RELATÓRIO DE COMPILADORES

Professor: Ricardo Terra Nunes Bueno Villela

Alunos: Felipe Crisóstomo Silva Oliveira
Henrique Assis Moreira
Julia Aparecida de Faria Morais

SUMÁRIO

1. INTRODUÇÃO

O trabalho prático da disciplina de **Compiladores (GCC 130)** do ***2° Semestre de 2025***, tem como objetivo apresentar o desenvolvimento na construção de um **analisador léxico** utilizando a ferramenta **Flex**. Essa etapa é fundamental, pois o analisador léxico corresponde ao primeiro módulo de um compilador, sendo responsável por transformar a sequência de caracteres do código-fonte em uma sequência de tokens significativos para as etapas posteriores da compilação.

A motivação se encontra na importância de compreender de forma prática como se dá o reconhecimento de padrões léxicos em linguagens de programação, além de exercitar a implementação de técnicas de análise que desconsidere espaços em branco e comentários, identifique corretamente lexemas e reporte erros com suas respectivas posições no código. Dessa forma, a construção do analisador léxico proporciona uma visão concreta do funcionamento inicial de um compilador, servindo como base para as etapas seguintes de análise sintática e semântica.

Neste relatório, o leitor encontrará uma descrição das principais **decisões de projeto** tomadas durante a implementação, a exposição das **dificuldades encontradas**, bem como a apresentação de **diagramas de transição (DFAs)** para duas classes de tokens. Além disso, são incluídos **arquivos de teste** utilizados para validar a implementação, juntamente com suas respectivas saídas.

2. DESCRIÇÕES DO PROJETO

2.1. Decisões

2.1.1. Desmembramento do conjunto de palavras reservadas

Para as próximas fases do trabalho, consideramos mais adequado que cada palavra reservada possua o seu próprio token, o que facilitará o tratamento e a análise durante o processo de compilação.

2.1.2. Regras para os identificadores

Mantivemos o padrão adotado pela maioria das linguagens de programação, no qual os identificadores podem iniciar com letras ou com o caractere “_”, e, a partir daí, podem conter números, letras e “_”. Não há restrições quanto ao caractere final do identificador.

2.1.3. Tratamento de erros

Definimos o tratamento de dois tipos de erros léxicos, cada um com mensagens específicas:

- **Identificadores iniciados por números**, que são inválidos;
- **Caracteres não reconhecidos pela linguagem**.

Em ambos os casos, o compilador gera uma mensagem de erro que indica a natureza do problema e o caracteriza como **erro léxico**, informando também a linha e a coluna em que ocorreu.

2.1.4. Resolução de Ambiguidade do *“Menus Unário”*

O maior desafio de ambiguidade, além do ***dangling else***, foi o token **`T_MINUS` (`-`)**, que pode ser um **operador binário (subtração)** ou **unário (negação)**. Para resolver isso, seguimos uma prática padrão do Yacc :

1. Criamos um *"token fantasma"* chamado `UMINUS` na declaração de precedência.
2. Atribuímos a `UMINUS` a maior precedência de todas (junto com `T_NOT`), com associatividade à direita.
3. Na regra gramatical do menos unário, usamos a diretiva `%prec UMINUS`:

Isso diz ao Bison: *"Embora esta regra termine com o token `expr`, sua precedência não deve ser a de `expr`. Em vez disso, sua precedência é a do `UMINUS`."*

Como resultado, em uma expressão como `x = -5 * 10;`, o parser sabe que o `-` (unário) tem precedência maior que o `*` (binário), agrupando a expressão corretamente como `x = (-5) * 10;`.

2.1.5. Resolução da Ambiguidade do *“dangling else”*

A solução adotada para lidar com a ambiguidade do *“dangling else”* foi incorporada diretamente na estrutura da gramática. A ideia principal é separar todas as regras de comando (`stmt`) em duas categorias: **comandos casados** (`matched_stmt`) e **comandos abertos** (`open_stmt`).

Um **comando casado** (`matched_stmt`) representa uma instrução sintaticamente completa, ou seja, que não termina com um `if` sem um `else` correspondente. Nessa categoria entram estruturas como `if-else` completas, `while`, blocos de código (`{...}`) e instruções simples, como atribuições.

Já um **comando aberto** (`open_stmt`) é especificamente um `if` que ainda não possui um `else` associado, ficando “aberto” para que um `else` futuro possa ser vinculado a ele.

Essa distinção é fundamental, pois elimina a ambiguidade: a produção da gramática para `if-else` exige que o corpo do `if` seja obrigatoriamente um `matched_stmt`. Com isso, quando o parser encontra um `else`, ele

automaticamente o associa ao `open_stmt` mais interno e próximo, que é a única forma sintaticamente correta de derivação.

Dessa maneira, o conflito **shift/reduce**, que seria o sinal da ambiguidade, é evitado já no nível da gramática.

2.1.6. Lógica de Precedência de Operadores

Um dos requisitos centrais do analisador sintático era analisar corretamente expressões aritméticas, relacionais e lógicas. Uma gramática ambígua, como `expr: expr T_SUM expr | expr T_MULT expr`, não consegue decidir por si só a ordem de avaliação, gerando conflitos de "shift-reduce".

Para resolver essas ambiguidades, utilizamos as diretivas de precedência e associatividade do Bison, conforme definido no início do arquivo `parser.y`. A ordem de declaração dessas diretivas é crucial, pois as últimas declaradas possuem a maior precedência.

Nossa hierarquia de precedência foi definida da seguinte forma (da mais baixa para a mais alta):

1. `%left T_OR`: O operador lógico "OU" tem a menor precedência.
2. `%left T_AND`: O operador lógico "E" tem precedência sobre "OU".
3. `%nonassoc T_EQUAL T_NOT_EQUAL`: Operadores de igualdade.
4. `%nonassoc T_LESSER T_GREATER T_LESSER_EQUAL T_GREATER_EQUAL`: Operadores relacionais.
5. `%left T_SUM T_MINUS`: Operadores de adição e subtração.
6. `%left T_MULT T_DIV T_MOD`: Operadores de multiplicação, divisão e módulo (resto).
7. `%right T_NOT UMINUS`: Operadores unários (NOT lógico e "menos unário"), que possuem a maior precedência de todas.

Decisões de Associatividade:

- **`%left` (Associatividade à Esquerda):** Foi usada para os operadores aritméticos (`+`, `-`, `*`, `/`) e lógicos (`&&`, `||`). Isso garante que expressões como `a - b + c` sejam interpretadas corretamente como `(a - b) + c`.

