CTC-41

Compiladores

Carlos Alberto Alonso Sanches

CTC-41

2) Análise léxica

Expressões regulares e autômatos finitos

Autômatos finitos: equivalências e minimizações

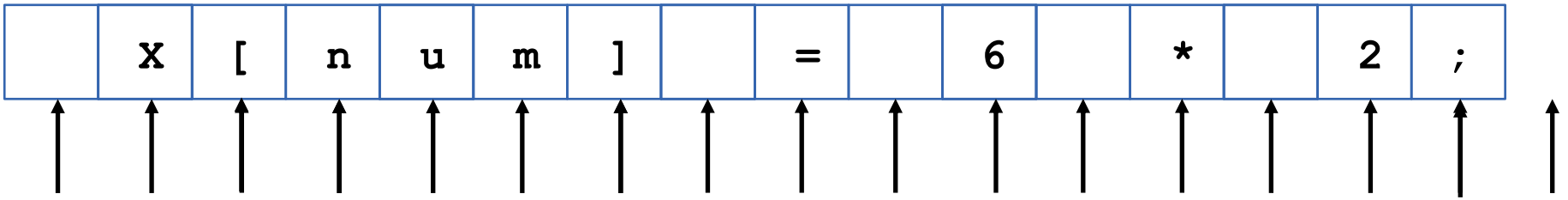
A ferramenta *Lex*

Análise léxica

- A primeira tarefa do compilador é a análise léxica, que consiste em obter os *tokens* do arquivo fonte.
  - *Tokens* (ou marcas) são sequências de caracteres que representam uma unidade válida do arquivo fonte. Exemplos: palavras reservadas da linguagem, identificadores, símbolos especiais, números, *strings*, etc.
- A partir dos caracteres do arquivo de entrada, o analisador léxico (*scanner*) realiza a extração e a classificação dos *tokens*, enviando-os ao analisador sintático (*parser*).
  - Portanto, cada *token* possui uma classificação (PR, ID, SE, NUM, etc.) e uma correspondente cadeia de caracteres (lexema).
- Outras tarefas realizadas pelo analisador léxico: consumir comentários e caracteres de separação, executar diretivas de pré-processamento (inclusão de arquivos, macros, etc.), diagnosticar erros léxicos e suas correspondentes linhas de entrada, etc.
- Para diminuir o tempo gasto com leituras do arquivo fonte, geralmente é utilizado o artifício de "*bufferização*": vários blocos são lidos simultaneamente e armazenadas numa única *string*.

Exemplo de análise léxica

```
X[num] = 6 * 4;
```



Token em análise
Lexema: "X"

Token ID é identificado
Lexema: "X"
ID e "X" enviados ao *parser*

Token em análise
Lexema: "["

Token SE é identificado
Lexema: "["
SE e "[" enviados ao *parser*

Token em análise
Lexema: "n"

Token em análise
Lexema: "nu"

Token em análise
Lexema: "num"

Token ID é identificado
Lexema: "num"
ID e "num" enviados ao *parser*

Token em análise
Lexema: "]"

Token SE é identificado
Lexema: "]"
SE e "]" enviados ao *parser*

Token em análise
Lexema: "="

Token SE é identificado
Lexema: "="
SE e "=" enviados ao *parser*

Token em análise
Lexema: "6"

Token NUM é identificado
Lexema: "6"
NUM e "6" enviados ao *parser*

Token em análise
Lexema: "*"

Token SE é identificado
Lexema: "*"
SE e "*" enviados ao *parser*

Token em análise
Lexema: "2"

Token NUM é identificado
Lexema: "2"
NUM e "2" enviados ao *parser*

Token em análise
Lexema: ";"

Token SE é identificado
Lexema: ";"
SE e ";" enviados ao *parser*

Linguagens regulares

- As linguagens regulares (tipo 3) correspondem à classe mais simples de linguagens formais, suficiente para definir os *tokens*.
- Seu reconhecimento também é bastante simples, através de um autômato de estados finitos.
- No entanto, têm fortes limitações: por exemplo, não são capazes de verificar balanceamento de parênteses.
- A geração de palavras de uma linguagem regular é feita através de um formalismo denotacional, conhecido como *expressões regulares* (ER).
- Chamamos de L(r) a linguagem gerada pela ER r, ou seja, o conjunto de cadeias de caracteres definidas por r.
- A seguir, veremos as possíveis expressões regulares.

Expressões regulares: primitivas

- Sejam: o alfabeto Σ; os caracteres a,b,c,d $\in \Sigma$; o caractere vazio $\varepsilon \in \Sigma$ que denota cadeia vazia; o caractere $\phi \in \Sigma$ que denota a linguagem vazia; as ER r e s.
- Símbolo: r = a é uma ER, onde L(r) = {a}.
  - Caso particular: $L(\phi) = \{\ \}$
- Alternativa: r|s é uma ER, onde L(r|s) = L(r) U L(s).
- Concatenação: rs é uma ER, onde L(rs) = L(r).L(s)
  - Exemplo: L((a|b)(c|d)) = {ac, ad, bc, bd}
  - Exemplo: L(a|ε) = L(ε|a) = L(a) = {a}
- Repetição: r* é uma ER, onde L(r*) = L(ε) U L(r) U L(rr) U ...
  - Exemplo: L((a|bb)*) = {ε, a, bb, abb, bba, aa, bbbb, ...}
- Precedência: repetição > concatenação > alternativa
  - Os parênteses são usados apenas para alterar essa precedência.

