UNIVERSIDADE FEDERAL DE LAVRAS

RELATÓRIO DE COMPILADORES

Professor: Ricardo Terra Nunes Bueno Villela
Alunos: Estevão Augusto da Fonseca Santos
Felipe Crisóstomo Silva Oliveira
Bernardo Coelho Pavani Marinho

1. INTRODUÇÃO

O desenvolvimento de compiladores e interpretadores é um processo multifacetado e essencial para a tradução de linguagens de programação. Neste contexto, a análise léxica e a análise sintática são fases primordiais e interligadas. A análise léxica, como etapa inicial, escaneia o código-fonte para identificar e categorizar seus elementos, transformando cadeias de caracteres em uma sequência de tokens significativos.

Este trabalho detalha a implementação de um analisador léxico com a ferramenta Flex (Fast Lexical Analyzer Generator) e, subsequentemente, a construção de um analisador sintático utilizando o Bison. Serão exploradas as funcionalidades de ambas as ferramentas, as estratégias de implementação, os testes realizados e os resultados obtidos em cada fase da construção.

2. REFERENCIAL TEÓRICO

2.1 ANALISADOR LÉXICO

A análise léxica é a primeira fase da compilação e tem como função principal escanear o código fonte e dividi-lo em unidades léxicas (tokens), que são passadas à etapa de análise sintática. Um token pode representar palavras-chave, identificadores, operadores, literais, entre outros.

O *Flex* é uma ferramenta para geração de analisadores léxicos baseada em expressões regulares. Ele gera código em C para reconhecer padrões definidos pelo programador. Os padrões são especificados em uma linguagem declarativa dividida em três seções: definição, regras e código do usuário.

```
Unset
%% // separa as seções

<expressão regular>  <ação>

%%

int main() {
       yylex();
       return 0;
}
```

2.2 ANALISADOR SINTÁTICO

A análise sintática é a segunda fase do processo de compilação. Sua principal função é verificar se a sequência de tokens, gerada pela análise léxica, está de acordo com a estrutura gramatical da linguagem de programação, definida por uma gramática livre de contexto (GLC).

Para essa etapa, é comum utilizar ferramentas como o *Bison*, que, assim como o *Flex*, gera código em C. O Bison implementa analisadores sintáticos baseados na técnica LALR(1) (Look-Ahead LR com uma unidade de lookahead). Ele lê uma gramática definida pelo programador e gera um parser capaz de construir a árvore sintática do código fonte.

A gramática no Bison é definida em termos de regras de produção, que especificam como os símbolos não-terminais podem ser compostos por terminais (tokens) e outros não-terminais. Cada regra pode estar associada a uma ação em C, que é executada quando a regra é reconhecida.

3. DESCRIÇÃO DO TRABALHO E ESTRATÉGIAS DE SOLUÇÃO

Neste projeto, desenvolvemos um analisador léxico e sintático para a linguagem de programação C- (subconjunto de C), contendo os seguintes elementos léxicos:

- Tipos de dados: inteiro, real, caractere, arranjo e registro.
- Funções: recursão, parâmetros passados por valor.
- Comandos: Atribuição, if/else, while, E/S simples (tratados como funções).
- Comentários: texto entre /* e */ (sem comentários aninhados).
- Palavras reservadas: int, oat, struct, if, else, while, void, return (caixa baixa)

E os seguintes elementos léxicos:

- Símbolo Inicial: <programa>
- Declarações: de variáveis simples, vetores e registros (struct), bem como declarações de funções.
- Parâmetros de funções: passados por valor, permitindo múltiplos parâmetros separados por vírgula.
- Blocos de comandos: compostos por declarações locais e sequências de comandos, incluindo comandos vazios (ε-produções).
- Comandos: atribuição, chamada de função, comandos condicionais (if/else), laços (while), comandos de retorno (return) e comandos de E/S simulados.
- Expressões: expressões aritméticas, relacionais, lógicas, bem como acesso a variáveis, vetores e membros de registros.

