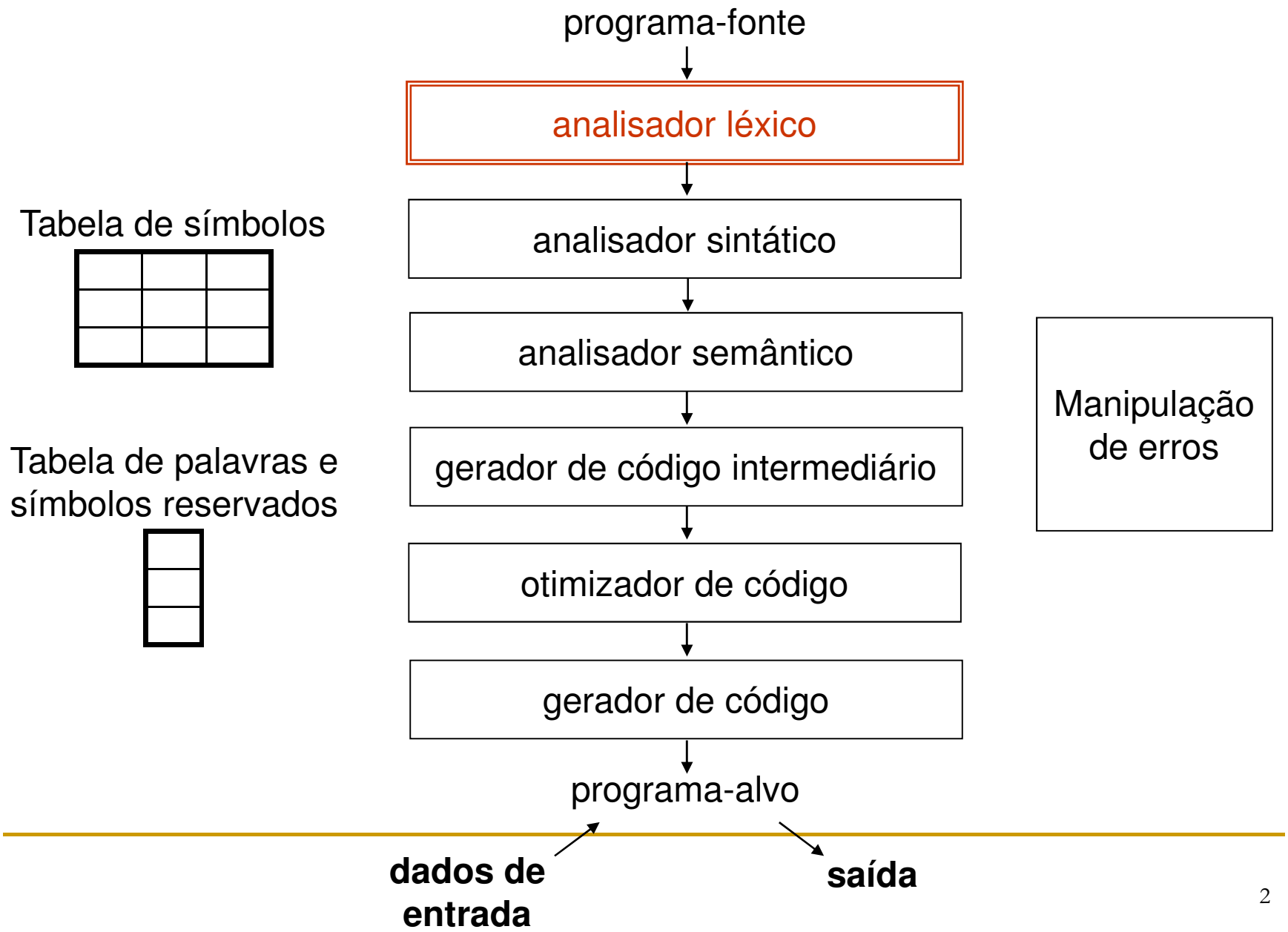
Análise léxica

Função, interação com o compilador
Especificação e reconhecimento de tokens
Implementação
Tratamento de erros

Prof. Thiago A. S. Pardo

Estrutura geral de um compilador



Analisador léxico

- Primeira etapa de um compilador

- Função
  - Ler o arquivo com o programa-fonte
  - Identificar os tokens correspondentes
  - Relatar erros

- Exemplos de tokens
  - Identificadores
  - Palavras reservadas e símbolos especiais
  - Números

Exemplo

- x:=y*2;

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

Exemplo: usando códigos numéricos

- x:=y*2;

