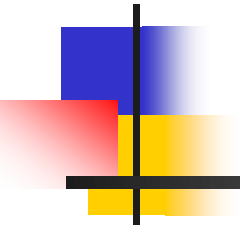
Análise Léxica



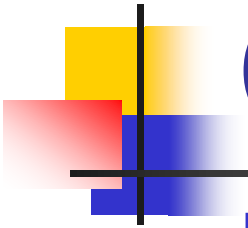
Função de um Analisador Léxico (AL)

Erros Léxicos

Métodos para a Especificação e Reconhecimento dos tokens: ER e AF

Prática: Tabela de Palavras Reservadas; Alocação de Espaço para Identificadores; Formas de Implementação de um ALéxico

Função de um Analisador Léxico (Scanner)

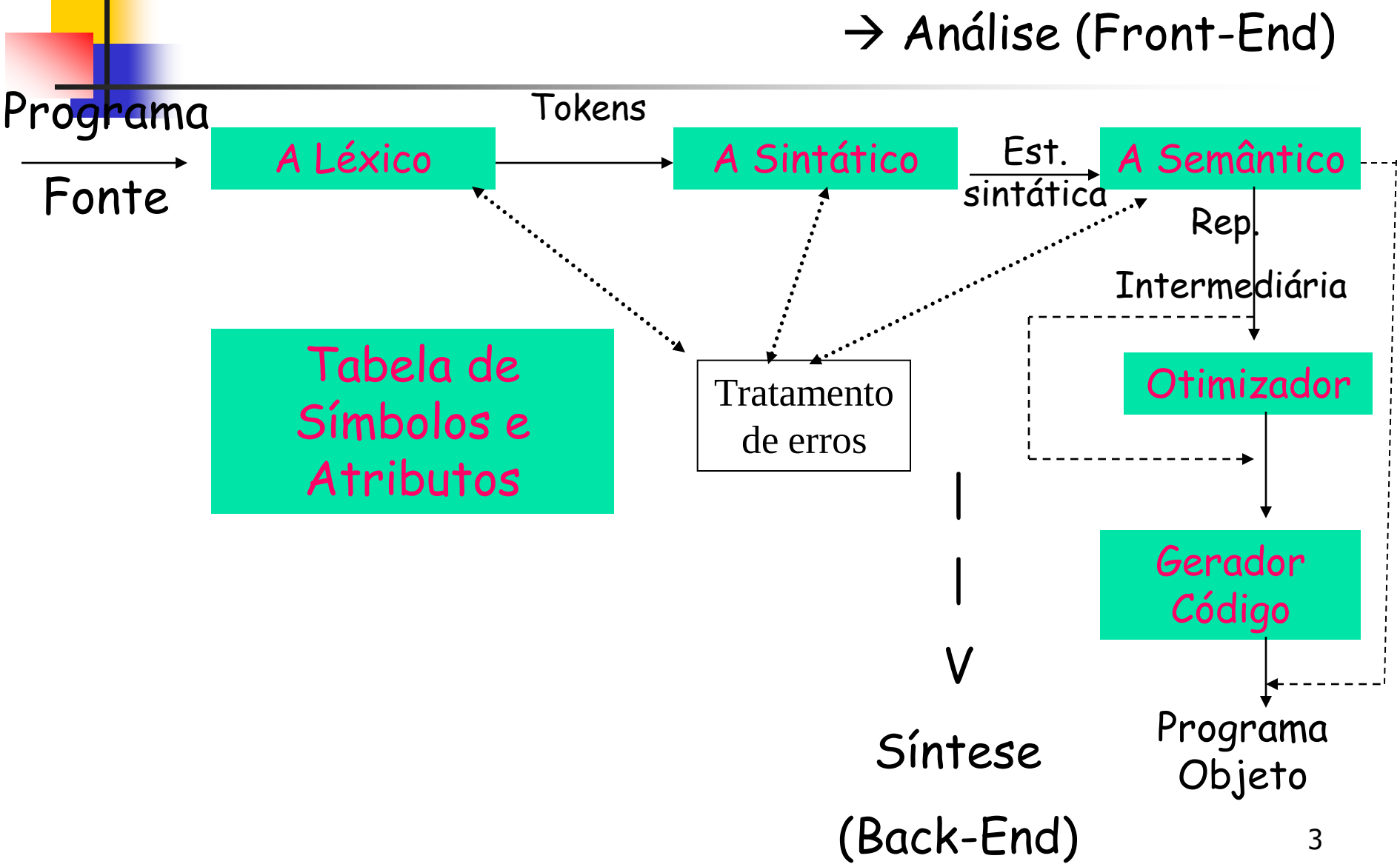


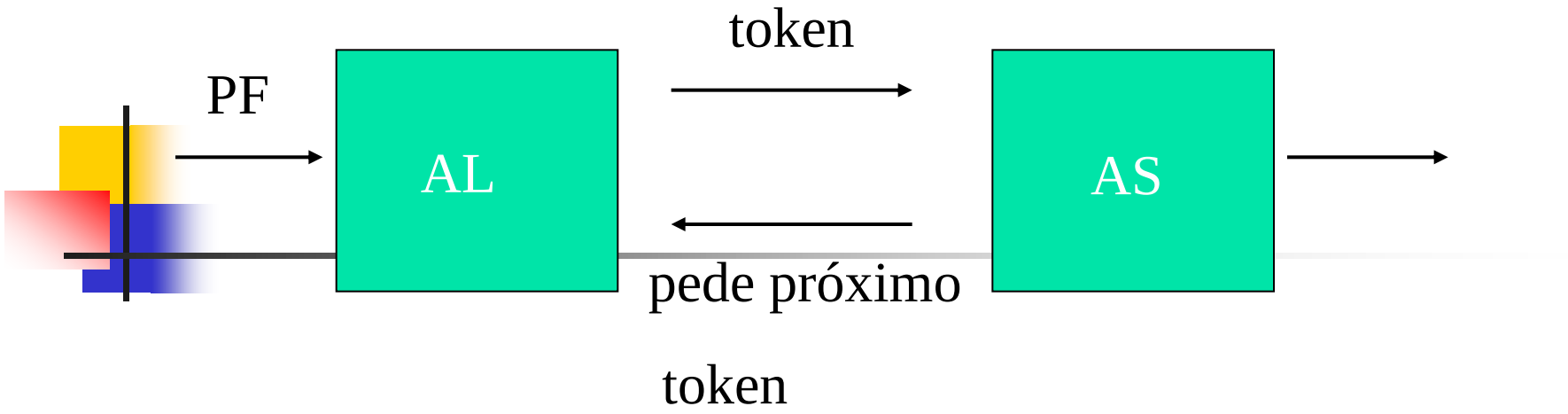
- **A Análise Léxica é a primeira fase de um compilador.**

- **Tarefa principal**
  - ler o arquivo onde se encontra o programa-fonte e
  - produzir como saída uma seqüência de tokens com seus respectivos códigos que o Analisador Sintático usará para validar regras da gramática

- **Exemplo de tokens:**
  - identificadores,
  - palavras-reservadas,
  - símbolos especiais simples e compostos, e
  - as constantes de todos os tipos permitidos na linguagem

Estrutura de um Compilador





- Esta interação é comumente implementada fazendo o AL como
  - Uma subrotina ou co-rotina do Analisador Sintático (AS)

- Quando o AS ativa a sub ou co-rotina,
  - o AL lê caracteres do arquivo até que ele possa identificar o próximo token e o devolve com seu código

Exemplo da tarefa da AL

- x:=y*2;

Token	Código
x	id
:=	simb_atrib
y	id
*	simb_mult
2	num
;	simb_pv

Exemplo: usando códigos numéricos (baixa inteligibilidade)

