UNIVERSIDADE FEDERAL DE LAVRAS

Bacharelado em Ciência da Computação - 10A

TRABALHO PRÁTICO DE COMPILADORES

RELATÓRIO 2

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

Lavras
2025

SUMÁRIO

1 INTRODUÇÃO

Esta segunda etapa do projeto GcC mini C compiler teve como objetivo a implementação do analisador sintático, componente responsável por verificar se a sequência de tokens produzida pelo analisador léxico forma estruturas válidas de acordo com a gramática da linguagem. Essa fase representa um avanço natural na construção do compilador, pois introduz o reconhecimento da estrutura hierárquica e lógica do código, permitindo identificar comandos compostos, expressões, condicionais e laços de repetição.

A partir do analisador léxico desenvolvido na primeira etapa, foi possível integrar o uso das ferramentas Flex e Bison, estabelecendo a comunicação entre as duas fases do processo de compilação. Enquanto o Flex é responsável por identificar padrões léxicos e gerar tokens, o Bison utiliza esses tokens como entrada para analisar a forma gramatical do programa. Dessa forma, o compilador passa a ser capaz não apenas de reconhecer palavras válidas, mas também de compreender como elas se organizam para formar instruções completas.

A implementação do analisador sintático possibilitou a detecção de erros estruturais no código, como chaves ou parênteses não balanceados, comandos mal formados e expressões incompletas. Além disso, o projeto consolidou a compreensão de conceitos fundamentais de teoria da computação, como gramáticas livres de contexto, autômatos LR(1), precedência de operadores e recuperação de erros.

Esta etapa foi essencial para aproximar o compilador de uma ferramenta funcional, capaz de interpretar a estrutura de programas escritos na linguagem proposta. A experiência também proporcionou uma visão mais ampla do papel do parser dentro do processo de compilação e da importância de uma gramática bem definida para garantir que a análise semântica e a geração de código ocorram de maneira correta nas próximas fases do projeto.

2 DECISÕES DO PROJETO

Durante o desenvolvimento do analisador sintático, algumas decisões importantes foram tomadas para garantir a corretude, robustez e clareza, especialmente no tratamento de ambiguidades e erros.

Em relação à ambiguidade "Dangling Else", o problema era que a gramática natural para if-then-else é ambígua, gerando um conflito Shift/Reduce, pois o parser não sabe a qual if associar um else em construções aninhadas como if (c1) if (c2) s1 else s2. A solução adotada foi resolver a ambiguidade gramaticalmente, sem depender das diretivas de precedência do Bison (%nonassoc *TK_ELSE*). Para isso, a regra statement foi dividida em duas: *matched_statement* (comandos sintaticamente completos) e *unmatched_statement* (comandos que terminam em um if sem else). Além disso, a regra then (corpo de if ou while) foi definida para aceitar um bloco {} ou apenas um *matched_statement*. Essa estrutura força o else a se associar ao if mais interno (*unmatched_statement*) que ainda não possui um else, implementando a semântica padrão da linguagem C e resultando em uma gramática LR(1)

sem conflitos reportados pelo Bison. Essa abordagem foi escolhida por demonstrar explicitamente a técnica de transformação gramatical.

Associada a essa solução, uma decisão importante foi não permitir blocos {} como statements genéricos, restringindo-os a aparecerem apenas como corpo de if e while (através da regra then), o que foi implementado na divisão matched/unmatched e na regra *while_stmt*, garantindo que o código se assemelhe mais à estrutura esperada em C, onde blocos soltos não são comuns fora de funções (que não existem nesta mini C).

Quanto à precedência e associatividade de expressões, o problema era que expressões aritméticas, lógicas e relacionais possuem diferentes níveis (como * antes de +) e associatividade, incluindo a distinção entre menos unário e binário, e fatorar a gramática manualmente seria complexo e verboso. A solução foi utilizar as diretivas de precedência do Bison (%left, %right), definindo a ordem crescente de precedência e a associatividade para todos os operadores. Para a ambiguidade do menos unário, foi utilizado um pseudo-token UMINUS com alta precedência e a diretiva %prec UMINUS na regra de negação unária (expression -> *TK_MINUS* expression %prec UMINUS), o que simplificou enormemente a regra expression, delegando a resolução de conflitos para o Bison.

