Linguagens de Programação e Compiladores

Analisador Léxico

José Osvano da Silva, PMP

Sumário

- Análise Léxica
- Reconhecimento de Tokens
- Diagramas de Transição
- Autômatos Finitos
- Identificando tokens
- Tarefas Secundárias do AL
- Erros Léxicos
- Tratamento de Constantes
- Especificação e Reconhecimento dos tokens
- Códigos dos Tokens e Processo de reconhecimento
- Palavras Reservadas x Identificadores
- Alocação de espaço para identificadores (e de tokens em geral)
- Formas de Implementação da Análise Léxica
- Exercícios

Análise Léxica

Analisador Léxico:

- 1ª fase de um compilador
- lê os caracteres de entrada e produz uma sequência de símbolos léxicos válidos (tokens)
- pode ser implementado como uma subrotina do *parser* ou uma co-rotina do *parser*;
- executa tarefas secundárias, como:
  - remover comentários;
  - remover espaços em branco
  - controlar posição dos elementos, visando mensagens de erros ao programador

Análise Léxica

Tokens:

- Elementos dados: elementos básicos de qualquer ling. de programação:
  - numéricos: inteiros, reais, complexos etc
  - lógicos: true, false
  - caracteres: “C”, “a”, “casa”
  - ponteiros
- Identificadores / nomes: variáveis dos procedimentos, associadas a um dado nome:
  - Nome { valor
    atributo

Análise Léxica

Tokens:

- Estruturas de dados
  - arrays
  - record
  - listas
  - pilhas
- Operadores aritméticos:
  - Unários:
    - pré-fixados
    - pós-fixados
  - Binários

Reconhecimento de Tokens

Diagramas de Transição:

- passo intermediário;
- apresentam ações executadas pelo Analisador Léxico;
- controla as informações a respeito de caracteres que são examinados, com a leitura do arquivo fonte.

Elementos dos Diagramas:

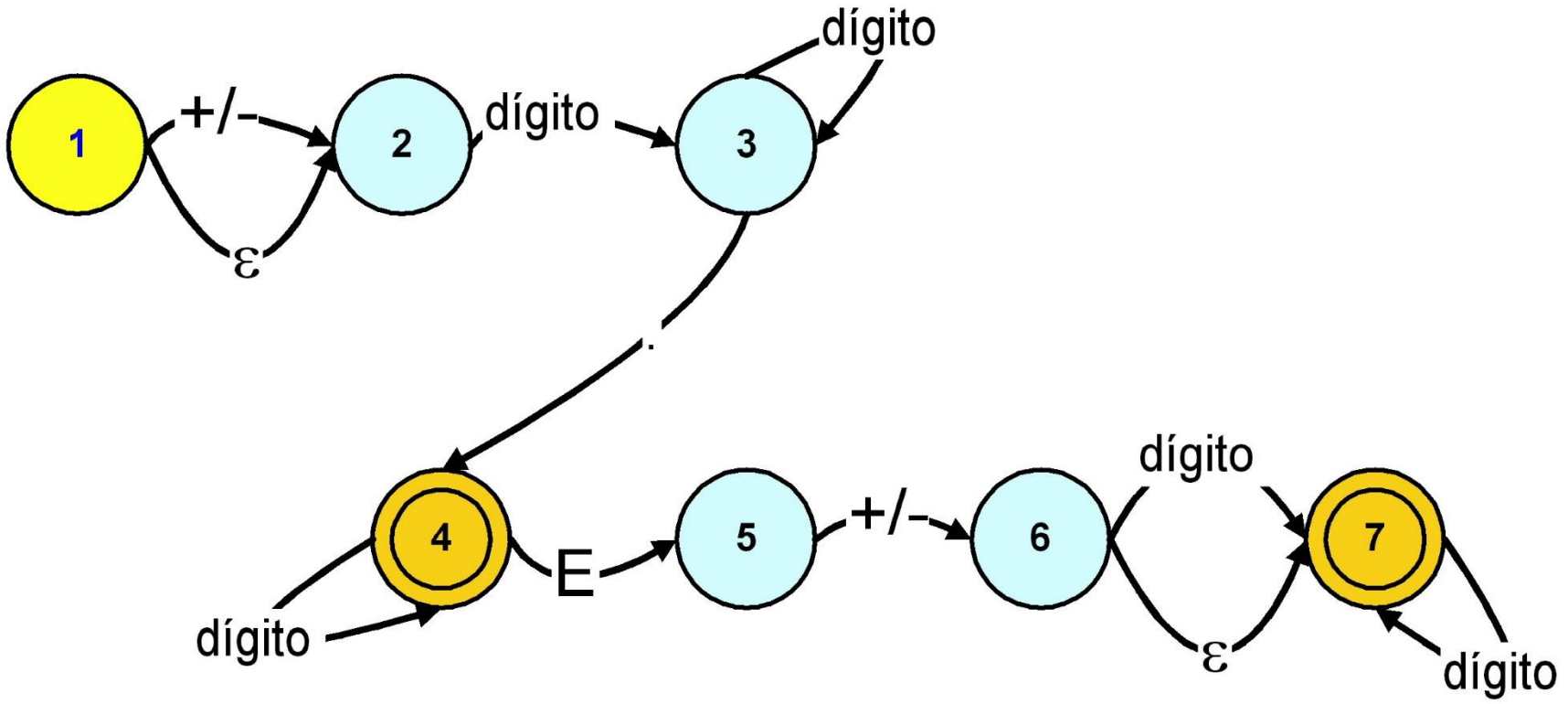
- *estados*: posições no diagrama;
- *lados:* setas que conectam os estados;
- *rótulos:* indicam os caracteres de entrada que podem aparecer após atingir-se um dado estado;

