
**UNIVERSIDADE FEDERAL DE LAVRAS**

Bacharelado em Ciência da Computação - 10A

# TRABALHO PRÁTICO DE COMPILADORES

RELATÓRIO 3

GUSTAVO COSTA ALMEIDA,
HENRIQUE CÉSAR SILVA SOARES,
PEDRO MILITÃO MELLO REIS

**Lavras**
**2025**

## 1. INTRODUÇÃO

Nesta terceira e última etapa do projeto de construção do compilador para a linguagem Mini C, o foco desloca-se da análise estrutural do código para a interpretação de seu significado e para a tradução do programa em uma forma intermediária executável. Após a consolidação das análises léxica e sintática nas etapas anteriores, esta fase tem como objetivo incorporar as regras de contexto da linguagem, garantindo que os programas reconhecidos sejam semanticamente válidos e passíveis de tradução.

A análise semântica introduz controles que não podem ser expressos apenas por meio da gramática, como o uso correto de identificadores, o respeito aos escopos léxicos e a compatibilidade de tipos em expressões e comandos. Para isso, o compilador passa a manter uma tabela de símbolos dinâmica, capaz de lidar com escopos aninhados e com o sombreamento de variáveis, além de realizar a verificação estática de tipos antes da geração de qualquer código intermediário.

Paralelamente, esta etapa implementa a geração de código intermediário na forma de código de três endereços, utilizando variáveis temporárias e rótulos explícitos para representar operações e estruturas de controle de fluxo. Essa representação linear facilita a visualização da execução do programa e serve como base para futuras fases de otimização e geração de código alvo.

Ao final do processo, o compilador não apenas valida semanticamente o programa de entrada, mas também produz uma saída detalhada que inclui a tabela de símbolos com informações de escopo, o rastreamento sintático, as mensagens de erro semântico e o código intermediário correspondente. Dessa forma, esta etapa conclui o ciclo fundamental de um compilador, integrando sintaxe, semântica e síntese de maneira coerente e consistente com os princípios estudados na disciplina.

## 2. TABELA DE SÍMBOLOS E GERENCIAMENTO DE ESCOPOS

A tabela de símbolos desempenha um papel central na análise semântica do compilador, sendo responsável por armazenar e organizar as informações associadas aos identificadores declarados no programa fonte. Nesta etapa do projeto, sua estrutura foi estendida para suportar escopos léxicos aninhados, permitindo a verificação correta da visibilidade de variáveis, o sombreamento entre identificadores e a detecção de usos inválidos de símbolos.

A implementação adota uma abordagem baseada em múltiplas tabelas de símbolos, organizadas hierarquicamente por meio de uma estrutura de escopos encadeados. Cada escopo é representado por uma tabela independente, composta por uma estrutura de dispersão (hash table), o que proporciona acesso eficiente aos símbolos locais. Para a função de mapeamento dessa estrutura, utilizou-se o algoritmo DJB2. Essa escolha fundamenta-se em sua eficiência e simplicidade, oferecendo uma excelente distribuição estatística para chaves de texto (strings) e baixas taxas de colisão, o que é crucial para o desempenho das operações de busca e

inserção no compilador. Além disso, cada tabela mantém uma referência para seu escopo pai, formando uma cadeia que reflete diretamente a estrutura léxica do programa.

Durante o processo de análise sintática, a criação de um novo escopo ocorre sempre que o parser identifica o início de um bloco que introduz um novo contexto léxico, como em comandos compostos delimitados por chaves. De maneira análoga, ao final do bloco, o escopo corrente é encerrado, e o compilador retorna ao escopo imediatamente superior. Esse controle garante que as declarações sejam restritas ao seu contexto apropriado e que identificadores não sejam acessíveis além de sua área de validade.

A inserção de símbolos na tabela ocorre no momento da declaração, associando ao identificador informações essenciais para a análise semântica e geração de código, como tipo, escopo de pertencimento e nome interno utilizado no código intermediário. Antes de qualquer inserção, o compilador verifica se já existe um símbolo com o mesmo nome no escopo corrente, permitindo a detecção de declarações duplicadas. No entanto, identificadores de escopos externos podem ser corretamente reutilizados em escopos internos, caracterizando o sombreamento de variáveis.

