Universidade do Minho

Escola de Engenharia

Licenciatura em Engenharia Informática

Processamento de Linguagens

Compilador para uma Sublinguagem de Pascal

Grupo 58:

Paulo Moreira (A64459)
Ana Pires (A96060)
Pedro Guerra (A105133)

1 de junho de 2025

Resumo

Este relatório descreve o desenvolvimento de um compilador para uma sublinguagem do Pascal, conforme os requisitos do projeto de Processamento de Linguagens. O compilador foi construído em Python e compreende as fases principais de compilação: análise léxica, análise sintática, análise semântica e geração de código para uma máquina virtual (VM) de pilha. Detalhamos a implementação de cada fase, as ferramentas utilizadas (PLY), a gestão de erros, os testes realizados e os desafios superados no processo, com o objetivo de traduzir código Pascal para um formato executável pela VM disponibilizada.

Conteúdo

1 Introdução

O presente projeto, desenvolvido no âmbito da unidade curricular de Processamento de Linguagens, consiste na construção de um compilador para uma porção simplificada da linguagem de programação Pascal Standard. O objetivo principal foi aplicar os conhecimentos teóricos sobre as fases clássicas de um compilador, desde a leitura do código-fonte até à produção de um código executável por uma máquina virtual de pilha (stack-based VM). O compilador foi desenvolvido integralmente em Python, utilizando a biblioteca PLY (Python Lex-Yacc) para as fases de análise léxica e sintática. O sistema implementado é capaz de analisar, interpretar e traduzir código Pascal, gerando código para a VM fornecida.

2 Arquitetura do Compilador

A arquitetura do compilador segue um design modular, dividindo o processo de compilação nas seguintes fases principais, orquestradas pelo script `main.py`:

- **Análise Léxica (Lexer):** Implementada em `lexer.py`, converte o código-fonte Pascal numa sequência de tokens (unidades léxicas).
- **Análise Sintática (Parser):** Implementada em `pascal_parser.py`, constrói uma Árvore de Sintaxe Abstrata (AST) a partir da sequência de tokens, validando a estrutura gramatical do código. Os nós da AST são definidos em `ast_nodes.py`.
- **Análise Semântica:** Implementada em `semantic_analyzer.py`, percorre a AST para verificar a correção semântica do código, incluindo declaração de variáveis e funções, compatibilidade de tipos e uso correto de identificadores, utilizando uma tabela de símbolos com escopos.
- **Geração de Código:** Implementada em `code_generator.py`, transforma a AST semanticamente verificada em instruções para a máquina virtual de pilha.

O ponto de entrada do compilador é o script `main.py`, que lê um ficheiro Pascal, executa as fases de compilação e, se bem-sucedido, grava o código VM resultante num ficheiro com extensão `.vm`.

Listing 1: main.py - Orquestração das Fases do Compilador

```
import ply.yacc as yacc # Necessário para parser.parse
from lexer import lexer # Importa o objeto lexer
from pascal_parser import parser # Importa o objeto parser
from semantic_analyzer import SemanticAnalyzer, SemanticError # Importa o analisador semntico
from code_generator import CodeGenerator
import sys

def test_and_compile(code, output_file=None):
    """
    Testa as fases de análise sintática e semntica para um dado código Pascal.
    """
    print(f"\n--- Analisando Sintaticamente o Código ---")
    print(code)
    try:
        # 1. Análise Sintática
        lexer.lineno = 1 # Reseta o contador de linhas do lexer

        ast = parser.parse(code, lexer=lexer)
        if ast:
            print("\nAnálise Sintática Concluída. Árvore de Sintaxe Abstrata (AST) construída.")

            # 2. Análise Semntica
            print("\n--- Iniciando Análise Semntica ---")
            semantic_analyzer = SemanticAnalyzer()
            sem = semantic_analyzer.analyze(ast) # Este método já imprime os erros ou sucesso
            print("\nAnálise Semntica Concluída.")
            if sem:
                # 3. Geração de Código (se necessário)
                print("\n--- Iniciando Geração de Código ---")
                gen = CodeGenerator(semantic_analyzer)
                generated_vm_code = gen.generate_code(ast)

```

```
                if hasattr(gen, 'vm_code') and gen.vm_code: # Verifica se vm_code foi populado
                    print("\nGeração de Código Concluída.")
                    print("Código gerado:")
                    # Escreve no ficheiro de saída ou no stdout
                    # A função generate_code já retorna a string do código VM
                    # A escrita para ficheiro éfeita no bloco if __name__ == '__main__'
                    with open(output_file, 'w') if output_file else sys.stdout as out:
                        # O generate_code agora retorna uma string com newlines
                        out.write(generated_vm_code)
                        print(generated_vm_code) # Imprime para consola também
                elif hasattr(gen, 'errors') and gen.errors:
                     print("\n--- Erros na Geração de Código ---")
                     for error_msg in gen.errors: print(error_msg)
                else:
                     print("Nenhum código VM foi gerado ou ocorreu um erro não capturado.")

        return ast
    except Exception as e:
        print(f"Erro durante a compilação: {e}")
        import traceback
        traceback.print_exc()
        return None

if __name__ == '__main__':
    filename = sys.argv[1] if len(sys.argv) > 1 else 'example.pas'
    try:
        with open(filename, 'r') as file:
            code = file.read()
        print(f"Lendo código de {filename}...")
        outname = filename.replace('.pas', '.vm')
        test_and_compile(code, output_file=outname)
    except FileNotFoundError:
        print(f"Arquivo {filename} não encontrado.")
        sys.exit(1)
    except Exception as e:
        print(f"Erro ao ler o arquivo {filename}: {e}")
        sys.exit(1)
```

3 Análise Léxica (Lexer)

A fase de análise léxica é implementada no ficheiro `lexer.py`, utilizando as funcionalidades de `lex` do PLY. O lexer define padrões (expressões regulares) para identificar e classificar os tokens da sublinguagem Pascal.

