Pre-Class warm up

Draw the RegEx 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+))

Announcements

- Reminder
  - P1 Part 1 due tonight!
- New assignment
  - HW1 assigned, due Tuesday
  - P1 Part 2 released tonight at midnight

Last time

- Explored NFAs
  - For every NFA there is an equivalent DFA (Rabin-Scott powerset construction)
  - ε-edges add no expressive power
- Introduced Regular Languages

This Time

- Review RegExs
- Convert RegExs to DFAs
- From Language Recognizers to Tokenizers

RegEx Review

- Consider the following RegExs for C identifiers

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

+ Not needed

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

Doesn't allow the legal string a1b2

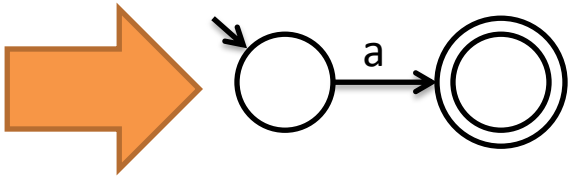
RegEx to NFAs

- Basic Idea
  - Literals/ε correspond to simple DFAs
  - Operators correspond to methods of joining operand DFAs

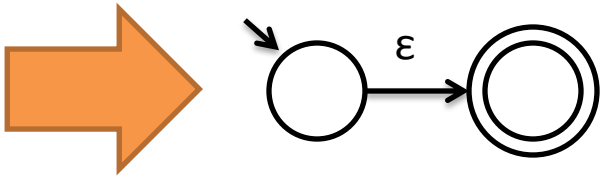
5 RegEx to NFA Conversion Rules

Rules for operands

Literal ‘a’

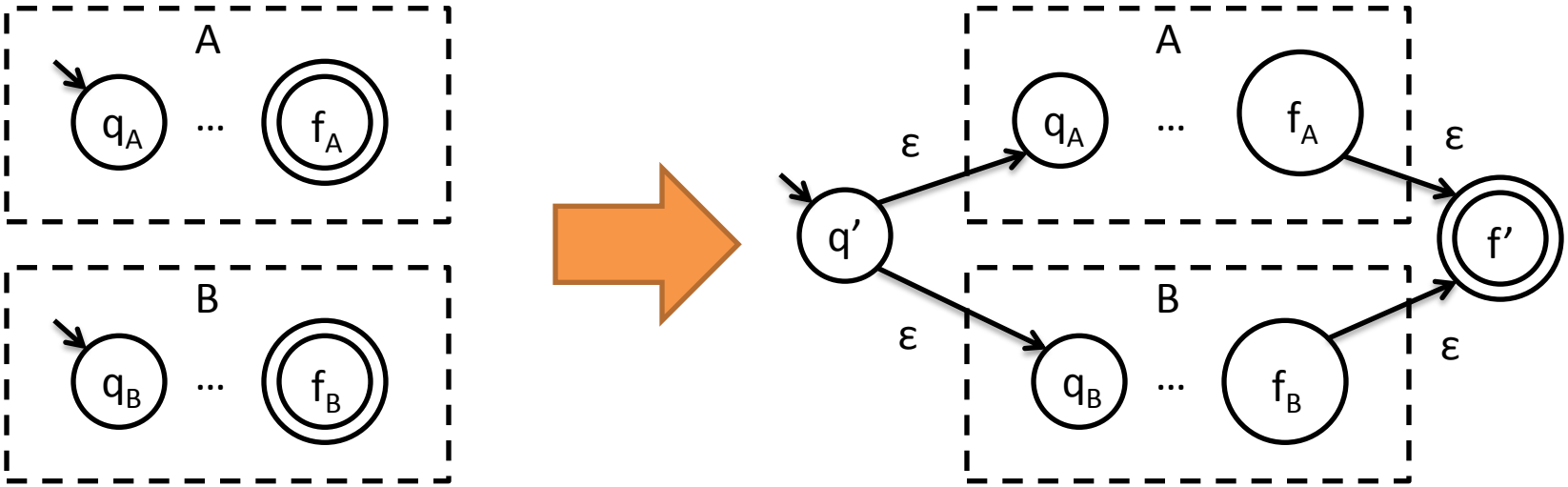


ε



5 RegEx to NFA Conversion Rules

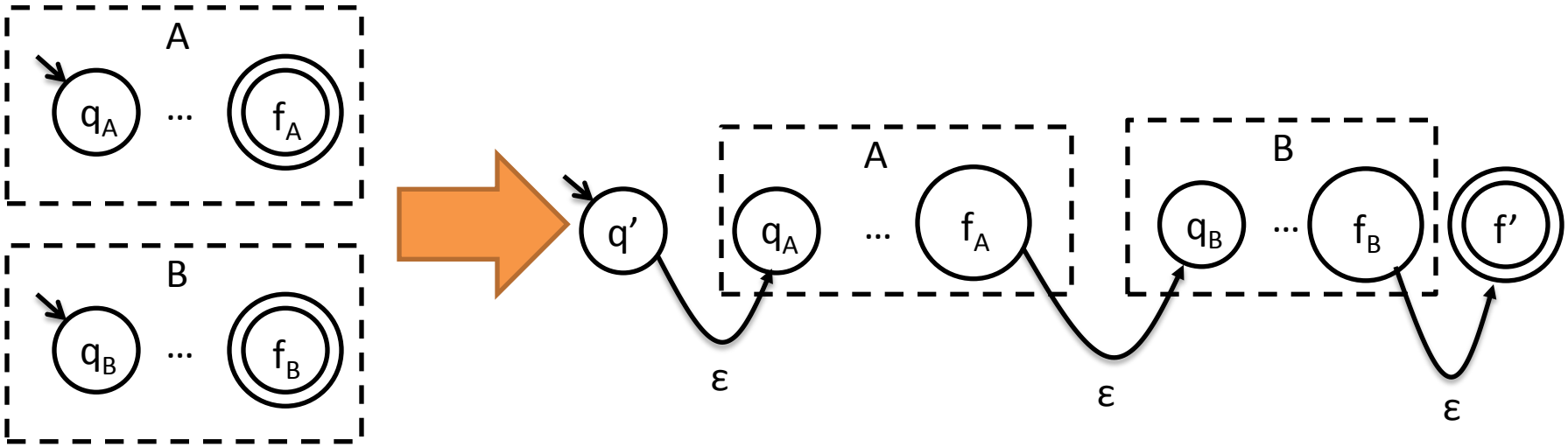
Rule for alternation A|B



- Make new start state q' and new final state f'
- Make original final states non-final
- Add to δ:
  - $q', \varepsilon \rightarrow q_A$
  - $q', \varepsilon \rightarrow q_B$
  - $F_a, \varepsilon \rightarrow f'$
  - $F_b, \varepsilon \rightarrow f'$