A busca por símbolos segue uma estratégia hierárquica, iniciando-se no escopo corrente e, em caso de falha, avançando recursivamente para os escopos ancestrais até alcançar o escopo global. Caso o identificador não seja encontrado em nenhum nível, um erro semântico é reportado, indicando o uso de uma variável não declarada.

Uma decisão importante do projeto foi a atribuição de nomes internos únicos aos identificadores durante a geração do código intermediário. Essa renomeação considera o escopo em que a variável foi declarada, evitando ambiguidades causadas por sombreamento e garantindo que cada acesso no código intermediário esteja corretamente associado ao símbolo correspondente. Essa estratégia simplifica a geração de código e elimina conflitos durante a tradução de programas com múltiplos níveis de escopo.

O gerenciamento explícito dos escopos, aliado à organização hierárquica da tabela de símbolos, assegura que todas as verificações semânticas relacionadas a declarações, usos de identificadores e conflitos de nomes sejam realizadas de forma precisa. Essa infraestrutura serve como base para a aplicação das regras de tipagem e para a geração consistente do código intermediário, tratadas nas seções seguintes.

## 3. REGRAS DE TIPAGEM DOS OPERADORES E CONSTRUÇÕES

A análise semântica do compilador é responsável por garantir que as operações realizadas no programa estejam semanticamente corretas, especialmente no que se refere aos tipos dos operandos envolvidos em expressões e comandos. Nesta etapa do projeto, foi implementado um sistema de verificação de tipos baseado em regras explícitas de tipagem, aplicadas durante a análise sintática por meio de ações semânticas associadas às produções da gramática.

A linguagem mini C adotada suporta dois tipos primitivos: int e bool. A partir desses

tipos, foram definidas regras formais que estabelecem quais operadores podem ser aplicados a quais tipos de operandos, bem como o tipo resultante de cada expressão. Sempre que uma regra é violada, o compilador emite um erro semântico, interrompendo a geração do código intermediário correspondente àquela construção.

A Tabela a seguir resume as principais regras de tipagem utilizadas no projeto:

| **Operador / Construção** | **Tipo dos Operandos** | **Tipo Resultante** |
|---|---|---|
| +, -, *, /, % | int, int | int |
| - (unário) | int | int |
| <, <=, >, >= | int, int | bool |
| ==, != | int, int ou bool, bool | bool |
| &&, \|\| | bool, bool | bool |
| ! | bool | bool |
| Atribuição (=) | tipo da variável compatível | tipo da variável |
| Condições (*if*, *while*) | bool | — |

As expressões aritméticas são restritas a operandos do tipo inteiro, refletindo o modelo simplificado da linguagem. Dessa forma, operadores como soma, subtração, multiplicação, divisão e módulo só são permitidos entre expressões do tipo int, produzindo como resultado também um valor inteiro. O operador de menos unário segue a mesma regra, sendo aplicado exclusivamente a operandos inteiros.

Os operadores relacionais, por sua vez, exigem dois operandos inteiros e produzem um valor booleano. Esse comportamento permite que comparações aritméticas sejam usadas diretamente em expressões condicionais. Já os operadores de igualdade e diferença possuem uma regra mais flexível, permitindo comparações entre valores do mesmo tipo, seja int ou bool, mas proibindo comparações entre tipos distintos.

As operações lógicas (&&, || e !) são restritas a operandos booleanos, garantindo que expressões lógicas não sejam combinadas indevidamente com expressões aritméticas. Essa restrição é fundamental para preservar a coerência semântica da linguagem e evitar ambiguidade na interpretação das expressões.

A atribuição exige que o tipo da expressão à direita seja compatível com o tipo da variável à esquerda. Qualquer tentativa de atribuir um valor de tipo incompatível resulta em erro semântico. Essa verificação é realizada consultando a tabela de símbolos, que fornece o tipo declarado da variável, e comparando-o com o tipo inferido da expressão.

Por fim, as expressões utilizadas como condição em comandos de controle de fluxo, como *if* e *while*, devem obrigatoriamente ser do tipo booleano. Essa decisão segue a semântica da linguagem C moderna e elimina interpretações implícitas de valores inteiros como condições lógicas, simplificando a análise semântica e tornando os erros mais explícitos para o programador.

