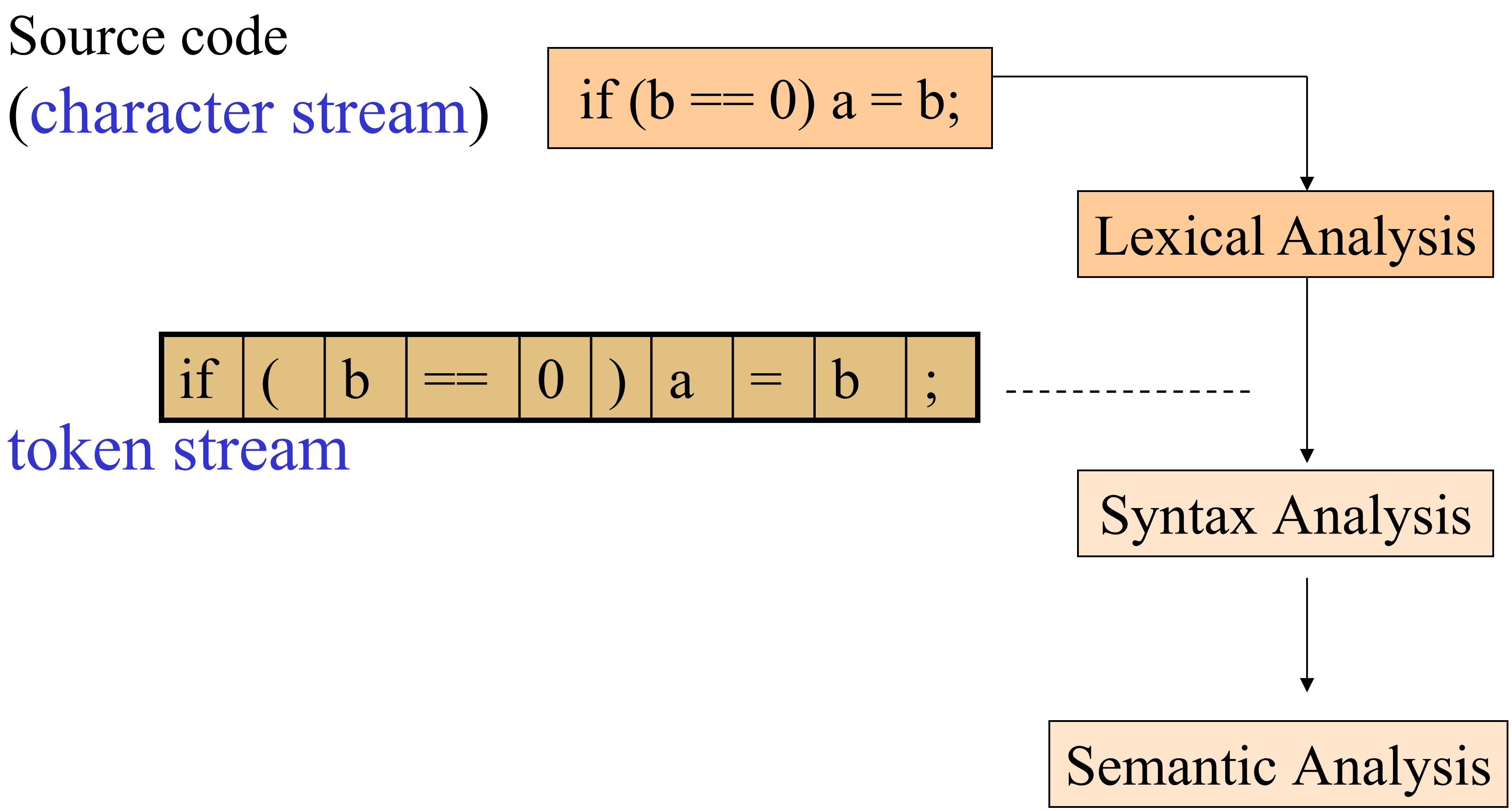
Chapter 2 Scanning (lexical analysis)

2022 Spring&Summer

Outline

- What is lexical analysis?
- Writing a lexer
- Specifying tokens: regular expressions
- DFAs, NFAs
- DFA simulation
- NFA-DFA conversion

2.1 Scanning Process



2.1 Scanning Process

- **Identifier:** x y11 elsen_i00
- **Integer:** 2 1000 -500 5L
- **Floating point:** 2.0 0.00020 .02 1. 1e5 0.e-10
- **String:** “x” “He said, \“Are you?\””
- **Comment:** /** don’t change this **/
- **Keyword:** if else while break
- **Symbol:** + * { } ++ < << [ ] >=

2.2 Regular Expression

- Describe programming language tokens using regular expressions!
- A Regular Expression (RE) is defined inductively:

  a ordinary character stands for itself

  ε the empty string

  R|S either R or S (alternation), where R, S = RE

  RS R followed by S (concatenation), where R, S = RE

  R* concatenation of a RE R zero or more times (R* = ε|R|RR|RRR|RRRR…)

- Precedence Rules and Parentheses

  alternation < concatenation < repetition

2.2 Regular Expression

Example

1) $\sum=\{a,b,c\}$

- the set of all strings over this alphabet that contain exactly one b.
- $(a|c)^*b(a|c)^*$

2) $\sum=\{a,b,c\}$

- the set of all strings that contain at most one b.
- $(a|c)^*|(a|c)^*b(a|c)^*$ $\quad$ $(a|c)^*(b|\varepsilon)(a|c)^*$

the same language may be generated by many different regular expressions.

2.2 Regular Expression

3) $\sum$={ a,b}

the set of strings consists of a single b surrounded by the same number of a's.

S = {b,aba,aabaa,aaabaaa,……} = { $a^n b a^n \mid n \neq 0$ }

This set can not be described by a regular expression.

"Regular expression can't count "

A Regular Set : a set of strings that is the language for a regular expression is distinguished from other sets

2.2 Regular Expression

- R+ one or more strings from L(R): R(R*)
- R? optional R: (R|ε)
- [abce] one of the listed characters: (a|b|c|e)
- [a-z] one character from this range:(a|b|c|d|e|…|y|z)
- [^ab] anything but one of the listed chars
- [^a-z] one character not from this range

2.2 Regular Expression

1. Numbers

   nat = [0-9]+

   signedNat = (+|-)?*nat*

   number = *signedNat*(“. ”*nat*)? (E *signedNat*)?

2. Reserved Words and Identifiers

   reserved = if | while | do |………

   letter = [a-z A-Z]

   digit = [0-9]

   identifier = *letter (letter* | *digit)**

2.2 Regular Expression

3. Comment

Several forms:

/* this is a C comment */ ba(~(ab))*ab *(wrong)*

"/*" ([^*/] | [^*] "/" | "*" [^/]) * "*/"

not include //。。。。。*

*not include /***/*

"/* ""/ " * ([^*/] | [^*] "/ " | "* "[^/]) * "* " * " */ "

{ this is a pascal comment } {(~})*}

; this is a schema comment

-- this is an Ada comment --(~*newline*)*

2.2 Regular Expression

4. Ambiguity, White Space, and Lookahead

Ambiguity: some strings can be matched by several different regular expressions.

- an identifier or a keyword (keyword interpretation is preferred.)
- a single token or a sequence of several tokens (the single-token interpretation is preferred.) -- the principle of longest substring.

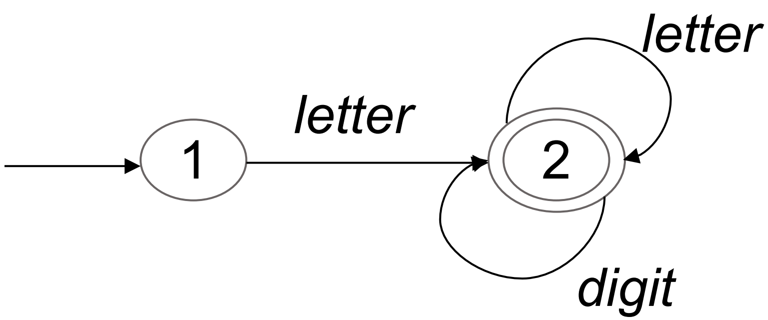
Delimiters: characters that are unambiguously part of other tokens are delimiters.

- *whitespace = (newline | blank | tab | comment)+*
- Lookahead: *buffering of input characters , marking places for backtracing*

2.3 Finite Automata

- Finite automata(finite-state machines) are a mathematical way of describing particular kinds of algorithms.
- A strong relationship between finite automata and regular expression

identifier = *letter* (*letter* | *digit*)*



- **Transition:** record a change from one state to another upon a match of the character or characters by which they are labeled.
- **start state:** the recognition process begins. drawing an unlabeled arrowed line to it coming "*from nowhere*"
- **accepting states:** represent the end of the recognition process. drawing a double-line border around the state in the diagram.