- x:=y*2;

Token	Código
id	1
num	2
simb_mult	3
simb_atrib	4
simb_pv	5

Token	Código
x	1
:=	4
y	1
*	3
2	2
;	5

Exemplo

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

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

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

AL: 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;
}
```

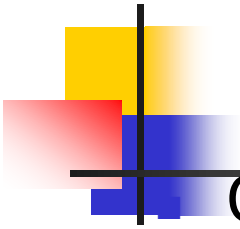
Especificação precisa dos tokens

- Devemos usar notações formais para especificar a estrutura precisa dos tokens para construir um AL sem erros.
  - Por exemplo, mesmo a definição simples de cadeias de caracteres pode ser definida erroneamente se nada for dito sobre os caracteres permitidos:

<string> ::= ‘ <caractere> { <caractere>} ‘

  - É permitido <CR> <LF> ? Não! Então <caractere> é definido como o conjunto dos imprimíveis.

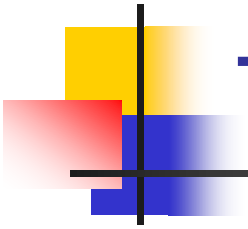
  - OBS: EM EBNF, {α} é zero ou mais vezes α



Especificação precisa dos tokens

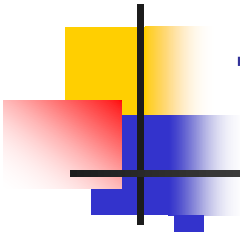
Outro exemplo:

- Números reais em notação de ponto fixo, por exemplo, 10.0 ou 0.1 são possíveis.
  - MAS 10. e .1 são permitidos???
- Em Fortran são, em Pascal não pela simples razão de não os confundir com o intervalo de inteiros (10..3, por exemplo)