No que tange à recuperação de erros sintáticos, o problema era que o Bison, por padrão, para no primeiro erro, enquanto o requisito era detectar múltiplos erros. A solução foi implementar a recuperação em modo pânico na regra statements, adicionando regras com o token especial error que permitem ao parser descartar tokens da entrada até encontrar um ponto de sincronização seguro (*TK_SEMICOLON* ou *TK_RBRACE*). A ação yyerrok é usada após sincronizar com “;” para indicar ao Bison que a recuperação foi bem-sucedida e ele pode continuar a análise. Para mensagens de erro mais claras, foi usado %define parse.error verbose e a saída em yyerror foi formatada, integrando-a à tabela de trace sintático.

Por fim, para o rastreamento (trace) da análise sintática, decidiu-se não usar o yydebug padrão do Bison (cuja saída é verbosa e difícil de formatar) nem construir uma AST, optando-se por criar uma tabela de trace Shift-Reduce personalizada. Na implementação, uma string global (*g_full_trace*) armazena a sequência de ações: a função *print_token* no lexer registra os eventos SHIFT, uma função *add_reduce_trace* é chamada nas ações de cada regra no parser para registrar os eventos REDUCE, e a função yyerror registra os ERROs. No final da análise (na função main), a função *parsing_table* processa essa string e imprime uma tabela formatada, separada da análise léxica.

3 GRAMÁTICA COMPLETA EM BNF

A seguir, apresentamos a gramática livre de contexto implementada para a linguagem mini C, em notação BNF (Backus-Naur Form). Esta gramática foi projetada para ser LR(1) e resolver ambiguidades inerentes.

None

```
<program> ::= <statements>

<statements> ::= /* epsilon */
               | <statements> <statement>

<statement> ::= <matched_statement>
              | <unmatched_statement>

<matched_statement> ::= <declaration>
                      | <assignment>
                      | <read>
                      | <print>
                      | <while_stmt>
                      | TK_IF TK_LPAREN <expression> TK_RPAREN <then> TK_ELSE <then>

<unmatched_statement> ::= TK_IF TK_LPAREN <expression> TK_RPAREN <then>
                          |  TK_IF  TK_LPAREN  <expression>  TK_RPAREN  <then>  TK_ELSE
<unmatched_statement>

<then> ::= TK_LBRACE <statements> TK_RBRACE
         | <matched_statement>

<declaration> ::= <type> <id_list> TK_SEMICOLON

<type> ::= TK_INT
         | TK_BOOL

<id_list> ::= TK_ID
            | <id_list> TK_COMMA TK_ID

<assignment> ::= TK_ID TK_ASSIGN <expression> TK_SEMICOLON

<read> ::= TK_READ TK_LPAREN TK_ID TK_RPAREN TK_SEMICOLON

<print> ::= TK_PRINT TK_LPAREN <expression> TK_RPAREN TK_SEMICOLON

<while_stmt>  ::=  TK_WHILE  TK_LPAREN  <expression>  TK_RPAREN  TK_LBRACE  <statements>
TK_RBRACE
               | TK_WHILE TK_LPAREN <expression> TK_RPAREN <matched_statement>

<expression> ::= TK_INTEGER
               | TK_TRUE
               | TK_FALSE
               | TK_ID
               | TK_LPAREN <expression> TK_RPAREN
               | <expression> TK_PLUS <expression>
               | <expression> TK_MINUS <expression>
               | <expression> TK_MULT <expression>
               | <expression> TK_DIV <expression>
               | <expression> TK_MOD <expression>
                | TK_MINUS <expression> /* Menos Unário, precedência definida via %prec
*/
               | <expression> TK_EQ <expression>
               | <expression> TK_NE <expression>
```

```
                | <expression> TK_LT <expression>
                | <expression> TK_LE <expression>
                | <expression> TK_GT <expression>
                | <expression> TK_GE <expression>
                | <expression> TK_LOGICAL_AND <expression>
                | <expression> TK_LOGICAL_OR <expression>
                | TK_LOGICAL_NOT <expression>
```

4 DIFICULDADES ENCONTRADAS

Durante o desenvolvimento do analisador sintático, a principal dificuldade foi alinhar a saída do lexer (escrita no Flex) com as expectativas do parser (gerado pelo Bison). As inconsistências entre nomes de tokens e a definição de precedência causaram erros de compilação inicialmente, especialmente ao substituir o token genérico TK_RELOP por versões específicas (TK_EQ, TK_LT, etc.), o que exigiu ajustes tanto na gramática quanto nas ações semânticas.

