

# Compilador C Simplificado com Geração de LLVM IR para Assembly RISC-V

Alunos:

Felipe Costa Santos, Hyon Bok Lucas Galvão Mota

UTFPR – CPGEI – 2025

Professores: J. A. Fabro & J. M. Simão

## Resumo

Este artigo apresenta o desenvolvimento de um compilador didático para um subconjunto da linguagem C, com geração de código intermediário em LLVM IR e posterior tradução para Assembly RISC-V. O projeto implementa todas as fases clássicas de um compilador moderno: análise léxica, análise sintática, análise semântica, geração de IR e geração de código final. O compilador foi projetado com uma estrutura modular e extensível, permitindo experimentações futuras e servindo como ferramenta educacional em disciplinas como compiladores, arquitetura de computadores e sistemas embarcados.

**Palavras-chave:** compiladores, LLVM IR, Flex, Bison, RISC-V, análise semântica, geração de código, C simplificado

## 1. Introdução

A crescente adoção da arquitetura RISC-V e a necessidade de ferramentas didáticas motivaram o desenvolvimento de um compilador para um subconjunto da linguagem C, focado na geração de LLVM IR e posterior conversão para Assembly RISC-V. O projeto tem como principal objetivo proporcionar uma base sólida para o entendimento do processo de compilação moderno, desde a análise léxica até a geração de código executável para arquiteturas reais.

## 2. Estrutura do Projeto

O compilador é organizado de forma modular:

- `compiler/`: arquivos do analisador léxico (`lexer.l`) e sintático (`parser.y`);
- `src/`: implementação da geração de IR, gerenciamento de escopos, tipos e símbolos;
- `include/`: cabeçalhos com definições de estruturas e funções auxiliares;

- Exemplos/: programas de teste em C simplificado;
- Makefile: automatiza todo o processo de compilação e testes.

Essa estrutura permite uma separação clara de responsabilidades, facilitando a manutenção e extensibilidade.

### 3. Árvore BNF

A árvore BNF(Backus-Naur Form) geradora é definida da seguinte forma:

```
C/C++
<program_globals> ::= /* vazio */
                    | <program_global> <program_global_list>

<program_global_list> ::= /* vazio */
                       | <program_global> <program_global_list>

<program_global> ::= <function>
                   | <declaration_global>
                   | error

<function> ::= <int_function>
              | <float_function>
              | <char_function>
              | <bool_function>
              | <void_function>

<int_function> ::= "int" ID "(" <parameters> ")" "{" <program_locals> "}"

<float_function> ::= "float" ID "(" <parameters> ")" "{" <program_locals>
"}"

<char_function> ::= "char" ID "(" <parameters> ")" "{" <program_locals> "}"

<bool_function> ::= "bool" ID "(" <parameters> ")" "{" <program_locals> "}"

<void_function> ::= "void" ID "(" <parameters> ")" "{" <program_locals> "}"

<parameters> ::= /* vazio */
               | <parameter> <parameter_list>

<parameter_list> ::= /* vazio */
                   | "," <parameter> <parameter_list>

<parameter> ::= "int" ID
               | "float" ID
```

```

| "char" ID
| "bool" ID

<declaration_global> ::= <int_declaration_globals>
                        | <float_declaration_globals>
                        | <char_declaration_globals>
                        | <bool_declaration_globals>

<int_declaration_globals> ::= "int" <int_declaration_global>
<int_declaration_global_list> ";"

<int_declaration_global_list> ::= /* vazio */
                                | "," <int_declaration_global>
<int_declaration_global_list>

<int_declaration_global> ::= ID
                        | ID "=" <term_const>
                        | <array_global>

<array_global> ::= ID "[" <term_const> "]"
                | ID "[" <term_const> "]" "=" "{" <array_values_global> "}"

<array_values_global> ::= <term_const>
                        | <array_values_global> "," <term_const>

<float_declaration_globals> ::= "float" <float_declaration_global>
<float_declaration_global_list> ";"
<float_declaration_global_list> ::= /* vazio */
                                | "," <float_declaration_global>
<float_declaration_global_list>
<float_declaration_global> ::= ID
                        | ID "=" <term_const>

<char_declaration_globals> ::= "char" <char_declaration_global>
<char_declaration_global_list> ";"
<char_declaration_global_list> ::= /* vazio */
                                | "," <char_declaration_global>
<char_declaration_global_list>
<char_declaration_global> ::= ID
                        | ID "=" <term_const>

<bool_declaration_globals> ::= "bool" <bool_declaration_global>
<bool_declaration_global_list> ";"
<bool_declaration_global_list> ::= /* vazio */
                                | "," <bool_declaration_global>
<bool_declaration_global_list>
<bool_declaration_global> ::= ID
                        | ID "=" <term_const>

