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 processo de análise léxica é uma das etapas fundamentais na construção de compiladores e interpretadores. Seu principal objetivo é identificar e classificar os símbolos válidos em uma linguagem de programação, transformando uma sequência de caracteres em uma sequência de tokens significativos. Este trabalho apresenta a implementação de um analisador léxico utilizando a ferramenta *Flex* (Fast Lexical Analyzer Generator) e de um analisador sintático utilizando a ferramenta Bison, demonstrando suas funcionalidades, estratégias de implementação, testes realizados e os resultados obtidos.

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
programa                        {declaracao_lista}
declaracao_lista                {declaracao}+
declaracao                      {var_declaracao}|{func_declaracao}
var_declaracao
{tipo_especificador}{ident};|{tipo_especificador}{ident}({abre_colchete}{num
_int}{fecha_colchete})+
```

```
tipo_especificador
(int|float|char|void|struct{ident}{abre_chave}{atributos_declaracao}{fecha_c
have})
atributos_declaracao             {var_declaracao}+
func_declaracao
{tipo_especificador}{ident}\({params}\){composto_decl}
params                           {params_lista}|void
params_lista                     {param}(,{param})*
param
({tipo_especificador}{ident}|{tipo_especificador}{ident}{abre_colchete}{fech
a_colchete})
composto_decl
{abre_chave}{local_declaracoes}{comando_lista}{fecha_chave}
local_declaracoes                {var_declaracao}*
comando_lista                    {comando}*
comando
{expressao_decl}|{composto_decl}|{selecao_decl}|{iteracao_decl}|{retorno_dec
l}
expressao_decl                   {expressao};|;
selecao_decl
(if\({expressao}\){comando}|if\({expressao}\){comando}else{comando})
iteracao_decl                    while\({expressao}\){comando}
retorno_decl                     return;|return{expressao};
expressao                        ({var}={expressao}|{expressao_simples})
var
{ident}|{ident}({abre_colchete}{expressao}{fecha_colchete})+
expressao_simples
{expressao_soma}{relacional}{expressao_soma}|{expressao_soma}
relacional                       <=|<|>=|>|==|!=
expressao_soma                   {termo}({soma}{termo})*
soma                             \+|\-
termo                            {fator}({multi}{fator})*
multi                            \/|\*
fator
\({expressao}\)|{var}|{ativacao}|{num}|{num_int}
ativacao                         {ident}\({args}\)
args                             {arg_lista}?
arg_lista                        {expressao}(,{expressao})*
num                              [+-]?[0-9]+(\.[0-9]+)?([E][+-]?[0-9]+)?
num_int                          {digito}+
digito                           [0-9]
ident                            {letra}({letra}|{digito})*
letra                            [a-z]
abre_chave                       \{
fecha_chave                      \}
abre_colchete                    \[
fecha_colchete                   \]
other                            .
```

```
delim                           [ \t\n]
ws                              {delim}+
comment                         "/*"([^*]|\*+[^/])*\*+"/"
```

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.

Além disso, a integração do analisador léxico com um analisador sintático desenvolvido utilizando o **Bison** tornou possível a construção de uma análise mais robusta e estruturada. O Bison permite definir as regras gramaticais da linguagem, identificando se a sequência de tokens reconhecida pelo Flex obedece à sintaxe da linguagem proposta. Por meio da utilização de uma gramática formal, o analisador sintático é capaz de validar comandos, estruturas de controle, declarações e expressões, identificando erros sintáticos de forma precisa e com mensagens detalhadas que incluem informações como linha e coluna do erro.

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/