5 RegEx to NFA Conversion Rules

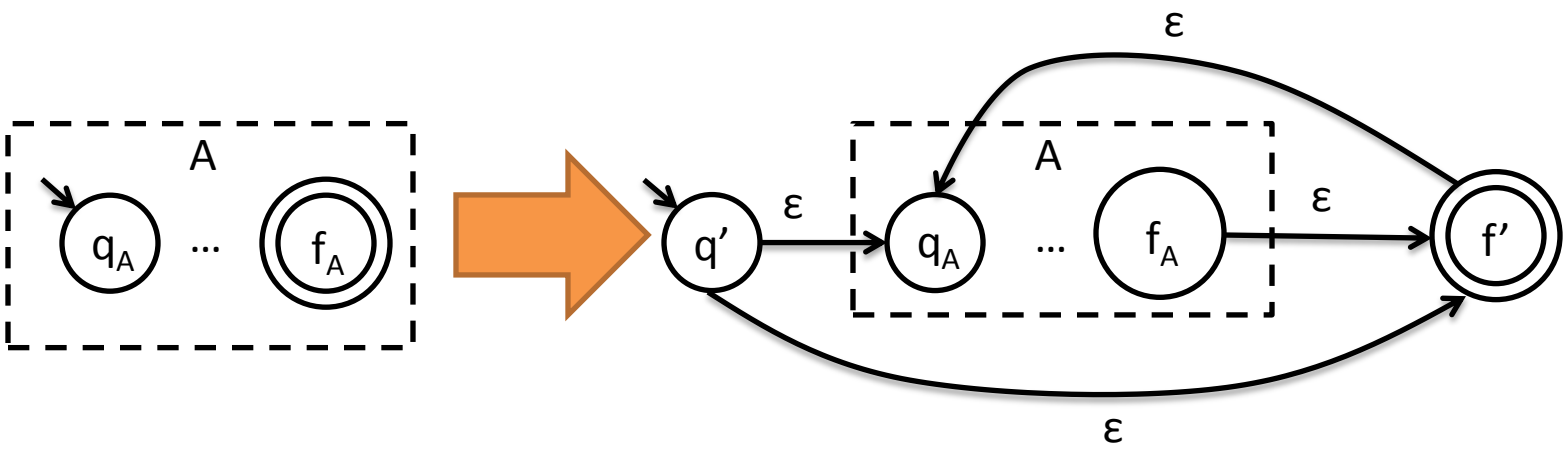
Rule for catenation A.B



- Make new start state q' and new final state f'
- Make original final states non-final
- Add to δ:
  - $q', \varepsilon \rightarrow q_A$
  - $f_A, \varepsilon \rightarrow q_B$
  - $f_b, \varepsilon \rightarrow f'$

5 RegEx to NFA Conversion Rules

Rule for iteration A*



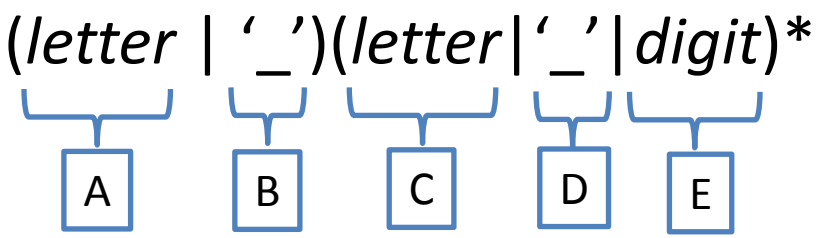
- Make new start state q' and new final state f'
- Make original final states non-final
- Add to δ:
  - q',ε → q_A
  - q',ε → f'
  - f', ε → q_A

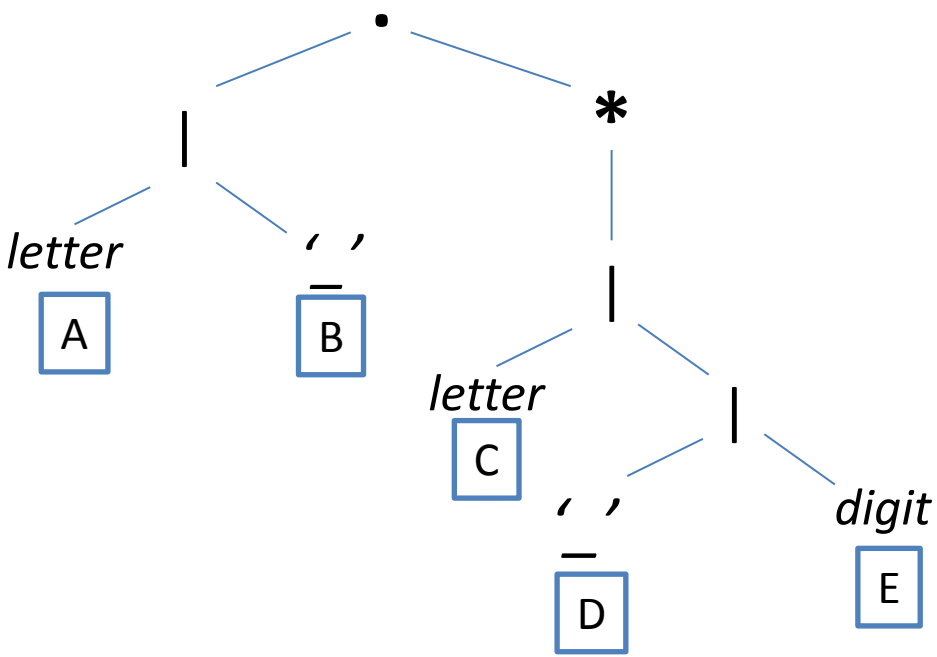
Regex Operator Precedence

Operator	Precedence	Analogous math operator
|	Low	Addition
.	Medium	Multiplication
*	High	Exponentiation

