CENTRO UNIVERSITÁRIO
UNI METROCAMP | WYDEN

Análise Léxica: técnicas de implementação

Abordagens

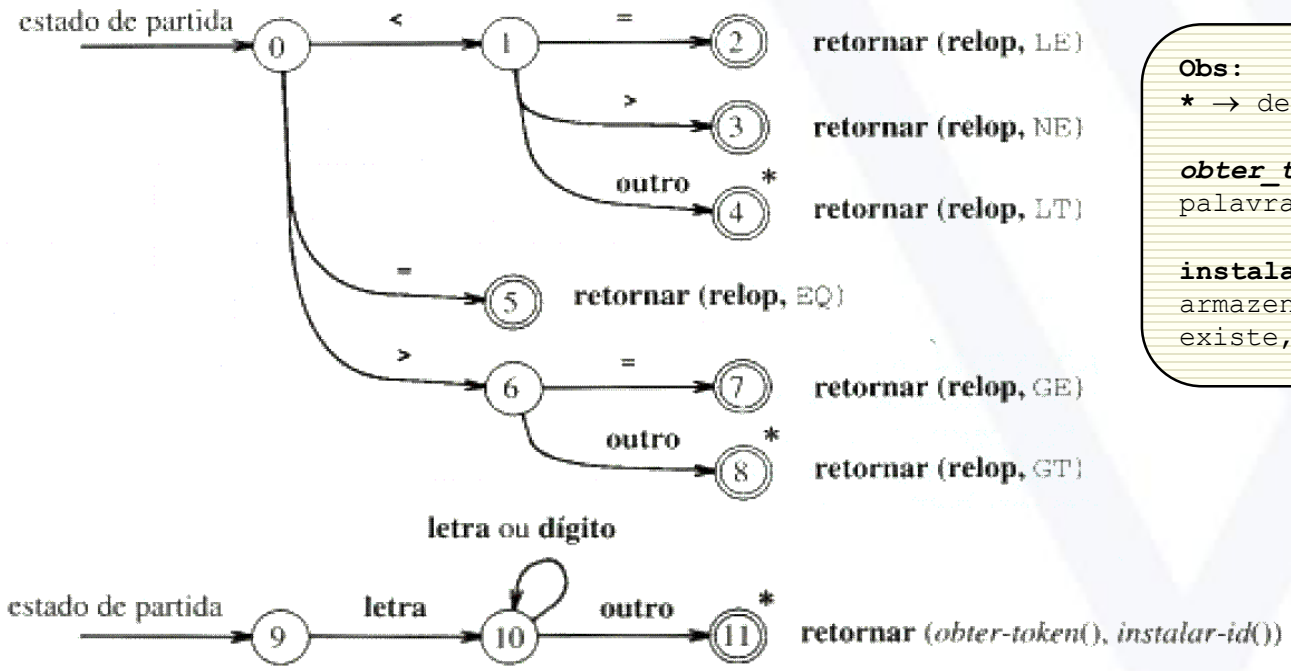
- Três procedimentos comuns:
  1. Uso de ferramenta de geração de Analisadores Léxicos (recebe como entrada uma especificação dos padrões dos *tokens*)
  2. Uso de linguagem de alto nível (C/C++, Java, Pascal)
  3. Uso de linguagem *Assembly*, com acesso às rotinas do SO para fazer E/S

- Facilidade de Escrita: 1 > 2 > 3
- Eficiência/desempenho: 3 > 2 > 1 (geralmente)

- Abordagem do curso: linguagens de alto nível (2.)

Com Linguagem de Alto Nível

- Estratégia:
  1. Passo: Identificar os *tokens* e definir padrões p/ lexemas
  2. Passo: Construir um diagrama de transição de estados para reconhecimento de *tokens* (autômato finito)
  3. Passo: Implementar o autômato finito numa linguagem de alto nível



Obs:
`*` → devolve último caractere lido

obter_token() → retorna o *token* da palavra chave ou **Id**

instalar_id() → retorna posição de armazenagem do lexema. Se ainda não existe, é inserido