Expressões regulares: extensões

- Repetição não vazia: L(r+) = L(r) U L(rr) U L(rrr) U ...
- Coringa: ponto (.) indica qualquer caractere do alfabeto
  - Exemplo: .*b.* corresponde às cadeias com ao menos um caractere b
- Intervalo: uso de colchetes e hífen
  - Exemplos: [0-9] corresponde a qualquer dígito; [abc] é equivalente a a|b|c
- Escape: barra invertida (\)
  - Exemplo: L([0-9]\.[0-9]) = {0.0, 0.1, 0.2, ..., 1.0, 1.1, 1.2, ...}
- Exclusão: til (~) ou circunflexo (^)
  - Exemplo: [^a] ou ~a corresponde a qualquer caractere diferente de a
- Opcional: interrogação (?)
  - Exemplo: (\+|\-)?[0-9]+ representa os números naturais com ou sem sinal

Expressões regulares na compilação

- As ER são geralmente utilizadas na definição dos *tokens* das linguagens de programação.
- Números: sequências de dígitos, com ou sem casa decimal, com ou sem expoente
  - Natural = [0-9]+
  - NaturalSinal = (\+|\-)? Natural
  - Número = NaturalSinal (\. Natural)? (E NaturalSinal)?
- Palavras reservadas: cadeias pré-determinadas de caracteres
  - Reservadas = if | while | do | ...
- Identificadores: cadeias de caracteres não reservadas que começam com uma letra e depois contém apenas letras ou dígitos
  - Letra = [a-zA-Z]
  - Dígito = [0-9]
  - Identificador = Letra (Letra | Dígito)*
- Comentários: devem ser descartados
  - Na linguagem Pascal (entre chaves): {(~})*}
  - Na linguagem C (entre /* e */): geralmente tratados de forma específica

Expressões regulares na compilação

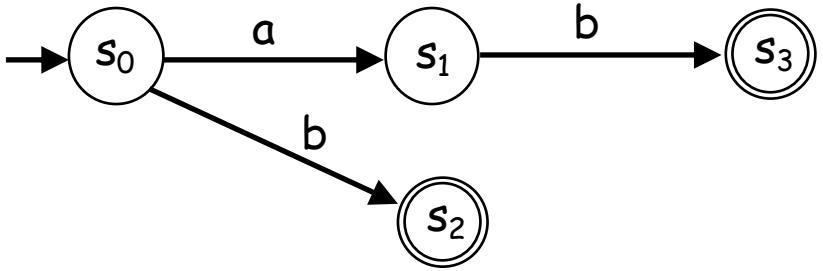
- Eventualmente podem ocorrer *ambiguidades*, isto é, uma cadeia de caracteres pode ser reconhecida por mais de uma ER.
- Isso pode ser resolvido de dois modos:
  - Priorização de uma determinada expressão regular.
    - Exemplo: reconhecimento das palavras reservadas, uma vez que elas também satisfazem a definição de identificadores.
  - Princípio da cadeia mais longa.
    - Exemplo: >= e >, == e =, <= e <, etc.
- Possíveis delimitadores dos lexemas: espaço em branco, tabulação, mudança de linha, final do reconhecimento de uma expressão regular, etc.
- Às vezes, será preciso voltar um ou mais caracteres já lidos.
  - Exemplo: `x=10;`
  - O identificador `x` somente será reconhecido após a leitura do =, que também precisa ser analisado.

Autômatos finitos determinísticos

- Um autômato finito determinístico (AFD) é uma máquina de estados, cujas transições são definidas a partir de ocorrências de símbolos de um determinado alfabeto.
- Pode-se demonstrar que um AFD é um dispositivo reconhecedor de ER.
- Geralmente, AFD são utilizados no reconhecimento dos *tokens* de um analisador léxico.
- O núcleo de um analisador léxico corresponde à implementação de um AFD, onde cada *token* é descrito como uma expressão regular.