3.1 EXPRESSÕES REGULARES UTILIZADAS

```
Unset

letra                          [a-z]
digito                         [0-9]
ident
{letra}({letra}|{digito}|_({letra}|{digito}))*
num_int                        {digito}+
```

```
num                             [+-]?[0-9]+(\.[0-9]+)?([E][+-]?[0-9]+)?

abre_chave                      \{
fecha_chave                     \}
abre_colchete                   \[
fecha_colchete                  \]
abre_parenteses                 \(
fecha_parenteses                \)

delim                           [ \t]
ws                              {delim}+
comment                         "/*"([^*]|\*+[^/])*\*+"/"
char_literal                    \'([^\\\n]|\\.)\'
other                           .

soma                            \+|\-
multi                           \/|\*

\"([^\\\"\n]|\\.)*\"            /* CONSTSTRING */

/* Erros Léxicos */

[0-9]+[a-zA-Z_]+        /* Lexical Error: identificador inválido iniciado
com número */
[\u201C\u201D]          /* Lexical Error: uso de aspas inválidas (aspas
curvas) */
\'([^\\\n]|\\.){2,}\'   /* Lexical Error: constante char com mais de um
caractere */
\"([^\\\"\n]|\\.)*      /* Lexical Error: string não fechada */
.                       /* Lexical Error: símbolo inválido
```

