# Aula 01 – Análise Léxica (*Scanning*)

Prof. Eduardo Zambon

Departamento de Informática (DI)
Centro Tecnológico (CT)
Universidade Federal do Espírito Santo (Ufes)

**Compiladores**
***Compiler Construction* (CC)**

# Introdução

- Por ser um sistema bastante complexo, um compilador é dividido em módulos e a compilação em fases.
- O primeiro módulo do compilador realiza a análise léxica (*scanning*).
- **Estes *slides***: discussão sobre os principais conceitos de análise léxica.
- **Objetivos**: apresentar a teoria que levou ao desenvolvimento e construção de *scanners*.

## Referências

**Chapter 2 – Scanning**
*K. C. Louden*
**Chapter 2 – Scanners**
*K. D. Cooper*
**Chapter 3 – Scanning**
*D. Thain*

# Função de um *Scanner*

- O analisador léxico (*scanner*) é o primeiro módulo que compõe o *front-end* do compilador.
- O *scanner* recebe como entrada o arquivo do programa fonte que deve ser compilado.
- Função do *scanner*: realizar a leitura de todos os caracteres do arquivo de entrada e agrupá-los em *tokens*.
- Tipos de *tokens* são entidades lógicas geralmente definidas por um tipo enumerado:

```
typedef enum {
  IF, THEN, ELSE, PLUS, MINUS, NUM, ID, ...
} TokenType;
```

- Em dialetos de C mais antigos a enumeração pode ser substituída por macros.

# Categorias de *Tokens*

- Os diferentes tipos de *tokens* podem ser agrupados em categorias.
- Algumas categorias usuais:
  - Palavras reservadas: como IF e THEN, que representam as *strings* “if” e “then”.
  - Símbolos especiais: como PLUS e MINUS, que representam os caracteres ‘+’ e ‘-’.
  - Outros tipos: como NUM e ID, que representam números e identificadores.

# Relação entre Tipos de *Tokens* e suas *Strings*

- Lexema: *string* associada ao *token*.
- Alguns tipos de *tokens* possuem um único lexema.
  *Exemplo*: palavras reservadas.
- Já outros tipos podem ser associados a infinitos lexemas.
  *Exemplo*: identificadores.

- Atributos do *token*: qualquer informação associada a um *token*.
  - O lexema é um atributo.
  - Um *token* do tipo NUM pode ter um atributo *string* "32767" e um outro atributo inteiro com o valor real 32767.
- $\Rightarrow$ Um *token* é uma tupla que reúne o tipo do *token* e a coleção de todos os seus atributos.

# Aspectos Práticos de um *Scanner*

É possível agrupar todas as informações de um *token* em uma estrutura.

```
typedef struct {
  TokenType type;
  char* lexeme;
  int value;
} TokenRecord;
```

Na ferramenta `flex` isso é feito de outra forma.

- Lexema: variável global `yytext`.
  - Tipo é `char*`.
  - Código gerado pelo `flex` cuida de ajustar o tamanho do vetor conforme a necessidade.
- Valor do *token*: variável global `yylval`.
  - Tipo definido pela macro `YYSTYPE`.
  - Padrão é `int`.
- Tipo do *token*: valor de retorno da função `yylex`.

# Aspectos Práticos de um *Scanner*

- Função do *scanner*: converter um *stream* de caracteres em um *stream* de *tokens*.
- Normalmente isso é feito sob demanda. Analisador léxico é executado de forma incremental.
- Controle é feito pelo analisador sintático (*parser*) que pede o próximo *token* ao *scanner*.
- Generalizado na Seção 2.1 do livro do Louden como a função `getToken`.
- No `flex` isso corresponde a uma chamada à função abaixo.

