



Universidade do Minho

Escola de Engenharia

Licenciatura em Engenharia Informática

Processamento de Linguagens

Ano Letivo de 2024/2025

Grupo 46

Gonçalo Gonçalves (a100833)

Diogo Fernandes (a100746)

Samuel Ferreira (a100654)

1 de junho de 2025

Índice

1	Introdução	1
2	Arquitetura do Sistema	2
2.1	Visão Geral da Arquitetura	2
2.2	Fluxo de Processamento	2
2.3	Estrutura de Dados Principal - AST	2
2.3.1	Nós de Estrutura do Programa	3
2.3.2	Nós de Controle de Fluxo	3
2.3.3	Nós de Operações	3
2.3.4	Nós de Dados e Acesso	4
2.3.5	Nós de I/O	4
2.3.6	Tratamento de Erros	4
2.4	Sistema de Geração de Código	5
2.4.1	Mapeamento de Variáveis	5
2.4.2	Gestão de Variáveis	5
2.4.3	Geração de Labels	5
2.4.4	Instruções Assembly Geradas	5
3	Análise Léxica (LEX)	6
3.1	Fundamentos da Análise Léxica	6
3.2	Implementação do Lexer	6
3.2.1	Palavras Reservadas	6
3.2.2	Definição de Tokens	7
3.2.3	Expressões Regulares	7
3.2.4	Operadores e Atribuição	7
3.3	Tratamento de Comentários	8
3.4	Gestão de Erros Léxicos	8
3.5	Características Avançadas do Lexer	8
3.5.1	Símbolos Literais	8
3.5.2	Controle de Linha	8
3.6	Criação do Lexer	8
4	Análise Sintática (YACC)	10
4.1	Fundamentos da Análise Sintática	10
4.2	Precedência de Operadores	10
4.3	Estrutura da Gramática	10
4.3.1	Programas Pascal	10
4.3.2	Declarações	11
4.3.3	Variáveis	11
4.3.4	Subprogramas	11
4.4	Bloco Principal	11

4.5	Comandos e Estruturas de Controle	12
4.5.1	Comando Composto	12
4.5.2	Atribuições	12
4.5.3	Condicional if	12
4.5.4	Laços for e while	12
4.6	Expressões	13
4.6.1	Expressões Booleanas e Aritméticas	13
4.6.2	Termos e Fatores	13
4.7	Entrada e Saída	13
4.8	Chamadas de Funções e Procedimentos	13
4.9	Tratamento de Erros Sintáticos	14
4.10	Construção da AST	14
4.11	Validações Semânticas Simples	14
5	Interpretação e Execução	15
5.1	Padrão Visitor	15
5.2	Gestão de Variáveis	15
5.3	Avaliação de Expressões	15
6	Conclusão	16
6.1	Pontos Fortes da Implementação	16
6.2	Áreas para Melhoramento	16
6.3	Valor Educacional e Aplicações	17
6.4	Considerações Finais	17

1 Introdução

Este relatório apresenta uma análise técnica detalhada de um interpretador para a linguagem Pascal implementado em Python. O projeto constitui um sistema completo de análise e execução de código Pascal, desenvolvido utilizando as ferramentas `PLY` (Python Lex-Yacc), que fornecem funcionalidades equivalentes aos clássicos `lex` e `yacc` do ambiente Unix.

A implementação segue uma arquitetura modular bem estruturada, dividindo claramente as responsabilidades entre análise léxica, análise sintática e interpretação. O sistema é capaz de processar um subconjunto significativo da linguagem Pascal, incluindo declarações de variáveis, estruturas de controle, operações aritméticas e lógicas, e operações de entrada/saída.

O objetivo principal deste interpretador é demonstrar os conceitos fundamentais de construção de compiladores e interpretadores, proporcionando uma compreensão prática de como transformar código fonte em execução através das fases de tokenização, parsing e avaliação da árvore sintática abstrata (AST).

