2. AUTÔMATO FINITO E ANÁLISE LÉXICA

CONCEITOS BÁSICOS EM GRAMÁTICAS E LINGUAGENS

Alfabeto **ou** ***Classe de Caracteres*** **denota um conjunto finito de símbolos (símbolo e caracter serão usados como sinônimos nesse curso).**

```
Exemplo:
    O conjunto {0, 1} é um alfabeto, composto dos dois símbolos 0 e
1, freqüentemente chamado de alfabeto binário.
```

Cadeia **é uma seqüência finita de símbolos de um alfabeto, tal como 001. O** ***tamanho de uma cadeia x*****, normalmente denotado por** ***|x|*****, é o número total de símbolos em** ***x*****. Uma cadeia especial é a** ***cadeia vazia*****, denotada por** ***&*****. O tamanho da cadeia vazia é 0 (zero).**

```
Exemplo:
    x = 010110      |x| = 6
```

Se ***x*** **e** ***y*** **são cadeias, então a** ***concatenação*** **de** ***x*** **e** ***y*****, com notação** ***x.y*** **ou simplesmente** ***xy*****, é a cadeia formada pelos símbolos de** ***x*** **imediatamente seguidos pelos símbolos de** ***y*****.**

```
Exemplo:
    x = abc     y = def     x.y = abcdef
```


A concatenação de qualquer cadeia x com a cadeia vazia resulta na mesma cadeia ***x*****. Formalmente expressamos** ***x.& = &.x = x*****.**

Se ***x*** **é uma cadeia, qualquer cadeia formada a partir da eliminação de zero ou mais símbolos finais de** ***x*** **é chamada de** ***prefixo*** **(ou** ***cabeça*****) de** ***x*****.**

```
Exemplo:
    x = abcde
    Prefixos de x = &, a, ab, abc, abcd, abcde
```

Do mesmo modo, um ***sufixo*** **(ou** ***rabo*****) de** ***x*** **é qualquer cadeia formada a partir da eliminação de zero ou mais símbolos iniciais de** ***x*****.**

```
Exemplo:
    x = abcde
    Sufixos de x = &, e, de, cde, bcde, abcde
```

Uma ***subcadeia*** **de** ***x*** **é qualquer cadeia formada pela eliminação de um prefixo e um sufixo de** ***x*****.**

Podemos chegar então às conclusões:

1. **para qualquer cadeia *x*, ambos *x* e *&* são prefixos, sufixos e subcadeias de *x*;**

2. **qualquer prefixo ou sufixo de *x* é uma subcadeia de *x*, mas uma subcadeia não é necessariamente um prefixo ou sufixo.**

```
Exemplo:
    x = abcde

    Subcadeia de x = bcd (não é prefixo nem sufixo)
```

Dizemos que uma cadeia *y* é um *prefixo, sufixo ou subcadeia própria* de *x*, se ela é um prefixo, sufixo ou subcadeia de *x* e $y \neq x$.

Conjuntos de cadeias de um alfabeto são denotados usualmente por letras maiúsculas A, B, ..., Z. O *produto AB* de dois conjuntos de cadeias *A* e *B* é definido por

$$AB = \{xy \mid x \in A \text{ e } y \in B\}$$

```
Exemplo:
       A = {a, b}      B = {c, d}

       AB = {ac, ad, bc, bd}
```


Vale para conjuntos de cadeias a regra

$$\{\&\}.A = A.\{\&\} = A$$

Podemos definir agora *potência de cadeias*. Se x é uma cadeia, x^0 é a cadeia vazia, $x^1 = x$, $x^2 = xx$ e, em geral, $x^n = xxx...x$ (n vezes).

Podemos definir também potência de um alfabeto A:

$$A^0 = \{\&\},\ A^1 = A,\ ...,\ A^n = A.A^{n-1} \text{ para } n > 0$$

Chegamos então à definição de *fechamento positivo* e *fechamento* de um conjunto A:

$$A+ = A^1 \cup A^2 \cup ... \cup A^n$$

$$A^* = A^0 \cup A+$$

```
Exemplo:
    A = {a, b}

    A* = {&, a, b, aa, ab, ba, bb, aaa, aab, ...}
```

Note que $A+ = A(A^*) = (A^*)A$.

Às vezes é mais conveniente (e mais claro) escrever

x...	**indicando**	**xy**
z = x...	**indicando**	**x é prefixo (cabeça) de z**
z = ...x	**indicando**	**x é sufixo (rabo) de z**
z = ...x...	**indicando**	**x é subcadeia de z**
z = S...	**indicando**	**S é o primeiro símbolo de z**
z = ...S	**indicando**	**S é o último símbolo de z**
z = ...S...	**indicando**	**o símbolo S aparece em z**

Finalmente podemos definir (genericamente) *linguagem* como sendo qualquer conjunto de cadeias formadas a partir de um alfabeto dado.

Uma *produção* ou *regra de produção* é um par ordenado *(U, x)*, usualmente escrito

U ::= x

onde *U* é um símbolo e *x* é uma cadeia finita não vazia de símbolos. *U* é a parte esquerda e *x* é a parte direita da produção. Nós usaremos a abreviação *regra* para nos referirmos a regra de produção.

Uma gramática *G[Z]* é um conjunto finito não vazio de regras. *Z* é o símbolo que deve aparecer como parte esquerda de pelo menos uma regra. Ele é chamado de *símbolo inicial*. Todos os símbolos usados nas partes esquerdas e direitas das regras formam o vocabulário *V*.

```
Exemplo:
    <número>    ::= <número> <dígito>
    <número>    ::= <dígito>
    <dígito>    ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

    V =     {0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
            <número>, <dígito>}

OBS. usamos a notação Backus-Normal-Form (BNF)
```

Dada uma gramática G, os símbolos que aparecem como parte esquerda de uma regra são chamados *não-terminais* ou *entidades sintáticas*. Eles formam o conjunto de não-terminais *VN*. Os símbolos não presentes em VN são chamados *terminais*. Eles formam o conjunto *VT*. Então, *V = VN* ∪ *VT*.

```
Exemplo: Da gramática do exemplo anterior, temos:

    VN = {<número>, <dígito>}

    VT = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

Agora que temos uma gramática, como indicar a linguagem correspondente à essa gramática? Quais são as sentenças geradas pela gramática? Vamos usar os símbolos => e =>+ para indicar a derivação de sentenças.

Dada a gramática G, dizemos que a cadeia *v produz diretamente* a cadeia *w*,

$$v => w$$

se nós podemos escrever

$$v = xUy, w = xuy$$

para alguma cadeia *x* e *y*, e *U ::= u* é uma regra de G. Nós podemos dizer também que *w* é uma *derivação direta* de *v*, ou que *w reduz-se diretamente* a *v*.

OBS. Como *x* e *y* podem ser a cadeia vazia, para cada regra *U ::= u* de G, nós temos *U => u*.

Dada a gramática G, dizemos que v *produz* w, ou que w *reduz-se* a v,

$$v =>+ w$$

se existe uma seqüência de derivações diretas

$$v = u_0 => u_1 => u_2 => ... => u_n = w$$

com n > 0. Essa seqüência é chamada de *derivação de tamanho n*.

Finalmente, nós escrevemos

$$v =>^* w$$

se v =>+ w ou v = w.

Dada uma gramática G[Z], nós dizemos que uma cadeia *x* é uma *forma sentencial* de G, se *x* é derivada do símbolo inicial Z - indicamos por *Z =>* x*. Uma *sentença* é uma forma sentencial que contém somente símbolos terminais. A linguagem L(G[Z]) é o conjunto de sentenças

$$L(G) = \{x \mid Z =>^* x \text{ e } x \in VT+ \}$$

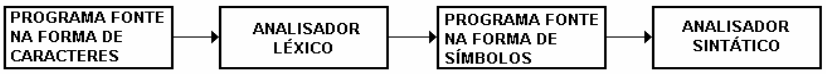
```
Exemplo:
    <número>    ::= <número> <dígito>
    <número>    ::= <dígito>
    <dígito>    ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

    <número>    =>      <dígito>

    <número>    =>+     123<dígito>

    <número>    =>*     <número> <dígito>

    123<dígito>         é forma sentencial

    1234                é sentença
```


ANALISADOR LÉXICO

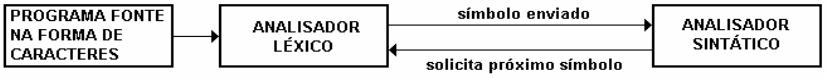
O analisador léxico (*scanner*) é a parte do compilador responsável por ler caracteres do programa fonte e transformá-los em uma representação conveniente para o analisador sintático.

O analisador léxico lê o programa fonte caracter a caracter, agrupando os caracteres lidos para formar os símbolos básicos (*tokens*) da linguagem - identificadores, palavras-chaves, operadores, parentisadores e sinais de pontuação.

O analisador léxico interage diretamente com o analisador sintático de duas possíveis formas, como vemos na figura 2.1.



Esquema 1 - Léxico gera arquivo com todos os símbolos do programa fonte



Esquema 2 - Léxico reconhece símbolo do programa fonte atendendo pedido do sintático

Figura 2.1 - Relação Léxico x Sintático

O esquema 2 é, em geral, melhor porque não necessitamos de que todo o programa fonte seja transformado em uma seqüência de símbolos (expressos por meio de uma codificação) de uma só vez (o que implica na necessidade de área de armazenamento para toda essa informação - imagine um programa fonte de 25.000 linhas).

Os símbolos reconhecidos pelo léxico se enquadram em uma das seguintes classes:

- **Identificadores (do usuário, da linguagem - palavras-chaves, funções embutidas);**
- **Números (inteiros, reais, complexos);**
- **Operadores (aritméticos, relacionais, lógicos);**
- **Parentisadores (abre/fecha aspas, abre/fecha parênteses, abre/fecha colchetes, abre/fecha chaves, BEGIN/END);**
- **Sinais de pontuação (ponto, vírgula, ponto-e-vírgula).**

Vejamos um exemplo. Seja a seguinte gramática:

<ATRIB> ::= IDENT = <E>
<E> ::= <T> + <E> | <T>
<T> ::= <F> * <T> | <F>
<F> ::= <P> ** <F> | <P>
<P> ::= IDENT | (<E>) | NÚMERO

As sentenças geradas por essa gramática são atribuições de expressões aritméticas, parentisadas ou não, envolvendo operações de adição, multiplicação e exponenciação entre operandos denominados genericamente de IDENT e NÚMERO.

Digamos, por exemplo, que IDENT define uma classe de identificadores que poderiam ser A, ALO, B612, C3PO, etc., e que NÚMERO define uma classe de inteiros naturais como 123, 004, 32767, etc.

A função do analisador léxico para a linguagem gerada pela gramática acima seria interpretar as cadeias de caracteres de entrada, agrupa-las e classifica-las como identificador, número, operador +, operador *, operador **, parentisador '(' e parentisador ')'.

Para a sentença A = A + BC ** 32, o léxico poderia produzir:

Chamada	Cadeia Lida	Símbolo	Código
1ª	A	IDENT	1
2ª	=	op =	11
3ª	A	IDENT	1
4ª	+	op +	12
5ª	BC	IDENT	1
6ª	**	op **	14
7ª	32	NÚMERO	2

Onde usamos a seguinte codificação de símbolos:

Símbolo	Código
IDENT	1
NÚMERO	2
=	11
+	12

Símbolo	Código
*	13
**	14
(	21
)	22

DESCRIÇÃO DE SÍMBOLOS COM GRAMÁTICAS REGULARES

Os símbolos de uma linguagem podem ser descritos facilmente através de gramáticas regulares[1]. Para a gramática do exemplo anterior, poderíamos ter:

<identificador> ::= L[2] | L <resto>
<resto> ::= L | D[3] | L <resto> | D <resto>
<número> ::= D | D <número>
<operador> ::= = | + | * | **
<parentisador> ::= (|)

1 Gramática Regular é aquela que tem regras da forma $\alpha ::= \beta$ ou $\alpha ::= \beta\varphi$, com $\alpha, \varphi \in VN$ e $\beta \in VT$.
2 L = Letra.
3 D = Dígito.

DESCRIÇÃO DE SÍMBOLOS COM AUTÔMATOS FINITOS

Outra forma de descrever os símbolos de uma linguagem seria com o uso de autômatos de estados finitos (que têm a vantagem de permitir uma tradução quase direta para código de programação).

Para a gramática do exemplo anterior teríamos os autômatos da figura 2.2.

Obter algoritmos a partir dos autômatos é uma tarefa bastante simples. Vejamos alguns exemplos a seguir.

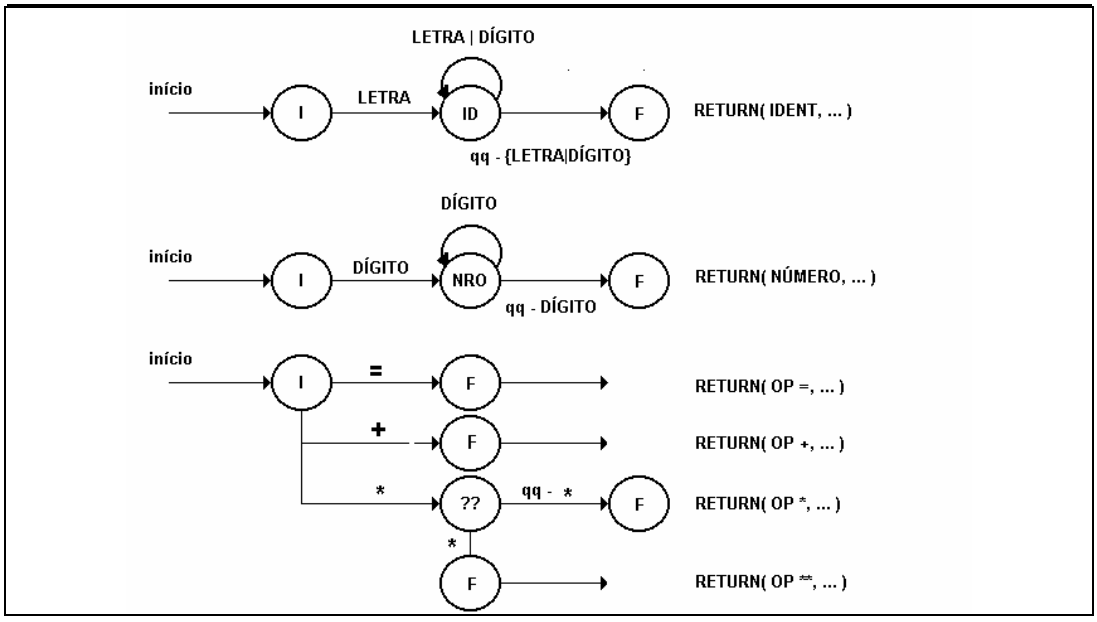


Figura 2.2 - Autômatos para reconhecimento de símbolos

RECONHECIMENTO DE IDENTIFICADOR

```
procedimento identificador;
{
    enquanto caracter é LETRA ou DÍGITO faça {
        símbolo = símbolo || caracter;
        caracter = leia_caracter();
    }

    return( 1, símbolo);
}
```


RECONHECIMENTO DE NÚMERO

```
procedimento número;
{
    enquanto caracter é DÍGITO {
        símbolo = símbolo || caracter;
        caracter = leia_caracter();
    }

    return( 2, símbolo );
}
```

Nas linguagens de programação é comum a existência de identificadores que pertençam à linguagem e têm seu uso reservado para o compilador (facilitando a tarefa de reconhecimento sintático). No procedimento de reconhecimento de identificador, portanto, deve ser feita uma consulta à uma tabela de palavras reservadas para verificar se um identificador reconhecido é uma palavra reservada ou não (é um nome definido pelo programador).

Para implementar o analisador léxico mais facilmente, podemos agrupar os autômatos para reconhecimento de identificadores, números e operadores em um único autômato que seria o próprio analisador. Vejamos isso na figura 2.3.

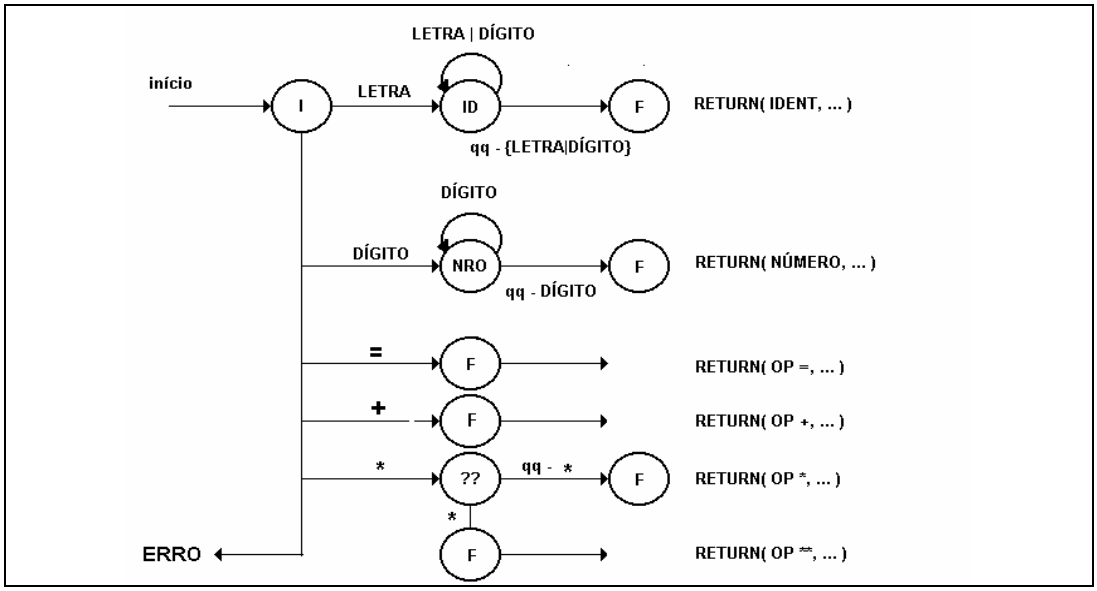


Figura 2.3 - Autômato final do analisador léxico

ROTINAS NECESSÁRIAS

- **leia_caracter(): função que obtém o próximo caracter do programa fonte;**
- **leia_caracter_não_branco(): função que obtém o próximo caracter diferente de branco (espaço, tabulação, mudança de linha, mudança de página) do programa fonte.**

VARIÁVEIS NECESSÁRIAS

- **caracter: armazena o último caracter lido e que está sendo analisado;**
- **símbolo: armazena a cadeia de caracteres que compõem o símbolo sendo formado.**

ALGORITMO

```
procedimento analisador léxico;
{
    obteve_símbolo = FALSE;

    enquanto não obteve_símbolo {
        caracter = leia_caracter_nao_branco();

        símbolo = &;

        se caracter é LETRA então
            identificador();
        senão se caracter é DÍGITO então
            número();
        senão operador();
    }

    return( código, símbolo );
}
```



```
procedimento identificador()
{
    enquanto caracter é LETRA ou DÍGITO {
        símbolo = símbolo || caracter;
        caracter = leia_caracter();
    }

    se tamanho( símbolo ) > MÁXIMO_PERMITIDO {
        escreva advertência;
        considere apenas o MÁXIMO_PERMITIDO;
    }

    se símbolo é palavra reservada então
        código = código_referente_à_palavra;
    senão
        código = código_de_identificador;

    obteve_símbolo = TRUE;
}
```

```
procedimento numero()
{
    enquanto caracter é DÍGITO {
        símbolo = símbolo || caracter;
        caracter = leia_caracter();
    }

    se valor( símbolo ) < MÍNIMO_ACEITO {
        escreva advertência;
        considere apenas o MÍNIMO_ACEITO;
    senão se valor( símbolo ) > MÁXIMO_ACEITO {
        escreva advertência;
        considere apenas o MÁXIMO_ACEITO;
    }

    código = código_de_número;

    obteve_símbolo = TRUE;
}
```



```
procedimento operador()
{
    obteve_símbolo = TRUE;

    se caracter = '+' {
        código = código_de_adição;
    } senão se caracter = '*' {
        caracter = leia_caracter();
        se caracter = '*' {
            código = código_de_exponenciação;
        } senão {
            código = código_de_multiplicação;
        }
    } senão se caracter = '=' {
        código = código_atribuição;
    } senão {
        escreva advertência;
        obteve_símbolo = FALSE;
    }
}
```