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, verificar tipos estaticamente e traduzir estruturas de controle para código de três endereços linear.

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 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 `scope_level` atual.
- **Saída de Escopo (`}`):** Ao sair de um bloco, percorre-se a lista removendo todos os símbolos que pertencem ao escopo atual, garantindo que variáveis locais não sejam visíveis fora de seu bloco.

3 Decisões de Projeto

- **Marcadores para Controle de Fluxo (Marker Non-Terminals):** A geração de código para `if` e `while` em uma única passagem (*one-pass compiler*) apresenta um desafio: é necessário gerar instruções de salto (`goto`) para rótulos que ainda não foram definidos ou posicionados. Para evitar o uso complexo de *backpatching* (listas de saltos pendentes), optamos pelo uso de **Marcadores Não-Terminais** na gramática do Bison (ex: `M_if_false`, `M_while_start`). Essas produções vazias permitem executar ações semânticas no "meio"de uma regra gramatical, gerando e empilhando os rótulos (L_1, L_2) no momento exato em que são necessários para a lógica de desvio.
- **Tipagem Estrita e Atribuição como Instrução:** Diferente da linguagem C padrão, onde a atribuição é uma expressão que retorna valor, decidimos tratar a atribuição estritamente como uma instrução (*statement*), retornando tipo `void` na análise semântica. Isso simplifica a análise ao impedir construções como atribuições encadeadas (`a = b = c`) ou atribuições dentro de condições (`if (a=1)`), que frequentemente são fontes de erros lógicos em programação.
- **Geração de Temporários e Rótulos:** Optou-se por um gerenciamento centralizado de nomes através das funções `new_temp()` e `new_label()`. Elas utilizam contadores estáticos globais para garantir que cada variável temporária

$(t_1, t_2...)$ e cada rótulo de desvio $(L_1, L_2...)$ sejam únicos em todo o programa, prevenindo colisões de nomes no código intermediário gerado.

4 Verificação de Tipos

O compilador implementa uma tipagem estática forte. Abaixo, a tabela de regras de tipagem implementadas no analisador semântico.

Tabela 1: Regras de Tipagem e Inferência

Categoria	Operadores	Operandos	Resultado
Aritmética	`+, -, *, /`	`int, int`	`int`
Relacional	`<, >, <=, >=`	`int, int`	`bool`
Igualdade	`==, !=`	`mesmo tipo`	`bool`
Lógica	`&&, ||`	`bool, bool`	`bool`
Unária (Aritmética)	`-`	`int`	`int`
Unária (Lógica)	`!`	`bool`	`bool`
Atribuição	`=`	`X, X`	`void`

Observação: As estruturas de controle `if` e `while` exigem estritamente que a expressão de condição seja do tipo `bool`.

5 Geração de Código Intermediário (IR)

O código gerado é uma representação linear de **3 Endereços**, utilizando temporários (t_n) e rótulos (L_n).

5.1 Tradução Dirigida por Sintaxe (TDS)

A geração de código ocorre simultaneamente à análise sintática (Syntax-Directed Translation). Abaixo, apresentamos a regra semântica (TDS) para a operação de soma:

Listing 1: Ação Semântica para Soma

```
expression: expression T_PLUS expression {
    // 1. Verificacao de Tipos
    $$.type = check_arithmetic($1.type, $3.type);

    // 2. Geracao de Codigo
    char *t = new_temp(); // Cria temporario t_n

    // 3. Emissao da Instrucao IR
    printf("%s = %s + %s\n", t, $1.addr, $3.addr);

    // 4. Propagacao de Atributos (Sintetizados)
    strcpy($$.addr, t);
```

```
}
```

5.2 Estratégia para Controle de Fluxo

Para resolver a geração de código de `if` e `while` em uma única passagem, utilizamos **Marcadores Não-Terminais** na gramática.

Exemplo da regra do `while`:

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

- `M_start`: Ação vazia que gera e retorna um rótulo L_{inicio} antes de avaliar a condição.
- `M_cond`: Ação vazia que gera um rótulo L_{fim} e emite a instrução `ifFalse expr goto L_fim`.
- Ao final da regra `while_stmt`, emite-se `goto L_inicio` e posiciona-se o rótulo L_{fim}.