Principais Características:

- **Tokens Reconhecidos:** Incluem palavras-chave (e.g., `PROGRAM`, `VAR`, `BEGIN`, `END`, `IF`, `THEN`, `ELSE`, `FOR`, `TO`, `DOWNTO`, `DO`, `WHILE`, `FUNCTION`, `ARRAY`, `OF`), operadores (`DIV`, `MOD`, `AND`, `OR`, `NOT`), identificadores, números (inteiros e reais), literais de string e de booleanos (`TRUE`, `FALSE`), e símbolos de pontuação (`+`, `-`, `*`, `/`, `:=`, `=`, `<>`, etc.).
- **Case-Insensitivity para Palavras-Chave:** As palavras-chave são reconhecidas independentemente da capitalização, utilizando expressões regulares explícitas para cada variação (e.g., `r'[Pp][Rr][Oo][Gg][Rr][Aa][Mm]'`). Um lookahead negativo (`(?![a-zA-Z0-9_])`) é usado para garantir que palavras como 'integerVar' não sejam confundidas com a palavra-chave 'integer'.
- **Tratamento de Comentários:**
  - Comentários de uma linha: `{...}` são ignorados.
  - Comentários aninhados: `(*...*)` são tratados utilizando estados do lexer (`pstarcomment`). Um contador (`lexer.comment_nesting_level`) gere o nível de aninhamento, e o lexer só retorna ao estado inicial quando todos os níveis de comentários aninhados são fechados.
- **Literais:** Números são convertidos para `int` ou `float`. Literais de string (e.g., `'olá mundo'`) têm as aspas externas removidas e sequências de duas aspas (e.g., `''`) são convertidas para uma única aspa.

- **Controlo de Linhas e Erros:** O número da linha é rastreado para reportar erros léxicos com a sua localização. Caracteres ilegais geram um erro e são ignorados.

Listing 2: Parte do lexer.py (Comentários Aninhados e Palavra-Chave)

```
# Estados para comentários aninhados (*...*)
states = (
    ('pstarcomment', 'exclusive'),
)

# Exemplo de Palavra-Chave (case-insensitive)
def t_PROGRAM(t):
    r'[Pp][Rr][Oo][Gg][Rr][Aa][Mm](?![a-zA-Z0-9_])'
    return t

# Abertura de comentário (* *)
def t_PSTAR_OPEN_INITIAL(t):
    r'\(\*'
    if not hasattr(t.lexer, 'comment_nesting_level'):
        t.lexer.comment_nesting_level = 0
    t.lexer.comment_nesting_level += 1
    t.lexer.push_state('pstarcomment')

# Dentro de um comentário (* *)
def t_pstarcomment_OPEN(t):
    r'\(\*'
    t.lexer.comment_nesting_level += 1

def t_pstarcomment_CLOSE(t):
    r'\*\)'
    t.lexer.comment_nesting_level -= 1
    if t.lexer.comment_nesting_level == 0:
        t.lexer.pop_state()

# Consome conteúdo do comentário, incluindo novas linhas
def t_pstarcomment_CONTENT(t):
    r'([^*()\n]+|\((?!\*)|\*(?!\))|\n)+'
    t.lexer.lineno += t.value.count('\n')
    pass

def t_pstarcomment_error(t):
    t.lexer.skip(1) # Ignora o caractere
```

4 Análise Sintática (Parser)

A análise sintática é implementada em `pascal_parser.py` usando `yacc` do PLY. O parser valida a estrutura gramatical do código Pascal com base numa gramática definida por regras de produção e constrói uma Árvore de Sintaxe Abstrata (AST).

Principais Características:

- **Gramática LL(1) (Adaptada):** As regras de produção definem a sintaxe da sublinguagem Pascal, cobrindo declarações de programas, variáveis e funções, tipos de dados (INTEGER, REAL, BOOLEAN, STRING, ARRAY), comandos (atribuição, IF-THEN-ELSE, WHILE-DO, FOR-TO/DOWNTO-DO, READLN, WRITELN, chamadas de procedimento/função) e expressões.
- **Construção da AST:** Cada regra de produção é associada a uma ação Python que instancia nós da AST (definidos em `ast_nodes.py`). Por exemplo, a regra para um programa cria um `ProgramNode`. O número da linha do token relevante é armazenado em cada nó da AST para posterior reporte de erros.
- **Precedência e Associatividade de Operadores:** A tabela `precedence` define a ordem de avaliação dos operadores (e.g., '*', '/' antes de '+', '-'; operadores relacionais por último) e a sua associatividade (maioritariamente à esquerda). O token especial `%prec UMINUS` é usado para dar ao menos unário a precedência correta.

- **Tratamento de Erros Sintáticos:** A função `p_error(p)` é chamada quando o parser encontra um token inesperado. Ela reporta o erro, incluindo o token problemático e a linha.
- **Produções Vazias (`empty`):** Usadas para tratar partes opcionais da gramática, como blocos de declaração vazios ou listas de argumentos vazias.

