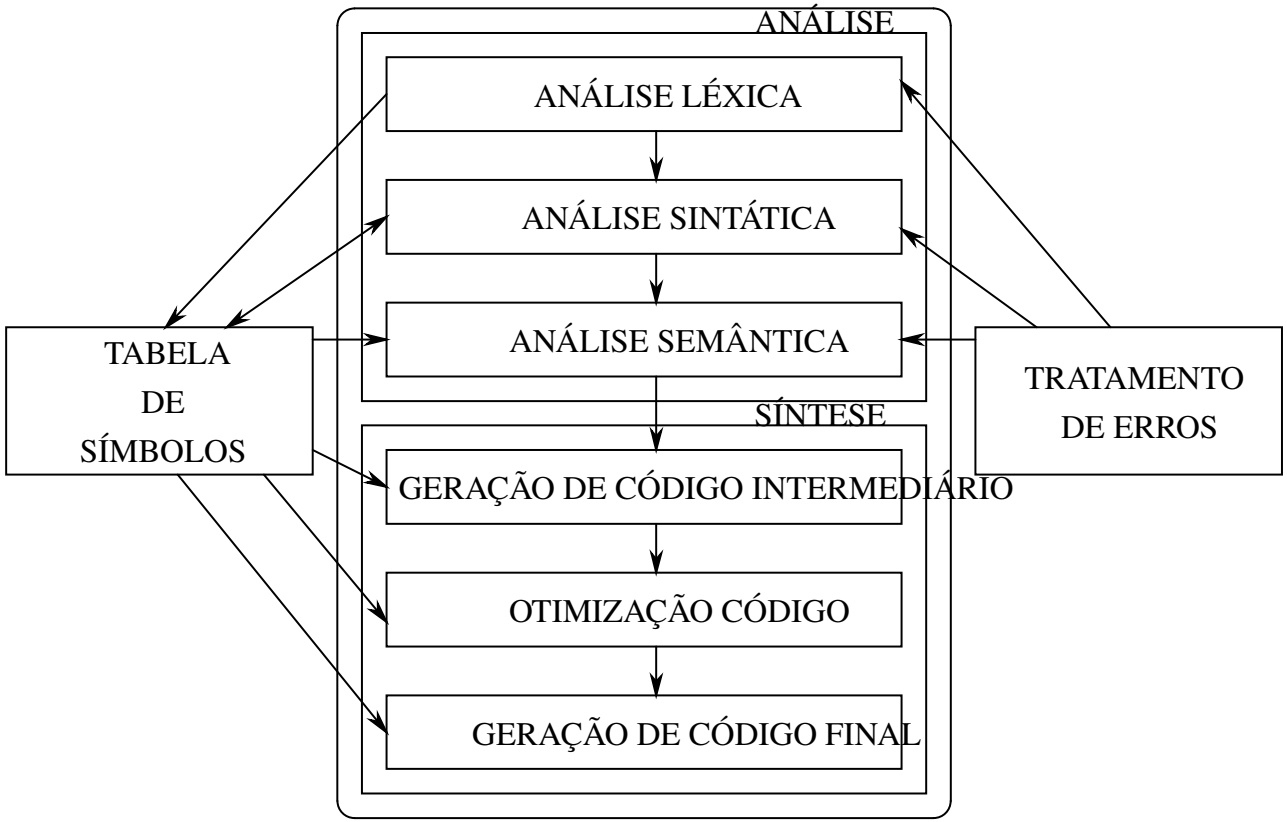
Análise Léxica

Profa. Jerusa Marchi[1]

[1]Departamento de Informática e Estatística
Universidade Federal de Santa Catarina