Obs: Chamamos o diagrama de *determinístico* quando o mesmo símbolo não figura como rótulo de lados diferentes que deixem um mesmo estado.

Diagramas de Transição

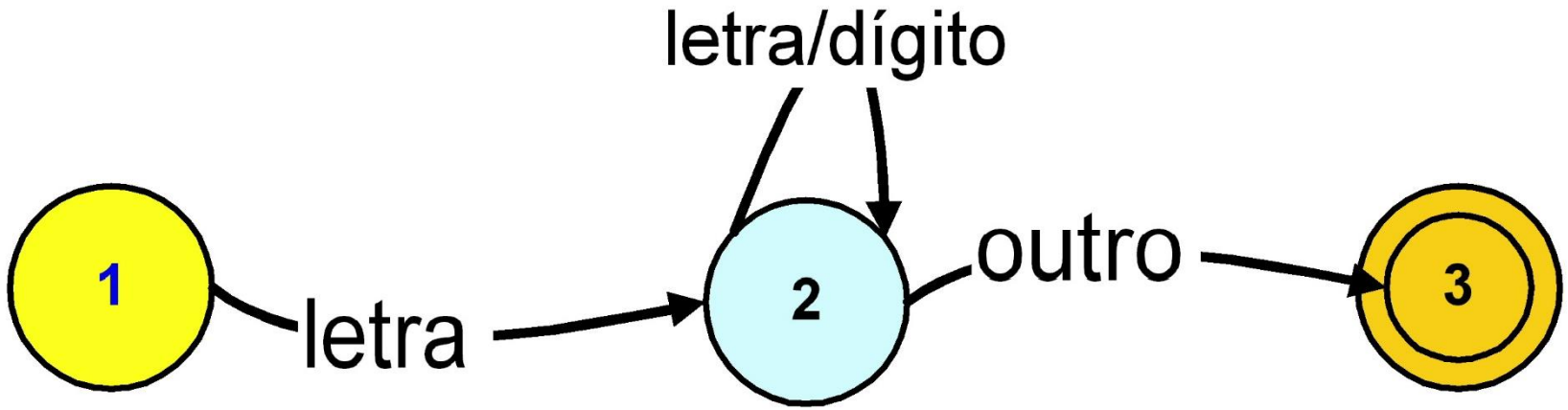
Supondo a que um número real possa ser dado por:

<sinal><parte_inteira>.<parte_fracionária>E<expoente>

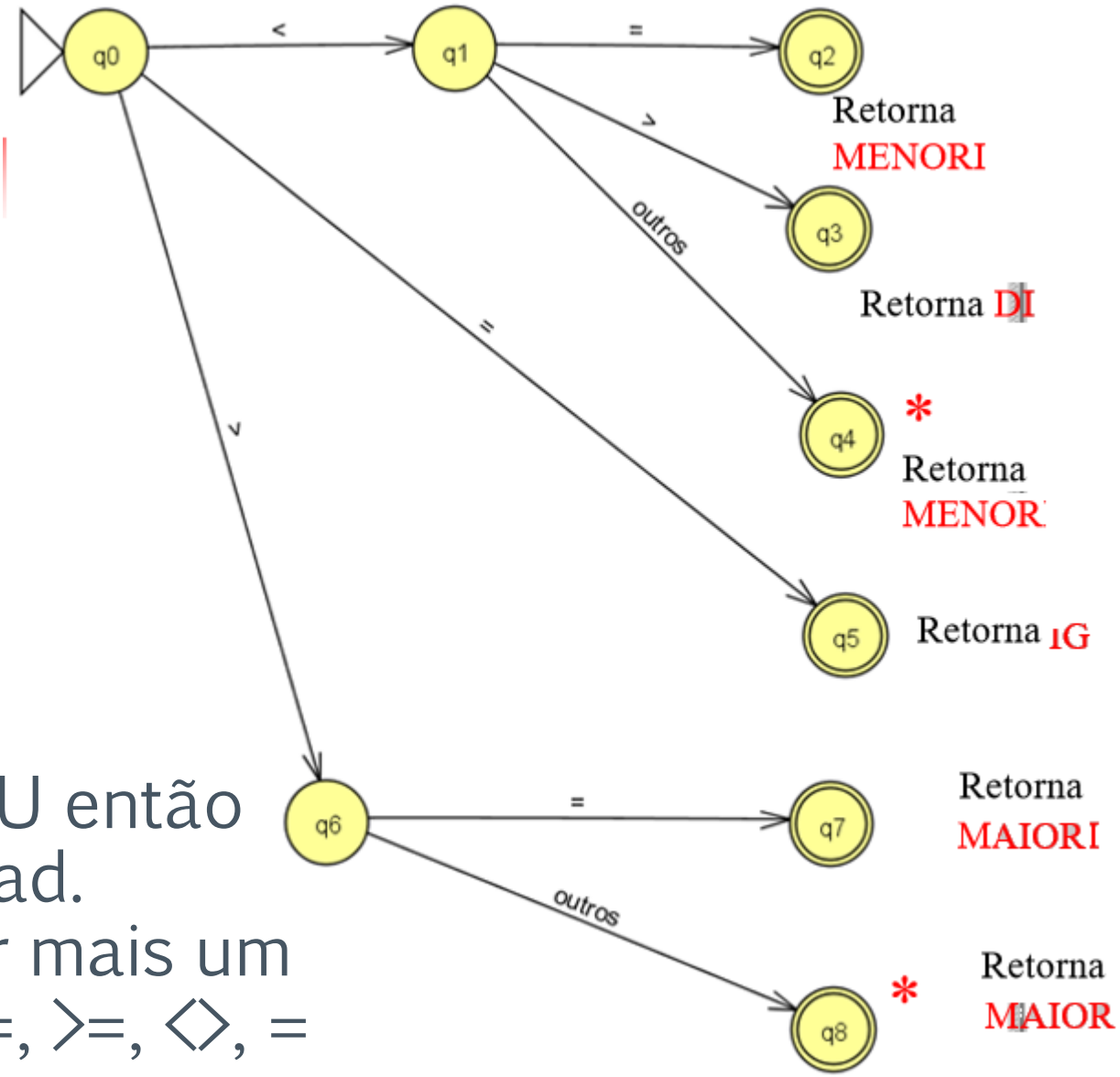


Diagramas de Transição

Para reconhecimento de identificadores:



Diagramas de Transição



- Retração da Entrada OU então deixa sempre um lookahead.
- Neste caso, deve-se ler mais um caractere nos casos de <=, >=, <>, =

Autômatos Finitos

- Os símbolos que deverão ser reconhecidos na análise léxica são representáveis por expressões regulares (ou equivalentemente por gramáticas regulares).
- Há uma correspondência unívoca entre expressões (ou gramáticas) regulares e *autômatos finitos*.

Autômatos Finitos: máquinas que podem ser utilizadas para reconhecer *strings* de uma dada linguagem. Autômatos Finitos são compostos por:

- um conjunto de **estados**, alguns dos quais são denominados estados finais. À medida que caracteres da *string* de entrada são lidos, o controle da máquina passa de um estado a outro;
- um conjunto de **regras de transição entre os estados.**

Autômatos Finitos

Formalmente, um autômato é descrito por cinco características:

- um conjunto finito de estados;
- um alfabeto de entrada finito;
- um conjunto de transições;
- um estado inicial;
- um conjunto de estados finais.

Quando, partindo de um estado inicial, varrendo a sentença dada (de acordo com as regras de transições), consegue-se atingir um estado final, a sentença dada é parte da linguagem.

Implementando An. Léxico

Funções do Analisador Léxico:

- localizar/abrir o arquivo fonte;
- separar tokens;
- classificar tokens;
- eliminar comentários;
- eliminar brancos;
- gerar uma lista dos tokens classificados;
- fechar arquivo.

Identificando tokens

```
program p;
var x: integer;
begin
        x:=1;
        while (x<3) do
                x:=x+1;
end.
```

Token	Código
program	simb_program
p	id
;	simb_pv
var	simb_var
x	id
:	simb_dp
integer	id
;	simb_pv
begin	simb_begin
x	id
:=	simb_atrib
1	num
;	simb_pv
while	simb_while
(	simb_apar

x	id
<	simb_menor
3	num
)	simb_fpar
do	simb_do
x	id
:=	simb_atrib
x	id
+	simb_mais
1	num
;	simb_pv
end	simb_end
.	simb_p

- O token integer em PASCAL é um identificador pré-definido, assim como outros tipos pré-definidos real, boolean, char, e também write, read, true e false

Tarefas Secundárias do AL

- Consumir comentários e separadores (branco, tab e CR LF) que não fazem parte da linguagem
- Processar diretivas de controle
- Relacionar as mensagens de erros do compilador com o programa-fonte
  - Manter a contagem dos CR LF"s e passar esse contador junto com a posição na linha para a rotina que imprime erros; indicar a coluna do erro também

Tarefas Secundárias do AL

- Impressão do programa-fonte
- Reedição do programa-fonte num formato mais legível, usando endentação
- Eventual manipulação da Tabela de Símbolos para inserir os identificadores
  - Pode-se optar para deixar para a Análise Semântica

Tarefas Secundárias do AL

- Diagnóstico e tratamento de alguns erros léxicos
  - Símbolo desconhecido (não pertence ao Vt - “Valor Terminal”)
  - Identificador ou constante mal formados
  - Fim de arquivo inesperado: quando se abre comentário mas não se fecha

