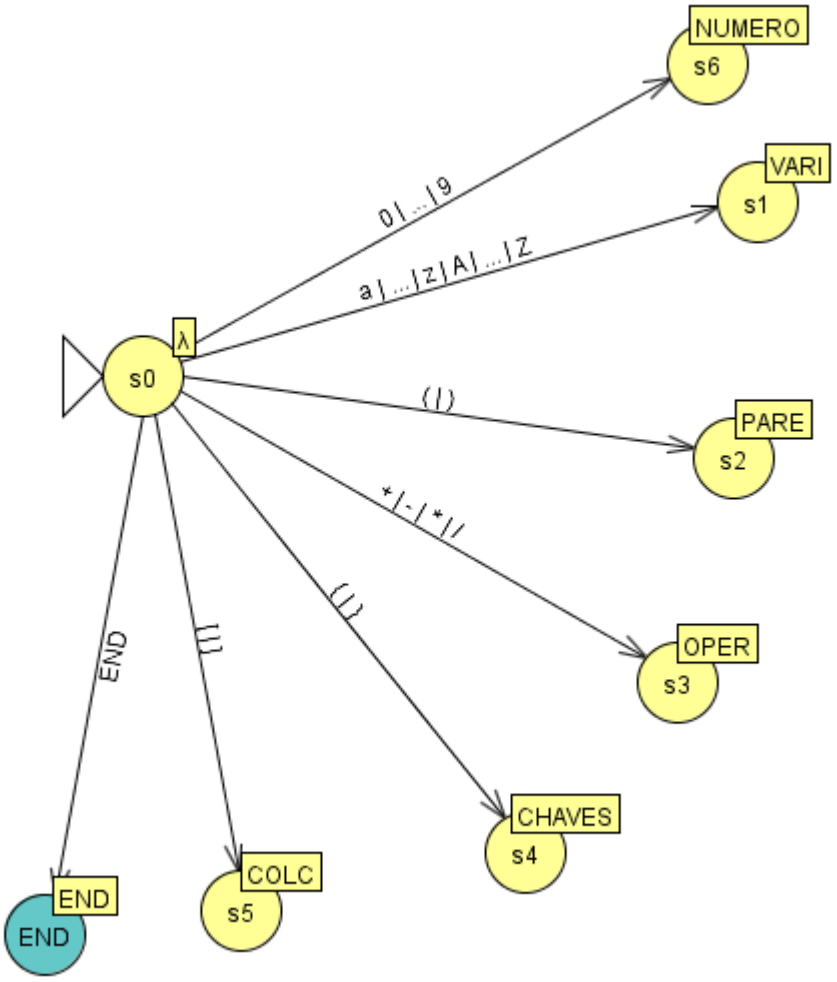
Alunos: Bernardo Seixas Grazziotti e Lucas Littig de Carvalho.

Expressões regulares para os tokens:

Token	ER	Exemplos de Lexemas
NUMERO	(0 | … | 9) +	1, 5, 50, 1000
COLC	[ ]	[ ]
CHAVES	{ }	{ }
OPER	(+ | - | * | /)	+, -, *, /
PARE	( )	( )
VARI	(a | … | z | A | … | Z)+	a, abc, cd, vc
END	Representa o final da análise léxica.	

Máquina de Moore



Testes do analisador léxico:

Teste 1:

4-5b+c*47

Console de saída:

NUME
OPER
NUME
VARI
OPER
VARI
OPER
NUME
Analise lexica realizada com sucesso no arquivo entrada.txt

Teste 2:

[b-(c)*a]

Console de saída:

COLC
VARI
OPER
PARE
VARI
PARE
OPER
VARI
COLC
Analise lexica realizada com sucesso no arquivo entrada.txt

Teste 3:

[b-(c)*a:b]

Console de saída:

COLC
VARI
OPER
PARE
VARI
PARE
OPER
VARI
Erro lexico encontrado: ‘:’
Caractere invalido.

Teste 4:

[b-(c)*ab-c*{d}]

Console de saída:

COLC
VARI
OPER
PARE
VARI
PARE
OPER
VARI
OPER
VARI
OPER
Erro lexico encontrado: '{'
'{}' nao eh permitido dentro de '[].'