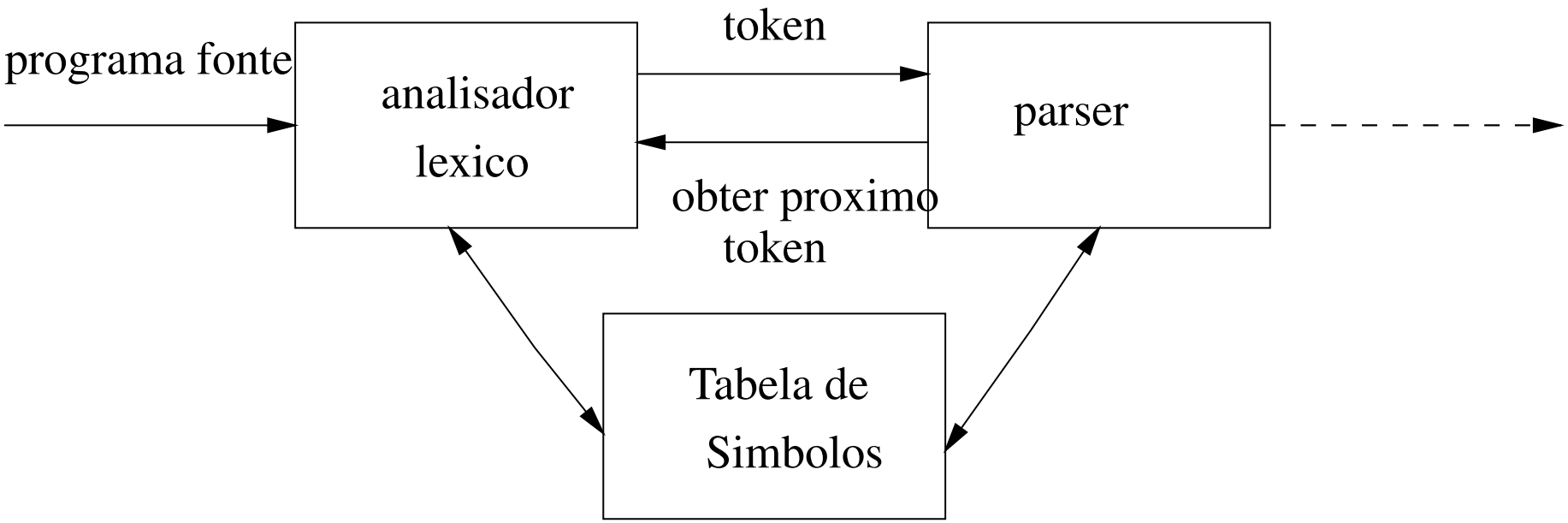
Estrutura Geral do Compilador



Análise Léxica

- Primeira das etapas de análise do compilador
- Principal função:
  - ler os caracteres de entrada do programa fonte e agrupá-los em *lexemas*, produzindo como saída uma sequência de tokens para cada lexema no programa fonte
    - ⋆ O fluxo de tokens é enviado ao analisador sintático para que a análise sintática seja efetuada

Análise Léxica



Analisador Léxico

- Outras funções:
  - preencher a tabela de símbolos com os lexemas (quando o analisador descobre que o lexema é um identificador ou uma constante, por exemplo)
  - remover espaços em branco (e demais marcadores) e comentários
    - ★ pode registrar o número de quebras de linha para associar um número de linha a cada mensagem de erro
  - apontar erros léxicos, associando um número de linha à mensagem de erro
    - ★ Em alguns compiladores, é feita uma cópia do pf com as mensagens de erro inseridas na posição apropriada

Analisador Léxico

- O analisador léxico lida com três conceitos importantes:
  - Token
  - Padrão
  - Lexema

Analisador Léxico

- Token
  - par consistindo em um nome e um valor de atributo opcional.
  - o nome do token é um símbolo abstrato que representa um tipo de unidade léxica (palavra-chave ou identificador, por exemplo)
  - os nomes dos tokens são os símbolos de entrada que o analisador sintático processa

Analisador Léxico

- Padrão
  - descrição da forma que os lexemas de um token podem assumir
    - palavras-chave tem por padrão a sequência de caracteres que a formam
    - identificadores e outros tokens assumem uma estrutura mais complexa, que "casa" com muitas sequências de caracteres

Analisador Léxico

- Lexema
  - sequência de caracteres no programa fonte que "casa" com o padrão para um token e é identificado pelo anallisador léxico como uma instância desse token

Tokens, Padrões e Lexemas

- Em muitas linguagens as classes a seguir abrangem a maioria ou todos os tokens:
  - Um token para cada palavra-chave. O padrão para uma palavra-chave é o mesmo que a própria palavra-chave
  - Tokens para os operadores, individuais ou em classes (<, >, <=, >= podem ser agupados num token `comparison`, por exemplo)
  - Um token representando todos os identificadores
  - Um ou mais tokens representando constantes, como números e literais
  - Tokens para cada símbolo de pontuação, como parênteses esquerdo e direito, vírgula e ponto-e-vírgula

Tokens, Padrões e Lexemas

- Exemplo:

Programa Fonte	Lexemas	Tokens
program exemplo;	program	1 - PR
var A, B : integer;	exemplo	2 - ID
begin	;	3 - SE
(*Inicio do programa*)	var	4 - PR
read (A);	A	5 - ID
B := A + 2.5;	,	6 - SE
...	...	...
	B	16 - ID
	:=	17 - SE
	A	18 - ID
	+	19 - OP
	2.5	20 - Cte Real
	...	...
end.	end	35 - PR
	.	36 - SE