3.2 GRAMÁTICAS FORMAIS UTILIZADAS

Unset

```
/*----- 1° -----*/
programa
    : declaracao_lista
    { printf("Redução: programa -> declaracao_lista\n"); }
    ;

/*----- 2° -----*/
declaracao_lista
```

```
    : declaracao
    { printf("Redução: declaracao_lista -> declaracao\n"); }
    | declaracao_lista declaracao
    { printf("Redução: declaracao_lista -> declaracao_lista declaracao\n");
}
    ;

/*----- 3° -----*/
declaracao
    : func_declaracao
    { printf("Redução: declaracao -> func_declaracao\n"); }
    | var_declaracao
    { printf("Redução: declaracao -> var_declaracao\n"); }
    ;

/*----- 4° -----*/
var_declaracao
    : tipo_especificador IDENTIFIER SEMICOLON
    { printf("Redução: var_declaracao -> tipo_especificador IDENTIFIER
SEMICOLON\n"); }
    | tipo_especificador IDENTIFIER LEFT_BRACKET CONSTINT RIGHT_BRACKET
arrayDimensao SEMICOLON
    | STRUCT IDENTIFIER LEFT_BRACE varDeclList RIGHT_BRACE
    { printf("Redução: var_declaracao -> tipo_especificador IDENTIFIER
LEFT_BRACKET CONSTINT RIGHT_BRACKET arrayDimensao SEMICOLON\n"); }
    | tipo_especificador error SEMICOLON
    { printf("ERRO: declaração de variavel invalida na linha %d, coluna
%d\n", line_number, column_number); yyerrok; }
    | tipo_especificador IDENTIFIER LEFT_BRACKET error RIGHT_BRACKET
SEMICOLON
    { printf("ERRO: valor invalido ou ausente para o tamanho do vetor na
linha %d, coluna %d\n", line_number, column_number); yyerrok; }
    ;

arrayDimensao
    : LEFT_BRACKET CONSTINT RIGHT_BRACKET arrayDimensao
    { printf("Redução: arrayDimensao -> LEFT_BRACKET CONSTINT RIGHT_BRACKET
arrayDimensao\n"); }
    | /* vazio */
    { printf("Redução: arrayDimensao -> ε\n"); }
    ;

/*----- 5° -----*/
tipo_especificador
    : INT
    { printf("Redução: tipo_especificador -> INT\n"); }
    | FLOAT
    { printf("Redução: tipo_especificador -> FLOAT\n"); }
```

```
    | CHAR
    { printf("Redução: tipo_especificador -> CHAR\n"); }
    | VOID
    { printf("Redução: tipo_especificador -> VOID\n"); }
    ;

/*----- 6°: sequência de declarações de variáveis -----*/
varDeclList
    : var_declaracao
    { printf("Redução: varDeclList -> var_declaracao\n"); }
    | var_declaracao varDeclList
    { printf("Redução: varDeclList -> var_declaracao varDeclList\n"); }
    ;

/*----- 7° -----*/
func_declaracao
    : tipo_especificador IDENTIFIER LEFT_PAREN params RIGHT_PAREN
composto_decl
    { printf("Redução: func_declaracao -> tipo_especificador IDENTIFIER
LEFT_PAREN params RIGHT_PAREN composto_decl\n"); }
    | tipo_especificador error LEFT_PAREN params RIGHT_PAREN composto_decl
    { printf("ERRO: Função inexistente ou invalida apos o tipo de retorno na
linha %d, coluna %d\n", line_number, column_number); yyerrok; }
    | tipo_especificador IDENTIFIER LEFT_PAREN error RIGHT_PAREN
composto_decl
    { printf("ERRO: Lista de parâmetros malformada na declaração de função
na linha %d, coluna %d\n", line_number, column_number); yyerrok; }
    | error '\n' { printf("Redução: Erro na reducao para
func_declaracao\n"); yyerrok;}
    ;

/*----- 8° -----*/
params
    : params_lista
    { printf("Redução: params -> params_lista\n"); }
    | VOID
    { printf("Redução: params -> VOID\n"); }
    ;

/*----- 9° -----*/
params_lista
    : param
    { printf("Redução: params_lista -> param\n"); }
    | params_lista COMMA param
    { printf("Redução: params_lista -> params_lista COMMA param\n"); }
    ;

/*----- 10° -----*/
```

```
param
    : tipo_especificador IDENTIFIER
    { printf("Redução: param -> tipo_especificador IDENTIFIER\n"); }
    | tipo_especificador IDENTIFIER LEFT_BRACKET RIGHT_BRACKET
    { printf("Redução: param -> tipo_especificador IDENTIFIER LEFT_BRACKET
RIGHT_BRACKET\n"); }
    ;

/*----- 11° -----*/
composto_decl
    : LEFT_BRACE local_declaracoes comando_lista RIGHT_BRACE
    { printf("Redução: composto_decl -> LEFT_BRACE local_declaracoes
comando_lista RIGHT_BRACE\n"); }
    ;

/*----- 12° -----*/
local_declaracoes
    : local_declaracoes var_declaracao
    { printf("Redução: local_declaracoes -> local_declaracoes
var_declaracao\n"); }
    | /* vazio */
    { printf("Redução: local_declaracoes -> ε\n"); }
    ;

/*----- 13° -----*/
comando_lista
    : comando_lista comando
    { printf("Redução: comando_lista -> comando_lista comando\n"); }
    | /* vazio */
    { printf("Redução: comando_lista -> ε\n"); }
    ;

/*----- 14° -----*/
comando
    : expressao_decl
    { printf("Redução: comando -> expressao_decl\n"); }
    | composto_decl
    { printf("Redução: comando -> composto_decl\n"); }
    | selecao_decl
    { printf("Redução: comando -> selecao_decl\n"); }
    | iteracao_decl
    { printf("Redução: comando -> iteracao_decl\n"); }
