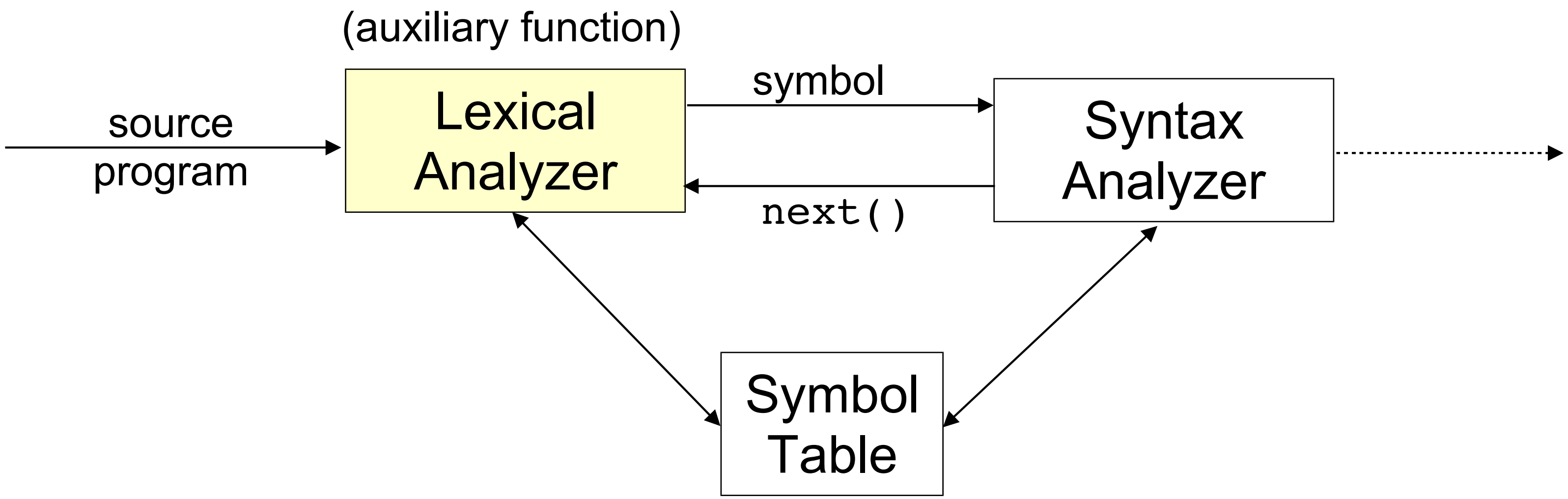
Lexical Analysis

- Primary role = abstraction process: $[characters] \rightarrow [symbols]$



- Secondary role
  - removal of spacing } pseudo-symbols
  - removal of comments } pseudo-symbols
  - linking of errors to source lines (supports clarity of error messages)

- Advantages in separating syntax analysis from lexical analysis:

  1. Simplicity of design (syntax do not care about white spaces, comments, ...)
  2. Efficiency
  3. Portability

Symbol Specification

- **Lexical string**: significant sequence of characters (*lexeme*)
- **Symbol**: abstraction of a class of lexical strings
- **Pattern**: rule to describe the extension of a symbol (≈ grammar of lexical symbol)
- **Lexical attribute**: when the symbol is the abstraction of several lexical strings

alpha = (beta * 3) ⟹ ⟨**id**, ↑alpha⟩ ⟨**assign**, ⟩ ⟨**left**, ⟩ ⟨**id**, ↑beta⟩ ⟨**times**, ⟩ ⟨**num**, 3⟩ ⟨**right**, ⟩

Regular Expressions

- Definition of regular expression (on alphabet Σ):
  1. ε is a regular expression denoting the language $\{ \varepsilon \}$
  2. If $a \in \Sigma$, then a is a regular expression denoting the language $\{ a \}$
  3. If x, y are regular expressions denoting languages $L(x)$, $L(y)$, then:
     - (x) is a regular expression denoting $L(x)$
     - (x) **|** (y) is a regular expression denoting $L(x) \cup L(y)$
     - (x) (y) is a regular expression denoting $\{ xy \mid x \in L(x), y \in L(y) \}$
     - $(x)^*$ is a regular expression (repetition zero or more times of strings in L(x))

- Extended regular expressions: $(x)^+$, • , [a-z], ~(a|b), (x)?

- Regular definitions: associations of names with regular expressions

  letter $\rightarrow$ [A-Za-z]

  digit $\rightarrow$ [0-9]

  id $\rightarrow$ **letter** (**letter** | **digit**)*

Regular Expressions for Tokens in Prog. Languages

- Classification of lexical elements:
  - **Keywords** = { `if`, `then`, `while`, `do`, `begin`, `end`, `procedure`, `function`, `case`, `repeat` ... }
  - **Special symbols** = { `+`, `-`, `*`, `/`, `=`, `>`, `>=`, `!=`, `++`, `--`, `&&`, `||`, ... }
  - **Identifiers** = { alphanumeric strings starting with a letter }
  - **Constants** = { `25`, `.34`, `2.7E-3`, `"alpha"`, `'a'`, ... }
  - **Pseudo-symbols** (spacing, comments)

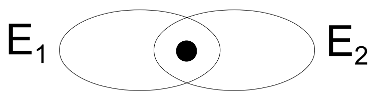
- Ambiguity: when the same string of characters is compatible with <u>several</u> regular expressions

<u>Examples</u>:

- **`while`** → **id** / **while**
- **<>** → **noteq** / **less greater**

PL → disambiguating rules:
- keywords = reserved words
- principle of longest substring (*maximal munch*)

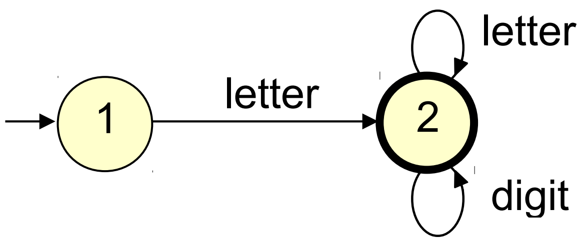
In general: insufficient



Finite Automata

- Mathematical formalism to describe certain types of algorithms (“machines”)
- In particular: to check the membership of a string in a regular language
  (matching string – regexpr)
- Strong correlation between
  - finite automata
  - regular expressions

id → letter (letter | digit)* ⇨ recognition process of an **id** by means of a diagram



- **States**: what was recognized
- **Transitions**: change of state caused by the matching of a character

- Recognition process of an **id** → defined by the sequence of involved transitions

$\rightarrow 1 \xrightarrow{a} 2 \xrightarrow{l} 2 \xrightarrow{p} 2 \xrightarrow{h} 2 \xrightarrow{a} 2$ ⇨ ∃ a path generating the string

- Recognized regular language = { paths within automaton }

Deterministic Finite Automata

- **Def**: A DFA is a 5-tuple $M = (\Sigma, S, T, s_0, F)$, where:
  - Σ = alphabet
  - S = set of states
  - $T: S \times \Sigma \to S$ = transition function (deterministic)
  - s_0 = initial state (unique)
  - $F \subseteq S$ = set of final states (nonempty set)