Outra dificuldade relevante foi a implementação da recuperação de erros sintáticos sem comprometer o fluxo de análise. O uso do símbolo error exigiu experimentação com diferentes pontos de inserção e com o comando yyerrok, que permite retomar o parsing após um erro. O desafio estava em evitar loops de erro ou consumo excessivo de tokens, garantindo uma recuperação estável e compreensível.

A resolução do menos unário também exigiu atenção. Sem a diretiva %prec UMINUS, o operador - poderia gerar conflitos de precedência com o operador binário. A solução foi declarar um pseudo-token com precedência alta e associatividade à direita, garantindo que expressões como -a + b fossem interpretadas corretamente.

Por fim, houve o desafio de gerar uma saída visualmente consistente com a da Etapa 1. A integração do trace de parsing e da tabela de símbolos em um formato de tabela colorida exigiu ajustes de formatação e manipulação de strings com strcat e sprintf, além do cuidado com o limite do buffer global.

5 CONJUNTOS FIRST E FOLLOW (NÃO-TERMINAIS DE EXPRESSÕES)

SYMBOL	FIRST	FOLLOW
assignment	TK_ID	$, TK_BOOL, TK_ELSE, TK_ID, TK_IF, TK_INT, TK_PRINT, TK_RBRACE, TK_READ, TK_WHILE, error
declaration	TK_BOOL, TK_INT	$, TK_BOOL, TK_ELSE, TK_ID, TK_IF, TK_INT, TK_PRINT, TK_RBRACE, TK_READ, TK_WHILE, error
expression	**TK_FALSE, TK_ID, TK_INTEGER, TK_LOGICAL_NOT, TK_LPAREN, TK_MINUS, TK_TRUE**	**TK_DIV, TK_EQ, TK_GE, TK_GT, TK_LE, TK_LOGICAL_AND, TK_LOGICAL_OR, TK_LT, TK_MINUS, TK_MOD, TK_MULT, TK_NE, TK_PLUS, TK_RPAREN, TK_SEMICOLON, UMINUS**
id_list	TK_ID	TK_COMMA, TK_SEMICOLON
matched_statement	TK_BOOL, TK_ID, TK_IF, TK_INT, TK_PRINT, TK_READ, TK_WHILE	$, TK_BOOL, TK_ELSE, TK_ID, TK_IF, TK_INT, TK_PRINT, TK_RBRACE, TK_READ, TK_WHILE, error
print	TK_PRINT	$, TK_BOOL, TK_ELSE, TK_ID, TK_IF, TK_INT, TK_PRINT, TK_RBRACE, TK_READ, TK_WHILE, error
program	TK_BOOL, TK_ID, TK_IF, TK_INT, TK_PRINT, TK_READ, TK_WHILE, error, ϵ	$
read	TK_READ	$, TK_BOOL, TK_ELSE, TK_ID, TK_IF, TK_INT, TK_PRINT, TK_RBRACE, TK_READ, TK_WHILE, error
statement	TK_BOOL, TK_ID, TK_IF, TK_INT, TK_PRINT, TK_READ, TK_WHILE	$, TK_BOOL, TK_ID, TK_IF, TK_INT, TK_PRINT, TK_RBRACE, TK_READ, TK_WHILE, error
statements	TK_BOOL, TK_ID, TK_IF, TK_INT, TK_PRINT, TK_READ, TK_WHILE, error, ϵ	$, TK_BOOL, TK_ID, TK_IF, TK_INT, TK_PRINT, TK_RBRACE, TK_READ, TK_WHILE, error
then	TK_BOOL, TK_ID, TK_IF, TK_INT, TK_LBRACE, TK_PRINT, TK_READ, TK_WHILE	$, TK_BOOL, TK_ELSE, TK_ID, TK_IF, TK_INT, TK_PRINT, TK_RBRACE, TK_READ, TK_WHILE, error
type	TK_BOOL, TK_INT	TK_ID
unmatched_statement	TK_IF	$, TK_BOOL, TK_ID, TK_IF, TK_INT, TK_PRINT, TK_RBRACE, TK_READ, TK_WHILE, error
while_stmt	TK_WHILE	$, TK_BOOL, TK_ELSE, TK_ID, TK_IF, TK_INT, TK_PRINT, TK_RBRACE, TK_READ, TK_WHILE, error