```

```

<program_locals> ::= /* vazio */
                    | <program_local> <program_local_list>

<program_local_list> ::= /* vazio */
                      | <program_local> <program_local_list>

<program_local> ::= <comand>
                  | <declaration_local>

<declaration_local> ::= <int_declaration_locals>
                      | <float_declaration_locals>
                      | <char_declaration_locals>
                      | <bool_declaration_locals>

<int_declaration_locals> ::= "int" <int_declaration_local>
<int_declaration_local_list> ";"
<int_declaration_local_list> ::= /* vazio */
                                | "," <int_declaration_local>
<int_declaration_local_list>
<int_declaration_local> ::= ID
                          | ID "=" <expression>
                          | <array_local>

<array_local> ::= ID "[" <expression> "]"
               | ID "[" <expression> "]" "=" "{" <array_values_local> "}"

<array_values_local> ::= <expression>
                     | <array_values_local> "," <expression>

<float_declaration_locals> ::= "float" <float_declaration_local>
<float_declaration_local_list> ";"
<float_declaration_local_list> ::= /* vazio */
                                | "," <float_declaration_local>
<float_declaration_local_list>
<float_declaration_local> ::= ID
                          | ID "=" <expression>

<char_declaration_locals> ::= "char" <char_declaration_local>
<char_declaration_local_list> ";"
<char_declaration_local_list> ::= /* vazio */
                                | "," <char_declaration_local>
<char_declaration_local_list>
<char_declaration_local> ::= ID
                          | ID "=" <expression>

<bool_declaration_locals> ::= "bool" <bool_declaration_local>
<bool_declaration_local_list> ";"

```

```

<bool_declaration_local_list> ::= /* vazio */
                                | "," <bool_declaration_local>
<bool_declaration_local_list>
<bool_declaration_local> ::= ID
                                | ID "=" <expression>

<comand> ::= <assignment>
            | <if_statement>
            | <while>
            | <for>
            | <printf>
            | <scanf>
            | <return>
            | <call_function> ";"

<assignment> ::= ID "=" <expression> ";"
               | ID "[" <expression> "]" "=" <expression> ";"

<if_statement> ::= "if" "(" <expression> ")" "{" <program_locals> }"
<else_if_chain>

<else_if_chain> ::= /* vazio */
                  | "else" "{" <program_locals> }"
                  | "elseif" "(" <expression> ")" "{" <program_locals> }"
<else_if_chain>

<while> ::= "while" <while_aux> "(" <expression> ")" "{" <program_locals>
}"

<while_aux> ::= /* bloco auxiliar para controle de fluxo, não afeta BNF
principal */

<for> ::= "for" <for_aux> "(" <declaration_local> <expression> ";"
<assignment> ")" "{" <program_locals> }"

<for_aux> ::= /* bloco auxiliar para controle de fluxo, não afeta BNF
principal */

<printf> ::= "printf" "(" STRING <printf_args> ")" ";"

<printf_args> ::= /* vazio */
                | "," <expression> <printf_args>

<scanf> ::= "scanf" "(" STRING <scanf_args> ")" ";"

<scanf_args> ::= /* vazio */
               | "," "&" ID <scanf_args>

