Relatório Técnico – Analisador Léxico e Sintático

1. **Integrantes da Equipe**
   A) Gustavo Neves
   B) Luiz Fernando Luth Junior

2. **Nome da Linguagem**

   MinicLang — Uma linguagem fictícia de inspiração C-like, usada para fins didáticos no desenvolvimento de um analisador léxico e sintático.

3. **Descrição da Linguagem:**

   A linguagem *MinicLang* suporta:

   - Tipos primitivos: int, float, double, char, void, bool
   - Controle de fluxo: if, else, for, break, while
   - Diretivas: import
   - Operadores: Aritméticos, relacionais, lógicos, atribuição
   - Símbolos: Pontuação e agrupadores

4. **Funcionamento do Software**

   A) Como executar o Software

   - Instale Lex/Yacc(Ferramenta Flex/Bison);
   - Instale um compilador C, como GCC;
   - Adicione as pastas bin para as variáveis de ambiente
     Codigos:
   - Acessar as pastas do projeto
   - Flex miniclang.l
   - Bison -d miniclang.y
   - Gcc utils/arvore.c utils/tokens.c lex.yy.c miniclang.tab.c -o compilador
   - ./compilador.exe
     Menu Interativo:
   - 1 – Inserir arquivo de teste
   - 2 – Mostrar arvore sintática
   - 3 – Mostrar tabela de símbolos
   - 4 – Palavras reservadas
   - 5 – Sair

B) Exemplo de uso:

Prepare um arquivo exemplo1.txt.

Escolha a opção 1 e insira o nome do arquivo.

Se o arquivo não contiver erros, a análise é finalizada e é possível acessar a árvore sintática através da opção 2, e essa árvore será salva em outro arquivo .txt.

Caso haja erros no arquivo, tanto erros léxicos quanto sintáticos, os erros serão apontados, identificando as linhas e colunas de cada erro.

5. Expressões Regulares Utilizadas

Token	**Expressão Regular**
LETRA	[a-zA-Z]
DIGITO	[0-9]
IDENTIFICADOR	{LETRA}({LETRA}|{DIGITO}|"_")*
NUMERO_REAL	{DIGITO}+"."{DIGITO}+
NUMERO_INTEIRO	[-+]?{DIGITO}+
OPER_ARIT	[\+\-*\/%]
OPER_LOG	(||||CC|!)
OPER_REL	(>=|<=|==|!=|>|<)
ATRIBUICAO	=
PONTUACAO	[;\(\)\{\}\[\]\,]
LOOP	(for|while)
CONDICIONAL	(if|else)
STRING	\"([^\\\"\n]|\\.)*\"
COMENTARIO_LINHA	"#"([^\n]*)
COMENTARIO_BLOCO	"/*"([^*]|[*]+([^*/]|[\r\n]))*"*/"
ESPACO	[\t\r]+
QUEBRA_DE_LINHA	\n
PALAVRA_CHAVE	(int | float | double | char | void | switch | case | break | default | return | struct | typedef | enum | const | sizeof | continue | goto | extern | register | unsigned | signed | long|short)

6. Tipos de Erros Tratados

Erro	**Condição**
STRING_MAL_FORMADA	Não fechar aspas (e.g. “isso está errado)
COMENTARIO_SEM_FECHADOR	Não fechar comentário (e.g. /*awdawdaw)
NUM_MAL_FORMADO	Numero escrito de maneira errada (e.g 2..)