6 ANÁLISE DE CONFLITOS LR(0)

Durante o processo de análise do autômato LR(0) gerado pelo Bison, verificou-se que a gramática final, construída com base na separação entre matched_statement e unmatched_statement, não apresenta conflitos de tipos shift/reduce ou reduce/reduce. Essa constatação foi confirmada por meio da inspeção do arquivo parser.output, obtido ao compilar o analisador com a opção -v. O resultado confirmou que a estrutura da gramática foi capaz de eliminar ambiguidades comuns que frequentemente surgem em linguagens que possuem condicionais aninhados e operadores com múltiplos contextos de uso.

Mesmo não havendo conflitos na versão final, é relevante discutir os casos clássicos em que esses problemas aparecem e como o projeto lidou com eles. Um dos conflitos mais conhecidos é o “dangling else”, presente em gramáticas que tratam comandos condicionais de forma direta. Em uma definição ingênua, o analisador se depara com uma ambiguidade ao encontrar uma sequência do tipo if (expr) statement seguida de um else: o parser pode optar por realizar um shift, assumindo que o else pertence ao if mais interno, comportamento adotado pela linguagem C, ou pode tentar reduzir prematuramente, tratando o primeiro if como completo e associando o else ao próximo comando externo. O Bison, por padrão, resolve esse impasse favorecendo o shift, o que garante o comportamento esperado, mas ainda assim gera um aviso de ambiguidade. A solução adotada neste projeto foi estrutural: ao dividir o não-terminal statement em duas versões, matched e unmatched, foi possível eliminar completamente a incerteza. Essa separação impõe, de forma explícita, que um else só possa ser associado a um comando if que já tenha um corpo bem definido, o que evita o conflito e reflete fielmente a semântica desejada.

Outro ponto de potencial conflito está relacionado ao uso do operador unário de negação (-). Em expressões como a * -b, o parser poderia se confundir quanto à precedência

correta, oscilando entre realizar um shift do símbolo - para formar o número negativo ou efetuar uma redução antecipada da multiplicação. Essa ambiguidade é um exemplo clássico de conflito shift/reduce em expressões aritméticas. Para contornar esse problema, a gramática utilizou diretivas de precedência do Bison, especificamente %right UMINUS em conjunto com a marcação %prec UMINUS na regra de expressão unária. Com isso, o analisador foi instruído a reconhecer o - como um operador unário de maior precedência, garantindo a interpretação correta e a eliminação definitiva do conflito.

Assim, as decisões gramaticais tomadas nesta etapa, tanto na estrutura dos condicionais quanto no tratamento de operadores unários, consolidaram uma gramática livre de ambiguidades, preservando a clareza das regras e a precisão da análise sintática. O resultado é um analisador robusto, fiel à semântica da linguagem e livre dos avisos de conflito que normalmente surgem em versões menos refinadas de gramáticas semelhantes.

7 ARQUIVO DE TESTE E SAÍDA

A seguir, incluímos os arquivos de teste utilizados para validar a gramática e a recuperação de erros, juntamente com a saída gerada pelo compilador.

7. 1 ARQUIVO DE TESTE VÁLIDO (TESTE_VALIDO.TXT)

```
C/C++
// Testa as regras essenciais de forma mínima.

int x;
bool y;

read(x);              // Testa read
y = true;             // Testa assignment bool

if (x) {              // Testa if (block)
    print(x);         // Testa print
    while(y)          // Testa while (block) aninhado
        y = false;    // Testa assignment bool, while corpo unico (matched_statement)
} else                // Testa else
    x = 0;            // Testa assignment int (matched_statement)

if(true) if(false) x=1; else x=2; // Testa dangling else com statement unico

// Fim
```