6 Testes e Validação

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

6.1 Arquivo de Teste (teste.bll)

```
int a;
int b;
bool flag;

int main() {
    // 1. Teste de Entrada e Atribuicao
    read(a);

    // Teste de Aritmetica Complexa + Unario (-)
    // Verifica precedencia (* antes de +) e o operador unario
    b = -a + 5 * 2;

    // Teste de Relacional e Logico
    // Verifica se (b > 0) E se nao e falso (true)
    flag = (b > 0) && !false;

    // Teste de Fluxo (If/Else) e Escopo Aninhado
    if (flag) {
        // Variavel 'temp' so existe dentro deste bloco (
           Escopo)
        int temp;
```

```
21          temp = b;
22          print(temp);
23      } else {
24          print(0);
25      }
26
27      // Teste de Laco (While)
28      while (a < 10) {
29          a = a + 1;
30      }
31
32      print(a);
33  }
```

6.2 Saída do Compilador

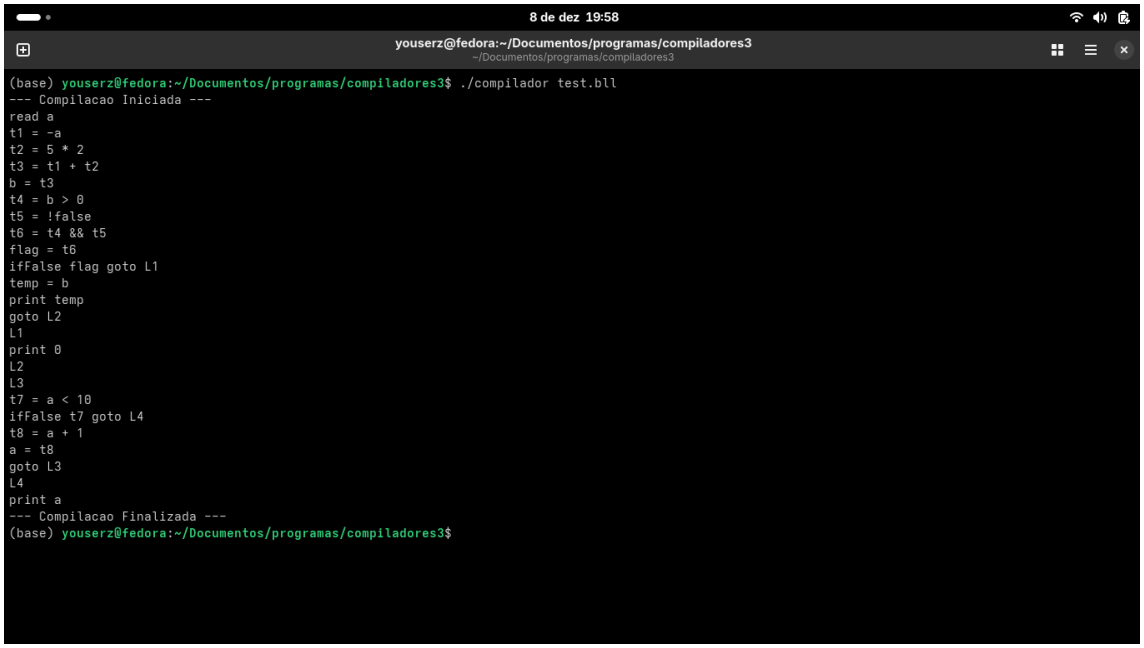

Figura 1: Saída gerada pelo compilador.

7 Dificuldades Encontradas

- **Resolução de Conflitos em Estruturas de Controle:** A introdução dos Marcadores Não-Terminais (`M_...`) no meio das regras de `if` e `while` inicialmente causou conflitos de *Shift/Reduce* no analisador sintático. O Bison tinha dificuldade em decidir se reduzia o marcador vazio ou continuava lendo a expressão. A solução envolveu reestruturar as regras para garantir que os marcadores fossem posicionados inequivocamente após os parênteses das con-

dições, permitindo que o parser "soubesse"que a expressão condicional havia terminado antes de gerar o rótulo de salto.

- **Implementação do Operador Unário (-):** Houve dificuldade em integrar a geração de código do operador "menos unário"(`-a`) na estrutura existente de expressões binárias. Como a precedência já havia sido resolvida na Etapa 2, o desafio na Etapa 3 foi garantir que a ação semântica verificasse corretamente o tipo (apenas `int`) e gerasse a instrução de três endereços correta (ex: `t1 = -t0`) sem interferir na lógica da subtração binária. Foi necessário adicionar uma regra específica com `%prec T_UMINUS` dentro do bloco de `expression`.

- **Gerenciamento de Tipos em Expressões Mistas:** Implementar a verificação rigorosa de tipos exigiu um cuidado extra nas funções auxiliares `check_arithmetic` e `check_logical`. Inicialmente, o compilador permitia operações mistas errôneas ou falhava em propagar o tipo `TYPE_ERROR`. A dificuldade foi garantir que, uma vez detectado um erro em uma sub-expressão, esse erro fosse propagado para cima na árvore sintática, evitando mensagens de erro em cascata ou geração de código inválido para aquela instrução.

- **Escopo de Variáveis em Blocos Aninhados:** Validar se uma variável estava sendo redeclarada no **mesmo** escopo ou apenas sombreando uma variável de um escopo externo foi complexo. A função `insert_symbol` teve que ser ajustada várias vezes para diferenciar a busca: ela precisava falhar apenas se encontrasse o símbolo com o `scope_level` exatamente igual ao atual, mas permitir a inserção se o símbolo existisse apenas em níveis inferiores.