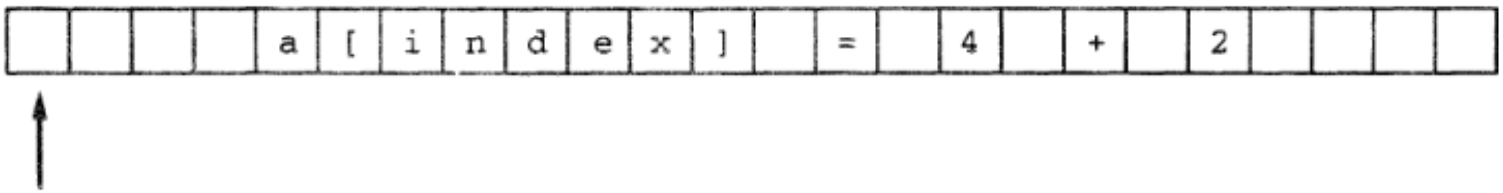
```
int yylex(void);
```

# Aspectos Práticos de um *Scanner*

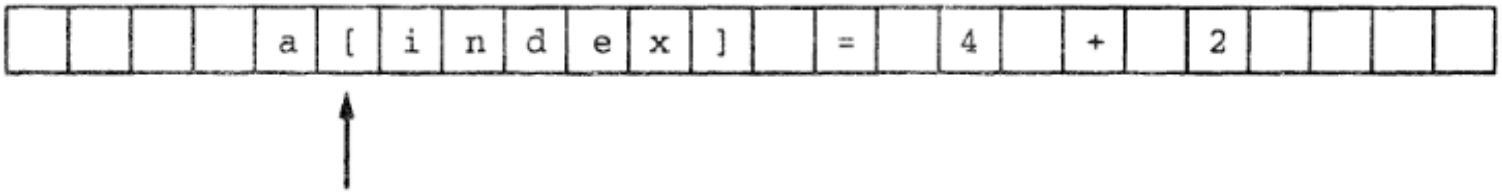
Como exemplo do funcionamento de um *scanner*, considere a seguinte linha de código como entrada:

```
a[index] = 4 + 2
```

Suponha que a linha acima esteja armazenada em um *buffer*, com o próximo caractere de entrada indicado pela seta.




Supondo que o *scanner* foi construído para ignorar espaços em branco, uma chamada de `getToken` descarta os próximos quatro brancos e retorna o *token* `<ID,"a">`.



# Linguagens e Expressões Regulares

- Linguagens Regulares são o tipo mais simples de linguagem segundo a Hierarquia de Chomsky (tipo 3).
- Possuem uma definição recursiva através de operações fundamentais sobre conjuntos de *strings*.
- Mais facilmente representáveis por expressões regulares (*regular expressions* – REs).
- REs representam padrões de *strings*. É o método usual de descrição dos *tokens* de uma LP.
- Uma RE é definida pelo conjunto de *strings* com as quais há um casamento da expressão.
- Se $r$ é uma RE, esse conjunto é $L(r)$, dito linguagem de $r$.
- *Strings* em $L(r)$ são formadas por símbolos de um alfabeto $\Sigma$.
- Uma RE é escrita com elementos de $\Sigma$ mais alguns caracteres especiais (meta-símbolos).

# Linguagens e Expressões Regulares

- Para todo $a \in \Sigma$: $L(a) = \{a\}$.
- A letra $\epsilon$ denota a *string* vazia, e $L(\epsilon) = \{\epsilon\}$.
- O símbolo $\emptyset$ denota a linguagem vazia: $L(\emptyset) = \{\}$.
- Operações fundamentais em REs são:
  - Escolha (união):
    $L(a \mid b) = L(a) \cup L(b) = \{a\} \cup \{b\} = \{a, b\}$.
  - Concatenação: $L(ab) = L(a)L(b) = \{ab\}$.
  - Repetições (fecho ou estrela de Kleene):
    $L(a^*) = \{\epsilon, a, aa, aaa, \ldots\}$.
- Demais operações podem ser derivadas das fundamentais.
  - Faixas: $[0 - 9] \equiv (0|1|2|3|4|5|6|7|8|9)$.
  - Negação: $[\hat{}\, a] \equiv \Sigma \setminus \{a\}$.
  - Um ou mais: $a^+ \equiv aa^*$.
  - Zero ou um: $a? \equiv (a \mid \epsilon)$.
- Ferramentas como `flex` oferecem mais opções. (Veja roteiro do Laboratório 01.)

# Expressões Regulares

Exemplo 1:

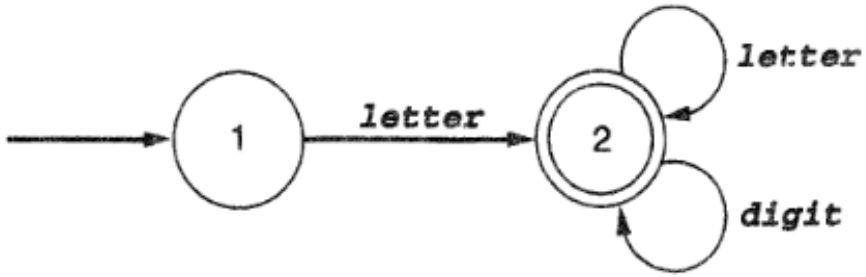
- $\Sigma = \{a, b, c\}$.
- $L$ = *strings* sobre $\Sigma$ que contém exatamente um $b$.
- $(a|c)^* b (a|c)^*$.

Exemplo 2:

- $\Sigma = \{a, b\}$.
- $L = \{ab, aabb, aaabbb, \ldots\} = \{a^n b^n \mid n > 0\}$.
- A linguagem acima não pode ser expressa por uma RE.
- $\Rightarrow$ "REs não conseguem contar."
- Se $n$ tiver um limite superior é possível reconhecer.
- Essa limitação pode causar alguns problemas para os *scanners*: algumas LPs admitem comentários aninhados, por exemplo.