Listing 3: Parte do pascal_parser.py (Regra de Programa e Expressão)

```
# ... importações de tokens e nós da AST ...

# Precedência de operadores (exemplo)
precedence = (
    ('left', 'EQ', 'NEQ', 'LT', 'LE', 'GT', 'GE'),
    ('left', 'PLUS', 'MINUS', 'OR_OP'),
    ('left', 'TIMES', 'DIVIDE', 'DIV_OP', 'MOD_OP', 'AND_OP'),
    ('right', 'NOT_OP'),
    ('right', 'UMINUS'),
)

def p_program(p):
    'program : PROGRAM ID SEMICOLON block DOT'
    p[0] = ProgramNode(IDNode(p[2], p.lineno(2)), p[4])
    p[0].lineno = p.lineno(1)

def p_expression(p):
    '''expression : term
                  | expression PLUS term
                  | expression MINUS term
                  | expression OR_OP term'''
    if len(p) == 2:
        p[0] = p[1]
    else:
        p[0] = BinOpNode(p[1], p.slice[2].type, p[3], p.lineno(2))

# ... outras regras de produção ...

def p_error(p):
    if p:
        print(f"Erro de Sintaxe: Token '{p.value}' (Tipo: {p.type}) inesperado na linha {p.lineno}")
    else:
        print("Erro de Sintaxe: Fim de arquivo inesperado (EOF) ou erro irrecuperável.")

parser = yacc.yacc()
```

5 Árvore de Sintaxe Abstrata (AST)

As classes que definem os nós da AST estão centralizadas no ficheiro `ast_nodes.py`. Cada classe herda de uma classe base `Node` e representa uma construção sintática da linguagem Pascal. Por exemplo, `ProgramNode`, `AssignmentNode`, `IfStatementNode`, `BinOpNode`, `IDNode`, `NumberNode`, etc. Os atributos de cada nó armazenam os seus componentes (e.g., um `BinOpNode` tem `left`, `op`, `right`) e, crucialmente, o `lineno` para rastreamento de erros. A AST serve como uma representação intermediária que facilita as fases de análise semântica e geração de código, abstraindo detalhes puramente sintáticos. Durante a análise semântica, os nós da AST podem ser anotados com informações adicionais, como o `inferred_type` (tipo inferido da expressão ou variável representada pelo nó).

Listing 4: Excerto de ast_nodes.py (Representativo)

```
class Node:
    """Classe base para todos os nós da AST."""
    pass

class ProgramNode(Node):
    def __init__(self, id_node, block_node):
        self.id = id_node
        self.block = block_node
```

```
        self.lineno = id_node.lineno # Ou p.lineno(1) do parser

class BinOpNode(Node):
    def __init__(self, left, op, right, lineno):
        self.left = left
        self.op = op # O token do operador (e.g., 'PLUS', 'TIMES')
        self.right = right
        self.lineno = lineno
        self.inferred_type = None # Será preenchido pelo analisador semntico
```

6 Análise Semântica

A análise semântica, implementada em `semantic_analyzer.py`, é uma fase crítica que verifica a correção lógica e a coerência do programa para além da sua estrutura sintática. Utiliza uma **Tabela de Símbolos** com escopos aninhados e percorre a AST (usando o padrão Visitor, com métodos `visit_NodeType`) para realizar as verificações. Erros semânticos detetados são acumulados e, se presentes, impedem a continuação para a geração de código.

6.1 Tabela de Símbolos (`SymbolTable`)

A classe `SymbolTable` é fundamental para a análise semântica, gerindo os identificadores e os seus atributos em diferentes escopos.

- **Escopos Aninhados:** Cada função cria um novo escopo, cujo pai (`parent`) é o escopo que a contém (neste projeto, tipicamente o escopo global). O nome do escopo (`scope_name`) ajuda na depuração e identificação.
- **Armazenamento de Símbolos:** A tabela armazena um dicionário `symbols` com informações sobre cada identificador:
  - `kind`: Tipo do símbolo (e.g., 'variable', 'function', 'parameter', 'program').
  - `type`: Tipo de dados base (e.g., 'INTEGER', 'REAL', 'BOOLEAN', 'STRING', 'ARRAY', ou o tipo de retorno para funções).
  - `full_type_spec`: Para tipos complexos como arrays, armazena detalhes como limites e tipo dos elementos. Para funções, armazena os detalhes dos parâmetros e o tipo de retorno completo.
  - `lineno`: Linha da declaração.
  - `address`: Endereço absoluto para variáveis globais (gerido por `global_address_counter` no `SemanticAnalyzer`) ou offset relativo ao Frame Pointer (FP) para variáveis locais e parâmetros.
  - `size`: Número de posições de memória que o símbolo ocupa (e.g., 1 para tipos simples, número de elementos para arrays).
  - `is_global`: Booleano que indica se a variável é global.
- **Adição de Símbolos (`add_symbol`):** Verifica redeclarações no escopo atual. Calcula o tamanho (`size`) do símbolo, especialmente para arrays. Atribui o `address` (offset) para variáveis locais (usando `next_local_offset`, que decresce a partir de -1) e para parâmetros (usando `next_param_offset`, que cresce a partir de +2). Atualiza `total_local_var_size` e `total_param_size` no escopo da função, informações cruciais para a alocação de stack frames pela EWVM.
- **Resolução de Nomes (`lookup`):** Procura um símbolo no escopo atual. Se não encontrado, a busca continua recursivamente nos escopos pais até ao escopo global.
- **Gestão de Escopos:** Os métodos `enter_scope` e `exit_scope` permitem a navegação na hierarquia de escopos durante a análise.

6.2 Processo de Análise Semântica

O `SemanticAnalyzer` percorre a AST, aplicando as seguintes verificações e ações:

- **Inicialização de Built-ins:** No método `_initialize_built_ins`, procedimentos e funções pré-definidas (e.g., `WRITELN`, `READLN`, `WRITE`, `READ`, `LENGTH`) são adicionados ao escopo global com as suas assinaturas (tipos de parâmetros esperados, tipo de retorno, se aceitam argumentos variáveis, se os argumentos de leitura devem ser L-values).