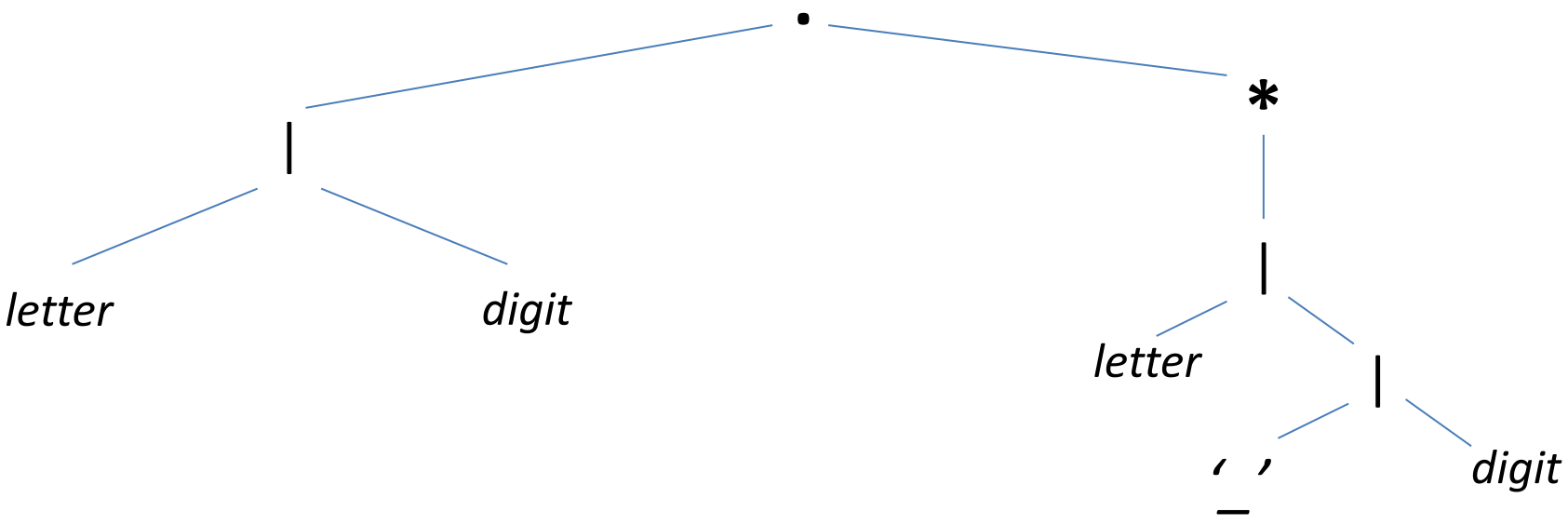
A Tree Representation of a RegEx

Op	Precedence
|	Low
.	Medium
*	High

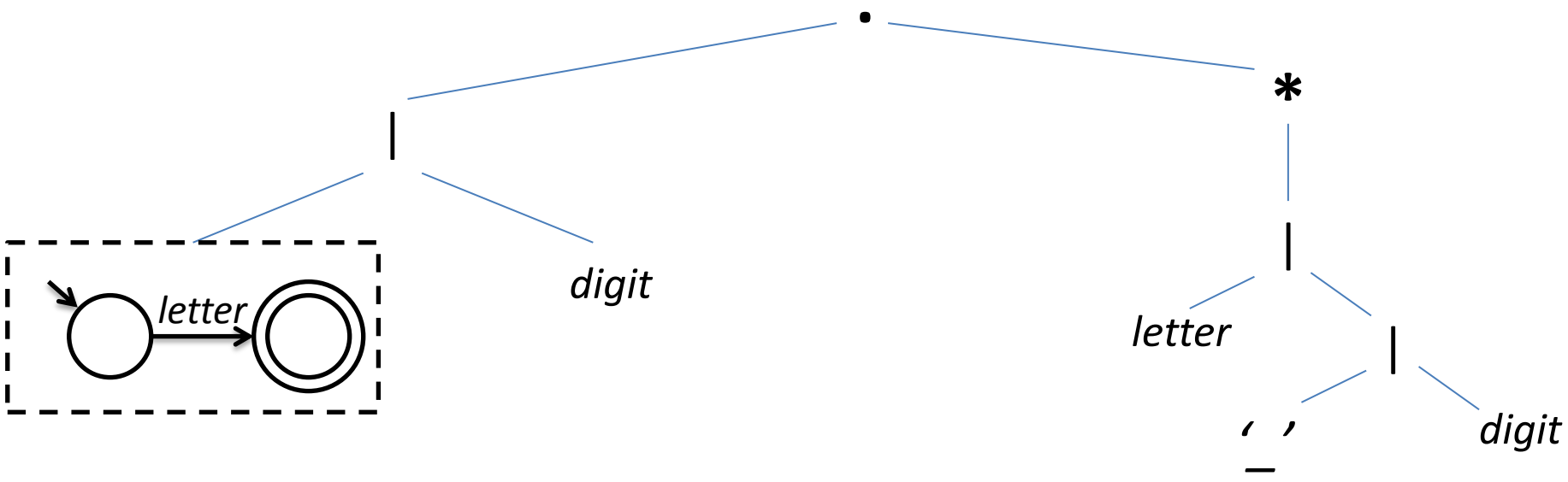




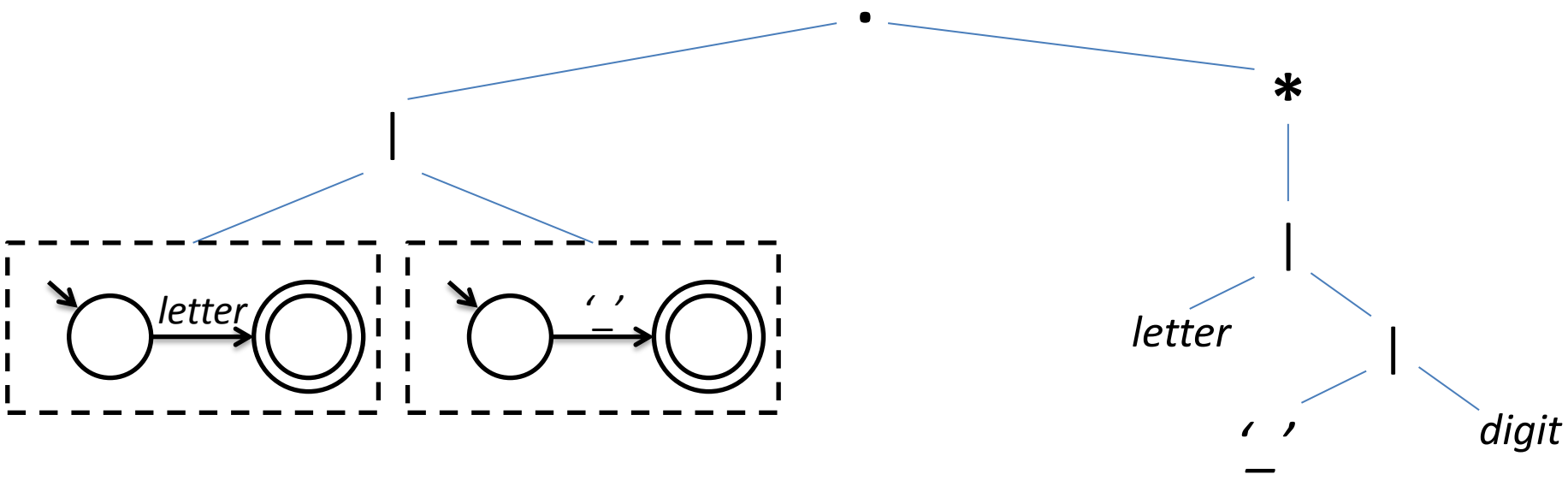
Replace RegEx Bottom-Up



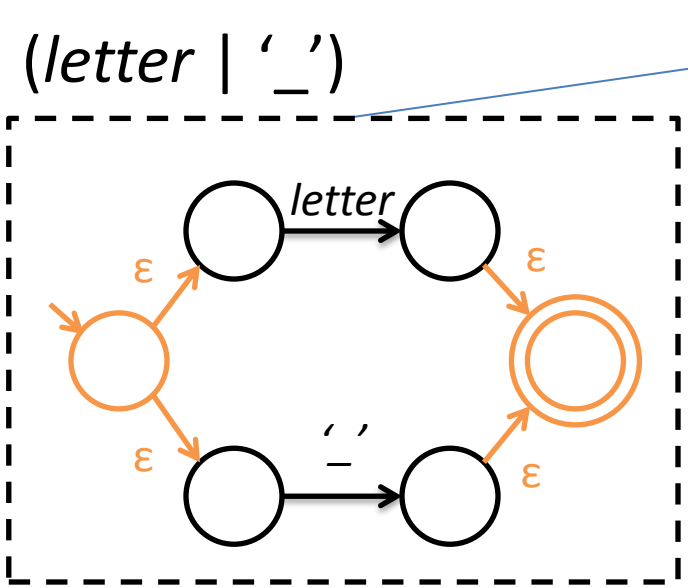