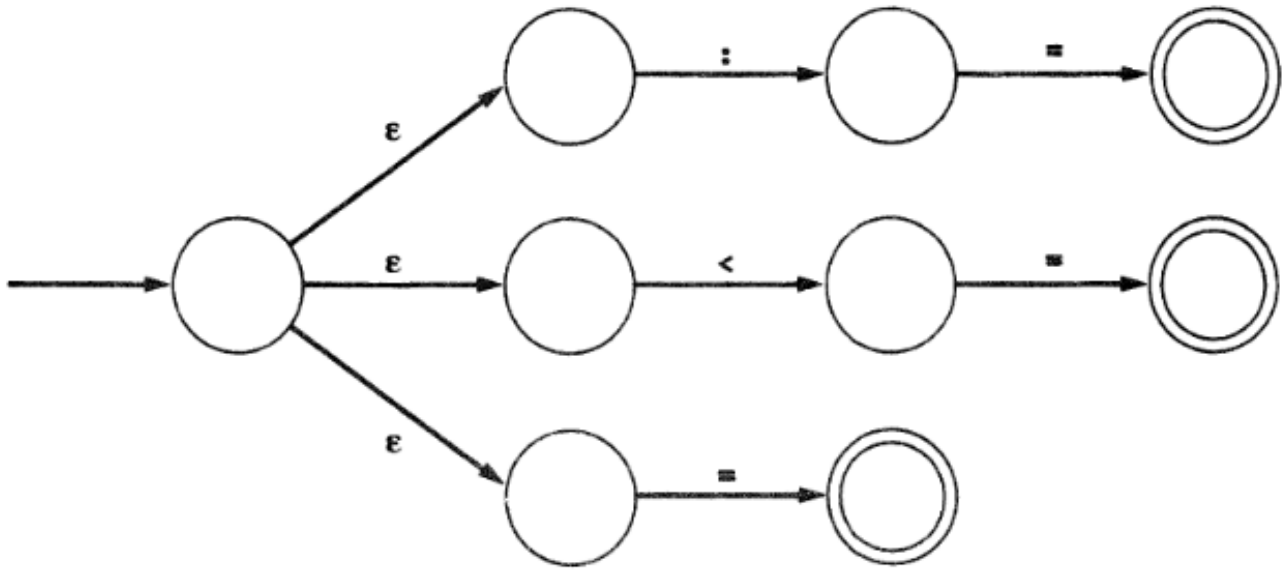
# Autômatos Finitos

- Linguagens regulares são reconhecidas por autômatos finitos.
- Quanto não há possibilidade de escolhas nas transições, o autômato é dito determinístico (*deterministic finite automata* – DFA).
- *Exemplo*: a RE `letter(letter|digit)*` é reconhecida pelo DFA abaixo.



# Autômatos Finitos

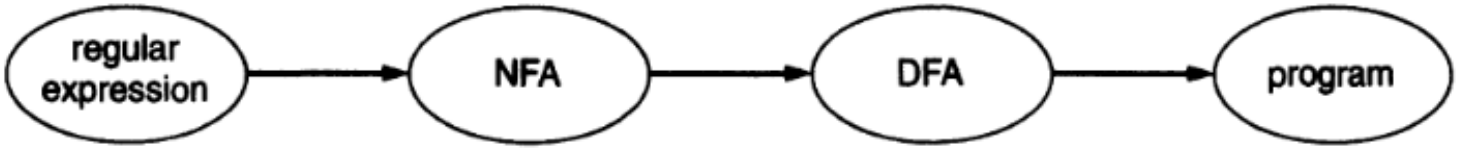
- É muito conveniente utilizar transições que não consomem a entrada (transições $\epsilon$).
- Possibilitam a construção de escolhas nas transições.
- Nesse caso, o autômato é dito não-determinístico (*non-deterministic finite automata* – NFA).
- *Exemplo*: autômato que reconhece os *tokens* $:=$, $<=$ e $=$.




(Aproveite o momento para relembrar o critério de aceite de um NFA, visto na disciplina de Linguagens Formais – LFA.)

# De Expressões Regulares para *Scanners*

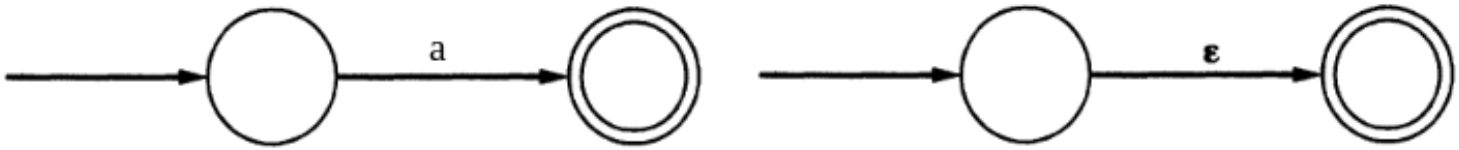
- Os *tokens* de uma LP podem ser facilmente descritos por REs.
- $\Rightarrow$ REs servem como uma linguagem de especificação de *tokens*.
- Por outro lado, buscamos um programa (módulo) que realiza a análise léxica.
- $\Rightarrow$ É necessária uma sequência de transformações como abaixo.




