# GCC130 Compiladores
# Relatório da Entrega 3: Análise Semântica e Geração de Código

Bernado Diniz
Luan Shimosaka
Luiz Phillip

Dezembro de 2025

# Sumário

# 1 Introdução

Esta etapa do projeto tem como objetivo implementar a análise semântica e a geração de Código Intermediário (IR) para a linguagem "Mini C". O compilador agora é capaz de:

- Gerenciar escopos aninhados através de uma tabela de símbolos dinâmica.
- Realizar verificação de tipos (Type Checking) em expressões e atribuições.
- Traduzir as estruturas de controle de fluxo (`if`, `while`) e expressões aritméticas para Código de 3 Endereços.

# 2 Estrutura da Tabela de Símbolos

Para suportar declarações de variáveis e escopos aninhados (blocos `{ ... }`), a tabela de símbolos foi implementada como uma **lista encadeada simples**, funcionando logicamente como uma pilha de escopos.

## 2.1 Definição da Estrutura

Cada entrada na tabela contém:

- **Lexema:** O nome do identificador (string).
- **Tipo:** O tipo de dado (`TYPE_INT`, `TYPE_BOOL`).
- **Nível de Escopo:** Um inteiro indicando a profundidade do bloco onde a variável foi declarada.

## 2.2 Gerenciamento de Escopo

Utilizamos uma abordagem de *Stack Allocation* simulada:

- **Entrada de Escopo (`{`):** Incrementa-se o contador global `current_scope`.
- **Inserção:** Novos símbolos são inseridos no início da lista (topo da pilha) com o `current_scope` atual.
- **Saída de Escopo (`}`):** Ao sair de um bloco, percorre-se a lista removendo todos os símbolos que pertencem ao escopo atual, liberando a memória e garantindo que variáveis locais não sejam visíveis fora de seu bloco.

# 3 Verificação de Tipos

O compilador implementa uma tipagem estática forte. As verificações são realizadas durante a análise sintática (Syntax-Directed Translation).

As regras implementadas no módulo `analisadorSemantico.c` incluem:

- **Aritmética:** Operadores `+, -, *, /` aceitam apenas operandos `int` e resultam em `int`.
- **Relacional:** Operadores `<, >, ==, !=` aceitam apenas `int` e resultam em `bool`.
- **Lógica:** Operadores `&&, ||, !` aceitam apenas `bool` e resultam em `bool`.
- **Controle de Fluxo:** As condições de `if` e `while` devem, obrigatoriamente, ser expressões do tipo `bool`.

Caso uma incompatibilidade seja detectada (ex: somar `int` com `bool`), o compilador reporta um erro semântico indicando a linha, mas tenta continuar a compilação para encontrar outros erros.

# 4 Geração de Código Intermediário (IR)

O código gerado é uma representação linear de **3 Endereços**, utilizando:

- **Temporários ($t_n$):** Variáveis criadas automaticamente para armazenar resultados parciais de expressões.
- **Rótulos ($L_n$):** Marcadores de posição para saltos condicionais e incondicionais.

## 4.1 Tradução Dirigida por Sintaxe (Exemplo)

Para gerar código sem a necessidade de múltiplos passos, utilizamos ações semânticas embutidas no Bison. Abaixo, um exemplo da regra de soma:

```
expression: expression T_PLUS expression {
    // 1. Verifica tipos
    $$.type = check_arithmetic($1.type, $3.type);

    // 2. Cria novo temporario (ex: t1)
    char *temp = new_temp();

    // 3. Emite instrucao IR: t1 = x + y
    emit("%s = %s + %s", temp, $1.addr, $3.addr);

    // 4. Propaga o endereco do resultado
    strcpy($$.addr, temp);
}
```

## 4.2 Estratégia de Controle de Fluxo (Marcadores)

Para resolver conflitos de *Shift/Reduce* e *Reduce/Reduce* durante a geração de código para `if` e `while`, abandonamos as ações de "meio de regra"implícitas e adotamos **Marcadores Não-Terminais** (Marker Non-Terminals).

Isso permitiu gerar os rótulos de salto (`goto`, `ifFalse`) nos momentos exatos sem confundir o analisador LR.

**Exemplo da estrutura do While:**

$\langle while\_stmt \rangle$ ::= **while (** $\langle M\_start \rangle$ $\langle expr \rangle$ **)** $\langle M\_cond \rangle$ $\langle stmt \rangle$

- `M_start`: Gera o rótulo de início do laço ($L_{inicio}$).
- `M_cond`: Gera a instrução `ifFalse expr goto L_fim`.
- Ao final da regra, gera-se `goto L_inicio` e posiciona-se o rótulo $L_{fim}$.

# 5 Testes e Validação

Para validar a Etapa 3, foi utilizado um arquivo de teste contendo estruturas aninhadas, operações aritméticas e relacionais.

## 5.1 Arquivo de Teste (teste.bll)

```
int a;
int b;
bool controle;

int main() {
    a = 10;
    b = 20;

    // Teste de Relacionais e If/Else
    if (a < b) {
        print(a);
    } else {
        print(b);
    }

    // Teste de While e Aritmetica
    controle = true;
    while (controle) {
        a = a + 1;
        // If aninhado
        if (a == 15) {
            controle = false;
        }
    }
    print(a); // Deve imprimir 15
}
```

### 5.2 Código Intermediário Gerado (Saída)

Abaixo, o código de 3 endereços gerado pelo compilador para a entrada acima. Nota-se a correta geração de rótulos para o fluxo de controle e o uso de temporários.

```
--- Compilacao Iniciada ---
a = 10
b = 20
t1 = a < b
ifFalse t1 goto L1
print a
goto L2
L1:
print b
L2:
controle = true
L3:
ifFalse controle goto L4
t2 = a + 1
a = t2
t3 = a == 15
ifFalse t3 goto L5
controle = false
goto L6
L5:
L6:
goto L3
L4:
print a
--- Compilacao Finalizada ---
```

# 6 Conclusão

A implementação da análise semântica e geração de código intermediário foi concluída com sucesso. O uso de uma tabela de símbolos baseada em pilha permitiu o controle eficaz de escopos, e a estratégia de Marcadores no Bison resolveu os conflitos de geração de código para estruturas de controle, resultando em um IR limpo e consistente com a lógica do programa fonte.