CS301 W'99 Lecture Notes
Lecture 5

Finite Automata

A non-deterministic finite automaton (NFA) consists of:

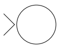
(1) An input alphabet Σ

e.g., Σ = {a,b}

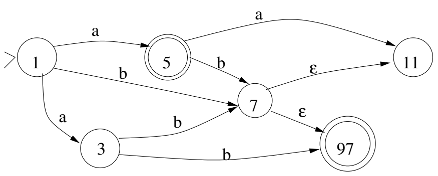
(2) A set of states S

e.g. S = {1,3,5,7,11,97}

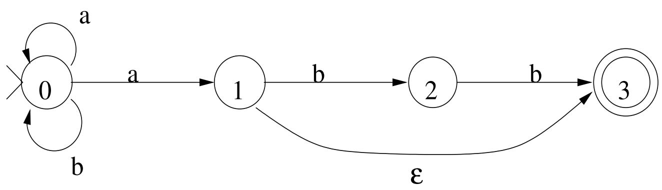
(3) A set of transitions from states to states, labelled by elements of A or e.

(4) A start state. 

(5) A set of final states. 



Small Example



Can also write as **transition table**.

State	Input a	b	ϵ
0	{0,1}	{0}	-
1	-	{2}	{3}
2	-	{3}	-
3	-	-	-

An NFA **accepts** the **string** x if there is a path from start to final state labeled by the the characters of x.

Example: NFA above accepts "*aaabbabb*"

An NFA **accepts** the **language** L if it accepts exactly the strings in L.

Example: NFA above accepts the language defined by the R.E. $(a^*b^*)^*a(bb|\epsilon\ )$

Fact: For every regular language L, there exists an NFA that accepts L.

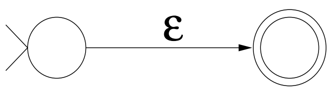
NFA's from R.E.'s

Can give an algorithm for constructing an NFA from an R.E., such that the NFA accepts the language defined by the R.E.

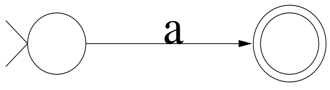
- Algorithm is recursive, and is based on the recursively defined structure of R.E.'s.
- Makes heavy use of ϵ -transitions.

Base Constructions

ε



a in Σ

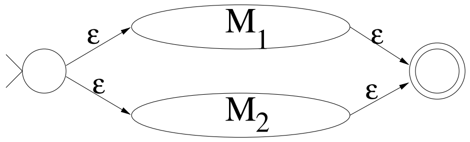


Inductive Constructions build new machines by connecting existing machines using ϵ -transitions to existing initial states and from existing final states.

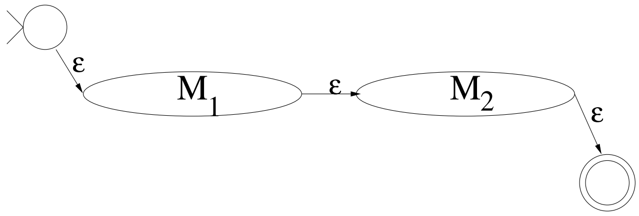
Note that each constructed machine has exactly one initial state and one final state.

Inductive Constructions

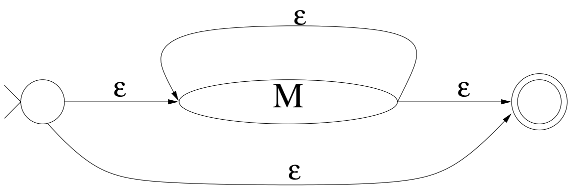
$R_1 | R_2$



$R_1 \cdot R_2$

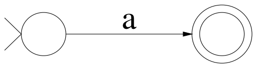


R^*

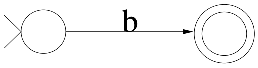


Example: (a|b)*|(cbb)

