

# RegExps & DFAs

CS 536

# Pre-class warm up

Write the regexp for Fortran real literals

An optional sign ('+' or '-')

An integer or:

1 or more digits followed by a '.' followed by 0 or more digits

or: A '.' followed by one or more digits

$( '+' | '-' | \epsilon )( \text{digit}^+ ( '.' | \epsilon ) | ( \text{digit}^* '.' \text{digit}^+ ) )$

# Last time

## Explored NFAs

for every NFA there is an equivalent DFA

epsilon edges add no expressive power

Introduce regular languages / expressions

# Today

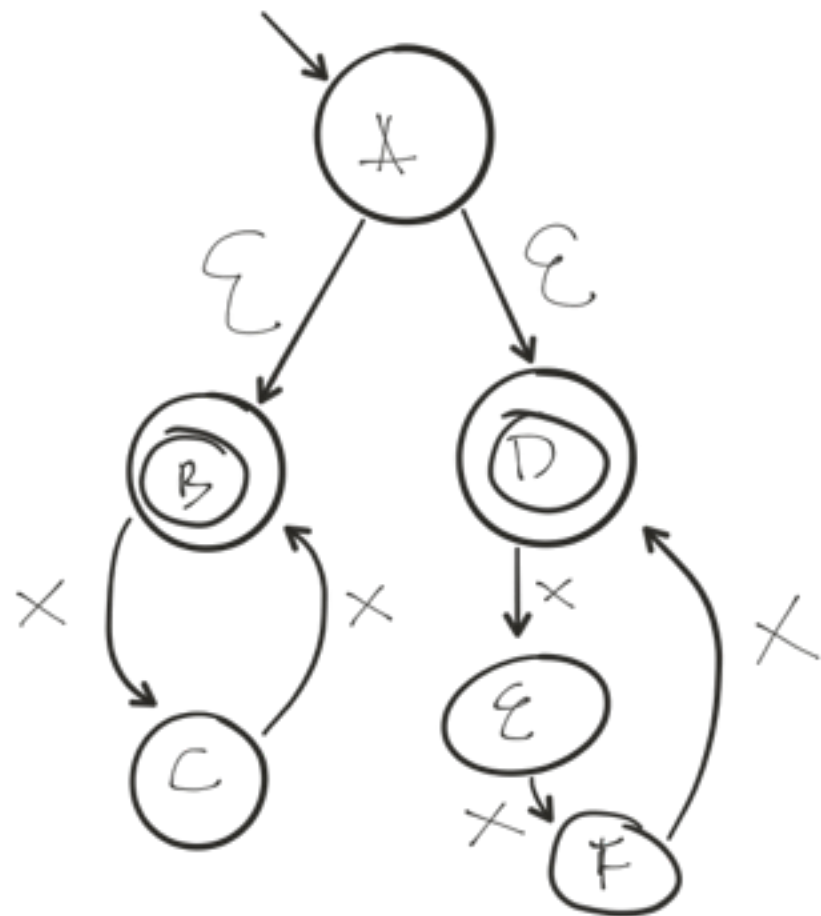
Convert regexps to DFAS

From language recognizers to tokenizers

# Regex to NFAs

Literals/epsilon correspond to simple DFAs

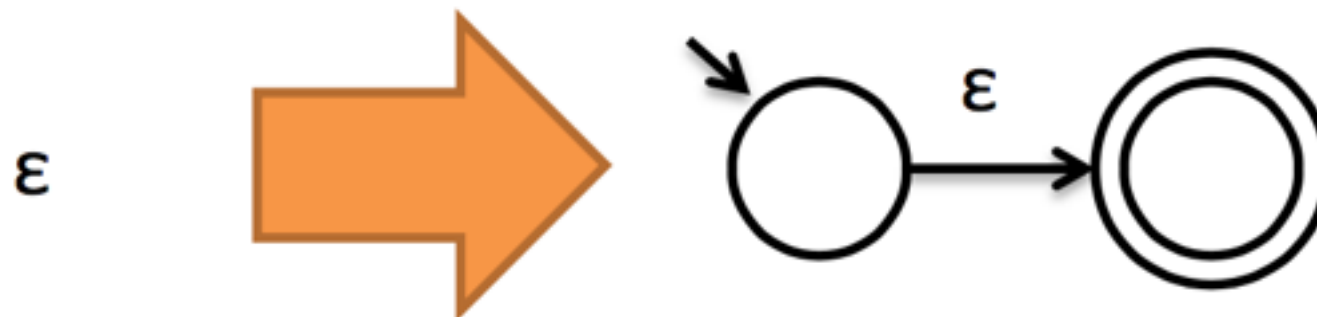
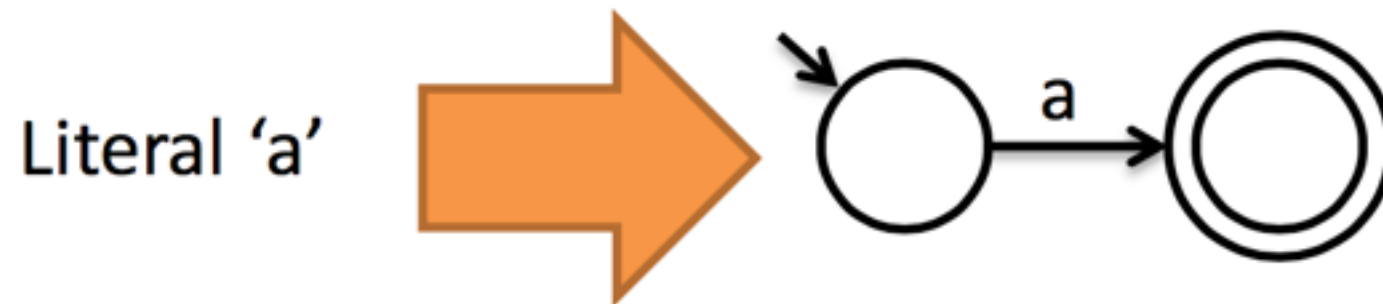
Operators correspond to methods of joining DFAs



$x^n$ , where  $n$  is even **or** divisible by 3

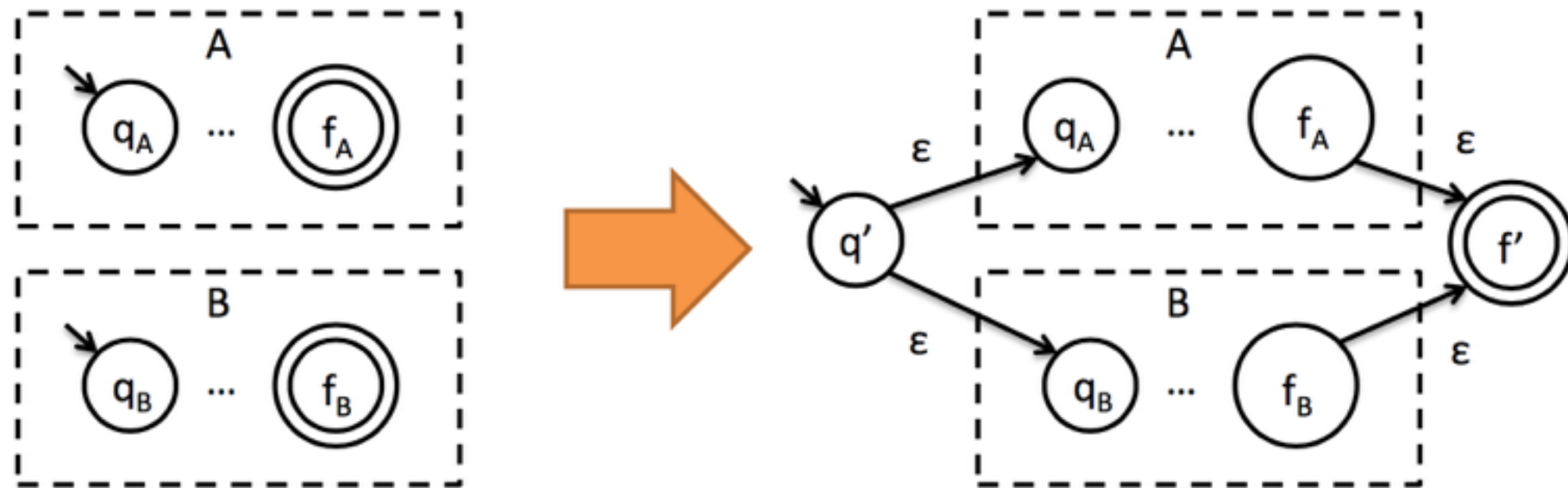
# Regex to NFA rules

## Rules for operands



# Regex to NFA rules

## Rules for alternation $A|B$



Make new start state  $q'$  and new final state  $f'$

Make original final states non-final

Add to  $\delta$ :