  - Se as notações 10. e .1 fossem permitidas, embora a existência de 2 reais não seja permitida pela gramática, o AL não conseguiria "segurar" esse erro.
- Lembrem que o papel de um AL é montar, empacotar um token com seu código (*um* por vez) e passar o pacote para o AS!



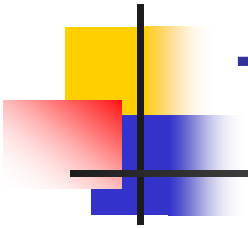
Tarefas Secundárias do AL

- 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 indentação
- Eventual manipulação da Tabela de Símbolos para inserir os identificadores
  - Pode-se optar para deixar para a Análise Semântica



Tarefas Secundárias do AL

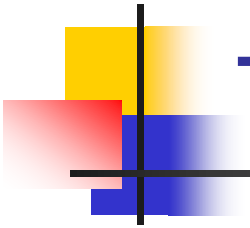
- Diagnóstico e tratamento de alguns erros léxicos
  - Símbolo desconhecido (não pertence ao Vt)
  - Identificador ou constante mal formados
  - Fim de arquivo inesperado: quando se abre comentário mas não se fecha

Vantagens da Separação entre AL e ASintática

- Simplificação
  - Um AS que tenha que fazer o tratamento de comentários e separadores é bem mais complexo do que um que assume que eles já foram removidos
- Eficiência
  - Uma parte apreciável do tempo de compilação corresponde à AL que separada facilita a introdução de certas otimizações
- Manutenção