2.3.1 Definite of Deterministic finite automation(DFA)

- DFA: automata where the next state is uniquely given by the current state and the current input character.
- Defintion of a DFA:

  A DFA (*deterministic finite automation*) M consist of an alphabet Σ, a set of states S, a transition function T : $S \times \Sigma \rightarrow S$, a *start* state $s_0 \in S$, and a set of *accepting* states $F \subset S$. The language accepted by M ,written *L(M)*, is defined to be the set of strings of characters $c_1c_2c_3\ldots.c_n$ with each $c_i \in \Sigma$ such that there exist states $s_1=t(s_0,c_1),s_2=t(s_1,c_2)$, $s_n = T(s_{n-1},c_n)$ with s_n is an element of F.

2.3.1 Definite of Deterministic finite automation(DFA)

- Accepting state s_n means the diagram:

  $\rightarrow S0 \rightarrow S1 \rightarrow S2 \rightarrow \ldots\ldots\ldots Sn\text{-}1 \rightarrow Sn$

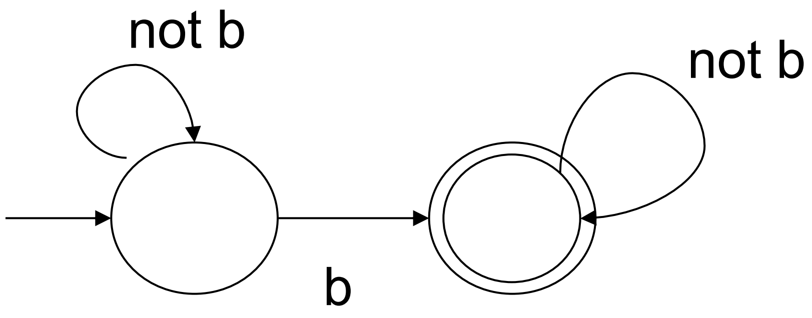
- Some difference:

  1. The definition does not restrict the set of states to numbers
  2. Don't label the transitions with characters but with names representing a set of characters
  3. Definitions T: $S \times \sum \rightarrow S$, T(s, c) must have a value for every s and c.

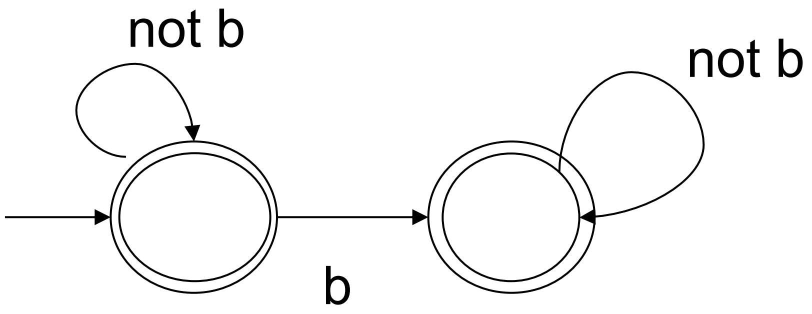
  But in the diagram, T(*start*,*c*) defined only if *c* is a letter, T(in_id, *c*) is defined only if *c* is a letter or a digit. The convention is that error transitions are not drawn in the diagram.

2.3.1 Definite of Deterministic finite automation(DFA)

- Example 2.6: exactly accept one b



- Example 2.7: at most one b



2.3.1 Definite of Deterministic finite automation(DFA)

- Example 2.8

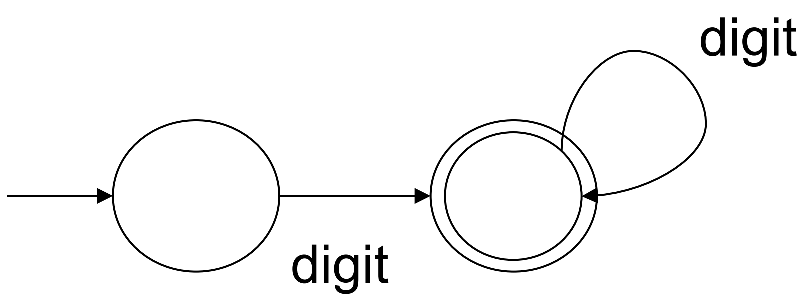
  digit = [0-9]

  nat = digit +

  signedNat = (+|-)? Nat

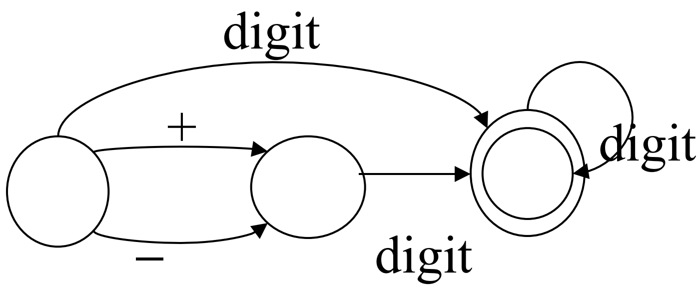
  Number = singedNat(“.”nat)?(E signedNat)?

- A DFA of nat:

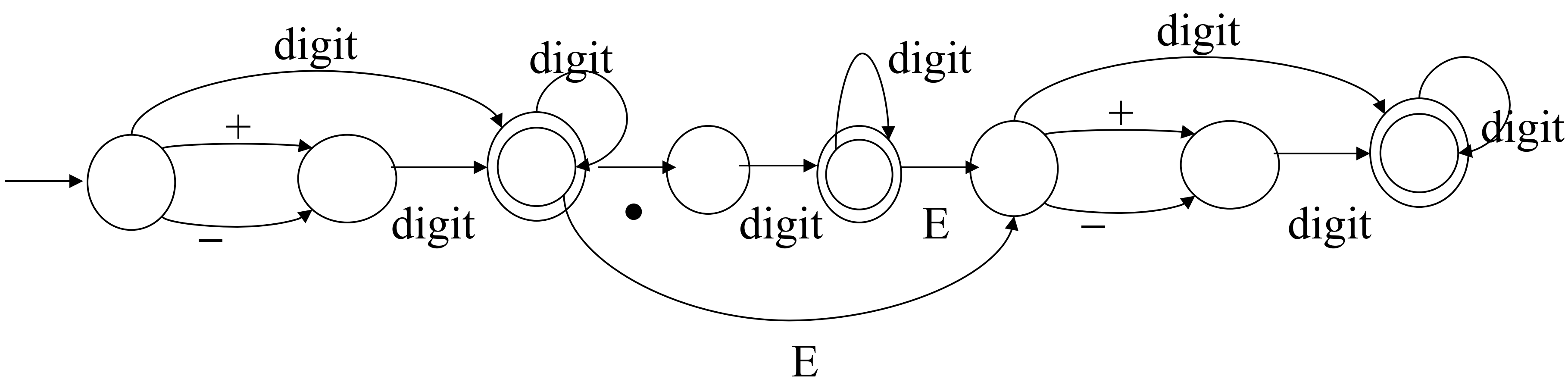


2.3.1 Definite of Deterministic finite automation(DFA)

- A DFA of signedNat:

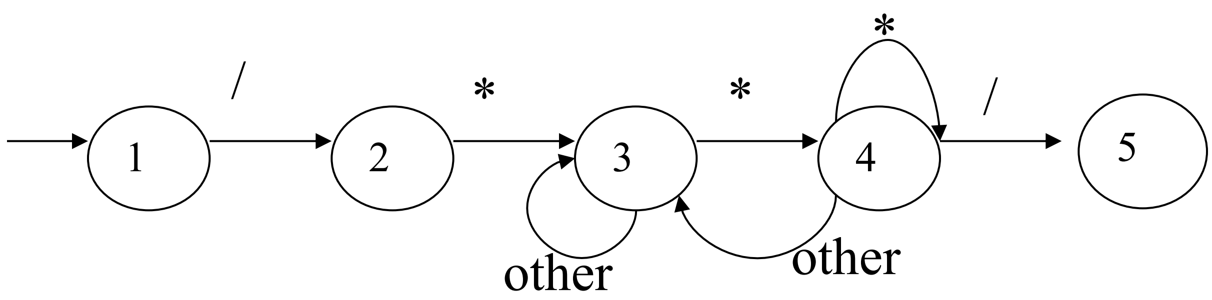


- A DFA of number



2.3.1 Definite of Deterministic finite automation(DFA)

- A DFA of C comments: (easily than write down a regular expression)

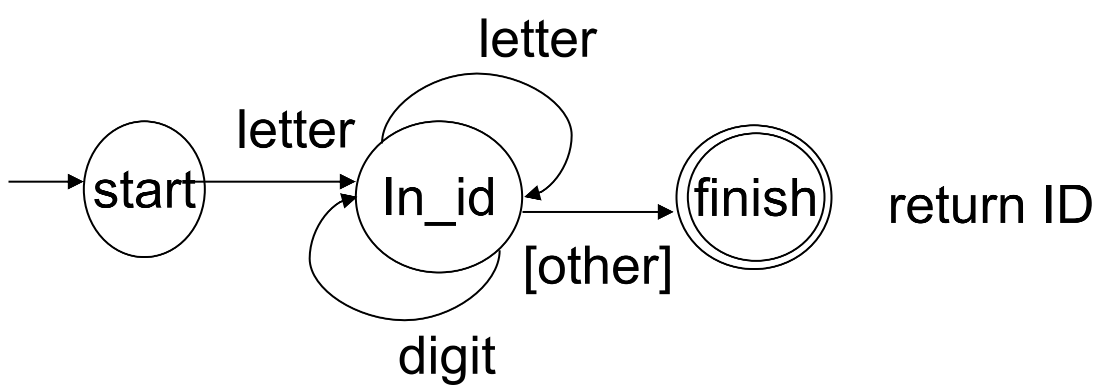


2.3.2 Lookahead,backtracking and nondeterministic automata

A typical action

1. making a transition: move the character from the input string to a string (the token string value or lexeme of the token)
2. reaching an accepting state : return the token just recognized, along with any associated attributes.
3. reaching an error state : either back up in the input (backtracking) or to generate an error token.

