Relatório de Projeto: Construção de um Compilador para Pascal Standard

Grupo 16

Fábio Mendes Castelhano — *A105728*
João Manuel Pinto Alves — *A108653*
Tiago SIlva Costa — *A108657*

---

1. Introdução

Este relatório descreve o desenvolvimento de um compilador para uma variante da linguagem Pascal, capaz de realizar análise léxica, sintática, semântica e geração de código para a VM que foi disponibilizada durante o semestre.

---

2. Arquitetura do Sistema

O compilador foi desenvolvido em Python utilizando a biblioteca **PLY (Python Lex-Yacc)**. A estrutura do projeto é modular, organizada da seguinte forma:

- `pascal_analex.py` : Analisador Léxico.
- `pascal_anasin.py` : Analisador Sintático (Parser).
- `pascal_anasem.py` : Gestão da Tabela de Símbolos e análise semântica.
- `pascal_codegen.py` : Gerador de Código Máquina para a VM.
- `main.py` : Script principal que coordena o fluxo de compilação.

---

3. Descrição dos Módulos

3.1 Analisador Léxico (`pascal_analex.py`)

O analisador léxico é o primeiro estágio do processo de compilação, responsável por ler o fluxo de caracteres do código-fonte e agrupá-los em unidades lógicas denominadas **tokens**. Este módulo utiliza a biblioteca `ply.lex` e foi projetado para lidar com as particularidades da linguagem Pascal.

A. Tratamento de Palavras Reservadas e Case-Insensitivity

A linguagem Pascal é tradicionalmente insensível à capitalização (*case-insensitive*). Para implementar esse comportamento, utilizamos um dicionário de mapeamento e uma função de identificação que normaliza os lexemas para minúsculas antes da verificação.

```
# Trecho de pascal_analex.py: Mapeamento e Regra de Identificadores
reserved_map = {
    'program': 'PROGRAM', 'begin': 'BEGIN', 'end': 'END',
    'var': 'VAR', 'if': 'IF', 'then': 'THEN', 'else': 'ELSE',
    'function': 'FUNCTION', 'procedure': 'PROCEDURE',
    'readln': 'READLN', 'writeln': 'WRITELN'
}

def t_ID(t):
    r'[a-zA-Z_][a-zA-Z0-9_]*'
    # Converte para minúsculas para garantir a insensibilidade à caixa
```

```
    t.type = reserved_map.get(t.value.lower(), 'ID')
    return t
```

B. Operadores Compostos e Símbolos

Certos operadores no Pascal consistem em dois caracteres que devem ser interpretados como um único token (tokens multi-caractere). No PLY, definimos estes operadores como variáveis de string (`t_TOKENNAME`) para garantir que o analisador léxico utilize a regra da "maior correspondência" (*longest match*). Isso evita, por exemplo, que o operador de atribuição `:=` seja incorretamente fragmentado nos tokens `:` e `=`.

```
# Trecho de pascal_analex.py: Definição de tokens multi-caractere
t_OP_ASSIGN = r':='   # Atribuição
t_OP_LE     = r'<='   # Menor ou Igual
t_OP_GE     = r'>='   # Maior ou Igual
t_OP_NE     = r'<>'   # Diferente
t_OP_DOTDOT = r'\.\.' # Intervalo
```

C. Ignorando Elementos Não-Sintáticos (Comentários e Espaços)

Para que as fases de análise sintática e semântica foquem exclusivamente na estrutura lógica do programa, o lexer filtra elementos que não possuem valor semântico para a execução. Isso inclui espaços em branco, tabulações e os diversos estilos de comentários permitidos no Pascal. Utilizamos expressões regulares que abrangem múltiplas linhas para garantir que blocos de comentários, mesmo os mais extensos, sejam completamente descartados pelo analisador.

```
# Trecho de pascal_analex.py: Ignorar espaços e tratar comentários
t_ignore = ' \t'

def t_COMMENT(t):
    # Suporta (* ... *), { ... } e //
    r'(\(\*[\s\S]*?\*\))|(\{[\s\S]*?\})|(//[^\n]*)'
    pass # O conteúdo capturado é ignorado

def t_newline(t):
    r'\n+'
    t.lexer.lineno += len(t.value) # Atualiza contador de linhas

def t_error(t):
    # Tratamento de caracteres não reconhecidos
    print('Carácter desconhecido: ', t.value[0], 'Linha: ', t.lexer.lineno)
    t.lexer.skip(1)
```

3.2 Analisador Sintático (`pascal_anasin.py`)

O analisador sintático é responsável por verificar se a sequência de tokens gerada pelo lexer segue as regras gramaticais da linguagem Pascal. Utilizamos o **PLY Yacc**, que implementa um algoritmo de análise **LALR (Bottom-Up)**.

