# Announcements

Contents

1. Regex to DFA: Regex -> NFA w/ epsilon -> NFA w/o epsilon -> DFA.
2. Use FSM in tokenization.

P2 will be assigned today or tomorrow

# 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

```
('+'|'-'|ε)(digit+('.'|ε))|
                (digit*'.'digit+))
```

You do not make any changes to the performace by writing the regex differently. All are compiled into NFAs in the end.

# 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

# Regexp to NFAs

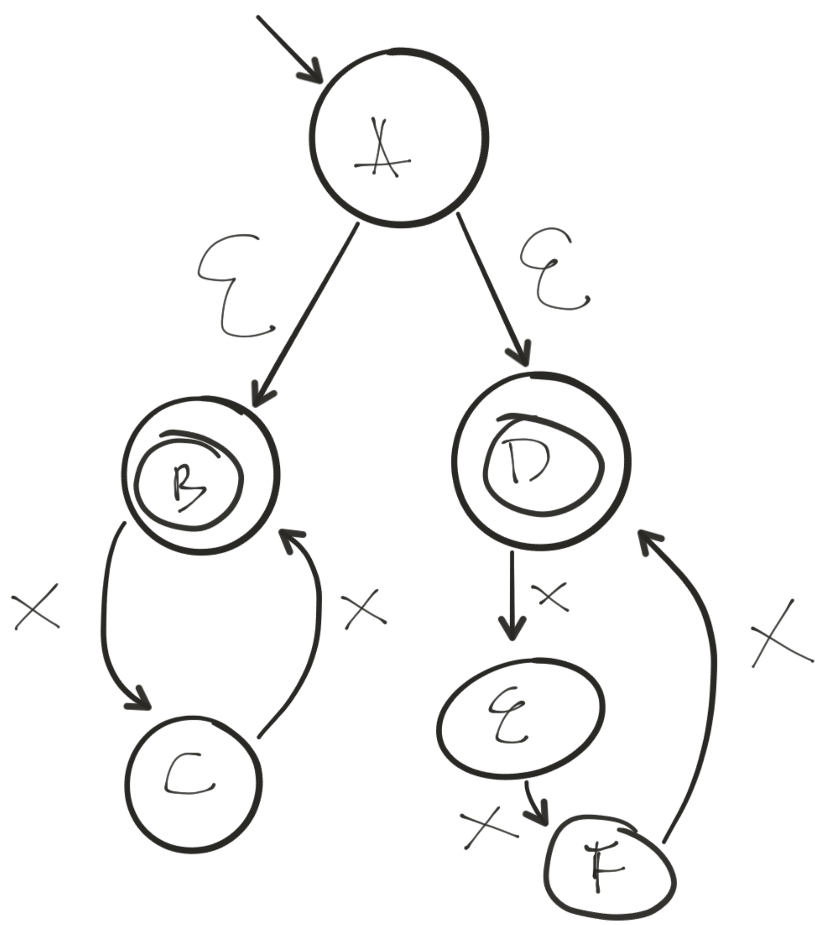
Theorem:
For every regular expression, there's a FSM that defines the same language, and vice versa.

Regexp can be considered as: operand + operator.

Literals/epsilon correspond to simple DFAs

Operators correspond to methods of joining DFAs

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



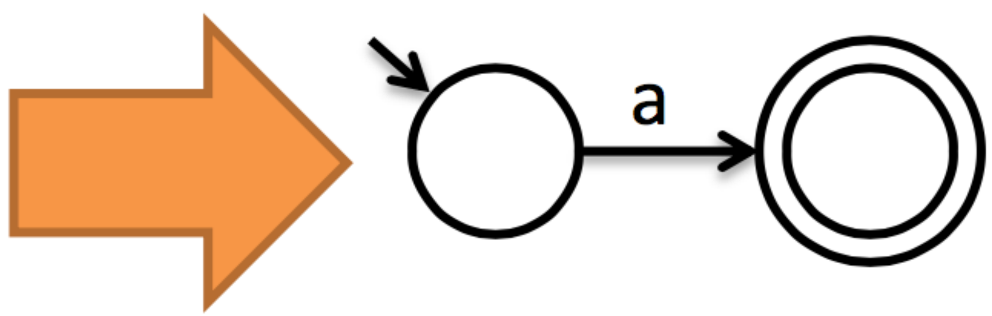
# Regexp to NFA rules

```
Construction rule for building an NFA for regexp R:
                    +------------+
-> InitState -> |              | -> EndState
                    +------------+
                      R
```

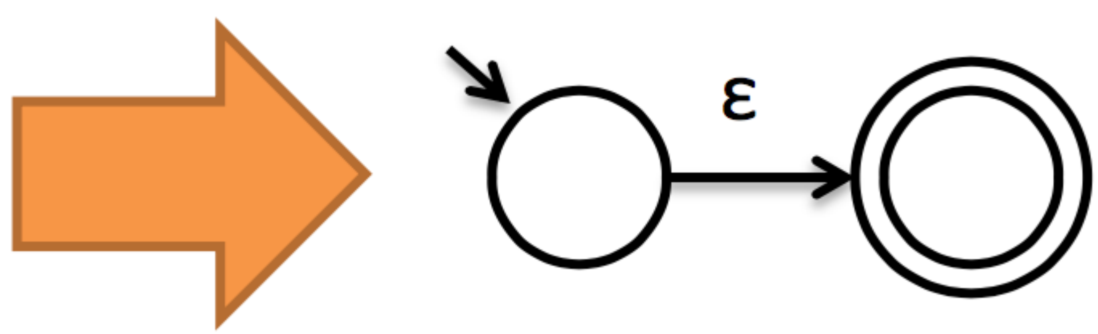
We want a unique InitState and a unique EndState. As long as a string starts from InitState and ends at EndState, we know that the string has been accepted.

## Rules for operands

Literal ‘a’



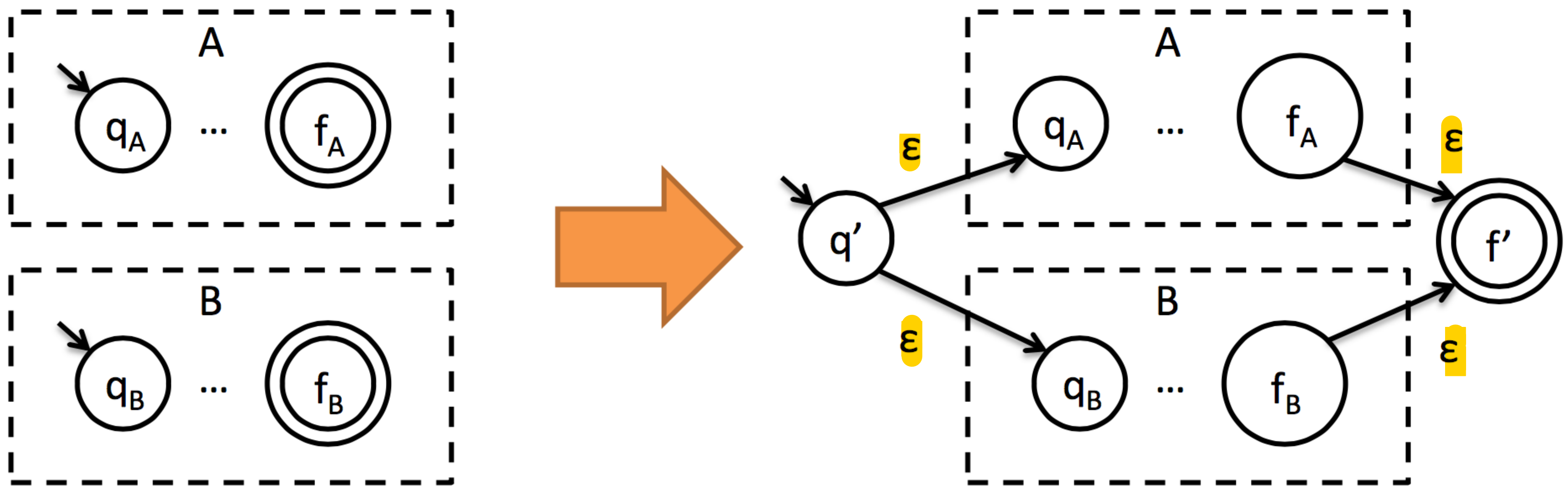

ε



# Regexp to NFA rules

## Rules for alternation A|B

Assumption: we already have the NFA for expression A and B. How to build A | B?




