar numa função ou procedimento, o compilador faz um **push** de um novo dicionário para a pilha, criando um novo escopo. Ao sair, faz um **pop**. Isto permite que variáveis locais existam apenas durante a execução do subprograma, evitando conflitos com variáveis globais. - **Deteção de Identificadores:** O método **lookup** percorre a pilha de escopos do topo (mais local) para a base (global), garantindo que o compilador encontra sempre a instância mais próxima de uma variável.

### 4.2 Verificação de Tipos (Type Checking)

O compilador realiza verificações estritas para garantir a integridade dos dados: - **Compatibilidade Aritmética:** Implementámos a lógica que permite operações entre **integer** e **real** (resultando em **real**), mas impede operações inválidas entre tipos incompatíveis. - **Arrays:** Validamos se o índice utilizado para aceder a um array é do tipo **integer**. Além disso, o sistema verifica a consistência entre o tipo base do array e o valor que está a ser atribuído. - **Chamadas de Funções:** O módulo **sem.py** verifica se o número e os tipos de argumentos passados numa chamada correspondem exatamente à assinatura definida na declaração da função.

### 4.3 Gestão de Memória e Contexto (context.py)

O ficheiro **context.py** atua como um repositório central de estado durante a compilação: - **Alocação de Endereços:** Gere contadores automáticos para endereços globais (**next\_global\_addr**) e locais (**next\_local\_addr\_stack**), garantindo que cada variável ocupa um slot único na stack da VM. - **Tratamento de Built-ins:** O sistema pré-declara funções essenciais (como **writeln**, **readln**, **length**) na tabela de símbolos global, impedindo que o utilizador as redeclare acidentalmente.

### 4.4 Tratamento de Erros Semânticos

Sempre que uma regra é violada (ex: usar uma variável **x** que não foi declarada em **VAR**), o compilador levanta uma **SemanticError**. Esta exceção interrompe o pipeline e fornece ao utilizador uma mensagem clara com a descrição do problema e a linha onde ocorreu.

## 5 Geração de Código (src/codegen.py)

A geração de código é a fase final (back-end) onde as estruturas validadas são convertidas em instruções de baixo nível para a Máquina Virtual (VM) de stack.

### 5.1 Modelo de Execução baseada em Stack

O código gerado assume uma arquitetura de stack, onde as operações retiram os operandos do topo da pilha e colocam o resultado no mesmo local. - **Instruções Aritméticas:** Operações como **a + b** são traduzidas para uma sequência de carregamento (**PUSH**) seguida da instrução de operação (**ADD**, **SUB**, **MUL**, **DIV**). - **Conversão de Tipos:** Quando o analisador semântico deteta uma operação entre um inteiro e um real, o gerador emite a instrução **ITOF** para garantir a coerência dos dados na stack.

### 5.2 Mapeamento de Memória e Endereçamento

A gestão de variáveis é feita distinguindo o seu escopo: - **Variáveis Globais:** Utilizam as instruções **PUSHG** e **STOREG**, apontando para endereços absolutos definidos no início da compilação através do **next\_global\_addr**. - **Variáveis Locais:** Utilizam **PUSHL** e **STOREL**, com endereços relativos à base da frame atual da stack (FP), geridos pelo **next\_local\_addr\_stack** no **context.py**.

### 5.3 Implementação de Estruturas de Controlo

Para implementar saltos e ciclos, o módulo **codegen.py** fornece um gerador de etiquetas únicas (**new\_label**). - **Condicionais (IF):** O gerador emite um **JZ** (jump if zero) para saltar o bloco **THEN** caso a condição seja falsa. Se existir um **ELSE**, é gerado um **JUMP** no final do bloco **THEN** para evitar a execução do código alternativo. - **Ciclos (WHILE/FOR):** São criadas etiquetas no início (para repetição) e no fim (para saída). No caso do ciclo **FOR**, o compilador gera automaticamente o código de incremento/decremento da variável de controlo e a verificação do limite.

### 5.4 Chamadas de Subprogramas (Funções e Procedimentos)

A geração de código para subprogramas segue um protocolo rigoroso: 1) **Ativação:** É gerada a etiqueta com o nome da função (ou **label** único). 2) **Passagem de Parâmetros:** O código coloca

os argumentos na stack antes da instrução **CALL**. 3) **Retorno de Valores:** Para funções, o compilador reserva um slot na stack (**push\_default\_for\_type**) para o valor de retorno antes de empilhar os argumentos, garantindo que o resultado fica disponível para o chamador após o **POP** dos argumentos.

## 5.5 Strings e I/O

- **Strings:** O compilador gera instruções **PUSHS** para literais de texto.
- **Input/Output:** As chamadas a **readln** e **writeln** são traduzidas para instruções nativas da VM (**READ**, **WRITE**, **WRITELN**), permitindo a interação com o utilizador.

