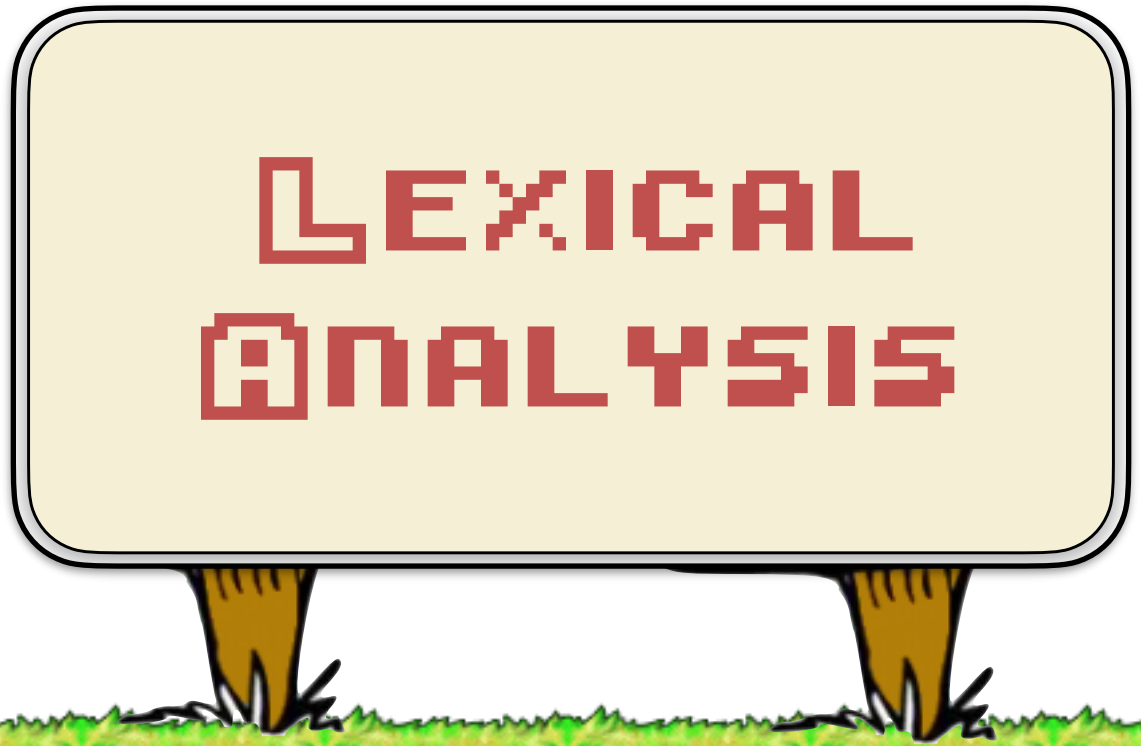
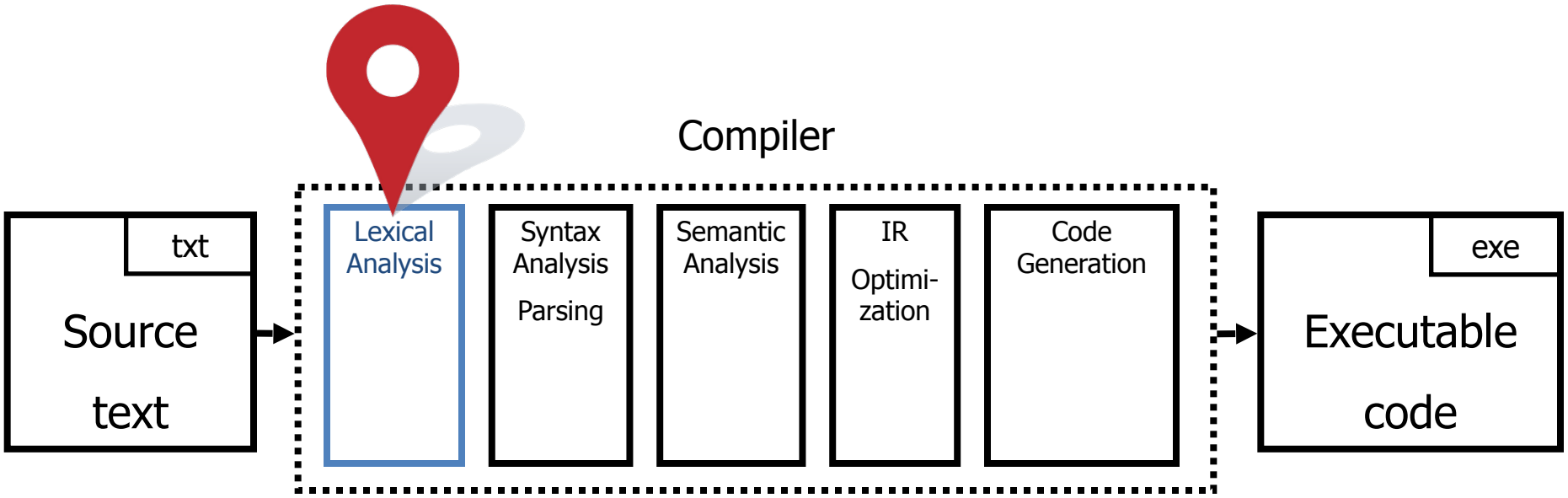
Theory of Compilation