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

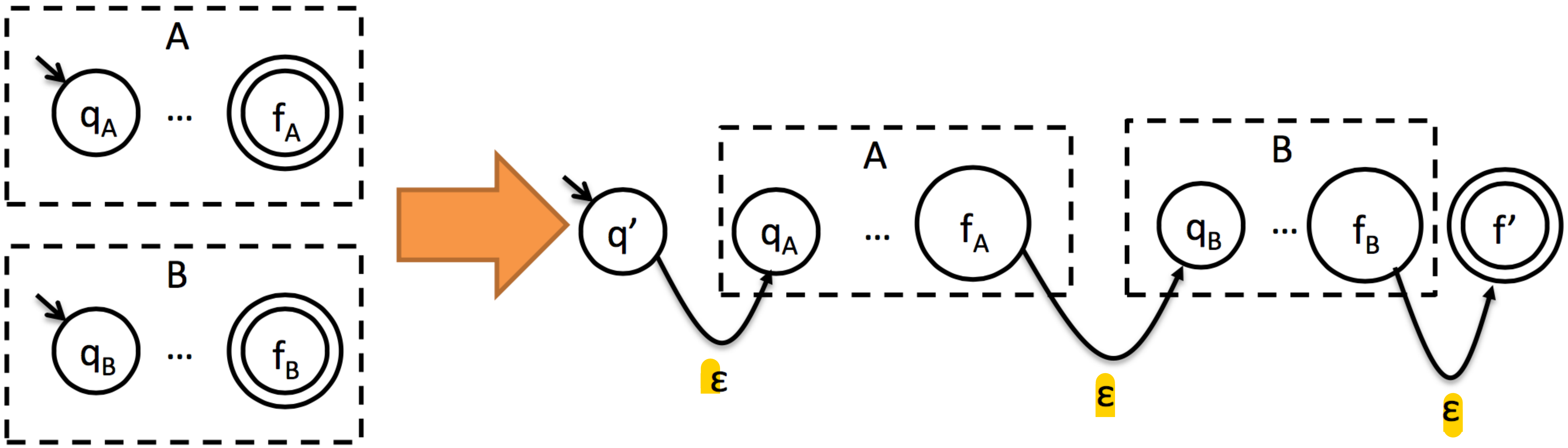
Make original final states non-final

Add to $\delta$:

$q', \varepsilon \rightarrow q_A$
$q', \varepsilon \rightarrow q_B$
$F_a, \varepsilon \rightarrow f'$
$F_b, \varepsilon \rightarrow f'$

# Regexp to NFA rules

## Rule for catenation A.B




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

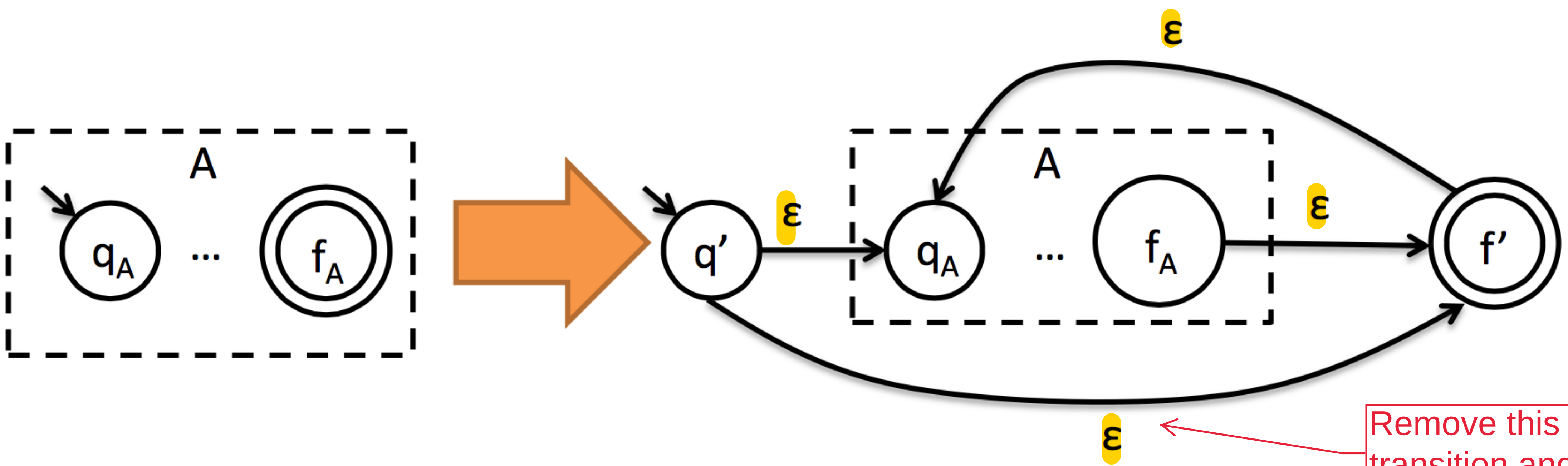
Make original final states non-final

Add to $\delta$:

$q', \varepsilon \rightarrow q_A$
$f_A, \varepsilon \rightarrow q_B$
$f_b, \varepsilon \rightarrow f'$

# Regexp to NFA rules

Rule for iteration A*




Remove this epsilon-transition and you get the rule for A+.

However, this is not needed as you can compile all syntactic sugar like A+ into (AA*) during compilation.

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

Make original final states non-final

Add to $\delta$:

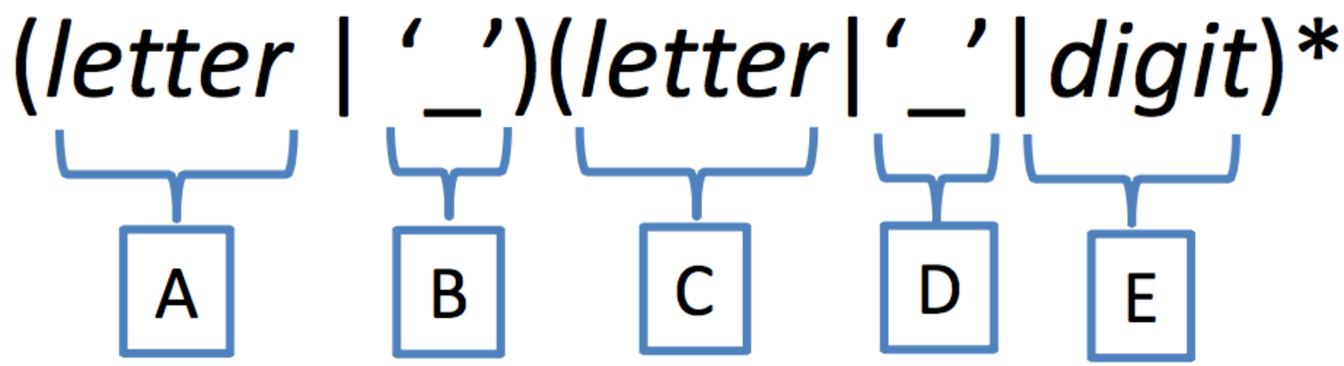
$q', \varepsilon \rightarrow q_A$
$q', \varepsilon \rightarrow f'$
$f', \varepsilon \rightarrow q_A$