$q', \epsilon \rightarrow q_A$

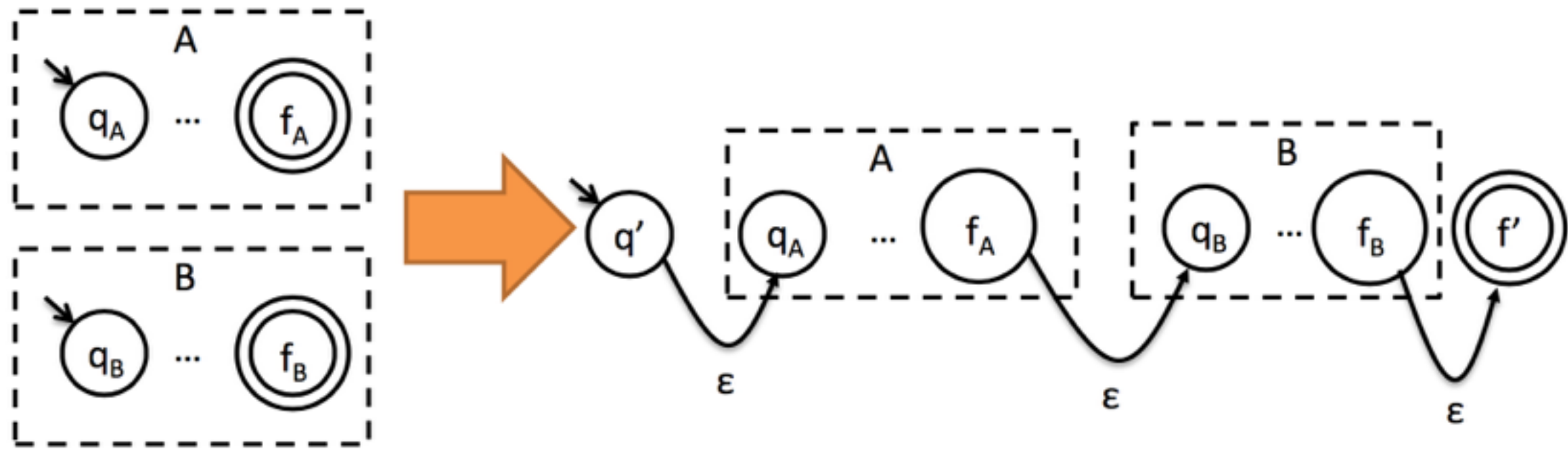
$q', \epsilon \rightarrow q_B$

$f_A, \epsilon \rightarrow f'$

$f_B, \epsilon \rightarrow f'$

# Regex to NFA rules

## Rule for catenation



Make new start state  $q'$  and new final state  $f'$

Make original final states non-final

Add to  $\delta$ :

$q', \epsilon \rightarrow q_A$

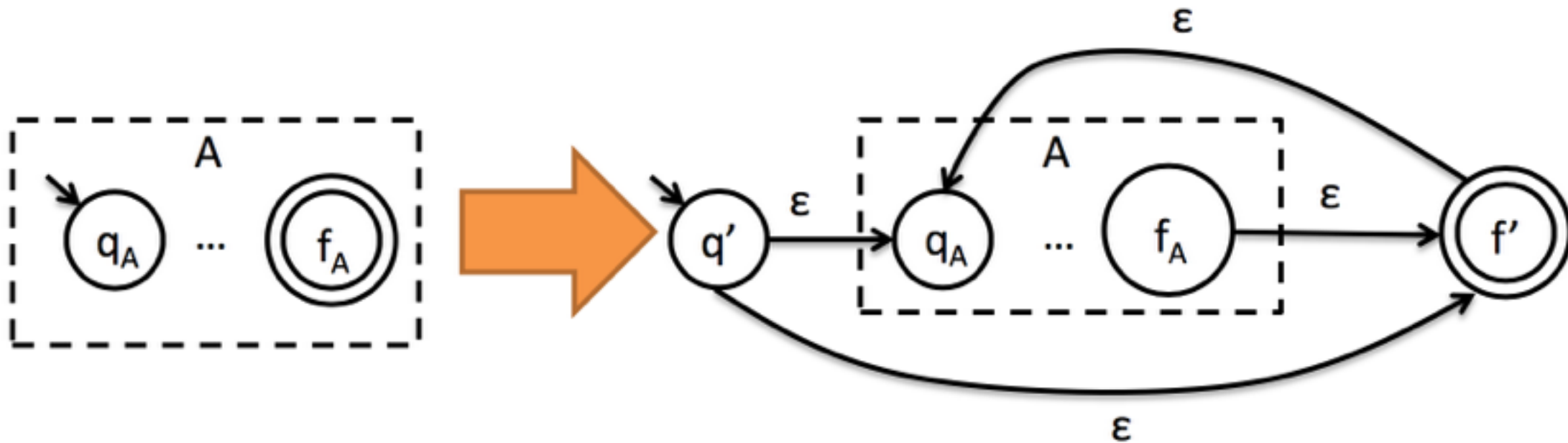
$f_A, \epsilon \rightarrow q_B$

$f_B, \epsilon \rightarrow f'$



# Regex to NFA rules

## Rule for catenation



Make new start state  $q'$  and new final state  $f'$

Make original final states non-final

Add to  $\delta$ :

$q', \epsilon \rightarrow q_A$

$q', \epsilon \rightarrow f'$

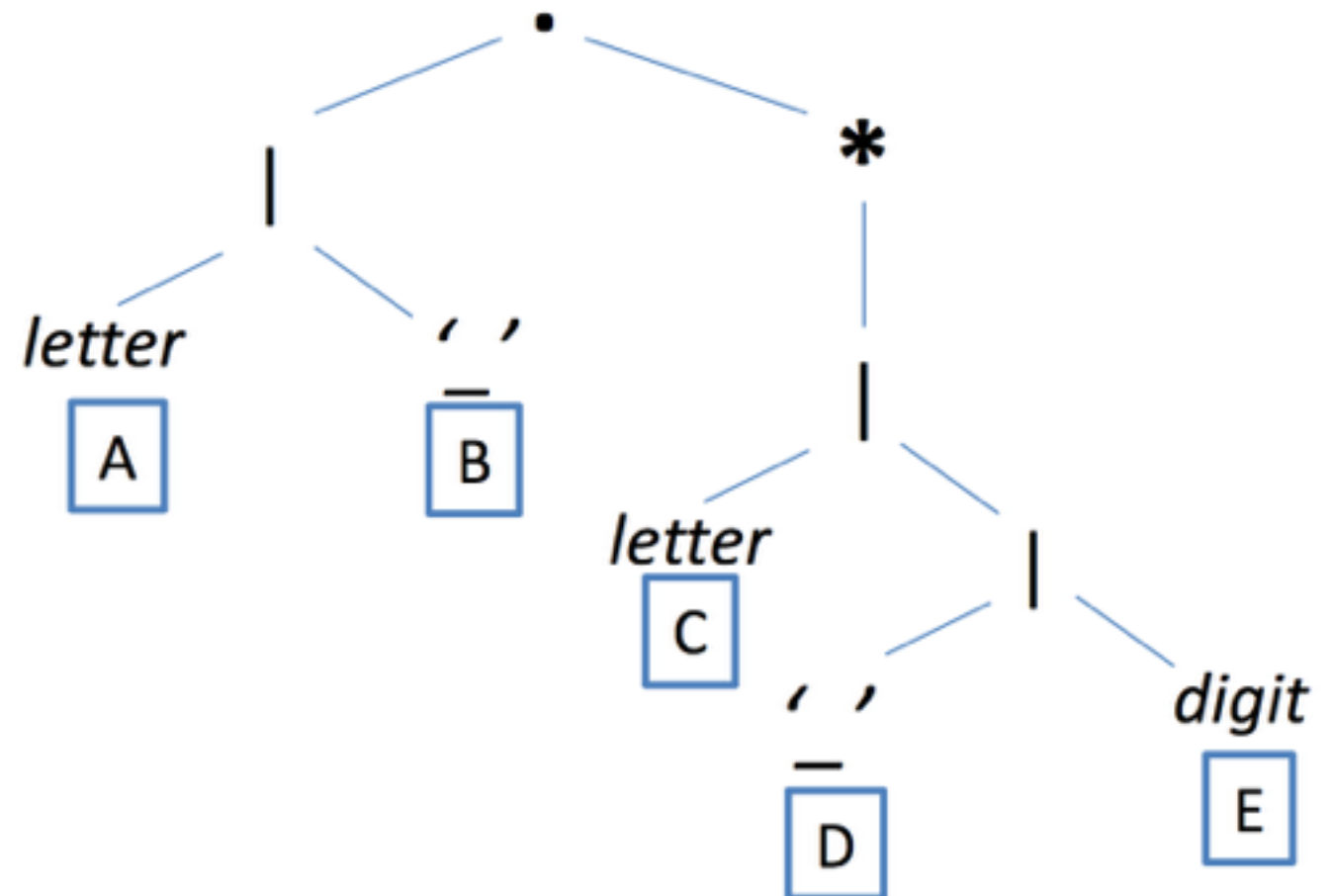
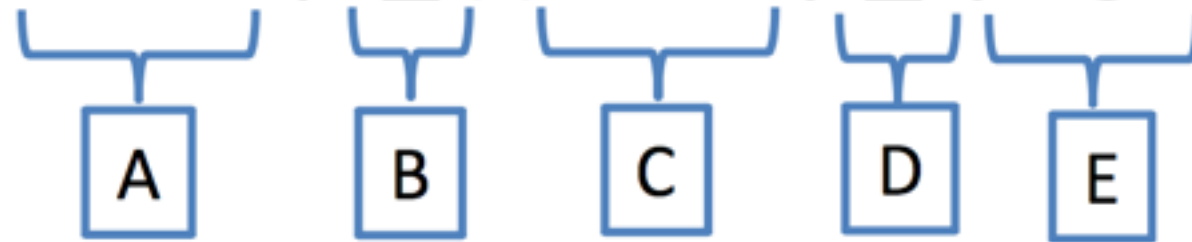
$f', \epsilon \rightarrow q_A$