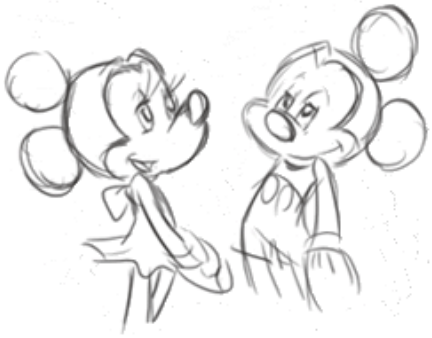
Lecture 01



You are here


Compiler
txt
Source text
Lexical Analysis
Syntax Analysis
Parsing
Semantic Analysis
IR
Optimi-zation
Code Generation
exe
Executable code

From Characters to Tokens



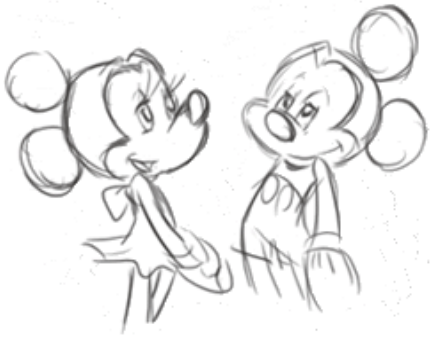
txt

```
x = b*b - 4*a*c
```

Token Stream

ID	EQ	ID	MULT	ID	MINUS	INT	MULT	ID	MULT	ID
name: "x"		name: "b"		name: "b"		value: 4		name: "a"		name: "c"

From Characters to Tokens



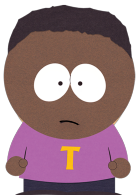
- What is a token?
  - Roughly – a "word" in the source language

Identifiers | Values | Keywords

  - Really – anything that should appear in the input to syntax analysis as a single unit
- A token has:
  - a kind
  - lexical attributes

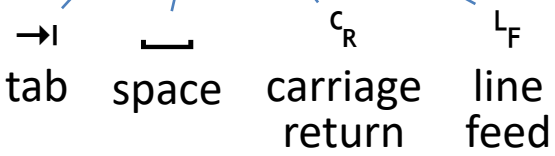
Example Tokens

Kind	Examples
ID	`x`, `y`, `z0`, `foo`, `bar`
NUM	`42`
FLOATNUM	`3.141592654`
STRING	`"so long, and thanks for all the fish"`
IF	`if`
LPAREN	`(`
RPAREN	`)`
MINUS	`-`



Strings with Special Handling

Kind	Examples
Comments	`/* Ceci n'est pas un commentaire */`
Preprocessor directives	`#include <foo.h>`
Macros	`#define THE_ANSWER 42`
White spaces	`\t \r\n`



✹ Note. these are *not* tokens, the are recognized, handled, and then discarded.

Some Basic Terminology

- **Lexeme** (aka symbol) – a series of characters separated from the rest of the program according to a convention (space, semicolon, word boundary, ...)

- **Pattern** – a rule specifying a set of strings.
  Example: "*a string that starts with a letter, followed by letters and digits*"

- **Token** – a pair of ⟨kind, attributes⟩

How can we define tokens?

- Keywords — easy!
  - `if`, `then`, `else`, `for`, `while`, …

- Identifiers? `x0`, `b2b`, `heIsNoOne`, …
- Numerical Values? `0`, `1`, `2`, `3.4`, `5.67`, …
- Strings? `"there are infinitely many"`, …

- Characterize **unbounded** sets of values using a **bounded** description?