2 Arquitetura do Sistema

2.1 Visão Geral da Arquitetura

O compilador Pascal segue uma arquitetura em camadas típica de sistemas de processamento de linguagens, composta por quatro módulos principais:

1. **Módulo de Análise Léxica** (`pascal_lex.py`) - Responsável pela tokenização
2. **Módulo de Análise Sintática** (`pascal_parser.py`) - Responsável pelo parsing e construção da AST
3. **Módulo de Nós AST** (`ast_nodes.py`) - Define a estrutura da árvore sintática abstrata e geração de código
4. **Módulo de Geração de Código** (integrado em `ast_nodes.py`) - Transforma a AST em código assembly

2.2 Fluxo de Processamento

O processamento do código Pascal segue um pipeline bem definido:

Código Pascal → Tokenização → Parsing → AST → Geração de Código → Assembly

1. **Tokenização:** O código fonte é decomposto em tokens (palavras-chave, identificadores, operadores, etc.)
2. **Análise Sintática:** Os tokens são organizados em uma estrutura hierárquica (AST) seguindo a gramática Pascal
3. **Mapeamento de Variáveis:** As variáveis são mapeadas para posições de memória global
4. **Geração de Código:** A AST é percorrida e transformada em código assembly através do método `generate_code()`

2.3 Estrutura de Dados Principal - AST

A árvore sintática abstrata é implementada através de uma hierarquia de classes que herdam de `ASTNode`. Cada tipo de construção Pascal tem sua representação correspondente.

É de notar que, diferentemente das demais classes de nós, o nodo `IndexNode` **não herda** de `ASTNode`.

Classes para Acesso por Índice

Existem duas classes distintas para manipulação de índices:

- `IndexNode`: Utilizada para acesso a arrays em operações de leitura e escrita.
- `IndexAccessNode`: Utilizada para acesso de caracteres em strings (com operação `charat`).

2.3.1 Nós de Estrutura do Programa

- `ProgramNode`: Representa o programa principal, gera código com `start` e `stop`
- `CompoundNode`: Agrupa comandos em blocos
- `FunctionDeclNode`: Declarações de funções com parâmetros e variáveis locais
- `ProcedureDeclNode`: Declarações de procedimentos

De modo a garantir um estado limpo para cada compilação, a função `ProgramNode.generate_code()` reinicializa as estruturas globais de mapeamento de variáveis (`var_positions`, `next_var_index`) no início de sua execução. Isto evita que definições de variáveis de uma compilação anterior afetem a compilação atual.

2.3.2 Nós de Controle de Fluxo

- `IfNode`, `WhileNode`, `ForNode`: Estruturas de controle com geração de labels para saltos

O construtor de `BinaryOpNode` verifica explicitamente se seus operandos são instâncias de `ASTNode`, lançando exceções detalhadas caso contrário. Isso permite detectar precocemente problemas de tipo na AST com mensagens como:

Exemplo de erro

```
[ERRO] Left operand não é ASTNode: {left} ({type(left).__name__})
```

2.3.3 Nós de Operações

- `AssignNode`, `BinaryOpNode`, `FuncCallNode`

Funções Embutidas

Apesar do dicionário `funcoes_embutidas` conter apenas três entradas (`length`, `ord`, `chr`), o método `FuncCallNode.generate_code()` dá suporte completo às seguintes funções:

- **Funções numéricas:** `abs`, `sqrt`, `round`, `trunc`, `sqr`, `succ`, `pred`
- **Conversão e manipulação de caracteres:** `ord`, `chr`, `upcase`, `lowercase`
- **Manipulação de strings:** `length`, `pos`, `copy`, `concat`, `insert`, `delete`
- **Conversão de tipos:** `val`, `str`

2.3.4 Nós de Dados e Acesso

- `VarNode`, `IndexNode`, `NumberNode`, `StringNode`, `BooleanNode`

Instrução de Debug Permanente

A função `IndexNode.generate_code()` contém uma instrução de depuração permanente no código:

```
1 print(">> DEBUG IndexNode: self.array =", self.array)
```

2.3.5 Nós de I/O

- `WriteNode`: Realiza escrita na saída padrão. Aplica heurística baseada no prefixo do nome da variável: se começar por `"msg"` ou `"txt"`, assume que o conteúdo é uma string e utiliza `writes`.
- `ReadNode`: Sempre aplica conversão de string para inteiro usando `atoi`. Isso implica que apenas leitura de inteiros está atualmente suportada.

É de notar que, para arrays, o nodo `ReadNode` usa a sequência `add+storen` para armazenar no elemento indexado, enquanto que, para variáveis simples, usa `storeg`.

2.3.6 Tratamento de Erros

- `ErrorNode`: Usado para sinalização e propagação de erros durante a análise sintática.

2.4 Sistema de Geração de Código

2.4.1 Mapeamento de Variáveis

`mapear_variaveis(ast)` é uma função fundamental para atribuir posições de memória a variáveis globais e locais. Ela percorre a AST e associa cada variável a um índice numérico único, armazenado em um dicionário global chamado `var_positions`. Esses índices são posteriormente utilizados durante a geração de código para instruções como `pushg`, `storeg`, etc.