# Regex operator precedence

Operator	Precedence	Analogous math operator
	low	addition
.	medium	multiplication
*	high	exponentiation

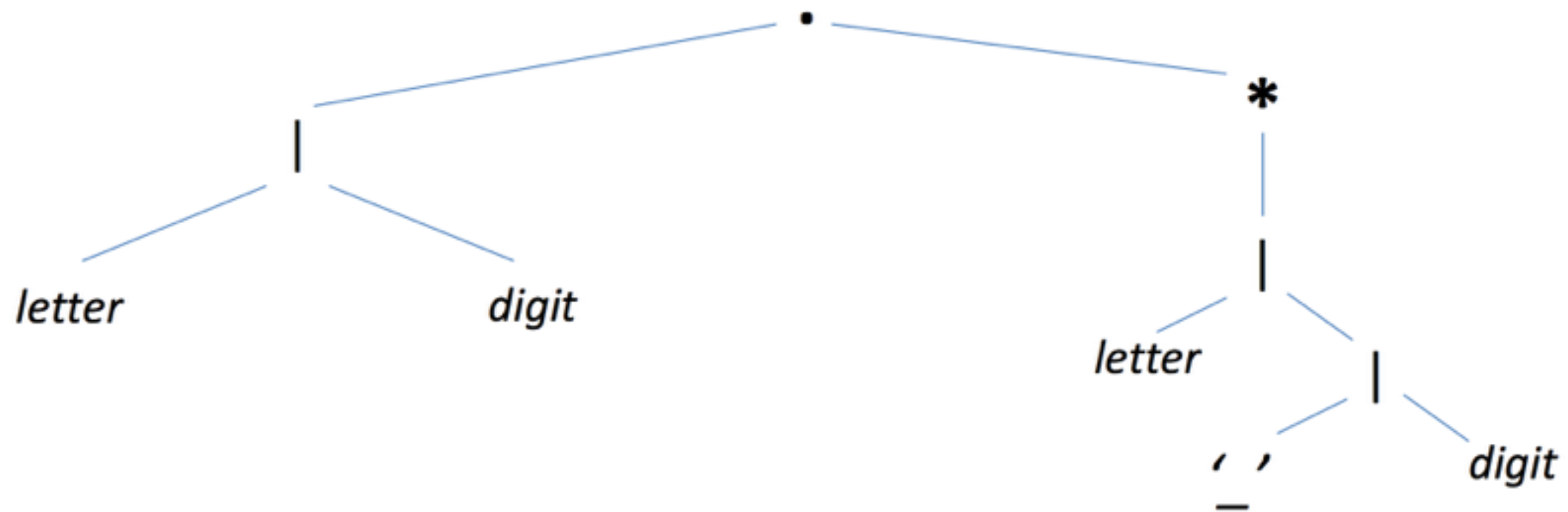
# Tree representation of a regexp

$(letter \mid \text{'\_'})(letter \mid \text{'\_'} \mid digit)^*$