Atributos de Tokens

- Quando mais de um lexema casar com um padrão, o analisador léxico precisa oferecer às fases subsequentes informações adicionais sobre qual foi o lexema em particular
  - O token `comparison` por exemplo, casa com < e com <=, mas é importante para o gerador de código saber qual lexema foi encontrado
- O AL passa para o AS além do token, um valor de atributo que descreve o lexema representado pelo token
- Todas as informações adicionais sobre um determinado lexema são mantidas na Tabela de Símbolos

Erros Léxicos

- É difícil para o AL sozinho saber que existe um erro no código fonte

```
fi (a == f(x)) ...
```

  - não há como o AL perceber que se trata da palavra reservada `if` escrita errada
  - Neste caso, o AL retornará o token **id** para o analisador sintático que deverá tratá-lo

Erros Léxicos

- Acontecem quando nenhum dos padrões para tokens casa com nenhum prefixo da entrada restante
- Estratégia mais simples : "Modo Pânico"de recuperação de erro
  - remover os caracteres seguintes da entrada restante, até que o analisador léxico possa encontrar um token bem formado no início da entrada que resta.

Erros Léxicos

- Outras ações possíveis:
  - Remover um caractere da entrada restante
  - Inserir um caractere que falta na entrada restante
  - Substituir um caractere por outro
  - Transpor dois caracteres adjacentes
- para tal é necessário ver se um prefixo da entrada restante pode ser transformado em um lexema válido por uma única transformação
- Na prática, a maior parte dos erros léxicos envolve um único caractere.

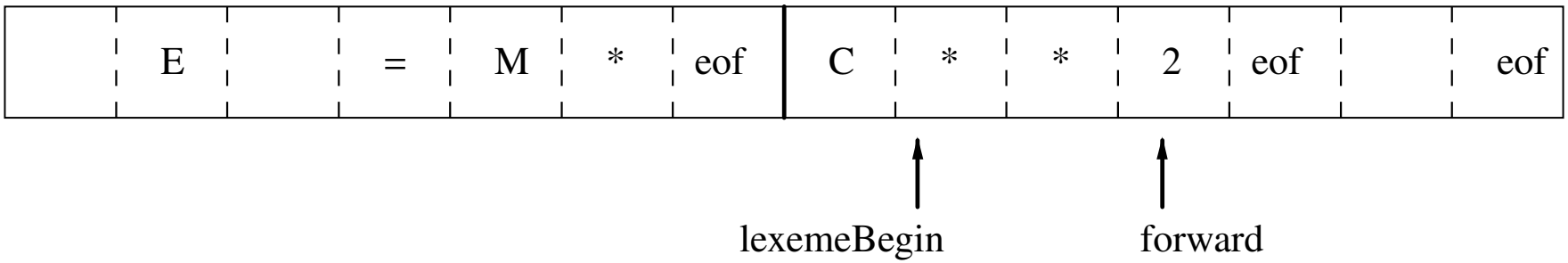
Erros Léxicos

- Uma abordagem mais geral, com encontrar o menor número de transformações necessárias, tende a ser dispendiosa na prática

Bufferes de Entrada

- Em algumas linguagens de programação é comum que alguns operadores sejam representados por dois caracteres, como `==`, `<=`, `>=`, `**`
- Mesmo para identificadores é necessário encontrar algum caractere não válido para se tomar a decisão (ou um espaço em branco)
- É preciso "olhar mais adiante"na entrada para que o analisador léxico possa prover a correta identificação
  - A melhor forma de fazer isso é através do uso de bufferes de entrada com dois apontadores (um para o início e outro para o final de um lexema válido)

Bufferes de Entrada



Bufferes de Entrada

- Solução de conflito: quando vários prefixos da entrada casam com um ou mais padrões
  1. Sempre prefira um prefixo mais longo a um prefixo mais curto;
  2. Se for possível casar o prefixo mais longo com dois ou mais padrões, prefira o padrão listado primeiro na TS (ou no programa de entrada do gerador de analisador léxico)

Especificação de Tokens

- Expressões regulares (e definições regulares) são uma importante notação para especificar os padrões dos lexemas

Especificação de Tokens

- Exemplo de definição regular para identificadores

$$
\begin{array}{lcl}
\mathit{letter\_} & \rightarrow & [A - Za - z\_] \\
\mathit{digit} & \rightarrow & [0 - 9] \\
\mathit{id} & \rightarrow & \mathit{letter\_}(\mathit{letter\_} \mid \mathit{digit})^*
\end{array}
$$