- A seguir, vamos estudar cada uma dessas transformações.

# De REs para NFAs

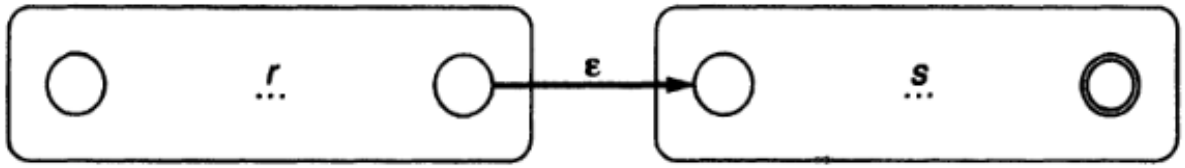
- Essa transformação é construtiva: a partir de NFAs fundamentais, são construídos NFAs cada vez maiores até chegar no resultado final.
- Método conhecido como Construção de (Ken) Thompson.
- Utiliza transições $\epsilon$ para "colar" cada pedaço (NFA) de uma RE.
- As REs básicas são $a \in \Sigma$, $\epsilon$ e $\emptyset$.
- Correspondem aos NFAs abaixo.



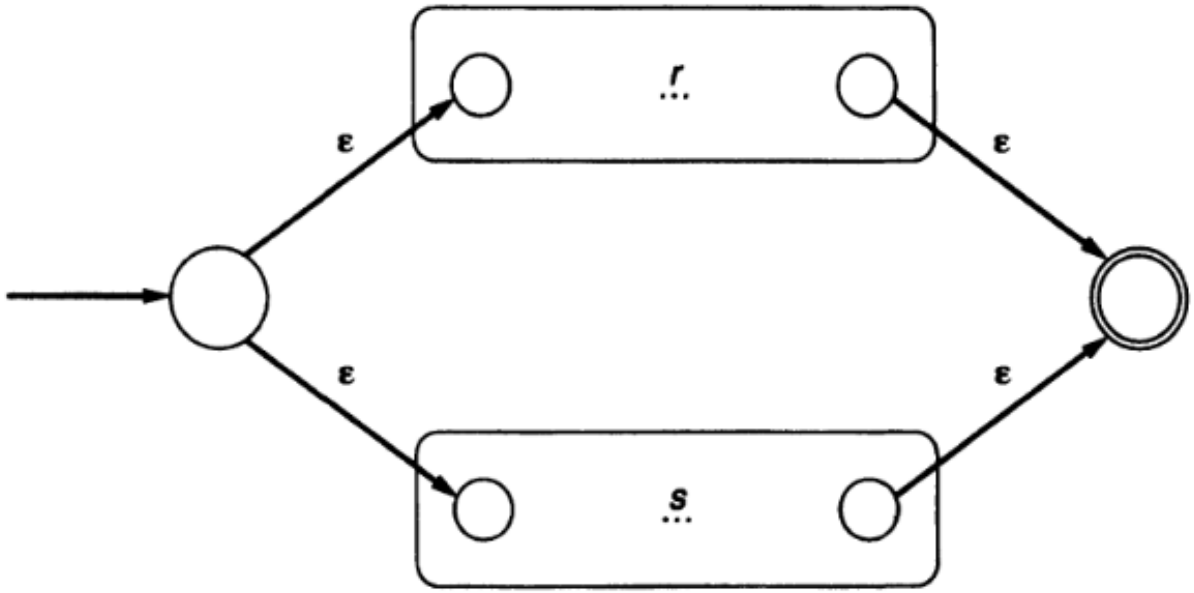

- A RE $\emptyset$ não ocorre na prática: não existe um *token* sem ao menos um lexema associado.

# De REs para NFAs

- Concatenação: se $r$ e $s$ são REs com respectivos NFAs associados, o NFA abaixo aceita $L(rs)$.

