Cálcula - Modelo de Analex

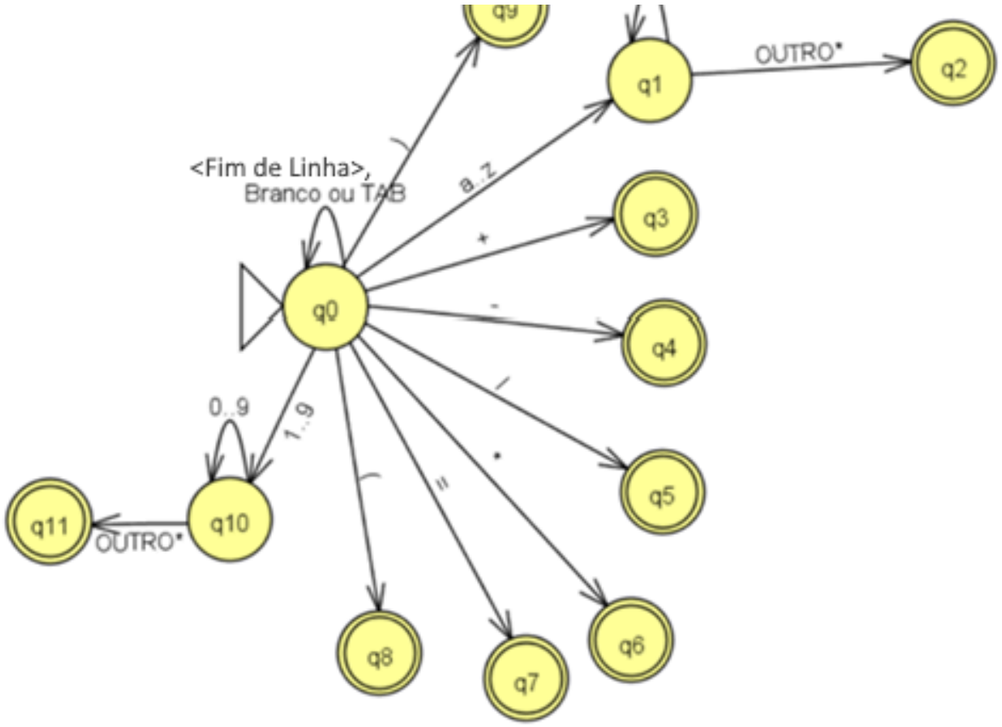
Descrição de Cálcula:

Considere uma linguagem de programação interpretada chamada *Cálcula*, que faz o cálculo de expressões aritméticas com variáveis e números inteiros. Os comandos dessa linguagem são todos escritos no formato de atribuição, como em α = β, onde α é sempre um identificador do tipo variável cujo lexema inicia sempre com letra minúscula e pode ser seguido por letras minúsculas, dígitos ou o caracter '_' e β é uma expressão aritmética (no formato *infixo*), constituída por identificadores variáveis e/ou constantes inteiras, como operandos, e pelas quatro operações aritméticas (+, –, *, /), como operadores; a expressão em β pode ainda conter parênteses (abre e fecha) para modificar a precedência das operações, que seguem as regras da aritmética comum. Caracteres em branco podem ocorrer em qualquer lado da expressão e devem ser desconsiderados. A execução de cada linha de comando da linguagem produz como resultado o valor atribuído ao lado esquerdo (α) da expressão. Além disso, esse valor é preservado na variável até que seja substituído em outra expressão atribuída a ela, assim como, uma vez inicializada, a variável pode ser usada do lado esquerdo (β) das expressões seguintes.
Um **AFD completo** que pode ser usado para implementar a **fase de análise léxica** de um interpretador para esta linguagem é apresentado a seguir, juntamente com o código C que o implementa (estruturas de dados + códigos fontes), correspondendo a um Analisador Léxico implementado aplicando a técnica de AFDs.

Veja o exemplo de execução de Cálcula a seguir para entender melhor como a linguagem funciona.

Exemplo de execução de Cálcula (linhas em preto indicam comandos da linguagem e em vermelho o valor resultante): Cmd> a=5 5 Cmd> b =7 7 Cmd> soma= a+ b 12 Cmd> multi10 = soma * 10 120	Cmd> c = 2 2 Cmd> delta=b*b – 4* a* c 9 Cmd> z = (b – 5) * (2 + c) 8 Cmd> a = a 5 Cmd> a = b * 2 / 7 2

AFD desenvolvido para Analisador Léxico de Cálcula:



Implementação em C do Analex de Cálcula usando o AFD acima:

Arquivo header Analex.h:

```
#ifndef ANALEX
#define ANALEX
#define TAM_MAX_LEXEMA 31

enum TOKEN_CAT {ID=1, SN, CT_I, FIM_EXPR, FIM_ARQ};
/* Onde: ID: Identificador, SN: Sinal; CT_I: Constante numérica inteira */

enum SINAIS {ATRIB = 1, ADICAO, SUBTRACAO, MULTIPLIC, DIVISAO, ABRE_PAR, FECHA_PAR}; // Sinais válidos
da linguagem

typedef
    struct {
        enum TOKEN_CAT cat;    // deve receber uma das constantes de enum TOKEN_CAT
        union {     // parte variável do registro
            int codigo;  // para tokens das categorias SN
            char lexema[TAM_MAX_LEXEMA];  // cadeia de caracteres que corresponde ao nome do token da
cat. ID
```

```
            int valInt; // valor da constante inteira em tokens da cat. CT_I
        };
    } TOKEN;    // Tipo TOKEN

#endif
/* Contador de linhas do código fonte */
int contLinha = 1;
```

Arquivo fonte Analex.c:

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>
#include <stdbool.h>
#include "Analex.h"
#define TAM_LEXEMA 50
#define TAM_NUM 20

void error(char msg[]) {
    printf("%s\n", msg);
    exit(1);
}

TOKEN AnaLex(FILE *fd) {

    int estado;
    char lexema[TAM_LEXEMA] = "";
    int tamL = 0;
    char digitos[TAM_NUM] = "";
    int tamD = 0;

    TOKEN t;

    estado = 0;
    while (true) {
        char c = fgetc(fd);
        switch (estado) {
            case 0: if (c == ' ' || c == '\t') estado = 0;
                    else if (c >= 'a' && c <= 'z') {  // inicio de identificador - inicializa lexema
                        estado = 1;
                        lexema[tamL] = c;
                        lexema[++tamL] = '\0';
                    }
                    else if (c >= '1' && c <= '9') {  // inicio de constante inteira - inicializa
digitos
                        estado = 10;
                        digitos[tamD] = c;
                        digitos[++tamD] = '\0';
                    }
                    else if (c == '+') {    // sinal de adicao - monta e devolve token
                        estado = 3;
                        t.cat = SN;
                        t.codigo = ADICAO;
                        return t;
                    }
```

```
            else if (c == '-') {    // sinal de subtracao - monta e devolve token
                estado = 4;
                t.cat = SN;
                t.codigo = SUBTRACAO;
                return t;
            }
            else if (c == '*') {    // sinal de multiplicacao - monta e devolve token
                estado = 6;
                t.cat = SN;
                t.codigo = MULTIPLIC;
                return t;
            }
            else if (c == '/') {    // sinal de divisao - monta e devolve token
                estado = 5;
                t.cat = SN;
                t.codigo = DIVISAO;
                return t;
            }
            else if (c == '=') {    // sinal de atribuicao - monta e devolve token
                estado = 7;
                t.cat = SN;
                t.codigo = ATRIB;
                return t;
            }
            else if (c == '(') {   // sinal de abre parenteses - monta e devolve token
                estado = 8;
                t.cat = SN;
                t.codigo = ABRE_PAR;
                return t;
            }
            else if (c == ')') {    // sinal de fecha parenteses - monta e devolve token
                estado = 9;
                t.cat = SN;
                t.codigo = FECHA_PAR;
                return t;
            }
            else if (c == '\n') {
                estado = 0;
                t.cat = FIM_EXPR;   // fim de linha (ou expressao) encontrado
                contLinha++;
                return t;
            }
            else if (c == EOF) {    // fim do arquivo fonte encontrado
                   t.cat = FIM_ARQ;
                   return t;
            }
            else
                error("Caracter invalido na expressao!");    // sem transicao valida no AFD
            break;
        case 1: if ((c >= 'a' && c <= 'z') || (c >= '0' && c <= '9') || (c == '_')) {
                estado = 1;
                lexema[tamL] = c;      // acumula caracteres lidos em lexema
                lexema[++tamL] = '\0';
            }
            else {                     // transicao OUTRO* do estado 1 do AFD
                estado = 2;            // monta token identificador e retorna
                ungetc(c, fd);
                t.cat = ID;
                strcpy(t.lexema, lexema);
```

```
                        return t;
                    }
                    break;
            case 10:if (c >= '0' && c <= '9') {
                        estado = 10;
                        digitos[tamD] = c;       // acumula digitos lidos na variaavel digitos
                        digitos[++tamD] = '\0';
                    }
                    else {                       // transicao OUTRO* do estado 10 do AFD
                        estado = 11;             // monta token constante inteira e retorna
                        ungetc(c, fd);
                        t.cat = CT_I;
                        t.valInt = atoi(digitos);
                        return t;
                    }
        }
    }
}

int main() {

    FILE *fd;
    TOKEN tk;

    if ((fd=fopen("expressao.dat", "r")) == NULL)
        error("Arquivo de entrada da expressao nao encontrado!");

    printf("LINHA %d: ", contLinha);

   while (1) {    // laço infinito para processar a expressão até o fim de arquivo
        tk = AnaLex(fd);
        switch (tk.cat) {
            case ID: printf("<ID, %s> ", tk.lexema);
                     break;
            case SN: switch (tk.codigo) {
                        case ADICAO: printf("<SN, ADICAO> ");
                        break;
                        case SUBTRACAO: printf("<SN, SUBTRACAO> ");
                        break;
                        case MULTIPLIC: printf("<SN, MULTIPLICACAO> ");
                        break;
                        case DIVISAO: printf("<SN, DIVISAO> ");
                        break;
                        case ATRIB: printf("<SN, ATRIBUICAO> ");
                        break;
                        case ABRE_PAR: printf("<SN, ABRE_PARENTESES> ");
                        break;
                        case FECHA_PAR: printf("<SN, FECHA_PARENTESES> ");
                        break;
                     }
                     break;
            case CT_I: printf("<CT_I, %d> ", tk.valInt);
                       break;
            case FIM_EXPR: printf("<FIM_EXPR, %d>\n", 0);
                           printf("LINHA %d: ", contLinha);
                           break;
            case FIM_ARQ: printf(" <Fim do arquivo encontrado>\n");
        }
        if (tk.cat == FIM_ARQ) break;
```

```
        }

        fclose(fd);
        return 0;
    }
```