Autômatos finitos determinísticos

- Um autômato finito determinístico (AFD) é definido como uma quíntupla $M = (K, \Sigma, \delta, s, F)$, onde:
  - K é o conjunto finito de estados;
  - Σ é o alfabeto, que não contém os caracteres ε e ϕ;
  - $s \in K$ é o estado inicial;
  - $F \subseteq K$ é o conjunto de estados finais ou de aceitação;
  - $\delta: K \times \Sigma \rightarrow K$ é a função de transição.
- Geralmente, os AFD são representados através de diagramas de estados.
- Exemplo:



$K = \{s_0, s_1, s_2, s_3\}$

$\Sigma = \{a, b\}$

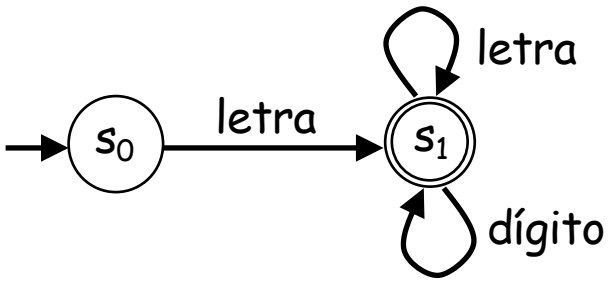
$s = s_0$

$F = \{s_2, s_3\}$

$\delta = \{((s_0,a),s_1), ((s_1,b),s_3), ((s_0,b),s_2)\}$

Identificador

- O *token* identificador pode ser reconhecido pelo AFD abaixo:



$K = \{s_0, s_1\}$

$\Sigma = \{\text{letra, dígito}\}$

$s = s_0$

$F = \{s_1\}$

$\delta = \{((s_0,\text{letra}),s_1), ((s_1,\text{letra}),s_1), ((s_1,\text{dígito}),s_1)\}$

- Geralmente, um AFD é implementado como uma matriz, cujas linhas são os estados e as colunas são os símbolos do alfabeto.
- Exemplo para o AFD de identificador:

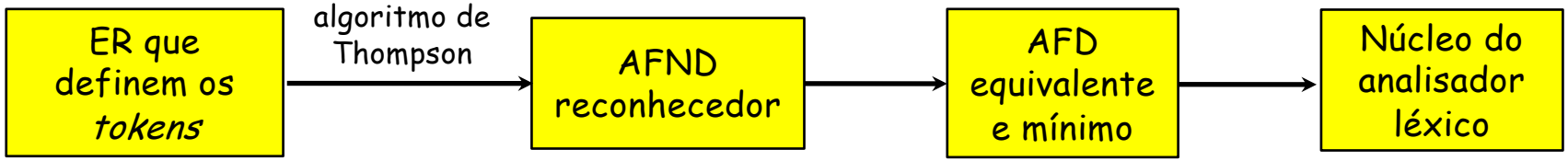
AFD	letra	dígito
s_0	s_1	-
s_1	s_1	s_1

Autômatos finitos não-determinísticos

- Um autômato finito não-determinístico (AFND) é definido como uma quíntupla $M = (K, \Sigma, \Delta, s, F)$, onde:
  - K é o conjunto finito de estados;
  - Σ é o alfabeto, que não contém os caracteres ε e ϕ;
  - $s \in K$ é o estado inicial;
  - $F \subseteq K$ é o conjunto de estados finais ou de aceitação;
  - $\Delta \subseteq K \times (\Sigma \cup \{\varepsilon\}) \times K$ é a relação de transição.
- Se M está no estado q e o símbolo de entrada é a, então M pode seguir qualquer transição $(q,a,p) \in \Delta$ ou $(q,\varepsilon,p) \in \Delta$.
- Quando uma transição (q,ε,p) é executada, nenhum símbolo da entrada é consumido.
- $(q,\varepsilon,p) \in \Delta$ é chamada de ε-transição.
- O que diferencia um AFND de um AFD são as ε-transições e as possíveis transições múltiplas a partir de um mesmo caractere.

Construção de um analisador léxico

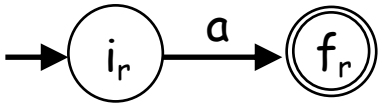
- Geralmente, o desenvolvimento de um analisador léxico é feito de acordo com o seguinte modelo:



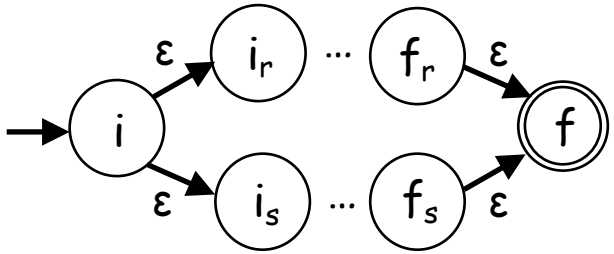
- Dada uma linguagem regular definida através de ER, o algoritmo de Thompson produz um AFND reconhecedor através da combinação de autômatos elementares que reconhecem ER primitivas.
  - É um método indutivo baseado em ε-transições.
- Em seguida, é encontrado um AFD equivalente, que depois é minimizado.
- Este processo facilita a construção do analisador léxico.