Especificação de Tokens

- Exemplo de definição regular para números sem sinal (inteiros ou ponto flutuante):

$$
\begin{array}{lcl}
\textit{digit} & \rightarrow & [0-9] \\
\textit{digits} & \rightarrow & \textit{digit}^{*} \\
\textit{optionalFraction} & \rightarrow & .\textit{digits} \mid \varepsilon \\
\textit{optionalExponent} & \rightarrow & (E(+ \mid - \mid \varepsilon)\textit{digits}) \mid \varepsilon \\
\textit{number} & \rightarrow & \textit{digits} \\
 & & \textit{optionalFraction} \\
 & & \textit{optionalExponent}
\end{array}
$$

Especificação de Tokens

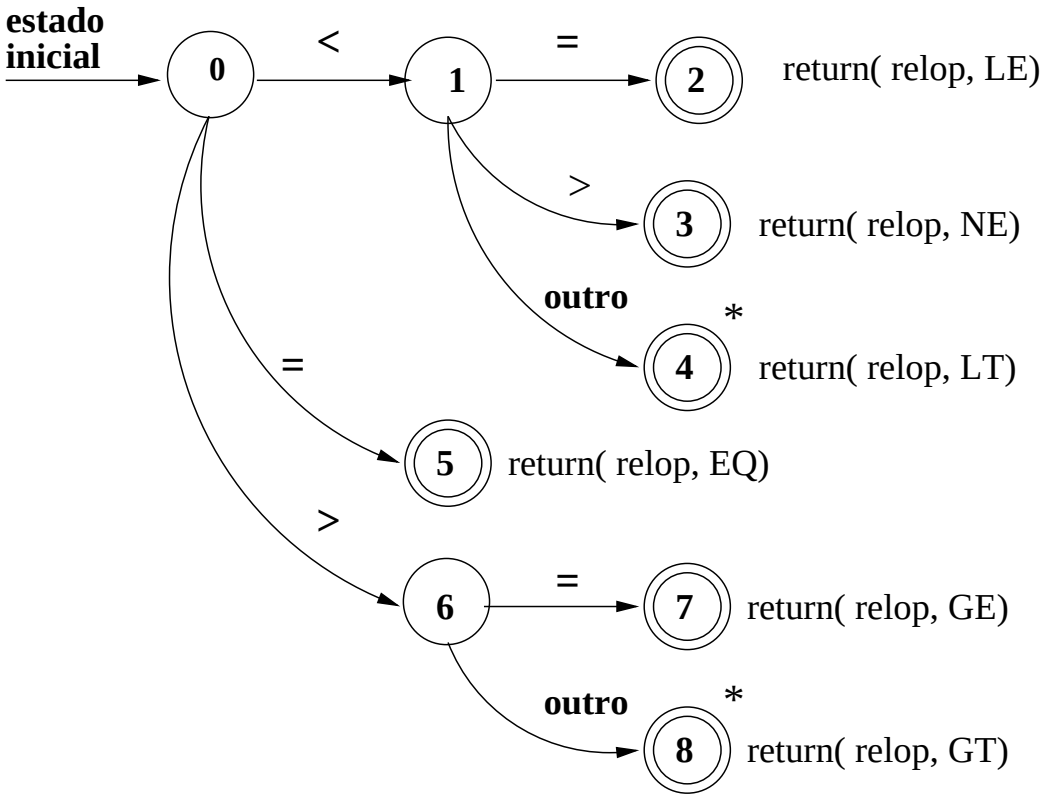
- Exemplo de definição regular para palavras reservadas e operadores relacionais:

$$
\begin{array}{lcl}
\textit{if} & \rightarrow & \texttt{if} \\
\textit{then} & \rightarrow & \texttt{then} \\
\textit{else} & \rightarrow & \texttt{else} \\
\textit{relop} & \rightarrow & \texttt{<|>|<=|>=|=|<>}
\end{array}
$$

Reconhecimento de Tokens

- Expressões regulares podem ser convertidas em Autômatos Finitos Determinísticos que são os mecanismos reconhecedores dos lexemas.