- L recognized by $M \equiv L(M) = \{ x \mid x = c_1 c_2 \ldots c_n,\ c_i \in \Sigma$, and $\exists$

$$s_1 = T(s_0, c_1),$$
$$s_2 = T(s_1, c_2),$$
$$\ldots$$
$$s_n = T(s_{n-1}, c_n),\ s_n \in F \}$$

$$\to s_0 \xrightarrow{c_1} s_1 \xrightarrow{c_2} s_2 \to \ldots \to s_{n-1} \xrightarrow{c_n} s_n$$

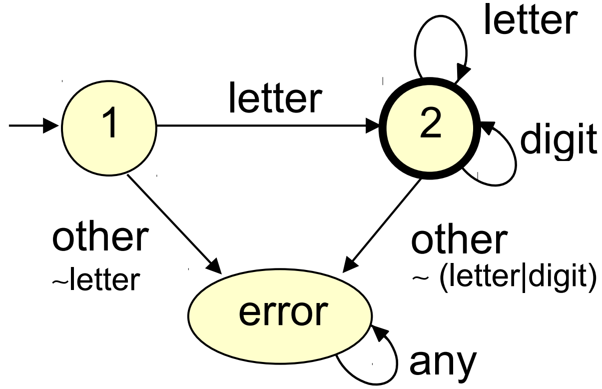
Deterministic Finite Automata (ii)

Notes comparing formal definition of DFA and diagrammatic example (**id**)

1. States identifiable by any type of labels (numbers, letters, strings, ...)
2. Factorization (abstraction) of *n* transitions, one ∀ letter/digit, by a single transition **letter/digit**
3. Def ⇒ T: S × Σ → S = function ⇒ T(s, c) must have a value ∀ (s, c) ⇒ missing transitions!

⇩

missing transitions ≡ **error transitions**!



possible input characters

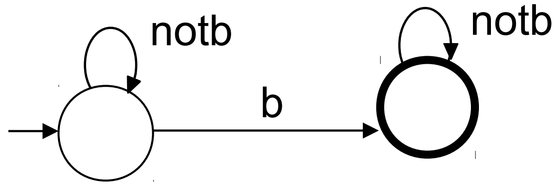
other ≡ c ∈ (**Σ** - { characters involved in the other transitions exiting the same state })

context-sensitive meaning

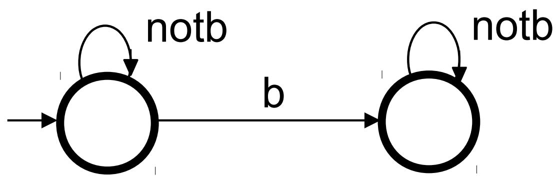
- Error state: double semantics
  - it is not an identifier (from non-final state)
  - identifier followed by separator (from final state) → maximal munch

Examples of DFAs in Diagrammatic Form

1. L = { *x* | *x* includes <u>exactly</u> one *b* }



2. L = { *x* | *x* includes <u>at most</u> one *b* }



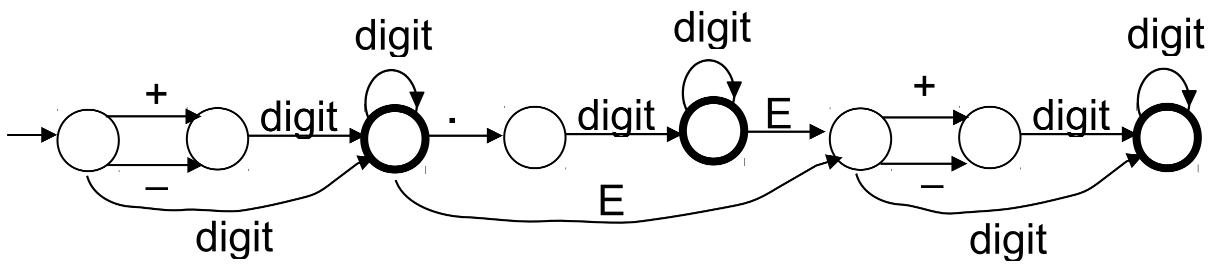
3. L = { *x* | *x* = numeric constant in scientific notation }

digit → [0-9]

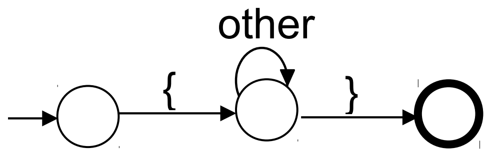
nat → **digit**$^{+}$

snat → (+|–) ? **nat**

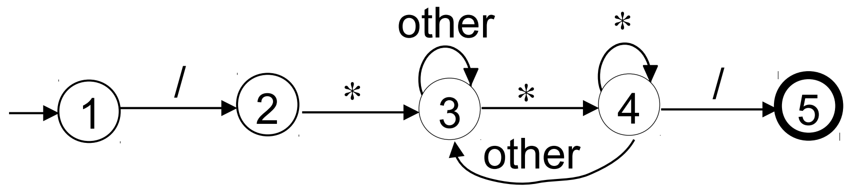
num → **snat** (“.”**nat**) ? (E **snat**) ?



4. L = { *x* | *x* = Pascal comment }



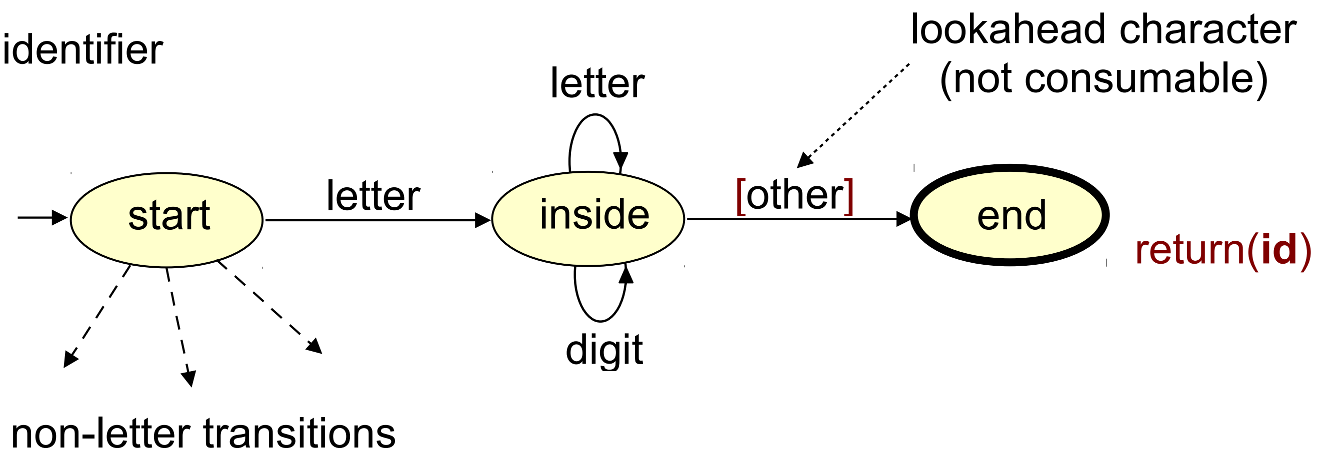
5. L = { *x* | *x* = C comment }



Augmentation of Semantics in DFAs

- Representation ⟨ diagrammatic / formal ⟩ does not describe every algorithmic aspect of the recognition process
  - Error → backtracking or return(ERROR) ?
  - Final state → ?
  - Matching of character → ? (accumulation in lexeme)
  - Maximal munch ?