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

Cadeia	Token
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.
```

Exemplo

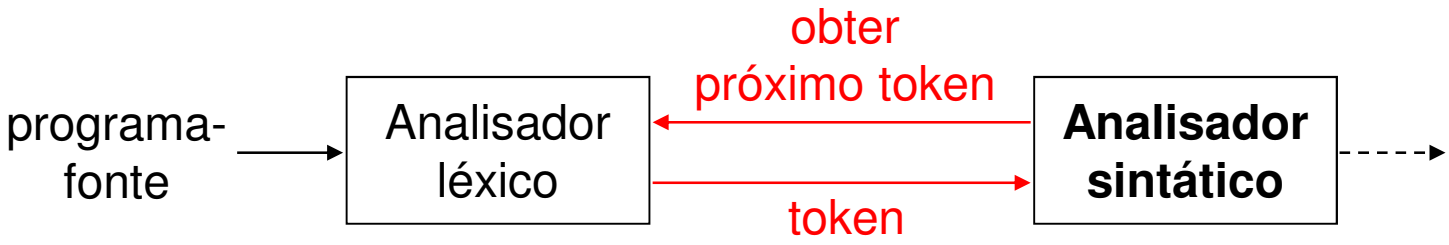
```
program p;
var x: integer;
begin
    x:=1;
    while (x<3) do
        x:=x+1;
end.
```

Cadeia	Token
program	simb_program
p	id
;	simb_pv
var	simb_var
x	id
:	simb_dp
integer	simb_tipo
;	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

Analisador léxico

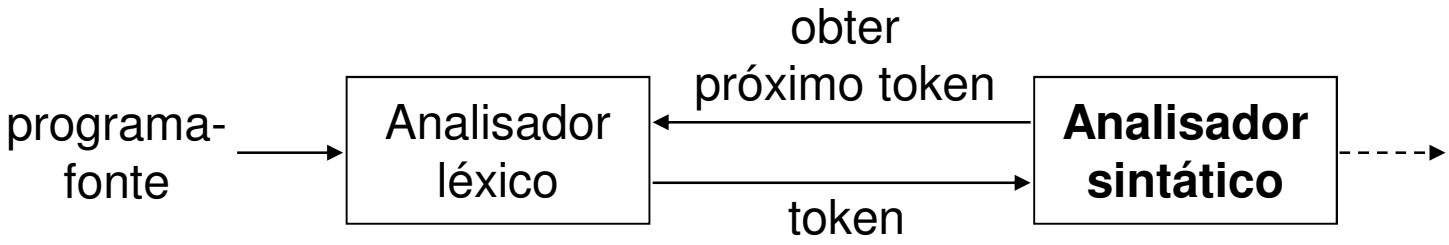
- Em geral, subordinado ao analisador sintático
  - Sub-rotina do analisador sintático: a cada chamada, o analisador léxico retorna para o analisador sintático uma cadeia lida e o token correspondente

- O analisador sintático combina os tokens e verifica a boa formação (sintaxe) do programa-fonte usando a gramática da linguagem



Analisador léxico

- Há necessidade dessa interação com o analisador sintático?
  - Não se poderia pré-processar o arquivo todo e produzir um "tabelão" com os tokens?



Por que separar o analisador léxico do sintático?

Por que separar o analisador léxico do sintático?

- Modularização
- Projeto mais simples de cada etapa
- Maior eficiência de cada processo: possibilidade de uso de técnicas específicas e métodos de otimização locais
- Maior portabilidade: especificidades da linguagem de programação podem ser resolvidas na análise léxica
- Facilidade de manutenção
- É mais fácil para o analisador léxico separar identificadores de palavras reservadas

Outras funções do analisador léxico

- Consumir comentários e caracteres não imprimíveis (espaço em branco, tabulação, código de nova linha)
  - Se a gramática fosse se responsabilizar por isso, ela seria demasiadamente complicada
    - Por quê?
- Possível manipulação da tabela de símbolos
- Relacionar as mensagens de erro emitidas pelo compilador com o programa-fonte
  - Deve-se manter contagem do número de linhas
- Diagnóstico e tratamento de erros

Erros léxicos

- Erros
  - Símbolo não pertencente ao conjunto de símbolos terminais da linguagem: @
  - Identificador mal formado: j@, 1a
  - Tamanho do identificador: minha_variável_para_...