- **Declarações:**
  - `visit_VarDeclNode`: Para cada variável declarada, calcula-se o seu tamanho e regista-se na tabela de símbolos do escopo atual, distinguindo entre variáveis globais (endereço absoluto) e locais (offset relativo ao FP). Verifica a validade dos limites de arrays.
  - `visit_FunctionDeclarationNode`: Adiciona o nome da função à tabela de símbolos do escopo pai. Cria um novo escopo para a função, onde os seus parâmetros formais são adicionados (com os respetivos tipos e offsets). O bloco da função é então visitado neste novo escopo. Os escopos de função são armazenados em `self.function_scopes` para uso posterior pelo gerador de código.
- **Verificação de Tipos e Uso de Identificadores:**
  - `visit_IDNode`: Verifica se o identificador foi declarado usando `lookup`. Retorna o tipo do símbolo. Trata `TRUE` e `FALSE` como literais do tipo `BOOLEAN`. Identifica o uso do nome da função dentro do seu próprio corpo como referência ao seu valor de retorno.
  - `visit_ArrayAccessNode`: Confirma que o identificador base é um array, que o tipo da expressão do índice é `INTEGER`, e retorna o tipo dos elementos do array.
  - `visit_AssignmentNode`: Verifica a compatibilidade de tipos entre o lado esquerdo (LHS) e o lado direito (RHS), permitindo a atribuição de `INTEGER` a `REAL`. Um tratamento especial é dado à atribuição ao nome da função dentro do seu corpo, que define o valor de retorno da função, verificando a compatibilidade com o tipo de retorno declarado.
  - `visit_BinOpNode` e `visit_UnaryOpNode`: Inferem o tipo resultante de operações aritméticas (+, -, *, /, DIV, MOD), relacionais (=, ¡¿, ¡, ¡=, ¿, ¿=) e lógicas (AND, OR, NOT), reportando erros para combinações de tipos inválidas.
  - `visit_FunctionCallNode`: Compara o número e os tipos dos argumentos fornecidos na chamada com os parâmetros esperados, quer para funções built-in (com base na sua `params_definition`) quer para funções definidas pelo utilizador. Verifica restrições como `arg_must_be_lvalue` para `READLN`.
  - `visit_ReadLnCallNode` e `visit_WriteLnCallNode`: Delegam a análise para `visit_FunctionCallNode` com os respetivos nomes de built-in.
  - `visit_IfStatementNode`, `visit_WhileStatementNode`: Asseguram que a expressão da condição resulta num tipo `BOOLEAN`. `visit_ForStatementNode`: Verificam se a variável de controlo e as expressões de limite são do tipo `INTEGER`.
- **Anotação da AST:** Durante a travessia, nós da AST que representam expressões ou o uso de variáveis são anotados com o seu tipo inferido através do atributo `inferred_type`. Esta informação é crucial para a fase de geração de código tomar decisões corretas (e.g., qual instrução de escrita usar para `WRITELN`).

Quaisquer erros semânticos detetados são armazenados na lista `self.errors`. Se esta lista não estiver vazia no final da análise, a compilação é interrompida.

Listing 5: Trecho de semantic_analyzer.py (Adição de Símbolo e Análise de Atribuição)

```
class SymbolTable:
    # ... (inicialização e lookup) ...
    def add_symbol(self, name, symbol_info):
        if name in self.symbols:
            raise SemanticError(f"Erro Semntico: ID '{name}' já declarado...")

        kind = symbol_info['kind']; size = 1 # Tamanho padrão
        full_type_spec = symbol_info.get('full_type_spec')
        if isinstance(full_type_spec, dict) and full_type_spec.get('type') == 'ARRAY':
            # ... (cálculo do 'size' para arrays) ...
            pass
        symbol_info['size'] = size

        if kind == 'parameter':
            symbol_info['address'] = self.next_param_offset
            self.next_param_offset += size
            self.total_param_size += size
        elif kind == 'variable':
            if not symbol_info.get('is_global'): # Variável local
                symbol_info['address'] = self.next_local_offset
                self.next_local_offset -= size
```

```
                self.total_local_var_size += size
        self.symbols[name] = symbol_info

class SemanticAnalyzer:
    # ... (inicialização, _initialize_built_ins, analyze, métodos visit_ genéricos) ...
    def visit_AssignmentNode(self, node):
        lhs_var_node = node.var
        lhs_type = self.visit(lhs_var_node)
        rhs_type = self.visit(node.expr)
        # ... (verificação de atribuição ao nome da função para valor de retorno) ...
        # ... (verificação de compatibilidade de tipos para atribuição normal) ...
        pass # Simplificado

    def visit_FunctionDeclarationNode(self, node):
        # ... (recolhe detalhes dos params, adiciona função ao escopo pai) ...
        old_scope = self.current_scope
        self.current_scope = self.current_scope.enter_scope(node.id.name.upper())
        self.function_scopes[node.id.name.upper()] = self.current_scope
        for param_detail in param_details_for_symbol: # param_details_for_symbol colhidos antes
            # ... (prepara param_data e adiciona ao self.current_scope) ...
            pass # Simplificado
        self.visit(node.block)
        self.current_scope = old_scope
```

7 Geração de Código

A fase de geração de código, implementada em `code_generator.py`, traduz a AST (já verificada e anotada pelo analisador semântico) num conjunto de instruções para a máquina virtual de pilha EWVM (documentação em `https://ewvm.epl.di.uminho.pt/manual`). Esta fase também utiliza o padrão Visitor. O código gerado é uma sequência de mnemónicas textuais que podem ser posteriormente processadas pela EWVM.