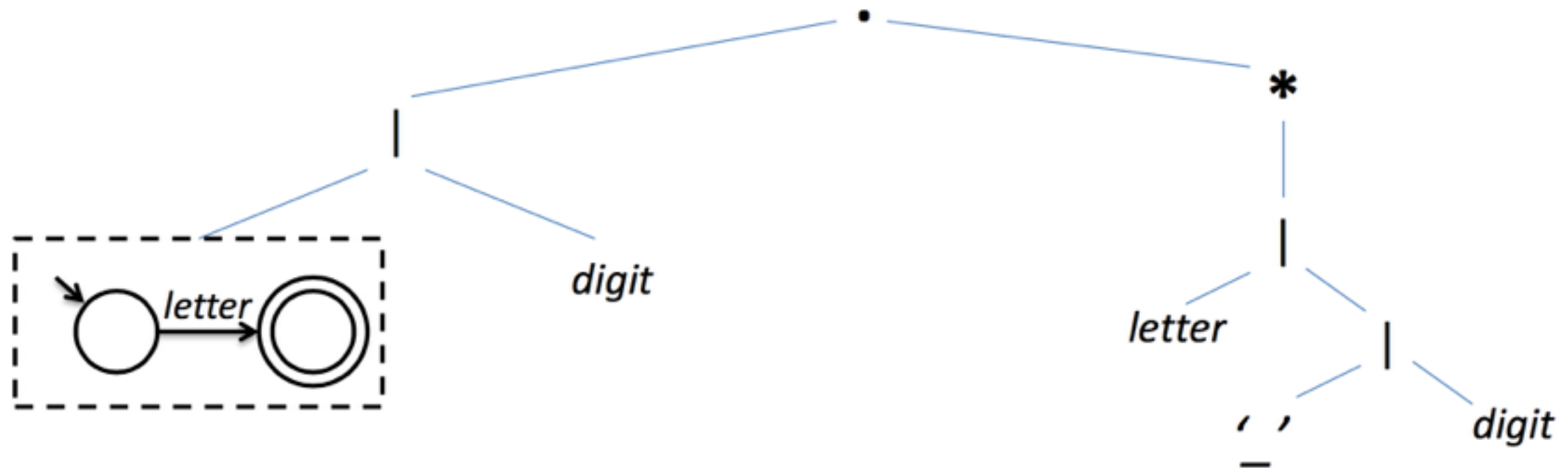


Operator	Precedence
$ $	low
$\cdot$	medium
$*$	high

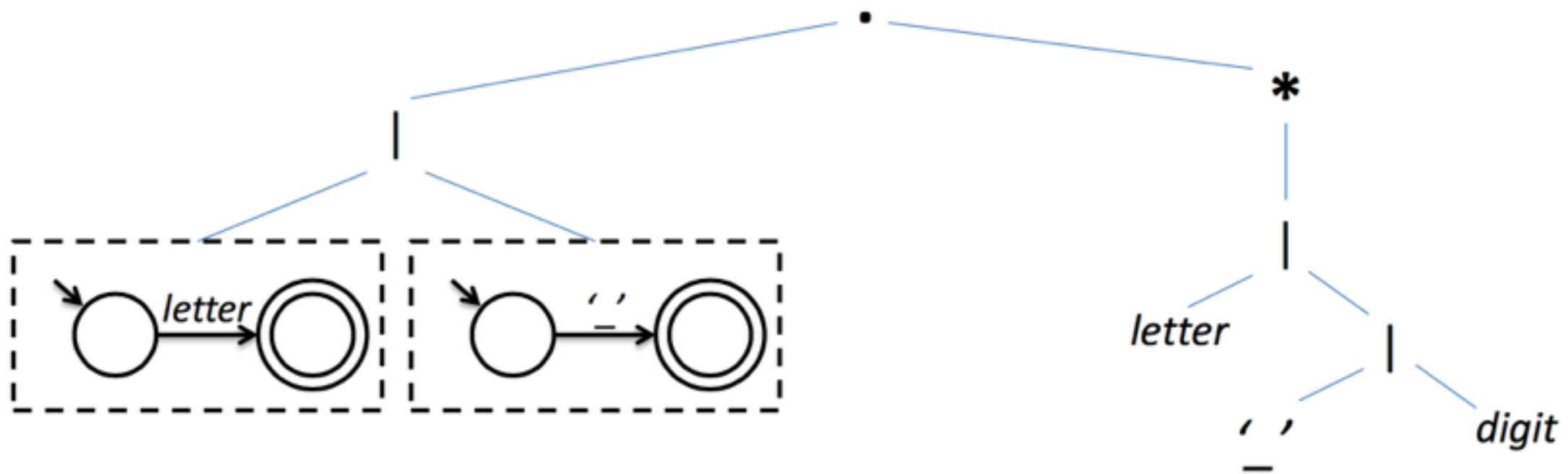
# Bottom-up conversion



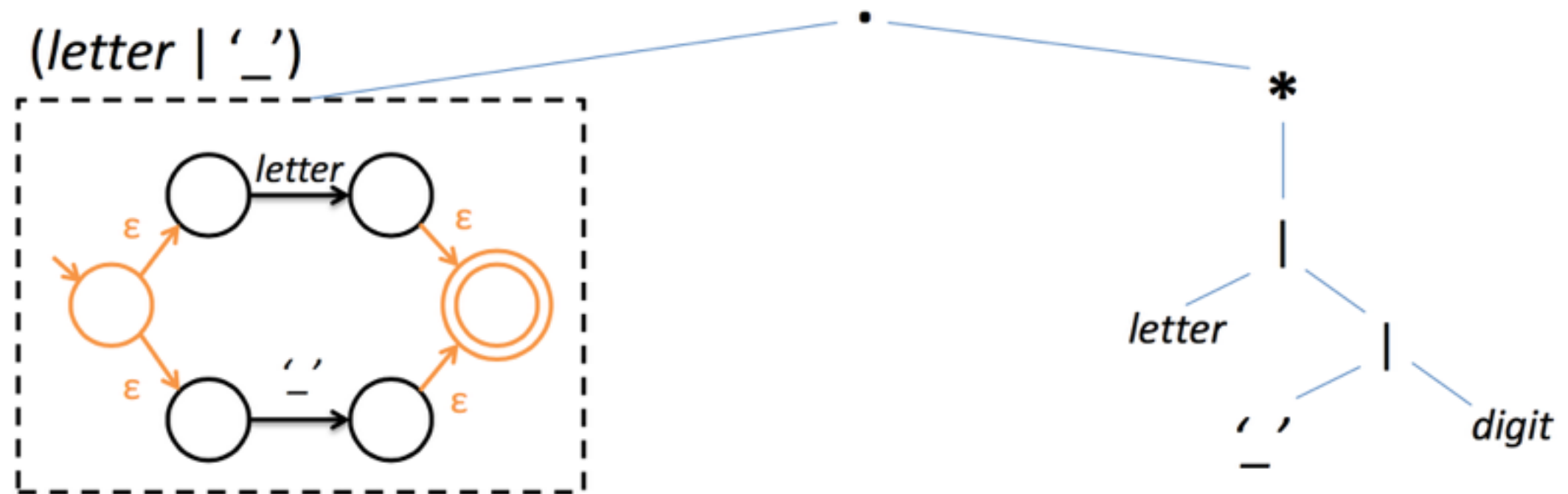
# Bottom-up conversion



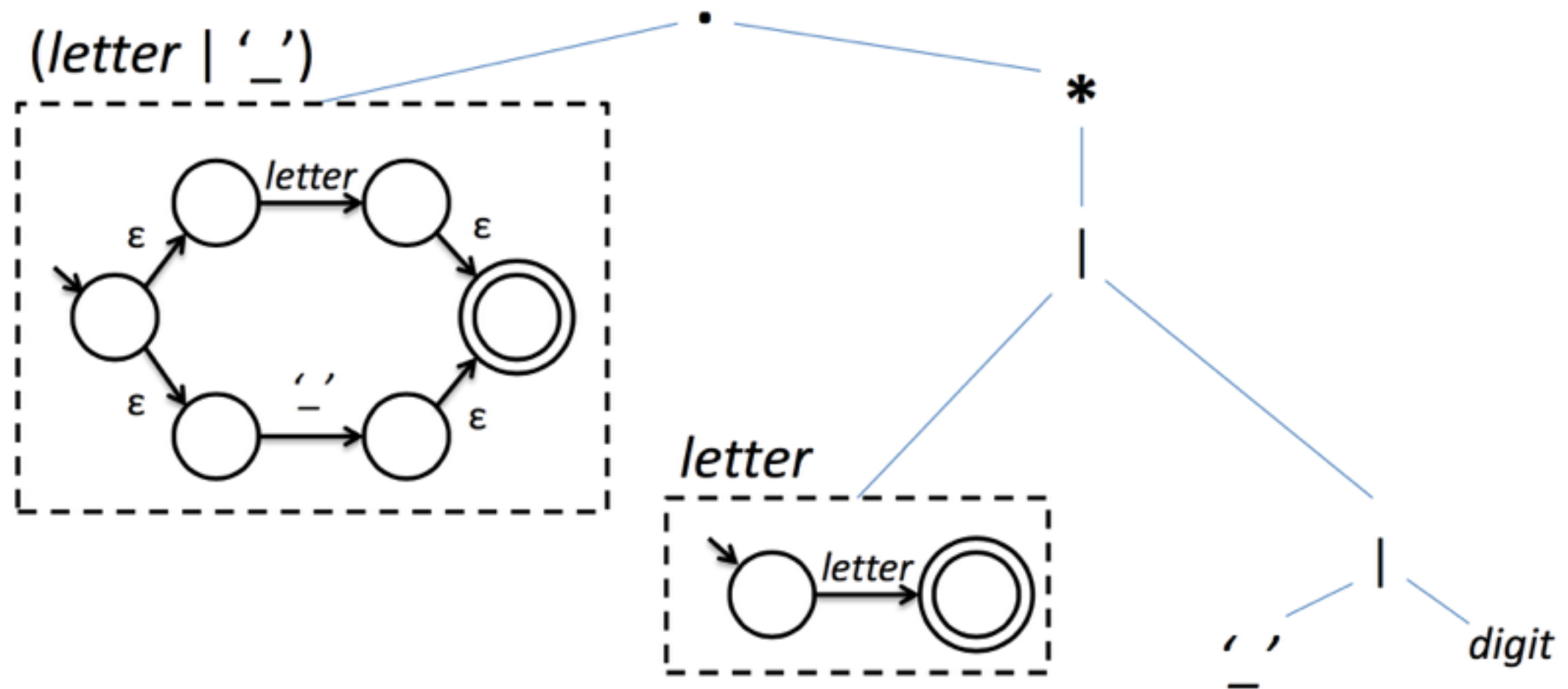