2.4.2 Gestão de Variáveis

- `var_positions`: Dicionário que mapeia nomes de variáveis para índices únicos
- `next_var_index`: Contador global para atribuição de posições

Nas funções e procedimentos, o estado global (`var_positions`, `next_var_index`, `declared_variables`) é salvo antes da geração de código da função e restaurado depois, garantindo isolamento de escopos.

2.4.3 Geração de Labels

- `gen_label()`: Função auxiliar para geração de rótulos únicos

2.4.4 Instruções Assembly Geradas

O código gerado é direcionado a uma máquina virtual baseada em pilha:

- **Stack**: `pushi`, `pushs`, `pushg`, `pop`
- **Armazenamento**: `storeg`
- **Aritmética**: `add`, `sub`, `mul`, `div`, `mod`
- **Relacional**: `equal`, `inf`, `sup`, `infeq`, `supeq`
- **Lógica**: `and`, `or`, `not`
- **Controle**: `jz`, `jump`
- **I/O**: `writeti`, `writes`, `writeln`, `read`, `atoi`
- **Subrotinas**: `pusha`, `call`, `return`

3 Análise Léxica (LEX)

3.1 Fundamentos da Análise Léxica

A análise léxica constitui a primeira fase do processo de compilação/interpretação. No contexto deste projeto, o módulo `pascal_lex.py` implementa um analisador léxico utilizando a biblioteca PLY (Python Lex-Yacc), que fornece funcionalidades equivalentes ao lex tradicional.

O analisador léxico transforma a sequência de caracteres do código fonte em uma sequência de tokens, removendo espaços em branco e comentários, e classificando cada elemento léxico de acordo com sua categoria sintática.

3.2 Implementação do Lexer

3.2.1 Palavras Reservadas

O sistema define um conjunto abrangente de palavras reservadas da linguagem Pascal como tokens individuais:

```
1 def t_PROGRAM(t):      r'program';      return t
2 def t_BEGIN(t):        r'begin';        return t
3 def t_END(t):           r'end';          return t
4 def t_VAR(t):           r'var';          return t
5 def t_IF(t):            r'if';           return t
6 def t_THEN(t):          r'then';         return t
7 def t_ELSE(t):          r'else';         return t
8 def t_FUNCTION(t):      r'function';     return t
9 def t_PROCEDURE(t):     r'procedure';    return t
10 def t_WRITELN(t):       r'writeln';     return t
11 def t_READLN(t):        r'readln';      return t
12 def t_FOR(t):           r'for';          return t
13 def t_TO(t):             r'to';          return t
14 def t_DOWNTO(t):        r'downto';      return t
15 def t_WHILE(t):         r'while';        return t
16 def t_DO(t):            r'do';           return t
17 def t_DIV(t):           r'div';          return t
18 def t_MOD(t):           r'mod';          return t
19 def t_AND(t):           r'and';          return t
20 def t_OR(t):            r'or';           return t
21 def t_NOT(t):           r'not';          return t
22 def t_TRUE(t):          r'true';         return t
23 def t_FALSE(t):         r'false';        return t
24 def t_ARRAY(t):         r'array';        return t
```

```
25 def t_OF(t):          r'of';          return t
```

Figura 3.1: Definição de tokens para palavras reservadas

3.2.2 Definição de Tokens

A lista de tokens inclui identificadores, números, strings, operadores e tipos:

```
1 tokens = [  
2     'PROGRAM', 'BEGIN', 'END', 'VAR', 'IF', 'THEN', 'ELSE', 'FUNCTION',  
3     'PROCEDURE', 'WRITELN', 'READLN', 'FOR', 'TO', 'DOWNT0', 'WHILE', 'DO',  
4     'DIV', 'MOD', 'AND', 'OR', 'NOT', 'TRUE', 'FALSE', 'ARRAY', 'OF',  
5     'TIPOVAR', 'TIPOSTRING',  
6     'ID', 'NUMBER', 'STRING',  
7     'ASSIGN', 'EQ', 'NEQ', 'LE', 'GE', 'LT', 'GT', 'DOTDOT'  
8 ]
```