Reconhecimento de Tokens

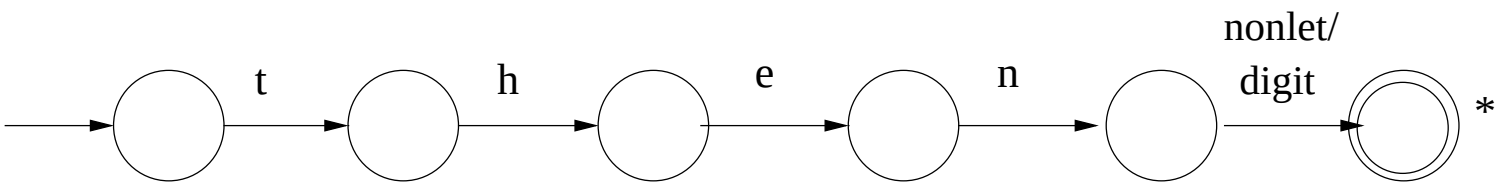


Reconhecimento de Tokens

- Para fazer o reconhecimento de palavras reservadas e identificadores, podemos usar uma de duas abordagens:
  - Instalar inicialmente as palavras reservadas na TS, juntamente com um campo adicional que indique que token essa palavra representa. Ao encontrar um identificador, uma função *InstallID* o coloca na TS, indicando que se trata de um ID. Todo o identificador encontrado que não estiver previamente na TS será um ID, se estiver na TS, então retorna o tipo do token ali registrado

Reconhecimento de Tokens

- Para fazer o reconhecimento de palavras reservadas e identificadores, podemos usar uma de duas abordagens:
  - Criar diagramas para cada palavra-chave isoladamente. Se durante a leitura de entrada, um estado aceitador que identifica tal palavra for alcançado, então o token referente é retornado. Esta abordagem precisa ser apoiada pelo esquema de bufferização, usando o *forward ou lookahead* para "contextualizar" o lexema.



Formas de implementação de um AF

- A implementação do autômato pode ser feita de modo:
  - Implícito - uso de uma estrutura bidimensional que permita o percorrimento dos estados de acordo com a entrada sobre o apontador *forward*
  - Explícito - programação do conceito de estados através de comandos do tipo *swith case*

Formas de implementação de um AF

```
TOKEN getRelop()
{
    TOKEN retToken = new(RELOP);
    while(1) { /* repeat character processing until a return
                  or failure occurs */
        switch(state) {
            case 0: c = nextChar();
                    if ( c == '<' ) state = 1;
                    else if ( c == '=' ) state = 5;
                    else if ( c == '>' ) state = 6;
                    else fail(); /* lexeme is not a relop */
                    break;
            case 1: ...
            ...
            case 8: retract();
                    retToken.attribute = GT;
                    return(retToken);
        }
    }
}
```

Construção de um AL baseado em AF

- Para construir um AL baseado no uso de Autômatos, a técnica mais usada consiste em combinar todos os autômatos que representam tokens da linguagem em um único autômato:
  - Cria-se um AF para cada padrão que define um token
  - Os AF devem ser determinísticos e mínimos
  - Une-se os AF por meio de transições ε
  - Determiniza-se o AF resultante