  - Número mal formado: 2.a3
  - Tamanho excessivo do número: 555555555555555
  - Fim de arquivo inesperado (comentário não fechado): {...
  - Char ou string mal formados: ‘a, “hello world

- São limitados os erros detectáveis nessa etapa
  - Visão local do programa-fonte, sem contexto

fi (a>b) then...

Projeto do analisador léxico

- É desejável que se usem notações formais para especificar e reconhecer a estrutura dos tokens que serão retornados pelo analisador léxico
  - Evitam-se erros
  - Mapeamento mais consistente e direto para o programa de análise léxica

- Notações
  - Gramáticas ou expressões regulares: especificação de tokens
  - Autômatos finitos: reconhecimento de tokens

Expressões regulares

- Determinam conjuntos de cadeias válidas
  - Linguagem

- Exemplos
  - Identificador: letra (letra | dígito)*
  - Número inteiro sem sinal: dígito^{+}
  - Número inteiro com sinal: (+ | -) dígito^{+}

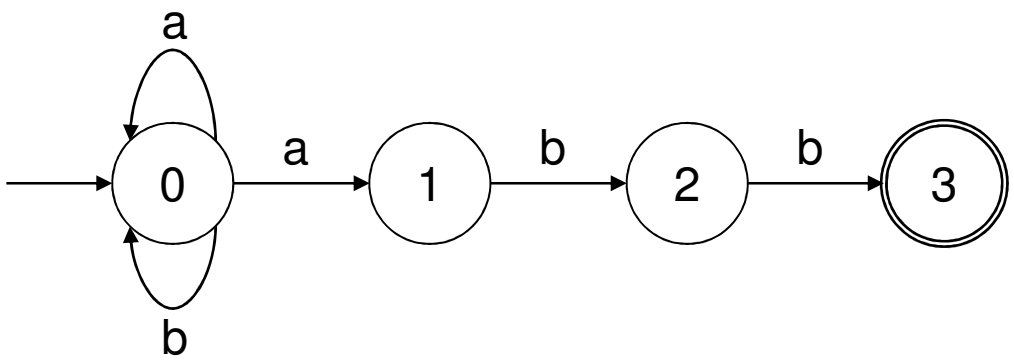
Autômatos finitos

- Modelos matemáticos
  - Conjunto de estados S
  - Conjunto de símbolos de entrada Σ
  - Funções de transição que mapeiam um par estado-símbolo de entrada em um novo estado
  - Um estado inicial s_0
  - Um conjunto de estados finais F para aceitação de cadeias

- Reconhecimento de cadeias válidas
  - Uma cadeia é reconhecida se existe um percurso do estado inicial até um estado final

Exemplo

- S={0,1,2,3}, Σ={a,b}, s_0=0, F={3}

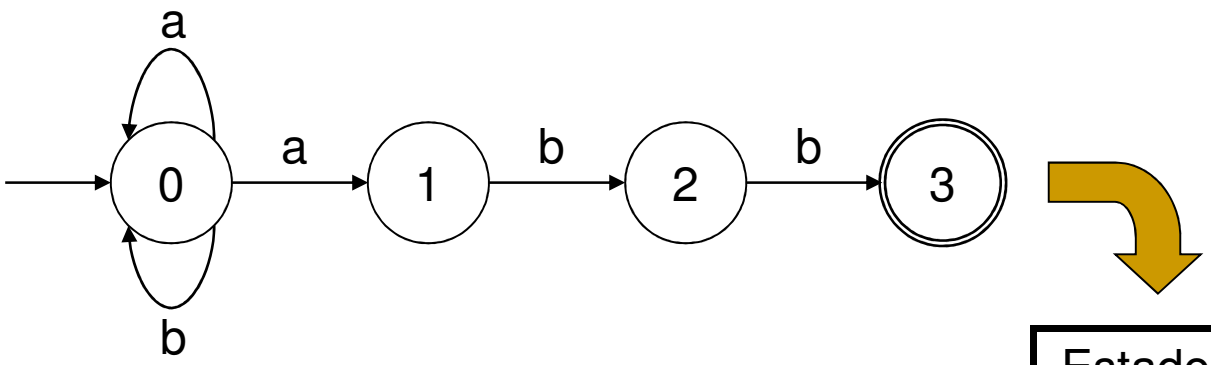


Quais cadeias esse autômato aceita?

$(a \mid b)^*abb$

Exemplo

- Representação em tabela de transição
  - Vantagem: elegância e generalidade
  - Desvantagem: pode ocupar grande espaço quando o alfabeto de entrada é grande; processamento mais lento



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

Execução do autômato

- Se 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

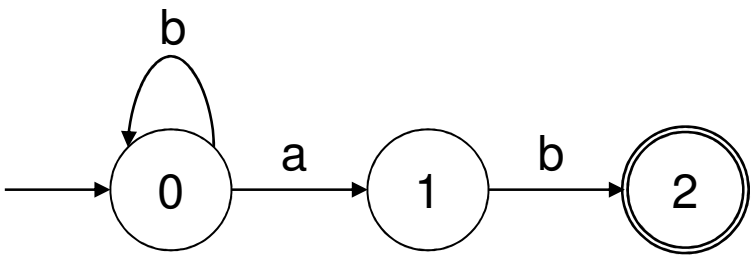
```
s:=s0
c:=próximo_caractere()
enquanto (c<>eof) faça
        s:=transição(s,c)
        c:=próximo_caractere()
fim
se s for um estado final
    então retornar “cadeia aceita”
    senão retornar “falhou”
```

Exemplo de execução do autômato