Algoritmo de Thompson

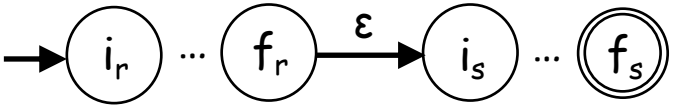
Símbolo: r = a



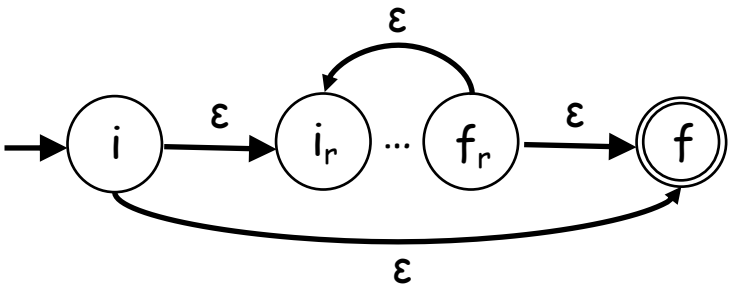
Alternativa: r | s



Concatenação: r s

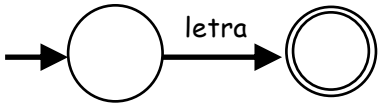


Repetição: r*

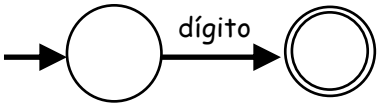


Exemplo: identificador

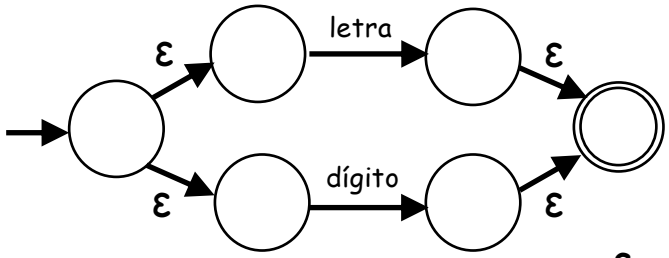
letra



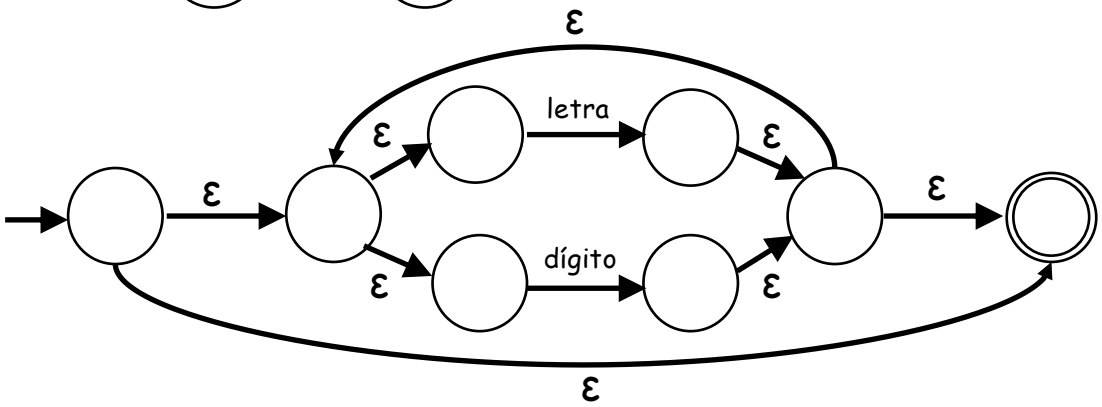
dígito



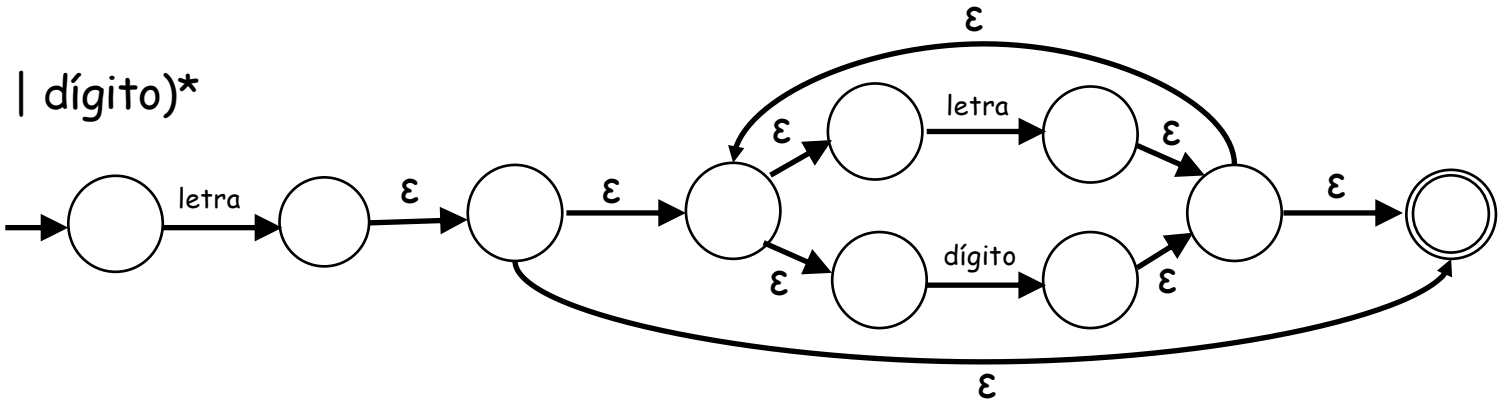
letra | dígito



(letra | dígito)*



letra (letra | dígito)*

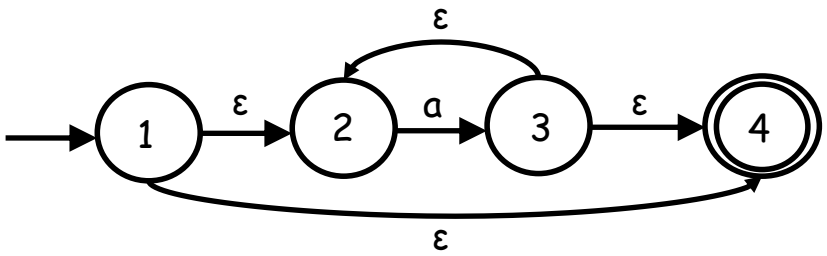


Implementações eficientes de AF

- Vimos que, graças às ε-transições, é possível gerar automaticamente um AFND reconhecedor de ER.
- Para que a sua implementação seja mais simples e eficiente, é conveniente encontrar um AFD equivalente.
- Além disso, quanto menos estados tiver esse AFD, mais rápida deverá ser a sua execução. Por isso, convém minimizá-lo antes da implementação.
- Veremos a seguir:
  - como encontrar um AFD equivalente a um AFND;
  - como minimizar o número de estados de um AFD.

AFD equivalente a um AFND

- Dado um AFND, um AFD equivalente pode ser obtido através de um processo construtivo, onde cada estado desse AFD é um subconjunto de estados do AFND original.
- Este processo também elimina eventuais transições múltiplas.
