



**UNIVERSIDADE ESTADUAL DE MARINGÁ – UEM**  
**CENTRO DE TECNOLOGIA – CTC**  
**DEPARTAMENTO DE INFORMÁTICA – DIN**

**ALUNO:**

**THIAGO HENRIQUE CALVI – RA 134955**

**PROFESSORA:**

**DRA. VALÉRIA DELISANDRA FELTRIM**

**COMPILADOR RASCAL**

**Maringá-PR**

**2025**

# Sumário

	Páginas
<b>1 Introdução</b>	<b>3</b>
<b>2 Decisões de Projeto e Implementação</b>	<b>3</b>
2.1 Linguagem e Ferramentas . . . . .	3
2.2 AST . . . . .	3
2.3 Gerenciamento de Memória com Smart Pointers . . . . .	3
2.4 Padrão de Projeto Visitor . . . . .	3
2.5 Tabela de Símbolos e Escopo . . . . .	4
<b>3 Visão Geral dos Módulos</b>	<b>4</b>
3.1 Análise Léxica (Scanner) . . . . .	4
3.2 Análise Sintática (Parser) . . . . .	5
3.3 Análise Semântica . . . . .	5
3.4 Estrutura da AST . . . . .	5
<b>4 Passo a Passo para Compilar e Executar</b>	<b>5</b>
4.1 Compilação do Compilador . . . . .	6
4.2 Execução do Compilador . . . . .	6
4.3 Execução do Código Gerado (MEPA) . . . . .	6
4.4 Limpeza . . . . .	6

# 1 Introdução

O objetivo deste trabalho foi desenvolver um compilador completo para a linguagem Rascal, uma linguagem imperativa com suporte a procedimentos, funções e tipos básicos (inteiro e booleano). O compilador traduz o código fonte Rascal para instruções da máquina virtual MEPA (Máquina de Execução de Pascal).

A implementação foi realizada na linguagem C++, utilizando as ferramentas Flex para a geração do analisador léxico e Bison para o analisador sintático.

## 2 Decisões de Projeto e Implementação

Esta seção detalha as principais decisões tomadas durante o desenvolvimento do compilador, abrangendo a escolha das ferramentas, a arquitetura e as estruturas de dados fundamentais.

### 2.1 Linguagem e Ferramentas

O compilador foi implementado utilizando a linguagem C++ (padrão C++17), escolhida por sua eficiência e pelo suporte a programação orientada a objetos e programação genérica através da *Standard Template Library* (STL). Para as fases de análise léxica e sintática, optou-se pelo uso das ferramentas **Flex** e **Bison**, respectivamente.

### 2.2 AST

As classes da AST foram projetadas utilizando uma hierarquia polimórfica, com uma classe base abstrata `ASTNode` e subclasses especializadas para expressões (`Expr`), comandos (`Stmt`) e declarações (`Decl`).

### 2.3 Gerenciamento de Memória com Smart Pointers

Para mitigar problemas comuns de gerenciamento de memória em C++, como vazamentos (*memory leaks*) e acesso a ponteiros inválidos, adotou-se o uso de **Smart Pointers**, especificamente `std::shared_ptr`. Todos os nós da AST são manipulados através de ponteiros inteligentes. Isso garante que a memória alocada para os nós seja liberada automaticamente quando eles não forem mais referenciados, simplificando o destrutor das classes e a lógica de limpeza da árvore.

### 2.4 Padrão de Projeto Visitor

Para realizar as operações sobre a AST (impressão, análise semântica e geração de código), foi adotado o padrão de projeto **Visitor**.

- **Justificativa:** O padrão Visitor permite adicionar novas operações sobre a estrutura da AST sem modificar as classes dos nós. Isso respeita o Princípio Aberto/Fechado (Open/Closed Principle), pois a lógica das operações fica encapsulada em classes visitantes separadas (ASTPrinter, SemanticChecker, CodeGenerator).
- **Implementação:** A classe base ASTNode define um método virtual puro `accept(Visitor& v)`, que é implementado por cada subclasse para despachar a chamada para o método `visit` correspondente no visitante.

## 2.5 Tabela de Símbolos e Escopo

A tabela de símbolos foi implementada para suportar escopo estático aninhado.