7. Descrição da Gramática

<expressao> ::= <IDENTIFICADOR> | <NUMERO_INTEIRO> |
<NUMERO_REAL>
| <STRING>
| <CARACTER>
| <expressao> <OPERADOR_SOMA> <expressao>
| <expressao> <OPERADOR_SUBTRACAO> <expressao>
| <expressao> <OPERADOR_MULTIPLICACAO> <expressao >
| <expressao> <OPERADOR_DIVISAO> <expressao >
| <expressao> <OPERADOR_MODULO> <expressao >
| <expressao > <ACUMULADOR_SOMA> <expressao >
| <expressao> <ACUMULADOR_SUBTRACAO> <expressao >
| <expressao> <INCREMENTO >
| <expressao> <DECREMENTO >
| <expressao> <GE> <expressao>
| <expressao> <LE> <expressao >
| <expressao> <EQ> <expressao >
| <expressao> <NE> <expressao >
| <expressao> <MAIOR_QUE> <expressao >
| <expressao> <MENOR_QUE> <expressao >
| <expressao> <LOGICO_AND> <expressao>
| <expressao> <LOGICO_OR> <expressao >
| <LOGICO_NOT> <expressao >
| <expressao> <ATRIBUICAO_CONDICIONAL> <expressao>
| <IDENTIFICADOR> <PARENTESE_E PARENTESE_D>
| <IDENTIFICADOR> <PARENTESE_E> <lista_argumentos> <PARENTESE_D>

<programa> ::= <lista_declaracoes>

<lista_declaracoes> ::= <lista_declaracoes> <declaracao> | <declaracao>

<lista_argumentos> ::= <expressao> | <lista_argumentos> <VIRGULA> <expressao>

<declaracao> ::= <declaracao_variavel> | <declaracao_funcao> | <estrutura_controle>
| <IMPORT> <IDENTIFICADOR> <PONTO_E_VIRGULA>
| <RETURN> <expressao> <PONTO_E_VIRGULA>

<declaracao_variavel> ::= <TIPO_INTEIRO> <IDENTIFICADOR>
<PONTO_E_VIRGULA> | <TIPO_INTEIRO> <IDENTIFICADOR>
<OPERADOR_ATRIBUICAO> <expressao> <PONTO_E_VIRGULA>

<declaracao_funcao> ::= <TIPO_INTEIRO> <IDENTIFICADOR>
<PARENTESE_E> <PARENTESE_D> <bloco>
| <TIPO_INTEIRO> <IDENTIFICADOR> <PARENTESE_E> <lista_parametros>
<PARENTESE_D> <bloco>

<lista_parametros> ::= <TIPO_INTEIRO> <IDENTIFICADOR>
| <lista_parametros> <VIRGULA> <TIPO_INTEIRO> <IDENTIFICADOR>

<estrutura_controle> ::= <IF> <PARENTESE_E> <expressao> <PARENTESE_D>
<bloco> <ELSE> <bloco>
| <IF> <PARENTESE_E> <expressao> <PARENTESE_D> <bloco>
| <WHILE> <PARENTESE_E> <expressao> <PARENTESE_D> <bloco>
| <FOR> <PARENTESE_E> <declaracao_variavel> <expressao>
<PONTO_E_VIRGULA> <expressao> <PARENTESE_D> <bloco>

<bloco> ::= <CHAVE_E> <lista_declaracoes> <CHAVE_D>
| <CHAVE_E> <CHAVE_D>

8. Tabela de Símbolos e Árvore sintática

```
int main() {
    if (a == 2) {
        return 1;
    } else {
        if (b != 1) {
            return 0;
        }
    }
}
```

```
Simbolos:
  - 0 (NUMERO_INTEIRO)
  - != (NE)
  - b (IDENTIFICADOR)
  - } (CHAVE_D)
  - ; (PONTO_E_VIRGULA)
  - 1 (NUMERO_INTEIRO)
  - 2 (NUMERO_INTEIRO)
  - == (EQ)
  - a (IDENTIFICADOR)
  - { (CHAVE_E)
  - ) (PARENTESE_D)
  - ( (PARENTESE_E)
  - main (IDENTIFICADOR)
```

9. Funções Utilizadas

int existe(Token* lista, const char* lexema):
Verifica se olexema já existe na lista.

void inserir_simbolo(const char* lexema, const char* tipo):
Adiciona um novo símbolo identificado (variável, número, operador, etc.) à lista.

void inserir_palavra_reservada(const char* lexema, const char* tpo):
Armazena palavras-chave e diretivas.

void imprimir_listas():
Imprimea lista depalavras reservadas e símbolos encontrados.

void liberar_listas():
Libera memória de todas as listas ao final da execução.

OBS: demais funções estão documentadas no próprio código.

10. Processo de Construção

IDE: Visual Studio Code