7.1 Princípios Gerais da Geração para EWVM

- **Ponto de Entrada e Saída:** A execução inicia com a instrução `start`, que, segundo o manual da EWVM, configura os apontadores globais (GP para dados globais, FP para o frame de stack inicial). O fim do programa principal é sinalizado por `stop`.
- **Gestão da Pilha e Registos:**
  - **GP (Global Pointer):** Usado para aceder a variáveis globais. Instruções como `PUSHG offset_GP` e `STOREG offset_GP` são usadas.
  - **FP (Frame Pointer):** Usado como referência para aceder a parâmetros e variáveis locais dentro de uma função. Parâmetros são acedidos com offsets positivos (e.g., FP+2, FP+3,...) e locais com offsets negativos (e.g., FP-1, FP-2,...). Instruções como `PUSHL offset_FP` e `STOREL offset_FP` são usadas.
  - **SP (Stack Pointer):** Gere o topo da pilha de execução.
- **Convenção de Chamada de Função:**
  - O chamador empilha os argumentos na pilha (da esquerda para a direita na declaração Pascal).
  - O chamador empilha o endereço da função a ser chamada usando `PUSHA label_funcao`.
  - A instrução `CALL` da EWVM transfere o controlo, salva o endereço de retorno e o FP antigo na pilha, e atualiza FP para o valor atual de SP (apontando para a base do novo frame, logo após os argumentos e informação de ligação).
  - Dentro da função chamada (callee), `PUSHN N` aloca espaço na pilha para N variáveis locais (inicializadas a zero pela EWVM).
  - Para retornar, o valor de retorno (se houver) é deixado no topo da pilha. A instrução `RETURN` da EWVM restaura SP para o valor de FP (libertando locais), restaura o FP antigo e o PC (endereço de retorno).
  - Após o retorno, o chamador remove os argumentos da pilha usando `POP N`, onde N é o número total de células de memória ocupadas pelos argumentos.

- **Rótulos e Saltos:** Rótulos são gerados para destinos de saltos. Instruções como `JUMP label` (salto incondicional) e `JZ label` (salta se o topo da pilha for zero/falso) são usadas para implementar estruturas de controlo.

7.2 Tradução de Construções Pascal para Instruções EWVM

- **Atribuições (`visit_AssignmentNode`):**
  - **Variável Simples:** Avalia-se a expressão RHS (o valor fica no topo da pilha). Depois, `STOREG offset` ou `STOREL offset` armazena o valor.
  - **Elemento de Array (`A[i] := expr`):**
    1. Empilha-se o endereço base do array (via `PUSHG` ou `PUSHL`).
    2. Avalia-se e empilha-se o índice `i`. Ajusta-se pelo limite inferior se necessário (`PUSHI low`, `SUB`).
    3. `PADD` calcula o endereço do elemento `A[i]`, que fica no topo da pilha.
    4. Avalia-se a `expr` RHS (o seu valor fica no topo da pilha).
    5. `STORE 0` armazena o valor da `expr` no endereço `A[i]` (que está abaixo do valor na pilha).
  - **Valor de Retorno de Função (`NomeFunc := expr`):** A `expr` é avaliada e o seu valor fica no topo da pilha, pronto para ser usado pela instrução `RETURN`.
- **Acesso a Variáveis e Arrays (Leitura):**
  - **Variáveis Simples (`visit_IDNode`):** `PUSHG offset` ou `PUSHL offset`. Literais `TRUE`/`FALSE` geram `PUSHI 1`/`PUSHI 0`.
  - **Elementos de Array (`expr := A[i]`):** Semelhante à atribuição, calcula-se o endereço do elemento com `PADD`. Depois, `LOAD 0` carrega o valor desse endereço para o topo da pilha.
- **Literais e Operações:**
  - `visit_NumberNode`, `visit_StringLiteralNode`: Geram `PUSHI`, `PUSHF`, `PUSHS`.
  - `visit_BinOpNode`: Os operandos são avaliados. As operações Pascal são mapeadas para instruções EWVM: '+' → `ADD`, '-' → `SUB`, '*' → `MUL`, '/' (divisão real) → `FDIV`, 'div' (divisão inteira) → `DIV`, 'mod' → `MOD`. Operadores relacionais ('=', '¡¿', '¡', '¡=', '¿', '¿=') mapeiam para `EQUAL`, `INF`, `INFEQ`, `SUP`, `SUPEQ` (com `NOT` adicional para '¡¿'). Operadores lógicos 'and'/'or' mapeiam para `AND`/`OR`.
  - `visit_UnaryOpNode`: '-' unário é implementado como `PUSHI 0, SWAP, SUB`. 'not' mapeia para `NOT`.
- **Estruturas de Controlo:**
  - `visit_IfStatementNode`: Avalia a condição. `JZ else_label` salta se falso. O bloco `then` é gerado, seguido de `JUMP endif_label` (se houver `else`). O `else_label` marca o início do bloco `else` (se existir). `endif_label` marca o fim da estrutura.
  - `visit_WhileStatementNode`: Um rótulo marca o início da avaliação da condição (`cond_label`). A condição é avaliada; `JZ end_label` salta para fora do ciclo se falso. O corpo do ciclo é gerado, seguido de `JUMP cond_label`. `end_label` marca a saída do ciclo.
  - `visit_ForStatementNode`: A atribuição inicial é feita. Um rótulo (`condition_label`) marca o ponto de verificação da condição. A variável de controlo é comparada com o limite (`INFEQ` para 'TO', `SUPEQ` para 'DOWNTO'). `JZ end_loop_label` sai se a condição do loop for falsa. Caso contrário, `JUMP body_label` executa o corpo. Após o corpo, a variável de controlo é incrementada/decrementada (`PUSHI 1`, `ADD`/`SUB`, `STOREL`/`STOREG`) e salta-se de volta para `condition_label`.
- **Entrada/Saída (Built-ins, tratados em `visit_FunctionCallNode`):**
  - `WRITELN`/`WRITE`: Para cada argumento, o seu valor é empilhado. Com base no tipo inferido, `WRITEI`, `WRITEF`, `WRITES` (ou `WRITEI` para BOOLEAN) é emitido. `WRITELN` (a instrução VM) é emitida no final para `WRITELN` Pascal.
  - `READLN`/`READ`: Para cada argumento (variável ou elemento de array):
    - Se for array, o endereço do elemento é preparado com `PADD`.
    - `READ` (instrução VM) lê a entrada.
    - `ATOI` ou `ATOF` converte a string lida para o tipo numérico apropriado.
    - `STOREG`/`STOREL` (para variáveis) ou `STORE 0` (para arrays após `PADD`) armazena o valor.

    - `READLN` Pascal pode consumir o resto da linha (e.g., emitindo `WRITELN` VM se não houver instrução específica de "consumir linha").
  - `LENGTH`: O argumento string é empilhado, seguido de `STRLEN`.