A. Estrutura da Gramática e Recursividade à Esquerda

Diferente de parsers *Top-Down*, o PLY Yacc processa de forma eficiente a **recursividade à esquerda**. Esta técnica foi amplamente utilizada em nossa gramática para definir listas de declarações, comandos e expressões, garantindo que a pilha de análise não transborde e que a associatividade dos operadores seja respeitada.

```
# Trecho de pascal_anasin.py: Exemplo de recursividade à esquerda
def p_lista_declaracoes_tipo(p):
    '''
    lista_declaracoes_tipo : lista_declaracoes_tipo declaracao_tipo ';'
                           | declaracao_tipo ';'
    '''
```

```
    if len(p) == 4:
        p[0] = p[1] + [p[2]] # Acumula na lista existente
    else:
        p[0] = [p[1]]        # Caso base: primeira declaração
```

B. Definição de Precedência e Associatividade

Para resolver ambiguidades clássicas em expressões matemáticas (como garantir que a multiplicação ocorra antes da adição) sem sobrecarregar a gramática com inúmeras sub-regras, utilizamos a variável precedence. Esta configuração define níveis de prioridade e a direção da associatividade para os operadores.

```
# Trecho de pascal_anasin.py: Hierarquia de operadores
precedence = (
    ('left', 'AND'),
    ('left', '=', 'OP_NE', 'OP_LE', 'OP_GE', '<', '>'),
    ('left', '+', '-'),
    ('left', '*', 'DIV', 'MOD'),
)
```

C. Construção da Árvore Sintática Abstrata (AST)

O resultado bem-sucedido da análise sintática não é apenas um "sim" ou "não", mas a construção de uma AST (Abstract Syntax Tree). Cada regra de produção no arquivo retorna uma tupla que representa um nó na árvore. Essa estrutura é vital para que as etapas subsequentes (Semântica e Geração de Código) possam "caminhar" pelo programa de forma organizada.

```
# Trecho de pascal_anasin.py: Geração de nós da AST
def p_programa(p):
    '''programa : PROGRAM ID ';' lista_definicoes bloco '.' '''
    # Cria o nó raiz da árvore com o nome do programa e seus componentes
    p[0] = ('PROGRAM', p[2], p[4], p[5])

def p_atribuicao(p):
    '''atribuicao : ID OP_ASSIGN expressao'''
    p[0] = ('ASSIGN', p[1], p[3])
```

D. Tratamento de Erros Sintáticos

Implementamos uma função de recuperação de erros (p_error) que captura tokens inesperados. Isso permite que o compilador forneça um feedback útil ao utilizador, indicando o token problemático e a linha exata onde a estrutura gramatical foi violada.

```
def p_error(p):
    if p:
        print(f"ERRO SINTÁTICO: Token inesperado '{p.value}' na linha {p.lineno}")
    else:
        print("ERRO SINTÁTICO: Fim de ficheiro inesperado")
```

3.3 Analisador Semântico e Tabela de Símbolos (`pascal_anasem.py`)

A análise semântica representa a "camada de inteligência" do compilador. Enquanto o parser valida a estrutura, o analisador semântico garante que as regras de validade da linguagem sejam respeitadas, utilizando a **Tabela de Símbolos (TS)** como estrutura de suporte central.

A. Gestão de Escopos e Pilha de Contextos

O Pascal permite a declaração de variáveis globais e locais (dentro de funções e procedimentos). Para gerir isto, a classe `SymbolTable` implementa uma **pilha de escopos** (`scope_stack`). Quando o compilador entra numa função, um novo contexto é empilhado; ao sair, esse contexto é removido, garantindo o isolamento das variáveis locais.

```python
# Trecho de pascal_anasem.py: Controle de Escopo
def enter_scope(self):
    """Cria um novo dicionário para variáveis locais e reinicia o offset."""
    self.scope_stack.append({})
    self.offset_stack.append(0)

def exit_scope(self):
    """Remove o escopo local, voltando ao nível anterior (ex: Global)."""
    if len(self.scope_stack) > 1:
        self.scope_stack.pop()
        self.offset_stack.pop()
```