ANÁLISE SINTÁTICA (Shift-Reduce)		
[Lin:Col]	AÇÃO	DETALHE (Token ou Produção)
[001:001]	REDUCE	statements ->
[003:001]	SHIFT	TK_INT_TYPE 'int'
[003:004]	REDUCE	type -> TK_INT
[003:005]	SHIFT	TK_ID 'x'
[003:006]	REDUCE	id_list -> TK_ID
[003:006]	SHIFT	TK_SEMICOLON ';'
[003:007]	REDUCE	declaration -> type id_list TK_SEMICOLON
[003:007]	REDUCE	matched_statement -> declaration
[003:007]	REDUCE	statement -> matched_statement
[003:007]	REDUCE	statements -> statements statement
[004:001]	SHIFT	TK_BOOL_TYPE 'bool'
[004:005]	REDUCE	type -> TK_BOOL
[004:006]	SHIFT	TK_ID 'y'
[004:007]	REDUCE	id_list -> TK_ID
[004:007]	SHIFT	TK_SEMICOLON ';'
[004:008]	REDUCE	declaration -> type id_list TK_SEMICOLON
[004:008]	REDUCE	matched_statement -> declaration
[004:008]	REDUCE	statement -> matched_statement
[004:008]	REDUCE	statements -> statements statement
[006:001]	SHIFT	TK_READ 'read'
[006:005]	SHIFT	TK_LPAREN '('
[006:006]	SHIFT	TK_ID 'x'
[006:007]	SHIFT	TK_RPAREN ')'
[006:008]	SHIFT	TK_SEMICOLON ';'
[006:009]	REDUCE	read -> TK_READ (TK_ID) TK_SEMICOLON
[006:009]	REDUCE	matched_statement -> read
[006:009]	REDUCE	statement -> matched_statement
[006:009]	REDUCE	statements -> statements statement
[007:001]	SHIFT	TK_ID 'y'
[007:003]	SHIFT	TK_ASSIGN '='
[007:005]	SHIFT	TK_TRUE 'true'
[007:009]	REDUCE	expression -> TK_TRUE
[007:009]	SHIFT	TK_SEMICOLON ';'
[007:010]	REDUCE	assignment -> TK_ID TK_ASSIGN expression TK_SEMICOLON
[007:010]	REDUCE	matched_statement -> assignment
[007:010]	REDUCE	statement -> matched_statement
[007:010]	REDUCE	statements -> statements statement
[009:001]	SHIFT	TK_IF 'if'
[009:004]	SHIFT	TK_LPAREN '('
[009:005]	SHIFT	TK_ID 'x'
[009:006]	REDUCE	expression -> TK_ID
[009:006]	SHIFT	TK_RPAREN ')'
[009:008]	SHIFT	TK_LBRACE '{'
[009:009]	REDUCE	statements ->
[010:005]	SHIFT	TK_PRINT 'print'
[010:010]	SHIFT	TK_LPAREN '('
[010:011]	SHIFT	TK_ID 'x'
[010:012]	REDUCE	expression -> TK_ID
[010:012]	SHIFT	TK_RPAREN ')'
[010:013]	SHIFT	TK_SEMICOLON ';'
[010:014]	REDUCE	print -> TK_PRINT (expression) TK_SEMICOLON
[010:014]	REDUCE	matched_statement -> print
[010:014]	REDUCE	statement -> matched_statement
[010:014]	REDUCE	statements -> statements statement
[011:005]	SHIFT	TK_WHILE 'while'
[011:010]	SHIFT	TK_LPAREN '('
[011:011]	SHIFT	TK_ID 'y'
[011:012]	REDUCE	expression -> TK_ID
[011:012]	SHIFT	TK_RPAREN ')'
[012:009]	SHIFT	TK_ID 'y'
[012:011]	SHIFT	TK_ASSIGN '='
[012:013]	SHIFT	TK_FALSE 'false'
[012:018]	REDUCE	expression -> TK_FALSE