O conjunto dessas regras forma uma base consistente para a geração correta do código intermediário, garantindo que as instruções produzidas estejam semanticamente bem definidas. Além disso, optou-se por implementar a linguagem Mini C com um sistema de tipagem estrita, sem suporte para conversão implícita ou explícita de tipos (casting). Diferentemente da linguagem C padrão, onde inteiros são frequentemente tratados como booleanos (0 é falso, não-zero é verdadeiro), nosso compilador exige que operações lógicas e condições de controle de fluxo (if, while) utilizem exclusivamente o tipo bool.

Essa decisão foi tomada para aumentar a segurança semântica do código, prevenindo erros comuns de lógica decorrentes de coerções automáticas indesejadas. Além disso, essa abordagem simplifica a geração de código intermediário, eliminando a necessidade de instruções de conversão de tipos e garantindo que o comportamento de cada operação seja determinístico e previsível com base apenas na assinatura dos operadores. A próxima etapa explora como essas informações de tipo são utilizadas na tradução das construções de controle de fluxo para a representação intermediária.

## 4. ESTRATÉGIA DE GERAÇÃO DE CÓDIGO INTERMEDIÁRIO

A geração de código intermediário representa a etapa final do compilador desenvolvido neste projeto. Nessa fase, o código fonte da linguagem mini C, previamente validado léxica, sintática e semanticamente, é traduzido para uma representação intermediária linear no formato de código de três endereços. Essa representação utiliza variáveis temporárias e rótulos explícitos para modelar o fluxo de controle, aproximando-se da estrutura interna adotada por compiladores reais.

A tradução foi implementada diretamente nas ações semânticas da gramática, integrando a verificação de tipos à emissão do código intermediário. Para isso, o compilador utiliza funções auxiliares responsáveis pela criação de novos temporários, geração de rótulos exclusivos e emissão formatada das instruções. Todos os comandos são produzidos de forma incremental à medida que as regras gramaticais são reduzidas.

As construções de controle de fluxo, em especial if/else e *while*, exigem estratégias específicas para garantir que o fluxo de execução seja representado corretamente na forma linear do código intermediário. Essas estratégias são descritas a seguir.

No comando condicional if, o código intermediário é gerado a partir da avaliação da expressão booleana associada à condição. O valor resultante dessa expressão é armazenado

em uma variável temporária, que é utilizada em uma instrução de desvio condicional. Caso a condição seja avaliada como falsa, o fluxo de execução é redirecionado para um rótulo que marca o início do bloco alternativo (else) ou, na ausência deste, diretamente para o ponto após o comando condicional. Quando o bloco else está presente, um desvio incondicional é inserido ao final do bloco if para evitar que o fluxo caia indevidamente no bloco alternativo, garantindo a semântica correta da estrutura.

Essa estratégia permite que o código do bloco verdadeiro seja executado de forma sequencial, evitando saltos desnecessários e tornando o código intermediário mais legível. O gerenciamento dos rótulos é realizado de maneira sistemática, assegurando que cada comando condicional possua pontos de entrada e saída bem definidos no fluxo de execução.

A geração do código intermediário para o comando de repetição *while* utiliza uma abordagem semelhante, porém adaptada à natureza cíclica da construção. Um rótulo é gerado para marcar o início da avaliação da condição do laço, permitindo que o fluxo retorne a esse ponto após a execução do corpo. A condição é avaliada e, caso resulte em falso, o fluxo é desviado para um rótulo que representa a saída do laço. Caso contrário, o código do corpo do *while* é executado normalmente, seguido por um salto incondicional que retorna ao início da condição, caracterizando o comportamento iterativo.

Essa organização garante que a condição seja testada antes de cada iteração, respeitando a semântica do laço *while*. A separação clara entre rótulo de início, corpo e ponto de saída facilita tanto a leitura do código intermediário quanto sua correta execução por estágios posteriores do compilador.