Erros Léxicos

- Poucos erros são discerníveis no nível léxico
  - O AL tem uma visão muito localizada do programa fonte
  - Exemplo: fi (a > b) then
    - O AL não consegue dizer que fi é a palavra reservada if mal escrita desde que fi é um identificador válido
    - O AL devolve o código de identificador e deixa para as próximas fases identificar os erros

Tratamento de Constantes

- Reais
  - há um limite para o número de casas decimais e
  - outro para o tamanho max e min do expoente (+ 38 e -38)
  - Se ferir os limites tanto em tamanho quanto em valor há erro de over/underflow

Tratamento de Constantes

- String: o token „aaaaaaaaaaaaaaaaaaa … não fecha antes do tamanho máximo
  - é exemplo de má formação de string → há um limite para o tamanho da string
  - Se ferir o limite há erro
- Char: o token ‘a em a:= ‘a;
  - Seria má formação de char na linguagem geral, mas pode confundir com string que não fechou ainda, se a gramática possui ambos os tipos

Tratamento de Constantes

- Inteiro: os tokens 555555555 ou –55555555
  - são exemplos de má formação de inteiro, pois o inteiro max/min é (+/- 32767) → há um limite para o número de digitos de inteiros e seu valor
- Mas quando tratar o sinal acoplado aos números?? AL ou Asintática???
  - Para <expressões>, em <termo>, há os sinais

Tratamento de Constantes

- Pode-se optar converter token de inteiros e reais em valor numérico:
  - no AL ou no ASemântico
  - Se for no AL, além do par token/código deve-se definir uma estrutura para guardar a conversão também
  - Se for no AS, pode-se retornar o erro de overflow logo na sua montagem, caso uma constante ultrapasse seu tamanho máximo
    - Mas geralmente opta-se por fazer a conversão no ASemântico

Outros Erros Léxicos

- Tamanho de identificadores → quem pretende estipular deve checar !!!
  - Geralmente, a linguagens aceitam até um tamanho de diferenciação e descartam o resto sem indicar erro
- Fim de arquivo inesperado
  - ocorre quando se abre comentário e não se fecha, por exemplo.
  - É conveniente tratar { ..} { ...} { ...} numa rotina só
- & é um símbolo não pertencente ao Vt
  - erros de símbolos não pertencentes ao Vt

Especificação e Reconhecimento dos tokens

- Gramáticas regulares ou expressões regulares
  - podem especificar os tokens
- Autômatos Finitos:
  - São usados para reconhecer os tokens
    - Vejam exemplos de reconhecimento de operadores relacionais
    - Vejam o papel do caractere lookahead.
    - Vejam as ações associadas aos estados finais.

Códigos dos Tokens e Processo de reconhecimento

- Exemplos de códigos para tokens possíveis
  - ID: x, y, minha_variável, meu_procedimento
  - As Palavras reservadas em si e os símbolos especiais (cada um tem um código dferente): while, for, :=, <>
  - NUM_INT (Números inteiros) e NUM_REAL (números reais)
- Não basta identificar o código, deve-se retorná-lo ao analisador sintático junto com o token correspondente
  - Concatenação do token conforme o autômato é percorrido
  - Associação de ações aos estados finais do autômato
- Às vezes, para se decidir por um código, temos que:
  - ler um caractere a mais, o qual deve ser devolvido à cadeia de entrada depois OU se trabalhar com um caractere lookahead

Análise Léxica - Exemplo

Supondo o trecho de programa abaixo:

if (5 = MAX) go to 100

A lista devolvida pelo léxico poderia ser:

if, (, [const,341], =, [ident,729],), go to, [label,554]

Índices para a tabela de símbolos: contém informações sobre constantes, variáveis e cédulas.

Tabela de Símbolos

- controla as informações de escopo e de amarrações dos nomes;
- é pesquisada toda vez que um nome é encontrado no texto fonte;
- é necessário permitir a entrada de novos nomes e a obtenção daqueles já inseridos;
- podem ser implementadas por:
  - listas lineares
  - tabelas *hash*

Tabela de Símbolos

- Supondo: *n* entradas e *e* inquisições:

Lista linear:

- fácil implementação;
- desempenho pobre para *n* e *e* grandes.

Tabela *hash*

- melhor desempenho;
- mais esforço de programação;
- mais espaço para armazenamento.

Obs: é útil para o Compilador a inserção em tempo de execução!

Tabela de Símbolos

- **Entradas:**

- declarações de nomes;
- formato não necessita ser uniforme;
- implementação: registro com uma sequência de palavras;
- pode ser necessário a utilização de apontadores.

- **Tipos de Entradas:**

- palavras reservadas: inseridas inicialmente na tabela (antes da análise léxica ser iniciada);
- fortemente relacionadas com o papel de um nome no contexto da linguagem fonte;
- o analisador léxico pode começar o processo de entrada dos nomes
- risco: nome pode denotar objetos distintos.

Tabela de Símbolos

- **Exemplo:**

```
int x;
struct x {float y, z;};
```

x: inteiro e rótulo de uma estrutura de campos.