# Bottom-up conversion



# Bottom-up conversion

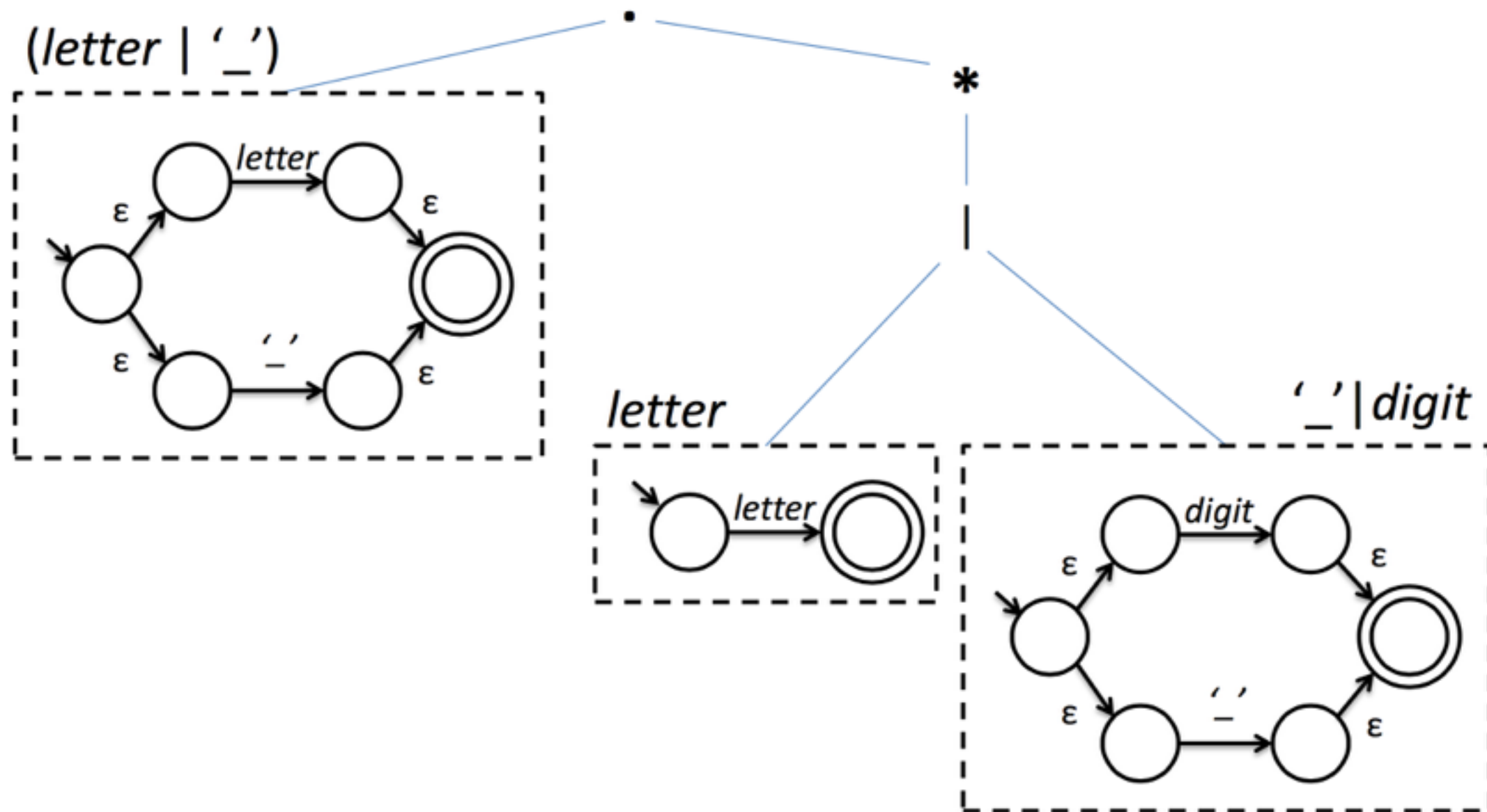


# Bottom-up conversion

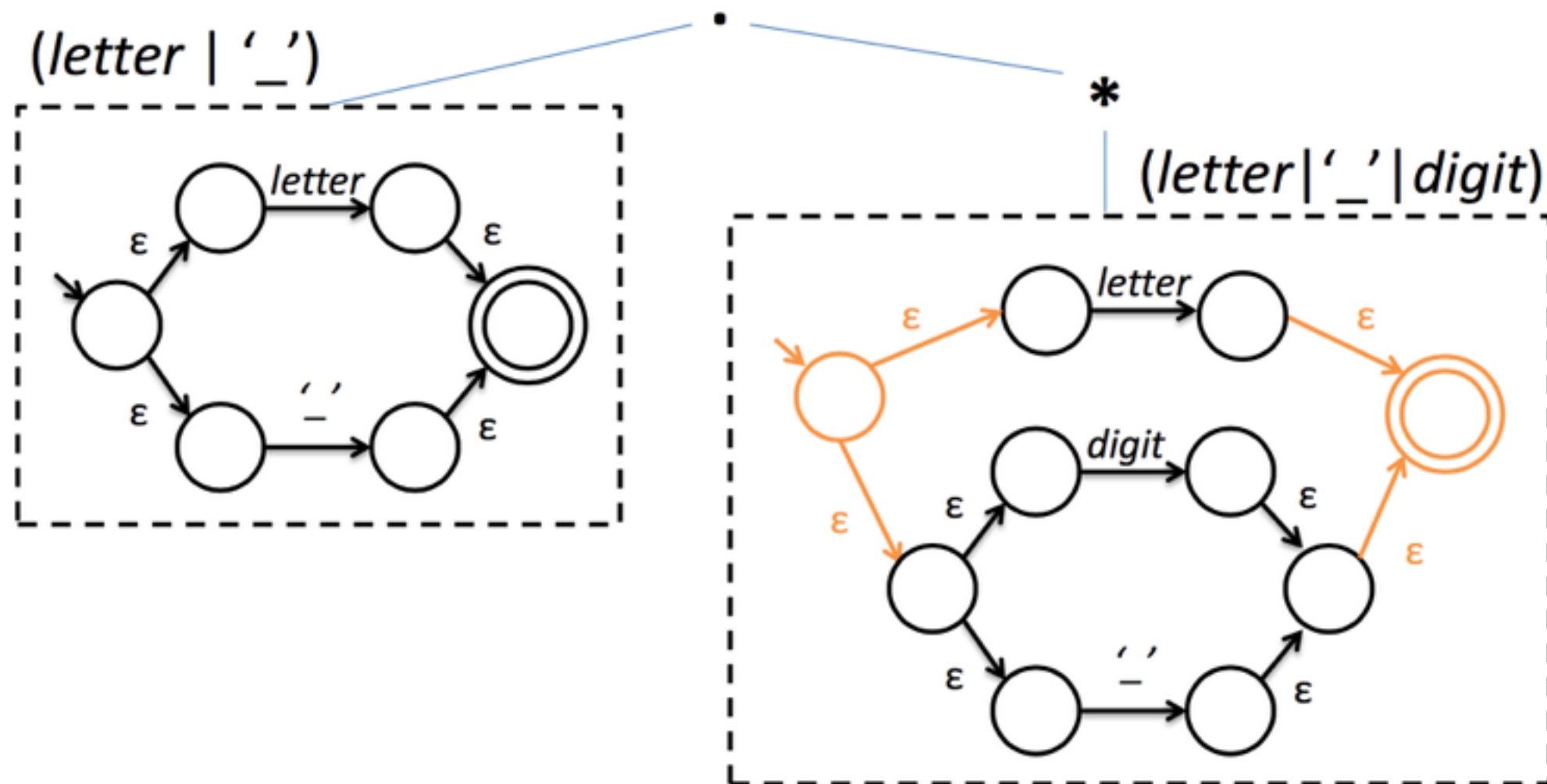




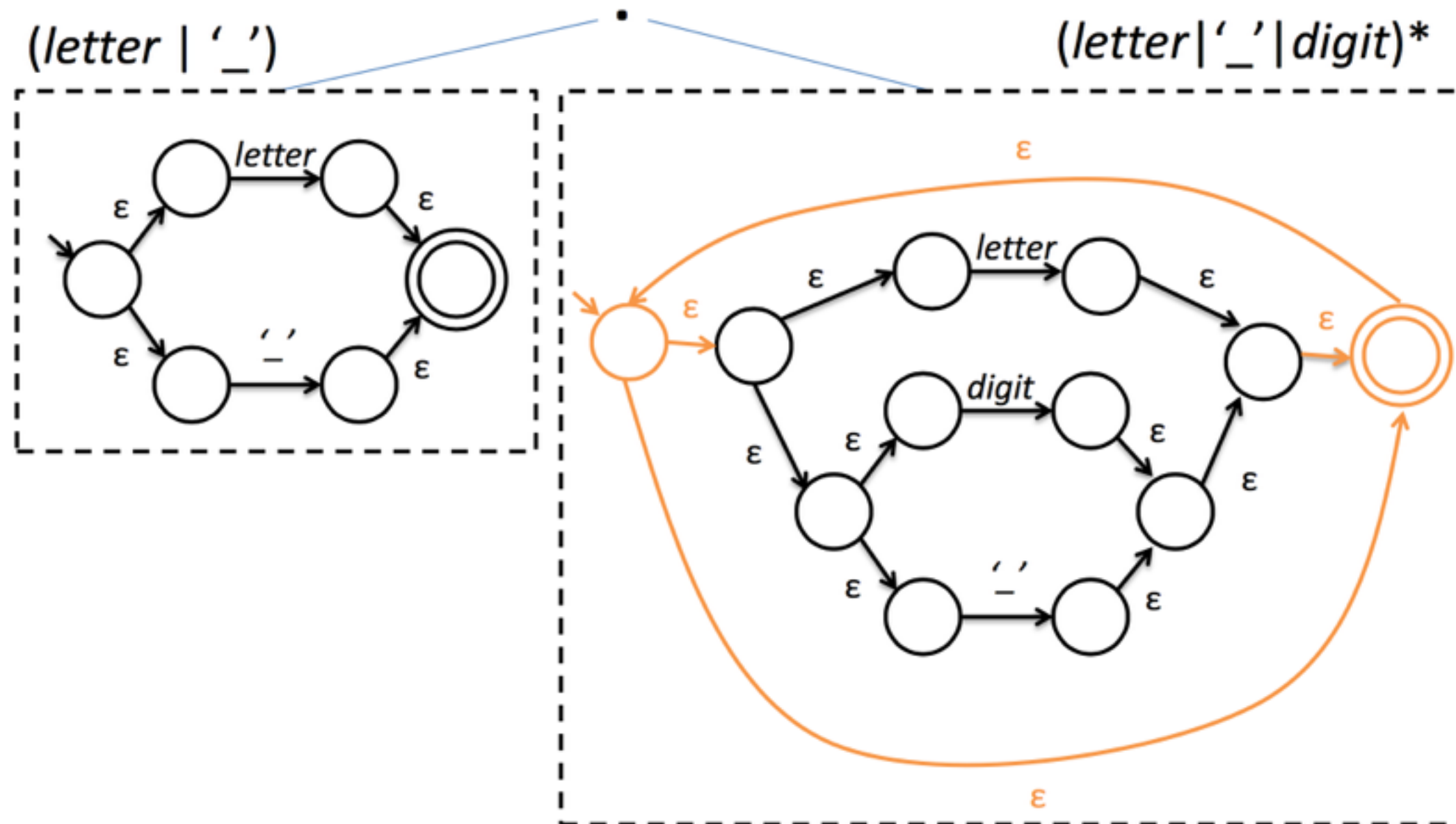
# Bottom-up conversion



# Bottom-up conversion

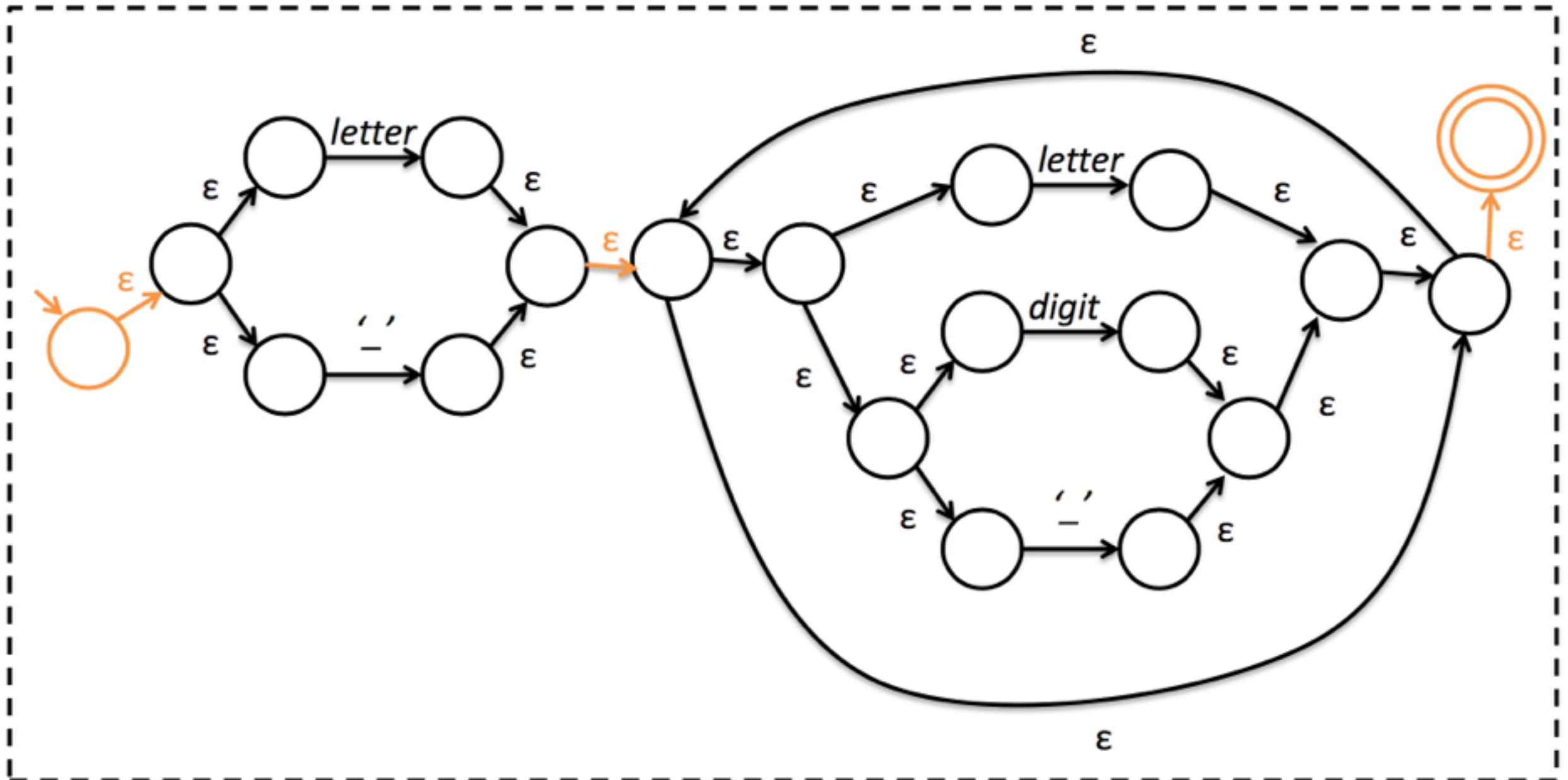