B. Tabela de Símbolos e Built-ins

A Tabela de Símbolos armazena o nome, tipo, categoria (VAR, FUNCTION, PROCEDURE) e o endereço de memória (offset) de cada identificador. Além disso, ela já nasce "povoada" com funções nativas do Pascal para que o compilador as reconheça sem declaração prévia pelo utilizador.

```python
# Trecho de pascal_anasem.py: Inicialização de funções nativas
def _init_builtins(self):
    self.add('writeln', ('TYPE', 'VOID'), 'PROCEDURE')
    self.add('readln', ('TYPE', 'VOID'), 'PROCEDURE')
    self.add('length', ('TYPE', 'INTEGER'), 'FUNCTION')
```

C. Verificação de Tipos (Type Checking)

Integrado diretamente nas regras do parser, o analisador semântico realiza a conferência de tipos em tempo de compilação. Se o utilizador tentar somar uma string com um inteiro ou atribuir um valor booleano a uma variável inteira, o sistema interrompe a geração de código e reporta o erro.

```python
# Trecho de pascal_anasin.py: Verificação semântica em operações binárias
def p_expressao_binaria(p):
    # ... extração de tipos dos operandos ...
    if op in ['+', '-', '*', 'DIV', 'MOD']:
        if tipo1 == ('TYPE', 'INTEGER') and tipo2 == ('TYPE', 'INTEGER'):
            tipo_resultado = ('TYPE', 'INTEGER')
        else:
            print(f"ERRO SEMÂNTICO: Operação '{op}' requer INTEGERS.")
```

D. Lookup e Visibilidade

O método lookup é responsável por encontrar um identificador percorrendo a pilha do escopo mais interno (local) para o mais externo (global). Isso permite o sombreamento de variáveis, onde uma variável local pode ter o mesmo nome de uma global sem causar conflito.

```python
def lookup(self, name, line=0):
    name_lower = name.lower()
    # Percorre a pilha do topo (local) para a base (global)
    for scope in reversed(self.scope_stack):
        if name_lower in scope:
            return scope[name_lower]
```

3.4 Gerador de Código (`pascal_codegen.py`)

O gerador de código é a etapa final do compilador. Ele percorre a Árvore Sintática Abstrata (AST) gerada pelo parser e traduz cada nó em instruções de baixo nível para a Máquina Virtual (VM) de pilha. Este módulo permite processar diferentes tipos de nós de forma organizada e extensível.

A. O Motor de Geração

O gerador utiliza despacho dinâmico para encontrar o método de visita correspondente ao tipo de nó da AST (ex: `visit_ASSIGN` , `visit_IF`). Isso permite que a lógica de tradução seja isolada por tipo de

comando.

```
# Trecho de pascal_codegen.py: Implementação do Visitor
def visit(self, node):
    if not isinstance(node, tuple):
        return
    node_type = node[0]
    method_name = f'visit_{node_type}'
    # Busca o método visit_TIPO, senão usa o genérico
    visitor = getattr(self, method_name, self.generic_visit)
    return visitor(node)
```

B. Gestão de Memória e Variáveis

O gerador traduz as declarações da Tabela de Símbolos em instruções de alocação de memória. Variáveis simples utilizam espaço na pilha, enquanto arrays são alocados dinamicamente na heap.

STOREG / PUSHG: Utilizados para acessar variáveis no escopo global.

STOREL / PUSHL: Utilizados para variáveis locais e parâmetros dentro de subprogramas.

ALLOCN: Reserva múltiplos espaços para arrays, guardando o endereço inicial na pilha.

C. Tradução de Estruturas de Controle

Para comandos como IF, WHILE e FOR, o gerador utiliza etiquetas (labels) e saltos condicionais (JZ) ou incondicionais (JUMP).

```
# Trecho de pascal_codegen.py: Geração de código para o laço WHILE
def visit_WHILE(self, node):
    l_start = self.get_new_label() # Ex: L1
    l_end = self.get_new_label()   # Ex: L2

    self.emit_label(l_start)
    self.visit(node[1])       # Gera código para a Condição
    self.emit(f"JZ {l_end}")  # Se falso, salta para o fim

    self.visit(node[2])       # Gera código para o Corpo
    self.emit(f"JUMP {l_start}")
    self.emit_label(l_end)
```