Teste 5:

{d}(b)-ab@

Console de saída:

CHAVE
VARI
CHAVE
PARE
VARI
PARE
OPER
VARI
Erro lexico encontrado: ‘@’
Caractere invalido.

Implementação do analisador léxico:

```
#include <iostream>
#include <fstream>
#include <string>

enum TokenType{
    NUMERO,
    COLC,
    CHAVES,
    OPER,
    PARE,
    VARI,
    END
};

struct Token {
    TokenType type;
};
```

Primeiro, é feita a enumeração dos tokens que serão utilizados e a declaração de uma estrutura para atribuir os tokens a cada char do arquivo.

```
/*-----------------------------------------------------------------------------------------------------------*/

class Lexer {
private:
    std::string input;
    std::size_t Posicao;
    char charAtual;
```

Aqui, está sendo declarado a classe Lexer que irá ser o analisador léxico. Seus atributos privados serão input, que irá armazenar a entrada do arquivo, e a Posicao será utilizada para armazenar a posição do input.

```
public:
    Lexer(const std::string& text) : input(text), Posicao(0), charAtual(input[0]) {}

    int pare = 0;
    int colc = 0;
    int chave = 0;
    int oper = 0;
```

Agora, é declarado um construtor que recebe o texto que será analisado (input), a posição inicial de análise (Posicao), e o primeiro caractere do texto (charAtual). Esses valores iniciais permitem que o lexer comece a análise do texto a partir da primeira posição e acompanhe o caractere atual que está sendo processado durante a análise. Os atributos

públicos declarados serão utilizados para determinar a abertura de parênteses, colchetes, chaves, leitura de operadores e um auxiliar para determinar erros léxicos.

```
void leProximoCaractere() {
    Posicao++;
    if (Posicao < input.length()) {
        charAtual = input[Posicao];
    } else {
        charAtual = '\0';  // Marca o fim da entrada
    }
}
```

Essa primeira função, como diz, avança no arquivo em um espaço para ler o caractere e armazená-lo em charAtual.

```
Token reconheceToken() {

    while (charAtual != '\0') {

        if (isspace(charAtual)) {
            leProximoCaractere();
            continue;
        }
```

Começando o código que reconhece os tokens, é feito uma função iterativa para rodar até o charAtual ser igual ao fim do arquivo, representado por \0. A primeira condição se destina a identificar os espaços, ignorar e avançar para o próximo caractere.

```
        if (isdigit(charAtual)) {
            std::string numero;
            while (isdigit(charAtual)) {
                numero.push_back(charAtual);
                leProximoCaractere();
            }
            oper = 0;
            std::cout << "NUME" << std::endl;
            return {NUMERO};
        }


        if (isalpha(charAtual)) {
            std::string alfabeto;
            while (isalpha(charAtual)) {
                alfabeto.push_back(charAtual);
                leProximoCaractere();
            }
            oper = 0;
            std::cout << "VARI" << std::endl;
```

```
        return {VARI};
    }
```

As duas condições acima se destinam a identificar os tokens números e variáveis utilizando as funções isdigit e isalpha. Ambas as condições armazenam o número ou variável em um vetor usando push_back. A variável oper será melhor explicada mais à frente no switch case de operações, mas basicamente serve para manter dizer ao código se uma variável ou número foi lida junto à operação.

```
    switch (charAtual) {
        case '(':
            pare++;
            std::cout << "PARE" << std::endl;
            leProximoCaractere();
            return {PARE};

        case ')':

            if(pare == 0){
                std::cout << "Erro léxico encontrado: ')'\nEsperado '(' antes." << std::endl;
                exit(1);
            }else{
                std::cout << "PARE" << std::endl;
                pare--;
                leProximoCaractere();
                return {PARE};
            }
```

Para evitar uma série de repetições de if else, o switch case foi utilizado para comparar o charAtual para gerar os tokens do arquivo de parênteses, colchetes, chaves e operações.

Dentro dos cases, de parênteses, colchetes e chaves, é verificado se eles foram abertos e fechados corretamente, isso funciona utilizando os seguintes passos:

- Quando o caractere lido for a abertura desses caracteres (‘(‘, ‘[‘, ‘{‘), é somado 1 ao contador respectivo de cada um.