Neste caso:

- léxico retorna ao sintático o nome (ou um apontador para o lexema que forma o nome);
- registro na tabela: criado quando o papel sintático do nome for detectado
- duas entradas para x: inteiro e estrutura.

Tabela de Símbolos

Operações sobre T. S.:

- verificar se um dado nome está na T. S.
- inserir um nome na T. S.
- acessar a info. associada a um nome
- adicionar info. associada a nome
- eliminar 1 ou grupo de nomes.

Entrada da T.S.

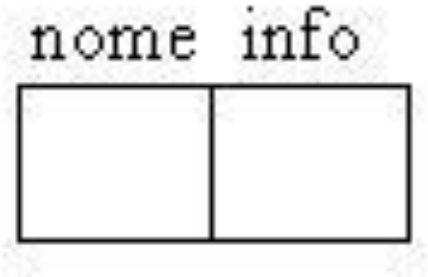
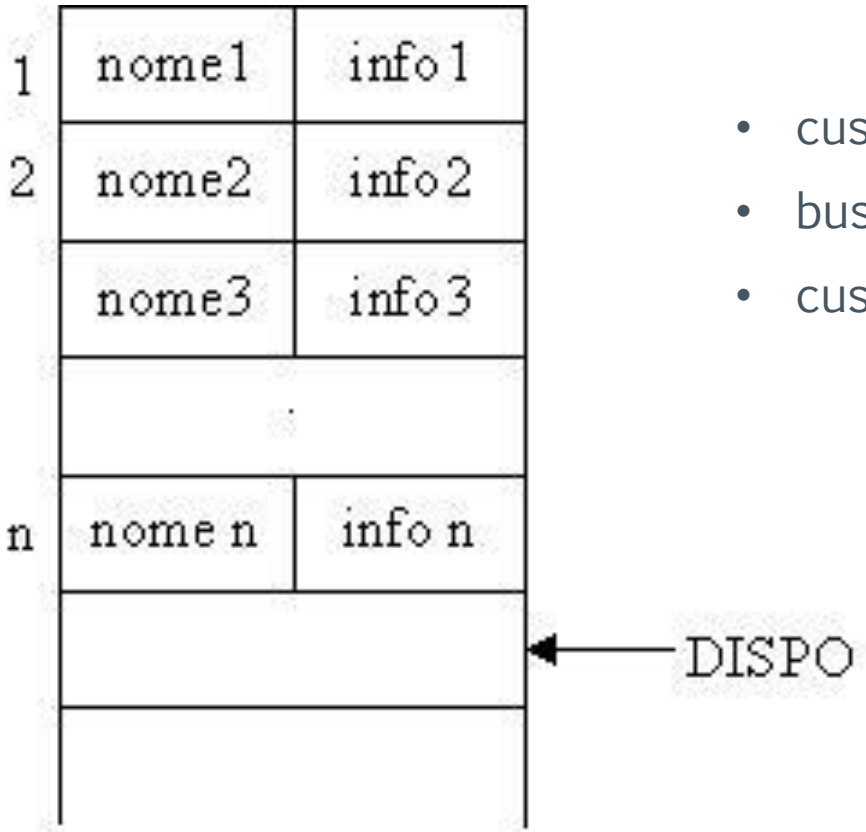


Tabela de Símbolos - Est. dados

(1) Listas Lineares Seqüenciais:

- Simples; fácil de implementar

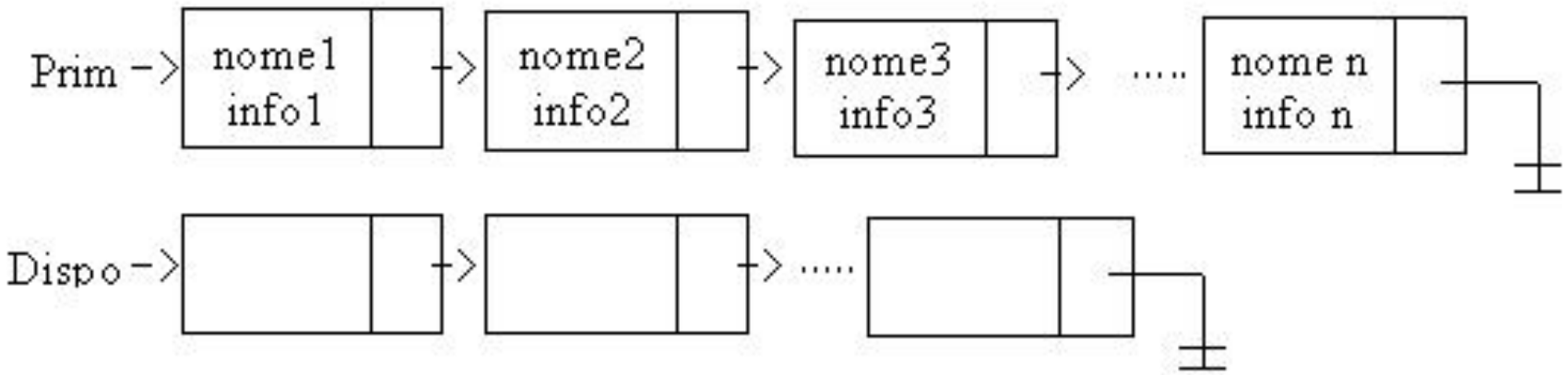


- custo p/ inserção: aproximadamente n
- busca média: n/2
- custo p/ n inserções e m buscas:
  $cn(n+m)=O(n^2)$