# Bottom-up conversion



# Bottom-up conversion

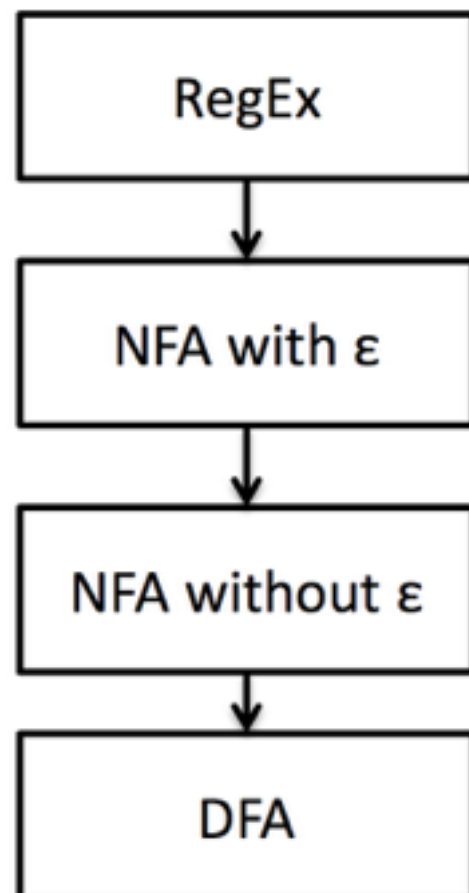
$(letter \mid ' \_ ')(letter \mid ' \_ ' \mid digit)^*$



# Regex to DFAs

We now have an NFA

We need to go to DFA

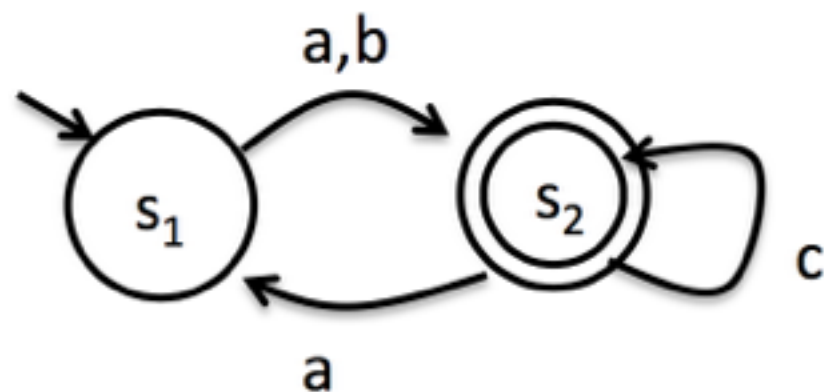


But what's so great about DFAs?