- `%nonassoc` **(Não Associativo):** Foi usada para os operadores relacionais e de igualdade. Essa é uma decisão de projeto que proíbe o encadeamento desses operadores (ex: `a < b == c`), forçando o programador a usar parênteses para definir a ordem, o que evita ambiguidades semânticas.
- `%right` **(Associatividade à Direita):** Foi usada para os operadores unários (`!` e o `UMINUS`), permitindo que sejam aplicados da direita para a esquerda.

2.1.7. Tratamento e Reporte de Erros no Sintático

A estratégia de tratamento e reporte de erros sintáticos foi focada em fornecer mensagens claras e específicas, conforme exigido pelo trabalho, em vez de depender da mensagem padrão do Bison.

Para isso, a função `yyerror` foi modificada. Implementamos uma lógica que intercepta a mensagem genérica ("syntax error") emitida pelo Bison e a modifica. Dessa forma, mensagens customizadas, definidas por nós, são exibidas ao usuário em conjunto com as emitidas pelo Bison.

Implementamos a recuperação de erros diretamente na gramática usando o token especial `error`. Definimos regras específicas para capturar padrões de erros comuns e, dentro dessas regras, chamamos a função `yyerror` com uma mensagem descritiva.

Todas as mensagens customizadas são formatadas pela `yyerror` para incluir a linha e a coluna exatas da ocorrência do erro, informações rastreadas pelo analisador léxico. Após reportar o erro, a macro `yyerrok` é chamada na maioria das regras de recuperação para limpar o estado de erro do parser, permitindo que a análise continue no restante do arquivo.

2.1.8. Estrutura de Dados

A Tabela de Símbolos foi implementada como uma **Pilha de Escopos** (Stack of Scopes) utilizando listas ligadas, o que permite o aninhamento.

1. `struct SymbolEntry`: Representa a entrada de um símbolo, armazenando o **identificador** (`id`), o **tipo** (`Tipo` - `int` ou `bool`) e o **nível de escopo**.

2. `struct Scope`: Representa um bloco de escopo. Contém a **cabeça** (`head`) da lista de `SymbolEntry` (símbolos declarados no bloco) e um ponteiro `prev` que aponta para o escopo pai, formando a pilha.

2.1.9. Gerenciamento de Escopo

O gerenciamento segue o princípio **LIFO** (Last-In, First-Out) para refletir a estrutura de blocos do código.

- **Abertura (`push_scope`)**: Chamada no terminal `T_LEFT_BRACKET` (`{`). Cria um novo `Scope`, incrementa o nível de escopo e o coloca no topo da pilha global (`current_scope`).
- **Fechamento (`pop_scope`)**: Chamada no terminal `T_RIGHT_BRACKET` (`}`). Remove o `Scope` do topo, restaura o ponteiro `current_scope` para o escopo anterior e, crucialmente, **libera toda a memória** alocada para os símbolos e a estrutura do escopo recém-saído.

2.1.10. Detecção de Erros Semânticos

Duas checagens essenciais foram implementadas:

1. **Redeclaração**: A função `check_redeclaration(id)` é invocada antes de toda inserção. Ela busca o identificador *apenas* no **escopo atual**. Se for encontrado, reporta um erro semântico.
2. **Identificador Não Declarado**: A função `lookup_symbol(id)` é usada para verificar o uso. Ela busca o identificador a partir do **escopo atual** e, se não encontrar, segue para os escopos pais (o `prev` da pilha) até o global. Se a busca falhar, reporta um erro semântico de uso indevido.

2.1.11. Integração com o Parser

A lógica da TS foi acoplada diretamente às regras gramaticais relevantes no `parser.y`:

- **Gerenciamento de Escopo**: `push_scope()` e `pop_scope()` foram integradas nas regras de `compound_stmt` (blocos `{}`).

- **Declaração (Inserção/Redeclaração)**: As funções `check_redeclaration()` e `insert_symbol()` são chamadas nas regras de `simple_stmt` que tratam de declaração de variáveis (`T_INT T_ID...`, `T_BOOLEAN T_ID...`).
- **Uso (Consulta)**: A função `lookup_symbol()` é chamada na regra `primary_expr` (para o token `T_ID`) para validar se a variável em uso foi declarada.

2.1.12. Sistema de Tipos

A gramática foi estendida para suportar atributos sintetizados, onde as produções não-terminais (`expr`, `primary_expr`) propagam o tipo resultante (`TIPO_INT` ou `TIPO_BOOL`) de forma ascendente (*bottom-up*). As regras de compatibilidade foram aplicadas conforme a tabela a seguir:

- **Aritmética (`+ - * / %`):** Aceita estritamente operandos inteiros, resultando em inteiro.
- **Relacional (`< > <= >=`):** Aceita estritamente operandos inteiros, resultando em booleano.
- **Lógica (`&& || !`):** Aceita estritamente operandos booleanos.
- **Igualdade (`== !=`):** Implementada para aceitar estritamente operandos do tipo inteiro (comparação `int` com `int`), resultando sempre em um valor booleano. Comparações envolvendo booleanos (ex: `bool == bool` ou `int == bool`) são reportadas como erros semânticos.
- **Controle de Fluxo (`if`, `while`):** Impõe que a expressão de condição seja estritamente do tipo booleano.

2.1.13. Tratamento de Erros Semânticos e Recuperação

Foi implementada a rotina *yysemanticerror*, responsável por reportar inconsistências semânticas utilizando as coordenadas de linha e coluna fornecidas pelo analisador léxico. Para a recuperação de erros, adotou-se a estratégia de propagação de tipo indefinido (`TIPO_UNDEFINED`). Ao detectar uma incompatibilidade (como somar `int` com `bool`), o compilador reporta o erro e retorna `TIPO_UNDEFINED` para o nó pai. As regras subsequentes ignoram