```
s:=s0
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”
```

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

$S=\{0,1,2\}$, $\Sigma=\{a,b\}$, $s_0=0$, $F=\{2\}$



Reconhecer cadeia bab

Execução do autômato

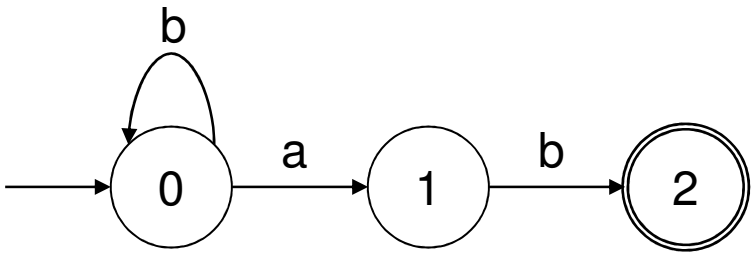
- Se autômato não determinístico, pode-se transformá-lo em um autômato determinístico

- Para a aplicação em compiladores, em geral, é muito simples construir um autômato determinístico

Execução do autômato

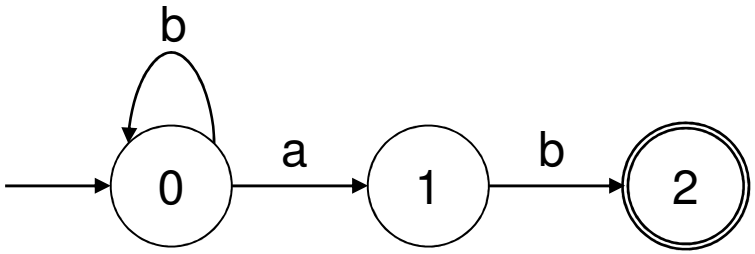
- Opção: incorporação das transições no código do programa
  - Tabela de transição não é mais necessária