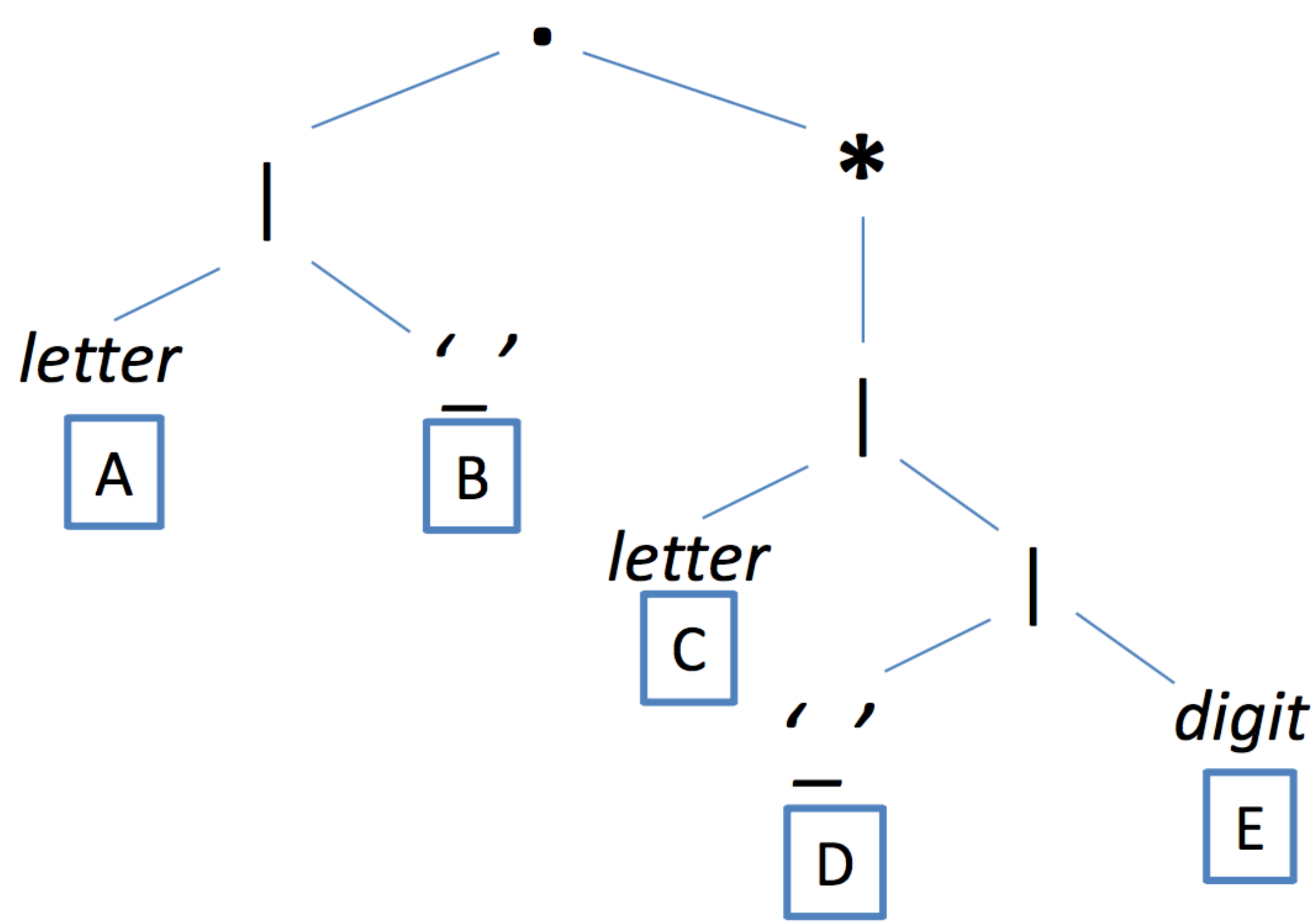
# Regexp operator precedence

<table>
<tr><th>Operator</th><th>Precedence</th><th>Analogous math operator</th></tr>
<tr><td>|</td><td>low</td><td>addition</td></tr>
<tr><td>.</td><td>medium</td><td>multiplication</td></tr>
<tr><td>*</td><td>high</td><td>exponentiation</td></tr>
</table>

# Tree representation of a regexp

Parenthesis is not needed for syntax tree. As the precedence is already captured.