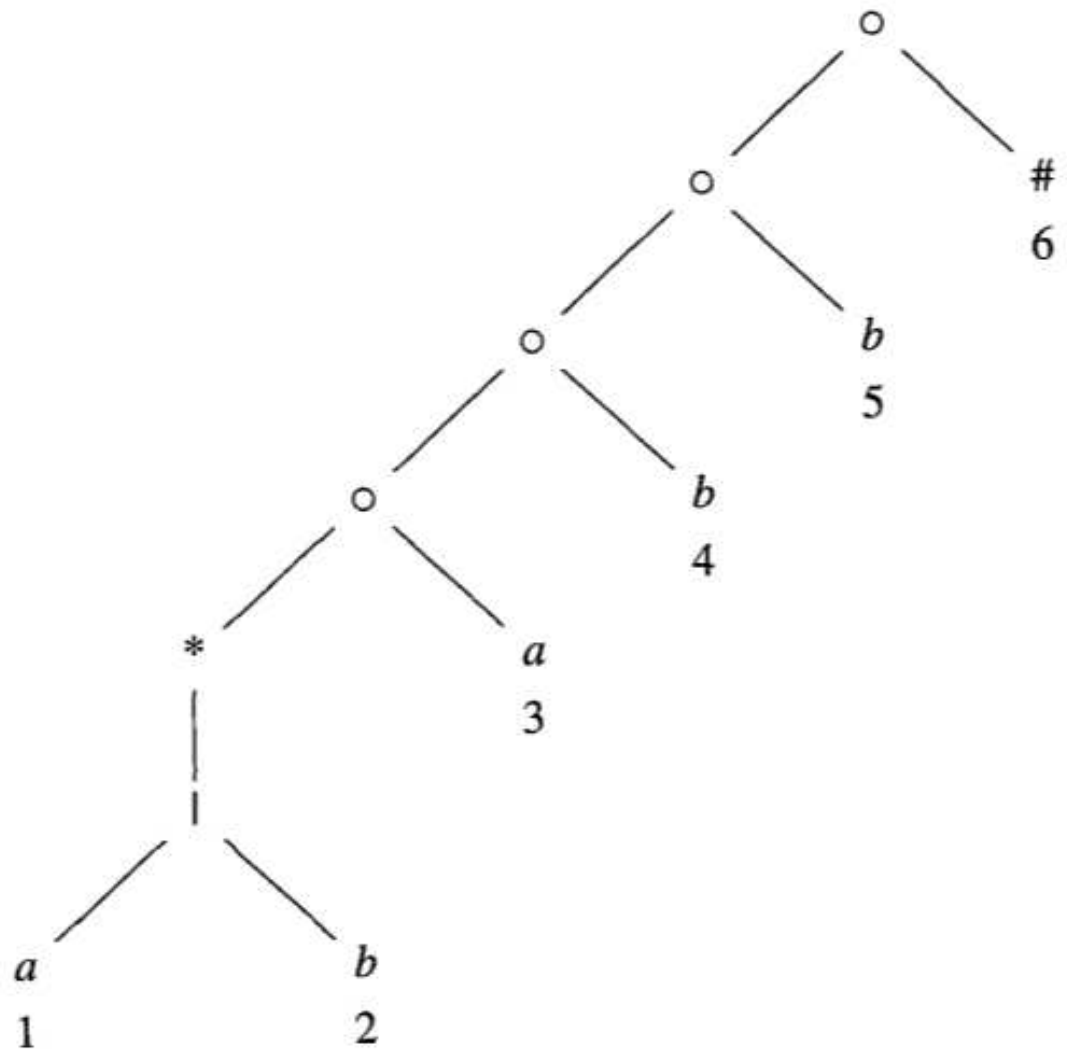
Construção de um AL baseado em AF

- Geradores de Analisadores Léxicos partem de definições regulares para construir AF
- A conversão de um AFND para um AFD pode gerar um número exponencial de estados, cujo tempo de execução pode ser afetado significativamente se houver a necessidade de fazer *swap*
  - Converter uma ER para um AFD diretamente

De ER para AFD

- 1) Constrói-se a árvore de sintaxe estendida para a expressão regula em questão (usa-se o símbolo $\#$ ao término da expressão $(r)\#$)
  - $(a \mid b)^* abb\#$

De ER para AFD



De ER para AFD

- 2) Calculam-se as funções *nullable*, *firstpos*, *lastpos*

De ER para AFD

- *nullable(n)* - é verdadeira para um nó *n* da árvore sintática sse a subexpressão representada por *n* tiver ε em sua linguagem (pode-se tornar a subexpressão nula)
  - Na expressão $(a \mid b)^* a$ a qual referes-se o nó-concatenação da árvore sintática para $(a \mid b)^* abb\#$, o *nullable* é falso. Mas para o nó-asterisco logo abaixo, *nullable* é true.

De ER para AFD

- *firstpos(n)* - é o conjunto de posições na subárvore com raiz em *n* que corresponde ao primeiro símbolo de pelo menos uma cadeia na linguagem da subexpressão cuja raiz é *n*
  - Na expressão $(a \mid b)^* a$ *firstpos(n)*$= \{1, 2, 3\}$, pois para a cadeia *aa*, usam-se os nós 1 e 3 e para *ba* usam-se os nós 2 e 3. Se considerarmos como cadeia somente *a*, este *a* vem da posição 3