Além disso, a estratégia de geração de código implementa curto-circuito lógico para os operadores && e || utilizados em expressões condicionais. Nesses casos, a avaliação completa da expressão pode ser interrompida assim que o resultado final for determinado. Isso é implementado por meio de desvios condicionais intermediários e rótulos adicionais, evitando a execução desnecessária de subexpressões e prevenindo erros de execução.

A escolha por gerar o código intermediário diretamente durante a análise sintática simplifica a arquitetura do compilador e elimina a necessidade de uma árvore sintática abstrata intermediária. Embora essa abordagem exija maior cuidado na organização das ações semânticas, ela resulta em uma implementação mais direta e alinhada com o objetivo pedagógico do projeto.

Com essa estratégia, o compilador é capaz de representar corretamente estruturas de decisão e repetição, mesmo quando aninhadas, preservando a semântica do programa fonte e produzindo uma representação intermediária clara, consistente e adequada para futuras etapas de otimização ou geração de código alvo.

## 5. TRADUÇÃO DIRIGIDA POR SINTAXE (TDS)

A integração entre análise semântica e geração de código intermediário neste projeto foi realizada por meio de um esquema de Tradução Dirigida por Sintaxe, no qual as ações semânticas são executadas diretamente durante o processo de redução das regras gramaticais. Esse modelo permite que a verificação de tipos e a emissão do código intermediário ocorram de forma incremental, acompanhando exatamente a estrutura sintática do programa analisado.

Para viabilizar essa abordagem, foram associados atributos sintetizados às principais categorias sintáticas da gramática, em especial às expressões. Cada expressão carrega informações suficientes para que o compilador compreenda tanto o seu significado semântico quanto sua representação no código intermediário. Esses atributos encapsulam o tipo da expressão, utilizado para a verificação semântica, e o endereço associado, que corresponde ao nome da variável, constante ou temporário onde o valor da expressão é armazenado no código de três endereços.

Durante a redução das produções, as ações semânticas são responsáveis por validar a compatibilidade dos operandos envolvidos, determinar o tipo resultante da expressão e gerar o código intermediário correspondente. Caso uma inconsistência semântica seja detectada, como a aplicação de um operador aritmético a operandos de tipo incompatível, um erro é reportado com informação de localização precisa, e a expressão resultante é marcada com um tipo especial de erro. Essa estratégia evita que falhas semânticas se propaguem de forma descontrolada ao longo da análise, permitindo que o compilador continue o processamento de maneira segura.

Um exemplo representativo da Tradução Dirigida por Sintaxe adotada é a produção que descreve a soma de duas expressões inteiras. Quando a regra sintática correspondente é reduzida, o compilador primeiro verifica se ambos os operandos possuem tipo inteiro. Em caso afirmativo, o tipo da expressão resultante é definido como inteiro, um novo temporário é alocado para armazenar o resultado da operação e uma instrução de três endereços é emitida, representando explicitamente a soma. O endereço desse temporário passa, então, a representar a expressão composta em produções superiores da árvore sintática.

```C/C++
 | expression TK_PLUS expression {
        if ($1.type == DT_INTEGER && $3.type == DT_INTEGER){
            $$.type = DT_INTEGER;
            $$.addr = newTemp();
            emit("%s = %s + %s", $$.addr, $1.addr, $3.addr);
        } else {
            yyerror("Semantic Error: Operacao '+' requer inteiros.");
            $$.type = DT_ERROR; $$.addr = "ERR";
        }
        add_reduce_trace("expr -> +");
    }
```

| Produção | Regras semânticas |
| --- | --- |
| expressao → expressao$_1$ + expressao$_2$ | if expressao$_1$.tipo == inteiro && expressao$_2$.tipo == inteiro<br>expressao.tipo = inteiro<br>expressao.temp = novo_temp()<br>adicionar_instrucao(expressao.temp = expressao$_1$.temp + expressao$_2$.temp)<br>else<br>erro("Operação '+' requer inteiros")<br>expressao.tipo = erro<br>expressao.temp = "ERR" |

Essa abordagem garante que cada subexpressão seja processada de forma independente e que suas informações semânticas estejam disponíveis no momento exato em que são necessárias. Além disso, o uso de temporários evita ambiguidades e reflete fielmente o comportamento de registradores intermediários em compiladores reais.