    | retorno_decl
    { printf("Redução: comando -> retorno_decl\n"); }
    | error SEMICOLON
    { printf("ERRO: Comando invalido sintaticamente ou incompleto na linha
%d, coluna %d\n", line_number, column_number); yyerrok; }
    ;
```

```
/*----- 15° -----*/
expressao_decl
    : expressao SEMICOLON
    { printf("Redução: expressao_decl -> expressao SEMICOLON\n"); }
    | SEMICOLON
    { printf("Redução: expressao_decl -> SEMICOLON\n"); }
    ;

/*----- 16° -----*/
selecao_decl
    : IF LEFT_PAREN expressao RIGHT_PAREN comando
    { printf("Redução: selecao_decl -> IF LEFT_PAREN expressao RIGHT_PAREN
comando\n"); }
    | IF LEFT_PAREN expressao RIGHT_PAREN comando ELSE comando
    { printf("Redução: selecao_decl -> IF LEFT_PAREN expressao RIGHT_PAREN
comando ELSE comando\n"); }
    | IF LEFT_PAREN error RIGHT_PAREN comando
    { printf("ERRO: Condição invalida no comando IF na linha %d, coluna
%d\n", line_number, column_number); yyerrok; }
    ;

/*----- 17° -----*/
iteracao_decl
    : WHILE LEFT_PAREN expressao RIGHT_PAREN comando
    { printf("Redução: iteracao_decl -> WHILE LEFT_PAREN expressao
RIGHT_PAREN comando\n"); }
    | WHILE LEFT_PAREN error RIGHT_PAREN comando
            { printf("ERRO: Comando WHILE inválido na linha %d, coluna
%d\n", line_number, column_number); yyerrok; }
            ;
    ;

/*----- 18° -----*/
retorno_decl
    : RETURN SEMICOLON
    { printf("Redução: retorno_decl -> RETURN SEMICOLON\n"); }
    | RETURN expressao SEMICOLON
    { printf("Redução: retorno_decl -> RETURN expressao SEMICOLON\n"); }
    | RETURN error SEMICOLON
    { printf("ERRO: Retorno inválido na linha %d, coluna %d\n", line_number,
column_number); yyerrok; }
    ;

/*----- 19° -----*/
expressao
    : var ASSIGN_OP expressao
    { printf("Redução: expressao -> var ASSIGN_OP expressao\n"); }
```

```
    | expressao_simples
    { printf("Redução: expressao -> expressao_simples\n"); }
    ;

/*----- 20° -----*/
expressao_simples
    : expressao_soma relacional expressao_soma
    { printf("Redução: expressao_simples -> expressao_soma relacional
expressao_soma\n"); }
    | expressao_soma
    { printf("Redução: expressao_simples -> expressao_soma\n"); }
    ;

/*----- 21° -----*/
relacional
    : LEFT_OP
    { printf("Redução: relacional -> LEFT_OP\n"); }
    | RIGHT_OP
    { printf("Redução: relacional -> RIGHT_OP\n"); }
    | LESS_EQUAL_OP
    { printf("Redução: relacional -> LESS_EQUAL_OP\n"); }
    | RIGHT_EQUAL_OP
    { printf("Redução: relacional -> RIGHT_EQUAL_OP\n"); }
    | EQUAL_OP
    { printf("Redução: relacional -> EQUAL_OP\n"); }
    | NOT_EQUAL_OP
    { printf("Redução: relacional -> NOT_EQUAL_OP\n"); }
    ;

/*----- 22° -----*/
expressao_soma
    : expressao_soma PLUS termo
    { printf("Redução: expressao_soma -> expressao_soma PLUS termo\n"); $$ =
$1 + $3;}
    | termo
    { printf("Redução: expressao_soma -> termo\n"); }
    ;

/*----- 23° -----*/
termo
    : termo MULTIPLY fator
    { printf("Redução: termo -> termo MULTIPLY fator\n"); }
    | fator
    { printf("Redução: termo -> fator\n"); }
    ;

/*----- 24° -----*/
fator
```

```
    : LEFT_PAREN expressao RIGHT_PAREN
    { printf("Redução: fator -> LEFT_PAREN expressao RIGHT_PAREN\n"); $$ =
$2;}
    | var
    { printf("Redução: fator -> var\n"); }
    | ativacao
    { printf("Redução: fator -> ativacao\n"); }
    | CONSTINT
    { printf("Redução: fator -> CONSTINT\n"); }
    | CONSTFLOAT
    { printf("Redução: fator -> CONSTFLOAT\n"); }
    | CONSTCHAR
    { printf("Redução: fator -> CONSTCHAR\n"); }
    | CONSTSTRING
    { printf("Redução: fator -> CONSTSTRING\n"); }
    | '-' fator %prec UMINUS
    { printf("Redução: fator -> - fator\n"); }
    ;

/*----- 25° -----*/
ativacao
    : IDENTIFIER LEFT_PAREN args RIGHT_PAREN
    { printf("Redução: ativacao -> IDENTIFIER LEFT_PAREN args
RIGHT_PAREN\n"); }
    ;

/*----- 26° -----*/
args
    : arg_lista
    { printf("Redução: args -> arg_lista\n"); }
    | /* vazio */
    { printf("Redução: args -> ε\n"); }
    ;

/*----- 27° -----*/
arg_lista
    : arg_lista COMMA expressao
    { printf("Redução: arg_lista -> arg_lista COMMA expressao\n"); }
    | expressao
    { printf("Redução: arg_lista -> expressao\n"); }
    ;

/*----- 28° -----*/
var
    : IDENTIFIER
    { printf("Redução: var -> IDENTIFIER\n"); }
    | IDENTIFIER LEFT_BRACKET expressao RIGHT_BRACKET indice
```

```
    { printf("Redução: var -> IDENTIFIER LEFT_BRACKET expressao
RIGHT_BRACKET indice\n"); $$ = $1;}
    ;

indice
    : indice LEFT_BRACKET expressao RIGHT_BRACKET
    { printf("Redução: indice -> indice LEFT_BRACKET expressao
RIGHT_BRACKET\n"); }
    | /* vazio */
    { printf("Redução: indice -> ε\n"); }
    ;
```