- Example: Pascal identifier

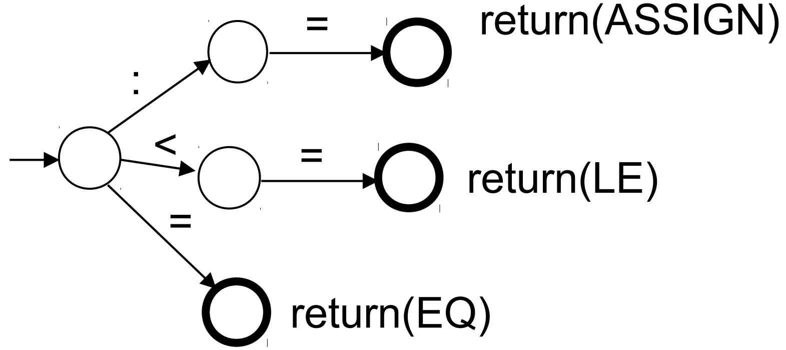


Furthermore: it expresses maximal munch

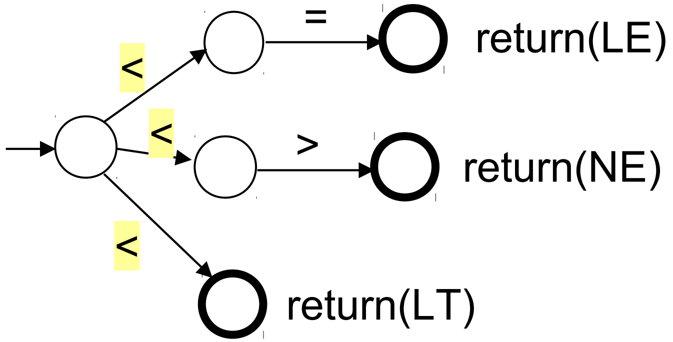
Problem of Initial State

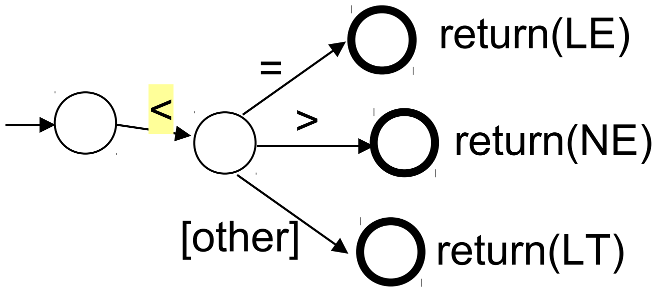
- Since ∃ different tokens in a PL → combination (fusion) of different automata

1. “Predictive” case: :=, <=, = ⟹
   (no ambiguity)



2. Ambiguous case: <=, <>, < ⟹ (we should)

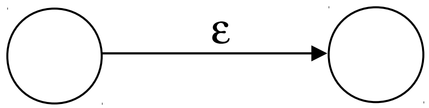




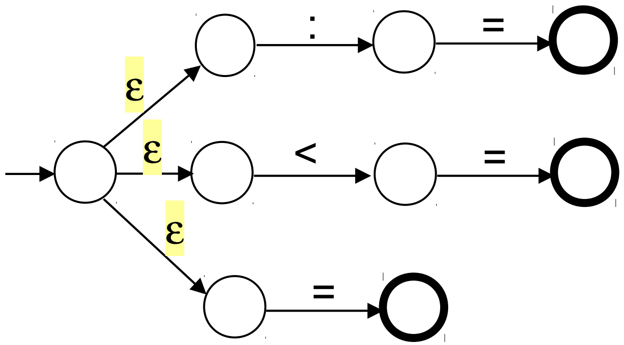
- <u>In theory</u>: manual combination of all tokens in one large DFA ⇒ complex!
  <u>Better</u>: extending DFA to NFA + algorithm: NFA → equivalent DFA (modular approach)

Empty Transition

- ε-transition
  - formally: matching of empty string (ε)
  - intuitively: performed “spontaneously” (without character consumption)



- Advantage: simple and systematic combination of symbol recognizers ("glue" for NFAs)



Nondeterministic Finite Automata

- **Def**: An NFA is a 5-tuple $M = (\Sigma, S, T, s_0, F)$, where:
  - Σ = alphabet
  - S = set of states
  - **T**: $S \times (\Sigma \cup \{\varepsilon\}) \rightarrow 2^S$ = transition function $\rightarrow$ 2 causes for nondeterminism
    - ε-transitions
    - same c for several trans.
  - s_0 = initial state
  - $F \subseteq S$ = set of final states

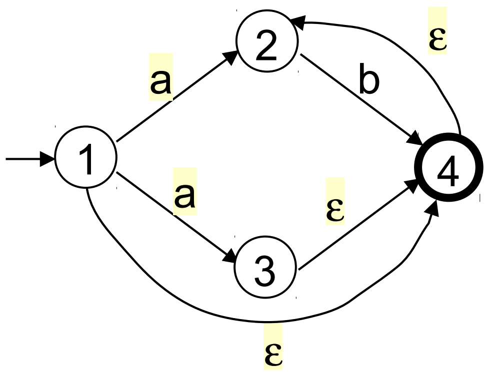
- L recognized by $M \equiv L(M) = \{ x \mid x = c_1c_2...c_n,\ c_i \in (\Sigma \cup \{\varepsilon\})$, and $\exists$

$$s_1 \in T(s_0, c_1),$$
$$s_2 \in T(s_1, c_2),$$
$$\ldots$$
$$s_n \in T(s_{n-1}, c_n),\ s_n \in F \}$$

- Notes:

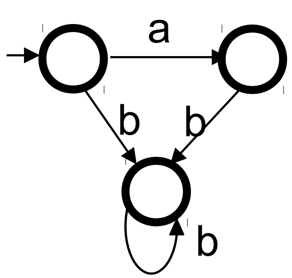
1. $x = c_1c_2...c_n$, actually $|x| \leq n$
2. Nondeterminism when choosing $s_i \in T(s_{i-1}, c_i)$
3. Formally, NFA <u>does not</u> represent an algorithm, yet can be simulated by an algorithm with backtracking ($\rightarrow$ efficiency problem)

