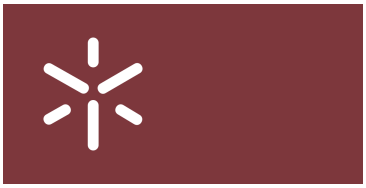

Universidade do Minho
Escola de Engenharia
Licenciatura em Engenharia Informática
Mestrado Integrado em Engenharia Informática

Unidade Curricular de Processamento de Linguagens

Ano Letivo de 2024/2025

Construção de um Compilador para Pascal Standard

André Miranda a104088
Diogo Outeiro a104092
José Soares a103995

Junho, 2025

PL

Data da Receção	
Responsável	
Avaliação	
Observações	

Construção de um Compilador para Pascal Standard

André Miranda a104088
Diogo Outeiro a104092
José Soares a103995

Junho, 2025

Índice

1. Introdução

O relatório descreve o desenvolvimento de um compilador para a linguagem Pascal Standard, como parte do projeto da UC de Processamento de Linguagens. O objetivo principal é construir um compilador capaz de Pascal para EWVM, uma linguagem de Máquina Virtual que interpreta comandos estilo *Assembly*.

Neste relatório iremos abordar todos os passos desde a leitura e análise léxica e sintática, até à análise semântica e geração de código para uma máquina virtual disponibilizada.

2. Arquitetura do Compilador

2.1. Introdução

A arquitetura do nosso compilador segue uma estrutura modular e escolheu-se esta abordagem porque permite uma separação clara de responsabilidades, o que facilitou o desenvolvimento, manutenção e extensão do compilador. O nosso sistema está organizado em quatro componentes principais que operam sequencialmente para transformar o código-fonte na linguagem Pascal em código executável para a máquina virtual EWVM.

2.2. Analisador Léxico (Lexer)

O analisador léxico, implementado no módulo analex.py, constitui a primeira fase do processo de compilação. Esta componente é responsável por decompor o texto do programa fonte numa sequência de tokens, que são as unidades léxicas básicas da linguagem.

2.2.1. Princípios de Funcionamento

O nosso lexer utiliza a biblioteca PLY (Python Lex-Yacc) para definir e reconhecer os tokens da linguagem Pascal. O processo consiste em primeiro definir e especificar os *tokens* com expressões regulares, e depois utilizar o lexer para fazer o reconhecimento do texto de entrada.

Definição de tokens: São definidos todos os tokens possíveis da linguagem, incluindo palavras-chave (como PROGRAM, BEGIN, END), operadores (como +, -, :=), delimitadores (como ;, (,)), e identificadores.

Regras de reconhecimento: Para cada token, é definida uma expressão regular que descreve o padrão léxico associado.

Processamento de entrada: O lexer percorre o texto de entrada e vai reconhecendo e classificando cada sequência de caracteres de acordo com as regras definidas.

2.2.2. Características Implementadas

O nosso lexer apresenta as seguintes características:

- Reconhecimento de palavras-chave: Identificação de todas as palavras reservadas da linguagem Pascal, como program, begin, end, if, then, etc.
- Insensibilidade a maiúsculas/minúsculas: As palavras-chave são reconhecidas independentemente da capitalização, seguindo o padrão da linguagem Pascal.
- Tratamento de diferentes tipos de dados: Suporte para identificadores, números inteiros, valores booleanos, strings e caracteres.
- Deteção de erros léxicos: Identifica e reporta caracteres ilegais no código fonte.

2.3. Analisador Sintático (Parser)

O analisador sintático, implementado no módulo anasin.py, constitui a segunda fase do processo de compilação, sendo esta de grande importância pois é aqui que se define a gramática. Esta componente recebe a sequência de tokens produzida pelo lexer e verifica se estes formam estruturas gramaticalmente válidas de acordo com as regras da linguagem Pascal.

2.3.1. Gramática

É nesta etapa que se define a gramática da linguagem Pascal, o elemento central e determinante para o funcionamento correto do analisador sintático. A gramática especifica, de forma formal, todas as construções válidas da linguagem, desde declarações simples até estruturas mais complexas e definições de funções. A sua definição deve ser feita de forma cautelosa, não só para garantir uma análisecorreta dos programas fonte, mas também para assegurar a coerência e a consistência de todo o processo de compilação daqui para a frente. Uma gramática bem estruturada facilita a manutenção e a extensão da linguagem.

