Etapa 1: Análise Léxica (Grupo 3)

Nomes: Rômulo Peigas Fonseca e Leonardo Bacellar

1) Defina formalmente, através de expressões/definições regulares, a sintaxe de cada um dos tipos de lexemas a serem extraídas do texto-fonte pelo analisador léxico, bem como de cada um dos espaçadores e comentários.

dígito → 0 | 1 | .. | 9

letra → a-z | A-Z

esp→ \a | \b | \f | \n | \r | \t | \v | \\ | \" | ***espaço***

del → (|) | { | } | [|] | ; | : | , | . | ..

op → + | - | * | / | ^ | = | ~= | <= | >= | < | > | == | !

num → dígito dígito*

id → ((_letra | _dígito) | letra)(letra | dígito | _)*

literal_string → “(letra | dígito | ws | op | del)*”

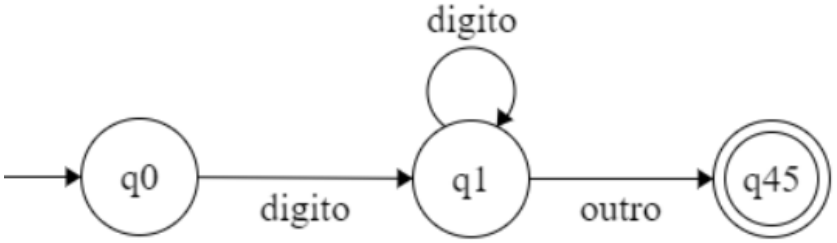
coment_S → - - (letra | dígito | esp - [\n] | op | del)*\n

coment_L → - - [[(letra | dígito | esp | op | del)*]]

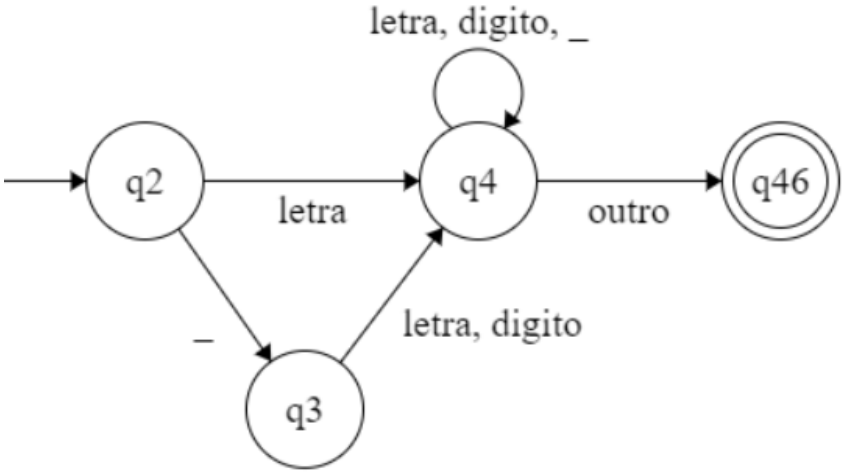
2) Converta cada uma das expressões regulares em autômatos finitos com saída nos estados, que emita como saída a lexema encontrada ao abandonar cada um dos estados finais para iniciar o reconhecimento de mais uma lexema do texto.

- **num** - Autômato feito para reconhecer números, elaborado a partir da definição regular “num”. O estado 45 foi colocado após os

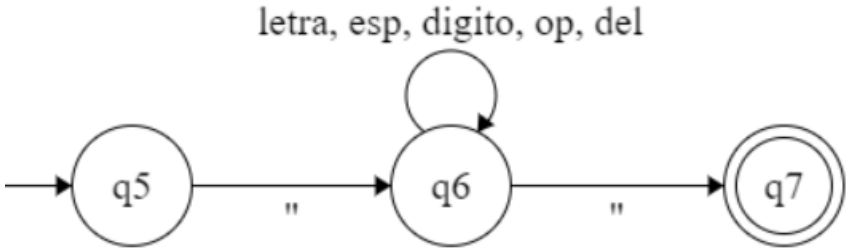
testes do programa, para ter um melhor controle sobre as ações executadas pelos estados. A palavra “outro” significa qualquer caractere que não seja um dígito.