- Example: finite automation for an identifier with delimiter and return value

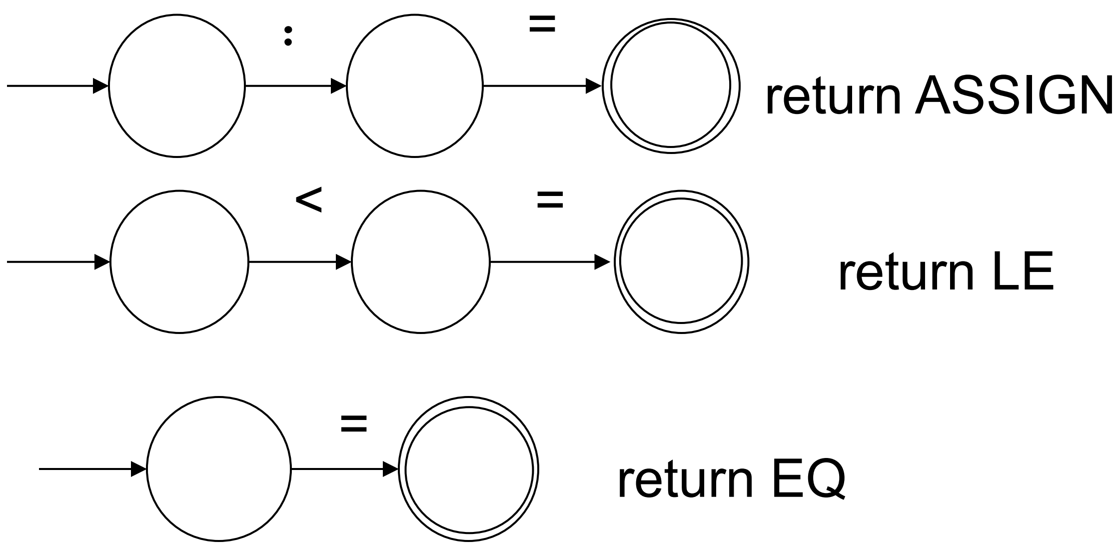


2.3.2 Lookahead,backtracking and nondeterministic automata

- How to arrive at the start state in the first place:

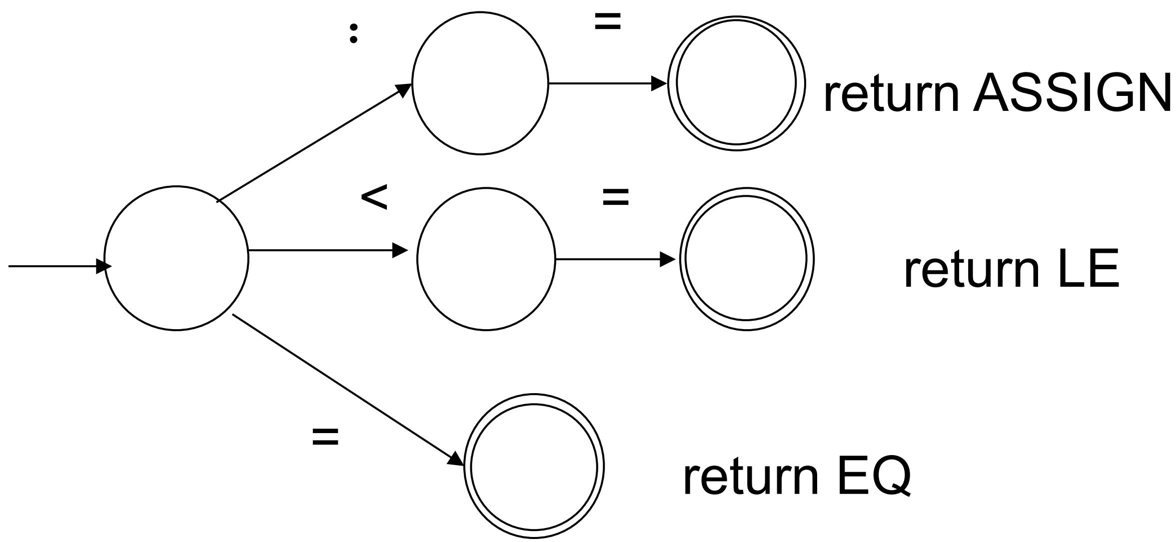
(combine all the tokens into one giant DFA)

1. each of these tokens begins with a different character uniting all of their start states into a single start state.



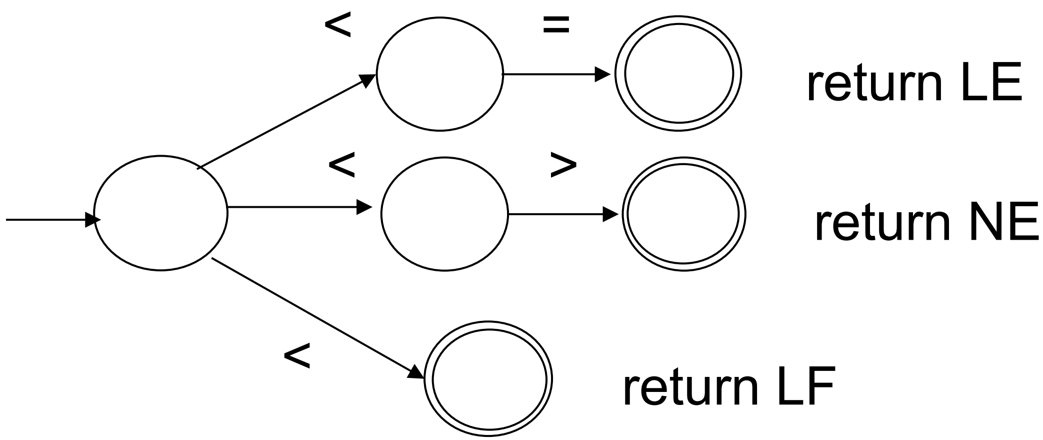
2.3.2 Lookahead,backtracking and nondeterministic automata

- Identify their start states to get the following DFA:

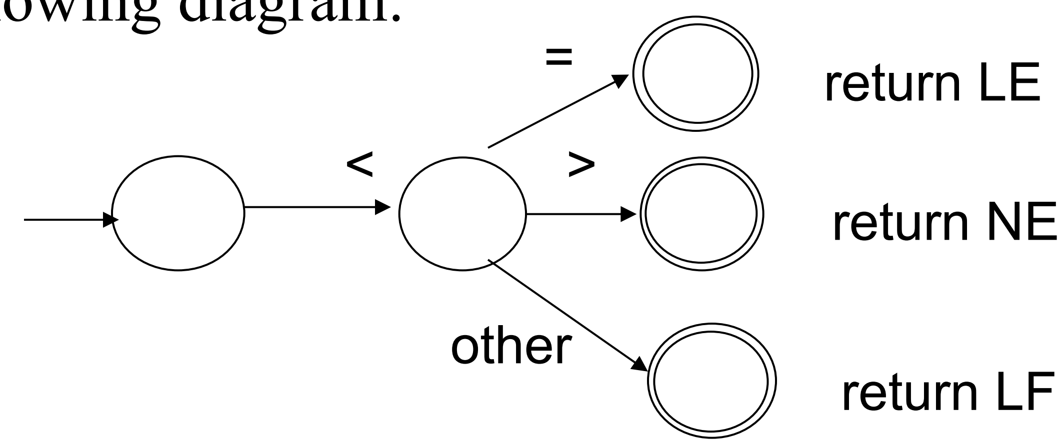


2.3.2 Lookahead,backtracking and nondeterministic automata

- Several tokens that begin with the same character, such as <, <=, and <>,



- There is a unique transition to be made in each state, such as in the following diagram:



2.3.2 Lookahead, backtracking and nondeterministic automata

NFA: *nondeterministic finite automaton*,

Developing an algorithm for turning these NFA into DFAs.

ε -transition: transition that may occur without consulting the input string (and without consuming any characters)

It may be viewed as a "*match*" of the empty string.