D. Chamadas de Subprogramas e Funções

O gerador implementa o protocolo de ativação de funções. Ele reserva espaço para o valor de retorno (PUSHI 0), empilha os argumentos e utiliza CALL para desviar a execução para o rótulo da função. Após o retorno, limpa os argumentos da pilha com a instrução POP.

```
# Trecho de pascal_codegen.py: Chamada de função
def visit_CALL_EXP(self, node):
    self.emit("PUSHI 0") # Espaço para o retorno
    for arg in node[2]:
        self.visit(arg)  # Empilha argumentos
    self.emit(f"PUSHA {node[1]}")
    self.emit("CALL")
    self.emit(f"POP {len(node[2])}") # Limpeza da pilha
```

3.5 Orquestração do Compilador (`main.py`)

O arquivo `main.py` é o ponto de entrada (entry point) do sistema. A sua função principal é gerir o fluxo de dados entre os diferentes módulos, garantindo que o código-fonte percorra todas as fases da compilação de forma ordenada e segura.

A. Interface de Linha de Comando e Gestão de Ficheiros

O script utiliza o módulo `sys` para receber o caminho do ficheiro `.pas` via terminal. Ele é responsável por abrir o ficheiro, ler o conteúdo bruto e passá-lo para os analisadores.

```
# Trecho de main.py: Leitura do ficheiro fonte
with open(sys.argv[1], 'r') as f:
    content = f.read()
```

B. Pipeline de Execução

O main.py executa o processo em duas etapas fundamentais:

1. Fase de Análise: Chama o parser.parse(), que desencadeia a análise léxica (através do lexer), sintática e semântica (preenchendo a Tabela de Símbolos ts).
2. Fase de Síntese: Se a análise não detetar erros (ou seja, se o result da AST for válido), ele instancia o CodeGenerator e inicia a produção das instruções para a VM.

```
# Trecho de main.py: Coordenação das fases
result = parser.parse(content, lexer=lexer) # Léxica, Sintática e Semântica

if result:
    # Inicia a Geração de Código com a AST e a TS preenchida
    codegen = CodeGenerator(result, ts)
    codegen.generate()
```

C. Integração de Resultados

Ao centralizar a execução, o main.py permite que o utilizador final interaja com o compilador como uma ferramenta única. Ele garante que a Tabela de Símbolos preenchida durante a análise sintática/semântica seja a mesma utilizada pelo Gerador de Código, assegurando a consistência dos endereços (offsets) de memória em todo o programa gerado.

---

4. Funcionalidades Extras e Otimizações

Nesta seção, detalhamos o cumprimento dos requisitos opcionais e avançados propostos no enunciado do projeto.

4.1 Implementação de Subprogramas (Procedures e Functions)

Embora listados como elementos opcionais no enunciado, o grupo optou por implementar suporte completo a **Procedures** e **Functions**. Esta implementação permite uma estruturação modular do código Pascal e exigiu:

- **Gestão de Escopos Locais**: Isolamento de variáveis dentro de funções para evitar efeitos secundários no programa principal.
- **Passagem de Parâmetros**: Implementação da manipulação de argumentos na pilha da VM, utilizando offsets em relação ao *Frame Pointer*.
- **Retorno de Valores**: Mecanismo de atribuição ao nome da função (ex: `BinToInt := valor`) para passar o resultado de volta ao chamador.

4.2 Representação Intermédia

O grupo utilizou uma estratégia de compilação baseada em uma **Representação Intermédia**:

- O parser não gera o código final imediatamente; em vez disso, constrói uma **Árvore Sintática Abstrata (AST)**.
- Esta abordagem separa a análise lógica (Sintática/Semântica) da síntese de código (Codegen), o que facilita a manutenção do código como um todo.

4.3 Otimização de Código

Conforme planejado para o escopo deste projeto, o grupo não focou em fazer otimização de código. A prioridade do grupo foi garantir a **correção total** da tradução para a máquina virtual fornecida, assegurando que as estruturas iterativas (como o `for` e `while`) e as chamadas de funções funcionem de forma robusta e previsível.

---

5. Testes e Validação de Resultados

Esta seção apresenta a validação do compilador através dos casos de teste sugeridos no enunciado do projeto. Para cada exemplo, apresentamos o código-fonte em Pascal e o respectivo código de máquina gerado para a Máquina Virtual (VM).