Figura 3.2: Lista de tokens

3.2.3 Expressões Regulares

Os padrões são definidos por expressões regulares:

```
1 def t_ID(t):  
2     r'[a-zA-Z_][a-zA-Z0-9_]*'  
3     return t  
4  
5 def t_NUMBER(t):  
6     r'\d+'  
7     t.value = int(t.value)  
8     return t  
9  
10 def t_STRING(t):  
11     r'\"([^\\"\\n]|(\\.\\.))*?\"'  
12     t.value = t.value[1:-1]  
13     return t
```

Figura 3.3: Expressões regulares para tokens

3.2.4 Operadores e Atribuição

```
1 t_ASSIGN = r':='  
2 t_EQ     = r'='  
3 t_NEQ    = r'<>'  
4 t_LE     = r'<='  
5 t_GE     = r'>='  
6 t_LT     = r'<'  
7 t_GT     = r'>'  
8 t_DOTDOT = r'\\.\\. '
```

Figura 3.4: Operadores de atribuição e comparação

3.3 Tratamento de Comentários

Dois tipos de comentários são reconhecidos e ignorados pelo lexer:

```
1 def t_COMMENT(t):
2     r'\{[^\}]*\}'
3     pass
4
5 def t_COMMENT_LINE(t):
6     r'//.*'
7     pass
```

Figura 3.5: Comentários

3.4 Gestão de Erros Léxicos

Caracteres não reconhecidos são tratados por:

```
1 def t_error(t):
2     print(f"Carácter ilegal '{t.value[0]}' na linha {t.lineno}")
3     t.lexer.skip(1)
```

Figura 3.6: Tratamento de erros léxicos

3.5 Características Avançadas do Lexer

3.5.1 Símbolos Literais

Os literais são tratados diretamente:

```
1 literals = ['+', '-', '*', '/', ';', ',', ':', '(', ')', '[', ']', '.']
```

3.5.2 Controle de Linha

A função abaixo mantém a contagem de linhas atualizada:

```
1 def t_newline(t):
2     r'\n+'
3     t.lexer.lineno += len(t.value)
```

3.6 Criação do Lexer

Por fim, o objeto lexer é criado com:

```
1 import ply.lex as lex
2 lexer = lex.lex()
```

Figura 3.7: Construção do lexer

O modo `re.IGNORECASE` é ativado para tornar a análise insensível a maiúsculas e minúsculas, refletindo o comportamento da linguagem Pascal.

4 Análise Sintática (YACC)

4.1 Fundamentos da Análise Sintática

A análise sintática é a segunda fase do compilador e verifica se a sequência de tokens gerada pelo analisador léxico segue as regras gramaticais da linguagem Pascal. Nesta etapa, também é construída a árvore sintática abstrata (AST). O analisador sintático é implementado no módulo `pascal_parser.py` usando a técnica LALR(1) através da biblioteca `PLY yacc`.

4.2 Precedência de Operadores

A tabela de precedência resolve ambiguidades na interpretação de expressões:

```
1 precedence = (  
2     ('left', 'OR'),  
3     ('left', 'AND'),  
4     ('nonassoc', 'EQ', 'NEQ', 'LT', 'LE', 'GT', 'GE'),  
5     ('left', '+', '-'),  
6     ('left', '*', '/', 'DIV', 'MOD'),  
7     ('nonassoc', 'LOWER_THAN_ELSE'),  
8     ('nonassoc', 'ELSE'),  
9 )
```

Figura 4.1: Tabela de precedência

4.3 Estrutura da Gramática

4.3.1 Programas Pascal

A definição de programa em Pascal inicia com a palavra-chave `PROGRAM`, seguida pelo identificador, declarações, um bloco principal e um ponto final:

```
1 def p_programa(p):  
2     'programa : PROGRAM ID ";" declaracoes bloco_simples "."'  
3     globais, funcoes = p[4]  
4     ...  
5     p[0] = ProgramNode(p[2], funcoes, p[5])
```

Figura 4.2: Regra gramatical para programas

4.3.2 Declarações

As declarações podem conter variáveis, funções e procedimentos:

```
1 def p_declaracoes(p):
2     '''declaracoes : declaracoes_variaveis declaracoes_subprogramas
   declaracoes_variaveis_opt
3         | declaracoes_subprogramas declaracoes_variaveis_opt
4         | declaracoes_variaveis declaracoes_subprogramas
5         | declaracoes_variaveis
6         | declaracoes_subprogramas
7         | empty'''
```