# Table-driven DFAs

Recall that  $\delta$  can be expressed as a table

This leads to a very efficient array representation



	a	b	c
s <sub>1</sub>	s <sub>2</sub>	s <sub>2</sub>	
s <sub>2</sub>	s <sub>1</sub>		s <sub>2</sub>

```
s = start state
while (more input){
    c = read char
    s = table[s][c]
}
if s is final, accept
```

# FSMs for tokenization

FSMs only check for language membership of a string

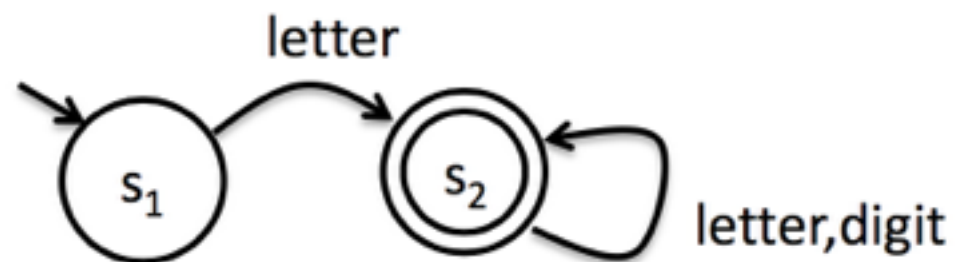
the scanner needs to recognize a stream of many different tokens using the longest match

the scanner needs to know what was matched

Idea: imbue states with actions that will fire when state is reached

# A first cut at actions

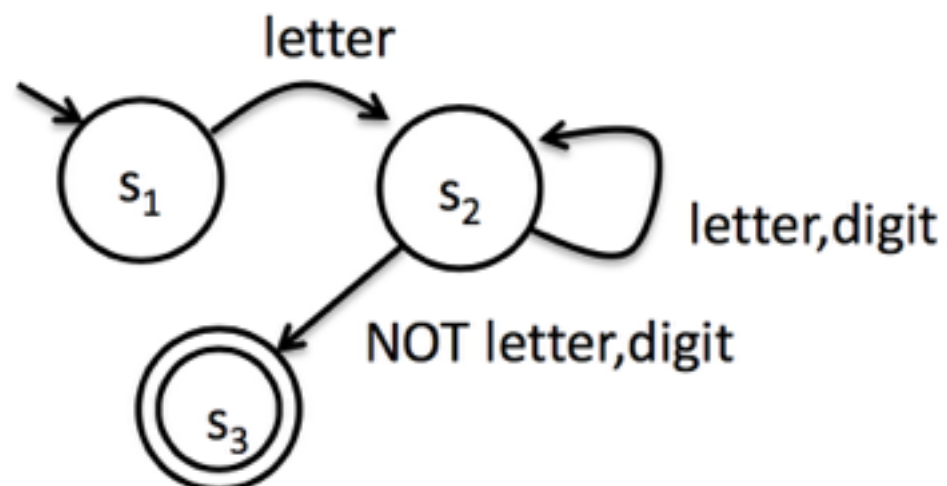
Consider the language of Pascal identifiers



State	Actions
$s_2$	return ID

BAD: not longest match

Accounting for longest matches



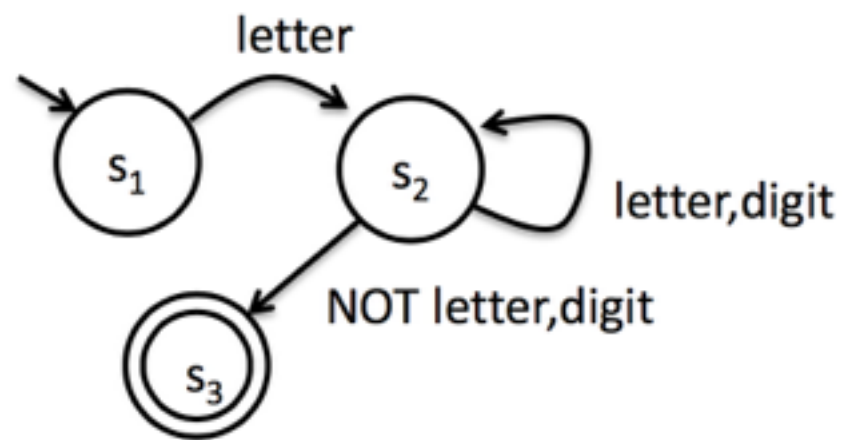
State	Actions
$s_3$	return ID

BAD: maybe we needed that  
NOT letter,digit character



# A second take at actions

Give our FSMs ability to put chars back



State	Actions
s <sub>3</sub>	Put 1 char back, return ID

Since we're allowing our FSM to peek at characters past the end of a valid token, it's also convenient to add an EOF symbol

# Our first scanner

Consider a language with two statements

assignments: `ID = expr`

increments: `ID += expr`

where **expr** is of the form

`ID + ID`

`ID ^ ID`

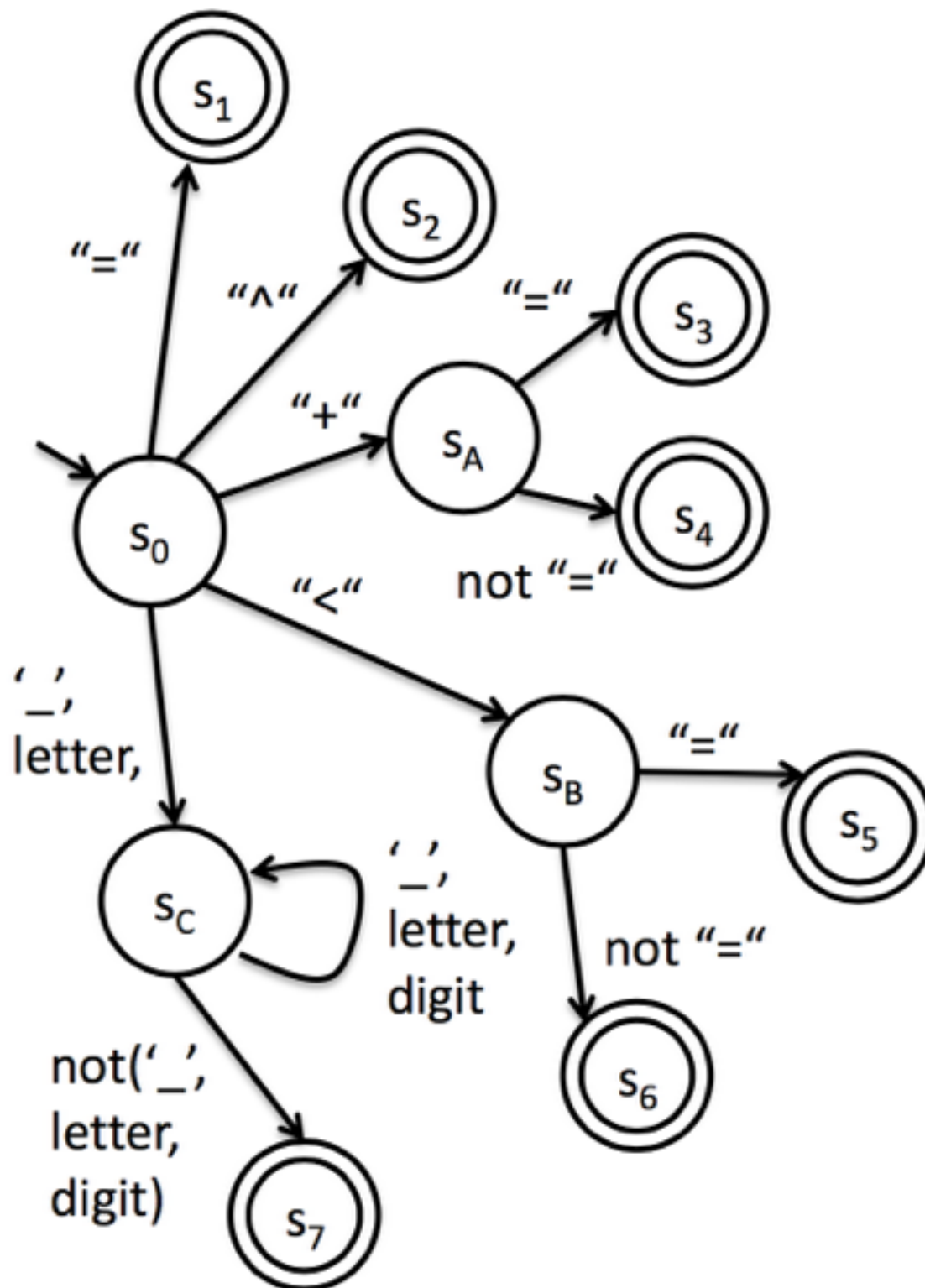
`ID < ID`

`ID <= ID`

Token name	Regular Expression
ASSIGN	"="
INC	"+="
PLUS	"+"
EXP	"^"
LT	"<"
LEQ	"<="
ID	<i>(letter _)(letter digit _)*</i>