Tabela de Símbolos - Est. dados

(2) Listas Encadeadas

- nomes referenciados são movidos para o início da lista;

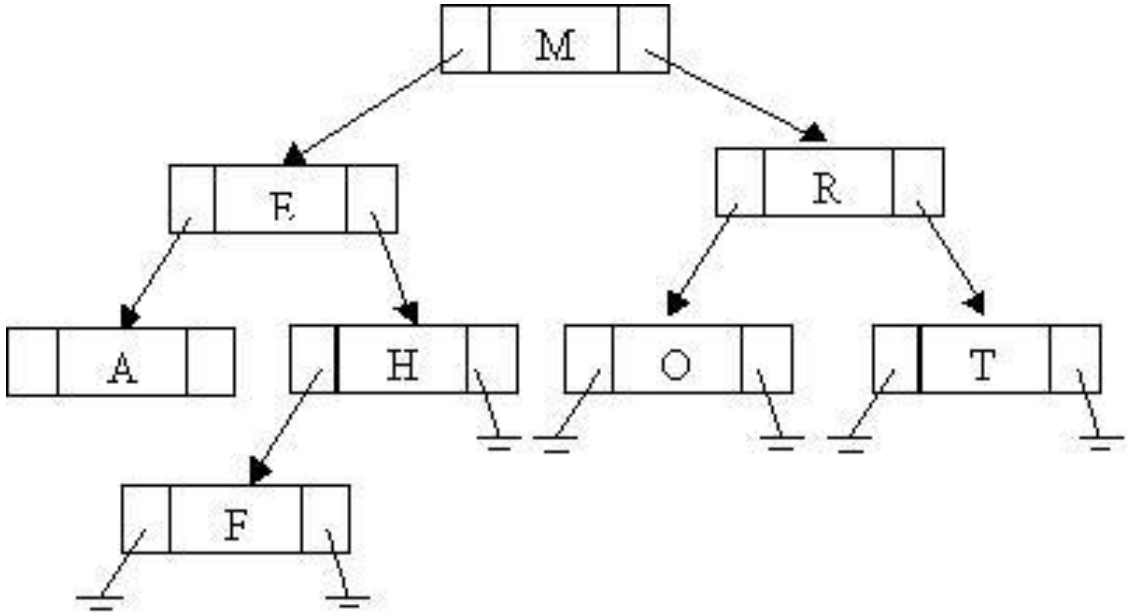


- redução de tempo, em relação ao método anterior.

Tabela de Símbolos - Est. dados

(3) Árvores de Busca

- nomes à esquerda ou direita de um outro nome:



Nomes em ordem aleatória:

- Comprimento médio de busca: ~ log(n)
- Custo p/ n inserções e m buscas: ~ (n + m) log (n)
- n > 50 ====>árvore busca é melhor solução que listas.

Tabela de Símbolos - Est. dados

(4) Tabelas *hash*

- relações que levam, de uma certa característica da chave em que se aplica a relação, a um endereço válido para cada caso específico;

- em particular, numa T.S. que comporta m identificadores, devemos construir uma relação de modo que, aplicando-a a um dado identificador obteremos o endereço onde deveriam estar as informações referentes a ele;

- o endereço obtido deve sempre estar dentro da T.S

- $m > E > 0$, onde: m: número de identificadores e E é o conjunto de endereços válidos para a tabela de símbolos

Tabela de Símbolos - Est. dados

(4) Tabelas *hash (continuação)*

-o número de identificadores possíveis é muito maior do que o número de posições na T.S.,