O mesmo esquema é estendido para operadores relacionais e lógicos, nos quais o tipo resultante é booleano, bem como para operadores unários, como o menos unário e a negação lógica. Nos casos de operadores lógicos com curto-circuito, as ações semânticas associadas às produções utilizam rótulos e desvios condicionais adicionais, garantindo que a avaliação da expressão respeite a semântica da linguagem e que o código intermediário reflita corretamente o fluxo de execução.

A Tradução Dirigida por Sintaxe também é utilizada na implementação das atribuições, nas quais o compilador verifica se o tipo da expressão à direita é compatível com o tipo da variável à esquerda antes de emitir a instrução correspondente no código intermediário. Dessa forma, o sistema assegura que apenas atribuições semanticamente válidas sejam incorporadas à saída.

A escolha por um esquema de TDS baseado exclusivamente em atributos sintetizados simplifica o fluxo de informações no analisador e evita a necessidade de estruturas auxiliares mais complexas, como atributos herdados ou árvores sintáticas intermediárias. Embora essa decisão exija maior disciplina na escrita das ações semânticas, ela resulta em uma implementação clara, eficiente e perfeitamente alinhada aos objetivos pedagógicos do projeto.

Com essa estratégia, o compilador consegue associar sintaxe, semântica e geração de código de maneira coesa, garantindo que cada construção da linguagem mini C seja analisada e traduzida de forma correta e sistemática.

## 6. ARQUIVO DE TESTE E SAÍDA (IR)

```C/C++
// teste_semantico_valido.mc
// Testa: Tipos, Escopos (Shadowing), Aritmética, Lógica, Relacional e Fluxo.
int g;          // Declaração Global
bool b;
int x = 10; // Inicialização INT
// Testa: Aritmética (+, *, -, /) e Menos Unário
x = x + 5 * -2 / 1;
// Testa: Relacional (>, <, ==) retornando BOOL
b = x > 0 == true;
// Testa: Lógica (&&, ||, !) e Controle de Fluxo (IF requer BOOL)
if (!b || x != 10) {
    // Testa: Shadowing (Redeclaração legal em novo escopo)
    bool x = false;
    // Testa: Controle de Fluxo (WHILE requer BOOL)
    while (x == false) {
        print(g);   // Acesso a global dentro de escopo aninhado
        read(g);
        x = true;   // Atribuição na variável local (bool)
    }
}
```

### 6.1 Código Intermediário Gerado (Saída)

A saída abaixo reflete exatamente a formatação implementada no codegen.c (tabela ASCII) e a lógica de renomeação de variáveis por escopo (sufixos _0, _1, etc.) implementada no parser.y.