[012:018]	REDUCE	expression -> TK_FALSE
[012:018]	SHIFT	TK_SEMICOLON ';'
[012:019]	REDUCE	assignment -> TK_ID TK_ASSIGN expression TK_SEMICOLON
[012:019]	REDUCE	matched_statement -> assignment
[012:019]	REDUCE	while_stmt -> WHILE (expr) matched_statement
[012:019]	REDUCE	matched_statement -> while_stmt
[012:019]	REDUCE	statement -> matched_statement
[012:019]	REDUCE	statements -> statements statement
[013:001]	SHIFT	TK_RBRACE '}'
[013:002]	REDUCE	then -> { statements }
[013:003]	SHIFT	TK_ELSE 'else'
[014:005]	SHIFT	TK_ID 'x'
[014:007]	SHIFT	TK_ASSIGN '='
[014:009]	SHIFT	TK_INTEGER '0'
[014:010]	REDUCE	expression -> TK_INTEGER
[014:010]	SHIFT	TK_SEMICOLON ';'
[014:011]	REDUCE	assignment -> TK_ID TK_ASSIGN expression TK_SEMICOLON
[014:011]	REDUCE	matched_statement -> assignment
[014:011]	REDUCE	then -> matched_statement
[014:011]	REDUCE	matched_statement -> IF (expr) then ELSE then
[014:011]	REDUCE	statement -> matched_statement
[014:011]	REDUCE	statements -> statements statement
[016:001]	SHIFT	TK_IF 'if'
[016:003]	SHIFT	TK_LPAREN '('
[016:004]	SHIFT	TK_TRUE 'true'
[016:008]	REDUCE	expression -> TK_TRUE
[016:008]	SHIFT	TK_RPAREN ')'
[016:010]	SHIFT	TK_IF 'if'
[016:012]	SHIFT	TK_LPAREN '('
[016:013]	SHIFT	TK_FALSE 'false'
[016:018]	REDUCE	expression -> TK_FALSE
[016:018]	SHIFT	TK_RPAREN ')'
[016:020]	SHIFT	TK_ID 'x'
[016:021]	SHIFT	TK_ASSIGN '='
[016:022]	SHIFT	TK_INTEGER '1'
[016:023]	REDUCE	expression -> TK_INTEGER
[016:023]	SHIFT	TK_SEMICOLON ';'
[016:024]	REDUCE	assignment -> TK_ID TK_ASSIGN expression TK_SEMICOLON
[016:024]	REDUCE	matched_statement -> assignment
[016:024]	REDUCE	then -> matched_statement
[016:025]	SHIFT	TK_ELSE 'else'
[016:030]	SHIFT	TK_ID 'x'
[016:031]	SHIFT	TK_ASSIGN '='
[016:032]	SHIFT	TK_INTEGER '2'
[016:033]	REDUCE	expression -> TK_INTEGER
[016:033]	SHIFT	TK_SEMICOLON ';'
[016:034]	REDUCE	assignment -> TK_ID TK_ASSIGN expression TK_SEMICOLON
[016:034]	REDUCE	matched_statement -> assignment
[016:034]	REDUCE	then -> matched_statement
[016:034]	REDUCE	matched_statement -> IF (expr) then ELSE then
[016:034]	REDUCE	then -> matched_statement
[018:007]	REDUCE	unmatched_statement -> IF (expr) then
[018:007]	REDUCE	statement -> unmatched_statement
[018:007]	REDUCE	statements -> statements statement
[018:007]	REDUCE	program -> statements

Análise Sintática concluída com sucesso!

TABELA DE SÍMBOLOS

[ID]	[Lin:Col]	LEXEMA	TIPO
[001]	[003:005]	x	TK_ID
[002]	[004:006]	y	TK_ID

Total de símbolos:2

7.2 ARQUIVO DE TESTE INVÁLIDO (TESTE_INVALIDO.TXT)

```C/C++
// Testa vários erros sintáticos e recuperação de forma mínima.

int a b = 5;        // Erro 1 (esperado ',' ou ';', encontrou 'b')

read(a+);           // Erro 2 (esperado ')', encontrou '+')

if (a > ) print(a;  // Erro 3 (esperado expr após '>')

while (true {       // Erro 4 (inesperado '{')
    a = ;           // Recuperção
    } else          // Erro 5 (inesperado '}')
    b = 1

int z               // Não houve recuperação pois não encontrou ',' nem '}'
```

ANÁLISE SINTÁTICA (Shift-Reduce)