Examples of NFAs in Diagrammatic Form

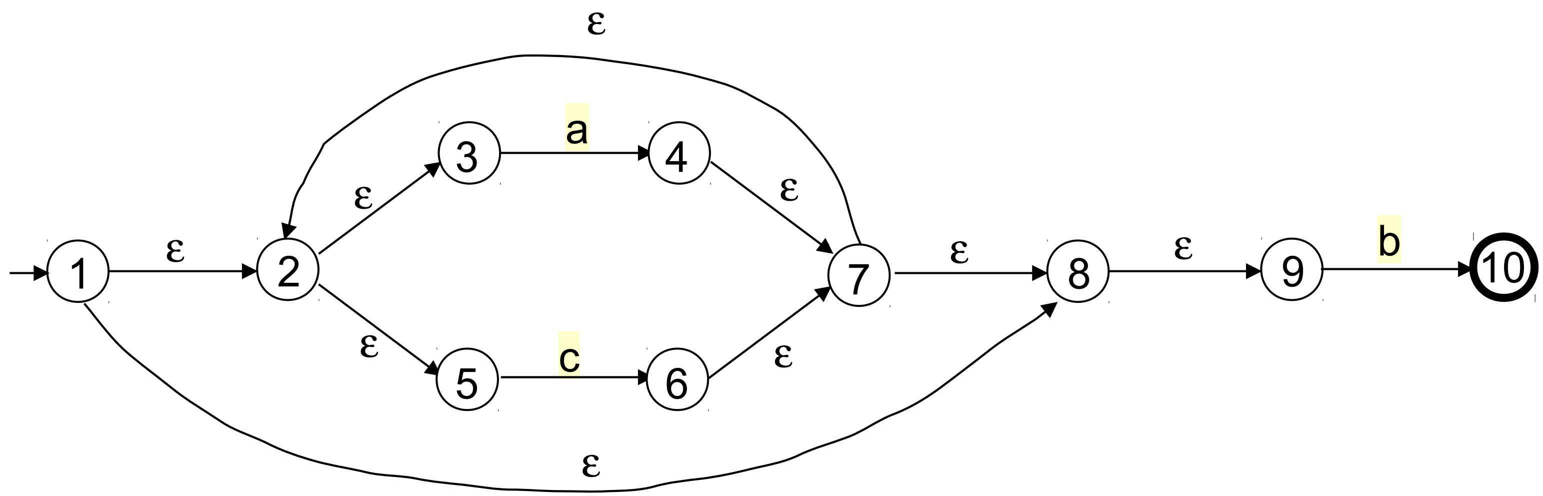


- Recognition of abb
  - $1 \xrightarrow{a} 2 \xrightarrow{b} 4 \xrightarrow{\varepsilon} 2 \xrightarrow{b} 4$
  - $1 \xrightarrow{a} 3 \xrightarrow{\varepsilon} 4 \xrightarrow{\varepsilon} 2 \xrightarrow{b} 4 \xrightarrow{\varepsilon} 2 \xrightarrow{b} 4$

  several paths for the same string
- Recognition of L((a | ε)b*) $\Rightarrow$ in particular: ε, a, b
- Equivalent DFA:

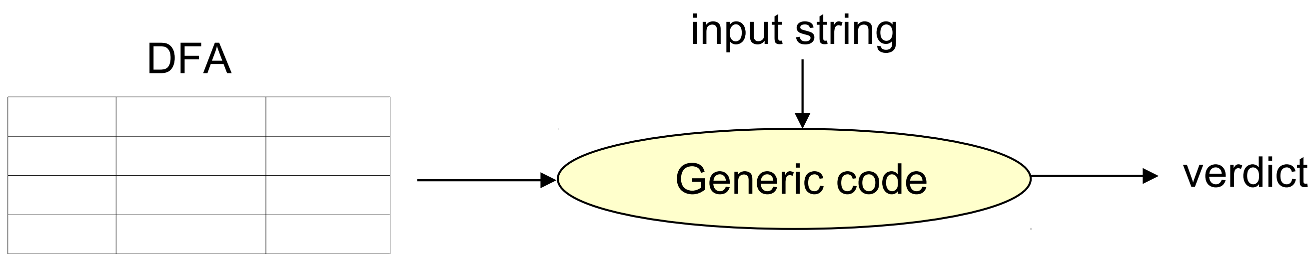


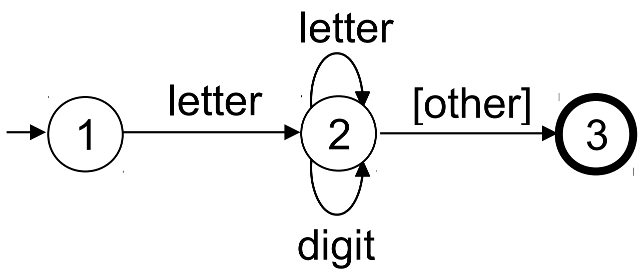
Examples of NFAs in Diagrammatic Form (ii)



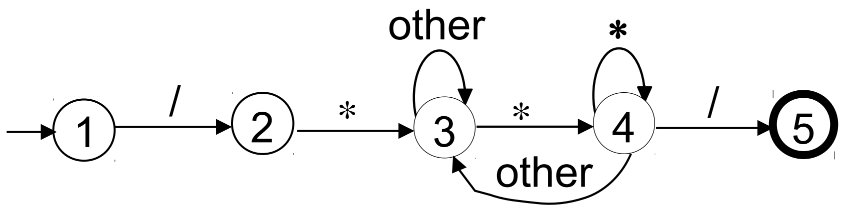
acab $\in$ (a | c)*b

Automata Specified by Transition Tables





	c			
	letter	*digit*	*other*	*F*
1	2			
2	2	2	[3]	
3				•



	c			
	/	*	*other*	*F*
1	2			
2		3		
3		4	3	
4	5	4	3	
5				•

- Assumptions:
  - Blank boxes → unspecified transitions
  - s_0 = first in list
  - Information extension
    - states ∈ F
    - consumption of *c*

Automata Specified by Transition Tables (ii)

- Generic code $\longrightarrow$ 3 data structures (simple, yet universal)
  - `T[state,c]` $\equiv$ transitions $\rightarrow$ integers
  - `Cons[state,c]` $\equiv$ Booleans $\rightarrow$ true: go forward (consumption)
  - `F[state]` $\equiv$ Booleans $\rightarrow$ true: final

- Table-driven algorithm:

<u>exmp</u>: state = -1

```
state := 1; c := next();
while not F[state] and not error(state) do
  begin
    new_state := T[state,c];
    if Cons[state, c] then
      c := next();
    state := new_state
  end;
if F[state] then accept.
```

Automata Specified by Transition Tables (iii)

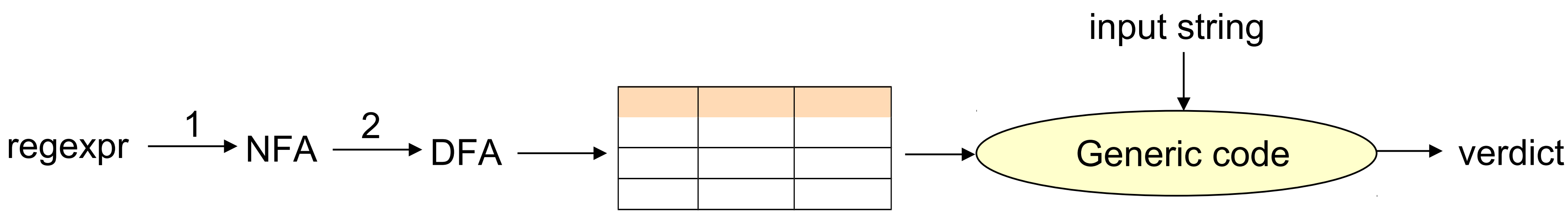
- Notes on table-driven algorithms :

  1. Pros:
     - Generic code
     - Reduced size of code
     - Easy code maintenance

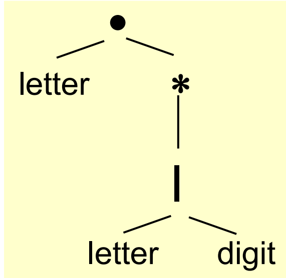
  2. Cons: waste of memory (sparse tables → compression)