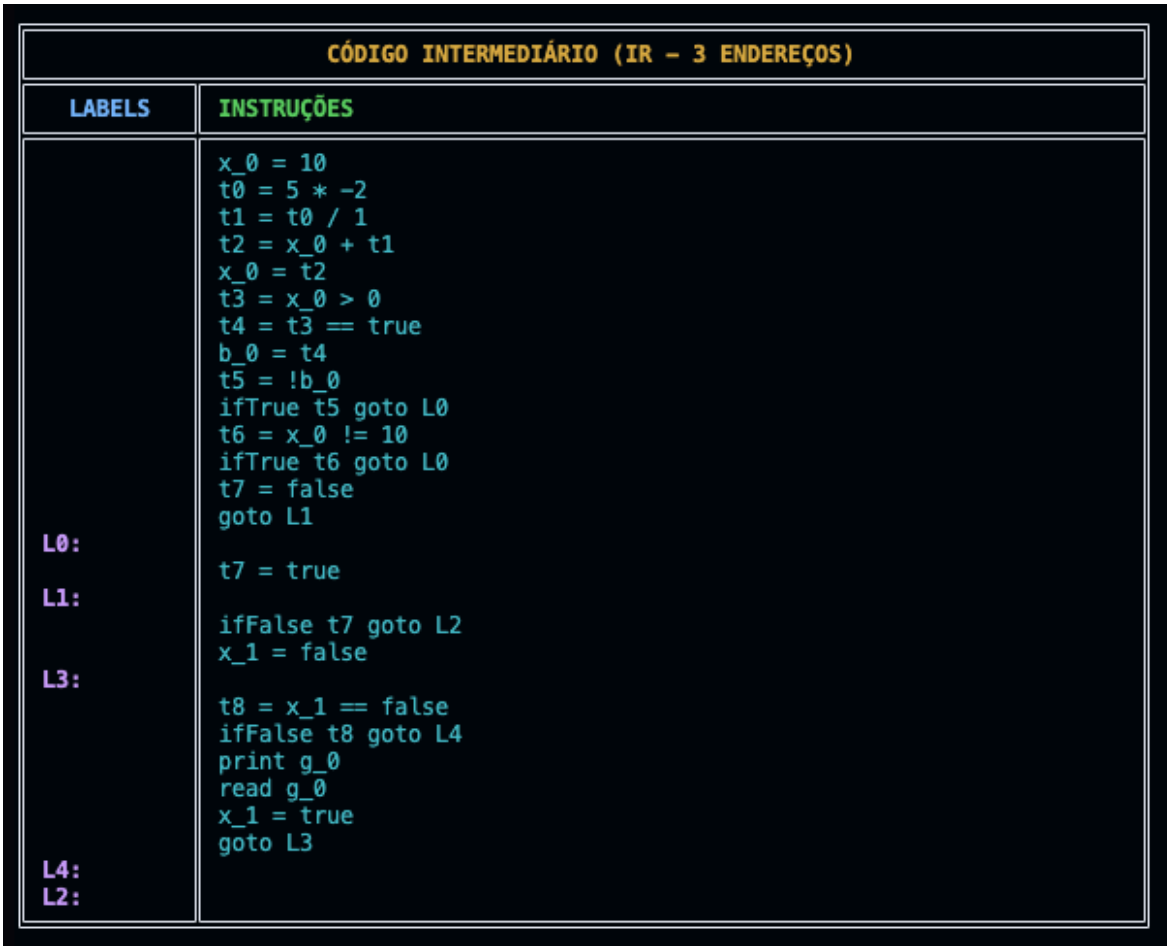


**Figura 1. IR gerado apartir do teste_semantico_valido.mc**

Nota-se no código acima a aplicação do curto-circuito lógico (labels L2/L3) na condição do *while* e o uso correto dos sufixos de escopo.

Para validar a implementação da análise semântica e da geração de código intermediário, foi elaborado um programa de teste que exercita simultaneamente as principais funcionalidades do compilador desenvolvido. O código fonte escolhido contempla declaração de variáveis globais, operações aritméticas e relacionais, operadores lógicos com curto-circuito, atribuições e estruturas de controle de fluxo aninhadas, permitindo avaliar o correto funcionamento integrado de todas as etapas da tradução.

A análise desse programa permite verificar não apenas a correção sintática e semântica, mas também a fidelidade da tradução para a representação intermediária. Em especial, o teste evidencia a atuação do mecanismo de curto-circuito lógico, garantindo que subexpressões sejam avaliadas apenas quando necessário, e confirma o uso adequado de rótulos e desvios condicionais na modelagem do fluxo de controle.

A saída gerada pelo compilador corresponde ao código intermediário no formato de três endereços, organizado em uma tabela para facilitar a leitura e a inspeção manual. Cada variável do código fonte é renomeada internamente com um sufixo que identifica o escopo em que foi declarada, eliminando ambiguidades e refletindo corretamente a resolução de

identificadores implementada pela tabela de símbolos. As expressões intermediárias são armazenadas em variáveis temporárias, enquanto os pontos de decisão e repetição são representados por rótulos explícitos e instruções de salto.

A sequência de instruções evidencia a tradução correta do laço *while*, com a presença de um rótulo que marca o início da avaliação da condição, desvios condicionais que controlam a permanência no laço e um salto incondicional que assegura a iteração. Da mesma forma, a estrutura if interna é traduzida por meio da avaliação da condição em um temporário, seguida por desvios que controlam a execução condicional do bloco associado.