Replace RegEx Bottom-Up

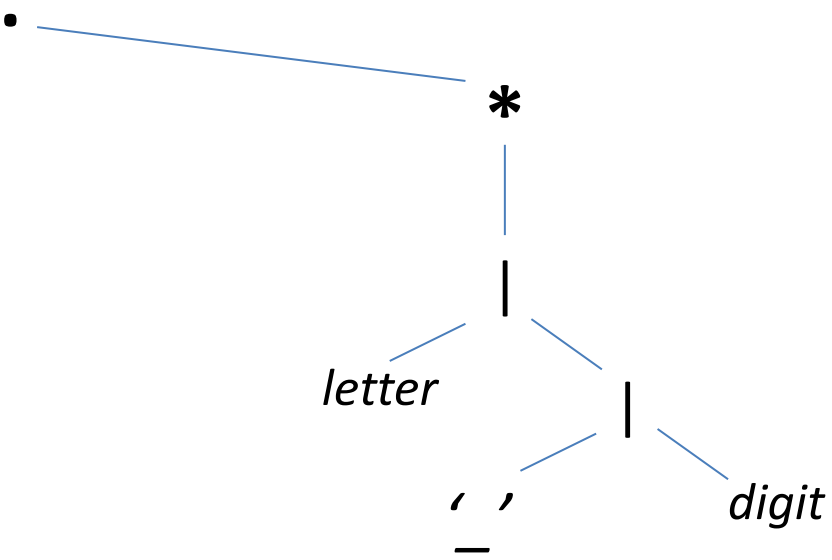


Replace RegEx Bottom-Up

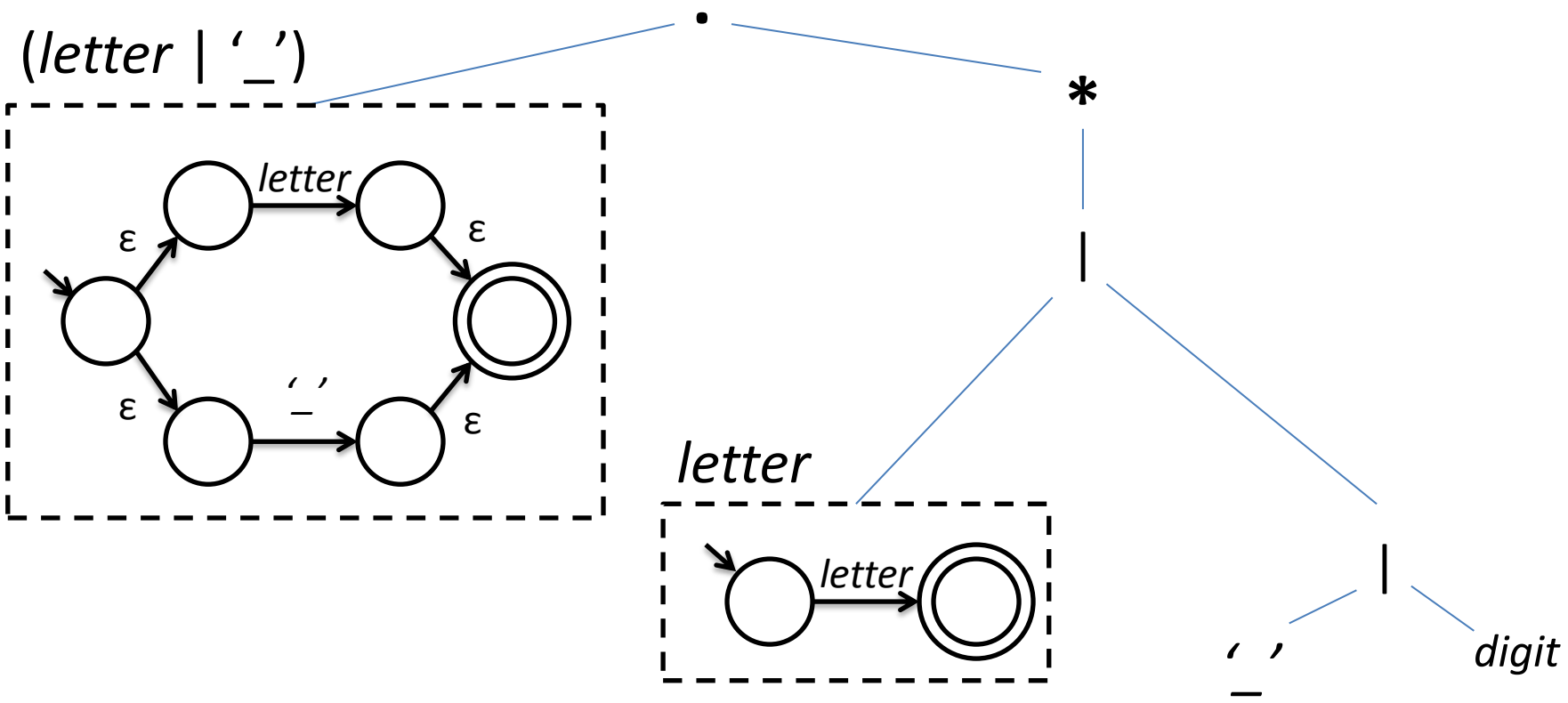


Replace RegEx Bottom-Up

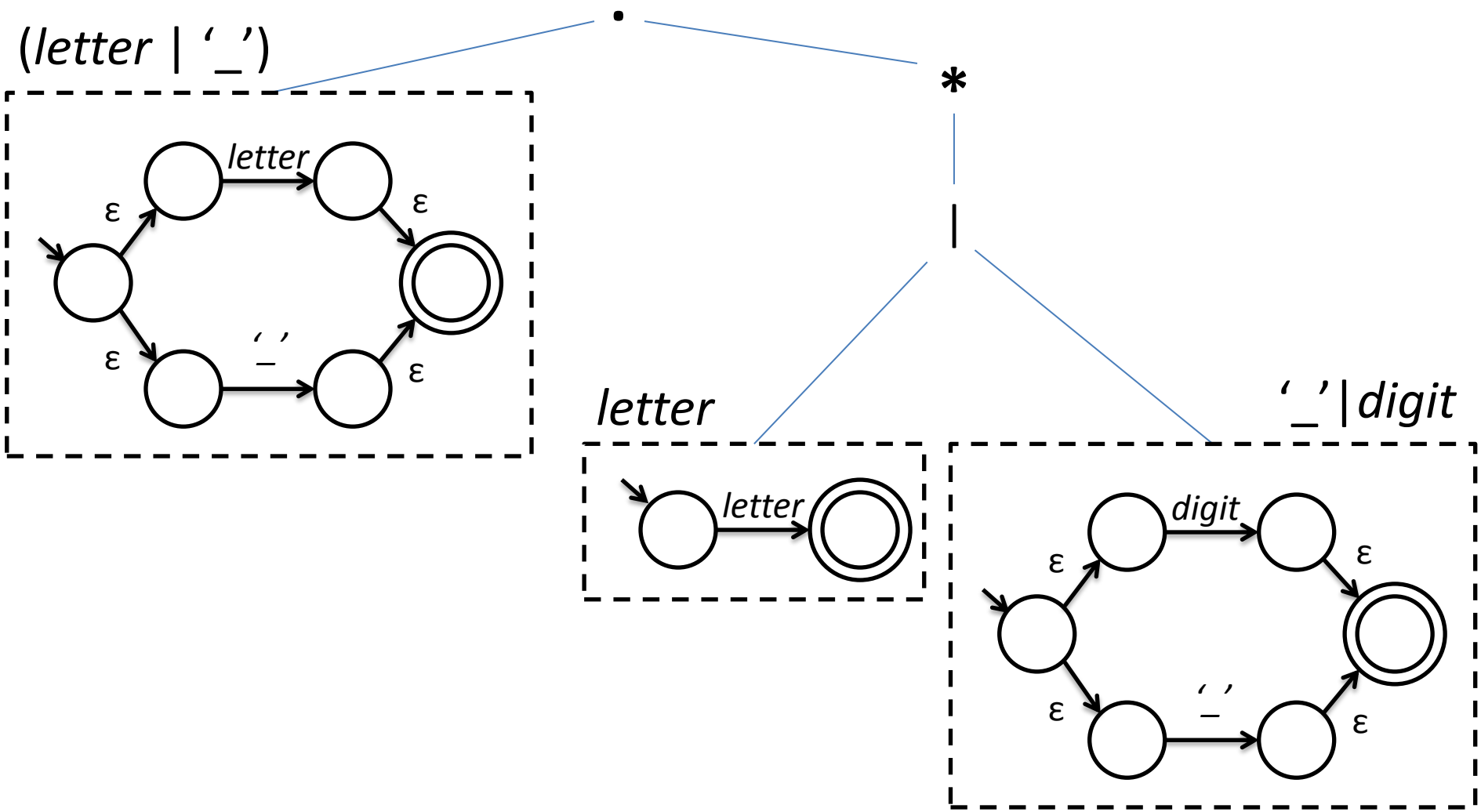




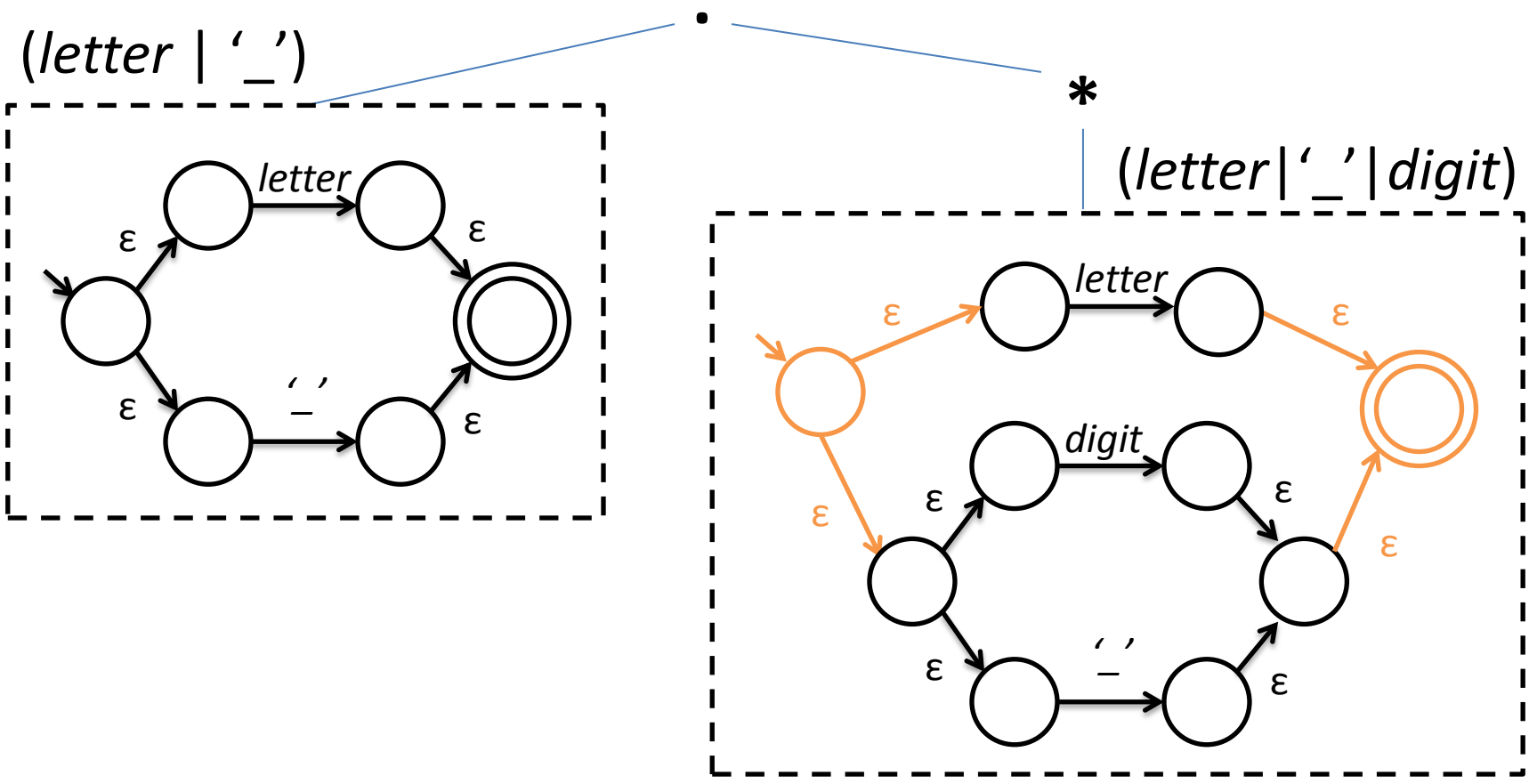