2.3.2. Princípios de Funcionamento

O parser recorre à biblioteca *PLY* (Python Lex-Yacc) para realizar uma análise ascendente (bottom-up), baseada em LR (Left-to-Right). A gramática da linguagem é definida através de um conjunto de regras que descrevem as construções válidas. Com base nessas regras, o PLY gera automaticamente uma tabela de análise, que serve de base ao processo de análise sintática. Durante essa análise, o parser utiliza a tabela para processar a sequência de tokens e verifica se esta corresponde a uma derivação válida segundo a gramática especificada.

2.4. Árvore de Sintaxe Abstrata (AST)

A AST é construída durante a análise sintática e serve como intermediário entre o parser e o gerador de código. A utilização desta estrutura abstrai detalhes sintáticos irrelevantes e destaca a estrutura lógica do programa que vai ser a parte principal utilizada para traduzir a linguagem.

Representação hierárquica: O programa é representado como uma árvore, onde cada nó corresponde a uma construção da linguagem.

Abstração de detalhes sintáticos: A AST elimina tokens puramente sintáticos (como ;, begin, end) que não afetam a semântica do programa.

Suporte à análise semântica: A estrutura da AST facilita a verificação de tipos, range de variáveis e outras verificações semânticas.

2.4.1. Implementação

A nossa implementação da AST baseia-se na classe ASTNode, que representa um nó genérico da árvore:

```
class ASTNode:
    def __init__(self, type, children=None, value=None, extra=None, lineno=None):
        self.type = type
        self.children = children if children else []
        self.extra = extra or {}
        self.value = value
        self.lineno = lineno
```

Cada nó da AST tem um tipo que indica a construção a ser feita, uma lista de filhos que são o subnodos da árvore, um valor associado ao nó assim como outro valor *extra* (se necessário) e também o número da linha no código fonte.

2.4.2. Padrão Visitor

Depois de termos a árvore construida era necessário definir como é que seria atravessada e processada, inicialmente tínhamos pensado em executar a tradução de forma recursiva, no entanto, decidimos procurar melhores alternativas e decidiu-se utilizar o *design pattern Visitor*, que permite percorrer a árvore de forma flexível e realizar diferentes operações em cada nó. Este desenho incluí um método chamado *accept()* que é o encarregue de selecionar a função de visita correspondente para o valor do nodo que a árvore toma. A função de visita por si só define o caso genérico de tradução, sempre tendo em conta o nodo que lhe é passado.

Posteriormente neste relatório, iremos voltar a focar-nos nestra travessia uma vez que é aqui que se constrói o código executável na VM.

2.5. Analisador Semântico

Antes de proceder à tradução do código, o nosso compilador realiza uma análise semântica, pensada por nós módulo semantics.py, porque sentimos que se o programa não está semanticamente correto, não faria sentido deixá lo passar à fase de tradução. Assim como num compilador, caso exista alguma anomalia com o código, normalmente, é nos dito a razão pelo sucedido.

2.5.1. Funcionalidades da Análise Semântica

As funcionalidades que pensamos em implementar para as verificações são as seguintes:

- Verificação de tipos: Assegura que as operações são realizadas entre tipos compatíveis e que as atribuições respeitam os tipos das variáveis.
- Verificação de declarações: Verifica se todas as variáveis foram declaradas antes de serem utilizadas.
- Deteção de variáveis não utilizadas: Identifica variáveis que foram declaradas mas nunca utilizadas.
- Validação de chamadas de funções: Verifica se as funções chamadas foram devidamente declaradas.

2.6. Tradutor para EWVM

O tradutor, implementado no módulo translator.py, constitui a fase final do processo de compilação. Esta componente recebe a AST validada e gera código para a máquina virtual EWVM.

2.6.1. Princípios de Funcionamento

O tradutor baseia-se no padrão Visitor, já enunciado, para percorrer a AST e gerar código correspondente para cada construção da linguagem.

Aqui assumimos alguns parâmetros como:

- Gestão do *Scope*: Manutenção de informações sobre variáveis do programa principal e das suas funções.
- Alocação de memória: Atribuição de endereços de memória para variáveis e estruturas de dados. Dicionário no formato identificador-endereço de memória.

2.7. Integração dos Componentes

A integração dos vários componentes do compilador é feita de forma sequencial:

1. Lexer: Converte o código fonte em tokens.
2. Parser: Analisa os tokens e constrói a AST.
3. Analisador Semântico: Verifica a correção semântica da AST.
4. Tradutor: Gera código EWVM a partir da AST.