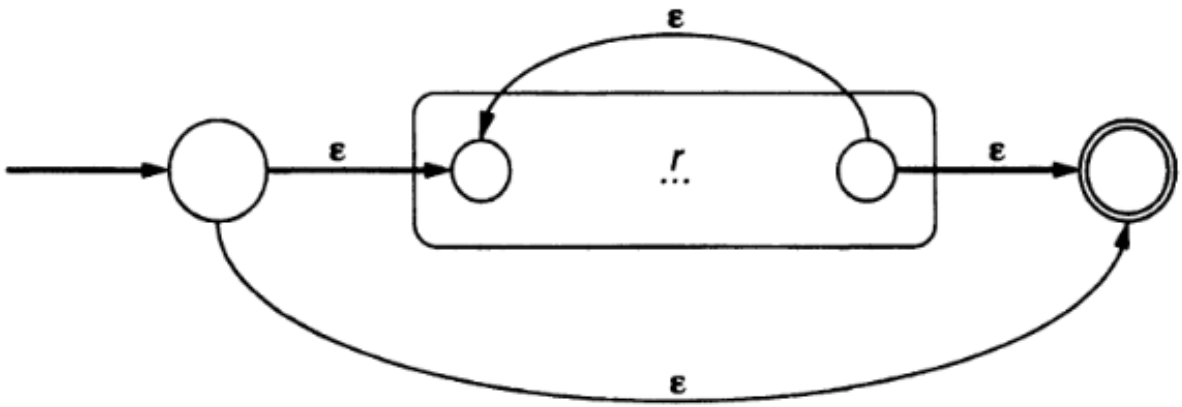



- Escolha: construção para $L(r\,|s)$.

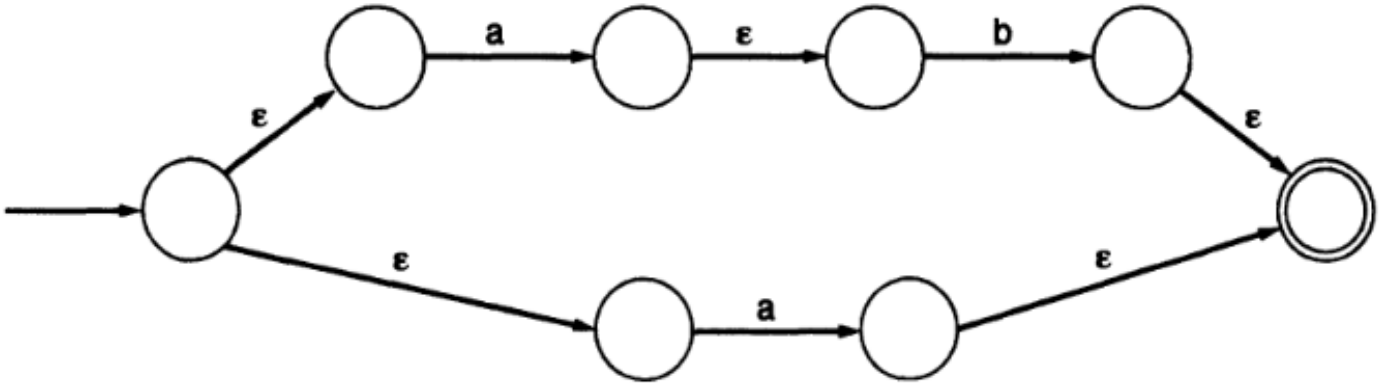


# De REs para NFAs

- Repetição: se $r$ é uma RE com um respectivo NFA associado, o NFA abaixo aceita $L(r^*)$.



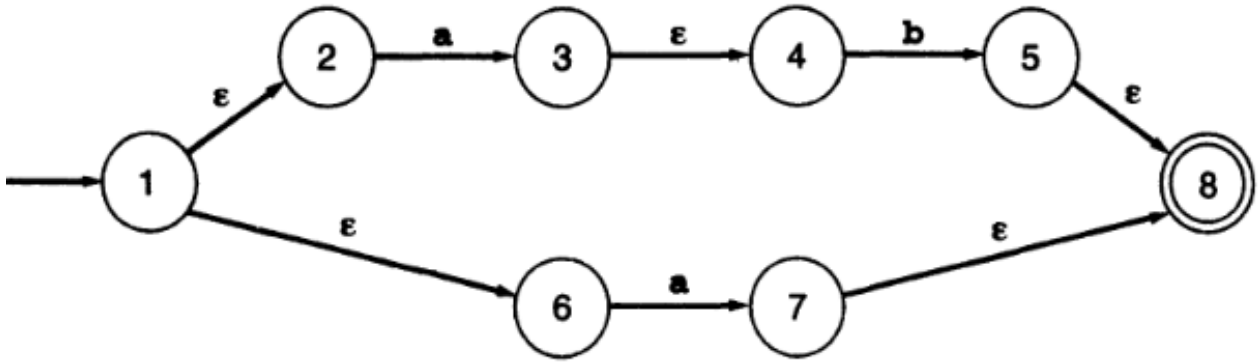

- *Exemplo*: Aplicando a Construção de Thompson para a RE $(ab|a)$, obtemos o NFA abaixo.