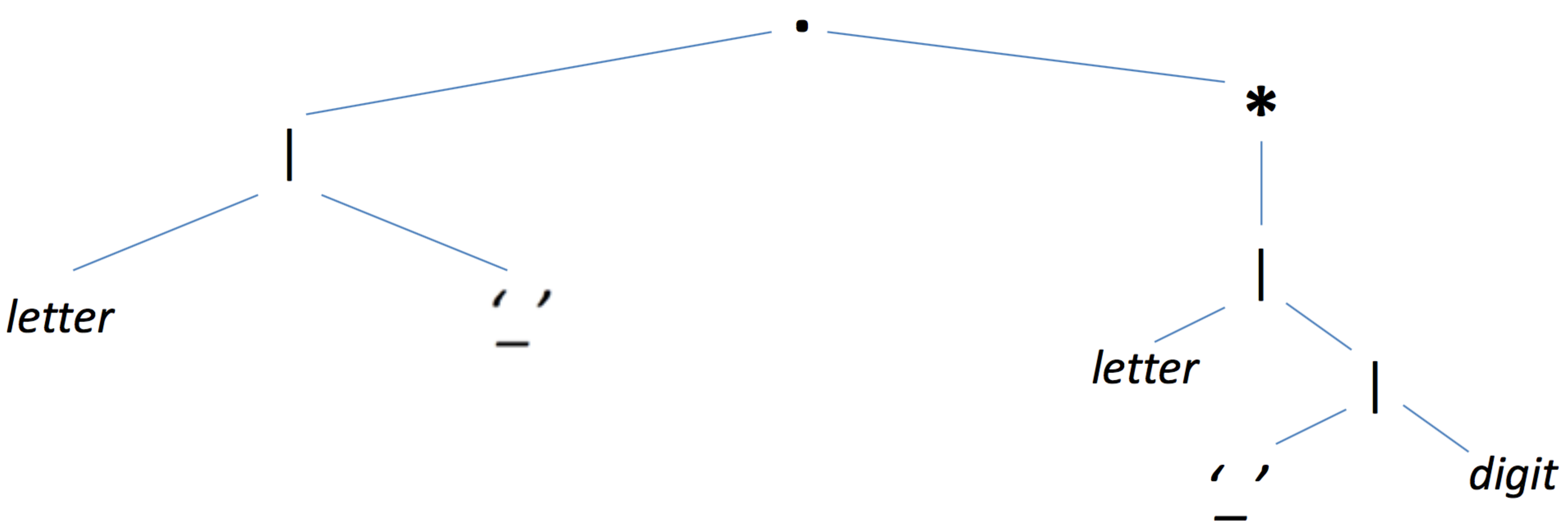


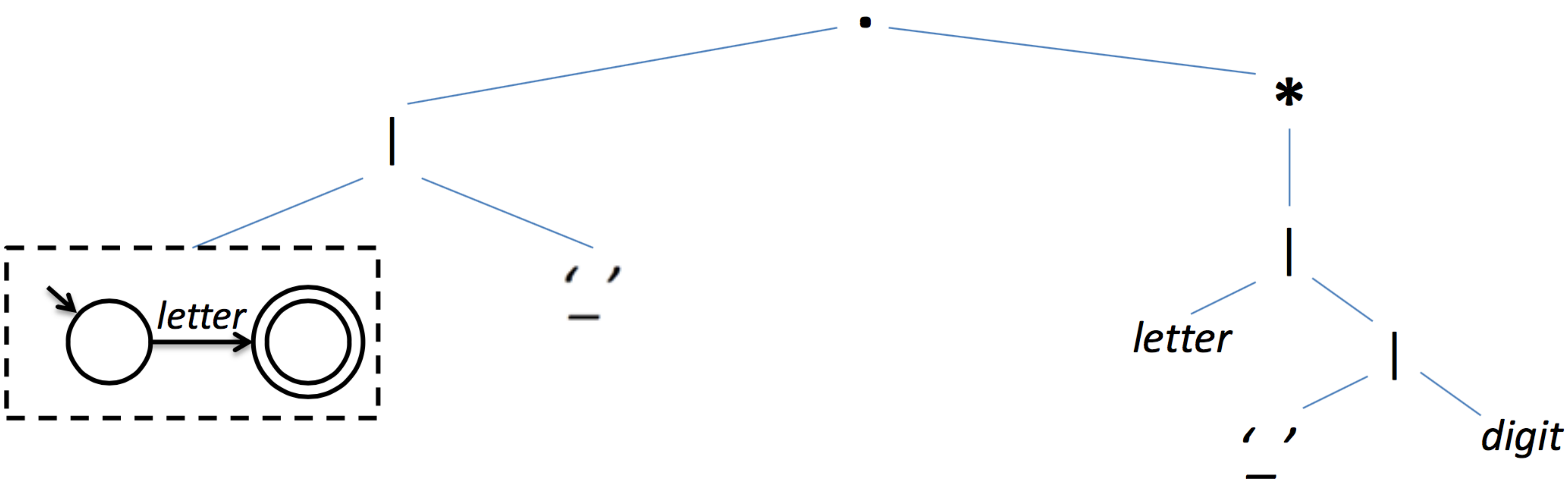

| Operator | Precedence |
|---|---|
| \| | low |
| . | medium |
| * | high |

# Bottom-up conversion

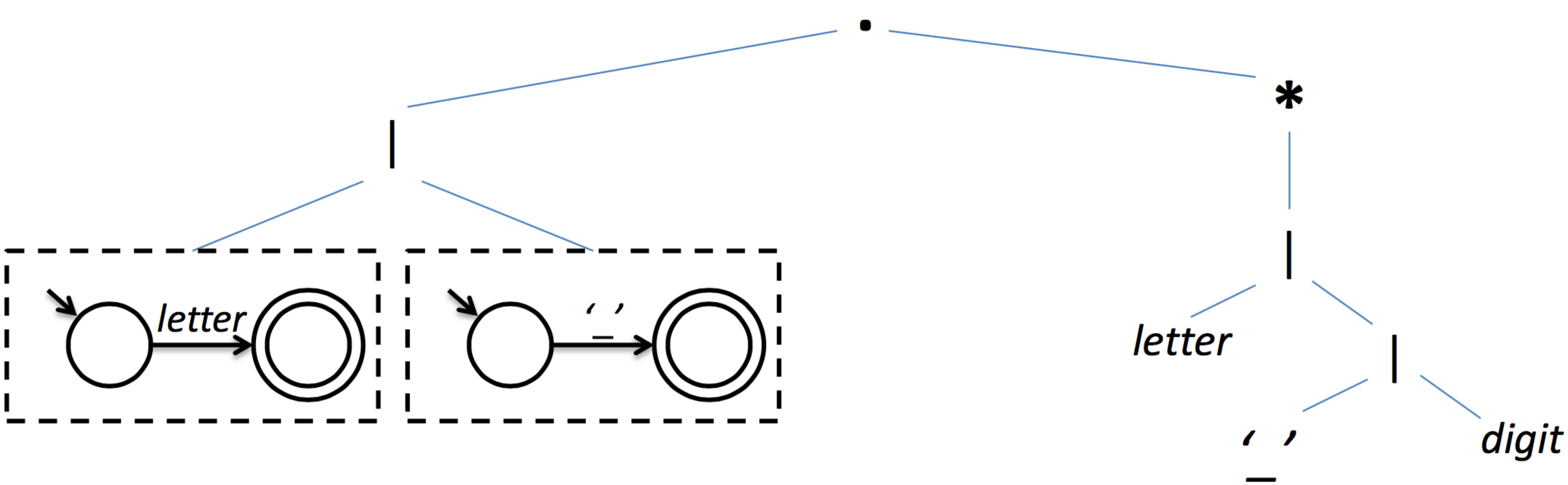
Would be asked in the exam.



