Relatório de Implementação: Compilador para Linguagem Pascal-like

1. Introdução

Este relatório apresenta a implementação de um compilador completo para uma linguagem de programação inspirada em Pascal. O projeto foi desenvolvido em Python e inclui todas as fases clássicas de compilação: análise léxica, análise sintática, análise semântica e interpretação do código.

2. Gramática da Linguagem

A linguagem implementada segue a seguinte gramática formal em notação BNF:

```
<prog>::=programa id; [<declarações>] <bloco> .
<declarações>::=var <lista_declaracao_var> {<lista_declaracao_var>}
<lista_declaração_var>::= id {,id} : <tipo>;
<bloco>::= início <lista comandos> fim
<tipo>::=inteiro | lógico
<lista comandos>::=<comando>; {<comando>;}
<comando>::=<atribuição> | <leitura> | <escrita> | <bloco> | <condicional> | <repetição>
<atribuição>::=id := <expr>
<leitura>::=ler (id {,id})
<escrita>::=escrever (<stringvar> {,<stringvar>})
<condicional>::=se <exprLogico> então <comando> [senão <comando>]
<repetição>::=enquanto <exprLogico> faça <comando>
<expr> ::= <termo> <expr2>
<expr2> ::= + <termo> <expr2> | - <termo> <expr2> | ε
<termo> ::= <fator> <termo2>
<termo2> ::= * <fator> <termo2> | / <fator> <termo2> | ε
<fator> ::= (<expr>) | - <fator> | id | num
<exprLogico>::= <expr> <opLogico> <expr> | id
<opLogico> ::= < | <= | > | >= | = | <>
<stringvar>::=str | <expr>
```

2.1 Características da Linguagem

- **Tipos de dados**: `inteiro` e `lógico`
- **Estruturas de controle**: condicionais (`se-então-senão`) e loops (`enquanto-faça`)
- **Operações aritméticas**: `+`, `-`, `*`, `/`
- **Operações lógicas**: `<`, `<=`, `>`, `>=`, `=`, `<>`
- **Entrada/Saída**: comandos `ler` e `escrever`
- **Comentários**: suporte a comentários multi-linha `/* ... */`

3. Instalação e Configuração

3.1 Pré-requisitos

- **Python 3.8+**: O projeto foi desenvolvido e testado com Python 3.12
- **Sistema Operacional**: Windows, Linux ou macOS

3.2 Passos para Instalação

1. **Clone ou baixe o projeto**:

```
git clone <repositório>
cd Compiladores
```

2. **Verificar instalação do Python**:

```
python --version
```

3. **Executar testes**:

```
python main.py
```

3.3 Estrutura do Projeto

```
Compiladores/
├── abstract_syntax_tree.py     # Definição da AST
├── analisador_lexico.py        # Análise léxica
├── analisador_sintatico.py     # Análise sintática
├── analisador_semantico.py     # Análise semântica
├── interpretador.py            # Interpretador
├── tabela_de_simbolos.py       # Tabela de símbolos
├── read_code_file.py           # Utilitário para leitura
├── main.py                     # Arquivo principal de testes
├── gramatica.txt              # Gramática formal
├── codigos/                   # Códigos de teste válidos
└── codigos_errados/           # Códigos com erros intencionais
```

4. Implementação Detalhada

4.1 Analisador Léxico (`analisador_lexico.py`)

Abordagem Implementada: Autômato Finito implementado com Expressões Regulares

Características:

- Utiliza a biblioteca `re` do Python para reconhecimento de padrões
- Implementa um sistema de tokens baseado em tuplas `(tipo, padrão_regex)`
- Gerencia posição, linha e coluna para relatório de erros
- Suporte a palavras reservadas através de dicionário

Tokens Reconhecidos:

```
self.especificacoes_tokens = [
    ("COMENTÁRIO", r"/\*.*?\*/"),
    ("STRING", r'"(?:\\.|[^"\\])*"'),
    ("ATRIBUIÇÃO", r":="),
    ("MENOR_IGUAL", r"<="),
    ("MAIOR_IGUAL", r">="),
    ("DIFERENTE", r"<>"),
    ("NÚMERO", r"\d+"),
    ("ID", r"[a-zA-Z_À-ú][a-zA-Z0-9_À-ú]*"),
    # ... outros tokens
]
```

Vantagens da Abordagem:

- Eficiência através de regex compiladas

- Facilidade de manutenção e extensão
- Tratamento automático de espaços e comentários

4.2 Analisador Sintático (`analisador_sintatico.py`)

Abordagem Implementada: Parser Recursivo Descendente (Top-Down)

Características:

- Cada regra da gramática corresponde a um método
- Implementa a técnica de Recursive Descent Parsing
- Constrói a Árvore Sintática Abstrata (AST) durante a análise
- Tratamento de erro com informações precisas de localização