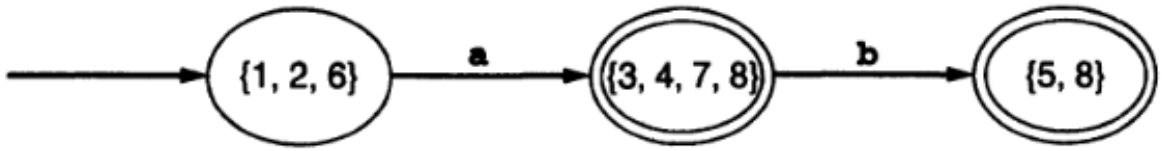
# De NFAs para DFAs

- Transformação pode ser automatizada através do algoritmo de construção de subconjuntos.
- Visto na disciplina de LFA, não será discutido aqui.

*Exemplo*: Dado o NFA abaixo




a determinização pelo algoritmo de construção de subconjuntos gera o seguinte DFA.



# Minimização de DFAs

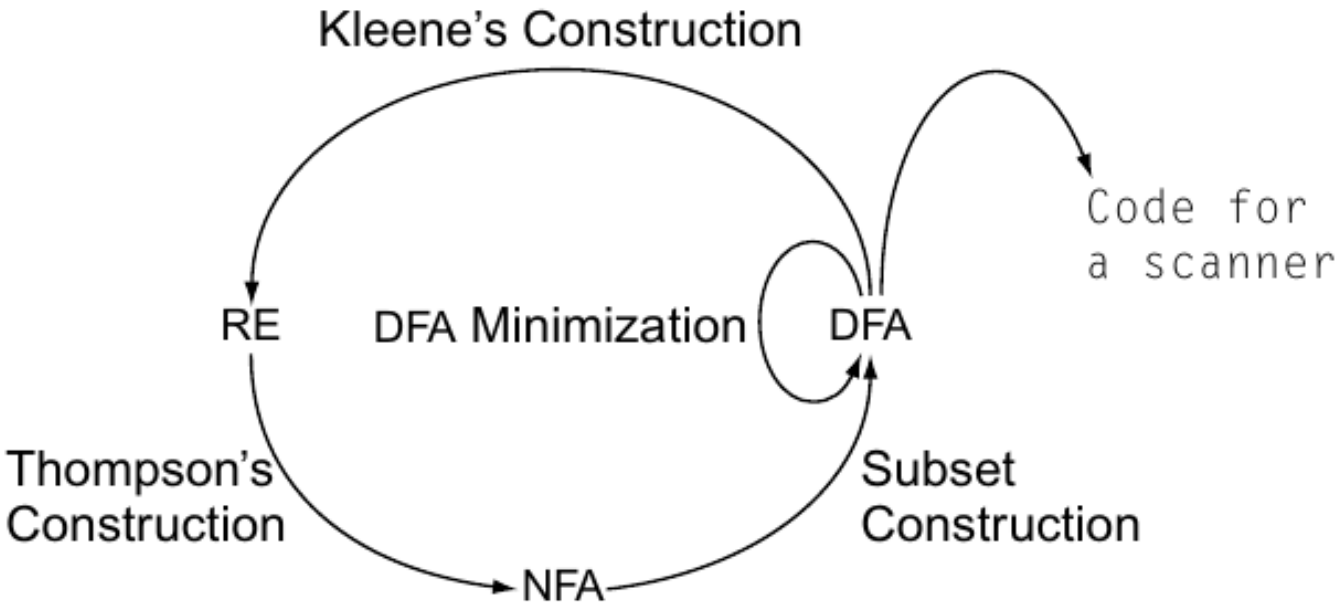
- O processo de derivar um DFA de forma algorítmica a partir de uma RE pode levar a um DFA complexo.
- Felizmente, para qualquer DFA *a*, existe um DFA equivalente *m* contendo um número mínimo de estados e esse DFA *m* é único.
- Implementável através de um algoritmo de *partition refinement*, baseado na construção de classes de equivalência de estados.
- Diferentes variantes propostas:
  - Algoritmo de Moore (1956).
  - Algoritmo de Brzozowski (1963).
  - Algoritmo de Hopcroft (1971).
- Também conteúdo de LFA, não será discutido aqui.

# De REs para DFAs

- Construção automática:
  1. Construir um NFA a partir da RE (Construção de Thompson).
  2. Transformar o NFA em DFA (Construção de subconjuntos).
  3. Minimizar o DFA (Algoritmo de Hopcroft).
  4. Usar o DFA para aceitar/reconhecer *tokens*.
- Passo dois pode "explodir" o número de estados do DFA.
  - Cada estado do DFA corresponde a um conjunto de estados do NFA.
  - Se o NFA tem $n$ estados, quantos estados o DFA pode ter?
  - $\Rightarrow 2^n$ estados (tamanho do conjunto potência de $n$).
- Normalmente isto não é um problema na prática.
  - Explosão só ocorre se a linguagem tem muitos *tokens* similares.
  - Isto é evitado no projeto da LP: também é inconveniente para humanos.

# De REs para DFAs

O processo discutido até aqui pode ser resumido na figura abaixo.