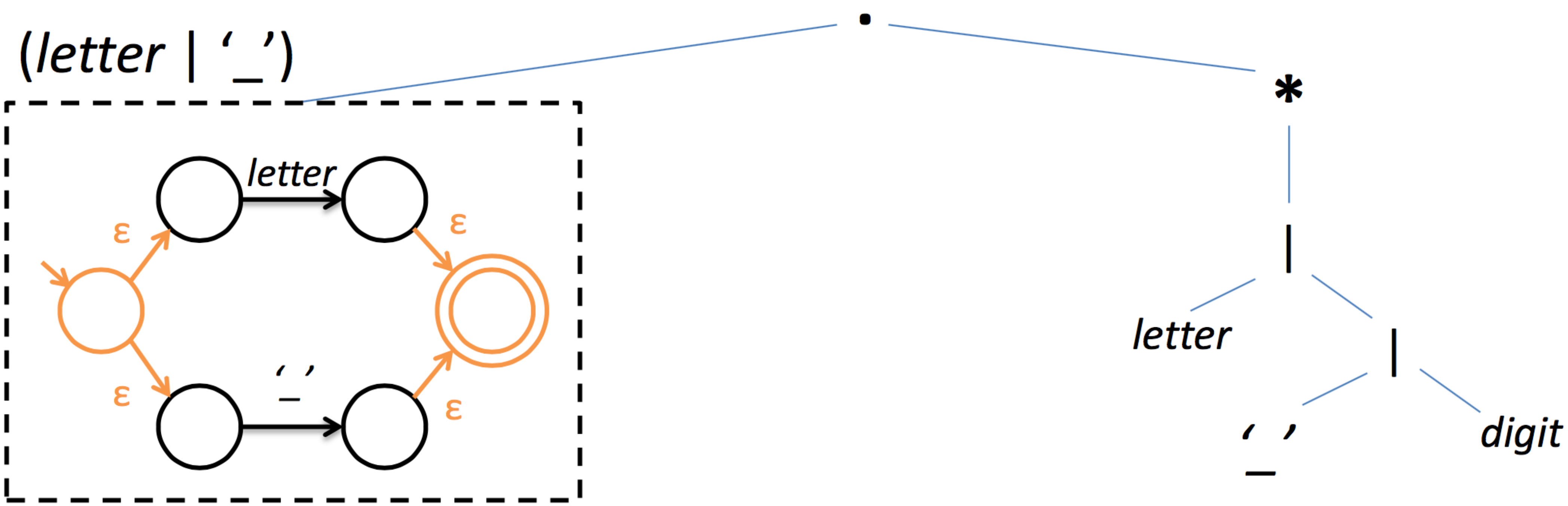
# Bottom-up conversion



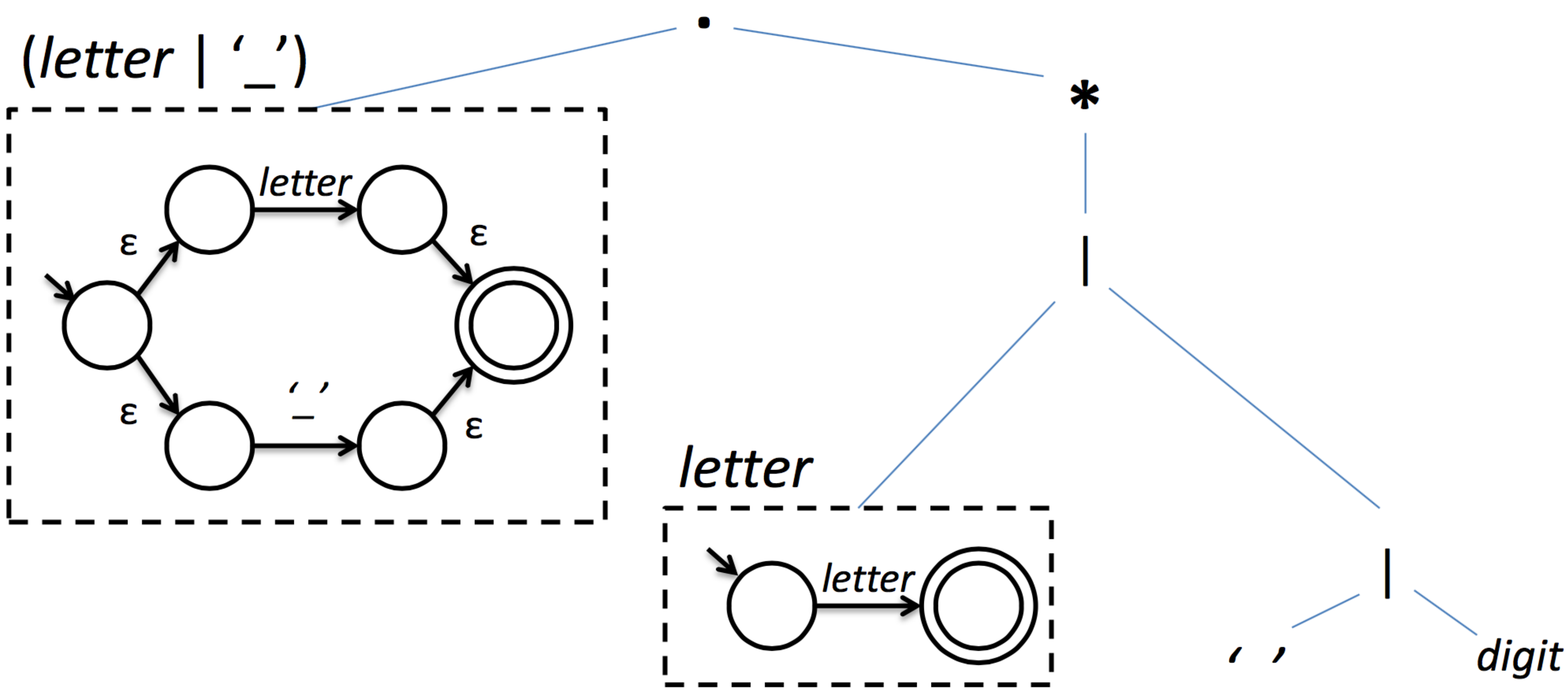
# Bottom-up conversion



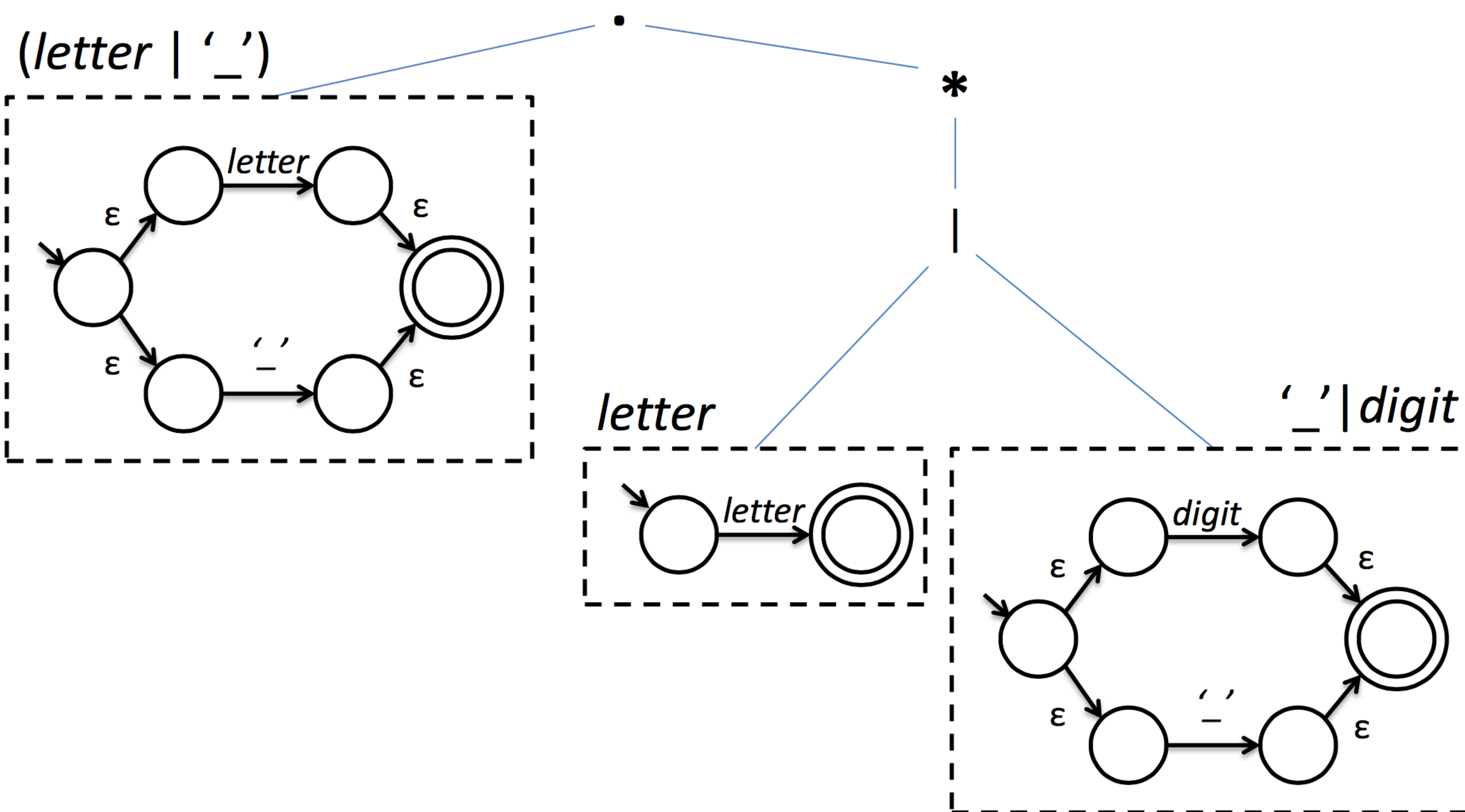
# Bottom-up conversion



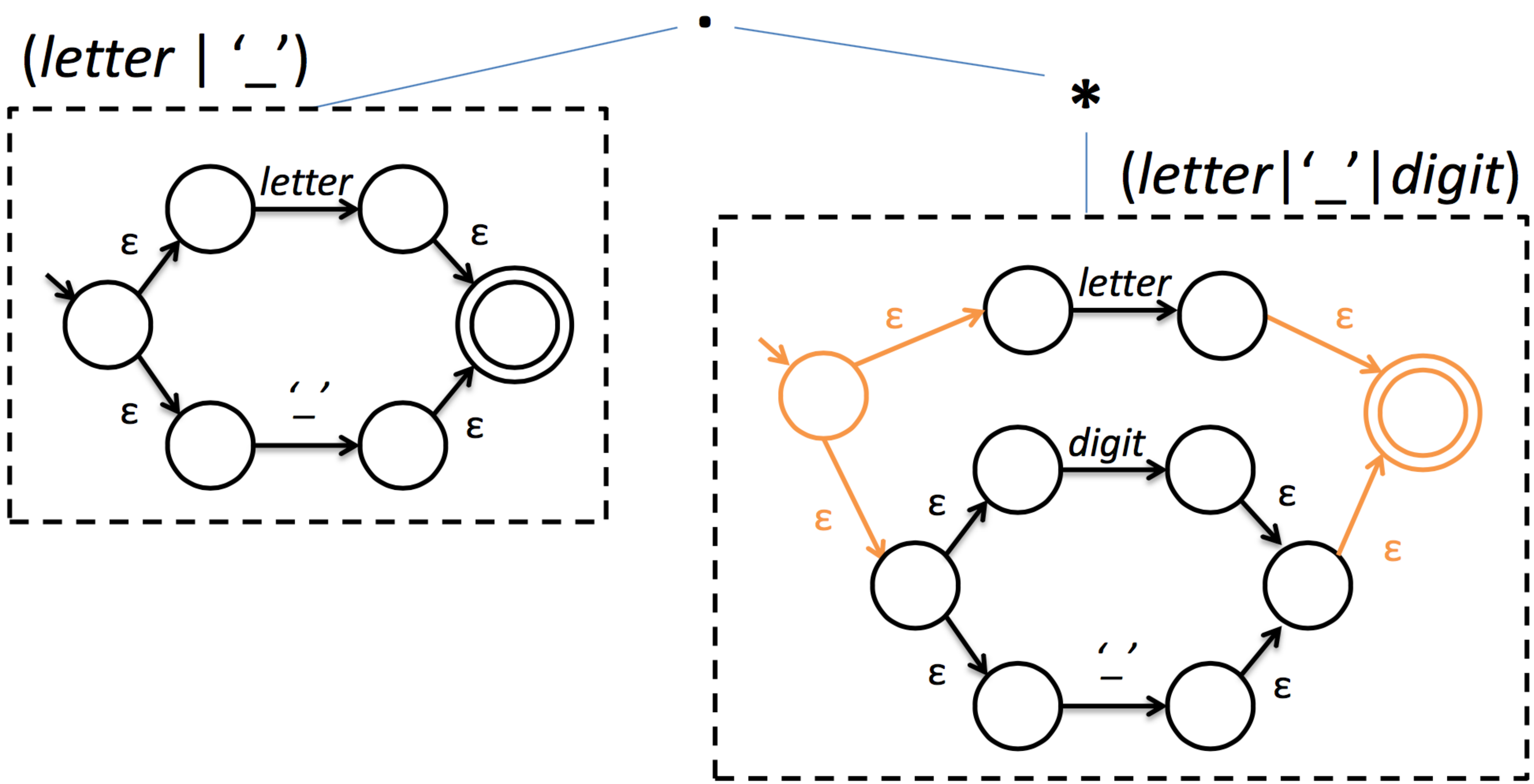
# Bottom-up conversion



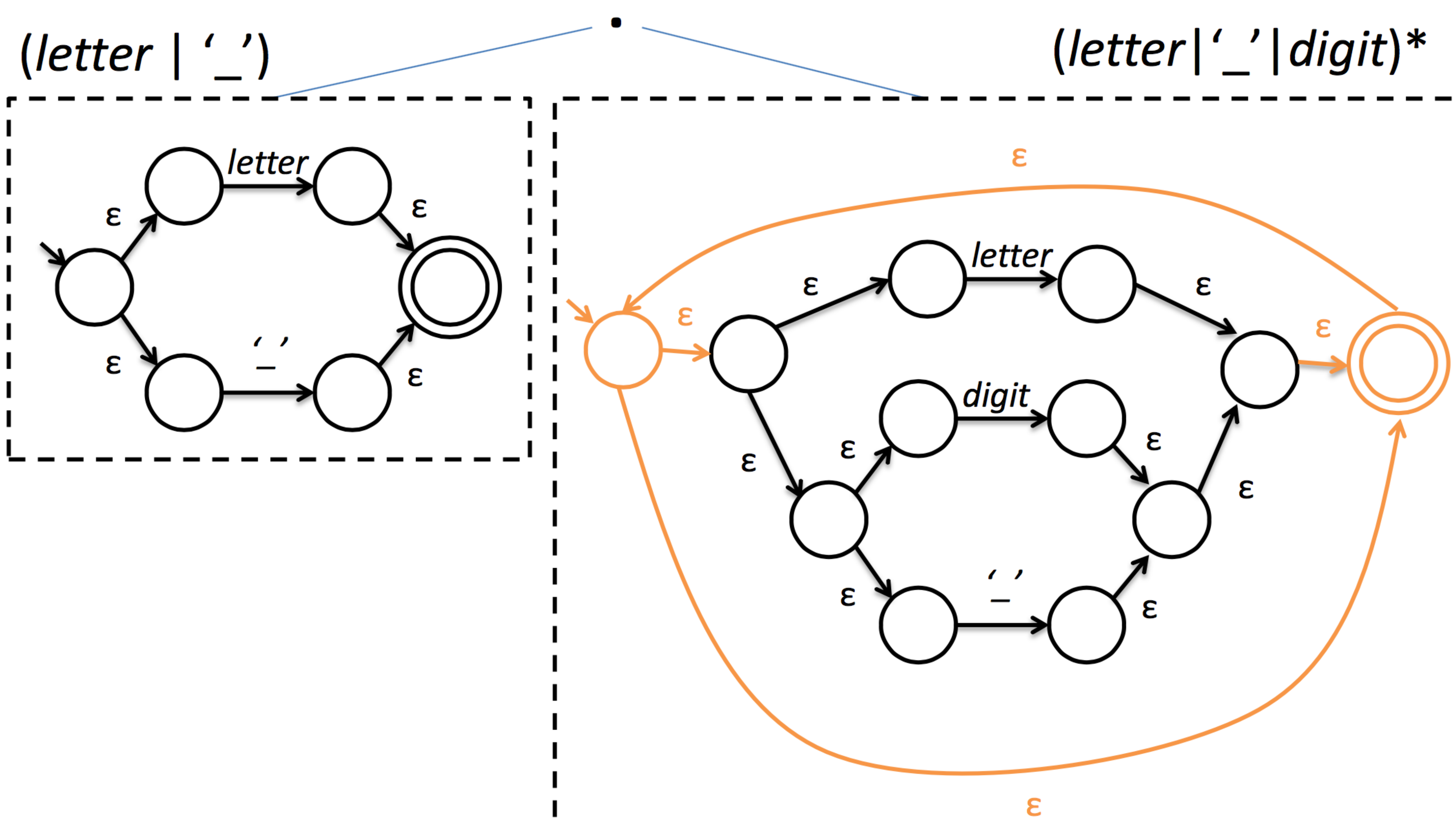
# Bottom-up conversion



# Bottom-up conversion

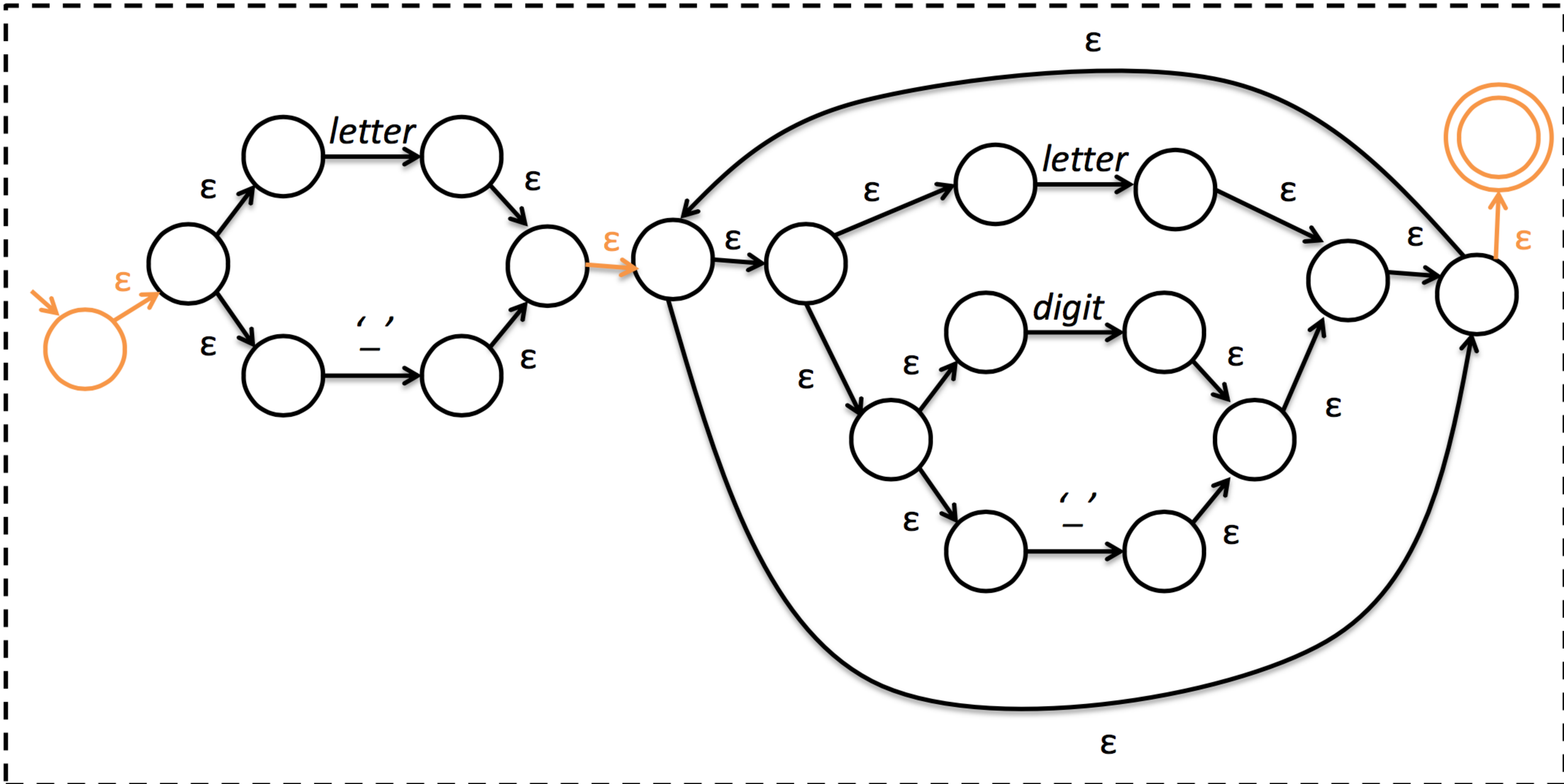


# Bottom-up conversion



# Bottom-up conversion

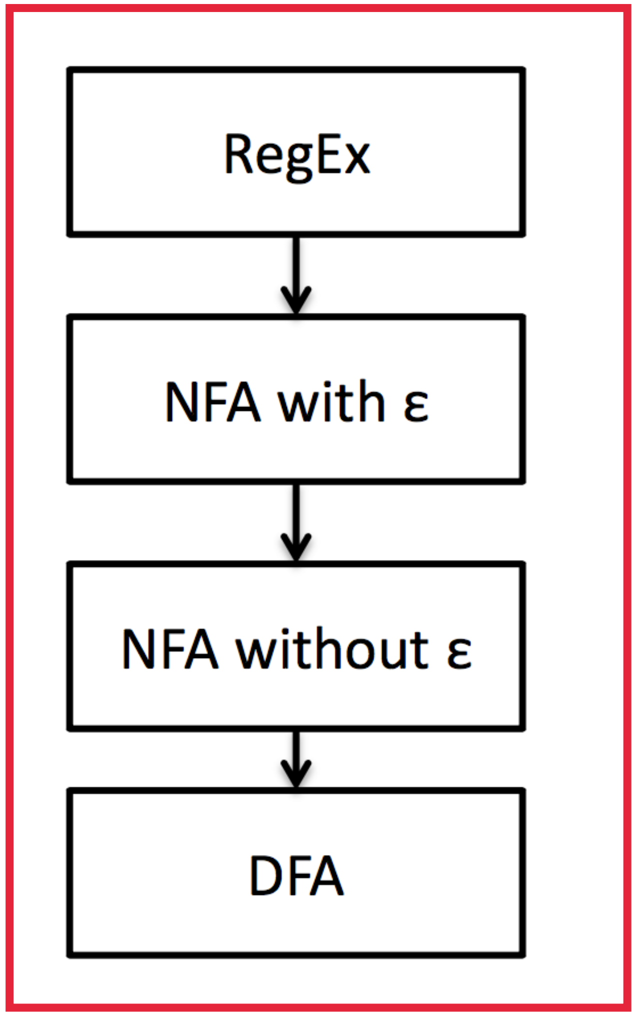
(*letter* | '_')(*letter*|'_'|*digit*)*



# Regexp to DFAs

We now have an NFA

We need to go to DFA

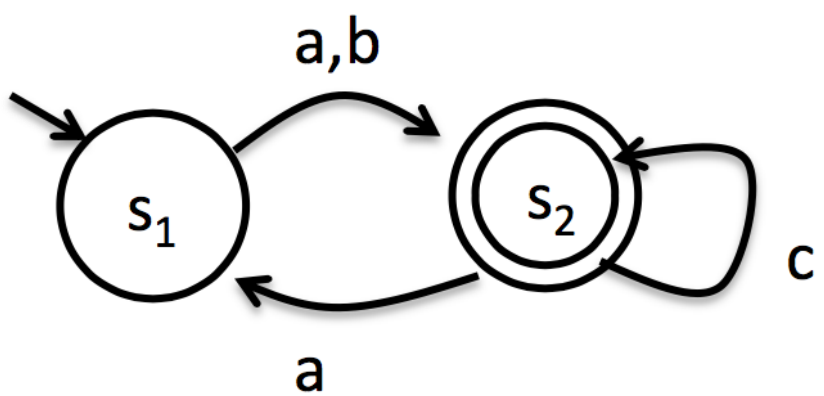



But what's so great about DFAs?