- A eliminação das ε-transições é feita através de ε-fechos:
  - O ε-fecho de um estado s é o conjunto de estados atingíveis a partir de s com zero ou mais ε-transições. Consequentemente, cada estado sempre pertence ao seu ε-fecho.
  - Denotaremos o ε-fecho de um estado s como ε_s^*.
  - De maneira similar, o ε-fecho de um conjunto de estados será a união dos ε-fechos dos estados deste conjunto.
- Exemplo:



$\varepsilon_1^* = \{1,2,4\}$

$\varepsilon_2^* = \{2\}$

$\varepsilon_3^* = \{2,3,4\}$

$\varepsilon_4^* = \{4\}$

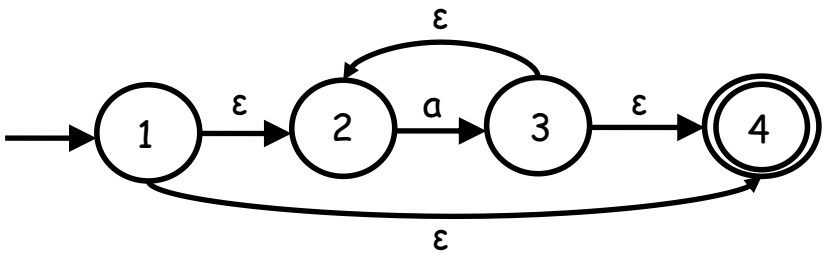
$\varepsilon_{\{1,3\}}^* = \{1,2,3,4\}$

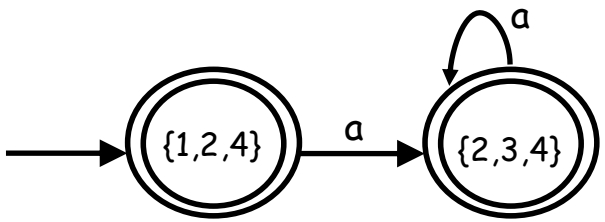
$\varepsilon_{\{2,4\}}^* = \{2,4\}$

$\varepsilon_{\{2\}}^* = \varepsilon_2^* = \{2\}$

Processo de construção

- Dado um AFND M, iremos construir um AFD M' equivalente:
  1) O estado inicial de M' será ε_s^*, onde s é o estado inicial de M.
  2) Dado um conjunto S de estados de M, que é um fecho já obtido, Sx é o conjunto de estados atingíveis com o caractere x a partir de algum estado de S.
  3) Cada fecho de Sx, chamado de ε_{Sx}^*, define um estado de M', desde que não seja vazio.
  4) Um estado de M' será final se contiver ao menos um estado final de M.
  5) M' é construído através da aplicação sucessiva dos passos (2), (3) e (4) acima.
- Considerando o exemplo anterior:



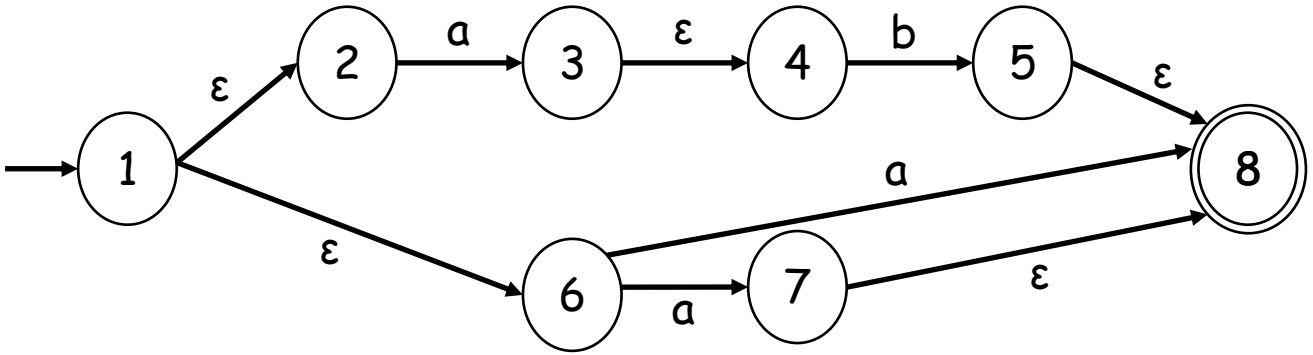


$\varepsilon_1^* = \{1,2,4\}$

$\varepsilon_{\{1,2,4\}a}^* = \varepsilon_{\{3\}}^* = \{2,3,4\}$

$\varepsilon_{\{2,3,4\}a}^* = \varepsilon_{\{3\}}^* = \{2,3,4\}$

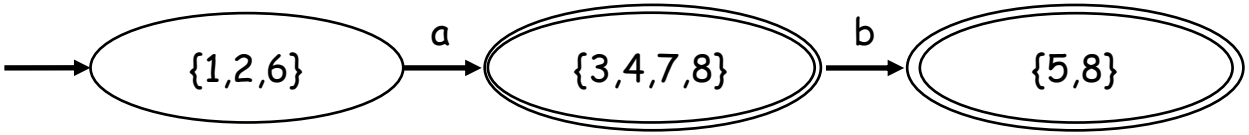
Exemplo



$\varepsilon_{\{1\}}^* = \{1,2,6\}$

$\varepsilon_{\{1,2,6\}a}^* = \varepsilon_{\{3,7,8\}}^* = \{3,4,7,8\}$

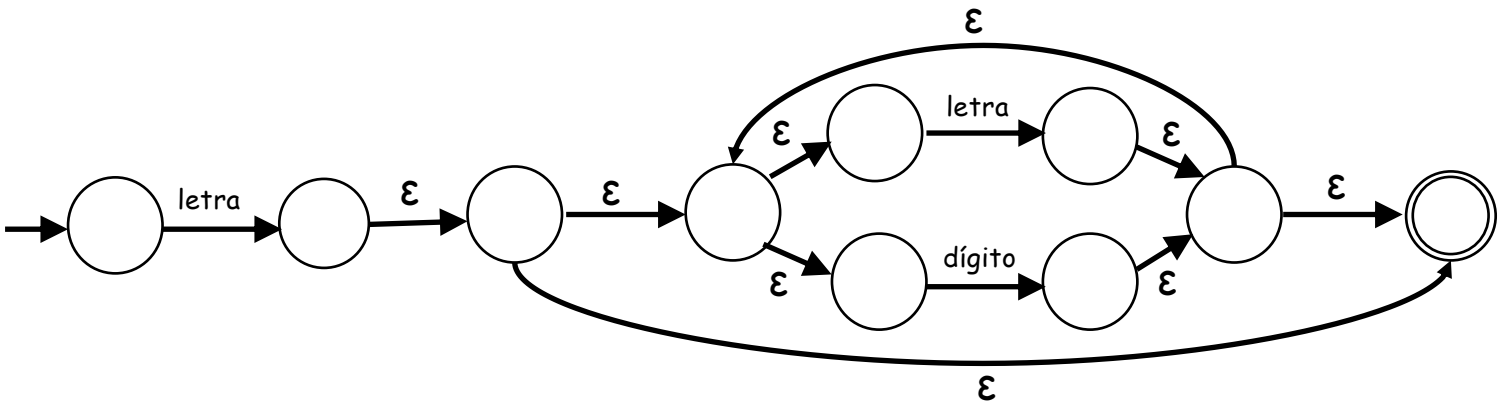
$\varepsilon_{\{3,4,7,8\}b}^* = \varepsilon_{\{5\}}^* = \{5,8\}$



Exercício 1

- Encontre um AFD equivalente para o AFND abaixo, que reconhece identificadores:

letra (letra | dígito)*



Minimização de estados de um AFD

- Um resultado teórico importante é que cada linguagem regular é reconhecida por um AFD mínimo, que é único.
- Este AFD mínimo pode ser obtido através de um processo de redução do número de estados, sem afetar a linguagem reconhecida:

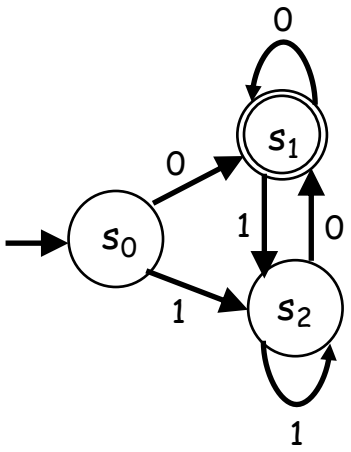
1) Separe os estados do AFD em dois grupos: um grupo G_1 com todos os estados de aceitação, e outro grupo G_2 com os demais estados.