```
s:=s0
enquanto (verdadeiro) faça
    c:=próximo_caractere()
    case (s)
       0: se (c=a) então s:=1
          senão se (c=b) então s:=0
          senão retornar “falhou”
       1: se (c=b) então s:=2
          senão retornar “falhou”
       2: se (c=eof) então retornar “cadeia aceita”
          senão retornar “falhou”
fim
```



Execução do autômato

- Solução *ad hoc*




```
c:=próximo_caractere()
se (c='b') então
        c:=próximo_caractere()
        enquanto (c=b) faça
                c:=próximo_caractere()
        se (c='a') então
                c:=próximo_caractere()
                se (c='b') e (acabou cadeia de entrada) então retornar "cadeia aceita"
                senão retornar "falhou"
        senão retornar "falhou"
senão se (c='a') então
        c:=próximo_caractere()
        se (c='b') e (acabou cadeia de entrada) então retornar "cadeia aceita"
        senão retornar "falhou"
senão retornar "falhou"
```

Tokens de um programa

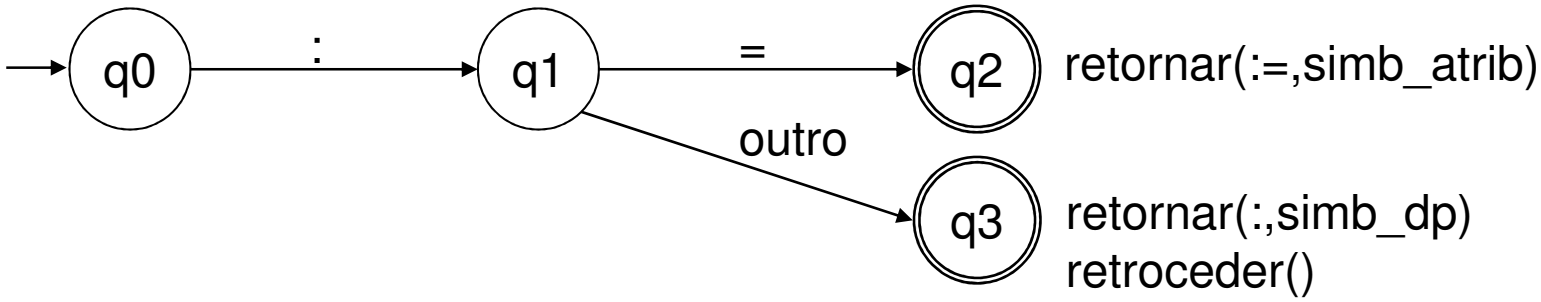
- Exemplos de tokens possíveis
  - Identificadores: x, y, minha_variável, meu_procedimento
  - Palavras reservadas e símbolos especiais: while, for, :=, <>
  - Números inteiros e reais

- Não basta identificar o token, deve-se retorná-lo ao analisador sintático junto com a cadeia correspondente