O código intermediário resultante demonstra que o compilador é capaz de linearizar estruturas de controle complexas sem perda de informação semântica, preservando fielmente o comportamento do programa original. Além disso, a clareza da saída gerada facilita a depuração e comprova a correção do processo de tradução, atendendo plenamente aos objetivos definidos para esta etapa do projeto.

## 7. DECISÕES DO PROJETO

Durante a implementação da análise semântica e da geração de código intermediário, foram tomadas decisões de projeto fundamentais para garantir a correção semântica, a clareza da tradução e a fidelidade do compilador à linguagem Mini C proposta. Nesta fase, optou-se por soluções que aproximassem o projeto do funcionamento de compiladores reais, mesmo ao custo de maior complexidade de implementação.

A principal decisão estrutural foi a adoção de uma tabela de símbolos com suporte a escopos léxicos aninhados, implementada como uma cadeia de tabelas hash interligadas por ponteiros para o escopo pai. Cada novo bloco estrutural da linguagem (como *if*, *else* e *while*) gera dinamicamente um novo escopo, permitindo modelar corretamente o escopo estático e o sombreamento de variáveis. Essa abordagem possibilita que a resolução de identificadores obedeça rigorosamente à regra de “definição mais interna prevalece”, além de permitir diagnósticos semânticos precisos.

Cada símbolo armazenado na tabela contém informações completas, como tipo de dado, profundidade e identificador do escopo, além da posição no código fonte. A decisão de associar um identificador único de escopo a cada símbolo foi essencial para a geração correta do código intermediário, pois permite diferenciar variáveis homônimas declaradas em contextos distintos. Essa estratégia viabilizou o renomeamento interno das variáveis no IR por meio de sufixos (_0, _1, etc.), eliminando ambiguidades e tornando explícito o contexto de cada acesso no código gerado.

Outra decisão importante foi a utilização exclusiva de atributos sintetizados na Tradução Dirigida por Sintaxe. Cada expressão carrega consigo tanto o tipo semântico quanto o endereço associado no código intermediário, encapsulados em uma estrutura própria. Essa escolha permitiu integrar verificação de tipos e geração de código de forma incremental,

garantindo que cada redução gramatical produza imediatamente uma instrução intermediária válida quando semanticamente possível.

A geração do código intermediário seguiu o modelo de três endereços, utilizando temporários e rótulos explícitos para controle de fluxo. Optou-se por realizar a emissão do código diretamente nas ações semânticas do Bison, dispensando a construção de uma árvore sintática abstrata intermediária. Embora essa decisão aumente a responsabilidade do parser, ela mantém o projeto mais compacto e proporciona maior visibilidade sobre o processo de tradução.

Para as estruturas de controle if/else e *while*, foi adotada uma estratégia baseada em saltos condicionais e incondicionais (ifFalse e goto), utilizando rótulos gerados dinamicamente. No caso do if/else, empregou-se a estratégia de "queda" (fall-through) para o bloco verdadeiro, evitando saltos desnecessários e produzindo código mais claro. Já o laço *while* foi implementado com rótulos explícitos para início e saída, permitindo a correta repetição do teste condicional.

Por fim, decidiu-se implementar curto-circuito lógico explícito para os operadores *&&* e ||, reproduzindo fielmente a semântica da linguagem C. Essa escolha exigiu o uso de ações de meio de regra no Bison para controlar o fluxo de execução e evitar a avaliação desnecessária do segundo operando, além de prevenir possíveis erros em tempo de execução. Apesar de aumentar a complexidade do parser, essa decisão garantiu correção semântica e maior realismo ao compilador implementado.

## 8. DIFICULDADES ENCONTRADAS

A etapa de análise semântica e geração de código intermediário apresentou dificuldades substancialmente mais complexas do que aquelas enfrentadas nas fases léxica e sintática, sobretudo devido à forte interdependência entre escopos, tipos e fluxo de controle. Muitos dos problemas encontrados não se manifestavam imediatamente, surgindo apenas após longas sequências de reduções, o que tornou o processo de depuração mais desafiador.