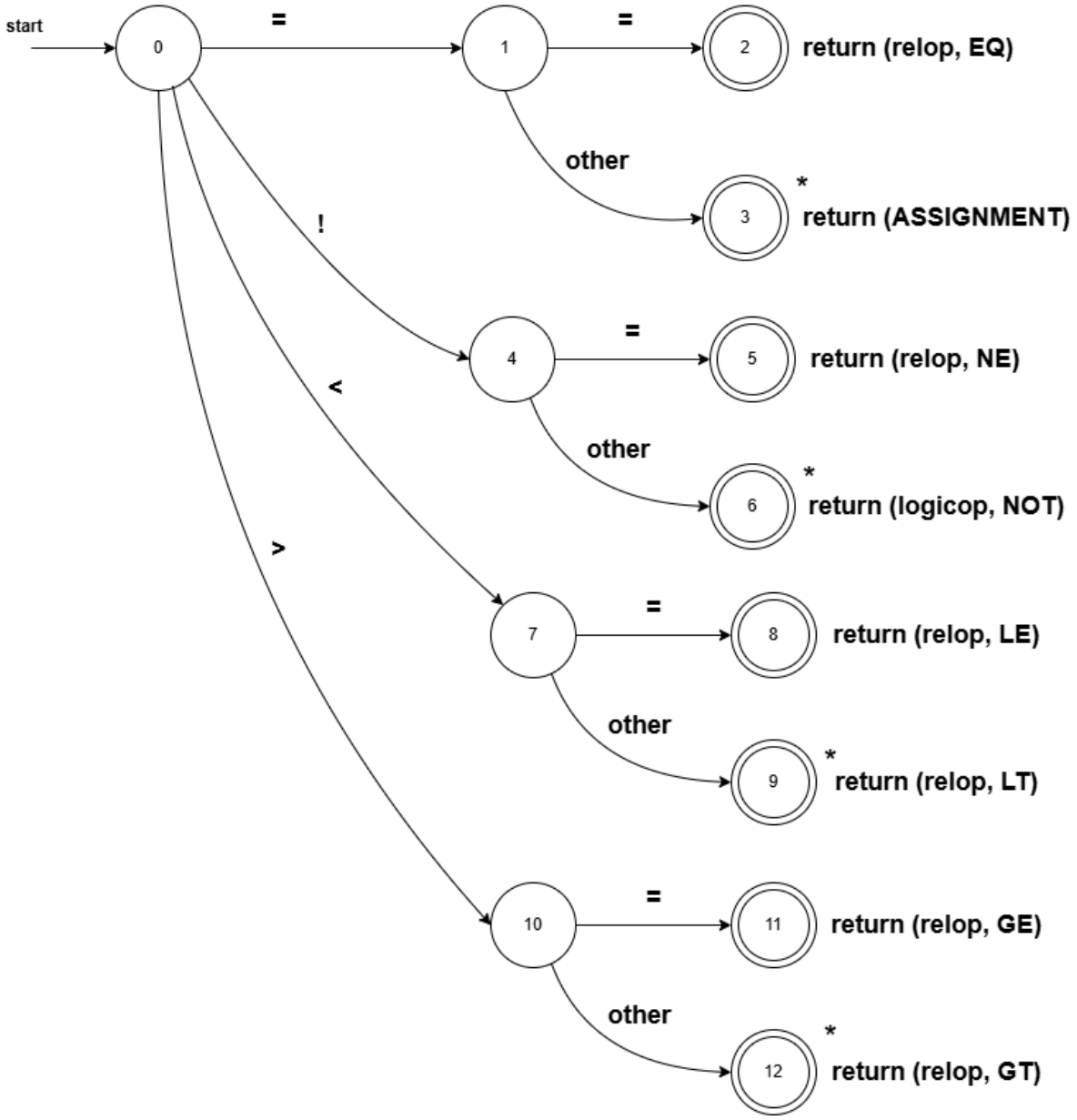
operações com esse tipo especial, prevenindo o "efeito cascata" de mensagens de erro derivadas de uma única falha original.

2.1.14. Estratégia de Geração de IR para if/else e while.

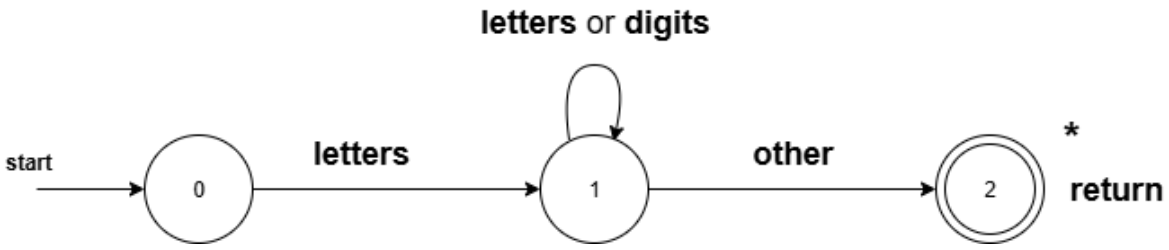
Para a geração de código intermediário das estruturas de controle de fluxo (`if`/`else` e `while`), adotou-se uma estratégia de linearização baseada na emissão de rótulos únicos (`labels`) e instruções de salto (`GOTO` e `IF_FALSE`). Para gerenciar o escopo e permitir o aninhamento correto dessas estruturas, utilizou-se uma pilha auxiliar de rótulos. No processamento de condicionais, o compilador gera saltos condicionais para desviar o fluxo caso a expressão lógica seja falsa, e saltos incondicionais para evitar a execução incorreta de blocos alternativos (`else`). De forma análoga, para os laços de repetição, definem-se rótulos de início e fim, permitindo que o fluxo retorne ao cabeçalho do laço para reavaliação da condição ou encerre a iteração quando a guarda se tornar falsa.

3. DIAGRAMAS DE TRANSIÇÃO

3.1. Diagrama de *RELOP*



3.2. Diagrama de *ID*



4. GRAMÁTICA COMPLETA EM BNF

```
None
<program> ::= <stmt_list>

<stmt_list> ::= <stmt>
              | <stmt_list> <stmt>

<stmt> ::= <matched_stmt>
         | <open_stmt>

<compound_stmt> ::= T_LEFT_BRACKET T_RIGHT_BRACKET
                  | T_LEFT_BRACKET <stmt_list> T_RIGHT_BRACKET

<matched_stmt> ::= <simple_stmt>
                 | <compound_stmt>
                 | T_IF T_LEFT_PAREN <expr> T_RIGHT_PAREN
<matched_stmt> T_ELSE <matched_stmt>
                 | T_WHILE T_LEFT_PAREN <expr> T_RIGHT_PAREN
<matched_stmt>

<open_stmt> ::= T_IF T_LEFT_PAREN <expr> T_RIGHT_PAREN <stmt>
              | T_IF T_LEFT_PAREN <expr> T_RIGHT_PAREN
<matched_stmt> T_ELSE <open_stmt>

<simple_stmt> ::= T_BOOLEAN T_ID T_ATRIBUTION <expr>
T_SEMICOLON
                | T_INT T_ID T_ATRIBUTION <expr> T_SEMICOLON
                | T_PRINT T_LEFT_PAREN <expr> T_RIGHT_PAREN
T_SEMICOLON
                | T_ID T_ATRIBUTION <expr> T_SEMICOLON
                | T_READ T_LEFT_PAREN T_ID T_RIGHT_PAREN
T_SEMICOLON

<expr> ::= <primary_expr>
         | <expr> T_SUM <expr>
         | <expr> T_MINUS <expr>
         | <expr> T_MULT <expr>
```

```
          | <expr> T_DIV <expr>
          | <expr> T_MOD <expr>
          | <expr> T_AND <expr>
          | <expr> T_OR <expr>
          | T_NOT <expr>
          | <expr> T_EQUAL <expr>
          | <expr> T_NOT_EQUAL <expr>
          | <expr> T_LESSER <expr>
          | <expr> T_GREATER <expr>
          | <expr> T_LESSER_EQUAL <expr>
          | <expr> T_GREATER_EQUAL <expr>
          | T_MINUS <expr>
          | T_LEFT_PAREN <expr> T_RIGHT_PAREN

<primary_expr> ::= T_NUMBER
                 | T_ID
                 | T_TRUE
                 | T_FALSE
```