- **Estrutura:** Utilizou-se uma pilha de escopos, representada por um `std::deque` de mapas (`std::map<std::string, Symbol>`).
- **Funcionamento:** Cada vez que um novo bloco ou sub-rotina é iniciado, um novo mapa vazio é empilhado. A busca por um identificador ocorre do topo da pilha (escopo mais interno) para a base (escopo global).
- **Justificativa:** Essa estrutura modela naturalmente o comportamento de empilhamento de escopos da linguagem, permitindo o sombreamento (*shadowing*) correto de variáveis e a limpeza eficiente de símbolos ao sair de um escopo.

## 3 Visão Geral dos Módulos

O código fonte do compilador está organizado de forma modular, com diretórios e arquivos específicos para cada fase da compilação e para as estruturas de dados compartilhadas.

### 3.1 Análise Léxica (Scanner)

O módulo de análise léxica está contido no arquivo `scanner.l`. Este arquivo é processado pela ferramenta Flex para gerar o código C responsável por tokenizar a entrada.

- **Definição de Tokens:** Utiliza expressões regulares para identificar palavras-chave (`program`, `var`, etc.), operadores, identificadores e literais.
- **Controle de Posição:** Mantém contadores globais (`line_num`, `col_num`) para rastrear a localização exata de cada token, essencial para mensagens de erro precisas.
- **Integração:** Comunica-se com o parser retornando códigos de tokens (definidos no `parser.tab.hpp`) e valores semânticos (via `yylval`).

### 3.2 Análise Sintática (Parser)

O núcleo do compilador reside no arquivo `parser.y`, processado pelo Bison.

- **Gramática:** Define as regras de produção da linguagem Rascal em formato BNF.
- **Construção da AST:** As ações semânticas associadas a cada regra de produção instanciam nós da AST (ex: `new IfStmt(...)`), construindo a árvore de baixo para cima (*bottom-up*).
- **Tratamento de Erros:** Implementa a função `yyerror` para reportar erros sintáticos.

### 3.3 Análise Semântica

Os módulos de análise semântica estão localizados no diretório `semantic/`.

- **SemanticChecker** (`SemanticChecker.hpp/cpp`): Uma classe que implementa a interface `ASTVisitor`. Ela percorre a AST completa validando regras que não podem ser capturadas pela gramática livre de contexto, como verificação de tipos em expressões e correspondência de parâmetros em chamadas de função.
- **SymbolTable** (`SymbolTable.hpp`): Gerencia os identificadores e seus atributos (tipo, categoria, escopo). É utilizada pelo `SemanticChecker` para resolver nomes de variáveis e funções.

### 3.4 Estrutura da AST

As definições das classes que compõem a Árvore Sintática Abstrata encontram-se no diretório `ast/`.

- **ASTNode.hpp:** Define a classe base abstrata para todos os nós.
- **ASTNodes.hpp:** Contém as definições das classes concretas (ex: `BinaryExpr`, `WhileStmt`, `VarDecl`).
- **ASTVisitor.hpp:** Define a interface abstrata para o padrão `Visitor`, garantindo que todos os visitantes implementem métodos para tratar cada tipo de nó da árvore.

## 4 Passo a Passo para Compilar e Executar

Para compilar o código fonte do compilador e utilizá-lo para processar programas Rascal, siga os passos abaixo. Assume-se um ambiente Linux com as ferramentas `g++`, `make`, `flex`, `bison` e `python3` instaladas.

## 4.1 Compilação do Compilador

No diretório raiz do projeto, execute o comando `make`. Este comando irá:

1. Processar o arquivo `parser.y` com o Bison para gerar `parser.tab.cpp` e `parser.tab.hpp`.
2. Processar o arquivo `scanner.l` com o Flex para gerar `lex.yy.c`.
3. Compilar todos os arquivos fonte C++ e ligá-los para gerar o executável `compilador`.

```
1 $ make
```

## 4.2 Execução do Compilador

Para compilar um programa escrito em Rascal, utilize o executável gerado passando o arquivo de entrada e o nome do arquivo de saída desejado (código MEPA):

```
1 $ ./compilador <arquivo_entrada.ras> <arquivo_saida.mepa>
```

Exemplo:

```
1 $ ./compilador testes/exemplo1.ras saida.mepa
```

## 4.3 Execução do Código Gerado (MEPA)

O código MEPA gerado pode ser executado utilizando o interpretador fornecido em Python:

```
1 $ python3 mepa/mepa_pt.py --progfile <arquivo_saida.mepa>
```

## 4.4 Limpeza

Para remover os arquivos gerados durante a compilação (executável e arquivos intermediários), execute:

```
1 $ make clean
```