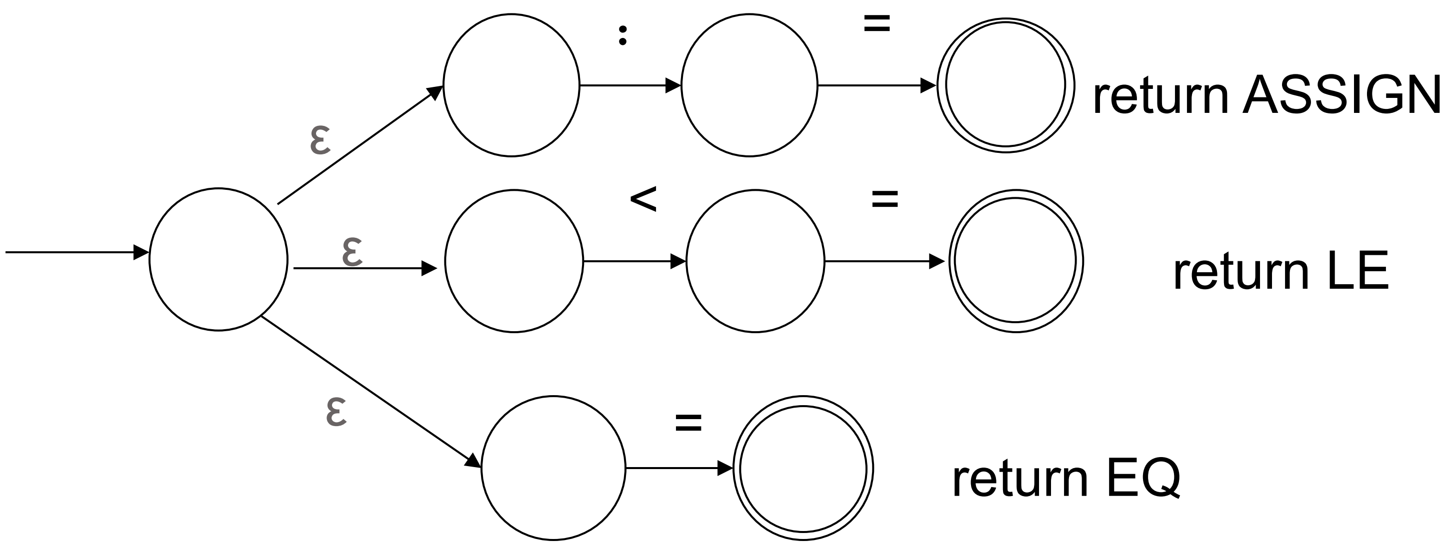
(This should not be confused with a match of the character ε in the input)

2.3.2 Lookahead, backtracking and nondeterministic automata

ε-transitions are useful in two ways.

I. Express a choice of alternatives in a way that does not involve combining states. *Advantage:* keeping the original automata intact and only adding a new start state to connect them.

II. Describe a match of the empty string explicitly.



2.3.2 Lookahead, backtracking and nondeterministic automata

- Definition: An **NFA** (*nondeterministic finite automaton*) *M* consists of an alphabet Σ, a set of states S,
  - a transition function T: $S \times (\Sigma \cup \{\varepsilon\}) \rightarrow \wp(S)$,
  - a start state s_0 from S, and a set of accepting states A from S.
  - The language accepted by *M*, written $L(M)$, is defined to be the set of strings of characters $c_1c_2 \ldots c_n$ with each c_i from $\Sigma \cup \{\varepsilon\}$ such that there exist states s_1 in $T(s_0, c_1)$, s_2 in (s_1, c_2),..., s_n in $T(s_{n-1}, c_n)$ with s_n an element of A.

2.3.2 Lookahead, backtracking and nondeterministic automata

- Note:

1. Any of the c_i in $c_1c_2\ldots\ldots c_n$ may be ε, the string $c_1c_2..\,.c_n$ may actually have fewer than n characters in it.

2. The sequence of states $s_1,\ldots, s_n$ are chosen from the *sets* of states $T(s_0, c_1),\ldots, T(s_{n-1}, c_n)$, and this choice will **not** always be *uniquely* determined.

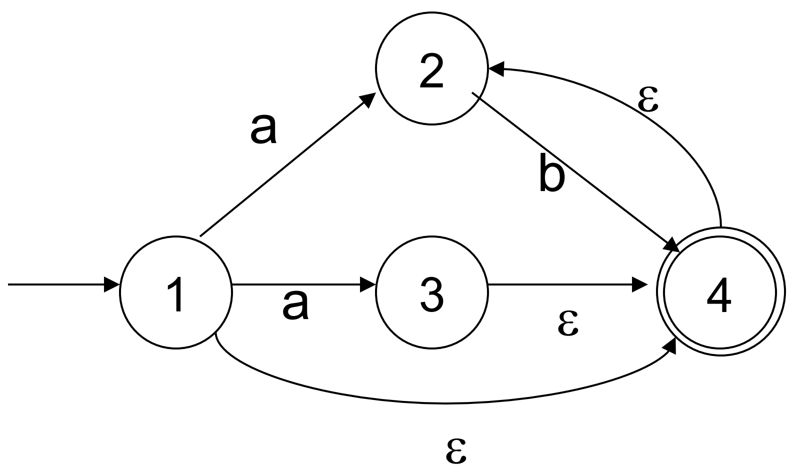
Arbitrary numbers of ε 's can be introduced into the string at any point, corresponding to any number of ε -transitions in the NFA.

2.3.2 Lookahead, backtracking and nondeterministic automata

- An NFA does not represent an algorithm. However, it can be simulated by an algorithm that backtracks through every nondeterministic choice.

Example: a string *abb*: $\rightarrow 1 \rightarrow 2 \rightarrow 4 \rightarrow 2 \rightarrow 4$

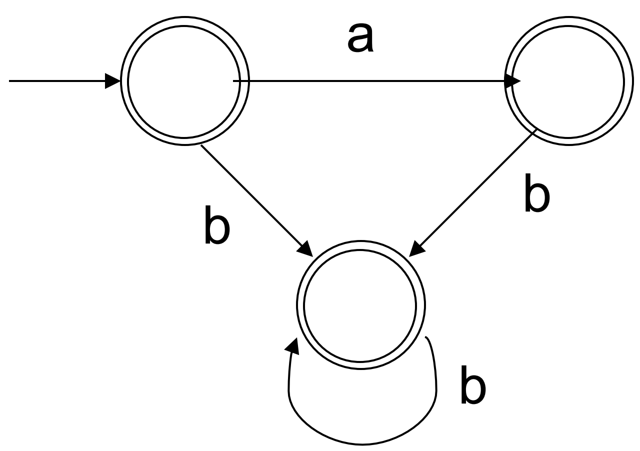
$\rightarrow 1 \rightarrow 3 \rightarrow 4 \rightarrow 2 \rightarrow 4 \rightarrow 2 \rightarrow 4$



2.3.2 Lookahead, backtracking and nondeterministic automata

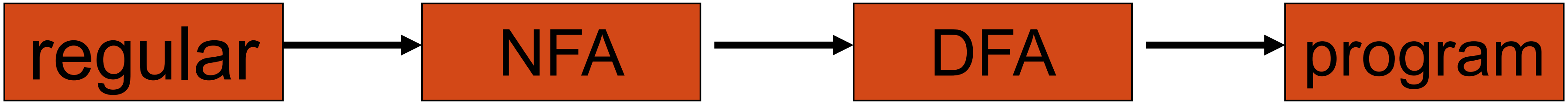
regular expression: $(a|\varepsilon)b^*$.

The DFA :



2.4 From Regular Expression To DFAs

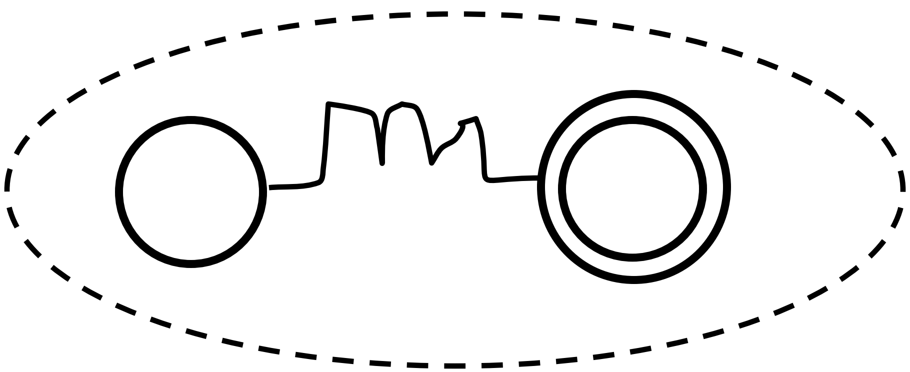
The algorithm : translating a regular expression into a DFA .