5. FIRST & FOLLOW

5.1. Conjuntos FIRST

FIRST(expr) = { T_NUMBER, T_ID, T_TRUE, T_FALSE, T_NOT, T_MINUS, T_LEFT_PAREN **}**

FIRST(primary_expr) = { T_NUMBER, T_ID, T_TRUE, T_FALSE **}**

5.2. Conjuntos FOLLOW

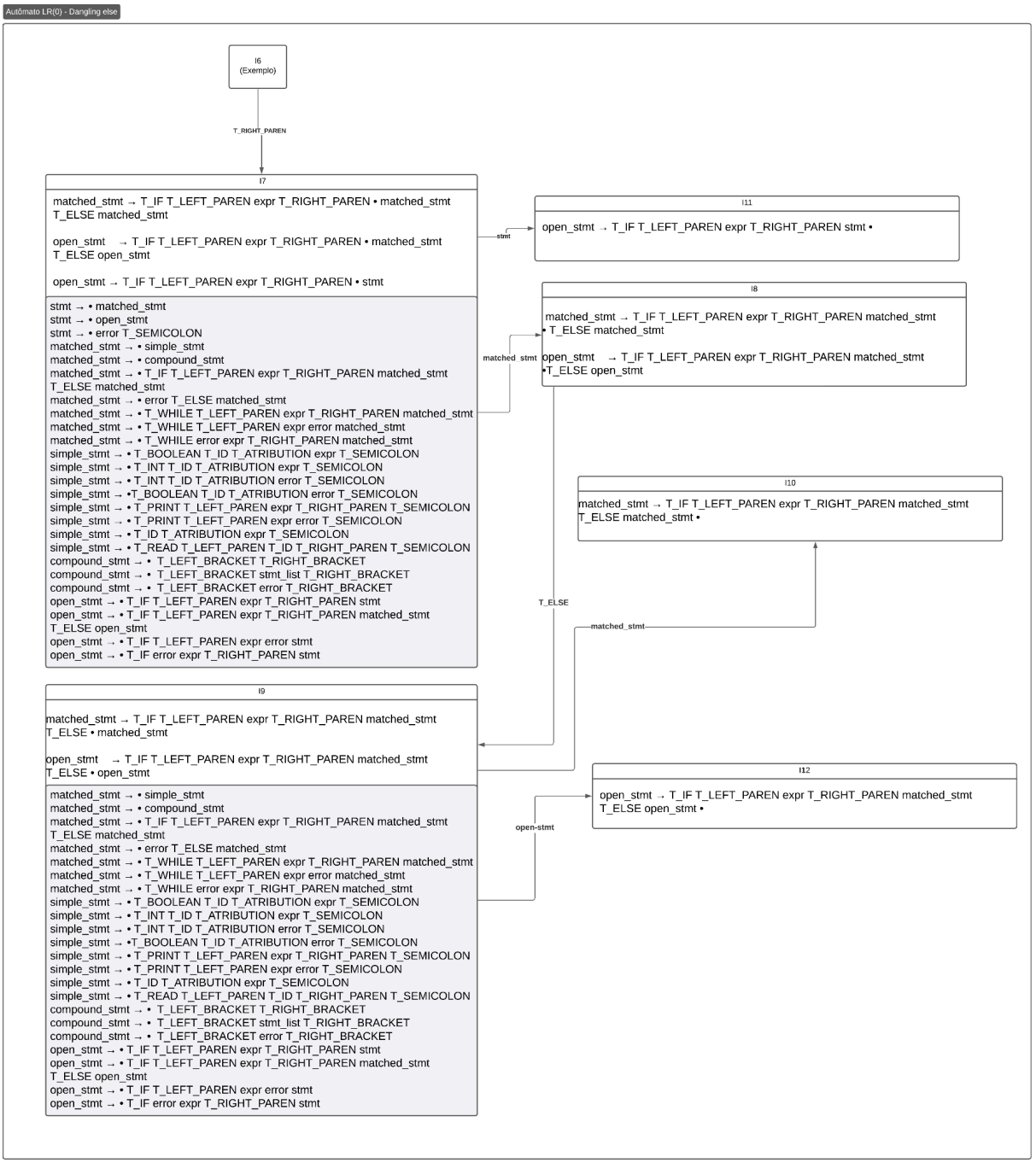
FOLLOW(expr) = { $, T_SUM, T_MINUS, T_MULT, T_DIV, T_AND, T_OR, T_EQUAL, T_NOT_EQUAL, T_LESSER, T_GREATER, T_LESSER_EQUAL, T_GREATER_EQUAL, T_RIGHT_PAREN **}**

FOLLOW(primary_expr) = FOLLOW(expr)

6. AUTÔMATO LR(0)

6.1. Dangling Else

Segue o autômato LR(0) com conflito *dangling else.*
Esse conflito ocorre quando o analisador sintático não consegue determinar a qual comando “if” um “else” deve ser associado, podendo se manifestar como um conflito do tipo shift/reduce durante a análise sintática.



A seguir, é apresentada a linha do tempo dos estados que ilustram o surgimento e a resolução do conflito.

- Estado I6 → I7

Após o reconhecimento de `T_RIGHT_PAREN`, o parser desloca para o estado I7. Nesse ponto, o analisador já leu `(expr)` e está prestes a processar o corpo do comando condicional.
Ainda não há conflitos, pois o parser apenas aguarda o início do corpo do `if`.

- Estado I7

Nesse estado, o parser terminou de ler a condição do `if` e o ponto está antes do corpo do comando. O conflito surge porque o analisador pode seguir por dois caminhos válidos:

1. Reconhecer `stmt` e reduzir para `open_stmt`;
2. Ou esperar um `else` para formar um comando completo `if-else`.

Como o autômato LR(0) não utiliza *lockheed*, ele não sabe se o próximo símbolo será `else` ou outro token. Dessa forma, o analisador não consegue decidir entre reduzir o if como um comando completo(sem `else`) ou fazer shift e continuar aguardando o `else`.
Esse é o ponto em que o conflito *dangling else* surge pela primeira vez.