3. Implementação do EWVMTranslator

A classe é composta pela própria árvore sintaxe abstrata, por listas onde são acumuladas as instruções EMVM e em alguns casos código de funções separadamente do código principal, uma scope_stack onde cada scope é um dicionário de variáveis locais visíveis e um var_offset que controla endereços de variáveis globais na memória da máquina virtual.

Todas as funções inicializam variáveis e o seu tratamento é diferente caso sejam normais(integer,real e string) e arrays. No primeiro caso a variável é criada no scope_stack, é emitido o comando *PUSHI 0* e o var_offset é incrementado para a próxima variável. No segundo caso, calcula-se o tamanho do array e faz-se *PUSHI* desse valor, seguido de um *ALLOCN* para alocar espaço e depois armazena a variável com o seu tamanho e tipo.

Com as variáveis declaradas, segue-se a geração do corpo do programa ou função, cujas instruções são envolvidas por START no início e STOP no fim. O código é gerado percorrendo a AST e emitindo instruções EWVM de acordo com os nós encontrados. Sempre que uma variável previamente declarada é utilizada, o tradutor emite um PUSHG *offset* para empilhar o seu valor. No caso de uma atribuição, o valor da expressão do lado direito é calculado e empilhado, e depois armazenado com um STOREG *offset*. Para arrays, o processo é ligeiramente mais complexo: o endereço base é empilhado com PUSHG, o índice é calculado e, conforme o caso, usa-se LOADN ou STOREN para ler ou escrever o valor no índice adequado.

Os literais também são tratados de forma diferenciada. Para valores booleanos, o tradutor emite *PUSHI 1* para true e *PUSHI 0* para false. Os literais inteiros e reais são tratados com PUSHI *valor* ou PUSHF *valor*, e literais string são manipulados com instruções específicas, podendo envolver referências a uma tabela de constantes ou alocação dinâmica.

As expressões aritméticas e lógicas são geradas de forma pós-fixada: os operandos são empilhados primeiro, e depois a operação é aplicada. Por exemplo, uma adição entre duas expressões gera as instruções das duas subexpressões, seguidas de um ADD. O mesmo se aplica a operações como SUB, MUL, DIV, assim como operadores de comparação (EQ, NEQ, LT, GT, etc.) e operadores booleanos (AND, OR, NOT).

As estruturas de controlo, como if, if-else, while e for, são geradas com base em rótulos e saltos condicionais. Em um if, por exemplo, o tradutor gera o código da condição, seguido de um salto JZ para o rótulo do bloco alternativo (ou fim, se não houver else). Em while, é gerado um rótulo para o início do loop, a condição é verificada, e se for falsa, salta-se para o fim com JZ. Dentro do corpo do loop, no final, um salto incondicional JUMP é responsável pela verificação da condição.

Funções e procedimentos são gerados separadamente do corpo principal. Ao encontrar uma definição, o tradutor gera um LABEL com o nome da função, seguido de START, declarações locais, o corpo da função, e termina com RETURN. Os argumentos são tratados como variáveis locais e são empilhados antes da chamada com PUSH das expressões correspondentes, sendo depois a função invocada com CALL *label*. Em funções, o valor de retorno é deixado no topo da pilha para ser usado no ponto de chamada.

4. Testes

A verificação da corretude do compilador foi realizada com base em um conjunto de testes funcionais organizados na pasta `tests/`, utilizando scripts automatizados presentes na pasta `scripts/`.

4.1. Estrutura dos testes

Cada ficheiro de teste, com extensão `.txt`, representa um pequeno programa Pascal válido ou inválido, projetado para cobrir construções diferentes da linguagem, tais como:

- Declarações de variáveis e tipos básicos;
- Estruturas de controlo (if, while, for, repeat);
- Escrita e leitura de dados com write, writeln, read, readln;
- Chamadas a funções e procedimentos.

Os ficheiros `test1.txt` a `test13.txt` foram usados para validar o comportamento de cada fase do compilador, desde a análise léxica até à geração de código para a EWVM.

O script `test_runner.sh` percorre todos os ficheiros de teste, invoca o compilador (`main.py`) e redireciona os resultados de execução e/ou erros para o ficheiro `test_results.log`.

4.2. Execução Automática

A execução dos testes foi automatizada através do script `scripts/test_runner.sh`, que processa os ficheiros de teste, executa o compilador e regista os resultados em `scripts/test_results.log`. Este processo permite processar os inputs, verificar se os programas são corretamente analisados e traduzidos e geram os resultados esperados.