```

```

<return> ::= "return" <expression> ";"
          | "return" ";"

<expression> ::= <soma_sub>
               | <mult_div>
               | "(" <expression> ")"
               | <comparison>
               | <log_exp>
               | <cast>
               | <call_function>
               | <term>

<soma_sub> ::= <expression> "+" <expression>
             | <expression> "-" <expression>

<mult_div> ::= <expression> "*" <expression>
             | <expression> "/" <expression>

<comparison> ::= <expression> "<" <expression>
               | <expression> ">" <expression>
               | <expression> "<=" <expression>
               | <expression> ">=" <expression>
               | <expression> "==" <expression>
               | <expression> "!=" <expression>

<log_exp> ::= <expression> "&&" <expression>
            | <expression> "||" <expression>
            | "!" <expression>

<cast> ::= "(" "int" ")" "(" <expression> ")"
        | "(" "int" ")" <term>
        | "(" "float" ")" "(" <expression> ")"
        | "(" "float" ")" <term>
        | "(" "char" ")" "(" <expression> ")"
        | "(" "char" ")" <term>
        | "(" "bool" ")" "(" <expression> ")"
        | "(" "bool" ")" <term>

<call_function> ::= ID "(" <call_parameters> ")"

<call_parameters> ::= /* vazio */
                   | <term> <call_parameter_list>

<call_parameter_list> ::= /* vazio */
                       | "," <term> <call_parameter_list>

<term> ::= NUMBER
        | ID

```

```
| ID "[" <expression> "]"  
| CARACTERE  
  
<term_const> ::= NUMBER  
| CARACTERE
```

## 4. Funcionalidades Suportadas

- **Tipos de Dados:** int, float, char, bool
- **Variáveis:** globais, locais, inicialização múltipla e escopos aninhados
- **Arrays:** unidimensionais com inicialização total ou parcial
- **Funções:** suporte a múltiplos parâmetros, retorno e recursão
- **Controle de Fluxo:** if, else, elseif, while, for, do-while, break, continue
- **Operadores:** aritméticos, relacionais, lógicos, atribuição composta, incremento/decremento, casts
- **Entrada/Saída:** printf, scanf
- **Mensagens de Erro:** detalhadas, com foco em semântica e sintaxe

## 5. Pipeline do Compilador

O pipeline segue a arquitetura clássica:

1. **Análise Léxica (Flex):** geração de tokens a partir do código-fonte.
2. **Análise Sintática (Bison):** construção da árvore sintática e verificação da estrutura gramatical.
3. **Análise Semântica (Bison):** gerenciamento de escopos e tipos, verificação de uso correto.
4. **Geração de LLVM IR:** conversão direta de construtos da linguagem para instruções intermediárias via API C do LLVM.
5. **Geração de Assembly RISC-V 32bits:** utilizando o llc via API C do LLVM.

## 6. Análise Léxica

O arquivo `lexer.l` define os padrões léxicos para:

- Palavras-chave (`int`, `float`, `if`, etc.);
- Identificadores, números, caracteres e strings;
- Operadores compostos;
- Literais booleanos e `char`.

O Flex gera a transição de autômato para uma função `yylex` que reconhece tokens e os repassa ao Bison com seus valores associados.

## 7. Análise Sintática

A gramática em `parser.y` define a estrutura da linguagem, incluindo:

- **Declarações:** globais e locais, com suporte a arrays e inicializações.
- **Funções:** com parâmetros, diferentes tipos de retorno e escopo aninhado.
- **Comandos:** `if`, `while`, `for`, `return`, `printf`, `scanf`, atribuições e chamadas.
- **Expressões:** aritméticas, relacionais, lógicas, com suporte a `cast`.

As ações semânticas acopladas às regras da gramática invocam rotinas que geram LLVM IR.

## 8. Geração de Código LLVM IR

O LLVM IR é gerado diretamente nas ações do parser:

- **Declarações:** `LLVMAddGlobal` para variáveis globais, `LLVMBuildAlloca` para locais.
- **Atribuições:** `LLVMBuildStore`.
- **Expressões:** `LLVMBuildAdd`, `LLVMBuildSub`, `LLVMBuildICmp`, entre outras.
- **Controle de Fluxo:** `LLVMBuildCondBr` e `LLVMBuildBr` com blocos básicos (`LLVMBasicBlockRef`).