Regular Expressions

Σ = alphabet (set of characters)

pro tip
to include newlines:
`.|[\r\n]`
or `[^]`

common shorthands:
`[a-z]` *(range)*
`[^xyz]` *(complement)*

- Basic Patterns

`x`	A single letter 'x' from the alphabet Σ
`.`	Any character (*usually excluding new-line characters*)
`[xyz]`	Any of the characters x,y,z

- Repetition Operators

`R?`	Either an R or nothing (= *optionally* R)
`R*`	Zero or more occurrences of R
`R+`	One or more occurrences of R

- Composition Operators

R_1R_2	An R_1 followed by R_2
$R_1 \mid R_2$	Either an R_1 or R_2

- Grouping

`(R)`	Same as R (*used to override default precedence*)

Highest precedence

Lowest precedence

More Examples

- if = `if`
- then = `then`
- word = `[A-Za-z][a-z]*`

- digit = [0-9]
- digits = digit+

useful shorthands: once we have defined "digit", *we can use it as a macro*

oops

int = (+|–)?digit+

int = [+–]?digit+

oops
(range separator)

Nit: Escape characters

- What is the expression for one or more “+” symbols?

✗ `(+)+` ✓ `(\+)+`

- backslash `\` before any operator turns it into a standard character
  - `\*`, `\?`, `\+`, `\[`, …

More Examples

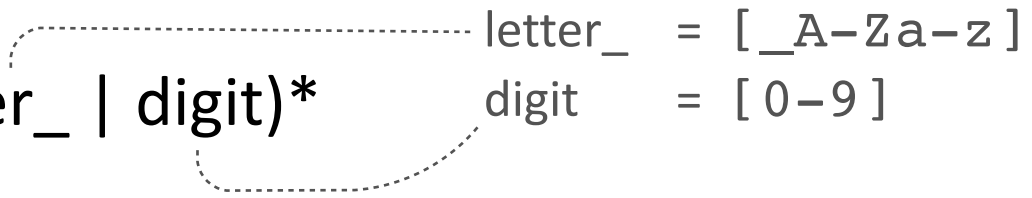
- if = `if`
- then = `then`
- word = `[A-Za-z][a-z]*`

- digit = `[0-9]`
- digits = digit+

- ?? = digits (`\.` digits)? (`e` (`\+|-`)? digits)?

Ambiguity

- if = `if`
- id = letter_ (letter_ | digit)*



- "`if`" matches both the pattern for reserved words and the pattern for identifiers… so what should it be?
- How about the identifier "`iffy`"?

- Solution
  - Always find longest matching token
  - Break ties using order of definitions; first definition wins

    (⇒ *tip: list rules for keywords before identifiers*)

Time for some action

And now, let the *real fun* begin!

- Generate a lexical analyzer *automatically* from token definitions

- Main idea
  - Use finite-state automata to match regular expressions

Construction

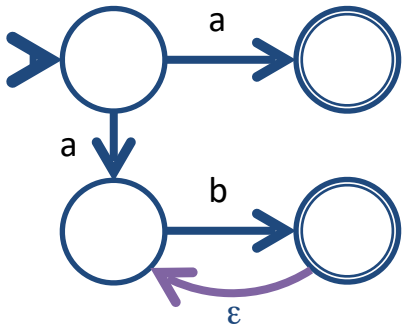
txt

```
x = b*b - 4*a*c
```

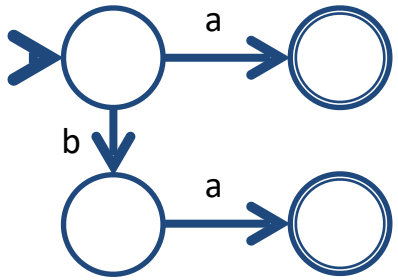
Token Definitions

```
IF :   if
TH :   then
REL :  [<>]=?
       | == | <>
INT :  [0-9]+
```