Listing 6: Trecho de code_generator.py (Geração para Função e Array)

```python
class CodeGenerator:
    # ... (inicialização, emit, new_label, generate_code) ...
    def visit_FunctionDeclarationNode(self, node):
        function_name_upper = node.id.name.upper()
        self.emit(f"{function_name_upper}:")
        function_scope = self.semantic_analyzer.function_scopes.get(function_name_upper)

        # Prólogo EWVM: CALL já tratou de FP, PC. Apenas alocar locais.
        if function_scope and function_scope.total_local_var_size > 0:
            self.emit(f"PUSHN {function_scope.total_local_var_size}")

        original_active_scope = self.active_scope_for_lookups
        self.active_scope_for_lookups = function_scope
        if node.block: self.visit(node.block) # Visita corpo da função
        self.active_scope_for_lookups = original_active_scope

        # Epílogo EWVM: RETURN trata de restaurar SP, FP, PC e usa valor no topo.
        self.emit("RETURN")

    def visit_FunctionCallNode(self, node):
        func_name_upper = node.id.name.upper()
        function_symbol = self.semantic_analyzer.current_scope.lookup(func_name_upper)
        # ... (Tratamento de built-ins como LENGTH, WRITE, READLN) ...

        # Funções definidas pelo utilizador:
        arg_total_size_slots = 0
        for arg_expr in node.args: # Empilha argumentos (esquerda para direita)
            self.visit(arg_expr)
            # ... (cálculo de arg_total_size_slots para POP N) ...
            param_def = function_symbol['params'][arg_total_size_slots] # Assume que existe
            arg_total_size_slots += param_def.get('size', 1)

        self.emit(f"PUSHA {func_name_upper}") # Endereço da função
        self.emit("CALL")
        if arg_total_size_slots > 0:
           self.emit(f"POP {arg_total_size_slots}") # Chamador limpa argumentos

    def visit_ArrayAccessNode(self, node): # Leitura de Array
        array_symbol = self._get_symbol_from_active_scope(node.id.name.upper())
        # ... (empilha endereço base: PUSHG/PUSHL) ...
        # ... (avalia e empilha índice, ajusta por low_bound com SUB) ...
        self.emit("PADD") # Calcula endereço do elemento
        self.emit("LOAD 0") # Carrega valor do elemento
```

8 Testes

Para verificar a correção do compilador, foi adotada uma estratégia de testes progressiva, cobrindo as diferentes fases. O script `main.py` inclui uma função `run_single_test` que permite executar o compilador para um dado código Pascal e observar a saída das fases de análise e o código VM gerado.

Uma suite de testes é definida no `main.py` através da lista `test_files_info`, que mapeia nomes de ficheiros `.pas` (localizados no diretório `semantic_tests/`) para descrições curtas. Estes ficheiros contêm programas Pascal que testam:

- **Análise Léxica e Sintática:** Casos simples e complexos de declarações, comandos e expressões para garantir que a AST é construída corretamente ou que erros sintáticos são reportados.
- **Análise Semântica:**

  - Declarações válidas e inválidas (e.g., uso de variável não declarada, redeclaração).
  - Atribuições e expressões com tipos compatíveis e incompatíveis.
  - Uso correto e incorreto de arrays (e.g., índice não inteiro, acesso a não-array).
  - Condições em estruturas de controlo (IF, WHILE, FOR) com tipos corretos e incorretos.
  - Chamadas a funções/procedimentos built-in e definidos pelo utilizador com número e tipo de argumentos corretos e incorretos.
- **Geração de Código:** Para os casos semanticamente corretos, o código VM gerado é inspecionado para verificar se implementa a lógica esperada do programa Pascal. Os exemplos do enunciado do projeto (Olá Mundo, Maior de 3, Fatorial, etc.) são usados como base para estes testes, comparando a saída do compilador com o comportamento esperado na VM.

A função `main.py` também permite a execução de um ficheiro Pascal específico passado como argumento na linha de comandos, facilitando testes individuais e a depuração.

9 Dificuldades e Trabalho Futuro

O desenvolvimento deste compilador para uma sublinguagem de Pascal, embora tenha alcançado os seus objetivos principais, apresentou desafios e revelou oportunidades para melhorias e expansões futuras. Analisamos o projeto à luz dos critérios de avaliação estabelecidos, identificando pontos fortes, dificuldades encontradas e áreas para trabalho futuro.

9.1 Processamento Correto de Programas Pascal

Estado Atual e Dificuldades: O compilador demonstra capacidade de processar corretamente um subconjunto significativo de programas Pascal, abrangendo as principais construções da linguagem. A análise léxica, sintática e semântica, juntamente com a geração de código para a EWVM, foi implementada para cobrir declarações, expressões, estruturas de controlo e funções. A principal dificuldade na garantia da correção total reside na vasta quantidade de casos de canto (*edge cases*) e interações complexas entre diferentes funcionalidades da linguagem. Embora uma suite de testes tenha sido desenvolvida (`semantic_tests/`), cobrir exaustivamente todos os cenários possíveis é um desafio contínuo. A fidelidade da tradução para as especificidades da EWVM, especialmente em relação à gestão da pilha e convenções de chamada, exigiu depuração e refinamento iterativos.

Trabalho Futuro para Melhorar a Correção:

- **Expansão da Suite de Testes:** Desenvolver um conjunto de testes ainda mais abrangente, incluindo programas Pascal mais complexos, testes de stress, e casos que explorem os limites das funcionalidades implementadas (e.g., aninhamento profundo de estruturas, expressões muito longas, limites de arrays). Automatizar a comparação da saída da VM com saídas esperadas seria ideal.
- **Tratamento de Erros Mais Robusto:** Melhorar o mecanismo de recuperação de erros no parser para que, após um erro sintático, o compilador possa tentar continuar a análise e reportar múltiplos erros, em vez de parar no primeiro. As mensagens de erro semântico, embora informativas, poderiam também incluir sugestões mais específicas.
- **Testes de Integração com a EWVM:** Intensificar os testes de execução do código VM gerado diretamente na EWVM para validar o comportamento em tempo de execução e a correção da interação com a máquina alvo.