# Table-driven DFAs

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

This leads to a very efficient array representation




| | a | b | c |
|---|---|---|---|
| $s_1$ | $s_2$ | $s_2$ | |
| $s_2$ | $s_1$ | | $s_2$ |

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

# FSMs for tokenization

Changes needed to use FSM for tokenization:
1. The FSM need to have the some actions. Specifically, it needs to be able to return the current token, as well as the ability to put back some number of characters;
2. It is no longer correct to run the FSM program to run until stuck, reach EOF, or a token is found, because in general the input consists of many tokens. Thus, we should just restart the FSM machine to look for more tokens.

FSMs only check for language membership of a string

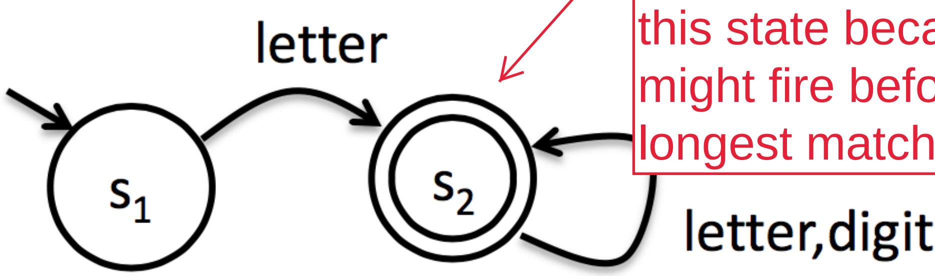
the scanner needs to recognize a stream of many different tokens using the longest match

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

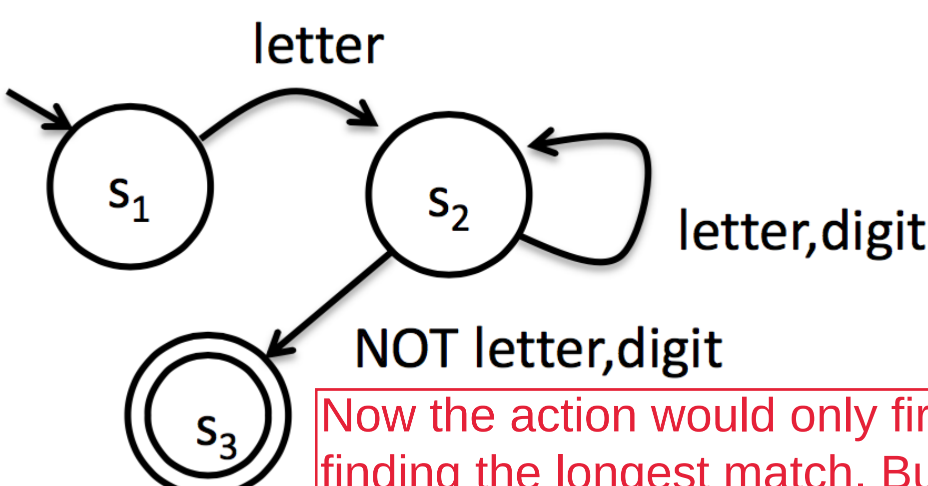



Should not just add the action to this state because the action might fire before reaching longest match.

| State | Actions |
|---|---|
| $s_2$ | return ID |

BAD: not longest match

Accounting for longest matches



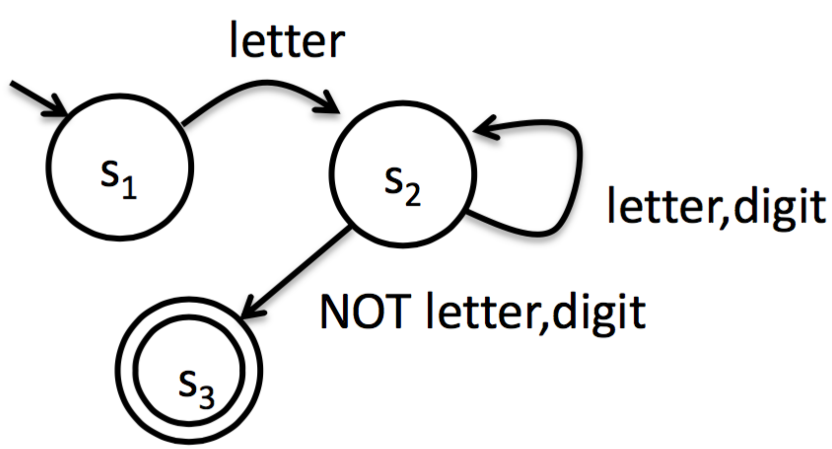

| State | Actions |
|---|---|
| $s_3$ | return ID |

Now the action would only fire when finding the longest match. But we have the problem of losing next character.