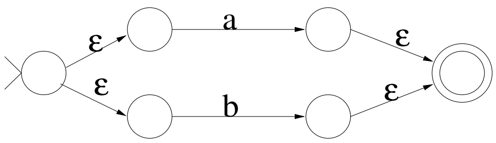
a



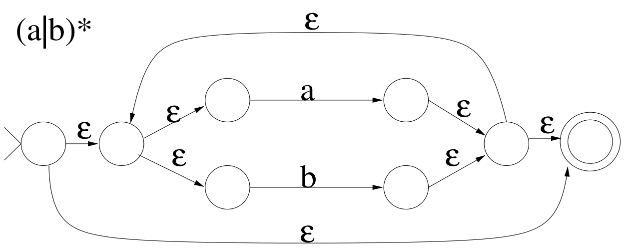
b



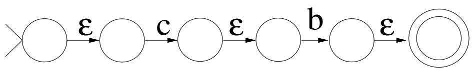
a|b



(a|b)*

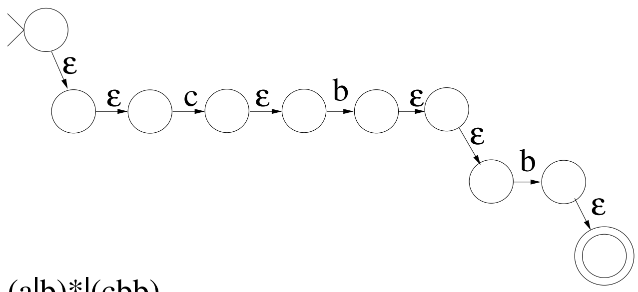


cb

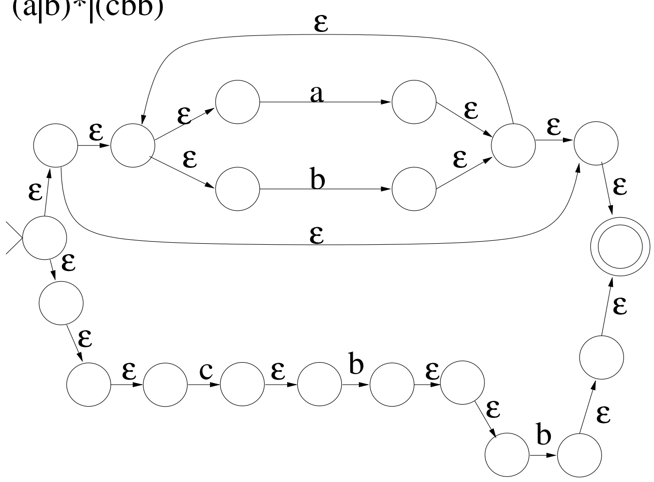


Example (continued)

cbb



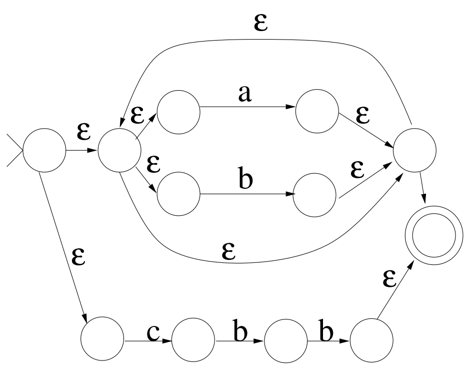
(a|b)*|(cbb)



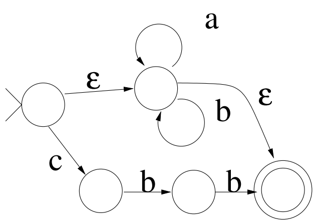
Example (continued)

Can simplify NFA's by removing useless empty-string transitions:

(a|b)*|(cbb)

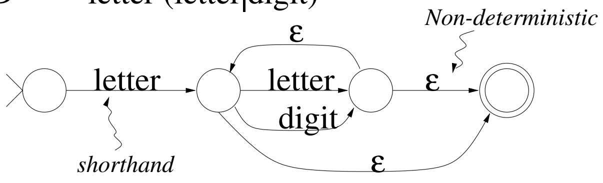


Or even simpler:

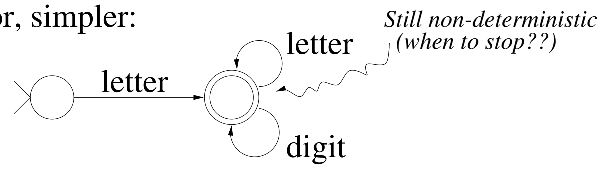


NFA's for lexical pattern R.E.s'

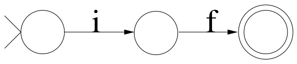
ID letter (letter|digit)*



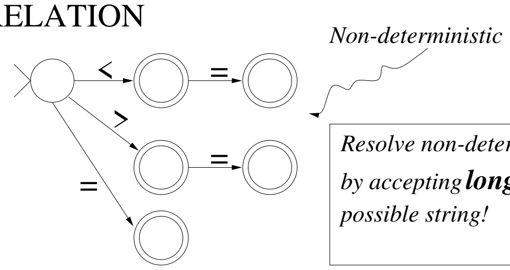
or, simpler:



IF



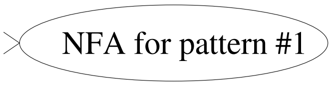
RELATION



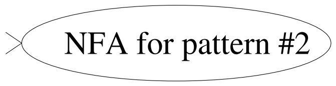
*Resolve non-determinism by accepting **longest** possible string!*

Lexical analyzer must find **best** match among a **set** of patterns.