  3. Technique extensible to NFA
     - `T[state,c]` = { states }
     - backtracking ⇒ inefficient: better transforming NFA → DFA

- Underlying problem: Regexpr → DFA ⇒ in two steps (automatic within generators):

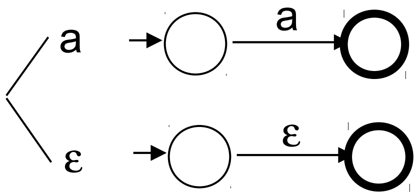


Regexpr $\rightarrow$ NFA (Construction of Thompson)

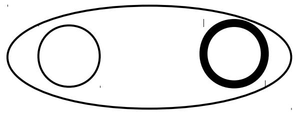


- Method:
  - Incremental construction starting from automata (a, ε)
  - ε-transitions: to glue together sub-automata
  - Final automaton= recognizer of L(r)

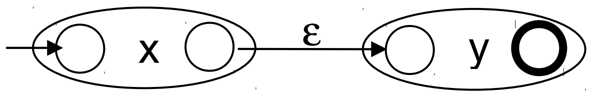
1. Atomic regexpr: x =



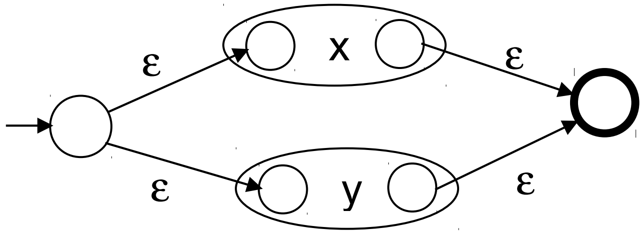
⇨ in general (standardized automata)



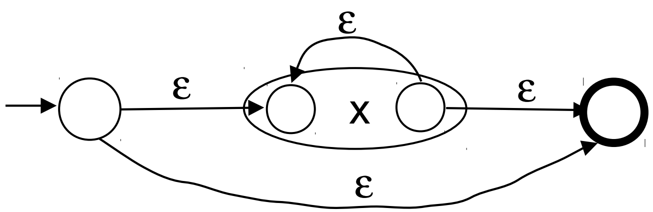
2. Concatenation: xy



3. Alternative: x | y



4. Repetition: x^*



5. Parentheses: (x) $\Rightarrow$ automaton unchanged

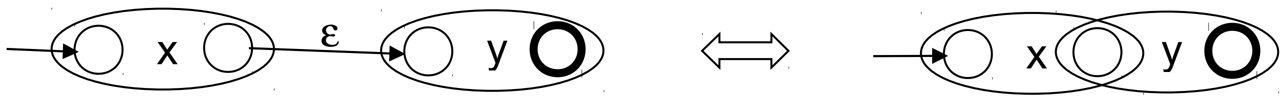
Regexpr → NFA (Construction of Thompson) (ii)

- Notes:
  - ∀ new state → new name
  - Properties of the final NFA (by construction):

1. NFA recognizes L(r)
2. Total-states(NFA) ≤ 2 * (symbols in r)
3. NFA has
   - 1 initial state (no entering transitions)
   - 1 final state (no exiting transitions)
4. ∀ state ∈ NFA: possible one of two cases
   - 1 exiting transition marked by $c \in \Sigma$
   - at most 2 exiting transitions marked by ε

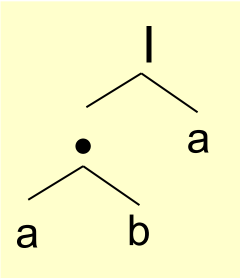
- Not unique construction (simplification):

xy:

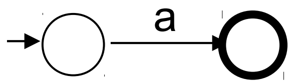


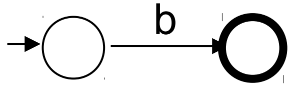
Construction of Thompson: Examples

ab | a

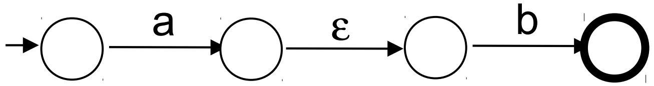


a

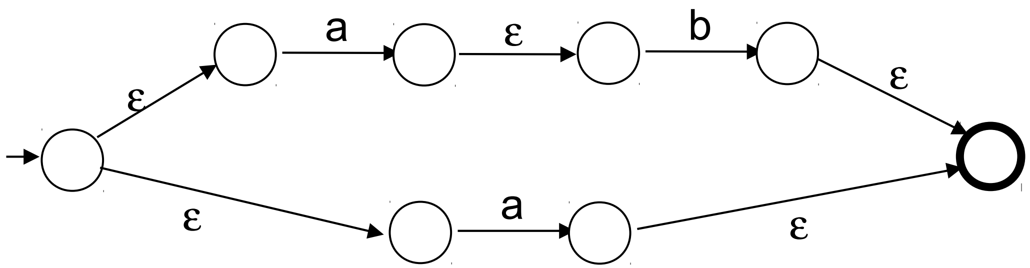




b



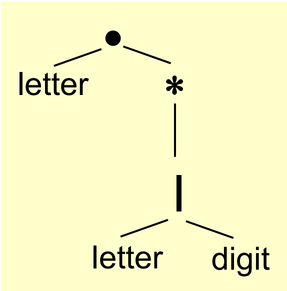
ab



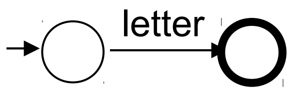
ab | a

Construction of Thompson: Examples (ii)