Replace RegEx Bottom-Up



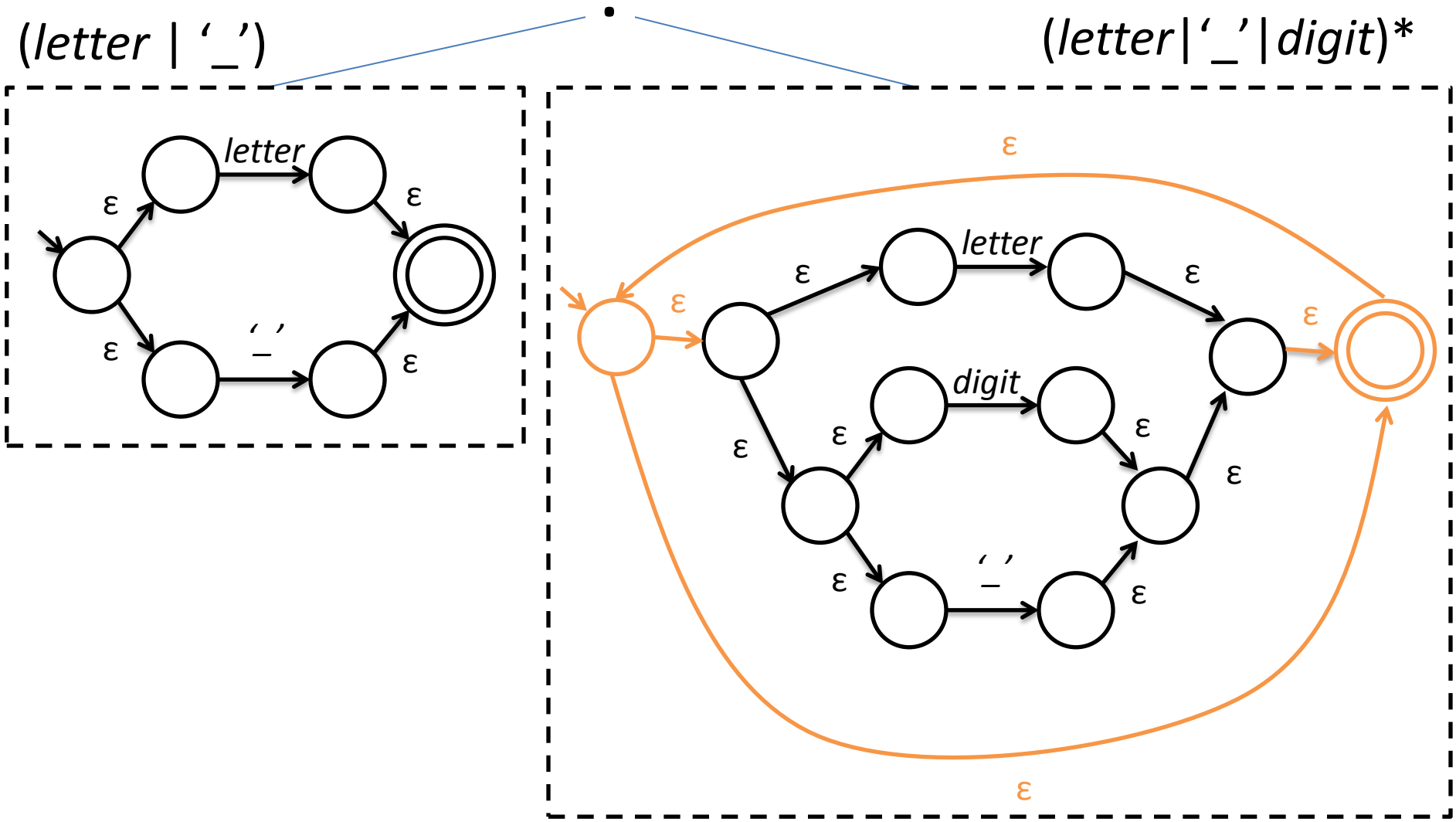
Replace RegEx Bottom-Up



Replace RegEx Bottom-Up

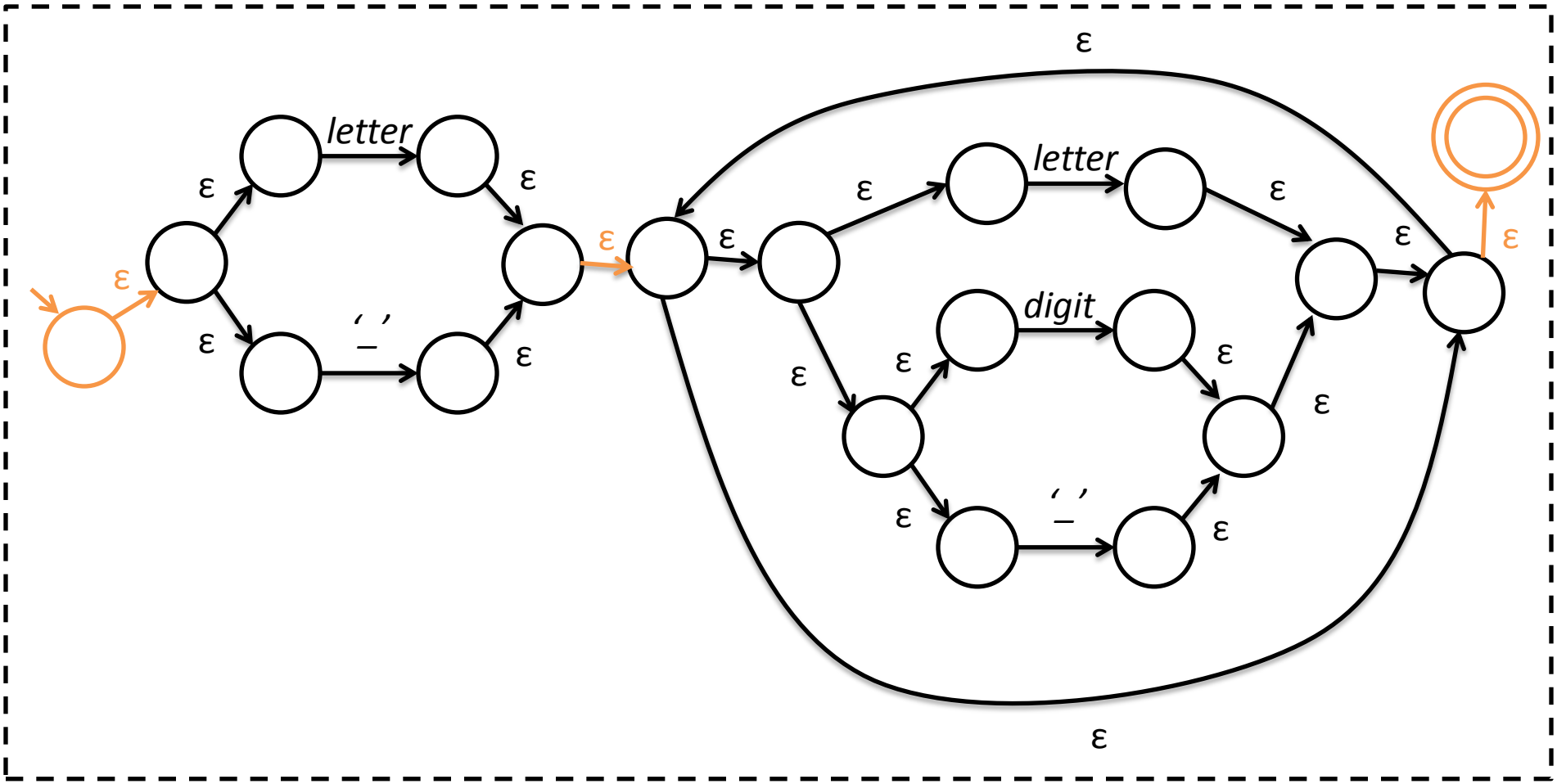


Replace RegEx Bottom-Up



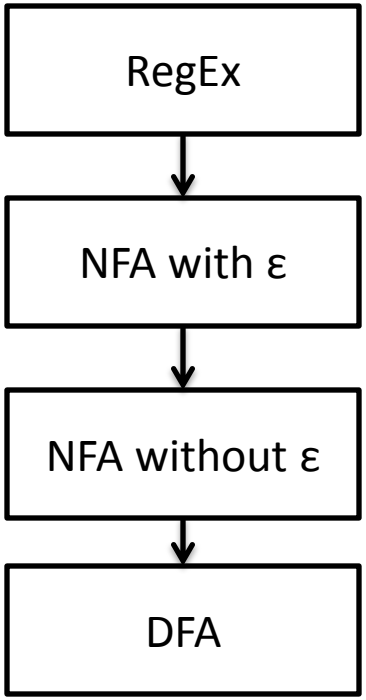
Replace RegEx Bottom-Up

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



RegEx to DFAs

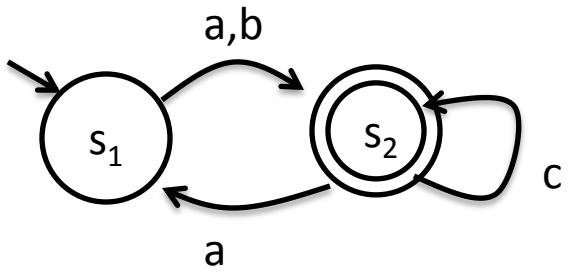
- Now that we've defined a way to go from RegEx to NFA with ε, we can go all the way from RegEx to DFA:



But what's so great about DFAs?