Figura 4.3: Combinação de declarações

4.3.3 Variáveis

A declaração de variáveis verifica se já foram declaradas e registra os nomes:

```
1 def p_decl(p):
2     'decl : lista_ids ":" tipo ";"'
3     ...
4     p[0] = nomes
```

Figura 4.4: Declaração de variáveis

4.3.4 Subprogramas

Funções e procedimentos são declarados conforme mostrado:

```
1 def p_funcao(p):
2     'funcao : FUNCTION ID "(" parametros ")" ":" tipo ";"
   declaracoes_variaveis_opt bloco_simples ";"'
3
4 def p_procedimento(p):
5     'procedimento : PROCEDURE ID "(" parametros ")" ";"
   declaracoes_variaveis_opt bloco_simples ";"'
```

Figura 4.5: Funções e procedimentos

4.4 Bloco Principal

O bloco principal pode conter apenas comandos ou também declarações locais:

```
1 def p_bloco_simples(p):
2     '''bloco_simples : BEGIN comandos END
3         | BEGIN declaracoes_variaveis comandos END'''
```

Figura 4.6: Bloco simples

4.5 Comandos e Estruturas de Controle

4.5.1 Comando Composto

Agrupa uma sequência de comandos entre BEGIN e END:

```
1 def p_comando_composto(p):  
2     'comando_composto : BEGIN comandos END'
```

Figura 4.7: Comando composto

4.5.2 Atribuições

Verifica a declaração da variável antes de gerar a atribuição:

```
1 def p_atribuicao(p):  
2     'atribuicao : ID ASSIGN expressao'
```

Figura 4.8: Comando de atribuição

4.5.3 Condicional if

O uso de precedência resolve ambiguidades como o “dangling else”:

```
1 def p_comando_if(p):  
2     '''comando_if : IF expressao THEN comando ELSE comando  
3     | IF expressao THEN comando %prec LOWER_THAN_ELSE'''
```

Figura 4.9: Comando if-then-else

4.5.4 Laços for e while

Ambas as formas de for são suportadas:

```
1 def p_comando_for(p):  
2     '''comando_for : FOR ID ASSIGN expressao TO expressao DO comando  
3     | FOR ID ASSIGN expressao DOWNTO expressao DO comando'''
```

Figura 4.10: Comando for

O laço while também é reconhecido:

```
1 def p_comando_while(p):  
2     'comando_while : WHILE expressao DO comando'
```

Figura 4.11: Comando while

4.6 Expressões

4.6.1 Expressões Booleanas e Aritméticas

A estrutura hierárquica garante a precedência correta:

```
1 def p_expr_bool(p):
2     '''expressao : expressao_relacional
3                   | expressao_relacional EQ expressao_relacional
4                   | expressao_relacional NEQ expressao_relacional
5                   | ...'''
```

Figura 4.12: Expressões binárias

4.6.2 Termos e Fatores

Os termos suportam operadores multiplicativos, lógicos e aritméticos:

```
1 def p_termo(p):
2     '''termo : fator
3              | termo '*' fator
4              | termo '/' fator
5              | termo DIV fator
6              | termo MOD fator
7              | termo AND fator'''
```

Figura 4.13: Operadores binários

4.7 Entrada e Saída

A linguagem reconhece comandos de entrada/saída como READLN e WRITELN:

```
1 def p_comando_write(p):
2     'comando_write : WRITELN "(" lista_exp ")"'
3
4 def p_comando_read(p):
5     'comando_read : READLN "(" lista_ids ")"'
```

Figura 4.14: Entrada e saída

4.8 Chamadas de Funções e Procedimentos

As chamadas são verificadas quanto à existência e número de argumentos:

```
1 def p_chamada_func_proc(p):
2     'chamada_func_proc : ID "(" argumentos ")"'
```

Figura 4.15: Chamada de função ou procedimento

4.9 Tratamento de Erros Sintáticos

O parser recupera erros reportando tokens inesperados:

```
1 def p_error(p):
2     if p:
3         print(f"Erro de sintaxe na linha {p.lineno}: token inesperado '{p.value}'")
4         parser.errok()
5     else:
6         print("Erro de sintaxe no fim do input")
```

Figura 4.16: Função de erro

4.10 Construção da AST

Cada regra constrói nós da árvore sintática:

```
1 def p_atribuicao(p):
2     'atribuicao : ID ASSIGN expressao'
3     ...
4     p[0] = AssignNode(var_node, expr_node)
```

Figura 4.17: Construção de AST

4.11 Validações Semânticas Simples

Além da análise sintática, são realizados testes semânticos básicos, como:

- Verificação de variáveis e funções já declaradas
- Verificação do número de argumentos em chamadas de subprogramas
- Atribuições inválidas

5 Interpretação e Execução

5.1 Padrão Visitor

O interpretador utiliza o padrão Visitor para percorrer e executar a AST. Cada tipo de nó tem um método de visita correspondente:

```
1 def run(self, node):
2     method_name = 'visit_' + type(node).__name__
3     visitor = getattr(self, method_name, self.generic_visit)
4     return visitor(node)
```

Figura 5.1: Método principal de execução

5.2 Gestão de Variáveis

O interpretador mantém um dicionário simples para armazenar variáveis:

```
1 def visit_AssignNode(self, node):
2     value = self.run(node.expr)
3     self.vars[node.var.name] = value
```

Figura 5.2: Execução de atribuições

5.3 Avaliação de Expressões

A avaliação de operações binárias suporta operadores aritméticos, lógicos e relacionais:

```
1 def visit_BinaryOpNode(self, node):
2     left = self.run(node.left)
3     right = self.run(node.right) if node.right is not None else None
4     op = node.op
5     if op == '+': return left + right
6     if op == '-': return left - right
7     # ... outros operadores
```

Figura 5.3: Avaliação de operadores binários

6 Conclusão

O interpretador Pascal apresentado demonstra uma implementação sólida e bem estruturada dos conceitos fundamentais de construção de linguagens de programação. A utilização das ferramentas PLY (Python Lex-Yacc) proporciona uma base robusta para a implementação das fases de análise léxica e sintática, seguindo as melhores práticas estabelecidas pelas ferramentas clássicas `lex` e `yacc`.

6.1 Pontos Fortes da Implementação

A arquitetura modular adotada facilita a manutenção e extensão do sistema. A separação clara entre análise léxica, sintática e interpretação permite modificações independentes em cada componente. O uso do padrão Visitor para a interpretação proporciona flexibilidade para futuras extensões, como geração de código ou otimizações.

A implementação da análise léxica demonstra compreensão sólida dos conceitos de tokenização, incluindo tratamento adequado de palavras reservadas, comentários e diferentes tipos de literais. O suporte a expressões regulares complexas para reconhecimento de strings e a gestão apropriada de números de linha para reportagem de erros evidenciam atenção aos detalhes práticos.

Na análise sintática, a gramática implementada cobre um subconjunto significativo da linguagem Pascal, incluindo estruturas de controle essenciais, declarações de variáveis e operações. O tratamento da ambiguidade "dangling else" através de precedências demonstra compreensão de questões sintáticas complexas.

6.2 Áreas para Melhoramento

Embora funcional, o interpretador apresenta algumas limitações que poderiam ser endereçadas em futuras iterações. A gestão de escopo é simplificada, utilizando apenas um dicionário global para variáveis. Uma implementação mais completa incluiria pilhas de escopo para suportar adequadamente subprogramas e variáveis locais.

O tratamento de tipos é básico, sem verificação estática de tipos ou conversões automáticas. A implementação de um sistema de tipos mais robusto melhoraria significativamente a qualidade do interpretador. Adicionalmente, o suporte a arrays está definido na gramática mas não completamente implementado no interpretador.

6.3 Valor Educacional e Aplicações

Este projeto serve como excelente exemplo educacional para compreensão dos princípios de construção de compiladores e interpretadores. A progressão lógica desde a tokenização até a execução ilustra claramente como código fonte é transformado em computação executável.

O uso de Python como linguagem de implementação torna o código acessível para estudantes, enquanto as ferramentas PLY proporcionam uma ponte natural para conceitos mais avançados encontrados em ferramentas profissionais de desenvolvimento de compiladores.

6.4 Considerações Finais

A implementação apresentada constitui uma base sólida para um interpretador Pascal completo. Com as extensões apropriadas — incluindo melhor gestão de escopo, sistema de tipos robusto e implementação completa de subprogramas — este projeto poderia evoluir para um interpretador Pascal totalmente funcional.

O trabalho demonstra que é possível criar sistemas de processamento de linguagens eficazes utilizando ferramentas modernas, mantendo os princípios teóricos estabelecidos pela literatura clássica da área. Esta abordagem híbrida — teoria sólida implementada com ferramentas contemporâneas — representa uma metodologia valiosa para o ensino e desenvolvimento de sistemas de software na área de linguagens de programação.