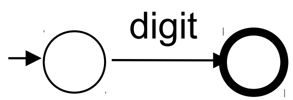
letter (letter | digit)*

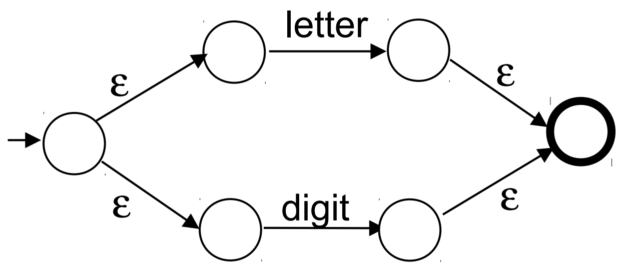


letter

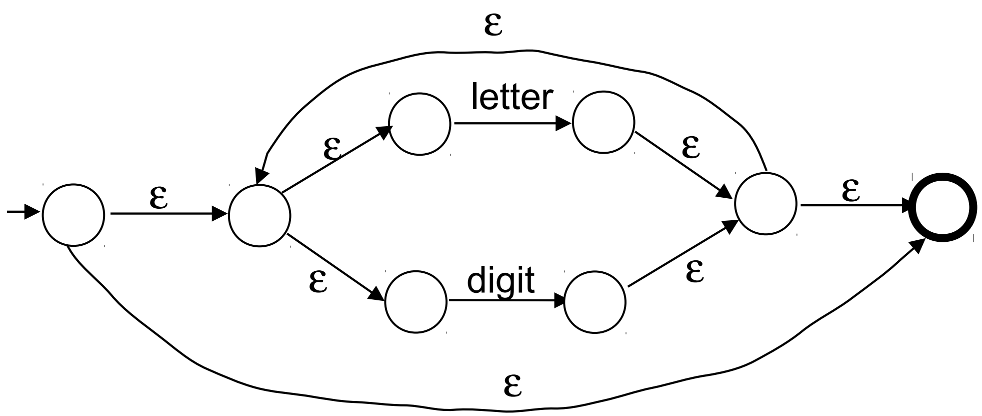


digit

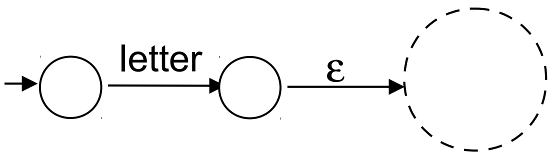




letter | digit



(letter | digit)*



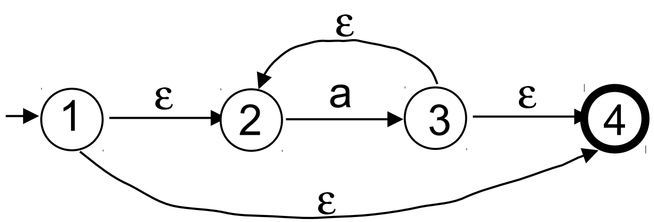
letter (letter | digit)*

NFA → DFA

- Removal of transitions (causes for nondeterminism)
  - empty (ε-transitions) $\Rightarrow$ ε-closure = { states reachable by ε-transitions }
  - multiple (on single *c*) $\Rightarrow$ { states reachable with same *c* }

- Def of **ε-closure**:

1. **Single state**: $\underline{s} \;\equiv\; \varepsilon\text{-closure}(s) \;\equiv\; \{\, s' \mid \exists\, s \xrightarrow{\varepsilon} \ldots \xrightarrow{\varepsilon} s' \text{ in } M \,\} \cup \{\, s \,\}$



= construction of Thompson for a^*

$$\begin{cases} \underline{1} = \{ 1, 2, 4 \} \\ \underline{2} = \{ 2 \} \\ \underline{3} = \{ 2, 3, 4 \} \\ \underline{4} = \{ 4 \} \end{cases}$$

2. **Set of states**: $\underline{S} \;\equiv\; \varepsilon\text{-closure}(S) \;\equiv\; \bigcup_{s \in S} \underline{s}$ $\Longrightarrow$ $\underline{\{1,3\}} = \underline{1} \cup \underline{3} = \{1,2,4\} \cup \{2,3,4\} = \{1,2,3,4\}$

Subset Construction

NFA DFA

$$M \rightarrow M' \begin{cases} M = (\Sigma, S, T, s_0, F) \\ M' = (\Sigma, S', T', s'_0, F'), \text{ where} \begin{cases} S' \subseteq 2^S \\ T': S' \times \Sigma \rightarrow S' \\ s'_0 = \underline{s_0} \\ F' \subseteq S' \end{cases} \end{cases}$$

- Computation of S', T', F':

$S' := \{ s'_0 \}$; $T' := \varnothing$;

repeat

  Choose an unmarked node $A \in S'$;

  for each $c \in \Sigma$ marking a transition of M exiting a node in A **do**

    begin

      $A'_c := \{ z \mid s \in A, s \xrightarrow{c} z \in T \}$;

      $\underline{A'}_c := \varepsilon\text{-closure}(A'_c)$;

      if $\underline{A'}_c \notin S'$ **then** $S' := S' \cup \{ \underline{A'}_c \}$;

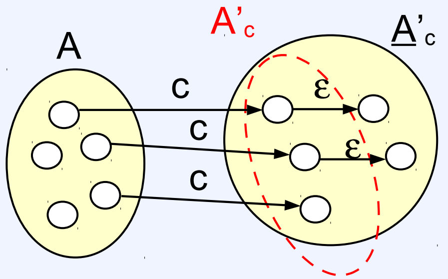
      $T' := T' \cup \{ A \xrightarrow{c} \underline{A'}_c \}$

    end;

  Mark A

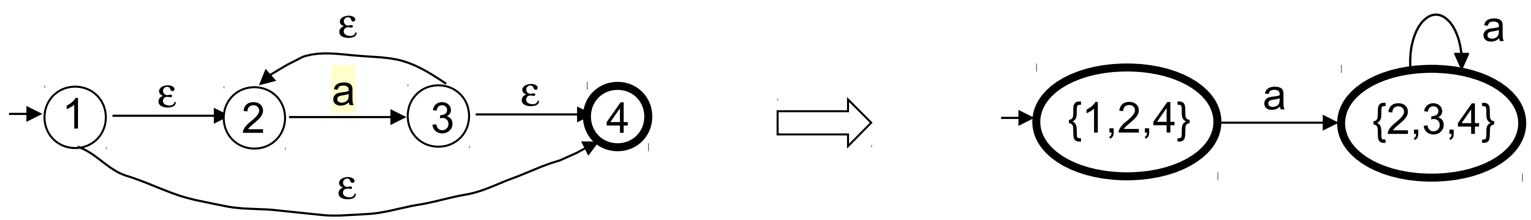
until all elements in S' are marked;

$F' := \{ A \mid A \in S', A \cap F \neq \varnothing \}$.