2) Avalie as transições dos estados de cada grupo. Se houver estados com transições idênticas, eles são redundantes e devem ser unificados.

3) Repita o passo (2) até que não existam estados redundantes em ambos os grupos.

Exemplo

- Minimização do AFD abaixo, que reconhece a expressão (0|1)*0 :



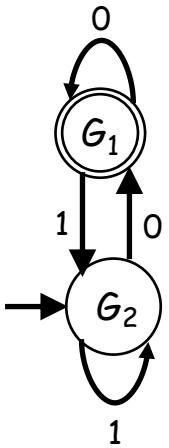
$G_1 = \{s_1\}$ $G_2 = \{s_0, s_2\}$

Em G_1, não pode haver redução.

Grupo G_2:

	0	1
s_0	s_1	s_2
s_2	s_1	s_2

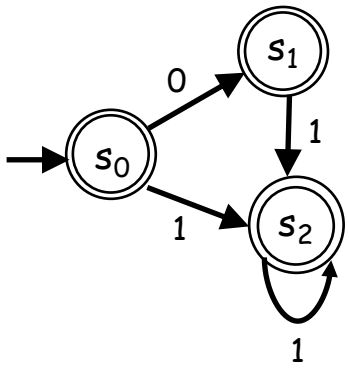
Portanto, s_0 e s_2 são redundantes.



AFD mínimo equivalente

Outro exemplo

- Minimização do AFD abaixo, que reconhece a expressão (0|ε)1*:



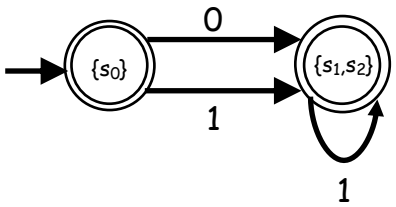
$G_1 = \{s_0, s_1, s_2\}$ $G_2 = \{\}$

Em G_2, não pode haver redução.

Grupo G_1:

	0	1
s_0	s_1	s_2
s_1	-	s_2
s_2	-	s_2

Portanto, s_1 e s_2 são redundantes.

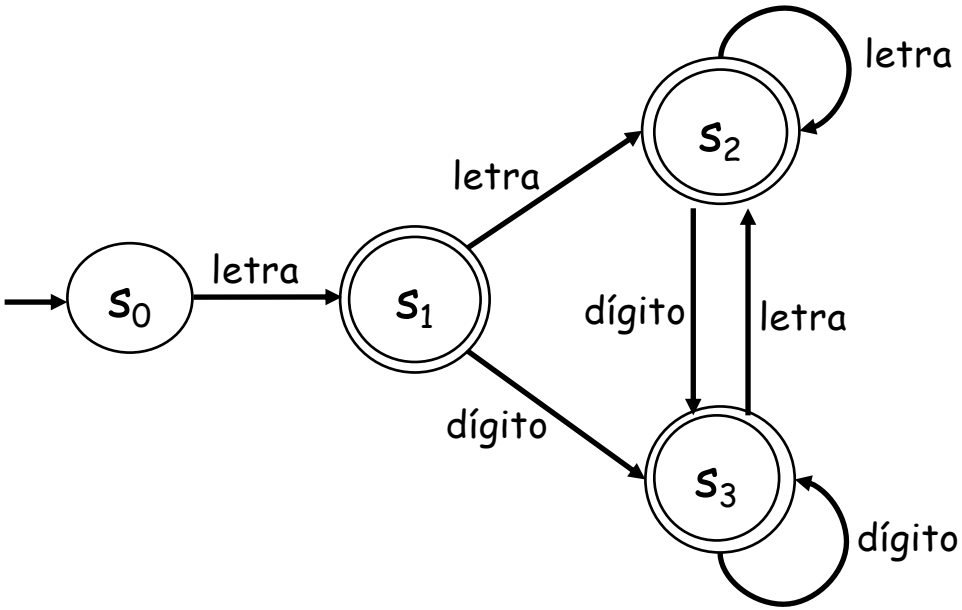


AFD mínimo equivalente

Exercício 2

- Minimize o AFD abaixo, que reconhece identificadores:

letra (letra | dígito)*



A ferramenta *Lex*

- *Lex* é um conhecido gerador de *scanners*, que possui diversas versões abertas e gratuitas.
- *Flex* (*Fast Lex*) é uma das suas versões mais populares, distribuída pelo *GNU Project* (www.gnu.org), da *Free Software Foundation*.
- Esta ferramenta recebe como entrada um arquivo texto com expressões regulares e ações associadas, e produz código C para um AFD baseado em tabelas, que faz o reconhecimento dessas expressões.
- O arquivo de entrada tem extensão `.l`, e o arquivo de saída chama-se **`lex.yy.c`** .