4. TESTES EXECUTADOS E RESULTADOS OBTIDOS

4.1 Entrada: int main() { return 0; }

Resultado obtido:

1(1): int (KEYWORD)

1(4): main (IDENTIFIER)

1(8): ((DELIMITER)

1(9):) (DELIMITER)

1(10): { (DELIMITER)

1(11): return (KEYWORD)

1(17): 0 (CONSTINT)

1(18): ; (DELIMITER)

1(19): } (DELIMITER)

4.2 Entrada: char []txt = “texto”;

Resultado obtido:

1(1): char (KEYWORD)

1(5): [(DELIMITER)

1(6):] (DELIMITER)

1(7): txt (IDENTIFIER)

1(10): = (RELOP)

1(11): “texto” (CONSTSTRING)

1(18): ; (DELIMITER)

4.3 Entrada: float quadrado(float x){ return x*x; }

Resultado obtido:

1(1): float (KEYWORD)

1(6): quadrado (IDENTIFIER)

1(14): ((DELIMITER)

1(15): float (KEYWORD)

1(20): x (IDENTIFIER)

1(21):) (DELIMITER)

1(22): { (DELIMITER)

1(23): return (KEYWORD)

1(29): x (IDENTIFIER)

1(30): * (ARITHOP)

1(31): x (IDENTIFIER)

1(32): ; (DELIMITER)

1(33): } (DELIMITER)

4.4 Entrada: int res = 2*a - 9;

Resultado obtido:

1(1): int (KEYWORD)

1(4): res (IDENTIFIER)

1(7): = (RELOP)

1(8): 2 (CONSTINT)

1(9): * (ARITHOP)

1(10): a (IDENTIFIER)

1(11): - (ARITHOP)

1(12): 9 (CONSTINT)
1(13): ; (DELIMITER)

5. CONCLUSÃO

A implementação de um analisador léxico com Flex demonstrou ser eficiente e prática para identificar estruturas léxicas em uma linguagem de programação simples. A clareza das expressões regulares e a flexibilidade do Flex permitem adaptações rápidas para novas linguagens. A ferramenta provou ser útil tanto em contextos acadêmicos quanto como base para compiladores reais.

A integração com o Bison ampliou ainda mais o alcance do projeto, permitindo a construção de uma análise sintática robusta e bem estruturada. Com o Bison, é possível definir regras gramaticais formais, que verificam se a sequência de tokens gerada pelo Flex segue corretamente a sintaxe da linguagem. O analisador sintático pode validar comandos, estruturas de controle, declarações e expressões, além de detectar erros de forma precisa, apresentando mensagens claras com a linha e coluna do problema.

A combinação do Flex com o Bison permite não apenas reconhecer os elementos básicos da linguagem, mas também validar sua organização estrutural, sendo uma etapa fundamental na construção de compiladores e interpretadores. Essa abordagem modular facilita a manutenção e a escalabilidade do projeto, permitindo que, futuramente, sejam adicionadas etapas como análise semântica, geração de código intermediário e otimizações. Assim, a utilização conjunta dessas ferramentas se mostra extremamente eficiente tanto no meio acadêmico quanto como base para o desenvolvimento de compiladores completos e profissionais.

6. REFERÊNCIAS BIBLIOGRÁFICAS.

Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2006). *Compiladores: Princípios, Técnicas e Ferramentas* (2ª ed.). Pearson.

Levine, J. R., Mason, T., & Brown, D. (1992). *Lex & Yacc*. O'Reilly Media.

Projeto GNU. *Flex - The Fast Lexical Analyzer*. Disponível em: https://www.gnu.org/software/flex/

GNU Project. (n.d.). *GNU Bison*. Obtido em https://www.gnu.org/software/bison/manual/