Non-deterministic Finite Automaton



Deterministic Finite Automaton

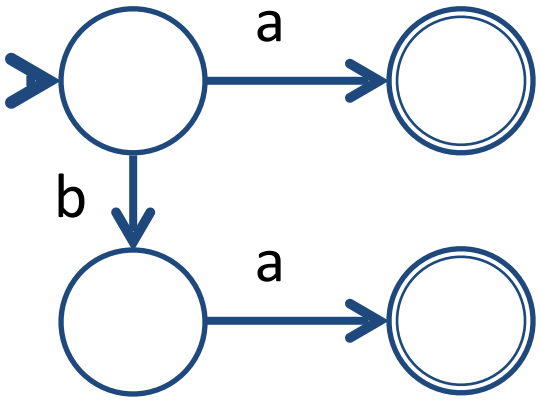


Token Stream

⟨ID,“**x**”⟩ ⟨EQ⟩ ⟨ID,“**b**”⟩ ⟨MULT⟩ ⟨ID,“**b**”⟩ ⟨MINUS⟩
⟨INT,**4**⟩ ⟨MULT⟩ ⟨ID,“**a**”⟩ ⟨MULT⟩ ⟨ID,“**c**”⟩

Reminder: Finite-State Automata

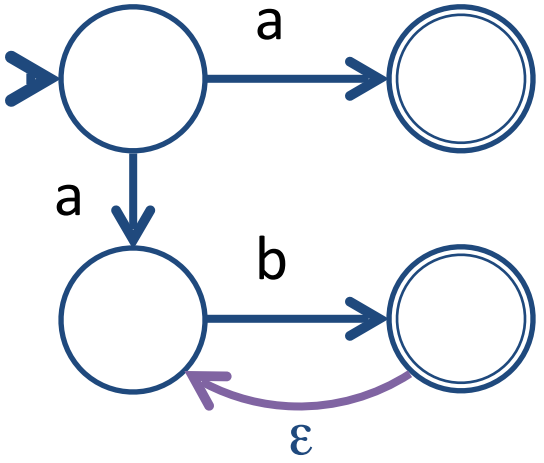
Deterministic automaton



- $M = (\Sigma, Q, \delta, q_0, F)$
  - Σ – alphabet
  - Q – finite set of state
  - $q_0 \in Q$ – initial state
  - $F \subseteq Q$ – final states
  - $\delta : Q \times \Sigma \rightarrow Q$ – transition function

Non-Deterministic automaton

- $M = (\Sigma, Q, \delta, q_0, F)$
  - Σ - alphabet
  - Q – finite set of state
  - $q_0 \in Q$ – initial state
  - $F \subseteq Q$ – final states
  - $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \rightarrow 2^Q$ – transition function



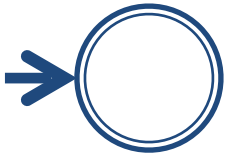
- Allows ε-transitions
- For a word *w*, M can reach ***a set of*** states or ***get stuck***. If some state reached is final, M accepts *w*.

From Regular Expressions to NFA

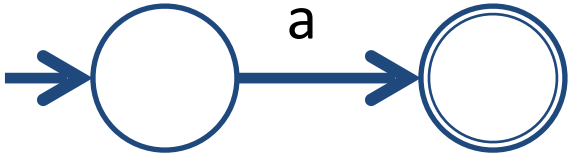
- Step 1: assign expression names and obtain pure regular expressions $R_1 \ldots R_m$
- Step 2: construct an NFA M_i for each regular expression R_i
- Step 3: combine all M_i into a single NFA
  - Ambiguity resolution: prefer ***longest*** accepting word