De ER para AFD

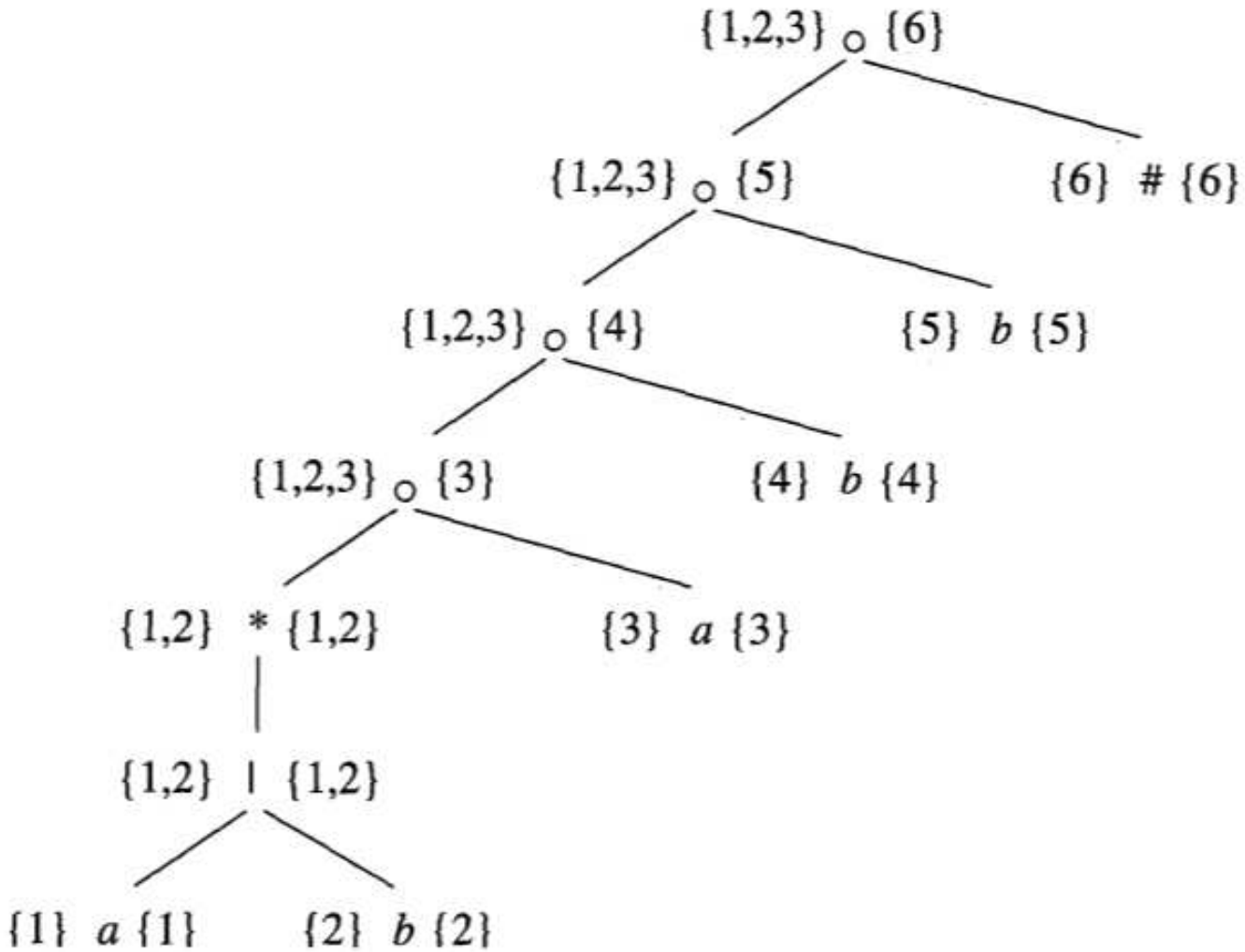
- *lastpos(n)* - é o conjunto de posições na subárvore com raiz em *n* que corresponde ao último símbolo de pelo menos uma cadeia na linguagem da subexpressão cuja raiz é *n*
  - Na expressão $(a \mid b)^* a$ *lastpos(n)*$= \{3\}$, pois não importa a cadeia gerada no nó *n*, o último símbolo sempre será *a* da posição 3 da árvore

De ER para AFD

NODE n	$nullable(n)$	$firstpos(n)$
A leaf labeled ϵ	**true**	$\emptyset$
A leaf with position i	**false**	$\{i\}$
An or-node $n = c_1 \mid c_2$	$nullable(c_1)$ **or** $nullable(c_2)$	$firstpos(c_1) \cup firstpos(c_2)$
A cat-node $n = c_1 c_2$	$nullable(c_1)$ **and** $nullable(c_2)$	**if** ($nullable(c_1)$) $firstpos(c_1) \cup firstpos(c_2)$ **else** $firstpos(c_1)$
A star-node $n = c_1{}^*$	**true**	$firstpos(c_1)$

- as regras para *lastpos* são as mesmas que para *firstpos*, porém invertendo-se os filhos c_1 e c_2 na regra de um nó-concatenação

De ER para AFD



De ER para AFD

- 3) Calcula-se a função *followpos*
  - Para uma posição p, é o conjunto de posições q em um árvore sintática completa tal que existe alguma cadeia $x = a_1 a_2 \cdots a_n$ em $L((r)\#)$ tal que para algum i existe um meio de explicar a inclusão de x como membro de $L((r)\#)$ casando a_i com a posição p da árvore sintática e a_{i+1} com a posição q

De ER para AFD

- Existem apenas duas maneiras pelas quais uma posição de uma ER pode ser criada para vir após outra:
  - Se n é um nó-concatenação com filho esquerdo c_1 e filho direito c_2, então para cada posição i em $lastpos(c_1)$, todas as posições em $firstpos(c_2)$ estão em $followpos(i)$
  - Se n é um nó-asterisco e i é uma posição em $lastpos(n)$ então todas as posições em $firstpos(n)$ estão em $followpos(i)$