Try:

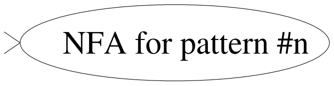


Then try:



...

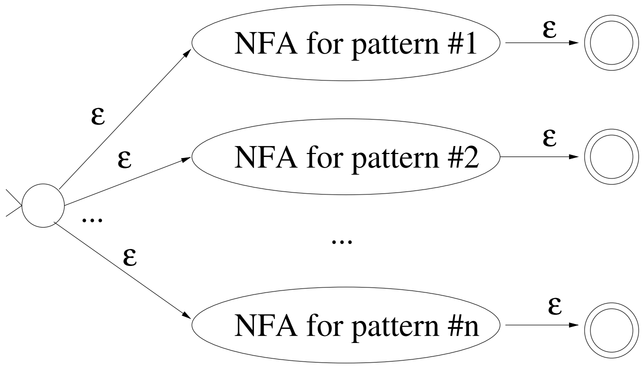
Finally, try:



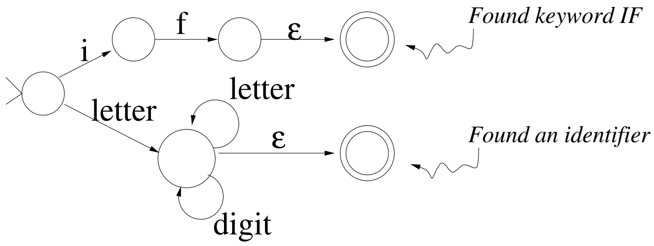
Must reset input string after each unsuccessful match attempt.

Always choose pattern that allows longest input string to match. Must specify which pattern should 'win' if two or more match the same length of input.

Alternatively, combine all the NFA's into one giant NFA, with distinguished final states:



Now can have non-determinism between patterns, as well as within a single pattern, e.g:



Implementing NFA's

Behavior of an NFA on a given input string is ambiguous.

So NFA's don't lead to a deterministic computer program.

Can convert to **deterministic** finite automaton (DFA).

- (Also called "finite state machine.")
- Like NFA, but has no ϵ -transitions and no symbol labels more than one transition from any given node.
- Easy to simulate on computer.
- There is an algorithm ("subset construction") that can convert **any** NFA to a DFA that accepts the same language.

Alternative approach: Simulate NFA directly by pretending to follow all possible paths "at once."

Tex has more details on both approaches.

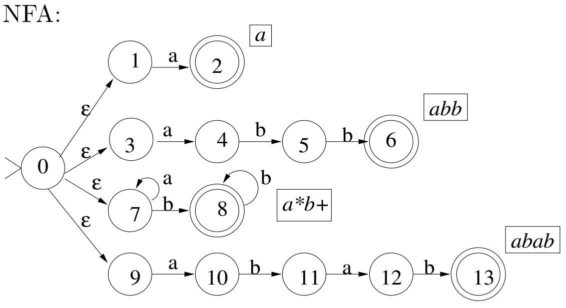
To handle "longest match" requirement, must keep track of last final state entered, and backtrack to that state ("un-reading" characters) if get stuck.

DFA and Backtracking Example

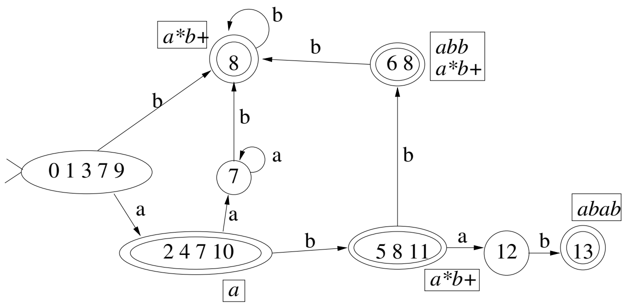
Given the following set of patterns, build a machine to find the longest match; in case of ties, favor the pattern listed first:

a
abb
a^*b^+
$abab$

NFA:



Corresponding DFA



Consider these inputs:

abaa

- Machine gets stuck after *aba* in state (12).
- Backs up to state (5 8 11).
- Pattern is a^*b^+
- Lexeme is *ab*; final *aa* will be read again.