BAD: maybe we needed that character

# A second take at actions

Give our FSMs ability to put chars back




| State | Actions |
| --- | --- |
| $s_3$ | 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`

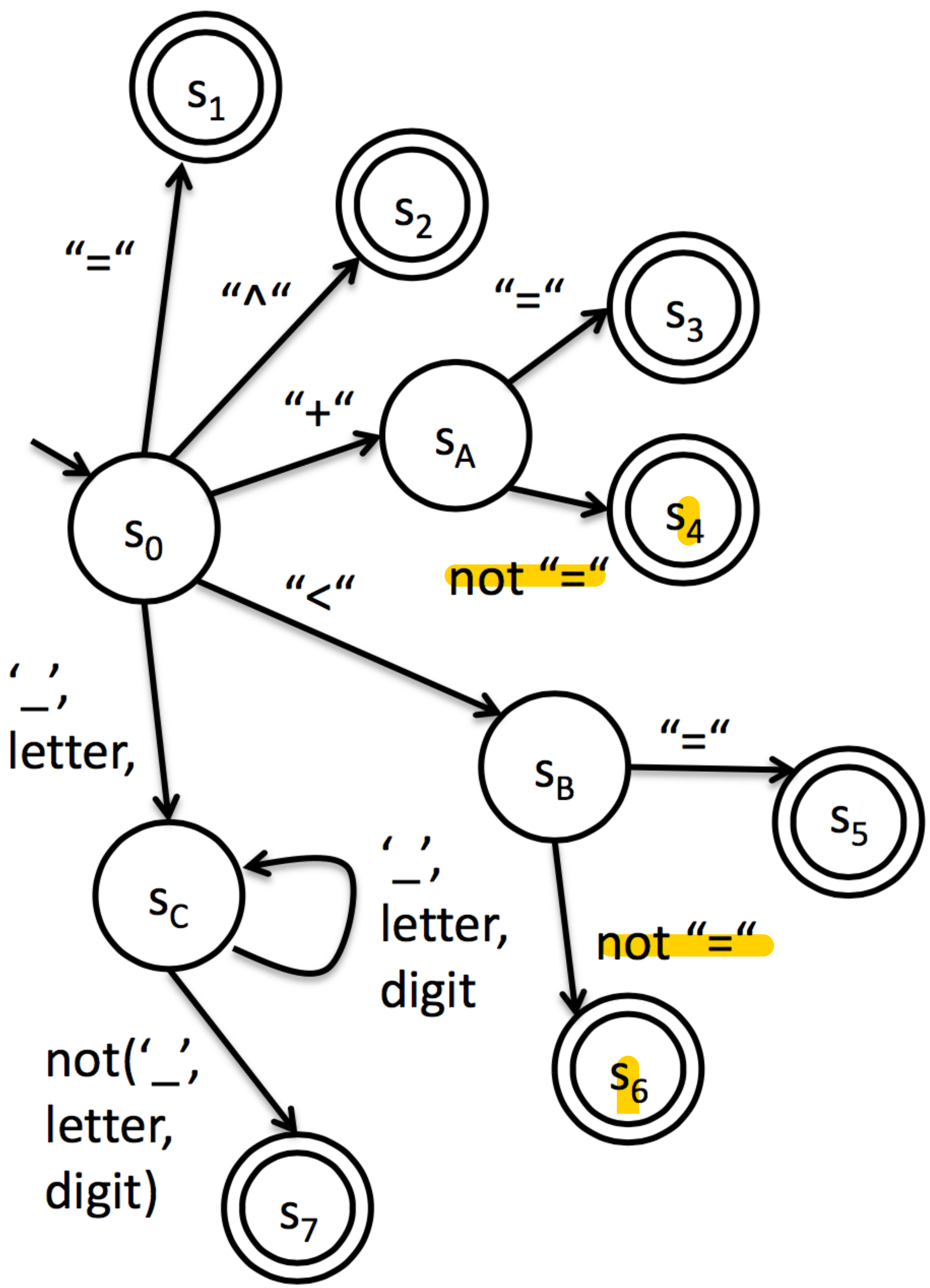
where **`expr`** is of the form

```
ID + ID

ID ^ ID

ID < ID

ID <= ID
```

| Token name | Regular Expression |
|---|---|
| ASSIGN | “=“ |
| INC | “+=“ |
| PLUS | “+” |
| EXP | “^” |
| LT | “<“ |
| LEQ | “<=“ |
| ID | $(\textit{letter} \mid \_)(\textit{letter} \mid \textit{digit} \mid \_)^*$ |

Identifiers ID follow C conventions

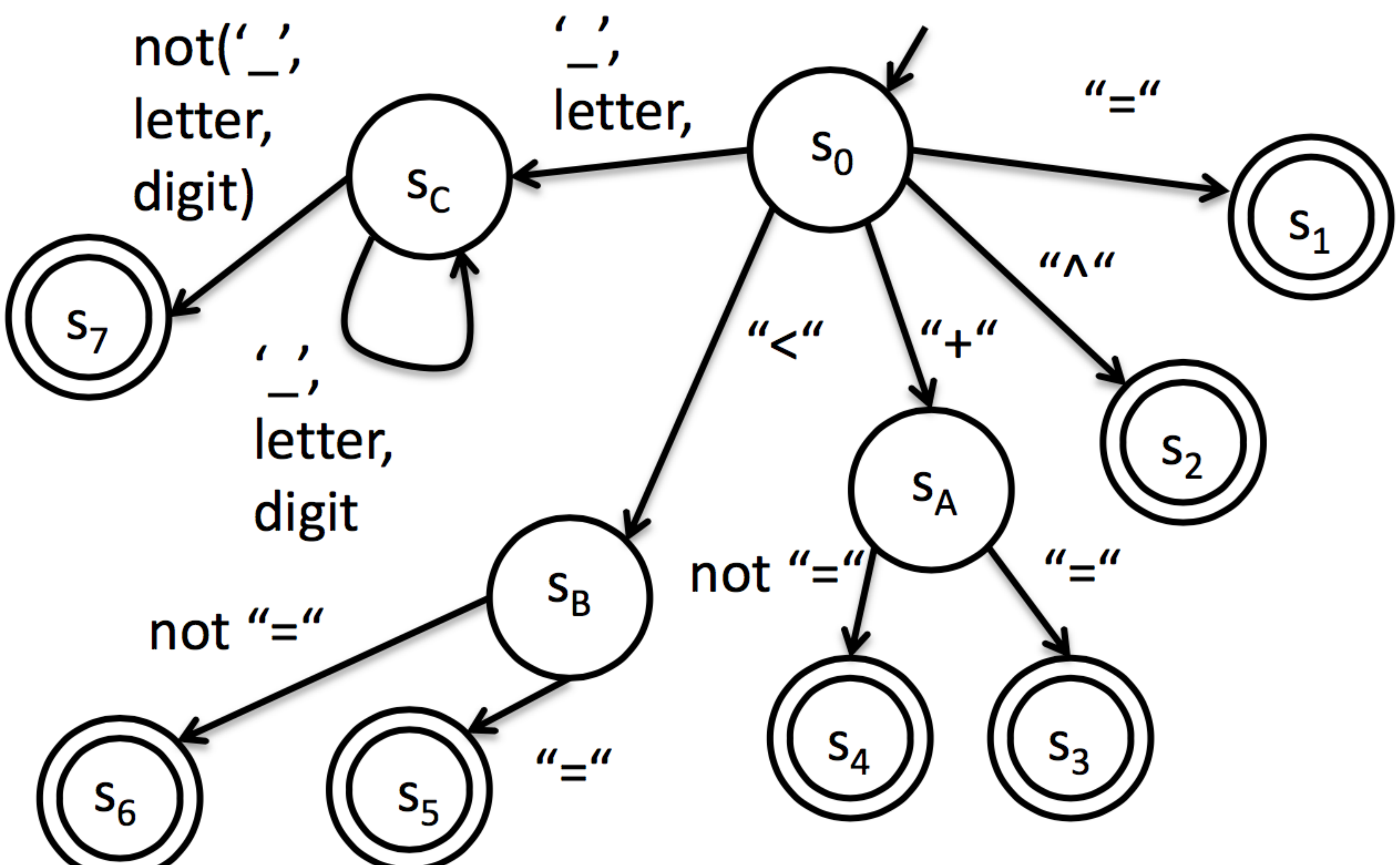
# Combined DFA




| Token name | Regular Expression |
|---|---|
| ASSIGN | "=" |
| INC | "+=" |
| PLUS | "+" |
| EXP | "^" |
| LT | "<" |
| LEQ | "<=" |
| ID | (*letter*\|_)(*letter*\|*digit*\|_)* |

| 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 |
|---|---|---|---|---|---|---|---|---|---|
| $S_0$ | Ret ASSIGN | $S_A$ | Ret EXP | $S_B$ | $S_C$ | $S_C$ | | 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 |




The table is a bit different from previously:
1. The table includes a column for EOF.
2. Each table could be an **action**, as well as a state.

| | = | + | ^ | < | _ | letter | digit | EOF | none |
|---|---|---|---|---|---|---|---|---|---|
| $S_0$ | Ret ASSIGN | $S_A$ | Ret EXP | $S_B$ | $S_C$ | $S_C$ | | 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 lexer 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]                   Macro definitions
WHITESPACE = [\040\t\n]

%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 \)

Regexp operators

- | * + ? () .

Character class operators

- \- range
- ^ not
- \ escape

`

```
"="              { 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"); }
```