Exemplo de Implementação:

```
def _programa(self) -> ProgramaNode:
    """<prog>::=programa id; [<declarações>] <bloco> ."""
    self._consumir("PROGRAMA")
    nome_programa = self._consumir("ID").valor
    self._consumir("PONTO_VÍRGULA")

    declaracoes = self._declaracoes()
    bloco = self._bloco()

    self._consumir("PONTO")
    return ProgramaNode(nome_programa, declaracoes, bloco)
```

Vantagens da Abordagem:

- Implementação intuitiva e legível
- Facilidade de debug e manutenção
- Construção natural da AST
- Tratamento direto de precedência de operadores

4.3 Árvore Sintática Abstrata (`abstract_syntax_tree.py`)

Abordagem Implementada: Padrão Composite com nós tipados

Estrutura Hierárquica:

```
class ASTNode:                    # Nó base abstrato
├── ProgramaNode                 # Raiz do programa
├── DeclaracoesNode              # Container de declarações
├── BlocoNode                    # Bloco de comandos
├── ComandoNode                  # Classe base para comandos
│   ├── AtribuicaoNode
│   ├── SeNode
│   ├── EnquantoNode
│   ├── EscreverNode
│   └── LerNode
├── ExprNode                     # Expressões aritméticas
├── ExprLogicoNode               # Expressões lógicas
└── TerminalNodes                # NumeroNode, VariavelNode, etc.
```

Características:

- Type hints completos para type safety
- Métodos `__str__` para debug e visualização
- Estrutura modular e extensível

4.4 Analisador Semântico (`analisador_semantico.py`)

Abordagem Implementada: Visitor Pattern com Tabela de Símbolos

Funcionalidades Implementadas:

- **Verificação de Declaração**: Todas as variáveis devem ser declaradas antes do uso
- **Verificação de Tipos**: Compatibilidade entre tipos em atribuições e operações
- **Verificação de Contexto**: Validação semântica específica para cada construção

Exemplo de Verificação de Tipos:

```
def visitar_AtribuicaoNode(self, no: AtribuicaoNode) -> None:
    tipo_var = self.visitar(no.esquerda)
    tipo_expr = self.visitar(no.direita)

    if not self.compatibilidade_tipos(tipo_var, tipo_expr):
        raise SemanticError(
            f"Incompatibilidade de tipos. Não é possível atribuir "
            f"'{tipo_expr}' para a variável do tipo '{tipo_var}'.",
            token=no.esquerda.token
        )
```

Tabela de Símbolos:

- Implementada como dicionário hash para acesso O(1)
- Armazena tipo, nome e localização de cada símbolo
- Suporte a escopo (preparado para futuras extensões)

4.5 Interpretador (`interpretador.py`)

Abordagem Implementada: Tree-Walking Interpreter com Padrão Visitor

Características:

- Execução direta sobre a AST sem código intermediário
- Ambiente de execução com dicionários para variáveis e tipos
- Sistema de dispatch dinâmico para diferentes tipos de nós

Exemplo de Interpretação:

```
def interpretar_AtribuicaoNode(self, no):
    nome_var = no.esquerda.valor
    valor = self.interpretar(no.direita)
    self.variaveis[nome_var] = valor
```

Tratamento de Tipos:

- `inteiro`: Valores numéricos inteiros
- `lógico`: Valores booleanos (True/False)
- Conversões automáticas para saída de strings

4.6 Sistema de Tratamento de Erros

Implementação Robusta:

- Erros léxicos: caracteres não reconhecidos
- Erros sintáticos: estruturas malformadas
- Erros semânticos: problemas de tipo e escopo
- Erros de execução: divisão por zero, variáveis não inicializadas

Informações de Debug:

- Linha e coluna exatas do erro
- Mensagens descritivas
- Stack trace completo para desenvolvimento

5. Exemplos de Uso

5.1 Programa Exemplo

```
programa testeMedia;
var
    cont, x, Media: inteiro;

início
    cont := 1;
    Media := 0;

    enquanto cont <= 5 faça
    início
        escrever("Digite o valor: ");
        ler(x);
        Media := Media + x;
        cont := cont + 1;
    fim;

    escrever("Media final: ", Media / 5);
fim.
```

5.2 Execução

```
python interpretador.py codigos/codigo1.txt
```

6. Testes e Validação

6.1 Cobertura de Testes

- **Códigos válidos**: 5 programas testando diferentes funcionalidades
- **Códigos com erros**: 5 programas testando diferentes tipos de erro
- **Testes automatizados**: Script `main.py` para execução em lote

6.2 Casos de Teste

- Programas com loops e condicionais
- Operações aritméticas e lógicas
- Entrada e saída de dados
- Declaração e uso de variáveis
- Tratamento de erros diversos