Table-Driven DFAs

- Recall that δ can be expressed as a table

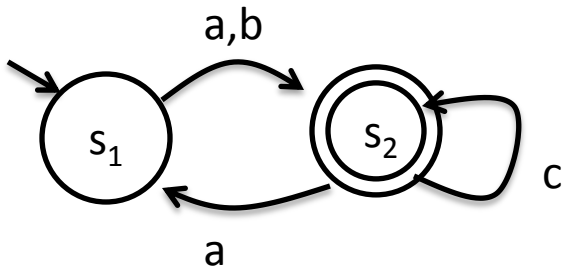


	a	b	c
s_1	s_2	s_2	
s_2	s_1		s_2

- This leads to a very efficient representation as a 2D array

Table-Driven DFAs

- Recall that δ can be expressed as a table



	a	b	c
s_1	s_2	s_2	
s_2	s_1		s_2

- This leads to a very efficient representation as a 2D array

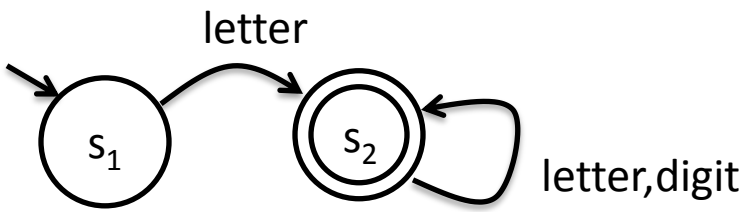
```
s = start state
While (more input){
   c = read char
   s = table[s][c]
}
if s is final, accept
```

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 the state is reached

A first cut at Actions

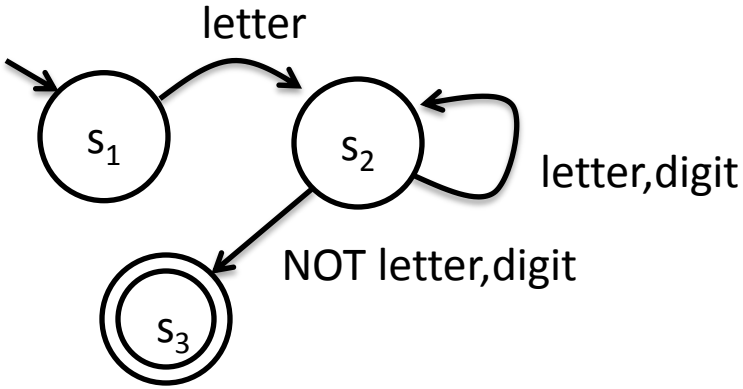
- Consider the language of Pascal identifiers:



State	Actions
s_2	return ID

BAD: Not longest match!

- Accounting for longest matches

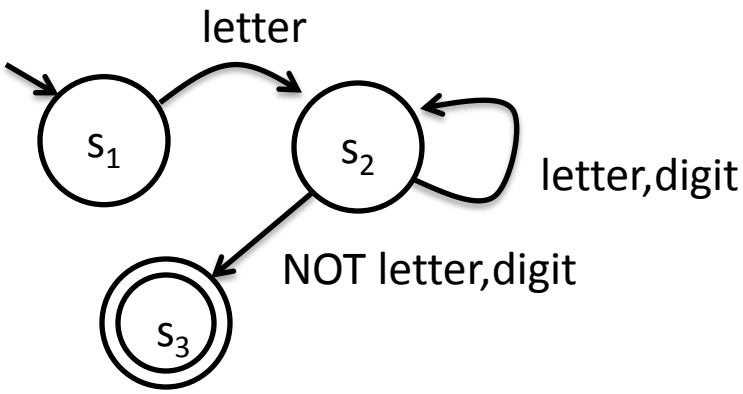


State	Actions
s_3	return ID

BAD: maybe we needed that NOT letter,digit character

A second try at Actions

- Give our FSMs the ability to put chars back



State	Actions
s_3	Put 1 char back, return ID

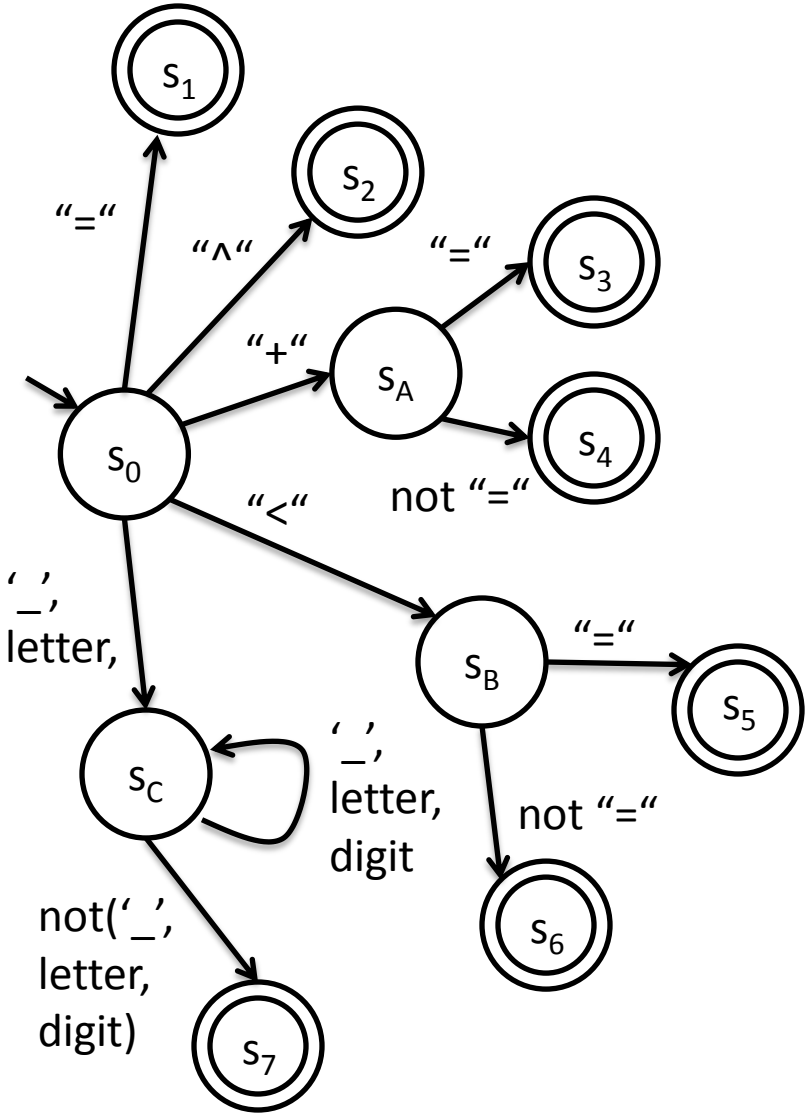
- Since we are allowing our FSM to "peek" at characters past the end of a valid token, it's also convenient to add an EOF (end of file) alphabet symbol