- Estados I8, I9, I10 e I12

Após o parser optar por continuar (shift), o fluxo segue pelos seguintes estados:

- **I8:** o corpo do `if` foi reconhecido e o parser aguarda `T_ELSE`;
- **I9:** o `else` é lido e o corpo correspondente é processado;
- **I10** e **I12:** o comando `if–else` completo é reduzido.

Nesses estados, não há ambiguidade, pois o `else` já foi associado ao `if` mais próximo.

- Estado I11

Neste estado, o parser terminou de reconhecer um comando `if (expr) stmt` e está pronto para reduzir para `open_stmt`.
Entretanto, se o próximo símbolo for `else`, o correto seria fazer shift para associar o `else` ao `if` atual.
Como o autômato LR(0) não possui *lookahead*, ele pode reduzir incorretamente, gerando novamente o conflito *shift/reduce*.

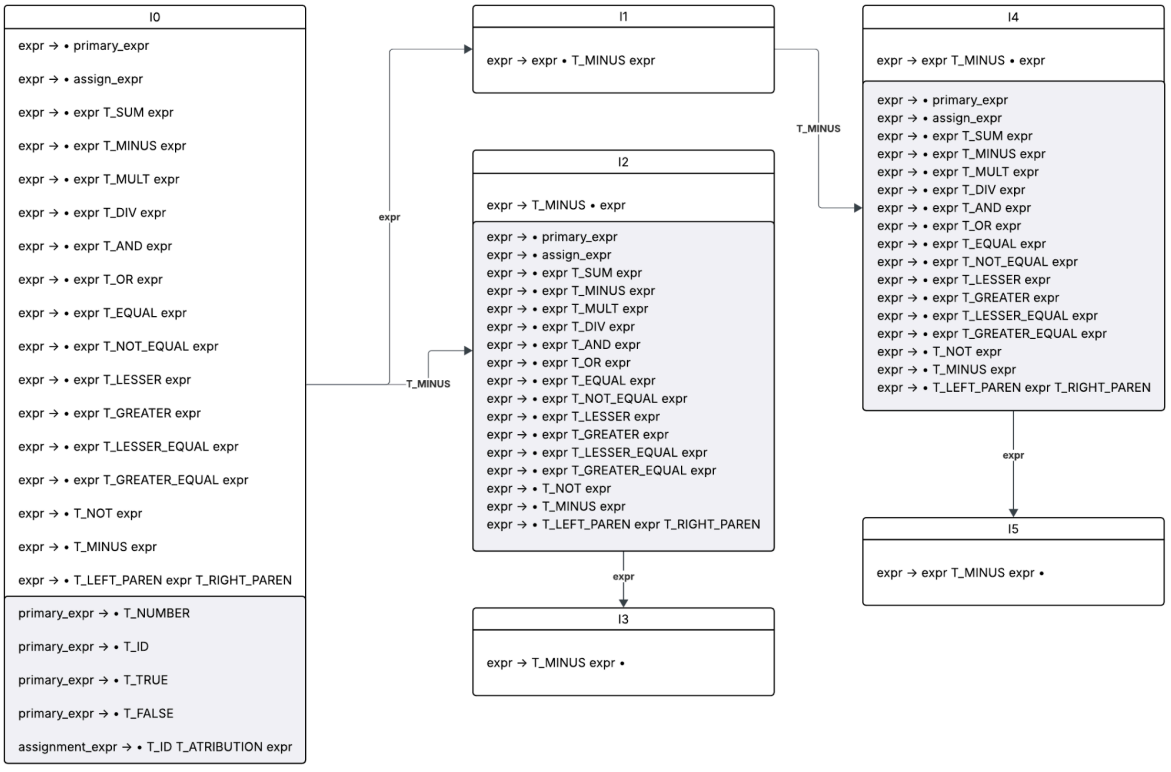
Assim, o mesmo problema observado em I7 reaparece aqui.

Observação

A numeração dos estados apresentada (I6, I7, I8, I9, I10, I11 e I12) **é apenas ilustrativa** e serve para representar o encadeamento lógico do conflito.
Como a gramática completa é extensa, a numeração real dos estados no autômato LR(0) pode variar conforme a ferramenta de geração e a ordem interna das regras.

6.2. Menos Unário

Segue o autômato LR(0) com conflito *unary minus.*
Esse conflito ocorre quando o analisador sintático não consegue determinar se o símbolo “-” é referente à uma operação binária ou unária.



Comentando os estados:

- Estado I0

Esse é o estado referente ao começo da expressão e todas as suas possíveis produções.

- Estado I1

Esse estado vem depois que um *shift* é feito em alguma *expr*, sinalizando que é esperado uma operação binária..

- Estado I2

Esse estado vem depois que um *shift* é feito no token T_MINUS, sinalizando que é esperado que o operador “-” faça a função de um operador unário.

- Estado I3

.Esse estado vem depois que um *shift* é feito em uma *expr* após um *shift* em um token T_MINUS, sinalizando que encerrou a expressão unária.

- Estado I4

Esse estado vem depois que um *shift* é feito no token T_MINUS após um *shift* em alguma *expr*, sinalizando que é esperado uma outra expressão para finalizar a operação binária.

- Estado I5

Esse estado vem depois do I4 quando ocorre um *shift* em *expr*, sinalizando que encerrou a operação binária.

7. TDS - TRADUÇÃO GERADA POR SINTAXE

7.1. TDS da expressão de adição

Produção Gramatical	Regras Semânticas (Ações)
(1) E → E_1 + E_2	E.addr = newTemp() emit({'ADD'}, E_1.addr, E_2.addr, E.addr)
(2) E → number	E.addr = {lookUp}(number.lexval)
(3) E → id	E.addr = {lookUp}(id.lexval)

8. CONJUNTO DE INSTRUÇÕES DO CÓDIGO IR

Conjunto de Instruções do Código Intermediário (IR)

Abaixo estão listadas as instruções de três endereços suportadas pelo compilador, onde:

- `op1` e `op2`: São operandos fonte (variáveis, temporários ou literais).
- `dest`: É o operando de destino (onde o resultado é armazenado).
- `label`: É um rótulo de marcação no código (ex: `L1:`).