Uma das principais dificuldades foi o gerenciamento correto dos escopos léxicos em conjunto com a tabela de símbolos. A abertura e o fechamento de escopos precisaram ser cuidadosamente sincronizados com as reduções do analisador sintático, especialmente em estruturas como *if*/*else* e *while* com corpo único. Pequenos erros nessa sincronização resultavam em falhas sutis, como variáveis permanecendo visíveis fora de seu escopo válido ou símbolos sendo indevidamente reportados como não declarados. Esses problemas tornavam-se particularmente difíceis de identificar em comandos aninhados, nos quais múltiplos escopos são criados e destruídos durante uma única derivação.

Outro desafio relevante esteve relacionado à propagação de erros semânticos. Uma vez identificado um erro de tipo ou de uso de identificador, era necessário impedir a geração de código intermediário inconsistente, sem interromper completamente a análise. Para isso, foi necessário introduzir um tipo especial de erro semântico que pudesse ser propagado pelas

expressões subsequentes. Determinar quando continuar a análise e quando suprimir a geração de código exigiu ajustes sucessivos nas ações semânticas, pois um tratamento inadequado frequentemente resultava em erros em cascata ou em código intermediário inválido.

A implementação do curto-circuito lógico para os operadores && e || também se mostrou particularmente complexa. Ao contrário das operações aritméticas, esses operadores exigem controle explícito do fluxo de execução, com geração antecipada de rótulos e desvios condicionais intermediários. O uso de ações de meio de regra introduziu dependências temporais entre diferentes partes de uma mesma produção gramatical, dificultando o raciocínio sobre a ordem correta de geração do código. Erros nessa etapa raramente eram reportados de forma direta, manifestando-se apenas como falhas lógicas no código intermediário, o que exigiu inspeção manual detalhada da saída gerada.

Outro ponto crítico foi a associação consistente entre os símbolos da tabela e seus endereços no código intermediário. Embora o renomeamento por escopo tenha eliminado ambiguidades, ele introduziu o risco de inconsistências entre a tabela de símbolos e o IR, especialmente em cenários com sombreamento de variáveis. Garantir que cada uso de identificador referencie exatamente o símbolo correto exigiu validações adicionais e testes específicos.

A integração da verificação semântica com a geração de código diretamente nas ações do Bison também impôs desafios de organização. Sem uma separação conceitual clara, pequenos erros de implementação podiam causar efeitos colaterais significativos, como código sendo emitido mesmo após a detecção de inconsistências semânticas. Isso exigiu uma organização rigorosa das ações e atenção constante à ordem de execução.

Além disso, o gerenciamento de memória dinâmica apresentou dificuldades recorrentes. O uso intensivo de strings para temporários, rótulos e endereços demandou cuidado especial para evitar vazamentos de memória e sobrescritas indevidas. Esses problemas nem sempre se manifestavam de forma imediata, aparecendo apenas em testes mais extensos, o que dificultou sua identificação e correção.

Por fim, a ausência de uma árvore sintática abstrata impactou diretamente o processo de depuração. Sem uma representação estrutural intermediária, a compreensão do comportamento do compilador dependeu fortemente da análise manual do código intermediário e do uso de logs de depuração. Embora isso tenha tornado o diagnóstico mais trabalhoso, também contribuiu para uma compreensão mais profunda do funcionamento interno da tradução dirigida por sintaxe e das decisões tomadas ao longo do projeto.

## 9. CONCLUSÃO

A terceira etapa do projeto conclui a construção do compilador da linguagem mini C, integrando análise semântica e geração de código intermediário de forma consistente e robusta. O trabalho permitiu aplicar, na prática, conceitos fundamentais como escopos léxicos, verificação de tipos, atributos sintetizados, controle de fluxo e tradução intermediária.

O compilador resultante é capaz de identificar erros em múltiplos níveis, compreender o significado dos programas analisados e produzir uma representação intermediária fiel à semântica da linguagem. A experiência proporcionou uma compreensão aprofundada da arquitetura interna de compiladores, evidenciando a interdependência das diferentes fases do processo e consolidando o aprendizado teórico da disciplina.

Com isso, o projeto atinge seu objetivo pedagógico, oferecendo uma visão completa e integrada do funcionamento de um compilador, desde a leitura do código fonte até a geração de código intermediário, preparando uma base sólida para estudos futuros em otimização e geração de código alvo.