- Quando, houver o caracter que é usado para fechar (‘)’, ‘]’, ‘}') é decrementado 1 do contador respectivo. Ao final do switch case, se esses contadores forem diferentes de 0, é exibido erro léxico, indicando que algum desses elementos abriu mas não fechou.

- Caso algum elemento que fecha (‘)’, ‘]’, ‘}') for identificado e o contador for 0, isso significa que nenhum elemento utilizado para abrir foi utilizado antes, indicando erro léxico.

Esses passos acima servem para todos, mas há alguns casos que servem apenas para as chaves e colchetes que poderão ser observados abaixo.

```
            case '[':
                colc++;
                if(pare == 1 && colc == 1){
                    std::cout << "Erro léxico encontrado: '['\n'[]' não é permitido dentro de '()'." <<
std::endl;
                    exit(1);
                }
                leProximoCaractere();
                std::cout << "COLC" << std::endl;
                return {COLC};

            case ']':
                if(pare == 1 && colc == 1){ //Caso em que [(], ou seja, parenteses ou colchentes
sem fechar
                    std::cout << "Erro léxico encontrado: ']'\nEsperado ')'." << std::endl;
                    exit(1);
                }
                if(colc == 0){
                    std::cout << "Erro léxico encontrado: ']'\nEsperado '[' antes." << std::endl;
                    exit(1);
                }else{
                    std::cout << "COLC" << std::endl;
                    colc--;
                    leProximoCaractere();
                    return {COLC};
                }

            case '{':
                chave++;
                if(colc == 1 && chave == 1){
                    std::cout << "Erro léxico encontrado: '{'\n'{}' nao é permitido dentro de '[].'" <<
std::endl;
                    exit(1);
                }
                if(pare == 1 && chave == 1){
                    std::cout << "Erro léxico encontrado: '{'\n'{}' nao é permitido dentro de '().'" <<
std::endl;
                    exit(1);
                }
                std::cout << "CHAVE" << std::endl;

                leProximoCaractere();
                return {CHAVES};

            case '}':
                if(pare == 1 && chave == 1){ //Caso em que {(} ou {[}, ou seja, parenteses ou
colchentes sem fechar
```

```
            std::cout << "Erro léxico encontrado: '}'\nEsperado ')'." << std::endl;
            exit(1);
        } else if(colc == 1 && chave == 1){
            std::cout << "Erro léxico encontrado: '}'\nEsperado ']'." << std::endl;
            exit(1);
        }

        if(chave == 0){
            std::cout << "Erro léxico encontrado: '}'\nEsperado '{' antes." << std::endl;
            exit(1);
        }else{
            std::cout << "CHAVE" << std::endl;
            chave--;
            leProximoCaractere();
            return {CHAVES};
        }
```

No caso, de colchetes e chaves, há uma ordem a ser seguida, as chaves devem ficar fora dos colchetes e parentes, e colchetes fora dos parentes. Tomando a chaves como exemplo, no caso da abertura dela ({), o contador é somado e depois se verifica se os contadores de colchetes e parênteses são iguais a 1, caso sim, há um erro léxico, pois significa que uma chave está sendo aberta dentro de colchetes e parênteses. Essa mesma lógica é aplicada ao case da abertura de colchetes, mas apenas com parênteses.

Outra situação que não deve ocorrer é o caso que um colchete ou chave feche antes que os elementos de dentro, como o parêntese, não tenham fechado ({(}). Então, se no case de fecha de chaves, por exemplo, o contador da chave for igual a 1 e os de parênteses e colchetes também forem igual a 1, é identificado como erro, pois os parênteses ou colchetes não tiveram seu valor decrementado, indicando que fecharam.