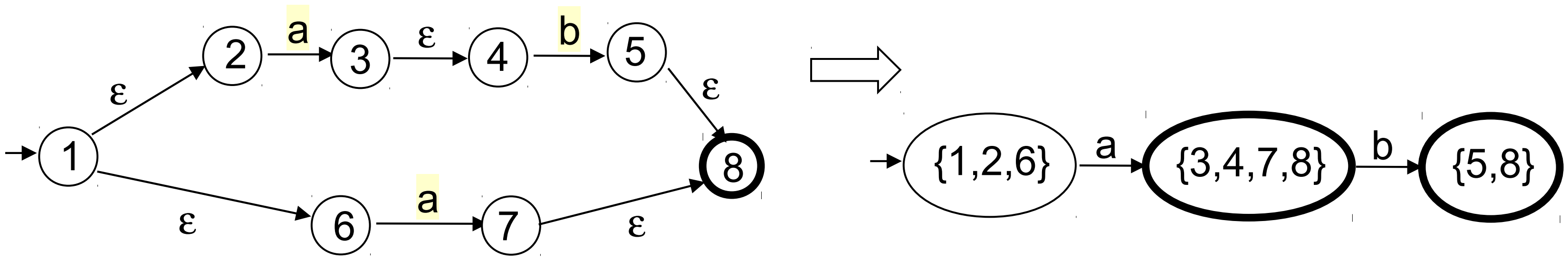
Examples NFA → DFA

1. a^*



$s'_0 = \underline{s_0} = \underline{1} = \{1, 2, 4\}$

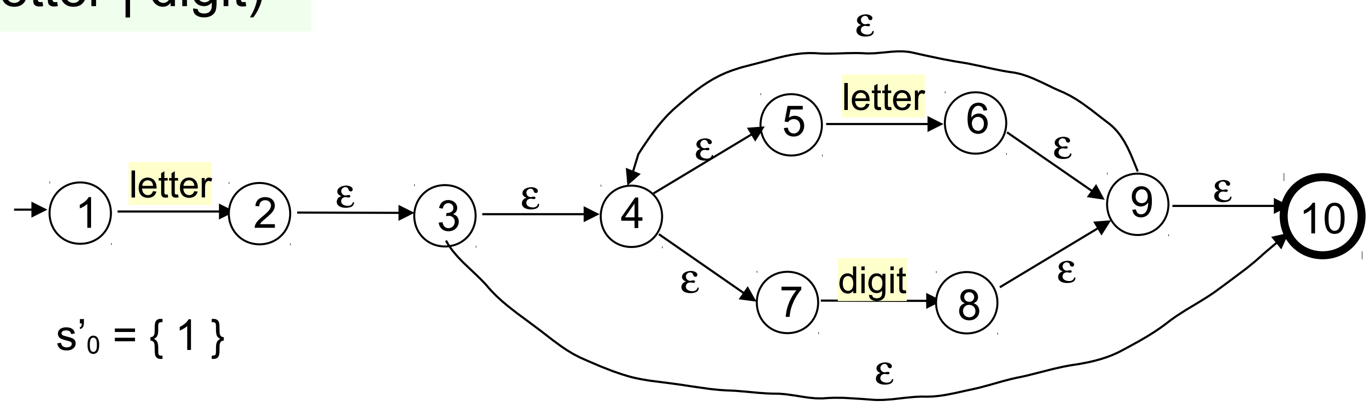
2. ab | a



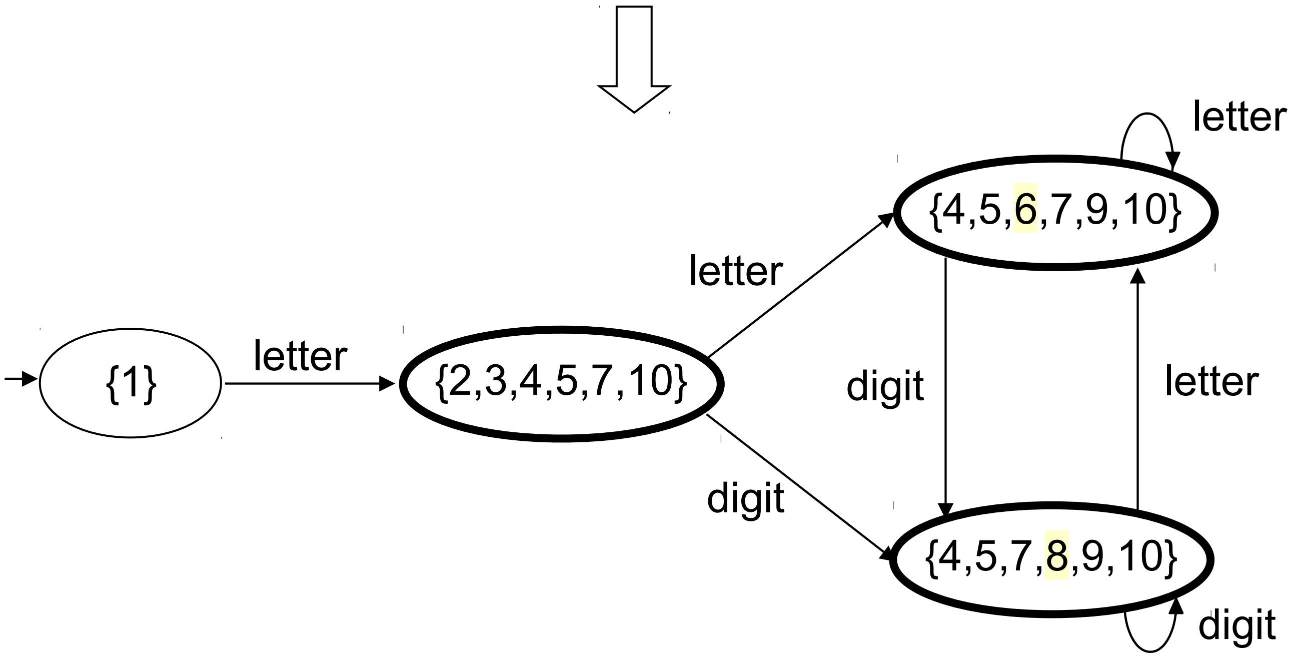
$s'_0 = \underline{1} = \{1, 2, 6\}$

Examples NFA → DFA (ii)

3. letter (letter | digit)*



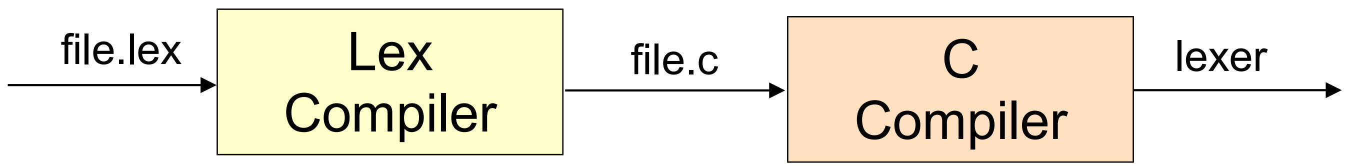
$s'_0 = \{ 1 \}$



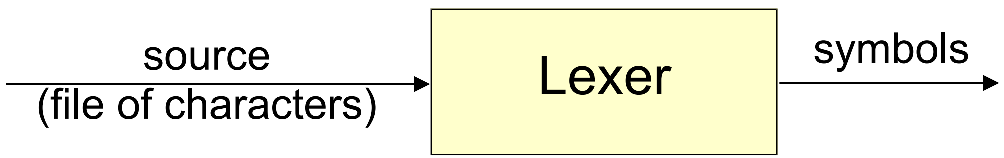
Lex (lexical analyzer generator)

- Broad applicability: sw tool recognizing patterns of characters defined by regexpr
  → then ?

- Lex
  - language (regexpr, actions)
  - compiler → C code (source-to-source)



- Lexical analyzer:



Lex (ii)

- Lex specification (program): 3 sections

Declarations
%%
Translation rules
%%
Auxiliary functions

- Declarations
  - black box (auxiliary definitions): **%{** #include, constants, variables **%}**
  - white box (regular definitions): *name* *expr*

- Translation rules

Regexpr	*C fragment*
r_1	{ $action_1$ }
r_2	{ $action_2$ }
r_n	{ $action_n$ }

- Auxiliary functions: necessary to actions → possibly compiled separately

- Flex (GNU) → `yylex()` = C function in `file.c` that
  - recognizes the symbol
  - executes the action

Lex (iii)

- Notation for the specification of regular expressions
  - Matching of strings of characters: `if` or `"if"`
  - Neutralization of the meta-character effect `"("` / `\(` → acts on a single character
  - Uniformity with C to denote space characters `\t` / `\n`
  - Meta-character • ≡ any character ≠ `\n`
  - Canonical interpretation of meta-characters `*  +  (  )  |  ?` → <u>textual</u> form

    <u>Example</u>: " Set of strings of *a*, *b*, which start with *aa* or *bb*, and end with an optional *c* "

    Σ = {a,b,c} `(aa|bb)(a|b)*c?`
  - Complement set: ^ = first character within the range `[^0-9abc]` = Σ – {0,1,...,9,a,b,c}