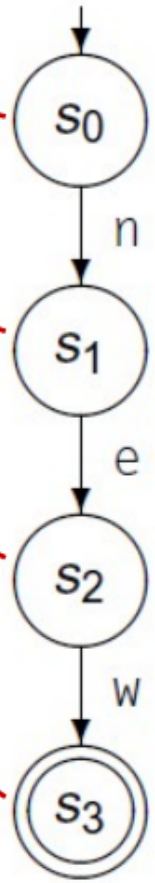

(A Construção de Kleene não foi abordada por não ser necessária na construção de um *scanner*.)

# De DFAs para Código

Reconhecendo o *token* da palavra reservada `new` (pseudo-código):

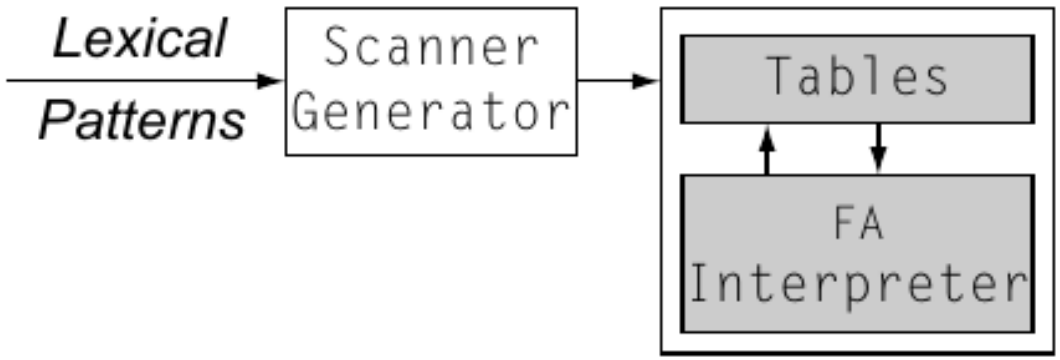
```
c ← NextChar();
if (c = 'n')
   then begin;
      c ← NextChar();
      if (c = 'e')
         then begin;
            c ← NextChar();
            if (c = 'w')
               then report success;
               else try something else;
         end;
         else try something else;
   end;
   else try something else;
```

Código ao lado simula o DFA abaixo:



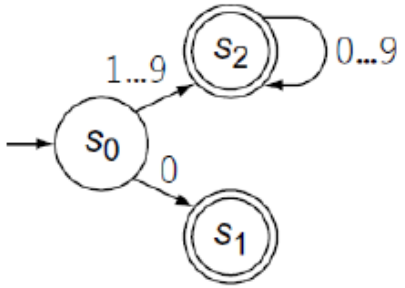
# Análise Léxica Direcionada por Tabelas

- Geradores de *scanners* como o `flex` não geram um código específico para cada DFA como mostrado no slide anterior.
- Ao contrário, é emitido um código geral de um interpretador de autômatos, cujo funcionamento é orientado por tabelas.
- Basta então gerar as tabelas de acordo com cada DFA.



# Algoritmo Geral de Aceite de um Único *Token*

Partindo de um DFA:




Crie uma tabela de transição:

- $\delta$: função transição
- $s_e$: estado de erro

| $\delta$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | Other |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $s_0$ | $s_1$ | $s_2$ | $s_2$ | $s_2$ | $s_2$ | $s_2$ | $s_2$ | $s_2$ | $s_2$ | $s_2$ | $s_e$ |
| $s_1$ | $s_e$ | $s_e$ | $s_e$ | $s_e$ | $s_e$ | $s_e$ | $s_e$ | $s_e$ | $s_e$ | $s_e$ | $s_e$ |
| $s_2$ | $s_2$ | $s_2$ | $s_2$ | $s_2$ | $s_2$ | $s_2$ | $s_2$ | $s_2$ | $s_2$ | $s_2$ | $s_e$ |
| $s_e$ | $s_e$ | $s_e$ | $s_e$ | $s_e$ | $s_e$ | $s_e$ | $s_e$ | $s_e$ | $s_e$ | $s_e$ | $s_e$ |

Execute o algoritmo: ($S_A$ é o conjunto de estados de aceite do DFA.)