  1. Concatenação da cadeia conforme o autômato é percorrido
  2. Associação de ações aos estados finais do autômato

- Às vezes, para se decidir por um token, tem-se que se ler um caractere a mais, o qual deve ser devolvido à cadeia de entrada depois

Tokens de um programa

- Autômato para os símbolos := e :

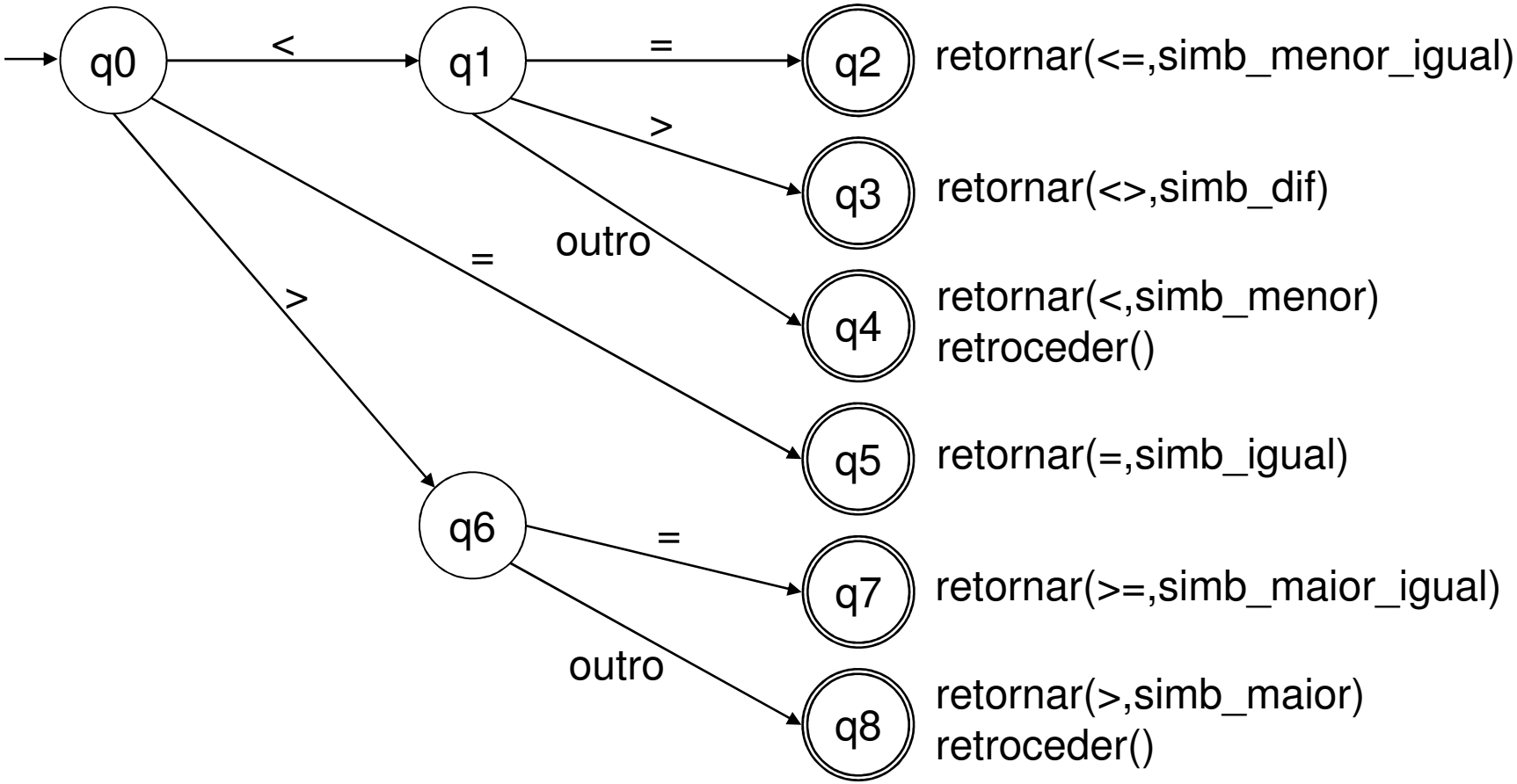


Tokens de um programa

- Exercício: autômato para operadores relacionais >, >=, <, <=, = e <>

Tokens de um programa

- Exercício: autômato para operadores relacionais >, >=, <, <=, = e <>

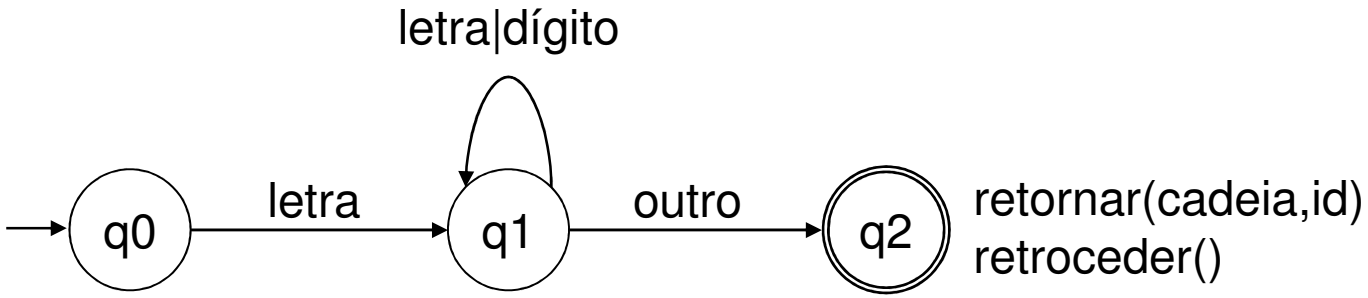


Tokens de um programa

- Autômato para identificadores: letra seguida de qualquer combinação de letras e dígitos

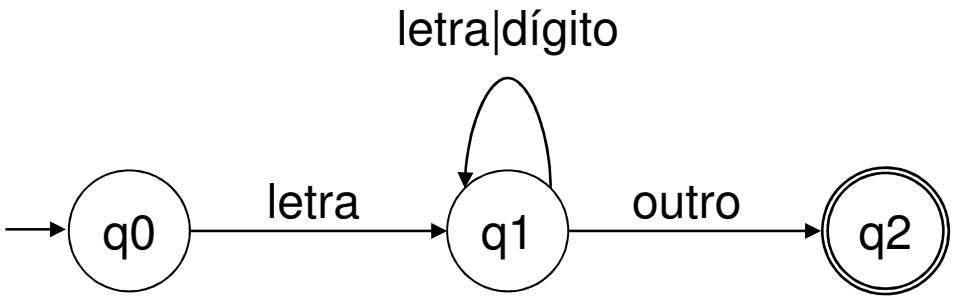
Tokens de um programa

- Autômato para identificadores: letra seguida de qualquer combinação de letras e dígitos



Tokens de um programa

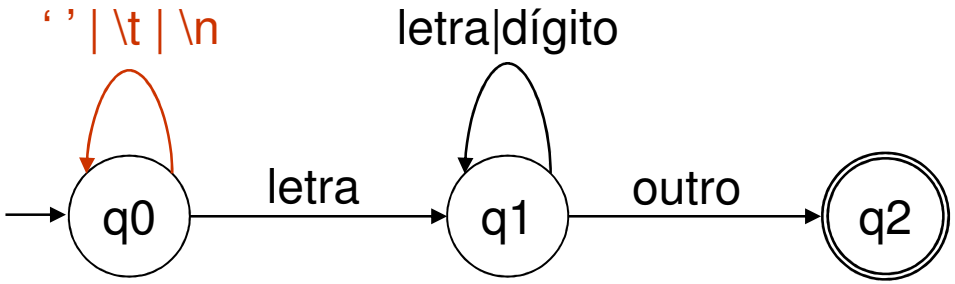
- Autômato para palavras reservadas: while, if, for, array, etc.
- Opções
  - Fazer um autômato para cada palavra-reservada
    - Trabalhoso e ineficiente
  - Deixar que o autômato para identificadores reconheça as palavras reservadas e, ao final, verifique na tabela de palavras reservadas se se trata de uma palavra reservada
    - Simples e elegante