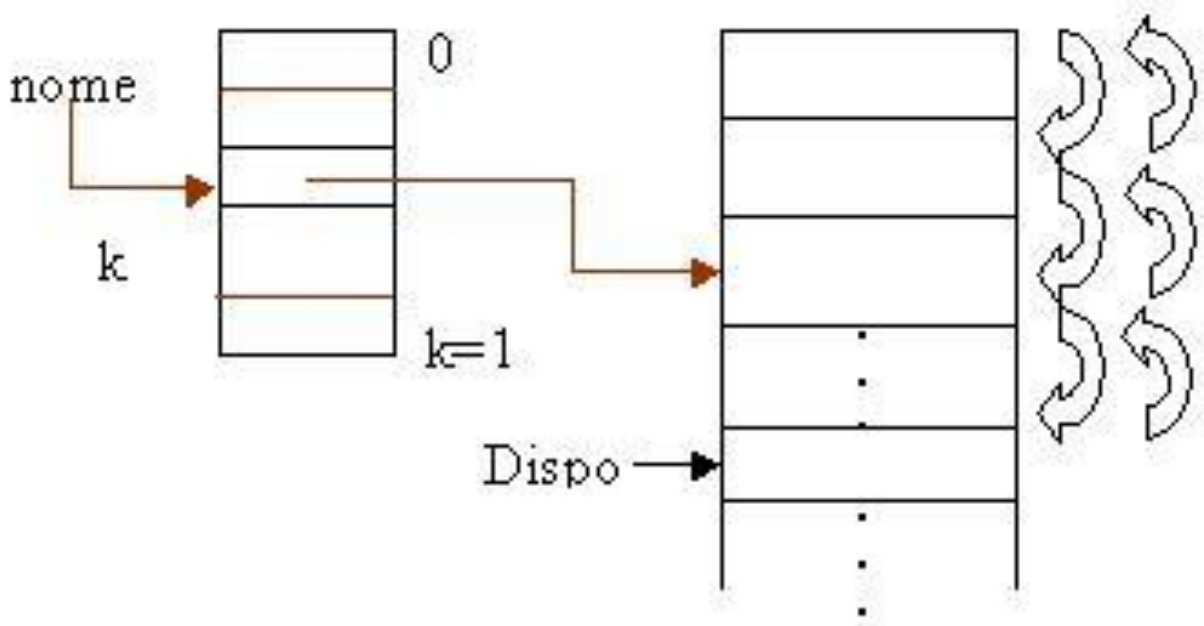
- haverá casos de conflitos; isto é, mais do que um identificador produzirá o mesmo endereço na T.S.;

- a eficiência do método de funções de "hash" está intimamente relacionada com a solução adotada para resolver os casos de conflito;

- implementação mais custosa que os processos anteriores.

Tabela de Símbolos - Est. dados

(4) Tabelas *hash (continuação)*



busca média:

n/k

Tabela de Símbolos - considerações

- É comum se ter mais do que uma tabela devido o espaço requerido por cada nome (que pode variar consideravelmente, dependendo do uso que se faz do nome);
- Se o formato das entradas de informação puder variar, uma única tabela pode bastar.
- Dependendo de como a análise léxica é implementada, pode ser útil inicializar a T.S. com as palavras reservadas.
- Se a linguagem não *reserva* palavras-reservadas (permite o uso de tais palavras como identificadores), então é essencial que as palavras reservadas sejam introduzidas na T.S. e que ela tenham associadas a si uma informação de que podem ser usadas como palavras-reservadas.

Palavras Reservadas x Identificadores

- Em implementações manuais do AL, é comum reconhecer uma palavra reservada como identificador
  - Depois fazer a checagem numa tabela de palavras reservadas
  - Solução simples e elegante
- A eficiência de um AL depende da eficiência da checagem na tabela
  - Em compiladores reais elas não são implementadas com busca linear!!!!!!!
  - Usa-se busca binária ou hashing sem colisões (dá para evitar, pois temos todas as palavras de antemão)

Alocação de espaço para identificadores (e de tokens em geral)

- Há um grande cuidado na implementação da variável token, que recebe os tokens do programa
  - Para certos casos como símbolos especiais basta definir como string de tamanho 2; palavras reservadas geralmente não ultrapassam de 10.
  - Mas como fazer para identificadores, strings, números???
  - Identificadores preocupam, pois eles ficam guardados na Tabela de Símbolos e reservar 256 caracteres para cada um pode ser abusivo em termos de espaço
  - Uma saída é usar alocação dinânima para alocar o tamanho exato de cada token.

Formas de Implementação da Análise Léxica

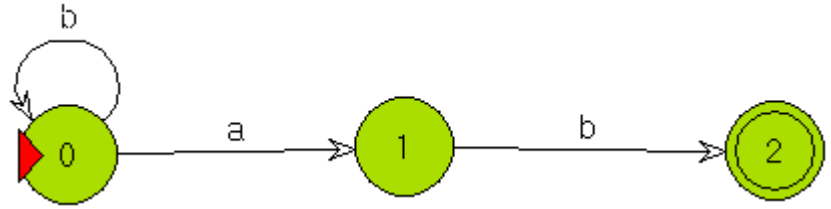
- Três formas de implementação manual do código – quando otimização é importante
  - Ad hoc – tem sido muito usada
  - Código que reflete diretamente um AF
  - Uso de Tabela de Transição e código genérico
- Existem muitos analisadores, Ex.: Lex (gerador de AL) ou outro compiler (JAVACC) –

muito utilizados em projetos reais

Solução ad hoc

Mantém o estado implicitamente, indicado nos comentários
Uso de avanço da entrada (chamada da função próximo_caractere)

```
{início –  estado 0}
c:=próximo_caractere()
se (c='b') então
        c:=próximo_caractere()
        enquanto (c=b) faça
                c:=próximo_caractere()
{estado 1}
        se (c='a') então
                c:=próximo_caractere()
{estado 2}
        se (c='b') e (acabou cadeia de entrada) então retornar "cadeia aceita"
                senão retornar "falhou"
        senão retornar "falhou"
{estado 1}
senão se (c='a') então
        c:=próximo_caractere()
{estado 2}
        se (c='b') e (acabou cadeia de entrada) então retornar "cadeia aceita"
                senão retornar "falhou"
        senão retornar "falhou"
```



Problemas?