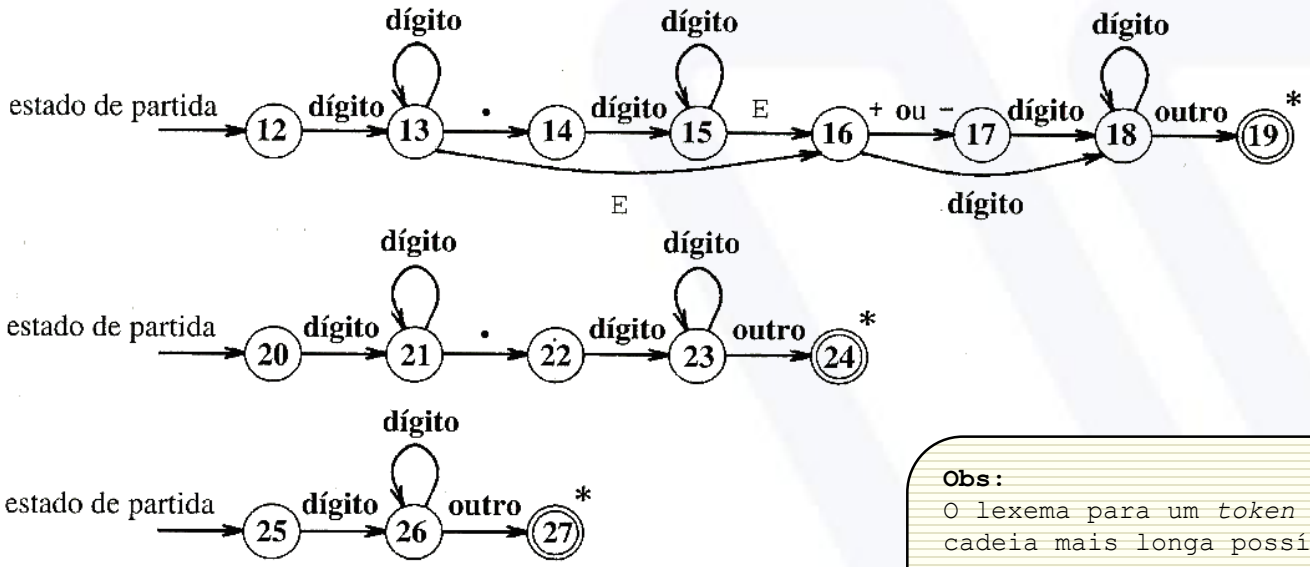
Características

- Quando o reconhecimento de um novo *token* é iniciado, o controle deve estar no estado de partida do diagrama
- Pode haver vários diagramas de transições, cada um especificando um grupo de *tokens*
- Falha: ponteiro de leitura deve retroceder até o ponto onde estava, quando o fluxo estava no estado de partida do diagrama.
  - Neste caso, um novo diagrama deve ser ativado
  - Um erro léxico só deve ser reportado quando a falha ocorrer para todos os diagramas.
- Obs: em analisadores Léxicos implementados manualmente, deve-se adotar a técnica de carregar a Tabela de Símbolos com as palavras-chave, e processá-las como identificadores, no momento da leitura

Exemplo

- Considere o reconhecimento de números sem sinal:

  `num ::= [0-9]+(\.[0-9]+)?(E(+|-)?[0-9]+)?`



Obs:
O lexema para um *token* deve sempre ser a cadeia mais longa possível.

Por isso, a ordem que os diagramas devem ser processados importa: 12, 20, 25

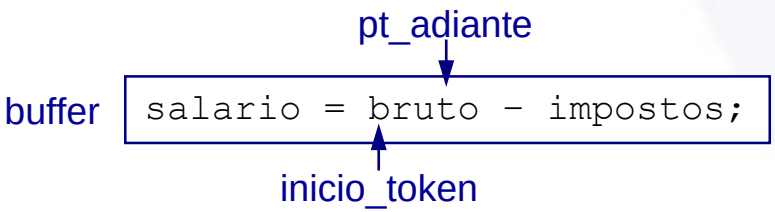
Ex: reconheça o número **12.3E4** usando a ordem inversa de diagramas

Controle de Processamento

- Como vimos anteriormente, a implementação de diagramas de transição deve ser cuidadosa
  - Garantir que os possíveis estados de partida serão executados na ordem correta.