```
se busca_tabela(cadeia)=verdadeiro
      então retornar(cadeia,cadeia/token)
      senão retornar(cadeia,id)
retroceder()
```

Tokens de um programa

- Autômato para consumir caracteres não imprimíveis: espaços em branco, tabulações e códigos de nova linha
  - O analisador léxico não deve produzir tokens para esses símbolos



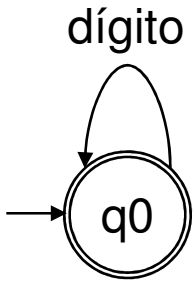

```
se busca_tabela(cadeia)=verdadeiro
      então retornar(cadeia,cadeia/token)
      senão retornar(cadeia,id)
retroceder()
```

Tokens de um programa

- Exercício
  - Construir autômatos para se reconhecer
    - Números inteiros com e sem sinal: 5, -1, 100
    - Números reais: 3.11, 0.1

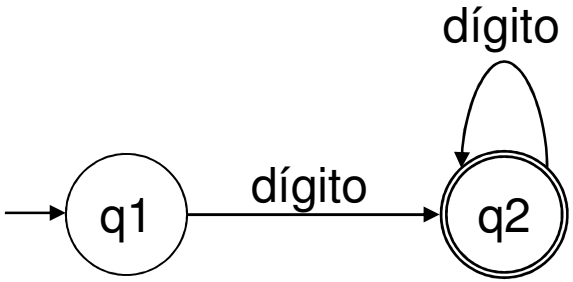
Tokens de um programa

- Autômato para números inteiros sem sinal
  - É apropriado? Por quê?



Tokens de um programa

- Autômato para números inteiros sem sinal
  - É apropriado? Por quê?



Tokens de um programa

- Exercício
  - Construir autômato para consumir comentários
    - {essa função serve para...}
    - /*essa função serve para...*/

Analisador léxico

- Conjunto de procedimentos que reconhecem cadeias válidas e lhes associam tokens
- Cada vez que é chamado, retorna par cadeia-token para o analisador sintático
- Consome caracteres irrelevantes: espaços em branco, tabulações, códigos de nova linha, comentários
- Produz mensagens de erro apropriadas quando uma cadeia não é reconhecida por algum autômato
  - Não se atinge o estado final do autômato
  - Tratamento apropriado na implementação do analisador léxico

Tratamento de erros

- Símbolo não pertencente ao conjunto de símbolos terminais da linguagem / identificador mal formado: j@
  - Não há autômato para reconhecer esses símbolos
- Número mal formado: 2.a3
  - Estado final do autômato de números reais não é atingido
- Tamanho do identificador / tamanho excessivo de número: minha_variável_para_..., 555555555555555
  - Dependência da especificação da linguagem
  - Verificável por ação associada ao estado final dos autômatos
- Fim de arquivo inesperado (comentário não fechado): {...
  - Dependência da especificação da linguagem: comentários de várias linhas?
  - Estado final do autômato de comentários não é atingido
- Char ou string mal formados: ‘a, “hello world
  - Dependência da especificação da linguagem: um único token ou conjunto de tokens?

Tratamento de erros

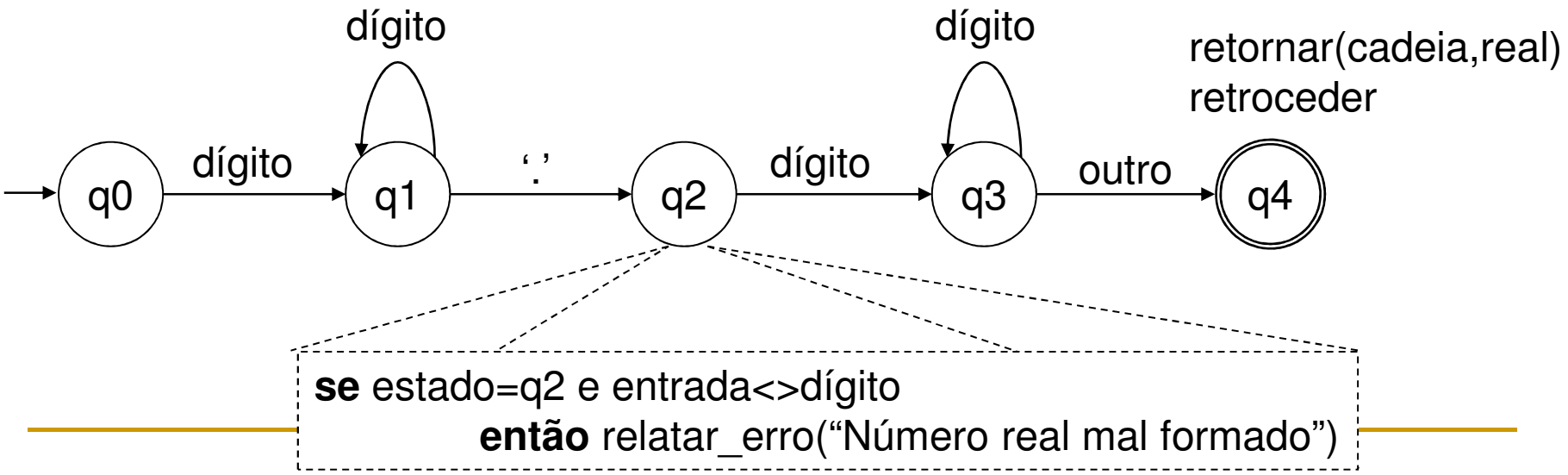
- Compilação não pode parar
  - Erros devem ser sempre relatados