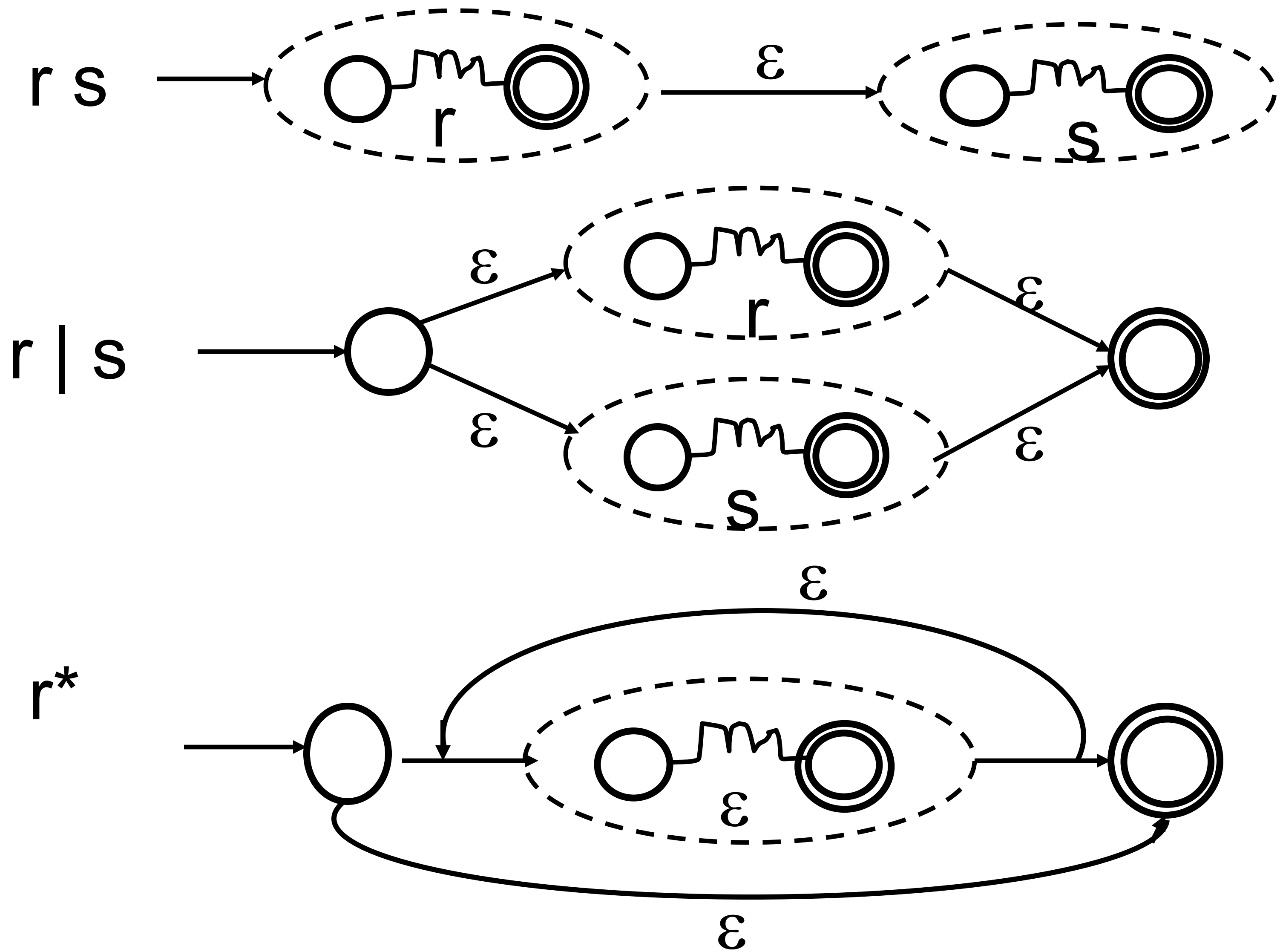
2.4.1 from a regular expression to an NFA

Thompson's construction:

ε-transitions: to "*glue together*" the machine of each piece of a regular expression to form a machine that corresponds to the whole expression.



2.4.1 from a regular expression to an NFA



2.4.2 from an NFA to a DFA

The algorithm is called the subset construction.

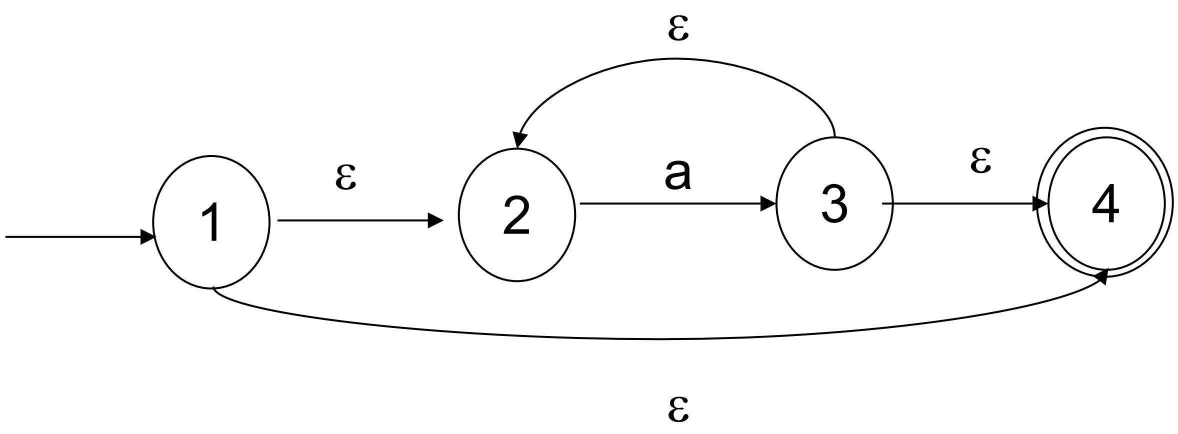
1、 ***the ε-closure of a Set of states:***

the ε-closure of a single state *s* is the set of states reachable by a series of zero or more ε-transitions.

the ε-closure of a set of states : the union of the ε-closures of each individual state.

2.4.2 from an NFA to a DFA

- Example 2.14: regular a*

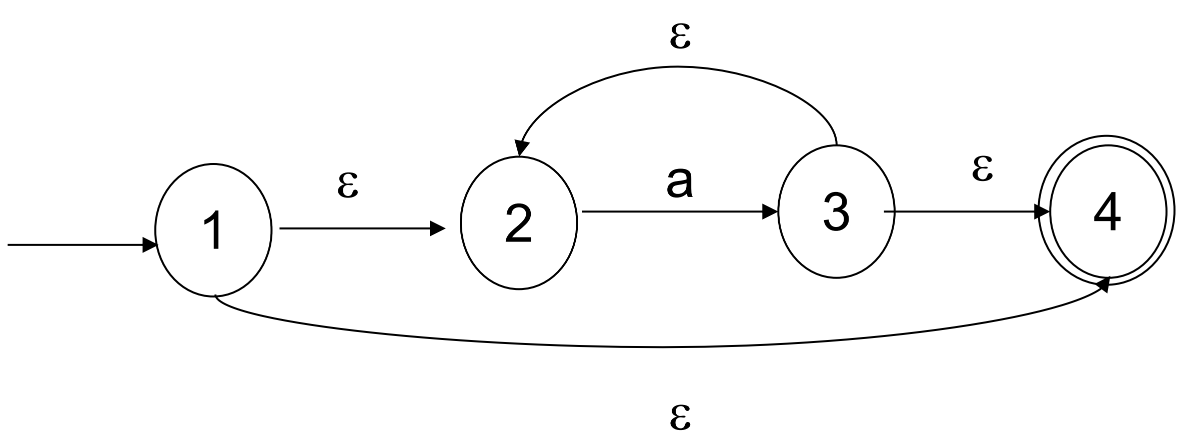


$\overline{1} = \{ 1, 2, 4 \}$, $\overline{2} = \{ 2 \}$, $\overline{3} = \{ 2, 3, 4 \}$, $\overline{4} = \{ 4 \}$

2.4.2 from an NFA to a DFA

The ε-closure of a set of states : the union of the ε-closures of each individual state.

$$\overline{S} = \bigcup_{s \, In \, S} \overline{s}$$



$\overline{\{1,3\}} = \overline{1} \cup \overline{3} = \{ 1,2,4\} \cup \{ 2,3,4\} = \{ 1,2,3,4\}$

2.4.2 from an NFA to a DFA

2、the *Subset Construction:*

1) Compute the *ε-closure* of the start state of *M;* this becomes the *start state*.

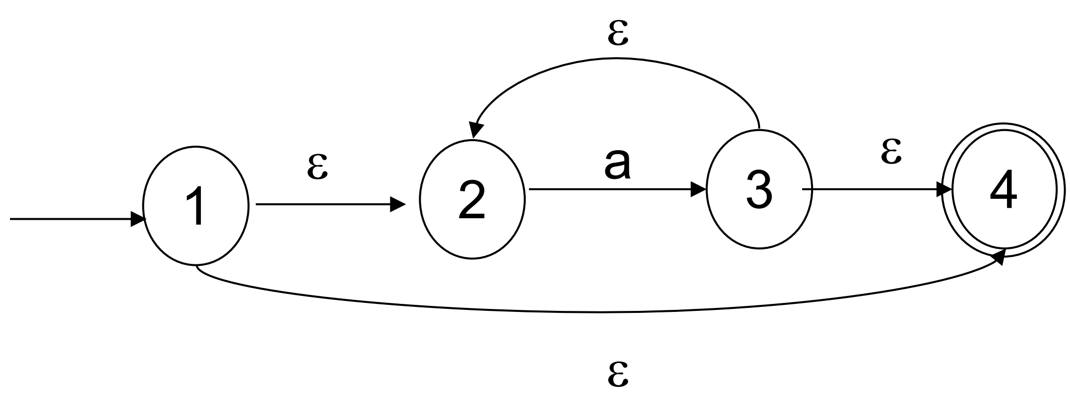
2) Given a set *S* of states and a character *a* in the alphabet, compute the set $S'_a = \{ t \mid \text{for some } s \text{ in S there is a transition from } s \text{ to } t \text{ on } a \}$. Then, compute , the ε-closure of $\overline{S'_a}$.