9.2 Organização do Código e Documentação

Estado Atual e Dificuldades: O projeto foi estruturado de forma modular, com cada fase do compilador implementada no seu próprio ficheiro Python (`lexer.py`, `pascal_parser.py`, `ast_nodes.py`, `semantic_analyzer.py`, `code_generator.py`, `main.py`). Esta modularidade facilitou o desenvolvimento e a manutenção. A AST e as classes como `SymbolTable` contribuem para uma boa organização interna. A documentação principal é este relatório, que procura detalhar a implementação. Uma dificuldade inerente a projetos de grupo é manter uma total consistência de estilo e documentação interna (comentários, docstrings) ao longo de todo o código.

Trabalho Futuro para Melhorar a Estrutura:

- **Reforço da Documentação Interna:** Adicionar ou completar *docstrings* para todas as classes, funções e métodos importantes, explicando o seu propósito, argumentos e valores de retorno. Comentários explicativos em secções de código mais complexas também seriam benéficos.

- **Adesão a Guias de Estilo:** Aplicar de forma mais rigorosa um guia de estilo Python (e.g., PEP 8) para melhorar a legibilidade e consistência do código. Ferramentas de *linting* podem auxiliar neste processo.
- **Refatoração:** Após a entrega inicial, revisitar o código para identificar oportunidades de refatoração, visando simplificar a lógica, reduzir redundâncias ou melhorar a clareza de certas secções.
- **Diagramas de Arquitetura:** Para além da descrição textual, incluir diagramas (e.g., UML ou diagramas de fluxo de dados) no relatório poderia ajudar a visualizar melhor a arquitetura do compilador e a interação entre os seus componentes.

9.3 Funcionalidade: Suporte às Principais Construções da Linguagem

Estado Atual e Dificuldades: O compilador suporta um conjunto robusto de construções do Pascal Standard, incluindo declarações de variáveis de tipos básicos e arrays, expressões aritméticas, lógicas e relacionais, as principais estruturas de controlo de fluxo (`IF`, `WHILE`, `FOR`) e subprogramas (`FUNCTION`). Funções built-in como `WRITELN`, `READLN`, `WRITE`, `READ` e `LENGTH` também foram implementadas. A principal dificuldade reside em definir o "limite"da sublinguagem a ser suportada e garantir que todas as interações entre as funcionalidades implementadas são tratadas corretamente.

Trabalho Futuro para Expandir a Funcionalidade:

- **Suporte a Procedimentos:** Implementar explicitamente `PROCEDURE` como uma construção distinta de `FUNCTION`, embora a funcionalidade atual de funções com retorno `VOID` possa simular procedimentos.
- **Tipos de Dados Adicionais:** Adicionar suporte para outros tipos de dados Pascal, como `CHAR`, `RECORD` (registos), ou até mesmo ponteiros (embora estes últimos aumentem consideravelmente a complexidade).
- **Mais Estruturas de Controlo:** Implementar `CASE` ou `REPEAT-UNTIL`.
- **Escopos Mais Complexos (Opcional):** Considerar o suporte para funções e procedimentos aninhados, o que exigiria uma gestão de escopos e da cadeia estática (static links) mais elaborada na tabela de símbolos e na geração de código para acesso a variáveis não locais.
- **Tratamento de Unidades/Módulos (Uses):** Embora fora do escopo típico de um primeiro compilador, permitir a compilação separada e o uso de unidades (units).

9.4 Eficiência: Desempenho do Compilador e do Código Gerado

Estado Atual e Dificuldades: O desempenho do próprio compilador (velocidade de compilação) não foi um foco primário, dado que Python e PLY foram escolhidos pela facilidade de desenvolvimento. Para os tamanhos de programas Pascal testados, o desempenho é considerado aceitável. Em relação à eficiência do código VM gerado, foram feitas traduções diretas das construções da AST para sequências de instruções VM, sem otimizações explícitas. A principal dificuldade seria introduzir passes de otimização sem aumentar excessivamente a complexidade do projeto.

Trabalho Futuro para Melhorar a Eficiência:

- **Otimização do Código Gerado (Conforme Sugerido como Extra no Enunciado):**
  - **Constant Folding:** Avaliar expressões constantes em tempo de compilação.
  - **Peephole Optimization:** Analisar pequenas sequências de instruções VM geradas para substituí-las por sequências mais eficientes.
  - **Eliminação de Código Morto/Inacessível:** Identificar e remover partes do código VM que nunca serão executadas.
  - **Gestão de Registos (se a VM suportasse):** Para VMs com registos, uma alocação inteligente de registos poderia melhorar significativamente o desempenho. A EWVM é baseada em pilha, limitando este tipo de otimização.
- **Análise de Desempenho do Compilador:** Se o compilador se tornasse lento para programas grandes, usar ferramentas de *profiling* Python para identificar gargalos e otimizar secções críticas do código do compilador (e.g., lookups na tabela de símbolos, travessias da AST).
- **Representação Intermediária (IR) Mais Rica:** Para otimizações mais avançadas, considerar a geração de uma Representação Intermediária (e.g., código de três endereços, SSA form) antes da geração final do código VM. A AST atual serve como uma IR de alto nível.

9.5 Demonstração Realizada Durante a Defesa

Preparação e Dificuldades Antecipadas: Uma demonstração eficaz requer a seleção cuidadosa de exemplos que ilustrem tanto as capacidades do compilador (processamento de código válido e complexo) como o seu tratamento de erros. Uma dificuldade pode ser gerir o tempo para mostrar as diferentes fases e responder a perguntas de forma clara.