- **Funções:** LLVMAddFunction, LLVMGetParam, LLVMBuildCall2.
- **Printf/Scanf:** uso de variáveis com ponteiros (& em scanf) e strings globais via LLVMBuildGEP2.

## 9. Tabela de Símbolos

A tabela de símbolos é uma pilha de escopos implementada em C:

- Suporte a variáveis, arrays, funções e seus parâmetros.
- Armazena metadados como tipo, nome e valor.
- Realiza validação semântica e previne uso incorreto de símbolos.

## 10. Tipos e Conversões

O compilador suporta os tipos `int`, `float`, `char` e `bool`, além de array de `int`. O arquivo `VarType.h` define:

- Enumerações de tipo;
- Conversão de tipos para string;

## 11. Controle de Fluxo

As estruturas `if`, `if-else`, `if-else if`, `if-else if-else`, `while`, `for`, `do-while` são transformadas em blocos básicos com ramificações condicionais. Cada bloco abre e fecha um escopo na tabela de símbolos, garantindo o correto controle de variáveis e contexto.

## 12. Suporte a Entrada e Saída

Funções variáveis `printf` e `scanf` são integradas ao LLVM IR com declarações específicas, strings de formatação globais e verificação do uso correto de ponteiros (& para leitura).

## 13. Exemplos e Testes

O diretório Exemplos/ contém arquivos como:

- `Casts.c`

Exemplos de cast aceitáveis, como `int` para `float` (e vice-versa) e conversões implícitas realizadas, como a soma de `int` e `float`.

- `Condicionais.c`

Exemplos de `if`, `else if` e `else`(aninhados ou não), com expressões lógicas e execução de seus respectivos blocos de código.

- `DeclaracaoInicializavel.c`

Exemplos de inicialização e atribuição(imediata ou não) de variáveis de todos os tipos definidos.

- `Escopo.c`

Demonstração simples de erro utilizando variável fora de escopo.

- `Funcoes.c`

Exemplos de funções com passagem de valores e diferentes tipos de retorno.

- `PrintfScanf.c`

Exemplos de escrita e leitura de valores com funções `printf` e `scanf`.

- `Vetor.c`

Exemplos com vetores(definidos apenas para os inteiros). Demonstrando as mesmas operações de variáveis simples, como atribuições, operações aritméticas e expressões lógicas.

- `WhileFor.c`

Exemplos com laços de repetição, como `for` e `while`.

- `Crivo.c`

Implementação de um algoritmo real do crivo de eratóstenes, que serve para descrever os números primos até determinado valor.

- `Bubble.c`

Implementação de um algoritmo real do bubble-sort, organizando um vetor.

Cada exemplo tem um alvo no `Makefile`, facilitando a compilação e execução.

## 14. Geração de Assembly RISC-V

A geração de Assembly é realizada com `llc`, utilizando o IR como entrada. O código gerado é compilável em RISC-V e executável. Para simulação da saída foi utilizado a IDE (Integrated Development Environment) RARS. Com ele é possível visualizar as saídas e debugar passo a passo das linhas executadas, tanto quanto ver os valores dos registradores.

## 15. Divisão de Trabalho

O desenvolvimento do compilador foi realizado em parceria, com a seguinte divisão de responsabilidades:

- **Felipe Costa Santos (50%)**
- **Hyon Bok Lucas Galvão Mota (50%)**

O trabalho em equipe permitiu o desenvolvimento de um compilador funcional e didático, cobrindo diversas funcionalidades importantes de forma integrada.

## 16. Conclusão

O projeto apresenta um compilador funcional, modular e extensível, cobrindo as fases principais de compilação com base em tecnologias modernas. Com foco didático, fornece uma base sólida para extensão com novas funcionalidades, backends e otimizações, especialmente visando integração com a arquitetura RISC-V.

## 17. Referências

- *LLVM Language Reference Manual*
- *Flex & Bison Manuals*
- [LLVM API C Documentation](#)
- *RISC-V LLVM Backend Documentation*
- GitHub Copilot, *Assistente de Codificação com Inteligência Artificial da OpenAI e GitHub*
- OpenAI ChatGPT, *Assistente de Escrita e Análise Técnica com Inteligência Artificial*