De ER para AFD

- Para calcular *followpos* para a árvore sintática da expressão $(a \mid b)^* abb$:
  - Olhamos o nó-concatenação e colocamos cada posição em *firstpos* do seu filho direito no *followpos* para cada posição em *lastpos* do seu filho esquerdo
    - ★ para o nó mais baixo, a posição 3 está em *followpos*(1) e *followpos*(2).
    - ★ para o próximo nó, 4 está em *followpos*(3) e os dois nós restantes colocam 5 em *followpos*(4) e 6 em *followpos*(5)
  - A regra 2 considera o nó-asterisco, onde as posições 1 e 2 estão em ambos *followpos*(1) e *followpos*(2), pois tanto *firstpos* quanto *lastpos* para esse nó são $\{1, 2\}$

De ER para AFD

NODE n	$followpos(n)$
1	$\{1, 2, 3\}$
2	$\{1, 2, 3\}$
3	$\{4\}$
4	$\{5\}$
5	$\{6\}$
6	$\emptyset$

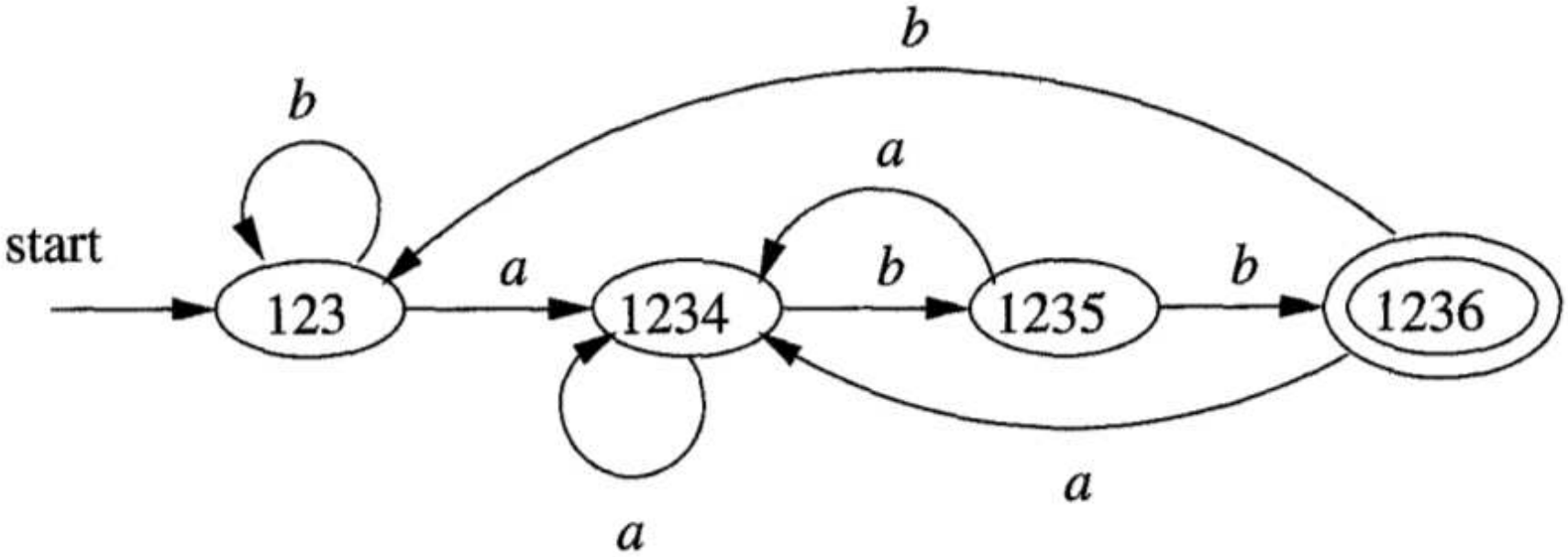
De ER para AFD

- 5) Construa *DStates* o conjunto de estados do AFD *D* e *Dtran* a função de transição para *D*, como segue:

De ER para AFD

initialize *Dstates* to contain only the unmarked state $firstpos(n_0)$,
    where n_0 is the root of syntax tree T for $(r)\#$;
while (there is an unmarked state S in *Dstates*) {
    mark S;
    for (each input symbol a) {
        let U be the union of $followpos(p)$ for all p
            in S that correspond to a;
        if (U is not in *Dstates*)
            add U as an unmarked state to *Dstates*;
        $Dtran[S, a] = U$;
    }
}

De ER para AFD



De ER para AFD

- Exemplos:

$$a(a \mid b)^* a$$

$$aa^*(bb^*aa^*)^*$$