Em resumo então, no caso de pare, chave, colc forem igual a ou maior a 1, indica que um caractere que os representa foi aberto e precisa ser fechado. A cada fechamento, essas variáveis são decrementadas em 1. Ao final, esse final deve ser igual a 0, representando assim, que todo colchete, chaves e parênteses que foram abertos, foram devidamente fechados.

```
    case '+':

        if(oper == 1){
            std::cout << "Erro léxico encontrado: '+'\nEra(m) esperado(s)
'0123456789abcdefghijklmnopqrstuvwxyz'." << std::endl;
            exit(1);
        }
        oper = 1;
        std::cout << "OPER" << std::endl;
        leProximoCaractere();
        return {OPER};
```

```
            case '/':

                if(oper == 1){
                    std::cout << "Erro léxico encontrado: '/'\nEra(m) esperado(s)
'0123456789abcdefghijklmnopqrstuvwxyz'." << std::endl;
                    exit(1);
                }
                oper = 1;
                std::cout << "OPER" << std::endl;
                leProximoCaractere();
                return {OPER};

            case '*':

                if(oper == 1){
                    std::cout << "Erro léxico encontrado: '*'\nEra(m) esperado(s)
'0123456789abcdefghijklmnopqrstuvwxyz'." << std::endl;
                    exit(1);
                }
                oper = 1;
                std::cout << "OPER" << std::endl;
                leProximoCaractere();
                return {OPER};

            case '-':

                if(oper == 1){
                    std::cout << "Erro léxico encontrado: '-'\nEra(m) esperado(s)
'0123456789abcdefghijklmnopqrstuvwxyz'." << std::endl;
                    exit(1);
                }
                oper = 1;
                std::cout << "OPER" << std::endl;
                leProximoCaractere();
                return {OPER};

                default:
                std::cerr << "Erro léxico encontrado: " << charAtual << “\nCaractere
inválido.”<<std::endl;
                exit(1);
        }
    }
```

Nos cases de operação, é verificado se a variável oper é igual a 1, pois, se for, significa que um char de operação já foi lido anteriormente e não foi completado com algum número ou variável, dessa forma, gera um erro léxico. Se isso não ocorrer, oper recebe 1 e o token é gerado normalmente.

Caso nenhum dos casos acima for cumprido, significa que o caractere que foi lido não pertence ao alfabeto da questão e não gera tokens.

```
        if(colc != 0){
            std::cout << "Erro léxico encontrado: Era(m) esperado(s) ']'" << std::endl;
        }
        if(pare != 0){
            std::cout << "Erro léxico encontrado: Era(m) esperado(s) ')'" << std::endl;
        }
        if(chave != 0){
            std::cout << "Erro léxico encontrado: Era(m) esperado(s) '}'" << std::endl;
        }
        if(oper == 1){
            std::cout << "Erro léxico encontrado: '''\nEra(m) esperado(s)
'0123456789abcdefghijklmnopqrstuvwxyz'." << std::endl;
        }

        return {END};
    }
};
```

Ao final da função, é verificado se as chaves, colchetes e parênteses foram fechados, ou seja, seus contadores devem ser 0. E também verifica-se se algum operador foi ou não o último caractere lido, caso sim, é identificado como erro léxico.

```
/*-----------------------------------------------------------------------------------------------------------*/

int main() {
    std::string nomeArquivo = "entrada.txt";
    std::ifstream arquivo(nomeArquivo);

    if (!arquivo.is_open()) {
        std::cerr << "Erro ao abrir o arquivo!" << nomeArquivo << std::endl;
        return 1;
    }

    std::string conteudoArquivo;
    std::string linha;
    while (std::getline(arquivo, linha)) {
        conteudoArquivo += linha + "\n";
    }

    arquivo.close();

    Lexer lexer(conteudoArquivo);

    Token token = lexer.reconheceToken();
```

```
    while (token.type != END) {
        token = lexer.reconheceToken();
    }

    std::cout << "Analise lexica realizada com sucesso no arquivo entrada.txt" << std::endl;


    return 0;
}
```

Na main, o endereço do arquivo a ser lido e seu conteúdo é passado inteiramente para a variável conteudoArquivo. Cada elemento separado por espaço do arquivo é separado por linha. Caso ler o arquivo passado dê errado, o programa é encerrado. Depois disso, um objeto do tipo Lexer é criado usando o seu construtor e passando o conteudoArquivo nos parâmetros. Depois, os tokens são gerados e reconhecidos chamando a função reconheceToken de cada linha, até que o token END seja reconhecido. Caso no meio das chamadas de reconheceToken não forem encontrados erros, então o texto do arquivo de entrada é aceito pela gramática.