  - Toda parte referente à representação dos terminais está concentrada numa única rotina tornando mais simples as modificações de representação

Erros Léxicos

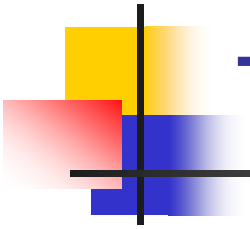
- Poucos erros são discerníveis no nível léxico
  - O AL tem uma visão muito localizada do programa-fonte
  - Exemplo: fi (a > b) then
    - O AL não consegue dizer que fi é a palavra reservada if mal escrita desde que fi é um identificador válido
    - O AL devolve o código de identificador e deixa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



Tratamento de Constantes

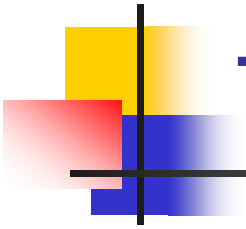
- Reais
  - há um limite para o número de casas decimais e
  - outro para o tamanho max e mim do expoente (+ 38 e -38)

→ Se ferir os limites tanto em tamanho quanto em valor há erro de over/underflow



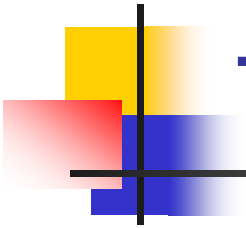
Tratamento de Constantes

- String: o token ‘aaaaaaaaaaaaaaaaaa … 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 AL, 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

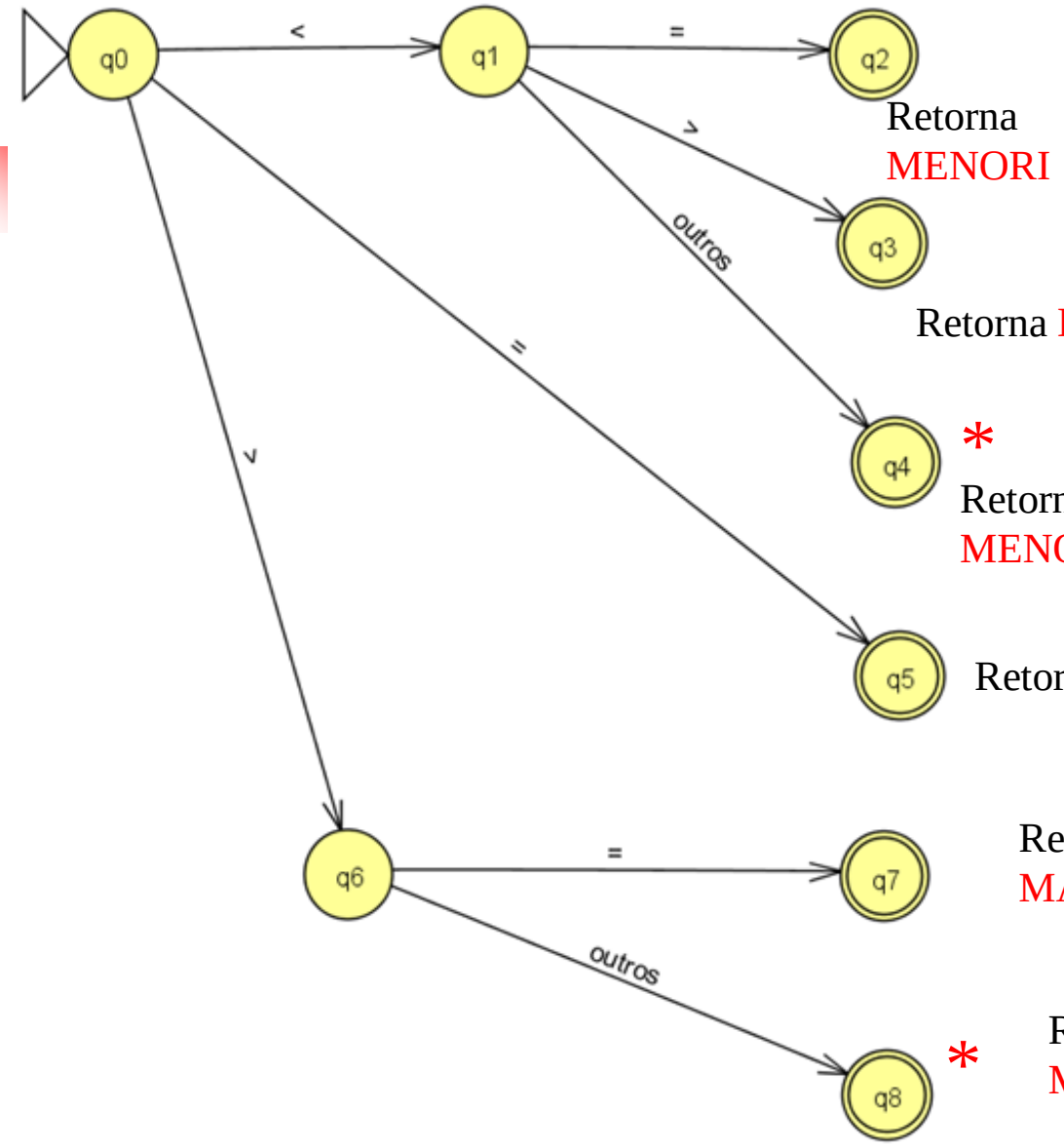
- Gramáticas regulares ou expressões regulares
  - podem especificar os tokens
- Autômatos Finitos:
  - São usados para reconhecer os tokens
    - Vejam exemplos de reconhecimento de operadores relacionais