8.1. Instruções Aritméticas

Instrução	Formato no IR	Descrição
ADD	ADD op1, op2, dest	Realiza a soma: `dest` = `op1` + `op2`
SUB	SUB op1, op2, dest	Realiza a subtração: `dest` = `op1` - `op2`
MUL	MUL op1, op2, dest	Realiza a multiplicação: `dest` = `op1` * `op2`
DIV	DIV op1, op2, dest	Realiza a divisão inteira: `dest` = `op1` / `op2`
MOD	MOD op1, op2, dest	Realiza o módulo: dest = op1 % op2

MINUS_UNARY	MINUS_UNARY op1, dest	Inverte o sinal (unário): `dest` = -`op1`

8.2. Instruções Lógicas e Relacionais

Instrução	Formato no IR	Descrição
AND	AND op1, op2, dest	E lógico: `dest` = `op1` && `op2`
OR	OR op1, op2, dest	OU lógico: `dest` = `op1`
NOT	NOT op1, dest	NÃO lógico: `dest` = !`op1`
EQ	EQ op1, op2, dest	Igualdade: `dest` = (`op1` == `op2`)
NEQ	NEQ op1, op2, dest	Diferença: `dest` = (`op1` != `op2`)
LT	LT op1, op2, dest	Menor que: `dest` = (`op1` < `op2`)
GT	GT op1, op2, dest	Maior que: `dest` = (`op1` > `op2`)
LTE	LTE op1, op2, dest	Menor ou igual: `dest` = (`op1` <= `op2`)
GTE	GTE op1, op2, dest	Maior ou igual: `dest` = (`op1` >= `op2`)

8.3. Transferência de Dados, Entrada e Saída

Instrução	Formato no IR	Descrição
ASSIGN	ASSIGN op1, dest	Atribuição simples: `dest` = `op1` (Copia o valor)
PRINT	PRINT op1	Imprime o valor de `op1` na saída padrão
READ	READ dest	Lê um valor da entrada padrão e salva em `dest`

8.4. Controle de Fluxo

Instrução	Formato no IR	Descrição
IF_FALSE	IF_FALSE op1, label	Desvio condicional: Se `op1` for falso (0), pula para `label`
GOTO	GOTO label	Desvio incondicional: Pula obrigatoriamente para `label`
Label	L{n}:	Define um rótulo de destino no código (ex: `L0:`, `L1:`)

9. TESTES EXECUTADOS

9.1. Arquivo de Teste

teste_sintatico.txt

```
int 1q = -0;        // Erro Léxico: ID começando com numero
int x = ;           // Erro Sintático: Esperava expressão depois de '='
bool z = /;         // Erro Sintático: Não deveria ter '/' atribuido
bool z = true;
if (x > 5 {         // Erro Sintático: Faltando ')'
    print(a);

    int @a = 1;     // Erro Léxico: Caractere especial inesperado

x = 5 + * 2;    // Erro Sintático: Operador inesperado

print(x             // Erro Sintático: Faltando ')'
;                   // O parser vai se recuperar neste ';', /*
                    //assim é necessário olhar para trás do ';' para enxergar o erro

else { }            // Erro Sintático: 'else' sem 'if'. O parser vai se recuperar no '}'

while (x) {         // Erro Sintático: '}' faltando no final do arquivo,
    x = x - 1;}     // mas que não consigo identificar ou tratar da maneira correta ao indificar o que falta
                    // Ele até conta colunas de comentários de tão potente que é

```

9.1. Saída no Terminal

9.1.1. Erros

```
Erro Sintatico na Linha 1, Coluna 13: -> Expressao ou atribuicao mal formada

Erro Sintatico na Linha 2, Coluna 10: syntax error
Erro Sintatico na Linha 2, Coluna 10: -> Atribuicao invalida para int

Erro Sintatico na Linha 3, Coluna 11: syntax error
Erro Sintatico na Linha 3, Coluna 12: -> Atribuicao invalida para boolean

Erro Sintatico na Linha 5, Coluna 12: syntax error
ERRO: @ - Linha: 8 | Coluna: 9 - Messagem: Token Inesperado: "@". Lexical Error.
Erro Sintatico na Linha 10, Coluna 10: syntax error
Erro Sintatico na Linha 10, Coluna 10: -> Detectado '}' ausente

Erro Sintatico na Linha 10, Coluna 10: -> Detectado ')' ausente no 'if'

Erro Sintatico na Linha 10, Coluna 10: syntax error
Erro Sintatico na Linha 10, Coluna 13: -> Expressao ou atribuicao mal formada

Erro Sintatico na Linha 13, Coluna 2: syntax error
Erro Sintatico na Linha 13, Coluna 2: -> Detectado ')' ausente no 'print'

Erro Sintatico na Linha 15, Coluna 5: syntax error
Erro Sintatico na Linha 15, Coluna 9: -> 'else' sem 'if' previamente


========================================
Analise sintatica concluida com sucesso!
========================================
```

9.1.2. Tabela de Símbolos

```
---- Tabela de Simbolos ----
1 Token: <NUMBER> | Value: 0 | Line: 1 | Column: 11
2 Token: <ID> | Value: x | Line: 2 | Column: 5
3 Token: <ID> | Value: z | Line: 3 | Column: 6
4 Token: <NUMBER> | Value: 5 | Line: 5 | Column: 9
5 Token: <ID> | Value: a | Line: 6 | Column: 11
6 Token: <NUMBER> | Value: 1 | Line: 8 | Column: 14
7 Token: <NUMBER> | Value: 5 | Line: 10 | Column: 5
8 Token: <NUMBER> | Value: 2 | Line: 10 | Column: 11
9 Token: <NUMBER> | Value: 1 | Line: 18 | Column: 13
----------------------------
```

9.2. Testes de IR Gerados

9.2.1. Arquivo de Teste

teste_geracao.txt

```
int a = 10;
int b = 20;
int soma = 0;

print(a);
print(b);

if (a < b) {
    soma = a + b;
    print(soma);
} else {
    print(0);
}

int contador = 0;

while (contador < 5) {
    print(contador);
    contador = contador + 1;
}
```

9.2.2. Arquivo gerado com o código IR

saida.ir

```
    ASSIGN 10, a
    ASSIGN 20, b
    ASSIGN 0, soma
    PRINT a
    PRINT b
    LT a, b, t0
    IF_FALSE t0, L0
    ADD a, b, t1
    ASSIGN t1, soma
    PRINT soma
    GOTO L1
L0:
    PRINT 0
L1:
    ASSIGN 0, contador
L2:
    LT contador, 5, t2
    IF_FALSE t2, L3
    PRINT contador
    ADD contador, 1, t3
    ASSIGN t3, contador
    GOTO L2
L3:

```

10. DIFICULDADES ENCONTRADAS

10.1. Analisador Léxico

10.1.1. Problema com Contagem de Colunas

Foi identificado um problema relacionado ao cálculo da posição da coluna no código-fonte. Inicialmente, o incremento da coluna era realizado de forma unitária, independentemente do tamanho do lexema reconhecido. Dessa forma, palavras compostas por múltiplos caracteres eram contabilizadas como se possuíssem apenas uma coluna, gerando inconsistências na indicação de posição dos tokens.