4.3. Resultados Obtidos

A execução dos testes demonstrou que o compilador lida corretamente os casos apresentados. Todos os ficheiros `.txt` que representam programas válidos foram processados com sucesso e o seu código para a EWVM aparece sem erros. Programas com erros léxicos, sintáticos ou semânticos resultaram em mensagens apropriadas, geradas na fase correspondente.

Em particular:

- A análise léxica foi eficaz a reconhecer tokens válidos e a rejeitar caracteres ilegais.
- A análise sintática detetou com precisão erros de estrutura como `begin`/`end` mal balanceados ou omissão de `;`.
- A análise semântica conseguiu identificar atribuições inválidas entre tipos, variáveis não declaradas e chamadas a funções inexistentes.
- A geração de código produziu instruções EWVM corretas, testadas com a ferramenta da máquina virtual fornecida.

Teste 1:

```
Árvore Sintática Abstrata (AST):
Program [HelloWorld]
  Block
    Declarations
    Compound
      StatementList
        Write [writeln]
          Literal [Ola, Mundo!]


Código Gerado:
START
PUSHS "Ola, Mundo!"
WRITES
STOP
```

Teste 4:

```
Árvore Sintática Abstrata (AST):
Program [NumeroPrimo]
  Block
    Declarations
      DeclList
        Declaration
          IdList
            Id [num]
            Id [i]
          Type [integer]
        Declaration
          IdList
            Id [primo]
          Type [boolean]
    Compound
      StatementList
        Write [writeln]
          Literal [Introduza um número inteiro positivo:]
        Read [readln]
          Variable [num]
        Assign
          Variable [primo]
          Literal [true]
        Assign
          Variable [i]
          Literal [2]
        While
          LogicalOp [and]
            RelOp [<=]
              Variable [i]
              MulOp [div]
                Variable [num]
                Literal [2]
            Variable [primo]
          CompoundBlock
            StatementList
              IfElse
                RelOp [=]
                  MulOp [mod]
                    Variable [num]
                    Variable [i]
                  Literal [0]
                Assign
                  Variable [primo]
                  Literal [false]
              Assign
                Variable [i]
                AddOp [+]
                  Variable [i]
                  Literal [1]
        IfElse
          Variable [primo]
          Write [writeln]
            Variable [num]
            Literal [ é um número primo]
          Write [writeln]
            Variable [num]
            Literal [ não é um número primo]
```

```
Código Gerado:
PUSHI 0
PUSHI 0
PUSHI 0
START
PUSHS "Introduza um número inteiro positivo:"
WRITES
READ
ATOI
STOREG 0
PUSHI 1
STOREG 2
PUSHI 2
STOREG 1
WHILE0:
PUSHG 1
PUSHG 0
PUSHI 2
DIV
INFEQ
PUSHG 2
AND
JZ ENDWHILE0
PUSHG 0
PUSHG 1
MOD
PUSHI 0
EQUAL
JZ ELSE1
PUSHI 0
STOREG 2
JUMP ENDIF1
ELSE1:
ENDIF1:
PUSHG 1
PUSHI 1
ADD
STOREG 1
JUMP WHILE0
ENDWHILE0:
PUSHG 2
JZ ELSE2
PUSHG 0
WRITEI
PUSHS " é um número primo"
WRITES
JUMP ENDIF2
ELSE2:
PUSHG 0
WRITEI
PUSHS " não é um número primo"
WRITES
ENDIF2:
STOP
```

O ficheiro `test_results.log` mostrou que os testes foram concluídos com sucesso, sem falhas inesperadas, evidenciando a robustez do sistema nas situações previstas. O comportamento consistente entre múltiplas execuções também atestou a estabilidade da implementação.

5. Conclusão sobre o trabalho

O desenvolvimento deste compilador para a linguagem Pascal Standard constituiu uma oportunidade prática para aplicar os conceitos aprendidos na unidade curricular de PL. Ao longo do projeto, nós passamos por todas as fases fundamentais de um compilador: análise léxica, sintática, semântica e geração de código.

A utilização da biblioteca PLY facilitou a implementação dos analisadores, enquanto a criação da AST e o uso do padrão Visitor permitiram uma travessia mais extensível e modular da estrutura do programa. A análise semântica complementou com a integridade lógica dos programas, e a tradução para a EWVM confirmou que o sistema é funcional e produz código executável.