[Lin:Col]	AÇÃO	DETALHE (Token ou Produção)
[001:001]	REDUCE	statements ->
[003:001]	SHIFT	TK_INT_TYPE 'int'
[003:004]	REDUCE	type -> TK_INT
[003:005]	SHIFT	TK_ID 'a'
[003:006]	REDUCE	id_list -> TK_ID
[003:007]	SHIFT	TK_ID 'b'
[003:008]	ERRO	unexpected TK_ID, expecting TK_SEMICOLON or TK_COMMA
[003:009]	SHIFT	TK_ASSIGN '='
[003:011]	SHIFT	TK_INTEGER '5'
[003:012]	SHIFT	TK_SEMICOLON ';'
[003:013]	REDUCE	statements -> statements error ; (Recuperacao)
[005:001]	SHIFT	TK_READ 'read'
[005:005]	SHIFT	TK_LPAREN '('
[005:006]	SHIFT	TK_ID 'a'
[005:007]	SHIFT	TK_PLUS '+'
[005:008]	ERRO	unexpected TK_PLUS, expecting TK_RPAREN
[005:008]	SHIFT	TK_RPAREN ')'
[005:009]	SHIFT	TK_SEMICOLON ';'
[005:010]	REDUCE	statements -> statements error ; (Recuperacao)
[007:001]	SHIFT	TK_IF 'if'
[007:004]	SHIFT	TK_LPAREN '('
[007:005]	SHIFT	TK_ID 'a'
[007:006]	REDUCE	expression -> TK_ID
[007:007]	SHIFT	TK_GT '>'
[007:009]	SHIFT	TK_RPAREN ')'
[007:010]	ERRO	unexpected TK_RPAREN
[007:011]	SHIFT	TK_PRINT 'print'
[007:016]	SHIFT	TK_LPAREN '('
[007:017]	SHIFT	TK_ID 'a'
[007:018]	SHIFT	TK_SEMICOLON ';'
[007:019]	REDUCE	statements -> statements error ; (Recuperacao)
[009:001]	SHIFT	TK_WHILE 'while'
[009:007]	SHIFT	TK_LPAREN '('
[009:008]	SHIFT	TK_TRUE 'true'
[009:012]	REDUCE	expression -> TK_TRUE
[009:013]	SHIFT	TK_LBRACE '{'
[009:014]	ERRO	unexpected TK_LBRACE
[010:005]	SHIFT	TK_ID 'a'
[010:007]	SHIFT	TK_ASSIGN '='
[010:009]	SHIFT	TK_SEMICOLON ';'
[010:010]	REDUCE	statements -> statements error ; (Recuperacao)
[011:005]	SHIFT	TK_RBRACE '}'
[011:006]	ERRO	unexpected TK_RBRACE
[011:006]	REDUCE	statements -> statements error } (Recuperacao Fim Bloco)
[011:007]	SHIFT	TK_ELSE 'else'
[012:005]	SHIFT	TK_ID 'b'
[012:007]	SHIFT	TK_ASSIGN '='
[012:009]	SHIFT	TK_INTEGER '1'
[014:001]	SHIFT	TK_INT_TYPE 'int'
[014:005]	SHIFT	TK_ID 'z'

Análise Sintática concluída com 005 erro(s).

TABELA DE SÍMBOLOS

[ID]	[Lin:Col]	LEXEMA	TIPO
[001]	[003:005]	a	TK_ID
[002]	[003:007]	b	TK_ID
[003]	[014:005]	z	TK_ID

Total de símbolos:3

8 CONCLUSÃO

A segunda etapa do projeto representou um avanço significativo na construção do compilador GcC, consolidando a integração entre análise léxica e sintática. O trabalho evidenciou a importância da definição de gramáticas bem estruturadas, da resolução de ambiguidades clássicas e da implementação de estratégias eficazes de recuperação de erros.

A ferramenta resultante é capaz de reconhecer programas completos, emitir diagnósticos precisos e oferecer uma representação detalhada do processo de análise sintática. A experiência proporcionou uma compreensão aprofundada dos mecanismos internos de parsing, consolidando os fundamentos teóricos sobre autômatos LR, precedência de operadores e hierarquia gramatical, e preparando o terreno para as próximas etapas do compilador, que tratarão da análise semântica e da geração de código.