    <u>Example</u>: "Numbers in scientific notation" `("+"|"-")?[0-9]+("."[0-9]+)?(E("+"|"-")?[0-9]+)?`
  - Reference to names of regexpr by {*name*}

    ```
    nat     [0-9]+
    snat    ("+"|"-")?{nat}
    ```
    → only when referenced
  - Further meta-characters
    - (expr){*n*}, (expr){*n*,*m*} (*m* missing → infinite)
    - ($expr_1$)/($expr_2$): matching of $expr_1$ only if followed by $expr_2$ (contextual)

Lex (ex1.lex)

Generation of lines preceded by their position number

```
%{
#include <stdio.h>
int l = 1;
%}
%option noyywrap
line     .*\n
%%
{line}   {printf("%d %s", l++, yytext);}
%%
main()
{
  yylex();
}
```

termination option

regular definition

translation rule

lexical string (lexeme)

Lex (`ex2.lex`)

Generation of lines in odd position

```
%{
#include <stdio.h>
int l = 1;
%}
%option  noyywrap
line     .*\n
%%
{line}   { if(l++%2)
              printf("%s", yytext);
         }
%%
main()
{
  yylex();
}
```

Lex (`ex3.lex`)

Substitution of numbers from decimal to hexadecimal notation + printing of number of actual substitutions

```
%{
#include  <stdio.h>
#include  <stdlib.h>
int cont  = 0;
%}
%option noyywrap
digit      [0-9]
num        {digit}+
%%
{num} {   int n = atoi(yytext);
          printf("%x", n);
          if (n > 9) cont++; }
%%
main()
{
   yylex();
   fprintf(stderr, "Tot substitutions = %d\n", cont);
}
```

Note: Default action: when a character (or a string of characters) is not part of any symbol

⇩

ECHO on output

Lex (ex4.lex)

Substitution of numbers with value >= 10 from decimal to hexadecimal notation + printing of number of substitutions

```
%{
#include  <stdio.h>
#include  <stdlib.h>
int cont  = 0;
%}
%option noyywrap
digit     [0-9]
num       {digit}{digit}+
%%
{num} {   int n = atoi(yytext);
          printf("%x", n);
          cont++; }
%%
main()
{
   yylex();
   fprintf(stderr, "Tot substitutions = %d\n", cont);
}
```

Lex (ex5.lex)

Selection of lines which either start or end with character a

```
%{
#include <stdio.h>
%}
%option noyywrap
a_line    a.*\n
line_a    .*a\n
%%
{a_line} ECHO;
{line_a} ECHO;
.*\n       ;
%%
main()
{
   yylex();
}
```

empty action

output of yytext

<u>Note</u>: Ambiguous set of rules (a string may correspond to different regexpr, e.g: *a*)

⇩

Priority rules (built-in)

1. Principle of maximal munch.
2. If ∃ several rules matching the string → select the rule specified <u>first</u>.

```
.*\n            ;
{a_line}    ECHO;
{line_a}    ECHO;
```

⇨ empty output!

```
{a_line}    ECHO;
{line_a}    ECHO;
```

⇨ output = input!

Lex (`ex6.lex`)

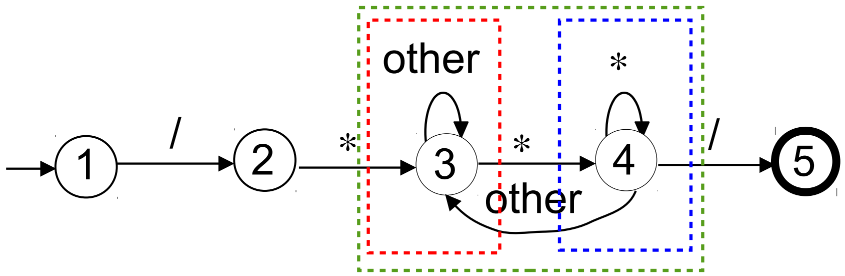
Selection of lines which both start and end with character a and contain other characters, all different from a

```
%{
#include <stdio.h>
%}
%option noyywrap
a_line_a   a[^a\n]+a\n
%%
{a_line_a} ECHO;
.*\n       ;
%%
main()
{
   yylex();
}
```

Lex (ex7.lex)

Substitution of uppercase with lowercase letters, except those in C comments

```
%{
#include <stdio.h>
#define FALSE 0
#define TRUE 1
%}
%option noyywrap
%%
[A-Z]     {putchar(tolower(yytext[0]));}
"/*"     {char c; int end = FALSE;
          ECHO;
          do { while ((c=input()) != '*')
                 putchar(c);
               putchar(c);
               while ((c=input()) == '*')
                 putchar(c);
               putchar(c);
               if(c == '/')
                 end = TRUE;
             } while(!end);
          }
%%
main(){ yylex(); }
```



Internal Lex identifiers:

Identifier	*Description*
`yylex()`	Function of lexical analysis
`yytext`	Lexical string *(lexeme)*
`yyleng`	`strlen(yytext)`
`yyin`	Input file (default: `stdin`)
`yyout`	Output file (default: `stdout`)
`input()`	Input of one character ← `yyin`
ECHO	Default action: prints `yytext` on `yyout`

Lex (ex8.lex)

Substitution of uppercase with lowercase letters, except those in Pascal comments

$\{ \; (\sim\})^{*} \}$

```
%{
#include <stdio.h>
%}
%option noyywrap
%%
[A-Z]      {putchar(tolower(yytext[0]));}
\{[^\}]*\}  ECHO;
%%
main()
{
  yylex();
}
```

Lex (ex9.lex)

Counting of chars, words and lines (wc), where word = list of non-blank chars

```
%{
#include <stdio.h>
int nc=0, nw=0, nl=0;
%}
%option noyywrap
word     [^ \t\n]+
eol      \n
%%
{word}   {nw++; nc+=yyleng;}
{eol}    {nl++; nc++;}
.        {nc++;}
%%
main()
{
  yylex();
  printf("%d %d %d\n", nl, nw, nc);
}
```

either space or tab

Lex: Lexical Analysis

Recognition of lexical symbols in a programming language

```
%{
#include <stdlib.h>
#include "def.h"  /* IF, THEN, ELSE, ID, NUM, RELOP, LT, LE, EQ, NE, GT, GE */
int lexval;
%}
delimiter     [ \t\n]
spacing       {delimiter}+
letter        [A-Za-z]
digit         [0-9]
id            {letter}({letter}|{digit})*
num           {digit}+
%%
{spacing}     ;
if            {return(IF);}
then          {return(THEN);}
else          {return(ELSE);}
{id}          {lexval = store_id(); return(ID);}
{num}         {lexval = atoi(yytext); return(NUM);}
"<"           {lexval = LT; return(RELOP);}
"<="          {lexval = LE; return(RELOP);}
"="           {lexval = EQ; return(RELOP);}
"<>"          {lexval = NE; return(RELOP);}
">"           {lexval = GT; return(RELOP);}
">="          {lexval = GE; return(RELOP);}
%%
int store_id()  /* symbol table without keywords */
{ int line;
  if((line = lookup(yytext)) == 0) line = insert(yytext);
  return(line);
}
```