Este trabalho reforçou a nossa compreensão sobre a estrutura interna de compiladores e destacou a importância de uma arquitetura modular, testes sistemáticos e validação rigorosa. Apesar dos desafios, consideramos que os objetivos foram plenamente atingidos e que o projeto contribuiu significativamente para o nosso desenvolvimento técnico.

Bibliografia

[1] T. Mitchell, *Machine Learning*. McGraw Hill, 2017.

[2] E. Alpaydin, *Introduction to Machine Learning*. The MIT Press, 2014.

[3] A. P. Engelbrecht, *Computational Intelligence: An Introduction*, 2nd ed. Wiley & Sons, 2007.

[4] T. Hastie, R. Tibshirani, e J. Friedman, *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*, 12th ed. Springer, 2016.

[5] K. P. Murphy, *Machine Learning: A Probabilistic Perspective*, 4th ed. The MIT Press, 2012.

Lista de Siglas e Acrónimos

PL Processamento de Linguagens

Anexos

Gramática

```
program
    : PROGRAM ID SEMI block DOT
    ;

block
    : declaration_section func_section compound_section
    | declaration_section compound_section
    ;

declaration_section
    : ε
    | VAR declaration_list
    ;

declaration_list
    : declaration
    | declaration_list declaration
    ;
declaration
    : id_list COLON type SEMI
    ;

id_list
    : ID
    | id_list COMMA ID
    ;

type
    : INTEGER
    | BOOLEAN
    | STRING
    | ARRAY LBRACK NUMBER DOTDOT NUMBER RBRACK OF type
    ;
func_section
    : ε
    | func_section func_declaration
    ;

func_declaration
    : FUNCTION ID LPAREN parameters RPAREN COLON type SEMI block SEMI
    | PROCEDURE ID LPAREN parameters RPAREN SEMI block SEMI
    ;
```

```
parameters
    : ε
    | parameter_list
    ;

parameter_list
    : parameter
    | parameter_list SEMI parameter
    ;

parameter
    : id_list COLON type
    ;

compound_section
    : BEGIN statement_list END
    ;

statement_list
    : statement SEMI
    | statement_list statement SEMI
    ;

statement
    : assign_statement
    | if_statement
    | while_statement
    | for_statement
    | repeat_statement
    | compound_section
    | function_call
    | write_statement
    | read_statement
    | ε
    ;

assign_statement
    : variable ASSIGN expression
    ;

variable
    : ID
    | ID LBRACK expression RBRACK
    ;

if_statement
    : IF expression THEN statement
    | IF expression THEN statement ELSE statement
    ;

while_statement
    : WHILE expression DO statement
    ;
```

```
for_statement
    : FOR ID ASSIGN expression TO expression DO statement
    | FOR ID ASSIGN expression DOWNTO expression DO statement
    ;

repeat_statement
    : REPEAT statement_list UNTIL expression
    ;

function_call
    : ID LPAREN argument_list RPAREN
    ;

argument_list
    : ε
    | expression
    | argument_list COMMA expression
    ;

write_statement
    : write_func LPAREN output_list RPAREN
    ;

write_func
    : WRITE
    | WRITELN
    ;

read_statement
    : read_func LPAREN variable_list RPAREN
    ;

read_func
    : READ
    | READLN
    ;

output_list
    : expression
    | output_list COMMA expression
    ;

variable_list
    : variable
    | variable_list COMMA variable
    ;

expression
    : logical_expression
    ;
```

```
logical_expression
    : relational_expression
    | logical_expression AND relational_expression
    | logical_expression OR relational_expression
    | NOT relational_expression
    ;

relational_expression
    : addition_expression
    | addition_expression relop addition_expression
    ;

addition_expression
    : mult_expression
    | addition_expression addop mult_expression
    ;
mult_expression
    : unary_expression
    | mult_expression mulop unary_expression
    ;

unary_expression
    : primary_expression
    | sign unary_expression
    ;

primary_expression
    : variable
    | NUMBER
    | STRING_LITERAL
    | TRUE
    | FALSE
    | LPAREN expression RPAREN
    | function_call
    ;

relop
    : EQ
    | NE
    | LT
    | LE
    | GT
    | GE
    ;

addop
    : PLUS
    | MINUS
    ;

mulop
    : TIMES
    | DIVIDE
    | DIV
    | MOD
    ;

sign
    : PLUS
    | MINUS
    ;
```