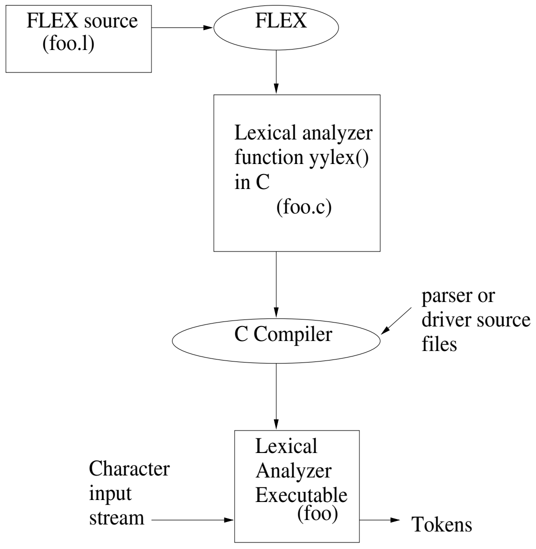
abba

- Machine stops after second *b* in state (6 8).
- Pattern is *abb* because it comes first in spec.

FLEX

`flex` is a **lexical analyzer generator**

- accepts **specification** of lexical analyzer.
- produces C program that **implements** specification.



`flex` is GNU version of original AT&T `lex`; many similar tools exist. Details of use may vary.

FLEX Specifications

Input to `flex` is given as sequence of rules, each consisting of a

- Pattern – regular expression (using ASCII as alphabet)
- Action – fragment of C code

When prefix of input matches a pattern, the generated analyzer executes the corresponding action.

Actions can make use of built-in global variables

- `yytext` contains lexeme
- `yyleng` contains length of `yytext`
- `yylineno` contains current line number (must use `-l` flag).

Example:

```
%%
integer       printf("found keyword INTEGER\n");
[0-9]+        printf("found number \n");
[A-Z][A-Z]*   printf("found ident %s\n, yytext);
[ \t\n]       /* ignore white space */ ;
```

As usual, if more than one pattern matches, the longest match is preferred; ties are broken in favor of rule that appears first.

FLEX Patterns and Actions

Patterns include literal text and meta-level operators.

Pattern	Matches
`x`	character "x"
`"x"`	character "x" even if it's an operator
`\x`	ditto
`[xy]`	"x" or "y"
`[x-y]`	characters between "x" and "y" inclusive
`^x`	any character but "x"
`.`	any character but "\n"
p?	an optional p
p*	zero or more p's
p+	one or more p's
p|q	p or q
()	grouping
{d}	substitute **definition** for d

Actions can be any valid C statement.

Ordinarily each action terminates with a statement `return` *t*; which causes `yylex()` to return with the token value *t*.

Otherwise, `yylex()` throws away the lexeme and continues searching for another pattern. This is suitable for handling white space. The simplest possible action is just the empty statement `;`.

Beware! Don't omit the action altogether; this makes `flex` generate a **default** action, which is to echo the lexeme to standard output. Seldom what you want!

Flex Definitions

The complete form of a flex specification is:

definitions
%%
rules
%%
arbitrary user code

Definitions abbreviate regular expressions for later use in patterns, e.g.,

```
LETTERS    [a-zA-Z_]
DIGITS     [0-9]
%%
{LETTERS}({LETTERS}|{DIGITS})*   return IDTOK;
```

User code is just copied directly to the generated `.c` file; it can contain functions and globals to be invoked from the actions.

Such code can also be included in the definitions section if enclosed between `%{` and `%}`.

Start Conditions

`flex` permits multiple sets of rules to coexist in the same specification. Each set of rules is called a **start condition**.

Rules prefixed with <*name*> are recognized (only) when `yylex()` is in the start condition *name*.

When `yylex()` starts running it is in the start condition with the predefined name `INITIAL`.

You declare new start condition names in a `%S` line in the definitions section of the spec. You put `yylex()` into the start condition *name* by including the pseudo-statement

```
BEGIN name;
```

in an action.

Example: multi-line comments in C.

```
%S COMMENT
%%
<INITIAL>"/*"          BEGIN COMMENT;
<COMMENT>"*/"          BEGIN INITIAL;
<COMMENT>.|"\n"        /* ignore comments */ ;
<INITIAL> ...           ordinary rules follow
<INITIAL> ...
```