- Cada vez que ocorre uma falha no processamento de um diagrama, temos que reiniciar o processo para o diagrama seguinte

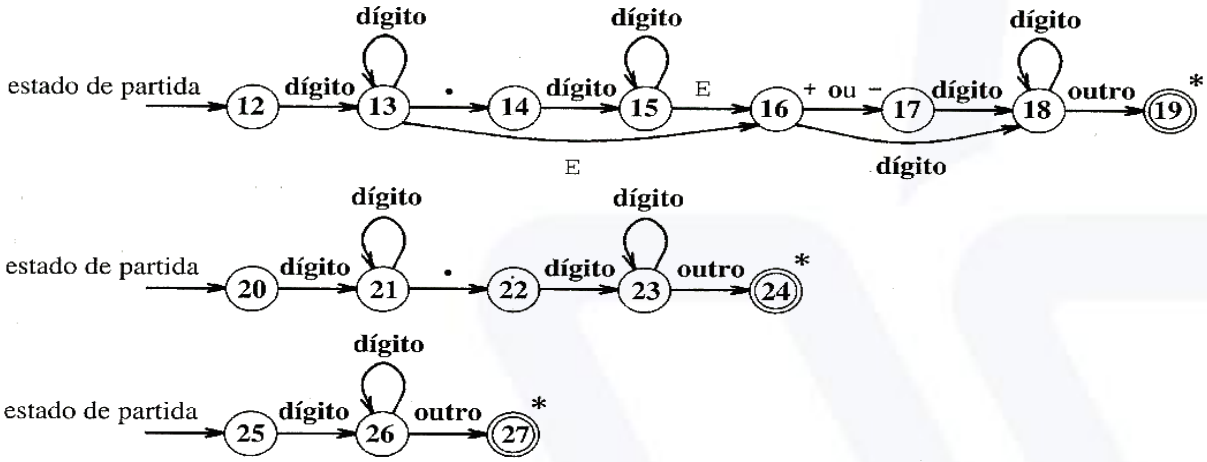
- Controle através de ponteiro de início de lexema e ponteiro adiante:



Obs:

- **pt_adiante** avança sempre que um novo caractere é solicitado.
- O caractere devolvido deve provocar uma transição no diagrama processado
- Caso não haja uma transição possível, o analisador deve **falhar** o processamento do diagrama em questão

Controle de Processamento



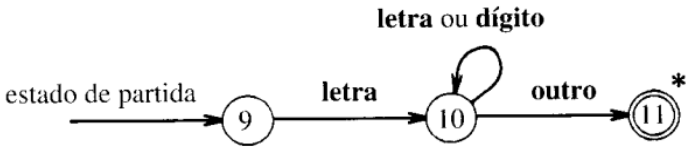
• Um erro léxico só deve ser levantado se o processo falhar para todos os diagramas.
• Ordem de processamento: 12, 20, 25
• Cada vez que uma falha ocorre, chamamos o seguinte método:

```
1.  int falhar(int partida) {
2.    pt_adiante = inicio_token;
3.    switch(partida) {
4.       case 12: partida = 20; break;
5.       case 20: pardida = 25; break;
6.       default:
7.          /* Erro léxico */
8.    }
9.    return partida;
10. }
```

Obs:
• o parâmetro **`partida`** representa o estado de partida do diagrama atualmente em processamento.

• quando uma falha ocorre, o ponteiro adiante é recuado até o início do token (linha 2)

Máquina de Estados



- Possível implementação:

```
1.  Token nextToken() {
2.    while(1) {
3.       switch(state) {
4.          case 0: c = nextChar();
5.                  ...
6.          case 9: c = nextChar();
7.            if(isLetter(c)) state = 10;
8.            else state = falhar(9);
9.            break;
10.         case 10: c = nextChar();
11.           if(isLetter(c)) state = 10;
12.           else if(isDigit(c))state = 10;
13.                else state = 11;
14.           break;
15.         case 11: retrair(1);
16.            instalarId();
17.            return (obter_token());
18.         ...
19.      }
20.   }
21. }
```

Especificação: exemplo

Reconhecimento de tokens:

Considere uma linguagem com os seguintes tokens

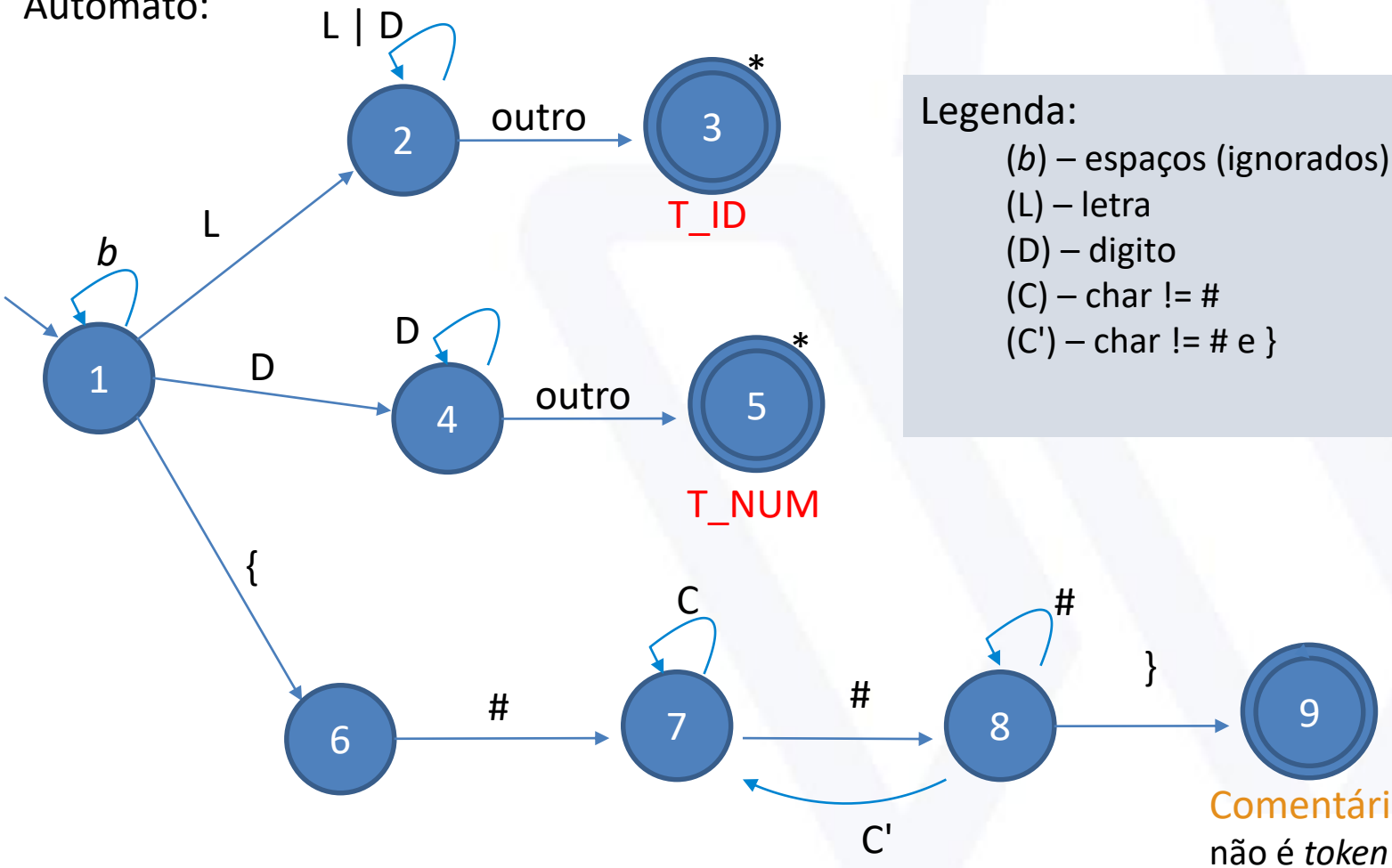
-**Identificadores:**
- Formados por uma letra seguida, opcionalmente, por uma ou mais letras e/ou digitos

-**Numeros inteiros:**
- Formados por um ou mais digitos

- **Comentarios:**
- Delimitados por {# e #}

Especificação: exemplo

Autômato:



Legenda:

(*b*) – espaços (ignorados)
(L) – letra
(D) – digito
(C) – char != #
(C') – char != # e }

Comentário:
não é *token*

Implementação: (parcial)

```
let: set of (a .. z);
dig: set of (0 .. 9);

ident := null;
num := null;

begin
 c := getchar;
 while c = ' ' do c := getchar;
 if c in let then
   do
     ident = ident || car
     c := getchar;
   while c in (let or dig);
   retchar;
   return Token[T_ID, ident];
 else
    if c in digitos then
    do
      num := num || car
      c := getchar;
    while c in dig;
    retchar;
    return Token[T_NUM, num];
 ...
```

Exercício (lab):

- Construir uma função em JAVA que implemente o AFD relativo à linguagem:

  L = {w ∈ {a,b}* | w contém **aa** ou **bb** como subpalavra}