3) Continue with this process until no new states or transitions are created.

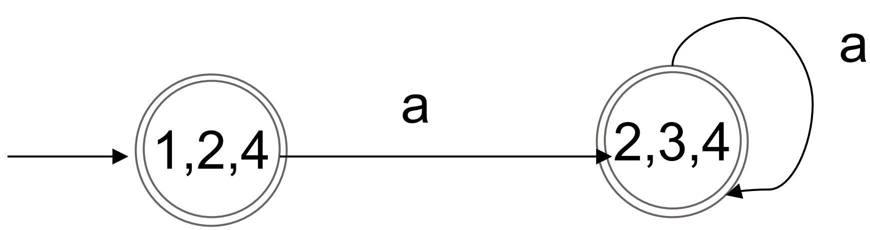
Mark as accepting those states constructed in this manner that contain an accepting state of M.

2.4.2 from an NFA to a DFA

Example 2.15: consider the NFA of example 2.14

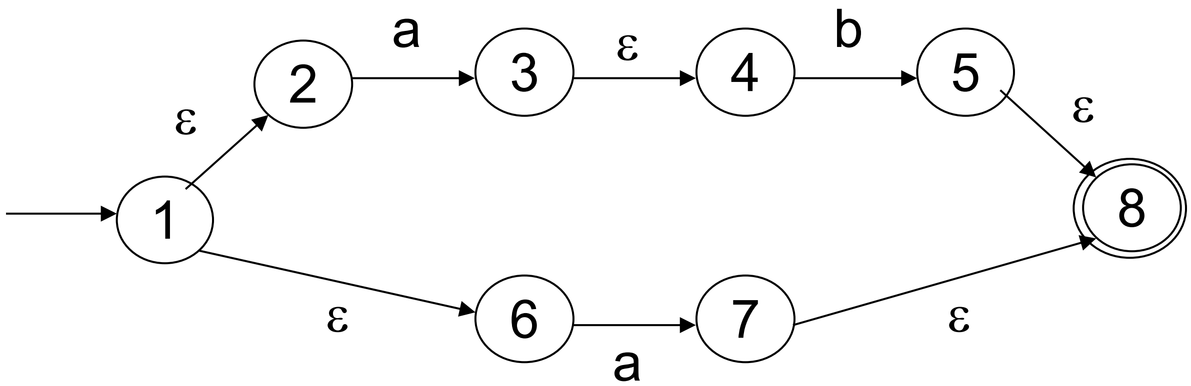


S	S'	$\overline{S'_a}$
1	1,2,4	2,3,4
3	2,3,4	2.3.4

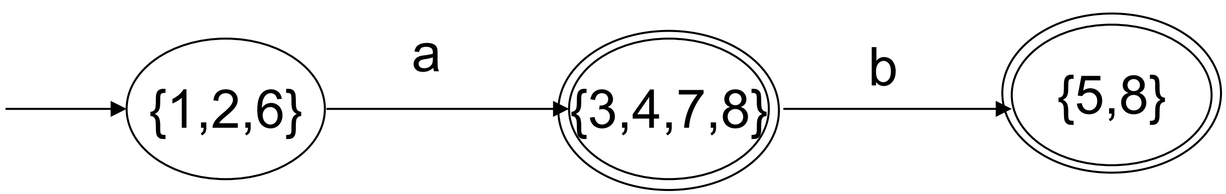


2.4.2 from an NFA to a DFA

Example 2.16: consider the NFA of Figure 2.8

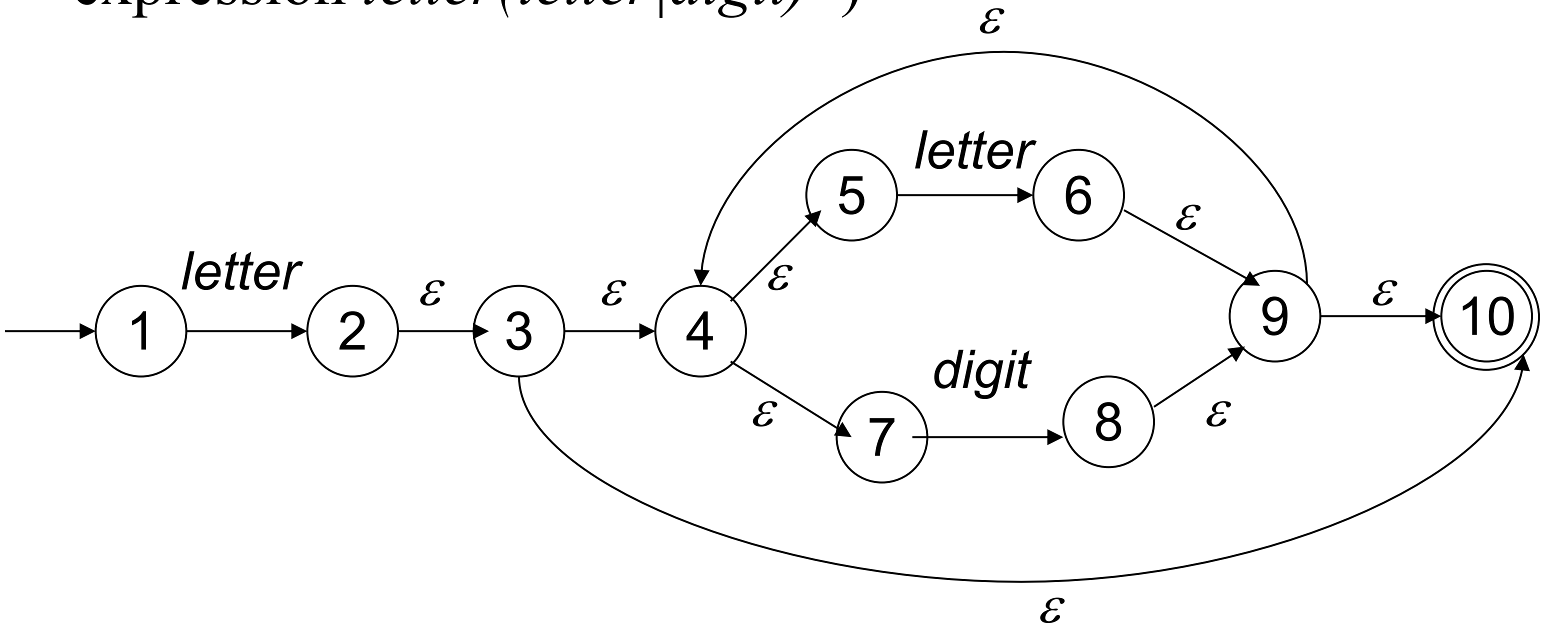


s	S'	$\overline{S'}_a$	$\overline{S'}_b$
1	1,2,6	3,4,7,8	
3,7	3,4,7,8		5,8
5	5,8		



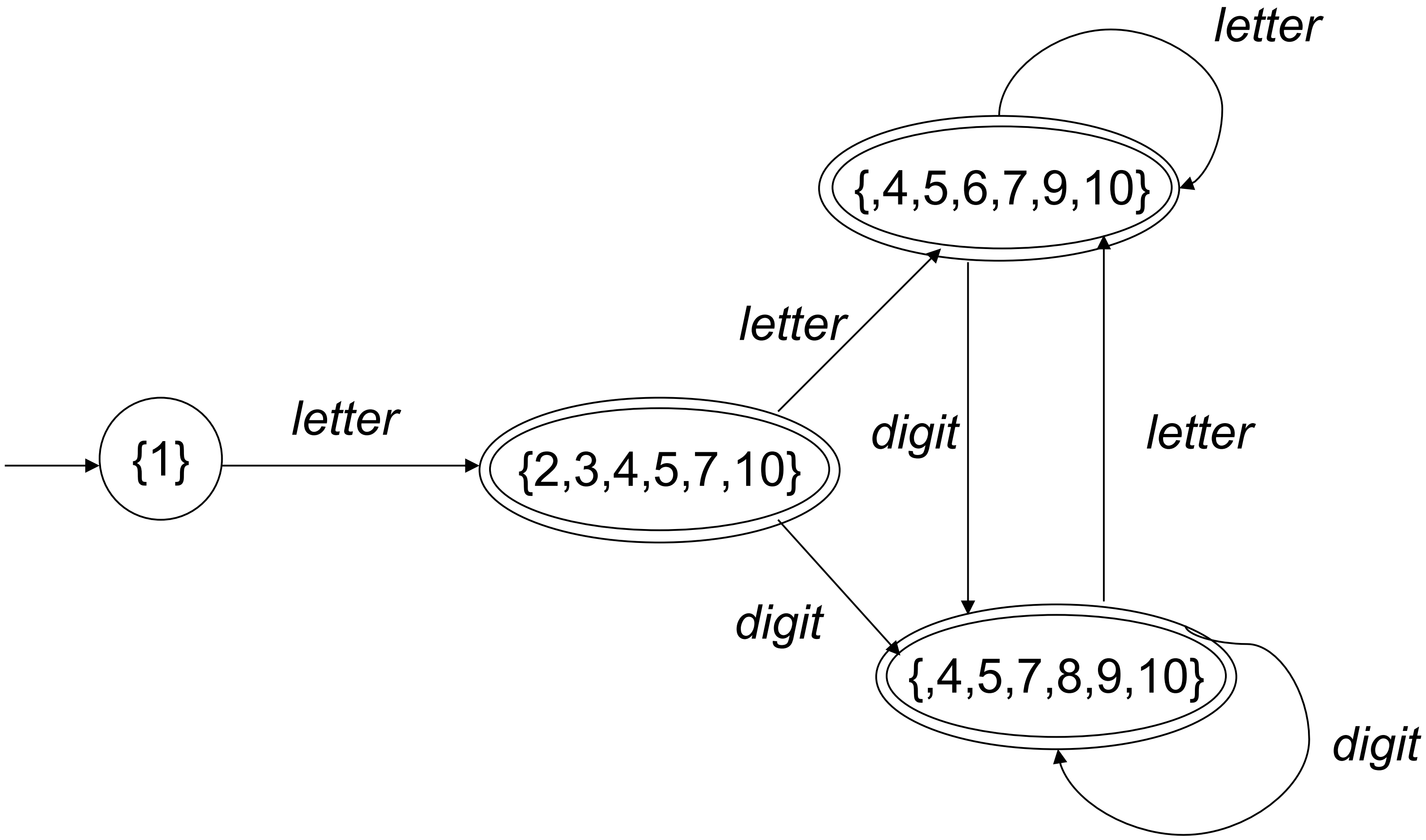
2.4.2 from an NFA to a DFA

Example 2.17: consider the NFA of Figure 2.9 (regular expression *letter(letter|digit)**)



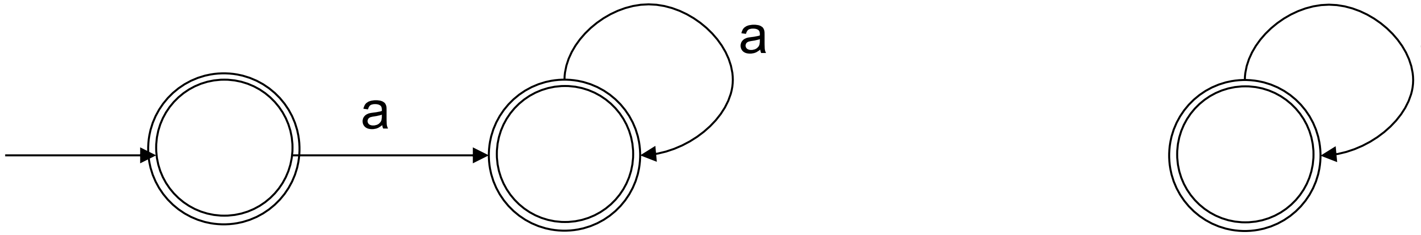
s	S	$\overline{S'_{letter}}$	$\overline{S'_{digit}}$
1	1	2,3,4,5,7,10	
2	2,3,4,5,7,10	4,5,6,7,9,10	4,5,7,8,9,10
6	4,5,6,7,9,10	4,5,6,7,9,10	4,5,7,8,9,10
8	4,5,7,8,9,10	4,5,6,7,9,10	4,5,7,8,9,10

2.4.2 from an NFA to a DFA



2.4.4 Minimizing the number of states in a DFA

- An important result from automata theory states:
  - Given any DFA, there is an equivalent DFA containing a minimum number of states, and that this minimum state DFA is unique (except for renaming of states).
  - It is also possible to directly obtain this minimum state DFA from any given DFA.
  - the resulting DFA may be more complex than necessary. (deriving a DFA algorithmically from a regular expression)



2.4.4 Minimizing the number of states in a DFA

Given the algorithm as follow:

1、It begins with the most optimistic assumption possible: it creates two sets

- One consisting of all the accepting states
- The other consisting of all the nonaccepting states.

2、Given this partition of the states of the original DFA, consider the transitions on each character *a* of the alphabet.

- If all accepting states have transitions on *a* to accepting states.

  defines an *a*-transition from the new accepting state (the set of all the old accepting states) to itself.
- If all accepting states have transitions on *a* to nonaccepting states

  defines an *a*-transition from the new accepting state to the new nonaccepting state (the set of all the old nonaccepting stales).

2.4.4 Minimizing the number of states in a DFA

2、Given this partition of the states of the original DFA, consider the transitions on each character *a* of the alphabet.

- If there are two accepting states *s* and *t* that have transitions on *a* that land in different sets,

  no a-transition can be defined for this grouping of the states. We say that *a* distinguishes the states *s* and *t*.

- If there are two accepting states *s* and *t* such that *s* has an a-transition to another accepting state, while *t* has no a-transition at all (i.e., an error transition) ,

  then *a* distinguishes *s* and *t*.

- If any further sets are split, we must return and repeat the process from the beginning.

This process continues until

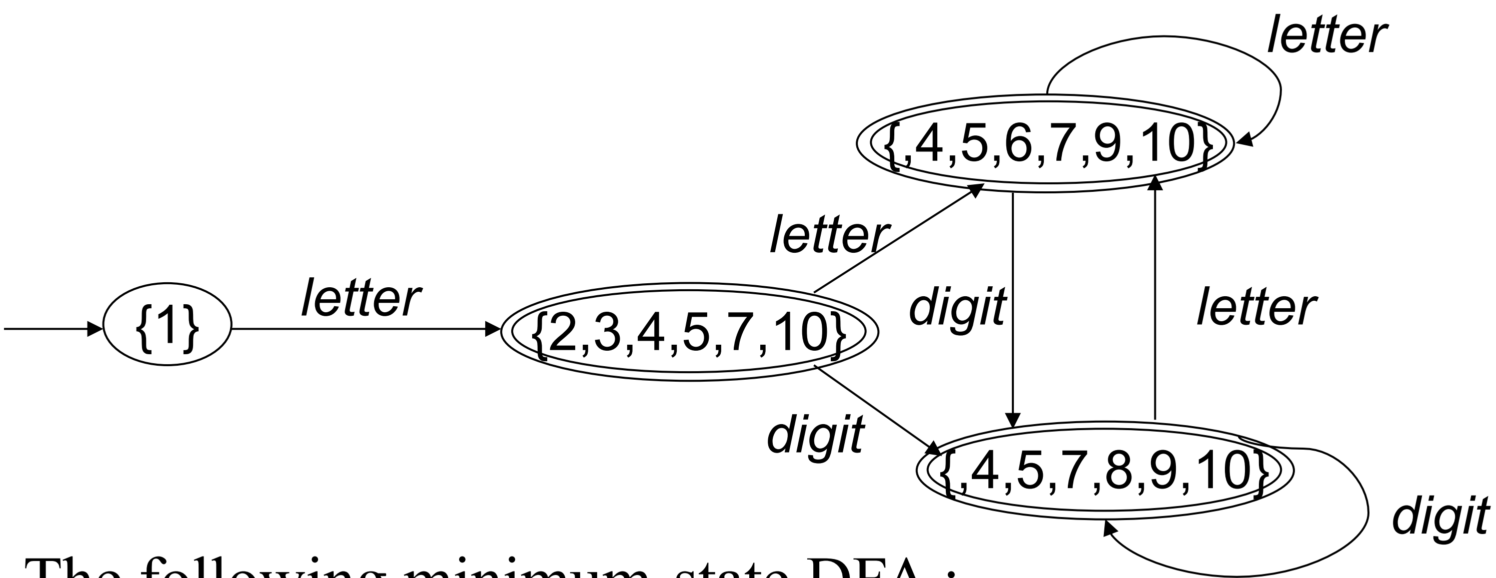
(1) all sets contain only one element (in which case, we have shown the original DFA to be minimal)

(2) until no further splitting of sets occurs.