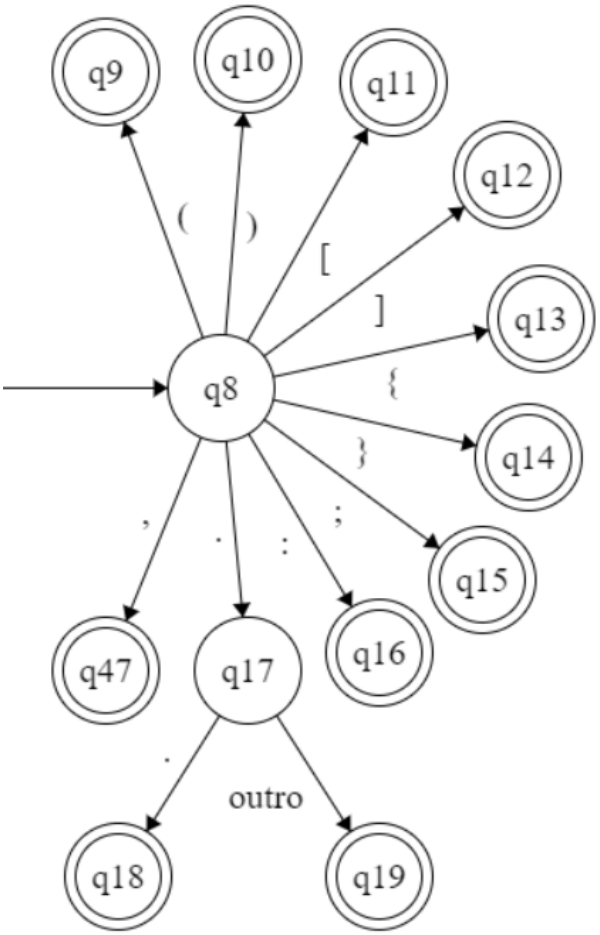
- **id** - Autômato feito para reconhecer identificadores, elaborado a partir da definição regular “id”. O estado 46 foi colocado após os testes do programa, para ter um melhor controle sobre as ações executadas pelos estados. A palavra “outro” significa qualquer caractere que não seja um dígito, letra ou _.



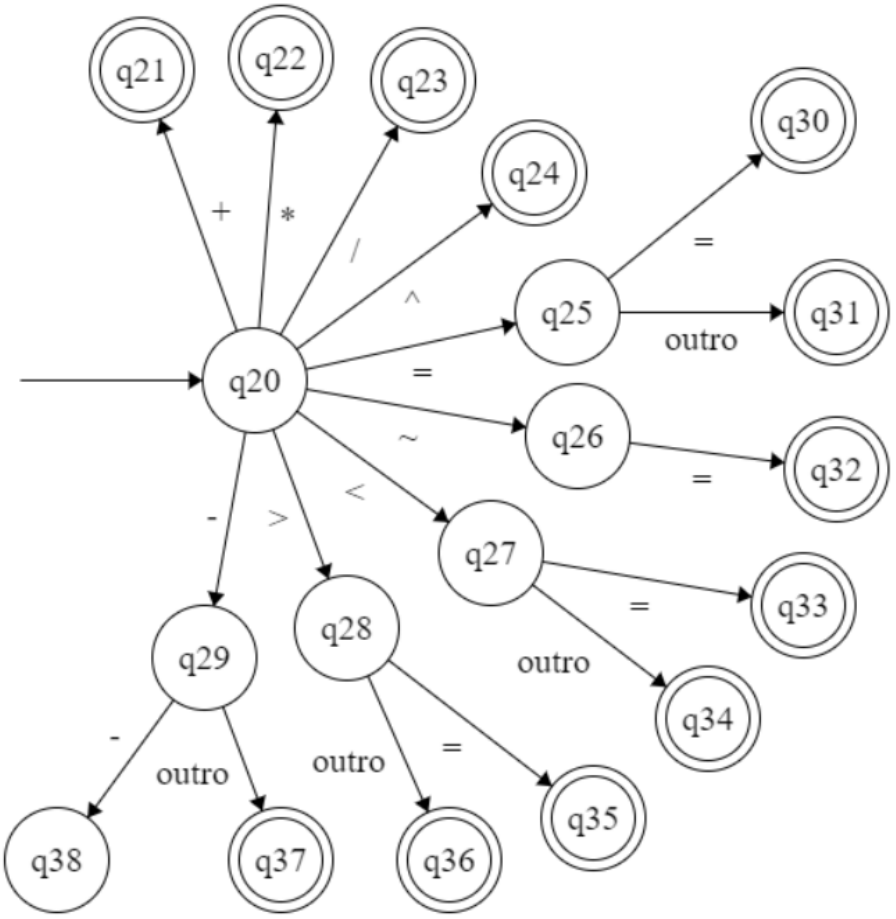
- **literal de string** - Autômato feito para reconhecer literais de string, elaborado a partir da definição regular “literal_string”.



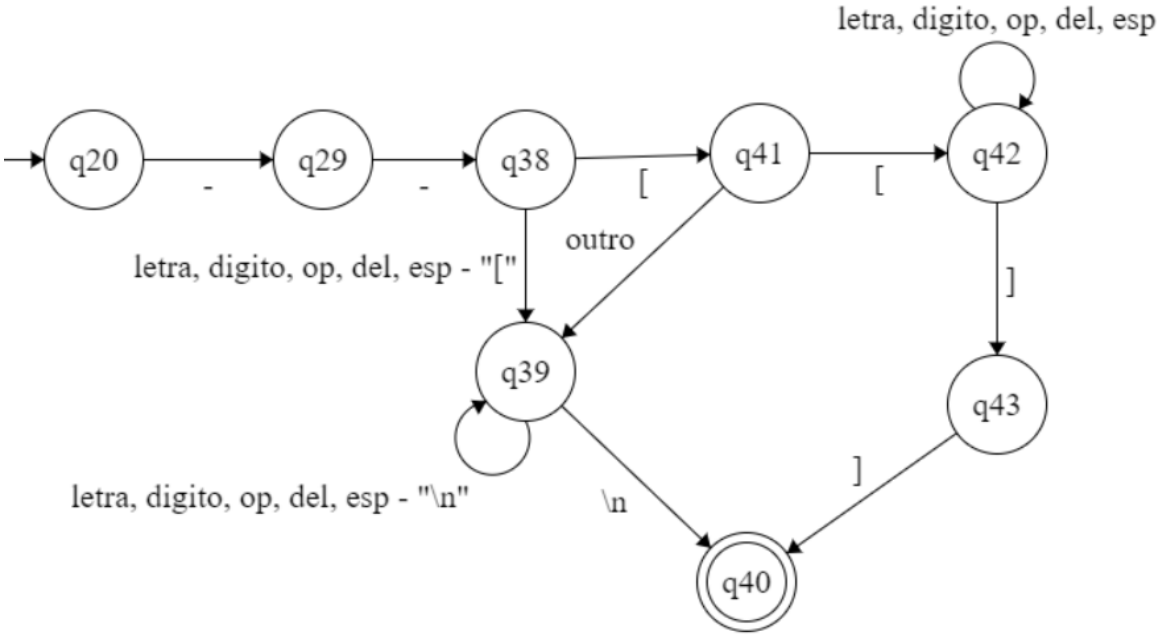
- **Delimitadores** - Autômato feito para reconhecer delimitadores, elaborado a partir da definição regular “del”. O estado 47 foi colocado após os testes do programa, pois percebemos que não tínhamos colocado este delimitador antes da implementação. A palavra “outro” significa qualquer caractere que não seja um “.” .



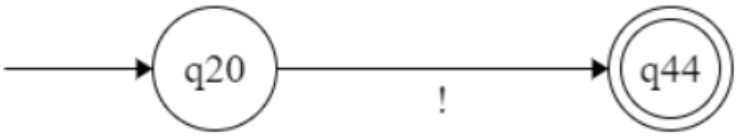
- **Operadores** - Autômato feito para reconhecer operadores, elaborado a partir da definição regular “op”.



- **Comentários** - Autômato feito para reconhecer comentários longos e curtos, elaborado a partir das definições regulares “comment_S” e “comment_L”. Aprimoramos a funcionalidade do autômato dos operadores para que seja atribuído à função de iniciar a leitura dos comentários curtos e longos.



- No autômato dos operadores também foi acrescentada a leitura do caractere “!”, que foi escolhido como nosso lexema especial indicativo da ausência de novas lexemas no texto de entrada.



3) Agrupe/converta os autômatos finitos em uma sub-rotina, escrita na linguagem de programação C/C++, que para cada lexema lido, deve retornar o nome-token e o atributo correspondentes.

4) Crie um programa principal, que chame repetidamente a sub-rotina construída, e a aplique sobre um arquivo do tipo texto contendo o texto-fonte a ser analisado. Após cada chamada, esse programa deve imprimir o token referente ao lexema lido.

Os itens 3 e 4 se encontram no programa em anexo.

5) Relate o funcionamento do analisador léxico construído, incluindo no relatório: descrição teórica do programa; descrição da sua estrutura; descrição de seu funcionamento; descrição dos testes realizados e das saídas obtidas.

Teoria

Esse programa tem o objetivo de receber os caracteres de um arquivo texto e realizar uma análise léxica para geração de tokens com seus respectivos atributos, quando houver.
O programa lê os caracteres dos lexemas um de cada vez mudando o estado de acordo com o caractere encontrado, caso não seja o caractere desejado, ele procura por um autômato que aceite este caractere, caso não haja um autômato capaz de lê-lo será mostrada uma mensagem de erro. Caso o lexema seja inteiramente lido, significa que ele pertence a um dos padrões da linguagem, o autômato que o reconheceu no seu estado de aceitação gera um token, nessa etapa

torna-se necessário em alguns casos a verificação na tabela de símbolos para identificar lexemas especiais, como por exemplo, as palavras reservadas.

Estrutura

Como especificado na descrição do trabalho definimos constantes para as palavras reservadas, nomes dos tokens e para os atributos.

```
//NOME TOKENS
#define AND 250;
#define BREAK 251;
#define DO 252;
#define ELSE 253;
#define ELSEIF 254;
#define END 255;
#define FALSE 256;
#define FOR 257;
#define FUNCTION 258;
#define IF 259;
#define IN 260;
#define LOCAL 261;
#define NIL 262;
#define NOT 263;
#define OR 264;
#define REPEAT 265;
#define RETURN 266;
#define THEN 267;
#define TRUE 268;
#define UNTIL 269;
#define WHILE 270;
#define ID 271;
#define NUM 272;
#define DEL_OP 273;
#define STRING 274;
#define FIM -1;
```

```
//ATRIBUTOS
#define DOISPF 500; // ..
#define LE 501; // <=
#define EQ 502; // ==
#define NE 503; // ~=
#define GE 504; // >=
#define SemAtrib -1;
```

Utilizamos o arquivo enviado junto com a descrição do trabalho como base para a implementação do nosso analisador léxico. A struct *Token* e a função *readFile* foram mantidas e a função *falhar* sofreu poucas modificações, para se ajustar a estrutura de autômatos que criamos.

```
struct Token{
  int nome_token;
  int atributo;
};
```

```
char *readFile(char *fileName)
{
    FILE *file = fopen(fileName, "r");
    char *code;
    int n = 0;
    int c;

    if(file == NULL) return NULL;

    fseek(file, 0, SEEK_END);
    long f_size = ftell(file);
    fseek(file, 0, SEEK_SET);

    code = new char (f_size);

    while ((c = fgetc(file))!= EOF)
    {
        code[n++] = (char) c;
    }
    code[n] = '\0';
    return code;
}
```

```
int falhar()
{
    switch(estado)
    {
    case 0: partida = 2; break; //num

    case 2: partida = 5; break; //id

    case 5: partida = 8; break; //string

    case 8: partida = 20; break; //del

    case 20: //op
        //retornar msg de erro
        printf("Erro encontrado no código\n");
        cont_sim_lido++;
        break;

    default: printf("Erro do compilador");
    }
    return(partida);
}
```

Acrescentamos duas novas variáveis globais, a primeira *lexema* é uma string usada para concatenar os caracteres lidos do arquivo fonte para formar os lexemas, a segunda *tabelaSimbolos* é um vetor dinâmico usado para armazenar os IDs e os literais de strings lidos, assim como as palavras reservadas da linguagem.

```
int estado = 0;
int partida = 0;
int cont_sim_lido = 0;
char *code;
string lexema;
vector<string> tabelaSimbolos;
```

Fizemos as modificações necessárias na função *proximo_token* para corresponder aos autômatos que criamos para identificar os padrões da linguagem.

```
Token proximo_token()
{
    Token token;
    char c;
    char aux;
    int count_colA;
    int count_colF;
    while(code[cont_sim_lido] != EOF)
    {
        switch(estado)
        {
            case 0:  //Estado 0 (NUM)
                lexema = "";
                c = code[cont_sim_lido];
                if((c == ' ')||(c == '\n')){
                    estado = 0;
                    cont_sim_lido++;
                }
                else if (isdigit(c)){
                    estado = 1;
                    lexema +=c;
                }
                else{
                     estado = falhar();
                }
                break;
```

E por na função *main* nós inserimos todas as palavras reservadas na tabela de símbolos e através de um comando de repetição, executamos a sub rotina até que o token retornado seja o caractere especial “!” .

```
int main ()
{
    Token token;
    tabelaSimbolos.push_back("and");
    tabelaSimbolos.push_back("break");
    tabelaSimbolos.push_back("do");
    tabelaSimbolos.push_back("else");
    tabelaSimbolos.push_back("elseif");
    tabelaSimbolos.push_back("end");
    tabelaSimbolos.push_back("false");
    tabelaSimbolos.push_back("for");
    tabelaSimbolos.push_back("function");
    tabelaSimbolos.push_back("if");
    tabelaSimbolos.push_back("in");
    tabelaSimbolos.push_back("local");
    tabelaSimbolos.push_back("nil");
    tabelaSimbolos.push_back("not");
    tabelaSimbolos.push_back("or");
    tabelaSimbolos.push_back("repeat");
    tabelaSimbolos.push_back("return");
    tabelaSimbolos.push_back("then");
    tabelaSimbolos.push_back("true");
    tabelaSimbolos.push_back("until");
    tabelaSimbolos.push_back("while");

    code = readFile("programa.txt");
    if (code == NULL){
        cout << "Arquivo nao encontrado"<<endl;
    }

    while(true){
        token = proximo_token();
        if ((token.nome_token == -1) && (token.atributo == -1)){
            break;
        }
    }
    return 0;
}
```

Funcionamento

O programa lê um caractere do arquivo fonte e passa para o primeiro autômato, que verifica se este caractere pertence ao padrão que ele reconhece. Caso não pertença o autômato vai chamar a função *falhar* que vai passar este caractere para o próximo autômato na tentativa reconhecê-lo, esse processo vai se repetir até que o caractere seja reconhecido ou até que uma mensagem de erro seja mostrada, caso nenhum autômato consiga reconhecê-lo.

Caso o reconhecimento do caractere seja bem sucedido, ele é concatenado com a string vazia *lexema* e o autômato muda de estado. Em seguida, o programa, através de um comando de loop, lê o próximo caractere e verifica se ele pertence ao mesmo padrão do anterior. Em caso afirmativo, ele também é concatenado com *lexema,* caso contrário o autômato muda para seu estado final, onde o *lexema* atual será usado para gerar um token que será retornado a função *main*.

O programa então torna *lexema* uma string vazia e continua repetindo o processo descrito nos parágrafos acima até que a *main* receba o token especial de parada <!,>.

Vale ressaltar que, para lexemas com uma quantidade definida de caracteres como “>=”, o loop não foi utilizado, pois as mudanças de estado dos autômatos usando o caracteres recebidos já são suficientes para fazer o reconhecimento desses lexemas.

Os token retornados pela sub-rotina podem ser de uma das quatro classes *num* (número), *del_op* (delimitadores e operadores), *strings* (literais de string) e *id* (identificadores).

- Para os números o token retornado será <num, numero_lido>;
- Para os delimitadores e operadores, se for um caractere simples como “+” o token retornado será <ASCII_de_“+”, >, caso contrário será <del_op, valor_definido>, sendo o *valor_definido* o valor colocado nos *#defines* para este lexema;

- Para os identificadores, antes de ser retornado o token, o lexema é comparado com as palavras reservadas que estão armazenadas na tabela de símbolos, se o lexema for uma palavra reservada o token retornado será <palavra_reservada, >, caso contrário o lexema será armazenado na tabela de símbolos e o seguinte token será retornado <id, número da linha na tabela de símbolos>;
- Para as literais de string, o lexema será armazenado na tabela de símbolos e o seguinte token será retornado <string, número da linha na tabela de símbolos>;

Testes

```
Lexema: function ---  token: <function,>
Lexema: add ---  token: <id,21>
Lexema: (  ---  token: <(,>
Lexema: a ---  token: <id,22>
Lexema: )  ---  token: <),>
Lexema: local ---  token: <local,>
Lexema: sum ---  token: <id,23>
Lexema: = ---  token: <=,>
Lexema: 0 --- token: <num, 0>
Lexema: for ---  token: <for,>
Lexema: i ---  token: <id,24>
Lexema: ,  ---  token: <,,>
Lexema: v ---  token: <id,25>
Lexema: in ---  token: <in,>
Lexema: ipairs ---  token: <id,26>
Lexema: (  ---  token: <(,>
Lexema: a ---  token: <id,22>
Lexema: )  ---  token: <),>
Lexema: do ---  token: <do,>
Lexema: sum ---  token: <id,23>
Lexema: = ---  token: <=,>
Lexema: sum ---  token: <id,23>
Lexema: + ---  token: <+,>
Lexema: v ---  token: <id,25>
Lexema: end ---  token: <end,>
Lexema: return ---  token: <return,>
Lexema: sum ---  token: <id,23>
Lexema: end ---  token: <end,>
Lexema: ! ---  token: <!,>
```

```
-- add all elements of array `a'
    function add (a)
      local sum = 0
      for i,v in ipairs(a) do
        sum = sum + v
      end
      return sum
    end!
```

```
Lexema: string1 ---  token: <id,21>
Lexema: = ---  token: <=,>
Lexema: "Lua" --- token: <string, 22>
Lexema: string2 ---  token: <id,23>
Lexema: = ---  token: <=,>
Lexema: "Tutorial" --- token: <string, 24>
Lexema: string_concat ---  token: <id,25>
Lexema: = ---  token: <=,>
Lexema: string1 ---  token: <id,21>
Lexema: .. ---  token: <del_op,..>
Lexema: string2 ---  token: <id,23>
Lexema: ! ---  token: <!,>
```

```
string1 = "Lua"
string2 = "Tutorial"

--[[Concatenando as strings 1 e 2
usando o operador ..]]
string_concat = string1..string2
!
```