Formato das ER no *Flex*

Padrão	Significado
`a`	Caractere `a`
`"a"`	Caractere `a`, mesmo se `a` for um metacaractere
`\a`	Caractere `a` se `a` for um metacaractere
`a*`	Zero ou mais repetições de `a`
`a+`	Uma ou mais repetições de `a`
`a?`	Um `a` opcional
`a|b`	`a` ou `b`
`(a)`	`a` propriamente dito
`[abc]`	Qualquer caractere entre `a`, `b` e `c`
`[a-d]`	Qualquer caractere entre `a`, `b`, `c` e `d`
`[^ab]`	Qualquer caractere, exceto `a` ou `b`
`.`	Qualquer caractere, exceto mudança de linha
`{xxx}`	A expressão regular representada pelo nome `xxx`

Exemplos

- `if` ou `"if"`: uma palavra reservada
- `"("` ou `\(`: parênteses à esquerda, pois é um metacaractere
- `\n` e `\t`: mudança de linha e tabulação (análogo à linguagem C)
- Expressões equivalentes:
  - `(aa|bb)(a|b)*c?`
  - `("aa"|"bb")("a"|"b")*"c"?`
  - `(aa|bb)[ab]*c?`
- Número natural com e sem sinal:
  - `nat [0-9]+`
  - `signedNat ("+"|"-")? nat`
- `[."?]`: qualquer um desses 3 caracteres, que perdem seu significado de metacaracteres dentro dos colchetes
- `[^0-9aeiou]`: qualquer caractere que não seja dígito ou vogal

Alguns nomes internos utilizados

Nome	Significado
`lex.yy.c`	Nome do arquivo de saída gerado
`yylex`	Procedimento para varredura (equivalente a `getToken`)
`yytext`	Cadeia de caracteres que casou com ação corrente (lexema)
`yyin`	Entrada (padrão: `stdin`)
`yyout`	Saída (padrão: `stdout`)
`input`	Procedimento de entrada
`ECHO`	Ação básica de saída (padrão: imprime `yytext` em `yyout`)

- A documentação completa do *Flex* está disponível em https://westes.github.io/flex/manual/

Formato do arquivo de entrada

- Um arquivo de entrada do *Flex* tem o seguinte formato:

```
{ definições }
%%
{ regras }
%%
{ rotinas auxiliares }
```

- Na seção de definições, estão:
  - código em C, indicado entre os delimitadores `%{` e `%}`, que será inserido no arquivo gerado, fora de todos os procedimentos;
  - nomes de expressões regulares.
- Na seção de regras, há as expressões regulares seguidas pelos códigos em C a serem executados em caso de reconhecimento.
- Na terceira seção, que é opcional, podem estar os códigos em C de rotinas auxiliares e do programa principal. Quando essa seção não existir, o segundo `%%` pode ser omitido.

Exemplo

```
/* Contagem de caracteres e de linhas da entrada padrão */
/* No final da entrada, deverá estar escrito "fim" */

%{
#include <stdio.h>
int nlines = 0, nchars = 0;   /* variáveis globais */
%}

%%

\n    ++nlines; ++nchars;
.     ++nchars;
fim   return 0;

%%

int main()                          /* código inserido no final */
{
  yylex();
  printf("Linhas = %d, caracteres = %d\n", nlines, nchars);
}
```

Tratamento de ambiguidades

- Se houver ambiguidades nas regras, *Flex* procurará resolvê-las do seguinte modo:
  - Em cada regra, será reconhecida o lexema mais longo possível.
  - Se duas ou mais regras reconhecem lexemas de mesmo tamanho, Flex selecionará a regra que foi descrita antes.
  - Se nenhuma regra reconhecer um determinado caractere, ele será copiado na saída padrão, e a análise léxica continuará.

Exemplo

```
/* Reconhecimento de números inteiros nos formatos: */
/* decimal, octal, hexadecimal, binário             */
/* Obs: será preciso linkar com biblioteca libfl    */

DIGIT [0-9]

%%
[1-9]{DIGIT}*     printf("DEC");
0[0-7]*           printf("OCT");
0x[0-9A-Fa-f]+    printf("HEX");
0b[01]+           printf("BIN");
<<EOF>>           return 0;

%%
int main(int argc, char *argv[])
{
  FILE *f_in;
  if (argc == 2)     /* entrada: arquivo */
  {
    if (f_in = fopen(argv[1],"r")) yyin = f_in;
    else perror(argv[0]);
  }
  else yyin = stdin; /* entrada: teclado */
  yylex();
  return 0;
}
```

Possível entrada:

```
12345
02343
0xFA
0b11
xxx
```

Saída:

```
DEC
OCT
HEX
BIN
xxx
```