Trabalho Futuro (Preparação para a Defesa):

- **Seleção Estratégica de Exemplos:** Preparar um conjunto de programas Pascal para a demonstração:
  - Um ou dois programas mais completos que usem várias funcionalidades (e.g., Fatorial com função, SomaArray, um que use IFs e WHILEs).
  - Exemplos curtos que demonstrem erros específicos (léxico, sintático, semântico de tipo, variável não declarada).
- **Script de Demonstração:** Ter um plano claro do que será mostrado, em que ordem, e o que será explicado em cada passo (e.g., mostrar o código Pascal, executar o compilador, mostrar a saída de erros ou o código VM gerado, e, se possível, executar o código VM na EWVM).
- **Materiais de Apoio Visuais:** Considerar o uso de slides simples para guiar a demonstração, destacando a arquitetura do compilador e os pontos chave de cada fase.
- **Antecipar Perguntas:** Pensar nas possíveis perguntas que podem surgir sobre a implementação, decisões de design e desafios.

Em suma, o projeto representa um esforço substancial na construção de um compilador funcional. As dificuldades encontradas são inerentes à complexidade da tarefa, e as áreas identificadas para trabalho futuro abrem caminho para um sistema ainda mais robusto, completo e eficiente.

10 Exemplos de Código Gerado

Para demonstrar o funcionamento prático do compilador, apresentamos dois exemplos de programas Pascal (já presentes no relatório inicial e validados com a implementação atual) e o respetivo código de máquina virtual gerado.

10.1 Exemplo 1: Determinação do Maior de Três Números

Código Pascal:

Listing 7: Maior3.pas

```
program Maior3;
var
    num1, num2, num3, maior: Integer;
begin
    Write('Introduza o primeiro número: '); {Implementado como WRITELN nos built-ins}
    ReadLn(num1);
    Write('Introduza o segundo número: ');
    ReadLn(num2);
    Write('Introduza o terceiro número: ');
    ReadLn(num3);
    if num1 > num2 then
        if num1 > num3 then maior := num1
        else maior := num3
    else
        if num2 > num3 then maior := num2
        else maior := num3;
    WriteLn('O maior é: ', maior);
end.
```

Código VM Gerado:

Listing 8: Saída para Maior3.pas

```
start
pushs 'Introduza o primeiro número: '
writes
read
atoi
storeg 0
pushs 'Introduza o segundo número: '
writes
read
atoi
```

```
storeg 1
pushs 'Introduza o terceiro número: '
writes
read
atoi
storeg 2
pushg 0
pushg 1
gt
jz ELSE0
pushg 0
pushg 2
gt
jz ELSE2
pushg 0
storeg 3
jump ENDIF3
ELSE2:
pushg 2
storeg 3
ENDIF3:
jump ENDIF1
ELSE0:
pushg 1
pushg 2
gt
jz ELSE4
pushg 1
storeg 3
jump ENDIF5
ELSE4:
pushg 2
storeg 3
ENDIF5:
ENDIF1:
pushs 'O maior é: '
writes
pushg 3
writei
writeln
stop
```

10.2 Exemplo 2: Cálculo do Fatorial

Código Pascal:

Listing 9: Fatorial.pas

```
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.
```

Código VM Gerado:

Listing 10: Saída para Fatorial.pas

```
start
pushs 'Introduza um número inteiro positivo:'
writes
writeln
read
atoi
storeg 0
pushi 1
storeg 2
pushi 1
storeg 1
jump FORCOND0
FORLOOP0:
pushg 2
pushg 1
mul
storeg 2
pushg 1
pushi 1
add
storeg 1
FORCOND0:
pushg 1
```

```
pushg 0
le
jnz FORLOOP0
ENDFOR0:
pushs 'Fatorial de '
writes
pushg 0
writei
pushs ': '
writes
pushg 2
writei
writeln
stop
```

Nota: A lógica do FOR no gerador de código pode variar ligeiramente (e.g., posição do `jump FORCOND0` e `ENDFORX:`), mas o fluxo deve ser equivalente. O exemplo acima reflete uma possível tradução correta. O uso de `jnz` após a condição `le` é para continuar o loop enquanto a condição for verdadeira.

10.3 Análise dos Exemplos

Os exemplos demonstram que o compilador traduz corretamente:

- Declaração e uso de variáveis globais (com `storeg`, `pushg`).
- Operações de entrada/saída (`read`, `atoi`, `writes`, `writei`, `writeln`).
- Estruturas condicionais `if-then-else` (com `jz`, `jump` e labels).
- Ciclos `for` (com labels, comparações como `le`, e saltos como `jnz`).
- Operações aritméticas e relacionais (`mul`, `add`, `gt`, `le`).

A análise semântica, como indicado pela ausência de erros reportados para estes exemplos válidos e a correta inferência de tipos (implícita na geração de código específico como `atoi` e `writei`), também é demonstrada.

11 Conclusão

O desenvolvimento deste projeto permitiu a construção de um compilador funcional para uma sublinguagem de Pascal, capaz de processar código-fonte desde a análise léxica até à geração de código para uma máquina virtual de pilha. A utilização da biblioteca PLY para as fases iniciais de análise provou ser eficaz, permitindo concentrar esforços na lógica mais complexa da análise semântica e da geração de código.

Os principais desafios, como a gestão de escopos na tabela de símbolos, a verificação e inferência de tipos, e a tradução de estruturas de controlo e chamadas de função para um código de baixo nível, foram abordados e resolvidos, resultando num sistema que cumpre os objetivos fundamentais do projeto. Os testes realizados com diversos programas Pascal, incluindo os exemplos fornecidos no enunciado, demonstram a capacidade do compilador em lidar com as construções da linguagem suportadas e gerar código executável correto.

Este projeto proporcionou uma valiosa experiência prática na aplicação dos conceitos teóricos de compiladores, solidificando a compreensão das suas diversas fases e interdependências.