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)

30 de maio de 2025

Índice

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 ferramenias 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:

$$\texttt{Código Pascal} \rightarrow \texttt{Tokenização} \rightarrow \texttt{Parsing} \rightarrow \texttt{AST} \rightarrow \texttt{Geração de Código} \rightarrow \texttt{Assembly}$$

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:

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

2.3.2 Nós de Controle de Fluxo

- `IfNode`: Estruturas condicionais com geração de labels para saltos
- `WhileNode`: Ciclos com labels de início e fim
- `ForNode`: Ciclos for/downto com controle automático da variável de iteração

2.3.3 Nós de Operações

- `AssignNode`: Operações de atribuição com validação de tipos
- `BinaryOpNode`: Operações binárias (aritméticas, relacionais, lógicas)
- `FuncCallNode`: Chamadas de função/procedimento

2.3.4 Nós de Dados e Acesso

- `VarNode`: Variáveis com mapeamento para posições globais
- `IndexNode`: Acesso a arrays (simulado)
- `NumberNode`, `StringNode`, `BooleanNode`: Nós terminais com valores literais

2.3.5 Nós de I/O

- `WriteNode`: Operações de escrita com distinção automática entre strings e números
- `ReadNode`: Operações de leitura com conversão automática string-para-inteiro

2.3.6 Tratamento de Erros

- `ErrorNode`: Nó especial para propagação de erros durante a compilação

2.4 Sistema de Geração de Código

2.4.1 Gestão de Variáveis

O sistema implementa um mapeador global de variáveis:

- `var_positions`: Dicionário que mapeia nomes de variáveis para índices únicos
- `next_var_index`: Contador global para atribuição de posições
- `mapear_variaveis()`: Função que percorre a AST e atribui posições às variáveis

2.4.2 Geração de Labels

Sistema automático de geração de labels únicos para controle de fluxo:

- `gen_label()`: Função que gera labels com prefixos personalizáveis
- Utilizado em estruturas condicionais e ciclos para implementar saltos

2.4.3 Instruções Assembly Geradas

O compilador gera código para uma máquina virtual stack-based:

- **Manipulação de Stack**: `pushi`, `pushs`, `pushg`
- **Armazenamento**: `storeg` para variáveis globais
- **Operações Aritméticas**: `add`, `sub`, `mul`, `div`, `mod`
- **Operações Relacionais**: `equal`, `inf`, `sup`, `infeq`, `supeq`
- **Operações Lógicas**: `and`, `or`, `not`
- **Controle de Fluxo**: `jz`, `jump`
- **I/O**: `writei`, `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:

```
def t_PROGRAM(t):     r'program';     return t
def t_BEGIN(t):       r'begin';       return t
def t_END(t):         r'end';         return t
def t_VAR(t):         r'var';         return t
def t_IF(t):          r'if';          return t
def t_THEN(t):        r'then';        return t
def t_ELSE(t):        r'else';        return t
def t_FUNCTION(t):    r'function';    return t
def t_PROCEDURE(t):   r'procedure';   return t
def t_WRITELN(t):     r'writeln';     return t
def t_READLN(t):      r'readln';      return t
def t_FOR(t):         r'for';         return t
def t_TO(t):          r'to';          return t
def t_DOWNTO(t):      r'downto';      return t
def t_WHILE(t):       r'while';       return t
def t_DO(t):          r'do';          return t
def t_DIV(t):         r'div';         return t
def t_MOD(t):         r'mod';         return t
def t_AND(t):         r'and';         return t
def t_OR(t):          r'or';          return t
def t_NOT(t):         r'not';         return t
def t_TRUE(t):        r'true';        return t
def t_FALSE(t):       r'false';       return t
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:

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

Figura 3.2: Lista de tokens

3.2.3 Expressões Regulares

Os padrões são definidos por expressões regulares:

```
def t_ID(t):
    r'[a-zA-Z_][a-zA-Z0-9_]*'
    return t

def t_NUMBER(t):
    r'\d+'
    t.value = int(t.value)
    return t

def t_STRING(t):
    r'\'([^\\\n]|(\\.))*?\''
    t.value = t.value[1:-1]
    return t
```

Figura 3.3: Expressões regulares para tokens

3.2.4 Operadores e Atribuição