5.1 Exemplo 1: Olá, Mundo!

Este teste valida a estrutura básica de um programa e a execução de procedimentos de saída de dados (`writeln`).

Código Pascal:

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

Código Máquina Virtual gerado:

```
    cmd
    START
            PUSHS "Ola, Mundo!"
            WRITES
            WRITELN
    STOP
```

5.2 Exemplo 2: Fatorial

Este teste valida a declaração de variáveis inteiras, leitura de dados (`readln`), estruturas de repetição (`for`) e operações aritméticas.

Código Pascal:

```
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 Máquina Virtual gerado:

```
    cmd
    START
            PUSHN 3
            PUSHS "Introduza um nÃºmero inteiro positivo:"
            WRITES
            WRITELN
            READ
```

```
            ATOI
            STOREG 0
            PUSHI 1
            STOREG 2
            PUSHI 1
            STOREG 1
    L1:
            PUSHG 1
            PUSHG 0
            INFEQ
            JZ L2
            PUSHG 2
            PUSHG 1
            MUL
            STOREG 2
            PUSHG 1
            PUSHI 1
            ADD
            STOREG 1
            JUMP L1
    L2:
            PUSHS "Fatorial de "
            WRITES
            PUSHG 0
            WRITEI
            PUSHS ": "
            WRITES
            PUSHG 2
            WRITEI
            WRITELN
    STOP
```

5.3 Exemplo 3: Verificação de Número Primo

Este teste valida a utilização de tipos booleanos, a combinação de operadores lógicos e aritméticos (`and` , `mod` , `div`), e o controle de fluxo complexo através de estruturas `while` e `if` - `then` - `else` aninhadas. Além disso, demonstra a correta precedência de operadores em expressões relacionais e a manipulação de variáveis de diferentes tipos no mesmo bloco de comandos.

Código Pascal:

```pascal
program NumeroPrimo;
var
  num, i: integer;
  primo: boolean;
begin
  writeln('Introduza um número inteiro positivo:');
  readln(num);
  primo := true;
  i := 2;
  while (i <= (num div 2)) and primo do
  begin
    if (num mod i) = 0 then
        primo := false;
    i := i + 1;
  end;
  if primo then
    writeln(num, ' é um número primo')
  else
    writeln(num, ' não é um número primo');
end.
```

Código Máquina Virtual gerado:

```
cmd
START
        PUSHN 2
        PUSHN 1
        PUSHS "Introduza um nÃºmero inteiro positivo:"
        WRITES
        WRITELN
        READ
        ATOI
        STOREG 0
        PUSHI 1
        STOREG 2
        PUSHI 2
        STOREG 1
L1:
        PUSHG 1
        PUSHG 0
        PUSHI 2
        DIV
        INFEQ
        PUSHG 2
        AND
        JZ L2
        PUSHG 0
        PUSHG 1
        MOD
        PUSHI 0
        EQUAL
        JZ L3
        PUSHI 0
        STOREG 2
        JUMP L4
L3:
L4:
        PUSHG 1
        PUSHI 1
        ADD
        STOREG 1
        JUMP L1
L2:
        PUSHG 2
        JZ L5
        PUSHG 0
        WRITEI
        PUSHS " Ã© um nÃºmero primo"
        WRITES
        WRITELN
        JUMP L6
L5:
        PUSHG 0
        WRITEI
        PUSHS " nÃ£o Ã© um nÃºmero primo"
        WRITES
        WRITELN
L6:
STOP
```

5.4 Exemplo 4: Soma de uma lista de inteiros

Este exemplo valida a implementação de tipos estruturados (`array`), o acesso a índices e a manipulação de limites de memória.

Código Pascal:

```
program SomaArray;
var
  numeros: array [1..5] of integer;
  i, soma: integer;
begin
  soma := 0;
  writeln('Introduza 5 números inteiros:');
  for i := 1 to 5 do
  begin
    readln(numeros[i]);
    soma := soma + numeros[i];
  end;
  writeln('A soma dos números é: ', soma);
end.
```

Código Máquina Virtual gerado:

```
    cmd
    START
            PUSHN 1
            PUSHI 5
            ALLOCN
            STOREG 0
            PUSHN 2
            PUSHI 0
            STOREG 2
            PUSHS "Introduza 5 nÃºmeros inteiros:"
            WRITES
            WRITELN
            PUSHI 1
            STOREG 1
    L1:
            PUSHG 1
            PUSHI 5
            INFEQ
            JZ L2
            PUSHG 0
            PUSHG 1
            PUSHI 1
            SUB
            READ
            ATOI
            STOREN
            PUSHG 2
            PUSHG 0
            PUSHG 1
            PUSHI 1
            SUB
            LOADN
            ADD
            STOREG 2
            PUSHG 1
            PUSHI 1
            ADD
            STOREG 1
            JUMP L1
    L2:
            PUSHS "A soma dos nÃºmeros Ã©: "
            WRITES
            PUSHG 2
            WRITEI
            WRITELN
    STOP
```

5.5 Exemplo 5: Conversão binário-decimal

Este teste valida a definição de funções, escopo local, parâmetros, manipulação de strings e funções nativas como `length`.

Código Pascal:

```
program BinarioParaInteiro;
function BinToInt(bin: string): integer;
var
  i, valor, potencia: integer;
begin
  valor := 0;
  potencia := 1;
  for i := length(bin) downto 1 do
  begin
    if bin[i] = '1' then
      valor := valor + potencia;
    potencia := potencia * 2;
  end;
  BinToInt := valor;
end;

var
  bin: string;
  valor: integer;
begin
  writeln('Introduza uma string binária:');
  readln(bin);
  valor := BinToInt(bin);
  writeln('O valor inteiro correspondente é: ', valor);
end.
```

Código Máquina Virtual gerado:

```
    cmd
    START
            PUSHN 1
            PUSHN 1
            PUSHS "Introduza uma string binÃ¡ria:"
            WRITES
            WRITELN
            READ
            STOREG 0
            PUSHI 0
            PUSHG 0
            PUSHA BinToInt
            CALL
            POP 1
            STOREG 1
            PUSHS "O valor inteiro correspondente Ã©: "
            WRITES
            PUSHG 1
            WRITEI
            WRITELN
    STOP
    BinToInt:
            PUSHN 3
            PUSHI 0
            STOREL 2
            PUSHI 1
            STOREL 3
            PUSHL -1
            STRLEN
            STOREL 1
    L1:
            PUSHL 1
```

```
        PUSHI 1
        SUPEQ
        JZ L2
        PUSHL -1
        PUSHL 1
        PUSHI 1
        SUB
        CHARAT
        PUSHI 49
        EQUAL
        JZ L3
        PUSHL 2
        PUSHL 3
        ADD
        STOREL 2
        JUMP L4
L3:
L4:
        PUSHL 3
        PUSHI 2
        MUL
        STOREL 3
        PUSHL 1
        PUSHI 1
        SUB
        STOREL 1
        JUMP L1
L2:
        PUSHL 2
        STOREL -2
        RETURN
```

6. Conclusão

O desenvolvimento deste compilador permitiu consolidar os conceitos fundamentais da teoria de linguagens e compiladores, aplicando-os na construção de uma ferramenta funcional para o Pascal Standard.

6.1 Resumo do Trabalho Desenvolvido

- **Fidelidade ao Enunciado**: O sistema processa corretamente todas as estruturas solicitadas no enunciado: controle de fluxo, operações aritméticas e tipos de dados exigidos, como `integer`, `boolean`, `string` e `array`.
- **Modularidade e Extensibilidade**: A opção por utilizar uma Árvore Sintática Abstrata (AST) como representação intermédia garantiu uma separação clara entre a análise e a geração de código, facilitando a depuração e garantindo a escalabilidade do projeto.
- **Implementação de Opcionais**: Superando os requisitos base, o grupo implementou o suporte a subprogramas (`function` e `procedure`), o que permitiu a execução de programas modulares e complexos, como a conversão de bases numéricas.
- **Gestão de Símbolos**: A implementação de uma Tabela de Símbolos dinâmica com suporte a escopos aninhados garantiu a integridade semântica do código, permitindo o uso seguro de variáveis locais e parâmetros.

6.2 Considerações Finais

Os testes realizados com os exemplos do enunciado demonstraram que o compilador é capaz de gerar código máquina válido e funcional para a VM disponibilizada. Embora o foco tenha sido a correção lógica em detrimento de otimizações agressivas, a base construída é sólida e robusta.

Com a conclusão deste projeto, o grupo entrega uma solução que cumpre integralmente os critérios de avaliação de correção, estrutura e funcionalidade propostos.