    - Vejam o papel do caractere lookahead
    - Vejam as ações associadas aos estados finais

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



Retorna MENORI

Retorna DI

*

Retorna MENOR

Retorna IG

Retorna MAIORI

*

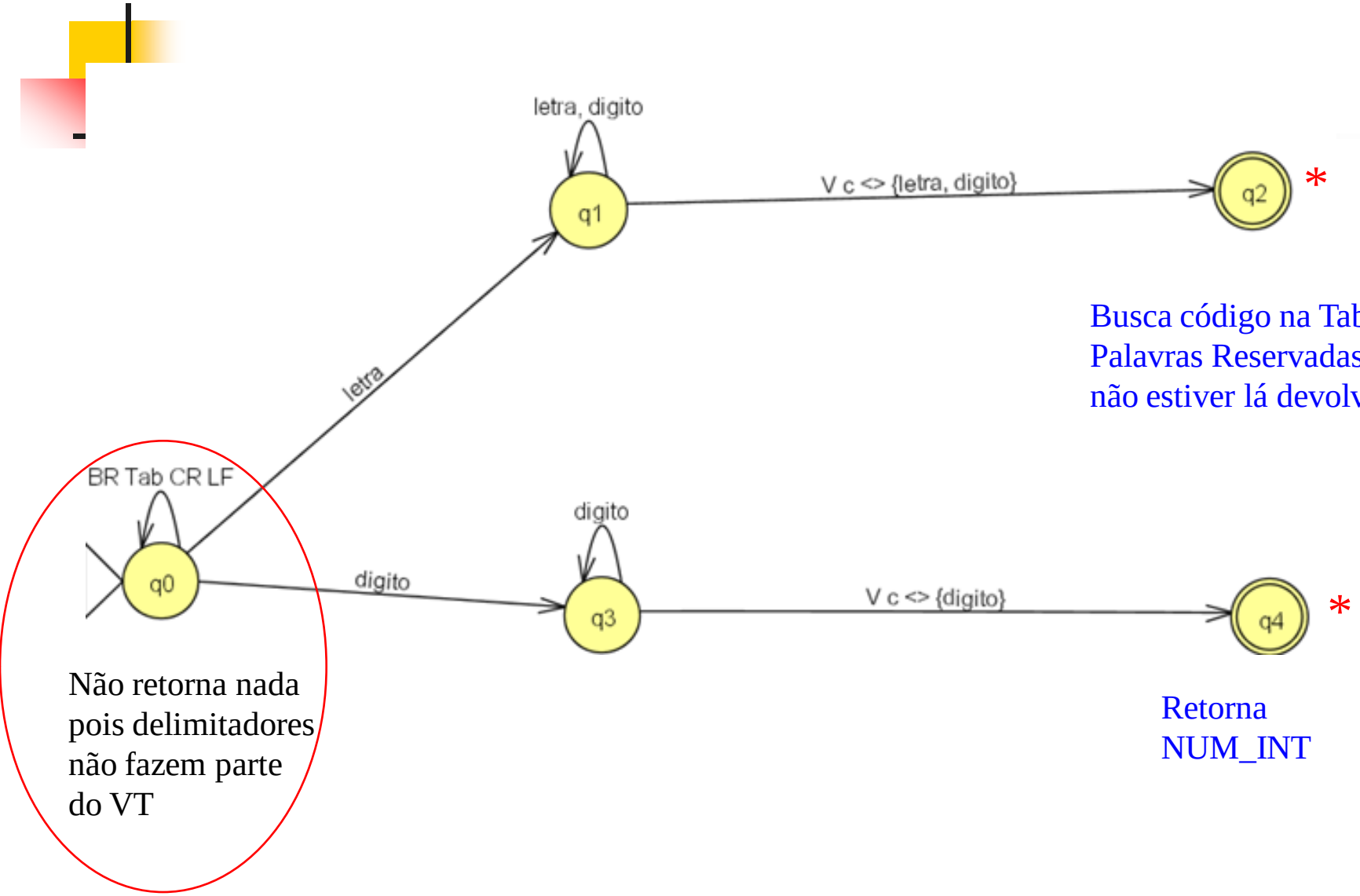
Retorna MAIOR

Input	Result
>	Reject
<	Reject
>=	Accept
<=	Accept
=	Accept
<>	Accept
+	Reject
-	Reject

*** Retração da Entrada**

OU

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


letra, digito
q1
V c <> {letra, digito}
q2
*
Busca código na Tabela de Palavras Reservadas, se não estiver lá devolver ID
letra
BR Tab CR LF
q0
digito
digito
q3
V c <> {digito}
q4
*
Não retorna nada pois delimitadores não fazem parte do VT
Retorna NUM_INT

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.

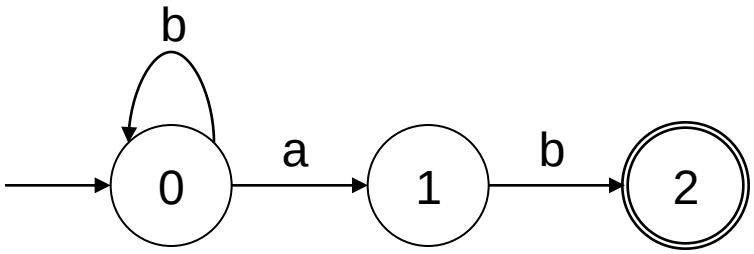
Formas de Implementação da Análise Léxica

- 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
- Uso do Lex (gerador de AL) ou outro compiler compiler (JAVACC) – muito utilizados em projetos reais

Solução ad hoc

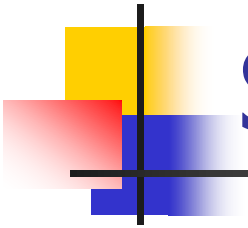
- Mantém o estado implicitamente, indicado nos comentários
- Uso de avanço da entrada (chamada da função próximo_caractere)