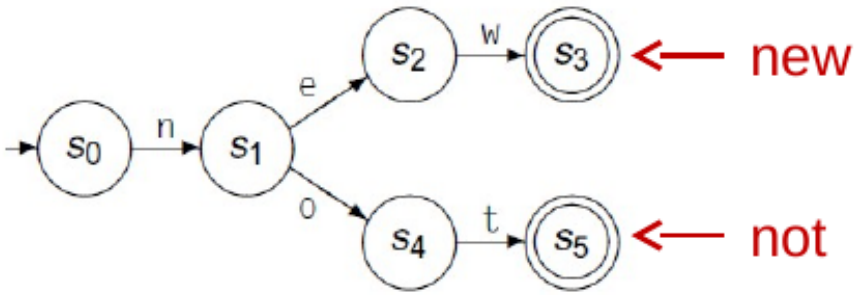
```
char ← NextChar( );
state ← s0 ;

while (char ≠ eof and state ≠ se) do
   state ← δ(state,char);
   char ← NextChar( );
end;

if (state ∈ SA)
    then report acceptance;
    else report failure;
```

# Combinando Vários Tipos de *Token*

- Uma LP real tem variados tipos de *token* (categorias).
  - Java: mais de 100 tipos de *token*.
  - Cada tipo de *token* é reconhecido por um DFA.
- DFAs distintos podem ser combinados em um único DFA.
  - Como distinguir os tipos de *token* no DFA composto?
  - $\Rightarrow$ Rotular os estados de aceite com o tipo de *token* correspondente.




  - Rótulos indicam o valor de retorno da função `getToken`.
- E se houver sobreposição dos tipos de *token*?
  - E.g., o tipo `ID` se sobrepõe a toda palavra reservada.
  - Ambiguidade: resolvida pela ordenação dos tipos na especificação.

# Reconhecendo Múltiplos *Tokens* em Sequência

- Entrada: *string* contendo múltiplos *tokens*.
- Problema: não sabemos aonde os *tokens* devem terminar.
- Nem todo *token* tem um ponto de término único.
- Por exemplo, como reconhecer:
  - `x==2`?
  - `x=2`?
- Escolha: usar um algoritmo guloso.
- Mesmo após reconhecer um *token*, tentar extendê-lo para um maior.
- Se a extensão falhar, fazer *backtrack* até o último *token* encontrado.
- Nunca faça *backtrack* a ponto de descartar um *token* que já tenha sido aceito!

# Algoritmo Geral para *Scan* de Múltiplos *Tokens*

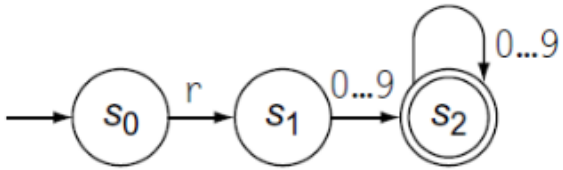
```
NextWord()
  state ← s0;
  lexeme ← "";
  clear stack;
  push(bad);
  while (state≠se) do
    NextChar(char);
    lexeme ← lexeme + char;
    if state ∈ SA
        then clear stack;
    push(state);
    cat ← CharCat[char];
    state ← δ[state,cat];
  end;
  while(state ∉ SA and
        state≠bad) do
    state ← pop();
    truncate lexeme;
    RollBack();
  end;
  return Type[state];
```

Simule o DFA até chegar no estado de erro

Guarde o caminho

Caso tenha *overshoot* faça *rollback* até o último *token* aceito

Exemplo: r(0..9)+
(nomes de regs.)




| r | 0, 1, 2, ..., 9 | EOF | **Other** |
|---|---|---|---|
| *Register* | *Digit* | *Other* | *Other* |

The Classifier Table, *CharCat*

| | ***Register*** | ***Digit*** | ***Other*** |
|---|---|---|---|
| $s_0$ | $s_1$ | $s_e$ | $s_e$ |
| $s_1$ | $s_e$ | $s_2$ | $s_e$ |
| $s_2$ | $s_e$ | $s_2$ | $s_e$ |
| $s_e$ | $s_e$ | $s_e$ | $s_e$ |

The Transition Table, $\delta$

| $s_0$ | $s_1$ | $s_2$ | $s_e$ |
|---|---|---|---|
| *invalid* | *invalid* | *register* | *invalid* |

The Token Type Table, *Type*

# Preocupações com Micro-Eficiência

- Tabelas do slide anterior foram quebradas para ficarem mais simples e consumirem menos memória.
- Considerações adicionais de eficiência:
  - Evitar excesso de *rollback*.
  - Otimizar o acesso à tabela de *lookup*: não armazenar a tabela inteira na memória.
  - Otimizar o acesso aos *buffers* de entrada: não armazenar o programa inteiro como uma única *string*.
  - Como armazenar e comparar lexemas?
- Variações do algoritmo geral de *scan*.
  - *Direct-coded scanners*: eliminam as tabelas de *lookup*.
  - *Hand-coded scanners*: código implementado na mão.
- Assuntos interessantes e relevantes para um projeto de compilador real, mas que não são essenciais para o curso.
- Se quiser estudar esses aspectos, veja o livro do Cooper.

# Aula 01 – Análise Léxica (*Scanning*)

Prof. Eduardo Zambon

Departamento de Informática (DI)
Centro Tecnológico (CT)
Universidade Federal do Espírito Santo (Ufes)

**Compiladores**
***Compiler Construction* (CC)**