## 6 Testes e Resultados

A validação do compilador foi realizada através de uma infraestrutura de testes automatizada, desenhada para garantir a fiabilidade da geração de código e a precisão dos diagnósticos de erro.

### 6.1 Infraestrutura de Testes Automática

Afastando-nos de uma abordagem de testes manuais, implementámos o script **run\_tests.py**. Este motor de testes percorre sistematicamente a diretoria **tests/cases/**, executando o pipeline para cada ficheiro **.pas** e comparando o resultado com as expetativas definidas no sistema.

### 6.2 Testes de Regressão (Casos de Sucesso)

Na pasta **tests/cases/ok/**, mantemos uma suite de testes que cobre todas as funcionalidades exigidas no enunciado: - **Expressões e Atribuições:** Testes de precedência aritmética e lógica. - **Estruturas de Controlo:** Validação de ciclos **FOR** (tanto **to** como **downto**), **WHILE** e condicionais **IF-THEN-ELSE**. - **Subprogramas:** Chamadas de funções e procedimentos com passagem de parâmetros e gestão de valor de retorno. - **Estruturas de Dados:** Manipulação de arrays (incluindo acesso indexado) e strings. - **Exemplos do Enunciado:** Os cinco exemplos propostos (Olá Mundo, Fatorial, Números Primos, Soma de Array e Binário para Inteiro) são compilados e os resultados em assembly são validados na VM.

### 6.3 Testes de Stress e Diagnóstico (Casos de Erro)

Um dos pontos fortes deste compilador é a sua capacidade de lidar com código inválido. Através do ficheiro **tests/manifests/error\_cases.json**, definimos um manifesto que mapeia ficheiros de erro (na pasta **cases/error/**) às respetivas mensagens que o compilador deve emitir.

O sistema valida se o compilador deteta erros como: - **Violações de Escopo:** Uso de variáveis não declaradas (ex: **T1.pas**). - **Incompatibilidade de Tipos:** Operações inválidas, como o uso do operador **MOD** com tipos reais ou condições de **IF** que não resultam em boolean (ex: **T11.pas**, **T19.pas**). - **Erros de Assinatura:** Chamadas de funções com número ou tipos de argumentos incorretos (ex: **T28.pas**, **T29.pas**).

### 6.4 Análise de Resultados

O sucesso na execução da suite completa de testes demonstra que o compilador possui: 1) **Estabilidade:** Alterações em módulos (como o **codegen.py**) não quebram funcionalidades anteriores. 2)



**Precisão:** As mensagens de erro são específicas, indicando exatamente a falha lógica (ex: “Comparação inválida” ou “Range inválido”). 3) **Conformidade:** O código da VM gerado respeita a stack discipline e os limites de memória definidos pela arquitetura de destino.

## 7 Conclusão e Dificuldades Superadas

O desenvolvimento deste compilador para Pascal Standard representou um desafio técnico significativo, exigindo a integração harmoniosa de conceitos teóricos de linguagens formais com soluções práticas de engenharia de software.

### 7.1 Síntese do Trabalho Realizado

Conseguimos implementar um sistema funcional que cobre desde a análise de texto bruto até à geração de código executável. A modularização em Python permitiu que cada fase (Léxica, Sintática, Semântica e Geração) fosse testada de forma independente, resultando num produto final robusto que cumpre todos os requisitos do enunciado, incluindo o suporte a subprogramas e tipos estruturados como `array` (Ver: `src/compiler.py`).

### 7.2 Dificuldades e Soluções Técnicas

Durante o desenvolvimento, enfrentámos desafios complexos que exigiram soluções criativas: - **Gestão da Stack Discipline:** Um dos maiores desafios foi garantir que a stack da VM permanecesse equilibrada após chamadas de funções. A solução passou por reservar antecipadamente um slot para o valor de retorno (`push_default_for_type`) e limpar os argumentos (`POP k`) logo após o retorno do subprograma (Ver: `src/codegen.py`). - **Compatibilidade de Tipos (Coerção):** A mistura de `integer` e `real` em expressões aritméticas exigiu a inserção automática da instrução `ITOF`. A implementação desta lógica no Middle-end permitiu que o Back-end gerasse código correto sem intervenção do utilizador (Ver: `src/parser.py`). - **Gestão de Identificadores Reservados:** Para impedir que utilizadores redeclarassem funções internas (como `writeln` ou `length`), implementámos um mecanismo de proteção na Tabela de Símbolos que bloqueia qualquer tentativa de shadowing de nomes built-in (Ver: `src/sem.py`).

### 7.3 Melhorias

Embora o compilador esteja totalmente funcional, a arquitetura modular adotada permite futuras expansões, tais como: - **Otimização de Código:** Implementação de Constant Folding (cálculo de expressões constantes em tempo de compilação). - **Novos Tipos:** Expansão para suportar `records` ou `pointers`. - **Geração de Código Nativo:** Adaptar o Back-end para gerar instruções x86 ou ARM.

[ ]: