COMPILADORES

ANÁLISE LÉXICA

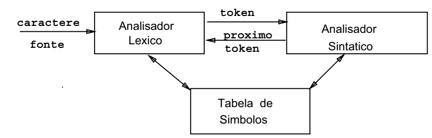
Roberto S. Bigonha e Mariza A.S.Bigonha

UFMG

9 de agosto de 2011

3. Considerações sobre o Analisador Léxico

- Funções adicionais
  - Remoção de brancos, comentários.
  - Indicação de posição de erro.
  - Listagem do fonte.
- Função Principal
  - Reconhecimento de símbolos terminais.
- Relacionamento com o ambiente



Análise Léxica versus Análise Sintática

1. **Simplicidade de projeto é a consideração mais importante.** A separação das análises léxica e sintática normalmente nos permite simplificar pelo menos uma dessas tarefas.

2. **A eficiência do compilador é melhorada.** Um analisador léxico separado nos permite aplicar técnicas especializadas, que servem apenas à tarefa léxica, e não à tarefa de análise sintática.

3. **A portabilidade do compilador é melhorada.** As peculiaridades específicas do dispositivo de entrada podem ser restringidas ao analisador léxico.

Tokens, Padrões e Lexemas

- **Um token** é um par consistindo em **um nome** e **um valor de atributo opcional**. Os nomes de token são os símbolos da entrada que o analisador sintático processa.

  O nome do token é um símbolo abstrato que representa um tipo de unidade léxica: uma palavra-chave, ou uma seqüência de caracteres da entrada denotando um identificador.

- **Um padrão** descrição da forma que os lexemas do token podem assumir.

- **Um lexema** é uma seqüência de caracteres no programa fonte que casa com o padrão para um token e é identificado pelo analisador léxico como uma instância desse token.

· · · Tokens, Padrões e Lexemas

Exemplos de *tokens*:
Para ver como esses conceitos são usados na prática, no comando em C

```
printf("Total = %d\n", score);
```

tanto `printf` **quanto** `score` **são lexemas casando com o padrão para o token** id, **e** `strTotal=%d\n` **é um lexema casando com** literal.

Token	Descrição Informal	Exemplos de Llexemas
if	caracteres `i`, `f`	`if`
else	caracteres `e`, `l`, `s`, `e`	`else`
comparação	$<$ ou $>$ ou $<=$ ou $>=$ ou $==$ ou $!=$	$<=$, $!=$
id	letra seguida por letras e dígitos	`pi`, `score`, `D2`
number	qualquer constante numérica	`3.14159`, `0`, `6.02e23`
literal	qualquer caractere diferente de `""`, cercado por `""`s	`"core dumped"`

· · · Tokens, Padrões e Lexemas

As classes a seguir abrangem a maioria ou todos os tokens:

1. **Um token para cada palavra-chave. O padrão para uma palavra-chave é o mesmo que a própria palavra-chave.**
2. **Tokens para os operadores, seja individualmente ou em classes, como o token** `comparação` **mencionado.**
3. **Um token representando todos os identificadores.**
4. **Um ou mais tokens representando constantes, como números e cadeias literais.**
5. **Tokens para cada símbolo de pontuação, como "(", ")", "," e ";".**

Atributos de Tokens

<tipo, valor>

tipo: **conteúdo sintático.**

valor: **conteúdo semântico.**

Exemplo: (id, **pt)**

(**relop**, ">")

(num,**5)**

Dificuldades com o Reconhecimento de Tokens

Linguagens Sem Palavras Chaves Reservadas:

- **Reconhecimento difícil**
  Exemplo:

PL/1:

IF DO = THEN THEN ELSE = DO
ELSE DO I = 1 TO 30

· · · Dificuldades com o Reconhecimento de Tokens

Linguagens Sem Brancos Separadores

Exemplo Fortran: DO 10 I = 1.5
⇑

DO 10 I = 1,3
⇑

· · · Dificuldades com o Reconhecimento de Tokens

Exemplo: Os nomes de token e valores de atributo associados para a instrução Fortran

```
E = M * C ** 2
```

são escritos como uma seqüência de pares:

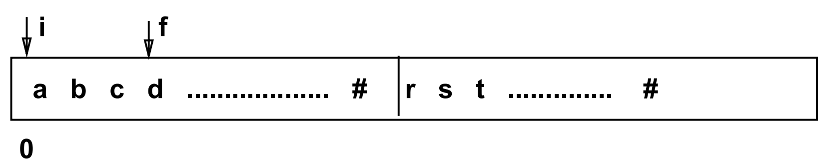
<id, apontador para a entrada da tabela de símbolos de `E`>
<assign_op>
<id, apontador para a entrada da tabela de símbolos de `M`>
<mult_op>
<id, apontador para a entrada da tabela de símbolos de `C`>
<exp_op>
<number, valor inteiro 2>

Erros em Análise Léxica

- Caractere inválido
  - Ignora até caractere válido.
  - Substitui caractere.
- Erro de ortografia.
  - Tenta corrigir.

BUFFERES DE ENTRADA

- Esquema de *BUFFERIZAÇÃO*
- Usado quando necessário olhar vários caracteres à frente.



número de caracteres que cabe em um bloco do disco: 1024, 4096, etc.

Algoritmo para Avanço do Apontador:

```
f := f + 1
if f↑ = # then
begin if f = "fim primeiro bloco"  then
          begin "carrega segundo bloco" ;
                    f := f + 1
          end
          else if f = "fim segundo bloco"  then
                    begin "carrega primeiro bloco" ;
                              f := 0
                    end
                    else "termina"  {eof dentro do buffer}
end
```


3.3 Especificação de Tokens

- Cadeias (strings) e Linguagens

Alfabeto

Um conjunto finito de símbolos ou caracteres.
Exemplo: {0,1} alfabeto binário
ASCII e EBCDIC alfabeto computador

Cadeias (Strings)

Uma seqüência finita de símbolos.
sentença ou palavra ≡ string em teoria de linguagem.
comprimento da cadeia s = | s | − define o número de ocorrencia de símbolos em s.

··· Especificação de Tokens

- ··· Cadeias e Linguagens

Linguagem

Um conjunto de *cadeias* formados a partir de um alfabeto específico.

Exemplos:

$\emptyset$ − conjunto vazio
$\{\mathcal{E}\}$ − conjunto contendo somente a cadeia vazio. { 00,11}, etc...

··· Especificação de Tokens

Concatenação

Se x e y são *cadeias* então: x y ou x.y é o *cadeias* formado juntando-se x e y.
Exemplos: x = dog, y = house, então: xy = doghouse $s\mathcal{E} = \mathcal{E}.s = s$
Olhando a concatenação como "produto" de *cadeias* podemos definir o *cadeia* potência:

Sendo s um *cadeias* define: $s^0 = \mathcal{E}$
Para i > 0 define $S^i = S^{i-1}$s :
como $\mathcal{E}$.s = s, então: s^1 = s
s^2 = ss
s^3 = sss
s^n = ss .....s. n vezes

Termos para Partes da Cadeia

Prefixo de uma cadeia
Seqüência formada desprezando-se zero ou mais caracteres no final de x.
Exemplo: ban é o prefixo de banana.

sufixo de x
Seqüência formada desprezando-se zero ou mais caracteres do inicio de x.
Exemplo: nana é o sufixo de banana.

substring de x
Seqüência obtida quando se apaga um prefixo e um sufixo de x.
Exemplo: nan é um substring de banana.

··· Termos para Partes da Cadeia

prefixo, sufixo ou substring "próprio" de s
qualquer cadeia não vazio x que é um prefixo, um sufixo ou substring de s tal que s $\neq$ x.

subseqüência de x
qualquer *cadeia* formado quando se apaga de x zero ou mais símbolos contíguos desnecessários.

Exemplo: baaa é uma subseqüência de banana.

Operações sobre Linguagens

- **União** L $\cup$ M = { x | x $\in$ L ou x $\in$ M }
  - $\emptyset$L = L$\emptyset$ = $\emptyset$ ("produto")
  - $\emptyset\cup$L = L$\cup\emptyset$ = L ("soma")
- **Concatenação** = L.M = LM = {st|s$\in$L e t$\in$M}
- **Fecho (Closure) (Kleene)** = "qualquer número de"

$$L^\star = \cup_{i=0}^{\infty} L^i$$

= concatenação de L com L, qualquer número de vezes

- **Fechamento Positivo (Positive Closure)** = "uma ou mais vezes"

$$L^+ = L.L^\star = L. \cup_{i=0}^{\infty} L^i = \cup_{i=0}^{\infty} L^{i+1} = \cup_{i=1}^{\infty} L^i$$

Exemplos de Operações com Linguagens

Seja L={A, B, ... Z, a, b, ..., z} e D={0, 1, ..., 9}

1. **União** – L $\cup$ D – é o conjunto de letras e dígitos.
2. **Concatenação:** L.D = LD
   É o conjunto de *strings* consistindo de uma letra seguida por um dígito.
3. L^4 – é o conjunto de todos os *strings* com 4 letras.
4. $L^* = L^0 \cup L^1 \cup L^2$ ... – é o conjunto de todos os *strings* de letras, incluindo o *string* $\mathcal{E}$.
5. L(L $\cup$ D)* – é o conjunto de todos os *strings* de letras e dígitos começando com uma letra. Exemplo: idenficador.
6. $D^+ = D^1 \cup D^2 \cup D^3$ ... – é o conjunto de todos os *strings* com um ou mais dígitos. Exemplo: constante inteira.

Expressão Regular

• Notação usada para definir linguagens simples (que correspondem ao conjunto de símbolos a serem reconhecidos pelo analisador léxico).

• Cada expressão regular r denota uma linguagem L(r).

• Uma expressão regular é construida a partir de expressões regulares simples usando um conjunto de regras.

• As regras especificam como L(r) é formada combinando de várias formas a linguagem especificada pela subexpressão de r.

Regras que Definem uma Expressão Regular Sobre um Alfabeto Σ:

1. $\mathcal{E}$ é uma expressão regular que denota $\{\ \mathcal{E}\}$, ou seja, é o conjunto que contém a *cadeia* vazia.
2. Se a em Σ então a é uma expressão regular e denota $\{$ a $\}$, ou seja, o conjunto contendo a *cadeia* a.
3. Se r e s são expressões regulares, denotando as linguagens L(r) e L(s) então:
   - 3.1. (r) | (s) é uma expressão regular e denota L(r) $\cup$ L(s).
   - 3.2. (r) (s) é uma expressão regular e denota L(r) L(s).
   - 3.3. $(r)^*$ é uma expressão regular e denota $(L(r))^*$.
   - 3.4. (r) é uma expressão regular e denota L(r).Esta regra diz que se desejarmos, pares extras de parênteses podem ser colocados na expressão regular.

· · · Expressão Regular

Parênteses desnecessários podem ser eliminados em uma expressão regular adotando-se as seguintes convenções:

1. Operador unário $\star$ tem a precedência mais alta e se associa à esquerda.
2. Concatenação tem a segunda mais alta precedência e se associa à esquerda.
3. O símbolo | possui a precedência mais baixa e se associa à esquerda.

$$\text{(a) | ((b)*(c))} \equiv \text{a | b* c}$$

Exemplos de Expressões Regulares

Seja Σ = {a, b}:

1. A expressão regular: a|b denota o conjunto {a,b}.

2. A expressão regular (a | b)(a | b) denota {aa, ab, ba, bb }, o conjunto de todos as *cadeias* de a's e b's de comprimento 2.

   Outra expressão regular para este mesmo conjunto é: aa | ab | ba | bb.

3. A expressão regular a* denota {$\mathcal{E}$,a,aa,aaa,$\cdots$}, o conjunto de todos as *cadeias* de zero ou mais a's.

· · · Exemplos de Expressões Regulares

4. A expressão regular (a|b)* denota o conjunto de todas as *cadeias* contendo zero ou mais símbolo a ou b, ou seja, o conjunto de todas as *cadeias* de a's e b's.

   Outra expressão regular para este conjunto é (a*b*)*.

5. A expressão regular a | a*b denota o conjunto contendo a cadeia "a" e todas *cadeias* consistindo de zero ou mais a's seguido de um b.

Se duas expressões regulares r e s denotam a mesma linguagem, r $\equiv$ s e são escritas r = s.
Exemplo: (a | b) = (b | a).

Conjunto Regular

Linguagem descritível por uma expressão regular.

Expressões regulares não descrevem, por exemplo, string com parênteses balanceados ou: $\{\ wcw \mid w$ é string de a's e b's $\}$

Seja as expressões regulares r, s e t. • Propriedades

Axioma	Descrição
$r\|s = s\|r$	| é comutativo
$r\|(s\|t) = (r\|s)\|t$	| é associativo
$r(st) = (rs)t$	concatenação é associativa
$r(s\|t) = rs\|rt;\ (s\|t)r = sr\|tr$	concatenação distribui entre |
$\epsilon r = r\epsilon = r$	$\mathcal{E}$é o elemento identidade para concatenação
$r^* = (r\|\epsilon)^*$	relação entre * e $\mathcal{E}$
$r^{**} = r^*$	* é igual potência

Definição Regular

Seja Σ um alfabeto de símbolos básicos, então uma definição regular é uma seqüência de definições da forma:

$d_1 \rightarrow r_1$
$d_2 \rightarrow r_2$
$\cdots$
$d_n \rightarrow r_n$

onde cada d_i é um nome diferente e cada r_i é uma expressão regular envolvendo símbolos em:

$$\Sigma \cup \{\ d_1,\ d_2,\ ...,\ d_{i-1}\ \}$$

ou seja, os símbolos básicos e os nomes previamente definidos.

Exemplos de Definição Regular

letter $\rightarrow$ [A | B | $\cdots$ | Z | a | $\cdots$ | z]
digit $\rightarrow$ [0 | 1 | 2 | $\cdots$ | 9]

Identificador:

id $\rightarrow$ letter $(letter|digit)^*$

Constantes:

digit $\rightarrow$ [0 | 1 | 2 | $\cdots$ | 9]
digits $\rightarrow$ digit digit*
optional-fraction $\rightarrow$. digits | $\mathcal{E}$
optional-exponent $\rightarrow$ (E (+ | $-$ | $\mathcal{E}$) digits) | $\mathcal{E}$
num $\rightarrow$ digits optional-fraction optional-exponent

Abreviaturas

Certas construções ocorrem com freqüência em expressões regulares, portanto é interessante introduzir uma notação abreviada para elas.

1. uma ou mais vezes – use o operador unário "+" na forma posfixada ≡ "uma ou mais vezes de"

   Exemplo: digits → digit digit*
   $\Downarrow$
   digits → digit+

2. classe de caracteres a notação [abc] onde a, b, e c são símbolos do alfabeto denotam a expressão regular: a | b | c.

   Exemplo : [a – z] → a | b | c | $\cdots$ | z

··· Abreviaturas

3. zero ou uma vez – use o operador unário ? na forma posfixada ≡ "zero ou uma vez de"

   Exemplo 1:
   optional-fraction → . digits | $\mathcal{E}$
   $\Downarrow$
   optional-fraction → (. digits)?

   Exemplo 2:
   optional-exponent → (E(+|−|$\mathcal{E}$)digits) | $\mathcal{E}$
   $\Downarrow$
   optional-exponent → (E (+ | −)? digits)?

3.4 Reconhecimento de Tokens

Dada a gramática:

stmt → if expr then stmt
  | if expr then stmt else stmt | $\mathcal{E}$
expr → term relop term | term
term → id | num

• Terminais: if, then, else, relop, id, num
geram conjuntos *cadeias* pelas seguintes expressões regulares:
if → **if**
then → **then**
else → **else**
relop → < | <= | = | <> | > | >=
id → letter $(letter|digit)$*
num → digits optional-fraction optional-exponent

··· Reconhecimento de Tokens

Assumindo que tokens são separados por brancos e que sua definição regular seja:

delim → branco | tab | newline
stoken → delim+

· · · Reconhecimento de Tokens

Objetivo: construir um Anal.Léxico que isolará o *lexeme* do próximo token no buffer de entrada e produzirá um par <token,valor>

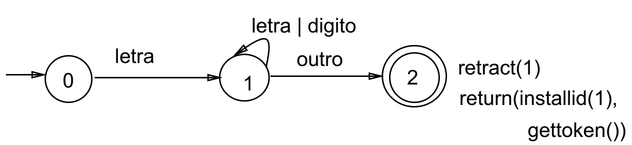
Lexemas	Nome de token	Valor do atributo
stoken		–
if	if	–
then	then	–
else	else	–
Qualquer id	id	apontador para a entrada da T.S.
Qualquer num	num	apontador para entrada de tabela.
<	relop	LT
<=	relop	LE
>	relop	GT
>=	relop	GE
=	relop	EQ
<>=	relop	NE

Diagramas de Transição de Estados

- Passo intermediário
- círculos → estados
- arestas → unem estados
- dois círculos: estado final
- $\xrightarrow{start}$ círculo → estado inicial
- No máximo duas arestas deixam um estado.
- Não há arestas chegando ao estado inicial.
- Não há arestas deixando o estado final.
- outros → qualquer caractere que não apareceu antes nas arestas deixando um estado.

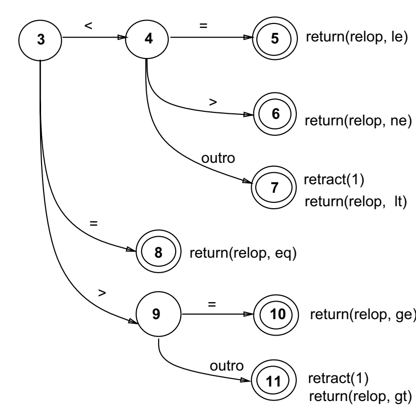
Observação: todo novo estado tem um novo nome.

Diagrama de Transição de Estados



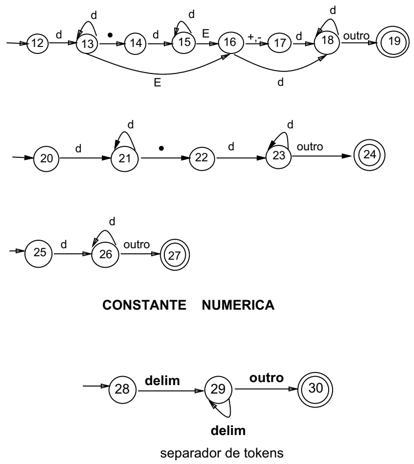
IDENTIFICADOR

· · · Diagramas de Transição de Estados



COMPARADORES

· · · Diagramas de Transição de Estados



Arquitetura de um Analisador Léxico baseado em Diagrama de Transição

- **Tentar juntar diagramas.**
- **Começar pelos símbolos usados mais freqüentemente.**
- **Tratar erros.**
- **Tentar reconhecer** *lexeme* **mais longo quando houver várias alternativas (como em constante numérica no exemplo anterior).**

Implementação

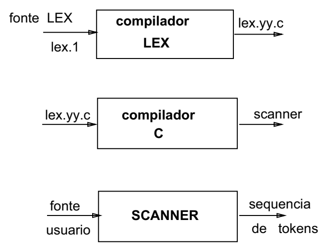
Código para mudar de diagrama (reconhecer outro símbolo).

```
int state = 0; start = 0;
int lexical-value;
int fail ( )
{ forward = token-beginning;
   switch (start)
   { case 0: start = 3; break;
   case 3: start = 12; break;
   case 12: start = 20; break;
   case 20: start = 25; break;
   case 25: recover ( ); break;
   default: /* erro no compilador */ }
   return start
}
```


Analisador Léxico

```
int nexttoken()
{  while(1) { switch(state)
      { case 0: c := nextchar ( );
         if (isletter (c)) state = 1; else state = fail ( ); break
      case 1: c := nextchar ( ); if (isletter (c)) state = 1;
         else if (isdigit (c)) state = 1;
         else state = 2; break
      case 2: retract (1); install-id ( ); return(gettoken ()); · · ·
      case 25: c := nextchar ( );
         if (isdigit (c)) state = 26; else  state = fail ( ); break
      case 26: c := nextchar ( );
         if (isdigit (c)) state = 26; else  state = 27; break
      case 27: retract (1); install-num ( ); return (num); } }
}
```

3.5 Uma Linguagem para Especificação de Analisadores Léxicos



- lex.1 - especificação do analisador léxico.
- lex.yy.c - consiste da representação tabular do diagrama de transição construido a partir das expressões regulares de lex.1 junto com as rotinas que usam a tabela para reconhecer lexemes.
- scanner - é o analisador léxico que transforma a entrada em uma seqüência de tokens.

Programa LEX

Especificação LEX: 3 partes separadas pelos símbolos %%:

1. **Declarações:**
   a. variáveis,
   b. constantes,
   c. definições regulares.
2. **Regras de tradução:** são comandos da forma:
   p_1 { $action_1$ }
   p_2 { $action_2$ } $\cdots$
   p_n { $action_n$ }
   onde cada p_i é uma expressão regular e cada $action_i$ é um trecho de programa descrevendo que ação o analisador léxico deve tomar quando o padrão p_i casa um lexeme.
3. **Procedimentos auxiliares:** procedimentos necessários pelas ações.

Um Programa LEX

% { /* declarações *

LE, LT, GE, GT, if, then, else3, relop, id , num
% }

/* definições regulares */

delim	[\t \n]
stoken	{ delim } +
letter	[A – Z a – z]
digit	[0 – 9]
id	{ letter } ({ letter } | { digit })*
number	{ digit } + (\ . { digit } +) ? (E [+ \-] ? { digit } +) ?

··· Um Programa LEX

%%	/* Regras de Tradução */
{stoken}	{ /* ignora */ }
if	{ *return* (if); }
then	{ *return* (then); }
{id}	{ yylval = *install-id* ( ); *return* (id); }
{number}	{ yylval = *install-num* ( ); *return* (num); }
{ "<" }	{ yylval = LT; *return* (relop); }
{ "≤" }	{ yylval = LE; *return* (relop); } ···

%% /* Procedimentos Auxiliares */

install-id ( ) { /* função para instalar o lexema, cujo primeiro caractere é apontado por *yytext*, e cujo tamanho é *yylval*, na tabela de símbolos, e retorna um apontador para lá */ }

install-num { /* semelhante a install-id, mas coloca constantes numéricas em uma tabela separada */ }

Operador LOOKAHEAD

Em LEX escrevemos um padrão da forma:
r_1/r_2, onde r_1 e r_2 são expressões regulares e significa: identifica r_1 se seguido de r_2

A expressão regular r_2 após o operador lookahead "/" indica o contexto à direita para o reconhecimento.

- Exemplo 1: **reconhecimento da palavra chave** DO: **DO5I = 1.25**
  DO5I = 1,25

DO/({letter} | {digit})* = ({letter} | {digit})*,

- Exemplo 2: **IF (I, J) = 5 : atribuição válida (mas não um comando if)**
- Especificação LEX: **IF /\ (. * /) {letter}**

. → qualquer caractere exceto *newline*.

Exemplo – Analise Léxica

- **Convenções:**

1. Comentários **delimitados por /* e */**
2. Brancos **entre** *tokens* **é opcional exceto que deve haver pelo menos um branco entre identificadores e palavras reservadas.**
3. Operadores de relação (oprel): = ¬ > < <= >=
4. Operadores de adição (opad): + − |
5. Operadores de multiplicação (opmul): ⋆ / &
6. Operador de atribuição: :=
7. Delimitadores: ( ) ; : , ~~b~~ *newline*
8. Eof •

· · · Exemplo – Análise Léxica

9. Identificadores
   id → letter $(letter|digit)^*$
   letter → [A | B | · · · | Z | a | · · · | z]
   digit → [0 | 1 | 2 | · · · | 9]

10. Constantes
   digit → [0 | 1 | 2 | · · · | 9]
   unsigned-integer → digit+
   sign → (+ | −)?
   scale-factor → E sign unsigned-integer
   unsigned-real → unsigned-integer . digit* scale-factor?
   const → unsigned-integer | unsigned-real

11. Palavras chaves (reservadas)
   program, begin, end, integer, boolean, char, type, procedure, value, reference, result, while, do, repeat, until, if, then, else, read, write, case, return, exit.

· · · Exemplo – Análise Léxica

token	tipo	valor
,	$virg(1)	–
;	$pv(2)	–
:	$dp(3)	–
(	$ap(4)	–
)	$fp(5)	–
:=	$atr(6)	–
+	$opad(7)	$mais (1)
—	$opad(7)	$menos (2)
|	$opad(7)	$or (3)
⋆	$opmul (8)	$vezes (1)
/	$opmul (8)	$div (2)
&	$opmul (8)	$and (3)
⇓	⇓	⇓
token≡expr. regular	tipo≡token	valor≡atributo

· · · Exemplo – Análise Léxica

token	tipo	valor
=	$oprel (9)	$igual (1)
<	$oprel (9)	$menor (2)
>	$oprel (9)	$maior (3)
>=	$oprel (9)	$mai (4)
<=	$oprel (9)	$mei (5)
¬=	$oprel (9)	$neg (6)
begin	$begin (13)	–
boolean	$boolean (14)	–
case	$case (15)	–
char	$char (16)	–
do	$do (17)	–
end	$end (18)	–
else	$else (19)	–
⇓	⇓	⇓
token≡expr. regular	tipo≡token	valor≡atributo

· · · Exemplo – Análise Léxica

token	tipo	valor
exit	$exit (20)	–
if	$if (21)	–
id	$id (10)	$ptr para TS
const. inteira	$consti (11)	ptr para tabela
const. real	$constr (12)	ptr para tabela
integer	$integer (22)	–
program	$program (23)	–
procedure	$procedure (24)	–
result	$result (25)	–
repeat	$repeat (26)	–
reference	$reference (27)	–
⇓	⇓	⇓
token≡expr. regular	tipo≡token	valor≡atributo

· · · Exemplo – Análise Léxica

token	tipo	valor
read	$read (28)	–
return	$return (29)	–
type	$type (30)	–
then	$then (31)	–
until	$until (32)	–
value result	$vresult (33)	–
while	$while (34)	–
write	$write (35)	–
•	$eof (36)	
⇓	⇓	⇓
token≡expr. regular	tipo≡token	valor≡atributo

Dicionário

Símbolo	Tipo	Valor
(	$ap	–
)	$fp	–
⋆	$mulop	$vezes
+	$opad	$mais
,	$virg	–
−	$opad	$menos
/	$opmul	$div
|	$opmul	$and
case	$case	–
. . .		
|	$opad	$or
while	$while	–
[	$ac	–
]	$fc	–

• Lookup(A,T,V): pesquisa a tabela Dicionário a procura do símbolo A. Se encontrou retorna em T o tipo do símbolo e em V o valor correspondente. Se não encontrou retorna T = 0.

Subrotina *Getchar* (Classe, CHAR)

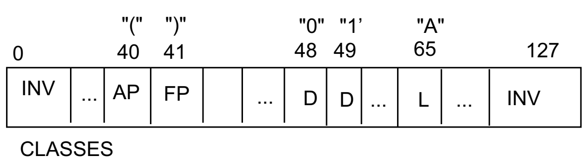
- Retorna em CHAR o próximo caractere no fluxo de entrada e em Classe a classe ou tipo do caractere em CHAR.
- Esta rotina deve ler registros do programa fonte e armazená-los em um buffer.
- Toda vez que o buffer estiver vazio, o caractere NEWLINE e sua classe devem ser retornados e um novo registro lido para o buffer. Sugere-se usar o caractere ASCII LF (line-feed) para codificar NEWLINE.
- Esta rotina poderá ser a responsável pela impressão do programa fonte.

Classificação dos Caracteres

Classe		Caractere(s)
L	(1)	letras
D	(2)	dígitos
$\not{b}$	(3)	branco
•	(4)	'•'
:	(5)	":"
$<$	(6)	'$<$'
$>$	(7)	'$>$'
newline	(8)	newline
DS	(9)	$+ - \star / = , ; ( )$
INV	(10)	demais caracteres

Determinação da Classe de um Caractere

- ASCII
- Use vetor de 128 posições cujos índices são os inteiros correspondentes aos códigos dos caracteres ASCII e cujos conteúdos as classes destes caracteres.



I := inteiro(CHAR)

CLASSE := CLASSES[I]

Ações no Grafo Léxico

Init: • se o caractere correntemente em CHAR for *newline* ou branco, chama *getchar* continuamente até que um caractere diferente de branco ou *newline* seja encontrado.
• A := ' '

GC: *getchar* (classe, char)

Add: A := A *cat* CHAR

Inst: *Install* (A,V)

Ret(x,y): indica o par a ser retornado pelo analisador léxico.

Lookup: *lookup* (A,T,V)

Definições Regulares para o Grafo Léxico 1

letter $\rightarrow$ **[A | B | $\cdots$ | Z | a | $\cdots$ | z]**
digit $\rightarrow$ **[0 | 1 | 2 | $\cdots$ | 9]**

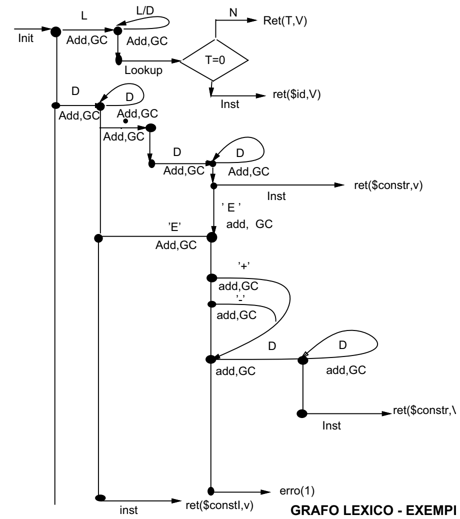
Identificador:

id $\rightarrow$ **letter** $(letter|digit)^*$

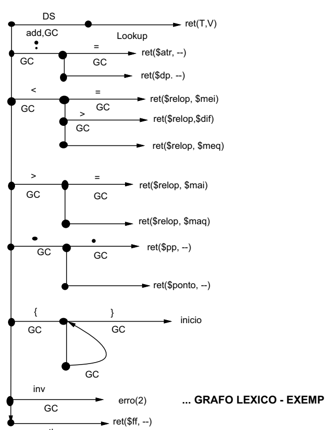
Constantes:

digit $\rightarrow$ **[0 | 1 | 2 | $\cdots$ | 9]**
digits $\rightarrow$ **digit digit***
optional-fraction $\rightarrow$ **. digits** | $\mathcal{E}$
optional-exponent $\rightarrow$ **(E (+ | $-$ | $\mathcal{E}$) digits)** | $\mathcal{E}$
num $\rightarrow$ **digits optional-fraction optional-exponent**

Grafo Léxico para o Exemplo 1



... Grafo Léxico para o Exemplo 1



Convenções Adotadas para a Análise Léxica

- Exemplo 2:

9. Identificadores
   id $\rightarrow$ letter $(letter|digit)^*$
   letter $\rightarrow$ [A | B | $\cdots$ | Z | a | $\cdots$ | z]
   digit $\rightarrow$ [0 | 1 | 2 | $\cdots$ | 9]

10. Constantes
   num $\rightarrow$ digits optional-fraction optional-exponent
   digit $\rightarrow$ [0 | 1 | 2 | $\cdots$ | 9]
   digits $\rightarrow$ digit+
   optional-fraction $\rightarrow$ (. digits)?
   optional-exponent $\rightarrow$ (E (+ | $-$)? digits)?

11. Palavras chaves (reservadas)
   program, begin, end, integer, boolean, char, type, procedure, value, reference, result, while, do, repeat, until, if, then, else, read, write, case, return, exit.

Grafo Léxico para o Exemplo 2

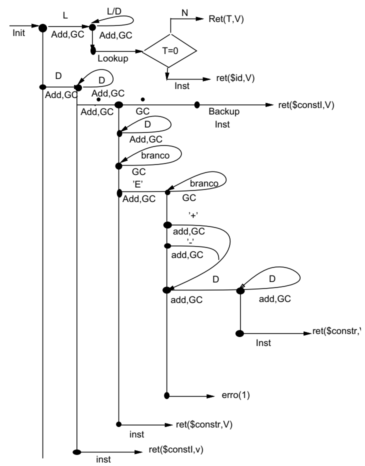
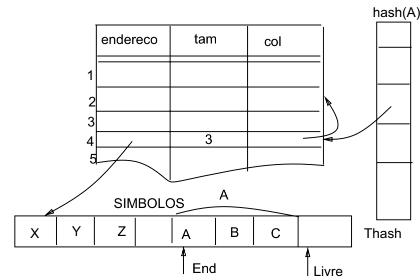


Tabela de Constantes e Identificadores



Add: **Move o caractere em CHAR, ou seja, o caractere retornado pela** *getchar* **, na posição LIVRE de simbolos.**

Install(V): **instala o string A nas tabelas acima caso A ainda não esteja instalado. Se A já estiver, faz** Livre = End; **do contrário** End := Livre. **Retorna endereço de A em Tab.**

Ações Redefinidas

Init: **não precisa fazer A := ' '.**

Add:
SIMBOLOS[Livre] := CHAR
Livre := Livre + 1;
if Livre > Maximo then **erro(3)**

Inst: *Install* **(V)(*)**

Lookup(T,V) **(*)**

(*) O valor de A é obtido da tabela SIMBOLOS, ou seja, nas posições END a Livre -1. Após a pesquisa Lookup faz Livre := END.

Estrutura de Dados Estática

BUFFER - buffer de entrada

PBUF – apontador do próximo caractere no BUFFER.

CLASSE, CHAR

TAB, THASH, SIMBOLOS (global)
END, LIVRE (global)

LINHA - contador de linha (global)

DICIONARIO

CLASSES

- **Inicialmente** PBUF := tamanho do buffer + 1

THASH[0..NHASH] := nada

LINHA := 0

END := LIVRE := primeira posição de SIMBOLOS

CLASSE := branco

CHAR := 'branco'

FIM

3.6 Autômatos Finitos

- Reconhecedor de uma linguagem L:

É um programa que lê um *string* x e responde "sim" se x $\in$ L e "não" se x $\notin$ L.
Reconhecedores de expressões regulares podem ser gerados automáticamente.

- Autômato Finito Determinístico

Apenas uma transição pode deixar um dado estado sob o mesmo símbolo de entrada.

··· Autômatos Finitos

- Autômato Finito Não-determinístico

Mais de uma transição pode deixar um dado estado sob o mesmo símbolo de entrada.

determinísticos: produzem reconhecedores mais rápidos, contudo, são grandes.

não-determinísticos: produzem reconhecedores mais lentos, contudo, são menores.

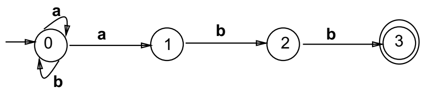
Autômato Finito

Determinístico (DFA) – (K, Σ, δ, q_0, F), onde:
K = conjunto não vazio de estados.
Σ = alfabeto de entrada.
$q_0 \in$ K = estado inicial.
F $\subseteq$ K = conjunto de estados finais.
δ : K x $\Sigma \rightarrow$ K.

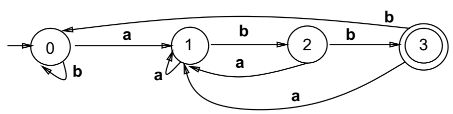
Não-determinístico (NFA) – (K, Σ, δ, q_0, F), onde:
K = conjunto não vazio de estados.
Σ = alfabeto de entrada.
$q_0 \in$ K = estado inicial.
F $\subseteq$ K = conjunto de estados finais.
δ : K x $\Sigma \rightarrow 2^K$
ou δ : K x $(\Sigma \cup \{\mathcal{E}\}) \rightarrow 2^K$

Exemplos

- Autômato não-determinístico para a expressão regular: (a|b)*abb

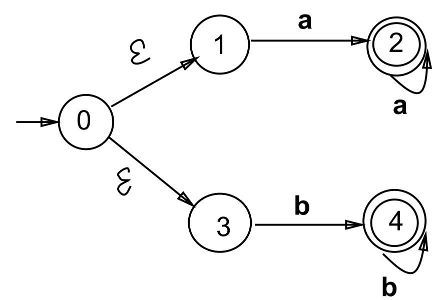


- Automato determinístico para a expressão regular: (a|b)*abb



Exemplos

- Autômato não-determinístico para a expressão regular: aa*|bb*

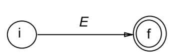


Construção de um NFA a partir de uma Expressão Regular

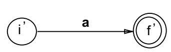
- Dado: Expressão regular R de um alfabeto Σ
- Saída: NFA que aceita linguagem denotada por R.
- Método:

1. Decomponha R em seus componentes primitivos e para cada componente construa um NFA:

a. Para $\mathcal{E}$:



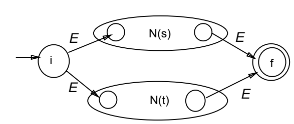
b. Para a $\in \Sigma$:



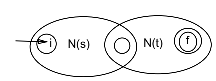
... Construção de um NFA a partir de uma Expressão Regular

2. Combine NFAs de expressões regulares mais simples para construir o NFA de expressão regular compostas. Seja N(s) e N(t) NFAs de expressões regulares s e t. Então:

a. Para a expressão regular s|t construa o NFA N(s|t)

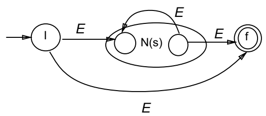


b. Para s.t construa o NFA N(st)

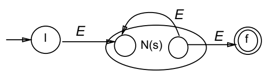


... Construção de um NFA a partir de uma Expressão Regular

c. Para s* construa o NFA N(s*)



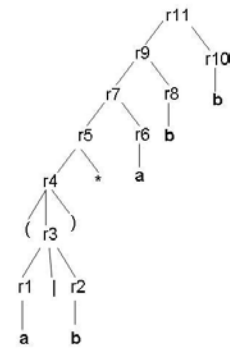
d. Para s+ construa o NFA N(s+)



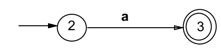
e. Para expressões regulares parentetizadas (s), use N(s) mesmo como NFA.

Exemplo da Construção de N(r) para a Expressão Regular: r = (a|b)*abb

- Árvore de Reconhecimento para r

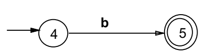


- Para o constituinte r_1, o 1^o a, NFA:

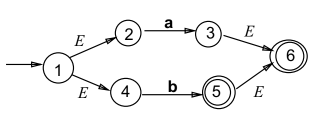


Exemplo da Construção de N(r) para a Expressão Regular: r = (a|b)*abb

- Para r_2 o NFA é:



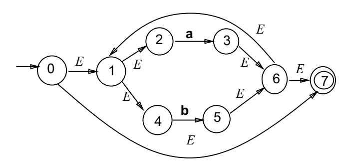
- Combinando N(r_1) e N(r_2) usando a regra de união temos o NFA para $r_3 = r_1 \mid r_2$



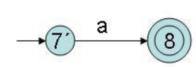
Exemplo da Construção de N(r) para a Expressão Regular: r = (a|b)*abb

- O NFA para (r_3) é o mesmo que para $r_3.r_4$=(r_3). Então o NFA para r_4 = NFA para r_3.

- O NFA para r_5 = r_4*:

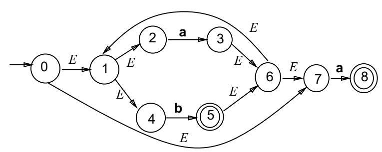


- O NFA para r_6 = a é

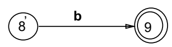


Exemplo da Construção de N(r) para a Expressão Regular: r = (a|b)*abb

- Para obter o autômato para $r_5 r_6$, unimos estados 7 e 7', chamamos o novo estado 7 para obter:

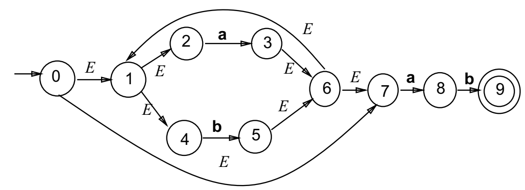


- O NFA para r_8 = b é:

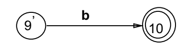


Exemplo da Construção de N(r) para a Expressão Regular: r = (a|b)*abb

- Para obter o autômato para r_9 unimos $r_7.r_8$

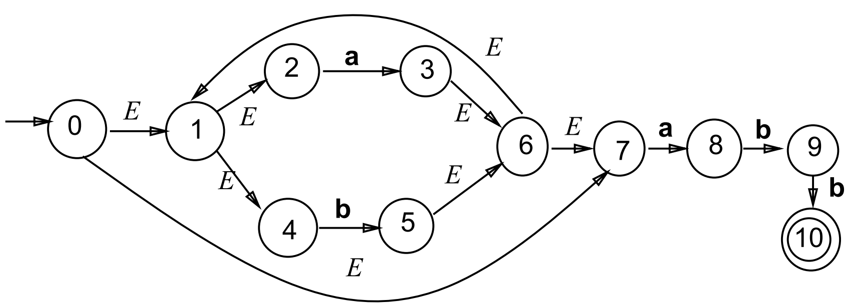


- O NFA para r_{10} = b é:



Exemplo da Construção de N(r) para a Expressão Regular: r = (a|b)*abb

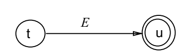
- Para obter o autômato para r_{11} unimos $r_9.r_{10}$



Conversão de um NFA em um DFA

- Dado: Um NFA N
- Saída: um DFA D que aceita a mesma linguagem N.
- Método: "Subset construction"
  S → representa um estado do NFA.
  T → representa um conjunto de estados do NFA.

- Definição 1: $\mathcal{E}$-closure(s) , onde s é um estado:
  1. s é membro de $\mathcal{E}$-closure(s).
  2. Se t está em $\mathcal{E}$-closure(s) e ∃



então u é membro de $\mathcal{E}$-closure(s) .

· · · Conversão de um NFA em um DFA

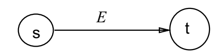
Note que: **$\mathcal{E}$-closure(s) = conjunto de estados que podem ser atingidos partindo-se de s e usando apenas transições $\mathcal{E}$.**

- Definição 2: **$\mathcal{E}$-closure(T) , onde T é um conjunto de estados:**
  $\cup_{i=1}^{N}$ $\mathcal{E}$-closure(s_i) $\forall$ s_i em T.
  N = cardinalidade de T.
  ou seja, conjunto de estados do NFA que são alcançados a partir de algum estado s em T somente com transições $\mathcal{E}$.

- Definição 3: **move (T,a) – conjunto de estados do NFA para os quais existe uma transição sobre o símbolo de entrada "a" a partir de algum estado s em T do NFA.**

Cálculo de $\mathcal{E}$-closure (T):

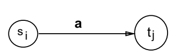
begin
  empilha em STACK todos os estados em T
  $\mathcal{E}$-closure (T) := T
  while STACK não vazia **do**
  begin
    desempilhe s de STACK
    for cada estado t tal que $\exists$



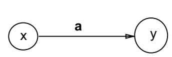
    do if t ainda não está em *$\mathcal{E}$-closure* (T)
      then begin
        inclua t em *$\mathcal{E}$-closure* (T)
        empilha t em STACK
      end
  end
end

Construção de um DFA D dado um NFA N

- Seja S_0 estado inicial de N; então estado inicial de D= *$\mathcal{E}$-closure* (s_0).
- Suponha que os estados de D estejam inicialmente sem marcas.
- **while** $\exists$ estado sem marca x= $\{s_1, s_2, \cdots\}$ em D
  do begin
    marque x;
    for cada símbolo "a" **do begin**
      seja T o conjunto dos estados t_j tais que $\exists$



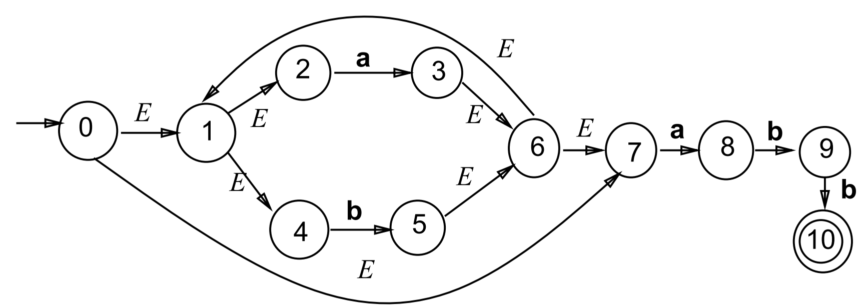
      para algum s_i de x;
      y := *$\mathcal{E}$-closure* (T);
      if y ainda não foi incluído
      then { inclua y sem marca como estado de D; }
        inclua



    end end

Exemplo: Linguagem (a|b)*abb, Σ = {a, b }

- **NFA N:**



Exemplo: Linguagem (a|b)*abb, $\Sigma = \{a, b\}$

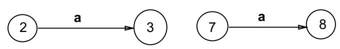
- Estado inicial de D:

A = $\mathcal{E}$-**closure (0)** = $\{0,1,2,4,7\}$

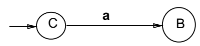
- **D:**

x = A

símbolo a:



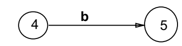
B = **y** = $\mathcal{E}$-**closure** $(\{3,8\}) = \{1,2,3,4,6,7,8\}$ então:



está em D.

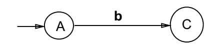
Exemplo: Linguagem (a|b)*abb, $\Sigma = \{a, b\}$

símbolo b:



C = **y** = $\mathcal{E}$-**closure** $(\{5\}) = \{1,2,4,5,6,7\}$

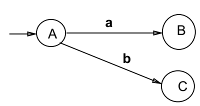
Então:



está em D.

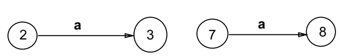
Exemplo: Linguagem (a|b)*abb, $\Sigma = \{a, b\}$

- D:



x = B = $\{1,2,3,4,6,7,8\}$

símbolo a:



y = $\mathcal{E}$-**closure** $(\{3,8\})$ = **B**

Então:

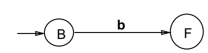
está em D.

Exemplo: Linguagem (a|b)*abb, $\Sigma = \{a, b\}$

símbolo b:

F = **y** = $\mathcal{E}$-**closure** $(\{5,9\}) = \{1,2,4,5,6,7,9\}$

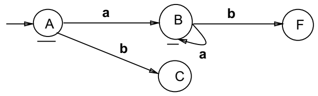
então:



está em D.

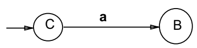
Exemplo: Linguagem (a|b)*abb, $\Sigma = \{a, b\}$

- **D:**



x = C = {1,2,4,5,6,7}
símbolo a:

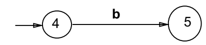
y = $\mathcal{E}$-closure ({3,8}) = B
então:



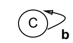
está em D.

Exemplo: Linguagem (a|b)*abb, $\Sigma = \{a, b\}$

Símbolo b:



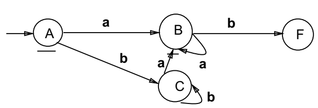
y = $\mathcal{E}$-closure ({5}) = C
Então:



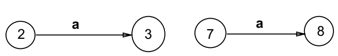
está em D.

Exemplo: Linguagem (a|b)*abb, $\Sigma = \{a, b\}$

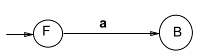
- **D:**



x = F = {1,2,4,5,6,7,9}
símbolo a:

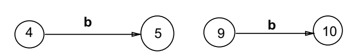


então:

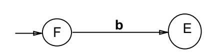


Exemplo: Linguagem (a|b)*abb, $\Sigma = \{a, b\}$

símbolo b:



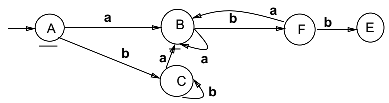
E = y = $\mathcal{E}$-closure ({5,10}) = {1,2,4,5,6,7,10}
então:



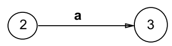
está em D.

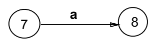
Exemplo: Linguagem (a|b)*abb, $\Sigma = \{a, b\}$

- **D:**

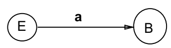


x = E = {1,2,4,5,6,7,10}
símbolo a:



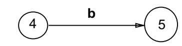


y = $\mathcal{E}$-closure ({3,8}) = B
então:

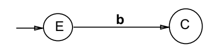


Exemplo: Linguagem (a|b)*abb, $\Sigma = \{a, b\}$

símbolo b:



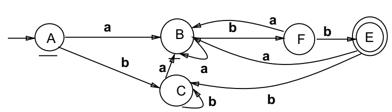
y = $\mathcal{E}$-closure ({5}) = C
então:



está em D.

Exemplo: Linguagem (a|b)*abb, $\Sigma = \{a, b\}$

- **Finalmente D:**



- **E é o estado final de D porque contém o estado 10 de N.**

Minimização do Número de Estados de um DFA

- **Definição: um estado *importante* de um NFA é um estado que tem uma transição $\neq \mathcal{E}$.**
- **Note que no algoritmo da página – são os estados importantes do subconjunto x que determinam os sucessores de x para cada "entrada" .**
- PORTANTO **dois subconjuntos (estados) podem ser identificados como um mesmo estado se:**
  1. **possuem os mesmos estados importantes,**
  2. **ambos incluem ou ambos excluem estados finais do NFA.**