```
{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?



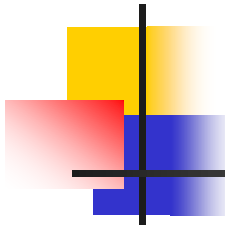
Solução ad hoc

- 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 ad hoc

```
Ch:= ' ';
{Ch está sempre preparado com um caractere}
function Analex (var S: string): CodAtomo
Enquanto Ch = ' ' faça Ch:= ler caractere; {elimina brancos}
Se Ch = '{' então comentário {elimina comentário}
Se letra(Ch) então ...
Senão
Se digito(Ch) então ...
Senão
      Caso Ch
            '<':      Ch:= ler caractere
                      Se Ch = '>'
                                Então
                                          S:= '<>'; Ch:= ler caractere;
                                          Analex:= Sdiferente
                      Senão
                                Se Ch = '='
                                          Então
                                                    S:= '<='; Ch:= ler caractere;
                                                    Analex:= Smenor-igual
                                          Senão
                                                    S:= '<'
                                                    Analex:= Smenor
...
            '$':      Se eof (arq) então
                                S:= '$'
                                Analex:= Sfim-arq
                      Senão
                                Ch:= ler caractere
                                Analex:= Snada
            Outrocaso:
                      begin
                      Repita
                                Inserir caractere
                      Até encontrar (letra ou digito ou caractere especial ou '$' )
                      Analex:= Snada
                      end
```



Drive para testar o Analex:

Programa principal:

```
    Begin
        acabou := falso
        enquanto não acabou faça
                x := analex (s);
                escrita (s)
                se x = Sfim-arq então acabou:=verdade
    end
```

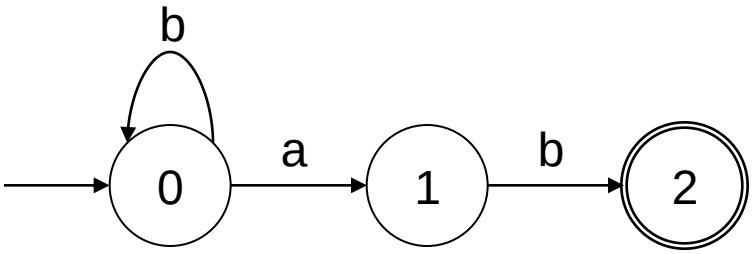
Modelo de escrita:

```
program                 código de programa
Teste                   código de ident
;                        código de ;
1 program Test ;
```

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)