Identifiers ID follow C conventions

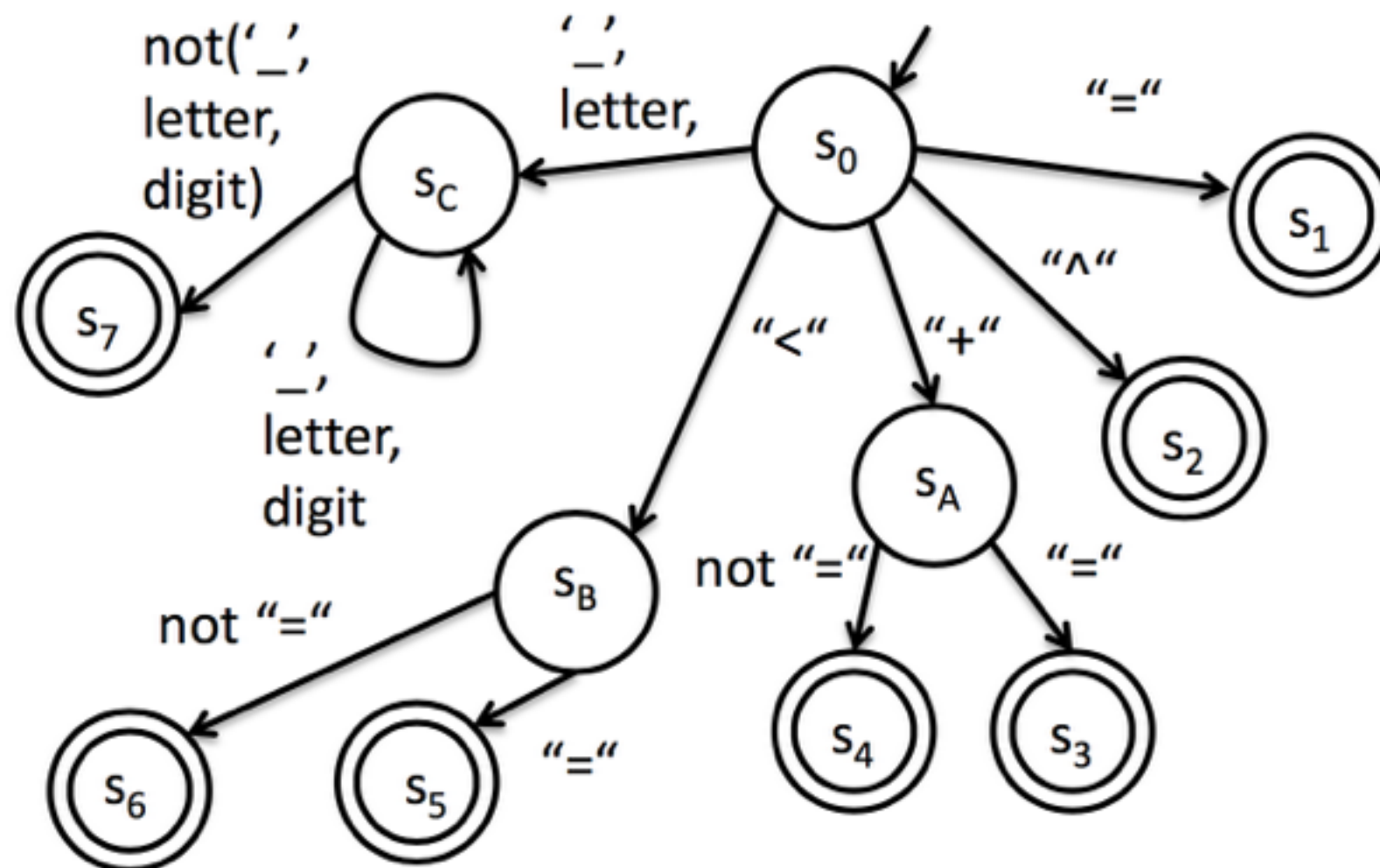
# Combined DFA



Token name	Regular Expression
ASSIGN	"=
INC	"+=
PLUS	"+"
EXP	"^"
LT	"<"
LEQ	"<="
ID	<i>(letter _)(letter digit _)*</i>

State	Action
S1	return ASSIGN
S2	return EXP
S3	return INC
S4	put back 1 char, return PLUS
S5	Return LEQ
S6	put back 1 char, return LT
S7	put back 1 char, return ID

	=	+	^	<	_	letter	digit	EOF	none
S <sub>0</sub>	Ret ASSIGN	S <sub>A</sub>	Ret EXP	B	C	C		Ret EOF	
S <sub>A</sub>	Ret INC	Back 1, Ret PLUS	Back 1, Ret PLUS	Back 1, Ret PLUS	Back 1, Ret PLUS	Back 1, Ret PLUS	Back 1, Ret PLUS	Back 1, Ret PLUS	Back 1, Ret PLUS
S <sub>B</sub>	Ret LEQ	Back 1, Ret LT	Back 1, Ret LT	Back 1, Ret LT	Back 1, Ret LT	Back 1, Ret LT	Back 1, Ret LT	Back 1, Ret LT	Back 1, Ret LT
S <sub>C</sub>	Back 1, Ret ID	Back 1 Ret ID	Back 1, Ret ID	Back 1, Ret ID	S <sub>C</sub>	S <sub>C</sub>	S <sub>C</sub>	Back 1, Ret ID	Back 1, Ret ID



	=	+	^	<	_	letter	digit	EOF	none
$S_0$	Ret ASSIGN	$S_A$	Ret EXP	B	C	C		Ret EOF	
$S_A$	Ret INC	Back 1, Ret PLUS	Back 1, Ret PLUS	Back 1, Ret PLUS	Back 1, Ret PLUS	Back 1, Ret PLUS	Back 1, Ret PLUS	Back 1, Ret PLUS	Back 1, Ret PLUS
$S_B$	Ret LEQ	Back 1, Ret LT	Back1, Ret LT	Back 1, Ret LT	Back 1, Ret LT	Back 1, Ret LT	Back 1, Ret LT	Back 1, Ret LT	Back 1, Ret LT
$S_C$	Back 1, Ret ID	Back 1 Ret ID	Back 1, Ret ID	Back 1, Ret ID	$S_C$	$S_C$	$S_C$	Back 1, Ret ID	Back 1, Ret ID

```

do{
    read char
    perform action / update state
    if (action was to return a token){
        start again in start state
    }
} (while not EOF or stuck);

```

# Lexical analyzer generators

aka scanner generators

The transformation from regexp to scanner is formally defined

Can write tools to synthesize a lecture automatically

Lex: unix scanner generator

Flex: fast lex

JLex: Java version of Lex

# JLex

Declarative specification

tell it what you want scanned, it will figure out the rest

**Input:** set of regexps + associated actions

`xyz.jlex` file

**Output:** Java source code for a scanner

`xyz.jlex.java` source code of scanner

# jlex format

3 sections separated by %%

user code section

directives

regular expressions + actions



**//User Code Section (uninterpreted java code)**

%%

**//Directives Section**

DIGIT = [0-9]  
LETTER = [a-zA-Z]  
WHITESPACE = [\040\t\n] } Macro definitions

%state SPECIALINTSTATE — State declaration

//Configure for use with java CUP (Parser generator)

%implements java\_cup.runtime.Scanner

%function next\_token

%type java\_cup.runtime.Symbol

//End of file behavior

%eofval{

System.out.println("All done");

return null;

%eofval}

//Turn on line counting

%line

%%

**//Regular Expression rules**

# Rules section

Format is `<regex>{code}` where `regex` is a regular expression for a single token

can use macros from the directive sections in `regex`, surround with curly braces

## Conventions

chars represent themselves (except special characters)

chars inside “” represent themselves (except \)

## Regex operators

| \* + ? () .

## Character class operators

- range

^ not

\ escape

# jflex rules example

```
"="      { System.out.println(yyline + 1 + ": ASSIGN"); }
"+"      { System.out.println(yyline + 1 + ": PLUS"); }
"^"      { System.out.println(yyline + 1 + ": EXP"); }
"<"      { System.out.println(yyline + 1 + ": LT"); }
"+="     { System.out.println(yyline + 1 + ": INC"); }
"<="     { System.out.println(yyline + 1 + ": LEQ"); }
{WHITESPACE} { }
({LETTER}|"_")({DIGIT}|{LETTER}|"_")* {
    System.out.println(yyline+1 + ": ID " + yytext());}
.      { System.out.println(yyline + 1 + ": badchar"); }
```