A solução adotada consistiu em ajustar o contador de colunas para considerar o comprimento efetivo do lexema reconhecido, de modo que a posição final fosse corretamente incrementada pelo número de caracteres do token, e não apenas por uma unidade. Essa abordagem garante maior precisão na localização dos tokens e, consequentemente, na geração de mensagens de erro e relatórios de análise.

10.1.2. Problema Com Sequência de Regras

Durante os testes, foi observado um problema relacionado à ordem de aplicação das regras no analisador léxico. Um lexema composto por múltiplos caracteres alfabéticos era reconhecido prematuramente pela regra destinada a um único caractere (`letters`), impedindo que fosse corretamente classificado como identificador (`id`).

A solução implementada consistiu em reordenar as regras de modo que identificadores fossem reconhecidos prioritariamente e, posteriormente, remover a regra específica de `letters`, visto que esta se tornou redundante e inacessível no fluxo de análise. Essa modificação assegurou o reconhecimento adequado de identificadores e eliminou conflitos de precedência entre as expressões regulares.

10.1.3. Aceitação de -0

Durante o desenvolvimento da Etapa 1 (Analisador Léxico), o protótipo inicial tratava números negativos, incluindo `-0`, como um único token diretamente no Flex.

Para a Etapa 2 (Analisador Sintático), essa abordagem foi refatorada. A responsabilidade de identificar números negativos foi movida do analisador léxico para o analisador sintático, o que representa um design mais robusto.

A implementação foi alterada da seguinte forma:

1. **Analisador Léxico (Flex):** A regra de expressão regular para números (`digits`) foi simplificada para reconhecer apenas inteiros não-negativos (ex: `([0-9]+)`). O sinal de subtração (`-`) agora é reconhecido como seu próprio token, `T_MINUS`.
2. **Analisador Sintático (Bison):** Foi definida uma regra gramatical para o operador unário menos (`| T_MINUS expr %prec UMINUS`).

Com essa mudança, uma entrada como `int x = -0;` não é mais vista pelo lexer como `T_INT`, `T_ID`, `T_ATRIBUTION`, `T_NUMBER(-0)`, `T_SEMICOLON`. Em vez disso, ela é tokenizada como a sequência `T_INT`, `T_ID`, `T_ATRIBUTION`, `T_MINUS`, `T_NUMBER(0)`, `T_SEMICOLON`.

10.1.4. Não Compreensão Sobre Tabela de Símbolos

Na Etapa 1, a estrutura apresentada como "Tabela de Símbolos" funcionava, na prática, como um registro cronológico de *todos* os tokens reconhecidos pelo analisador léxico. Como evidenciado no relatório anterior, ela listava indiscriminadamente identificadores, números, operadores (`<ASSIGNMENT>`, `<RELOP>`), palavras-chave (`<IF>`) e símbolos.

Essa abordagem não cumpria o papel de uma tabela de símbolos tradicional, que é armazenar e gerenciar atributos de lexemas específicos (como variáveis e constantes) para consulta nas etapas posteriores da compilação.

Para a Etapa 2, a implementação foi corrigida e alinhada aos requisitos do projeto. Conforme o código atualizado em `lexer.l`, a função `addSymbol` agora é chamada seletivamente:

1. Apenas tokens das classes **<ID>** e **<NUMBER>** são adicionados à tabela.
2. Operadores, palavras-chave e pontuação não são mais inseridos, pois não representam dados a serem armazenados.
3. Foi implementada a função `findSymbol`, que é invocada antes de adicionar um `<ID>` para evitar a inserção duplicada de identificadores.

A nova "Tabela de Simbolos" gerada reflete essa mudança, agindo agora como um repositório útil para futuras análises semânticas, ao invés de um simples log de tokens.

10.1.5. Comentário Multilinha (/* */) Não Atualiza Linhas

Na Etapa 1, foi identificado um problema em que o analisador léxico não atualizava corretamente o contador global de linhas ao processar comentários de múltiplas linhas (`/* ... */`). Isso causava inconsistências na localização de tokens e erros que aparecessem após um bloco de comentário.

O problema foi solucionado na implementação atual. A regra de expressão regular para comentários (`{comment}`) agora está associada a uma ação em C que, em vez de simplesmente descartar o lexema, itera por todo o texto capturado (armazenado na variável `yytext`).

Dentro desse loop, o código verifica explicitamente a existência de caracteres de quebra de linha (`\n`). Para cada `\n` encontrado dentro do comentário, o contador global `line` é incrementado , e o contador `column` é reiniciado para 1.

Essa abordagem garante que, mesmo que um comentário se estenda por dezenas de linhas, o analisador mantém a contagem de linhas precisa, assegurando que a posição de todos os tokens subsequentes seja reportada corretamente.

10.2. Analisador Sintático

10.2.1. O “Fecha Parênteses” — Um Erro?

Uma dificuldade notável foi observada ao tratar de erros de sintaxe onde um bloco de escopo (`{ ... }`) não é fechado antes do fim do arquivo, conforme ilustrado no `teste_sintatico.txt` .

O comportamento identificado é o seguinte:

1. O analisador léxico processa todo o arquivo, incluindo os comentários finais, atualizando as variáveis globais de posição `line` e `column` a cada token e comentário processado .
2. O analisador sintático (`parser.y`) empilha os tokens do `while` e de seu `stmt_list` interno.
3. Ao final da entrada, o lexer retorna o token de fim de arquivo (`EOF`).
4. Nesse momento, o parser, que estava em um estado aguardando um `T_RIGHT_BRACKET` (`}`) para fechar o escopo, recebe o `EOF`. Isso corretamente dispara um "syntax error".

O ponto de confusão é que o erro é reportado na posição do `EOF`, que, no caso do arquivo de teste, corresponde à coluna final da última linha de comentário.

Embora a gramática inclua uma regra de recuperação para blocos malformados (`T_LEFT_BRACKET stmt_list error`), esta regra específica é projetada para ser ativada por um token inesperado *antes* do fim do arquivo. Ela não é invocada pelo `EOF`.

Este comportamento não é um erro do parser, mas sim o funcionamento esperado. O analisador sintático só pode confirmar que o `}` está ausente quando não há mais tokens para ler (ou seja, no `EOF`). Portanto, o erro é corretamente reportado na posição final da entrada de dados, que o nosso lexer rastreou com precisão.