```
t_ASSIGN = r':='
t_EQ     = r'='
t_NEQ    = r'<>'
t_LE     = r'<='
t_GE     = r'>='
t_LT     = r'<'
t_GT     = r'>'
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:

```
def t_COMMENT(t):
    r'\{[^}]*\}'
    pass

def t_COMMENT_LINE(t):
    r'//.*'
    pass
```

Figura 3.5: Comentários

3.4 Gestão de Erros Léxicos

Caracteres não reconhecidos são tratados por:

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

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

3.5.2 Controle de Linha

A função abaixo mantém a contagem de linhas atualizada:

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

3.6 Criação do Lexer

Por fim, o objeto lexer é criado com:

```
import ply.lex as lex
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:

```
precedence = (
    ('left', 'OR'),
    ('left', 'AND'),
    ('nonassoc', 'EQ', 'NEQ', 'LT', 'LE', 'GT', 'GE'),
    ('left', '+', '-'),
    ('left', '*', '/', 'DIV', 'MOD'),
    ('nonassoc', 'LOWER_THAN_ELSE'),
    ('nonassoc', 'ELSE'),
)
```

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:

```
def p_programa(p):
    'programa : PROGRAM ID ";" declaracoes bloco_simples "."'
    globais, funcoes = p[4]
    ...
    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:

```
def p_declaracoes(p):
    '''declaracoes : declaracoes_variaveis declaracoes_subprogramas
    declaracoes_variaveis_opt
                   | declaracoes_subprogramas declaracoes_variaveis_opt
                   | declaracoes_variaveis declaracoes_subprogramas
                   | declaracoes_variaveis
                   | declaracoes_subprogramas
                   | empty'''
```

Figura 4.3: Combinação de declarações

4.3.3 Variáveis

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

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

Figura 4.4: Declaração de variáveis

4.3.4 Subprogramas

Funções e procedimentos são declarados conforme mostrado:

```
def p_funcao(p):
    'funcao : FUNCTION ID "(" parametros ")" ":" tipo ";"
    declaracoes_variaveis_opt bloco_simples ";"'

def p_procedimento(p):
    '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:

```
def p_bloco_simples(p):
    '''bloco_simples : BEGIN comandos END
                     | 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`:

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

Figura 4.7: Comando composto

4.5.2 Atribuições

Verifica a declaração da variável antes de gerar a atribuição:

```
def p_atribuicao(p):
    '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":

```
def p_comando_if(p):
    '''comando_if : IF expressao THEN comando ELSE comando
                  | 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:

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

Figura 4.10: Comando for

O laço `while` também é reconhecido:

```
def p_comando_while(p):
    '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:

```
def p_expr_bool(p):
    '''expressao : expressao_relacional
                 | expressao_relacional EQ expressao_relacional
                 | expressao_relacional NEQ expressao_relacional
                 | ...'''
```

Figura 4.12: Expressões binárias

4.6.2 Termos e Fatores

Os termos suportam operadores multiplicativos, lógicos e aritméticos:

```
def p_termo(p):
    '''termo : fator
             | termo '*' fator
             | termo '/' fator
             | termo DIV fator
             | termo MOD fator
             | termo AND fator'''
```

Figura 4.13: Operadores binários

4.7 Entrada e Saída

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

```
def p_comando_write(p):
    'comando_write : WRITELN "(" lista_exp ")"'

def p_comando_read(p):
    '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:

```
def p_chamada_func_proc(p):
    '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:

```
def p_error(p):
    if p:
        print(f"Erro de sintaxe na linha {p.lineno}: token inesperado '{p.value}'")
        parser.errok()
    else:
        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:

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

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

Figura 5.1: Método principal de execução

5.2 Gestão de Variáveis

O interpretador mantém um dicionário simples para armazenar variáveis:

```
def visit_AssignNode(self, node):
    value = self.run(node.expr)
    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:

```
def visit_BinaryOpNode(self, node):
    left = self.run(node.left)
    right = self.run(node.right) if node.right is not None else None
    op = node.op
    if op == '+': return left + right
    if op == '-': return left - right
    # ... 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.