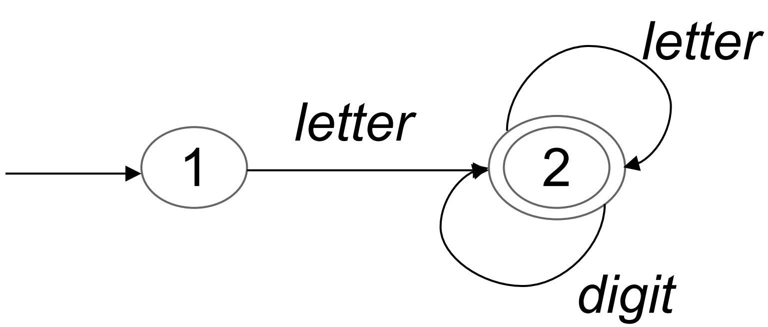
2.4.4 Minimizing the number of states in a DFA

Example 2.18 :the regular expression *letter(letter|digit)**

The accepting sets	{2,3,4,5,7,10},{4,5,6,7,9,10},{4,5,7,8,9,10}
The nonaccepting sets	{1}

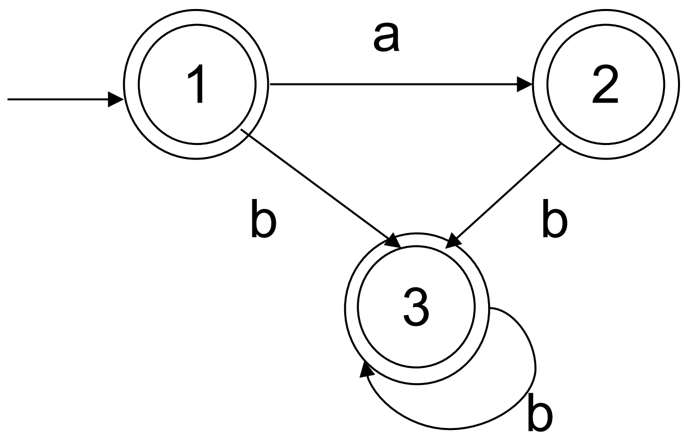


The following minimum-state DFA :

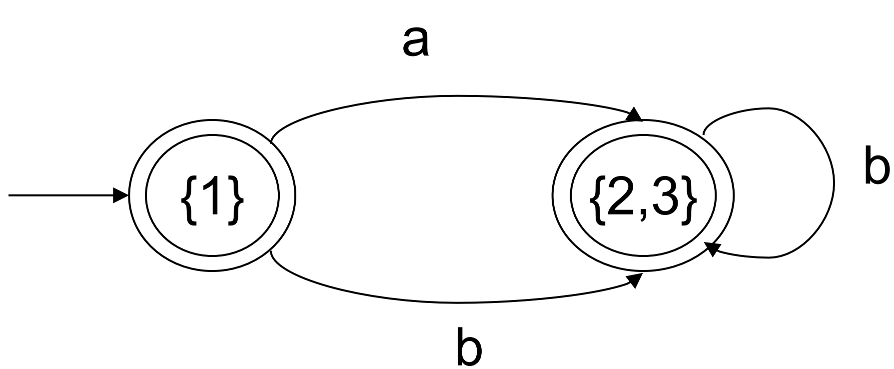


2.4.4 Minimizing the number of states in a DFA

Example 2.19: the regular expression (a| ε)b*



The accepting sets	{1,2,3}



2.5 Implementation of a tiny scanner

- Develop the actual code for a scanner to illustrate the concepts studied so far in this chapter.
- Do this for the TINY language that we introduced informally in Chapter 1 (Section 1.7).

2.5 Implementation of a tiny scanner

.Reserved Words	Special Symbols	Other
if	+	number
then	-	(1 or more digits)
else	*	
end	/	
repeat	=	
until	<	identifier
read	(	(1 or more letters)
write	)	
	;	
	:=	

2.5 Implementation of a tiny scanner

```
{ sample program
In TINY language  -
Computes  factorial
  }
 read x; { input on integer }
 if  0 < x then  { don't compute if x <= 0 }
fact := 1 ;
repeat
fact := fact * x;
x := x - 1
until x = 0;
write fact { output factorial of x }
end
```

2.5 Implementation of a tiny scanner

```
TINY COMPILATION: sample.tny
 1:  { Sample program
 2:   in TINY language –
 3:   computes factorial
 4:  }
 5:  read x; { input an integer }
        5: reserved word: read
        5: id, name= x
        5: ;
 6: if 0 < x then { don't compute if x <= 0 }
        6: reserved word: if
        6: mum, val= 0
        6: <
        6: id, name= x
        6: reserved word: then
```

2.5 Implementation of a tiny scanner

```
7: fact := 1;
        7: id, name= fact
        7: :=
        7: num, val= 1
        7:  ;
8: repeat
        8: reserved word: repeat
9: fact := fact * x;
        9: id, name= fact
        9: :=
        9: id, name= fact
        9: *
        9: id, name= x
        9: ;
```

2.5 Implementation of a tiny scanner

10:x := x – 1

10: id, name= x

10: :=

10: id, name=x

10: -

10: mum, val = 1

11:until x = 0;

11: reserved word: until

11: id, name= x

11: =

11: mum, val= 0

11: ;

12: write fact { output factorial of x }

12: reserved words: write

12: id, name= fact

13:end

13: reserved word: end

14: EOF

2.6 Use of Lex to generate a scanner automatically

- we will use the Lex scanner generator to generate a scanner from a description of the tokens of TINY as regular expressions.
- The most popular version of Lex is called flex {for Fast Lex).

  It is distributed as part of the **Gnu compiler package** produced by the Free Software Foundation, and is also freely available at many Internet sites.

2.6 Use of Lex to generate a scanner automatically

Lex is a program:

- Input : a text file containing regular expressions, together with the actions to be taken when each expression is matched
- Output : Contains C source code defining a procedure yylex that is *a* table-driven implementation of a DFA corresponding to the regular expressions of the input file, and that operates like a **getToken** procedure.

2.6 Use of Lex to generate a scanner automatically

- The format of a Lex input file

```
{ definitions }
%%
{ rules }
%%
{ auxiliary routines}
```

2.6 Use of Lex to generate a scanner automatically

1、The first section :definitions

The definition section occurs before the first %%.

1) any C code that must be inserted external to any function should appear in this section between the delimiters %{and %}, ***(Note the order of these characters!)***
2) names for regular expressions must also be defined in this section.

2.6 Use of Lex to generate a scanner automatically

2、 The second section : rules

These consist of a sequence of regular expressions followed by the C code that is to be executed when the corresponding regular expression is matched.

3、 The third section: auxiliary routines

Routines are called in the second section and not defined elsewhere.

2.6 Use of Lex to generate a scanner automatically

```
%{
/* a Lex program that changes all numbers from decimal to hexadecimal
 notation, printing a summary statistic to stdeer
*/
#include <stdlib.h>
#include <stdio.h>
int count = 0;
%}
digit  [0-9]
number {digit}+
%%
{ number }   { int n = atoi (yytext);
               printf(“%x”,  n);
               if  (n > 9) count ++; }
%%
main( )
{  yylex ( );
   fprintf(stdeer, “number of replacements = %d”, count);
   return 0;
}
```

2.6.1 Lex conventions for regular expression

1、Lex allows the matching of single characters, or strings of characters, simply by writing the characters in sequence.

2、Lex allows metacharacters to be matched as actual characters by surrounding the characters in quotes.

Quotes can also be written around characters that are not metacharacters, where they have no effect.

- **if** and **“if”** are same meaning.

- To match the character sequence **(*** have to write **\(*** or **" (* "**.
- A special meaning: **\n** matches a newline and **\t** matches a tab.

2.6.1 Lex conventions for regular expression

3、metacharacters : *, +, (,) , |, ?

for example :   **(aa|bb)(a|b)*c?** or **("aa"|"bb")("a"|"b")*"c"?**

4、**Lex** convention for character classes (sets of characters) is to write them between square brackets.

for example :[abcd]   **(aa|bb) [ab]*c?**

5、Ranges of characters : the expression [0-9] means in Lex any of the digits zero through nine.

6、A period (.) is a metacharacter that also represents a set of characters: it represents any character except a new-line.

2.6.1 Lex conventions for regular expression

- 7、Complementary sets—that is, sets that do ***not*** contain certain characters

  Using the carat ^ as the first character inside the brackets.

  For example: **[^0-9abc]** means any character that is not a digit and is not one of the letters *a, b,* or *c*.

2.6.1 Lex conventions for regular expression

8、**One curious feature** in Lex is that inside square brackets (representing a character class), most of the metacharacters lose their special status and do not need to be quoted.

- Written [-+] instead of ("+" | "-") . (but not **[+-]** because of the metacharacter use of - to express a range of characters).
- **[."?]** means any of the three characters period, quotation mark, or question mark.
- Some characters, however, are still metacharacters even inside the square brackets. we must precede the character by a backslash (quotes cannot be used as they have lost their metacharacter meaning). Thus, [\^ \ \] means either of the actual characters ^ or \.

2.6.1 Lex conventions for regular expression

10、The use of curly brackets to denote names of regular expressions. These names can be used in other regular expressions as long as there are no recursive references.

nat [0-9]+

```
signedNat  (+|-)?{nat}
```

2.6.1 Lex conventions for regular expression

metacharacter conventions in lex

Pattern	**Meaning**
a	the character a
“a”	the character a, even if a is a metacharacter
\a	the character a when a is a metacharacter
a*	zero or more repetitions of a
a+	one or more repetitions of a
a?	an optional a
a|b	a or b
(a)	a itself
[abc]	any of the characters a, b, or c .
[a-d]	any of the characters a. b, c or d
[^ab]	any character except a or b
.	any character except a newline
{xxx}	the regular expression that the name xxx represents

Homework of Chapter 2

- 2.1 Write regular expressions for the following character sets, or give reasons why no regular expression can be written:
  - 1) All strings of lowercase letters that begin and end in **a**.
  - 2) All strings of digits that contain no leading zeros.
  - 3) All strings of digits that represent even numbers.

- 2.8 Draw DFAs for each of the sets of characters of (1,2,3) in 2.1, or state why no DFA exists.

- 2.12 a. Use Thompson's construction to convert the regular expression $(a|b)^*a(a|b|\varepsilon)$ into an NFA
  - b. Convert the NFA of part (a) into a DFA using the subset construction.