    - <#,erro_léxico> ou <#,nada> ou <#,caractere_inválido>

- Opções para recuperação de erro: beg#in
  - Retornar par <beg#,ERRO> e, na próxima chamada, <in,id>
  - Separar o caractere ilegal em um outro pacote
    - Para chamadas sucessivas, retorna pares <beg,id>, <#,ERRO> e <in,id>

- Classificação dos erros
  - Não distinguir erros léxicos, i.e., associa-se um token 'nada' (ou qualquer outro que indique um erro) ao erro e deixa-se a identificação do erro para uma próxima etapa
    - Retorna pares <beg,id>, <#,nada> e <in,id>
  - Análise mais informada do erro léxico (slide anterior)

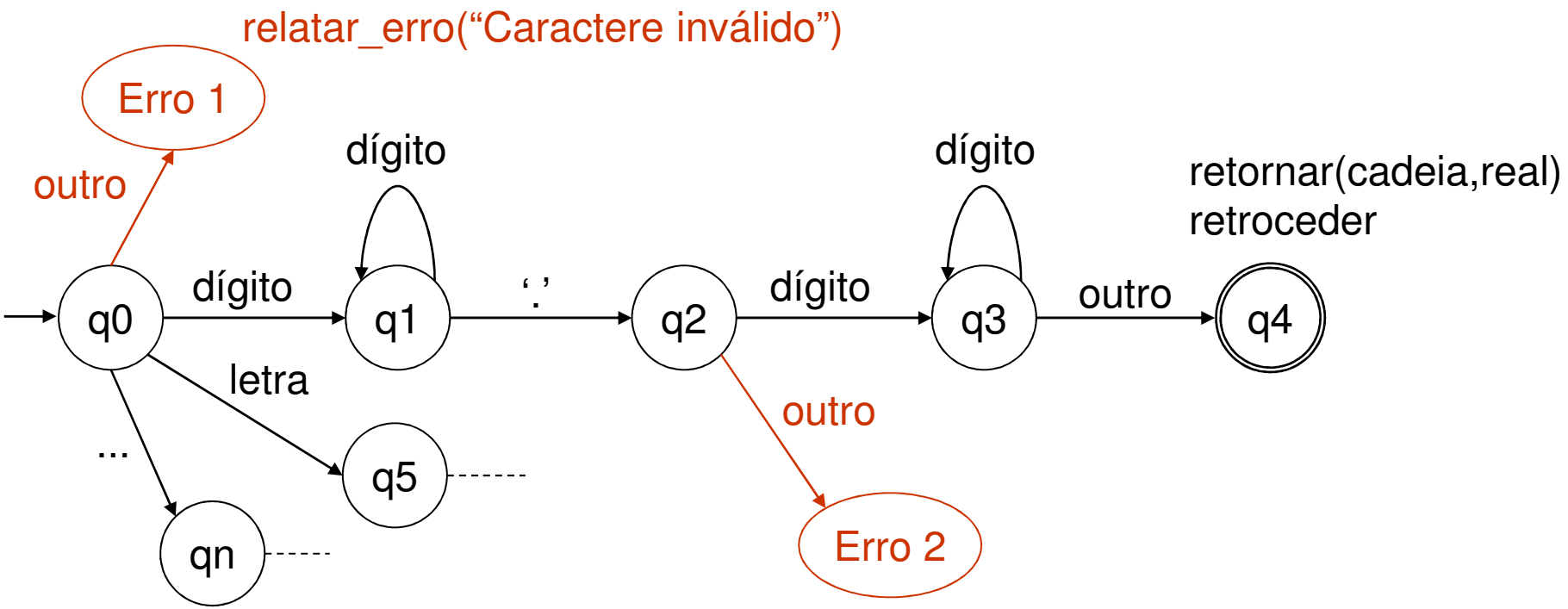
Tratamento de erros

- Algumas opções
  - Associar tratamento de erros individuais a cada estado do autômato, de forma que haja uma relação unívoca entre o estado e o erro possível
    - Vantagem: autômato mais compacto
  - Exemplo: autômato para números reais



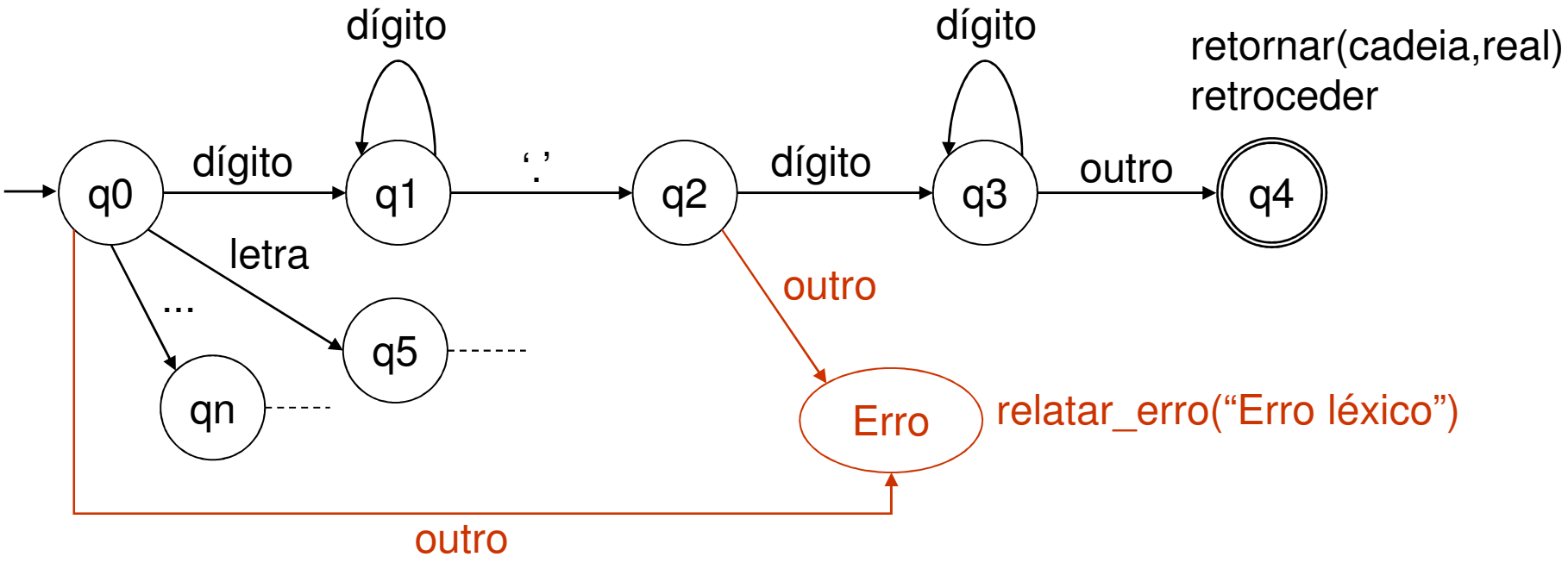
Tratamento de erros

- Algumas opções
  - Criar estados extras no autômato: estados de erro
    - Individuais (um para cada tipo de erro): tratamento mais informado
      - Vantagem: maior clareza



Tratamento de erros

- Algumas opções
  - Criar estados extras no autômato: estados de erro
    - Genéricos: erros podem não ser diferenciados
      - Vantagem: maior clareza



Tratamento de erros

- Exercício: adicione o tratamento de erros no autômato de identificadores

- Como os seguintes erros seriam reconhecidos?
  - @minha_variavel
  - minha@_variavel
  - minha_variavel@

Questões de implementação

- Tabela de palavras reservadas
  - Carregada no início da execução do compilador
  - Busca deve ser eficiente
    - *Hashing*, sem colisões

- Reconhecimento de tokens
  - Criação e manutenção de um buffer
    - Facilidade de leitura e devolução de caracteres
  - Ter sempre um caractere lido no início de um processamento (símbolo *lookahead*)
    - Uniformidade e consistência na análise léxica

Analisador léxico

- Idealmente, deveria ser representado por apenas um autômato com vários estados finais, possivelmente

- Cada estado final deve conter ações semânticas adequadas relativas à manipulação das cadeias e tokens identificados
  - Retroceder
  - Retornar
  - Outras?

- Erros devem ser tratados devidamente, com mensagens de erros precisas e significativas para o usuário

Questões de implementação

- O analisador léxico é uma das fases que mais consome tempo na compilação
  - Entre 20 e 30% do tempo de compilação

- Bom planejamento é necessário