10.2.2. Quais Erros poderiam ser tratados ou não?

Conforme os requisitos do trabalho, o objetivo foi fornecer mensagens de erro claras e específicas, em vez de depender apenas do "syntax error" genérico do Bison. Para isso, implementamos no `parser.y` cerca de 10 regras de recuperação usando o token `error`.

Essas regras nos permitiram tratar erros comuns de forma precisa, como:

- Ausência de parênteses em comandos: (ex: `T_WHILE T_LEFT_PAREN expr error ...`).
- `else` sem `if`: (ex: `error T_ELSE matched_stmt`).
- Atribuições malformadas: (ex: `T_INT T_ID T_ATRIBUTION error T_SEMICOLON`).

No entanto, durante o desenvolvimento, percebemos que tentar adicionar regras de recuperação para *todos* os casos de erro possíveis era impraticável. A adição de novas regras de `error` frequentemente introduzia novos conflitos de `shift/reduce` no parser, como pode ser verificado no arquivo `parser.output` (por exemplo, no Estado 27).

Esses conflitos ocorrem porque o parser fica em dúvida sobre qual regra de recuperação deve aplicar (ex: "devo recuperar usando o `T_SEMICOLON` ou o `T_RIGHT_BRACKET`?").

Um exemplo claro dessa dificuldade foi o tratamento de chaves de fechamento (`}`) ausentes. Embora tenhamos criado uma regra para detectar um bloco não fechado (`T_LEFT_BRACKET stmt_list error`), sua aplicação é muito específica e conflita com outras regras de recuperação em escopos mais complexos.

Por essa razão, optamos por focar nas 10 regras de erro mais claras e comuns. Para erros mais ambíguos ou complexos (como a maioria dos casos de chaves ausentes), deixamos que o mecanismo de recuperação padrão do Bison (modo de pânico) atuasse, resultando na mensagem genérica, que é então formatada pela nossa função `yyerror` para incluir a linha e coluna.

10.3. Analisador Semântico

10.3.1. Escolha: Incrementar o Parser ou Novos Arquivos?

Para a implementação da Análise Semântica e da Geração de Código Intermediário, enfrentamos a decisão arquitetural de escolher entre **Modularizar o Código** em arquivos C separados ou **Incrementar o Parser** adicionando todas as ações semânticas e o código auxiliar diretamente ao arquivo `parser.y`.

Em decorrência do escopo compacto do **Mini C Didático** e dos prazos do trabalho prático, optamos pela abordagem de **Incrementar o Parser**.

Optamos por incorporar todas as estruturas de dados (como a **Tabela de Símbolos** com escopos aninhados) e as funções de **Verificação de Tipos** diretamente nas seções de código C do `parser.y`.

Essa decisão priorizou a **velocidade de desenvolvimento** e a **facilidade de depuração**, pois permitiu o acesso imediato e direto às variáveis essenciais do parser (`$1`, `$3`, `$$`) e às variáveis globais de rastreamento de posição (`line`, `column`).

Reconhecemos, contudo, que essa abordagem resulta em um arquivo gerado (`parser.tab.c`) **menos modular**. Em um projeto de compilador em larga escala, a separação em módulos externos seria indispensável para manter a clareza e facilitar a manutenção futura, mas foi um trade-off aceitável para o contexto didático do nosso trabalho.

10.3.2. Ajuste da Gramática para Suporte a Blocos Vazios

Durante a validação da Análise Semântica, verificou-se que a gramática original não permitia blocos vazios, o que impedia testar corretamente a criação e remoção de escopos na Tabela de Símbolos. A regra *stmt_list* exigia ao menos um comando, resultando em erro sintático para estruturas como `{ }`.

Para corrigir essa limitação, foi introduzido o não-terminal *optional_stmt_list*, que admite derivação vazia. A definição de blocos foi atualizada para utilizar essa nova regra, permitindo escopos vazios e tornando possível a construção dos casos de teste semânticos necessários.

10.4. Geração de código IR

10.4.1. Conflitos Gramaticais e Estratégia de Linearização no Controle de Fluxo

Durante o desenvolvimento da geração de código para estruturas de controle, a inserção de ações semânticas intermediárias (*mid-rule actions*) para a criação de rótulos causou conflitos severos de *shift/reduce* e *reduce/reduce* no analisador sintático gerado pelo Bison. A estratégia inicial de separar as regras em `matched_stmt` e `open_stmt` para tratar a ambiguidade do *Dangling Else* mostrou-se incompatível com a necessidade de emitir instruções de salto (`GOTO`, `IF_FALSE`) antes de processar os blocos internos. A solução adotada foi a

simplificação da gramática para uma regra única de `stmt`, utilizando diretivas de precedência (`%nonassoc` para `IFX` e `T_ELSE`) e não-terminais auxiliares para linearizar o fluxo e garantir a correta manipulação da pilha de rótulos sem quebrar a lógica de análise LR.

10.4.2. Inconsistências na Parametrização da Função de Emissão de Instruções (emit)

Houve dificuldades na padronização da função `emit()`, projetada inicialmente para suportar instruções padrão de três endereços (operador, dois operandos e um destino). Ao tentar gerar instruções unárias ou que não produzem resultado armazenado em registrador, como o comando `PRINT`, a lógica interna da função omitia o argumento, gerando saídas incompletas (apenas o mnemônico da instrução). O problema residia no mapeamento dos parâmetros: a função esperava que operações unárias tivessem seu argumento no campo de destino (`dest`), mas o parser o enviava como primeiro operando (`arg1`). A correção exigiu o ajuste nas chamadas da função dentro das regras semânticas, garantindo que o valor a ser impresso ou manipulado fosse passado no parâmetro correto para que a formatação da string de saída (TAC) fosse gerada adequadamente.

CONCLUSÃO

Este projeto permitiu a construção prática de um compilador, consolidando os conceitos teóricos estudados. Desenvolvemos um analisador léxico robusto usando Flex, capaz de identificar tokens precisamente e reportar erros com localização exata. Na análise sintática com Bison, resolvemos ambiguidades como o dangling else e o menos unário através de diretivas de precedência e uma gramática bem estruturada.

Superamos desafios significativos: refinamos a contagem de colunas, reorganizamos regras léxicas e transformamos a tabela de símbolos em uma estrutura útil para fases posteriores. Implementamos um sistema de tratamento de erros com mensagens claras, equilibrando a precisão e estabilidade do parser.

O trabalho demonstrou na prática a complexidade envolvida na criação de ferramentas de compilação, proporcionando uma base sólida para etapas futuras como análise semântica e geração de código, e confirmou a importância do projeto cuidadoso em cada fase do processo de compilação.