Our first Scanner

- Consider a language with two simple statements
  - Assignments: ID = expr
  - Increments: ID += expr
- Where expr is of the form:
  - ID + ID
  - ID ^ ID
  - ID < ID
  - ID <= ID
- Identifiers ID follow C conventions

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

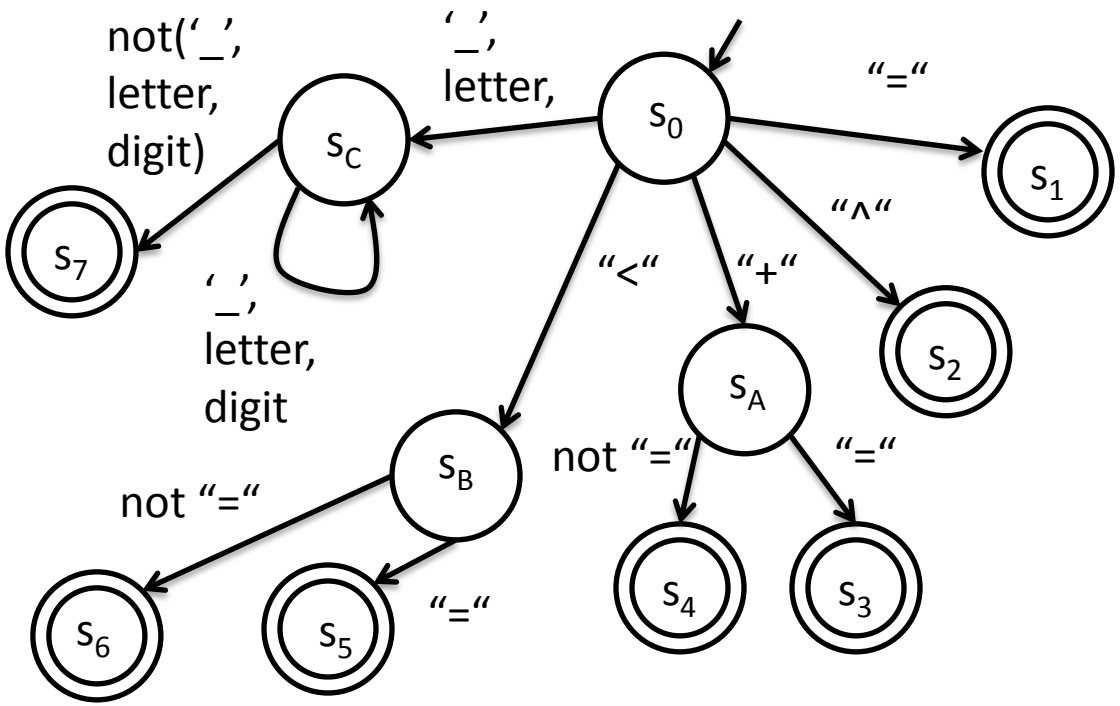
Combined DFA



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	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



	=	+	^	<	_	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

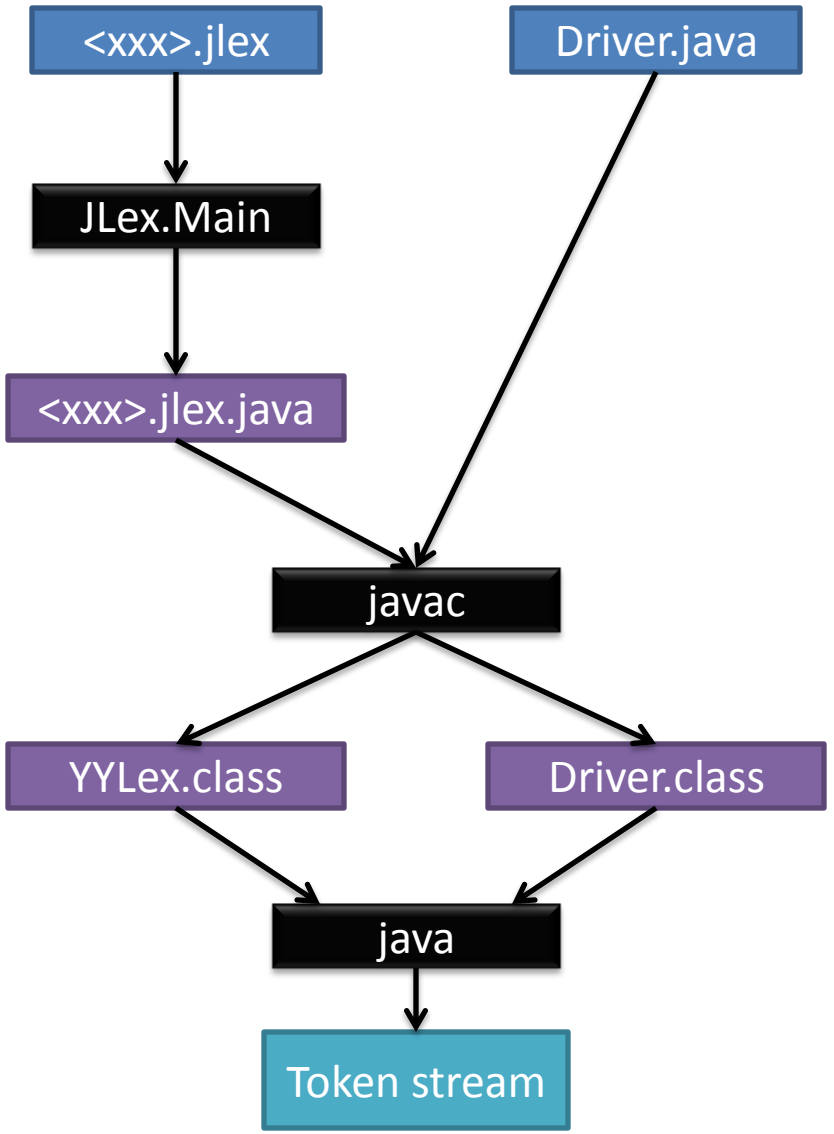
```
do{
    read char
    perform action / update state
    if (action was to return a token){
        start again in start state
    }
} (while not EOF or stuck);
```

Lexical Analyzer Generators

- The transformation from RegEx set to Scanner is formally well-defined
- Can write tools to synthesize a Lexer automatically
  - Lex: Unix scanner generator, build in C
  - Flex: Fast Lex
  - Jlex: Java version of Lex

JLex

- Declarative Specification
  - Tell it what you want scanned, it will figure out how to do it
- Input: set of RegExs + associated actions
  - Called the JLex specification: <xxx>.jlex
- Output: Java source code for a scanner
  - <xxx>.jlex.java



<xxx>.jlex format

- 3 sections separated by %%
  - User code section
  - Directives
  - Regular Expressions + Actions Section: “rules”

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

%%

//Directives Section

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

%state SPECIALINTSTATE

//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
```

Macro definitions

State declaration

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 chars)
  - Chars inside “ ” represent themselves (except \”)
- Regex Operators

  | * + ? ( ) .
- Character Class Operators

  - range

  ^ not

  \ escape

JLex 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"); }
```