π

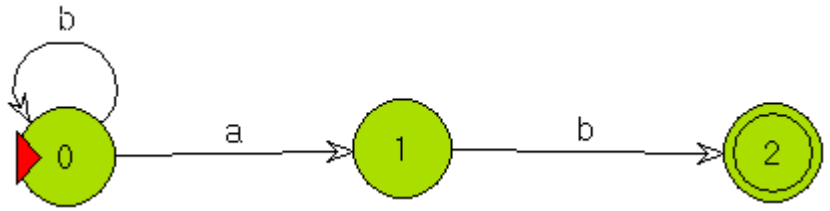
Solução ad hoc

- Simples e fácil
- Mantém o estado implicitamente, indicado nos comentários
- Razoável se não houver muitos estados, pois a complexidade do código cresce com o aumento do número de estados
- Problema: por ser ad hoc, se mudar o AF temos que mudar o código

Solução: Incorporação das transições no código do programa

- Uso de uma variável para manter o estado corrente e
- Uso de avanço da entrada (chamada da função próximo_caractere)

```
s:=0 {uso de uma variável para manter o estado corrente}
enquanto s = 0 ou 1 faça
        c:=próximo_caractere()
        caso (s) seja
                0: se (c=a) então s:=1
                        senão se (c=b) então :=0
                        senão retornar "falhou"; s:= outro
                1: se (c=b) então s:=2
                        senão retornar "falhou"; s:= outro
        fim caso
fim enquanto
Se s = 2 então "aceitar" senão "falhou";
```



Problemas?

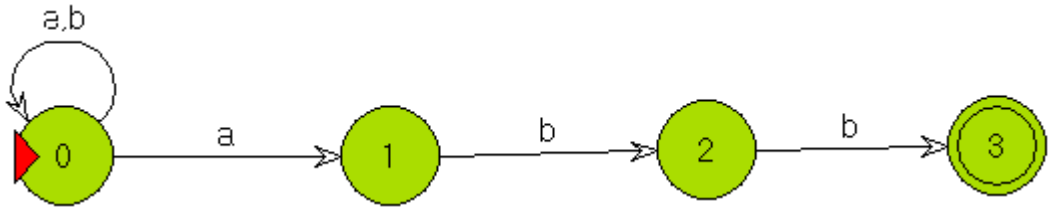
Case externo => trata do caractere de entrada. IFs internos tratam do estado corrente.

Solução: Incorporação das transições no código do programa

- Reflete diretamente o AF
- Problema: cada código é ainda diferente, caso mude o AF ele deve ser modificado

Solução: Representação em tabela de transição – Métodos Dirigidos por Tabela

- Uso de um código genérico e expressar o AF como estrutura de dados



Q/Σ	a	b
0	0, 1	1
1	-	2
2	-	3
3	-	-

Solução: Representação em tabela de transição – Métodos Dirigidos por Tabela

- Problema: tabela não indica estados de aceitação nem quando não se consome entrada. Temos que estendê-la

Execução do autômato

- Se autômato determinístico (i.e., não há transições ? e, para cada estado S e símbolo de entrada a, existe somente uma transição possível), o seguinte algoritmo pode ser aplicado

s:=s_0

c:=próximo_caractere()

enquanto (c<>eof) faça
    s:=transição(s,c)
    c:=próximo_caractere()

fim

se s for um estado final
    então retornar “cadeia aceita”
    senão retornar “falhou”

Checando por estados de erro

```
Read(caracter_corr);
estado := estado_inicial;
While (estado < > final) and (estado < > erro)do
        begin
                prox_estado := Tab(estado,caracter_corr);
                Read(caracter_corr);
                estado := prox_estado
        end;
If estado in final then RetornaToken
Else Erro;
```

Solução: Métodos Dirigidos por Tabela

- Vantagem: elegância (código é reduzido) e generalidade (mesmo código para várias linguagens);
- Desvantagem: pode ocupar grande espaço quando o alfabeto de entrada é grande;
- Grande parte do espaço é desperdiçada. Se forem usados métodos de compressão de tabelas (p.ex. rep. de mat. esparsas como listas) o processamento fica mais lento;
- Métodos dirigidos por tabela são usados em geradores como o Lex.

Problemas da modelagem com AF

- Observem, entretanto que a modelagem com AFND mostra o que o Analisador Léxico deve reconhecer MAS não mostra como.
- Por exemplo, nada diz sobre o que fazer quando uma cadeia pode ter 2 análises como é o caso de:
- 2.3 (real ou inteiro seguido de ponto seguido de real)
- Ou
- <= (menor seguido de igual ou menor igual)
- OU
- Program(identificador ou palavra reservada program)

Valor: 1,0 ponto

Exercício 11

› Identifiquem tokens e dêem códigos apropriados

Pascal

```
function max (i, j: integer): integer;
{ return maximum of integers I and j}
begin
  if i > j then max := i
  else max := j
end;
```

C

```
int max (i , j) int i, j;
{ /* maximum of integers i and  j */
return i > j ? i : j;
}
```

Dúvidas

José Osvano da Silva
joseosvano@unipac.br