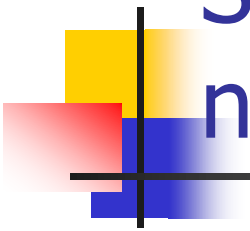
```
s:=0 {uso de uma variável para manter o estado corrente}
enquanto s = 0 ou 1 faça
    c:=próximo_caractere()
    caso (s) seja
        0: se (c=a) então s:=1
           senão se (c=b) então s:=0
           senão retornar "falhou"; s:= outro
        1: se (c=b) então s:=2
           senão retornar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



Problemas?

Case externo => trata do caractere de entrada. IF´s internos tratam do estado corrente.

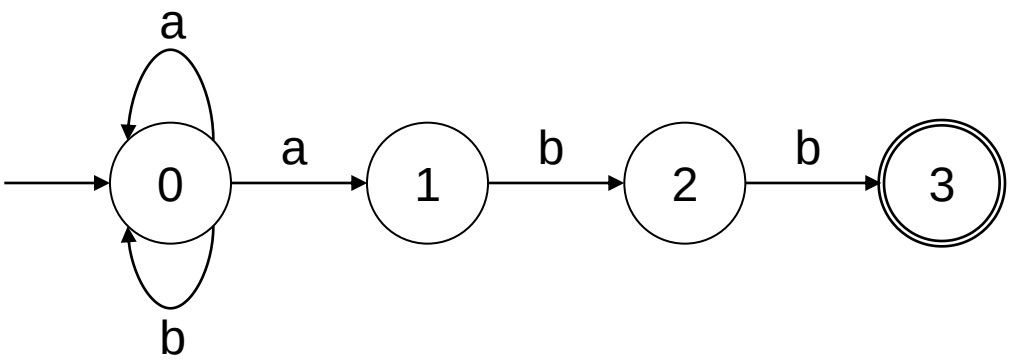
Solução: Incorporação das transições no código do programa



- 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



Estado	Símbolo de entrada	
	a	b
0	{0,1}	{0}
1	---	{2}
2	---	{3}

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

Execução do autômato

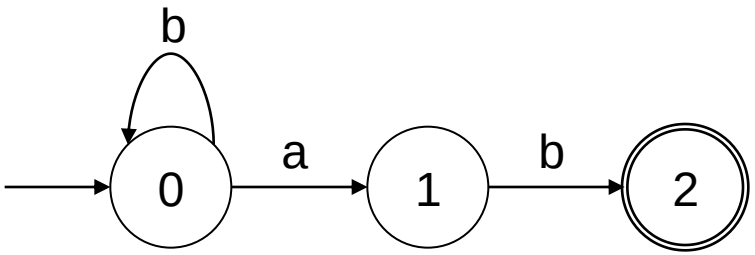
Se for autômato determinístico (i.e., não há transições λ e, para cada estado s e símbolo de entrada a, existe somente uma transição possível), o seguinte algoritmo pode ser aplicado

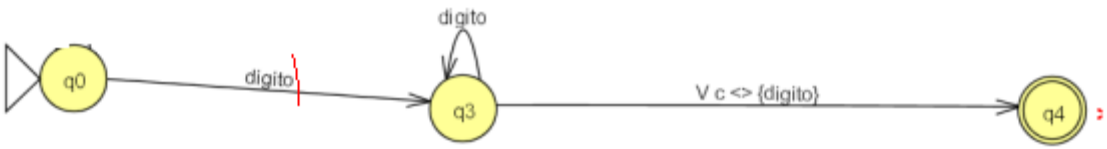
```
s:=s0;
c:=próximo_caractere();
enquanto (c<>eof) faça
   inicio
      s:=transição(s,c);
      c:=próximo_caractere();
   fim
se s for um estado final
    então retornar "cadeia aceita"  senão retornar "falhou"
```

Estendendo a Tabela

Estado/ Final	Símbolo de entrada		
	a	b	outro
0/não	1	0	---
1/não	---	2	---
2/sim	---	---	---

s:=s_0
c:=próximo_caractere()
enquanto (s <> final) faça
    s:=transição(s,c)
    c:=próximo_caractere()
fim
se s for um estado final
    então retornar "cadeia aceita"
    senão retornar "falhou"



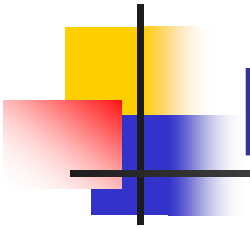


[ ] transição que não consome entrada: V c <> {digito}

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.

Recuperação de Erros Léxicos

- Para que a compilação não pare por causa de erros pequenos
  - é necessário tentar algum tipo de recuperação
  - Existem 3 opções, pelo menos

Exemplo: beg#in

1. Deletar todos os caracteres lidos e começar o AL no próximo não lido:
   devolve in = identificador

2. Deletar o caractere lido e continuar o AL no próximo caractere:
   devolve 2 identificadores: beg e in

3. Não parar para erros léxicos, i.e. empacota tudo. Para os não pertencentes ao Vt devolve um novo código = NADA e deixa o AS cuidar de erros
   devolve beg NADA in

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:

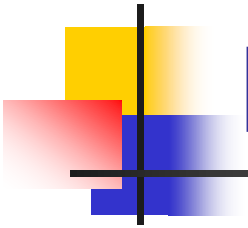
2.3 (real ou inteiro seguido de ponto seguido de real)

Ou

<= (menor seguido de igual ou menor igual)

OU

Program (identificador ou palavra reservada program)



Regras de desambiguação

- Assim, precisamos de regras para desambiguar esses casos.
- Usamos as regras:
  - escolha a maior cadeia
  - Dê preferência para a formação de:
    - palavras-reservadas em detrimento de identificadores, usando a ordem de definição de palavras-reservadas ANTES da definição de identificadores
- Estas regras são implementadas em compiler compilers como o LEX e JAVACC.