Basic constructs

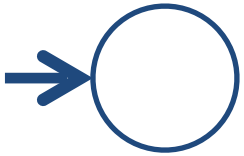
R = ε



R = a

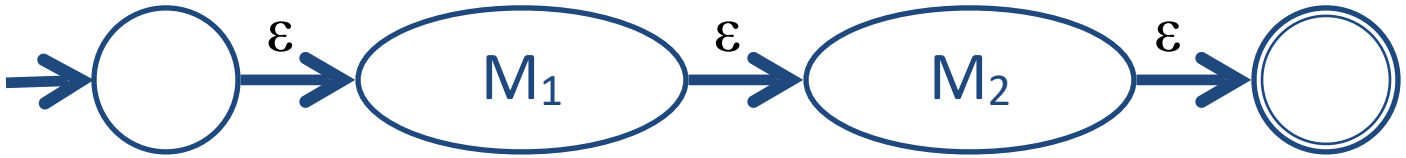


R = ∅

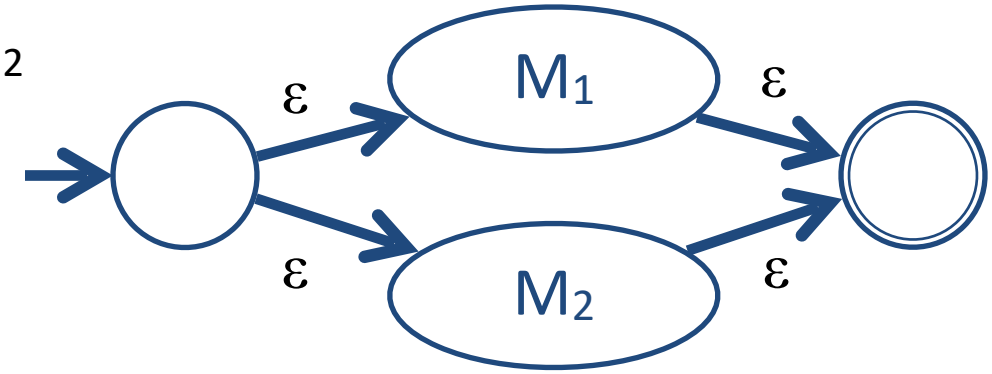


Composition

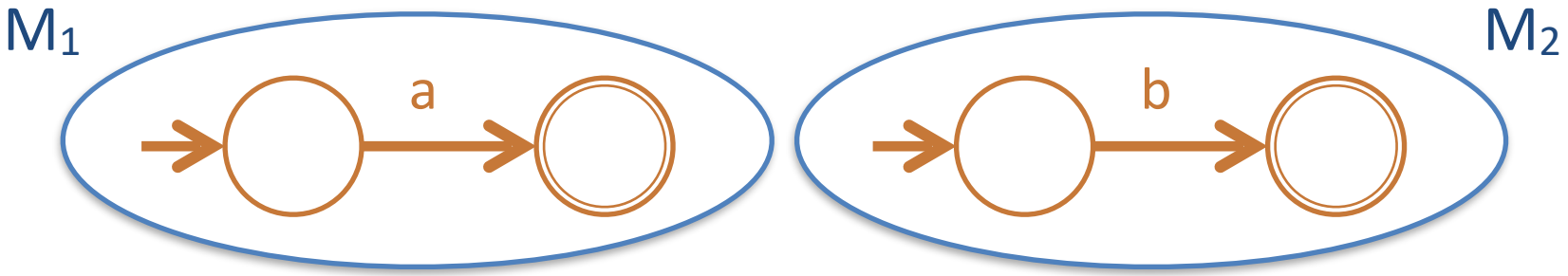
$R = R_1 R_2$



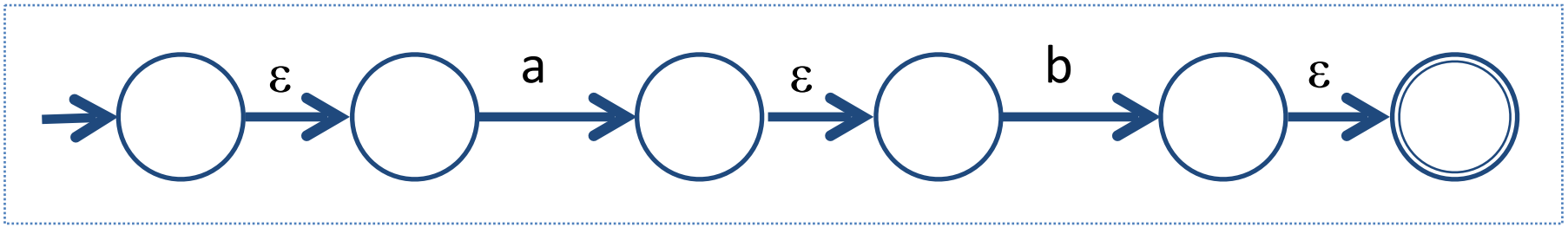
$R = R_1 \mid R_2$



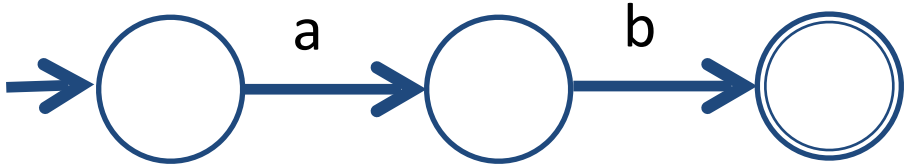
Composition



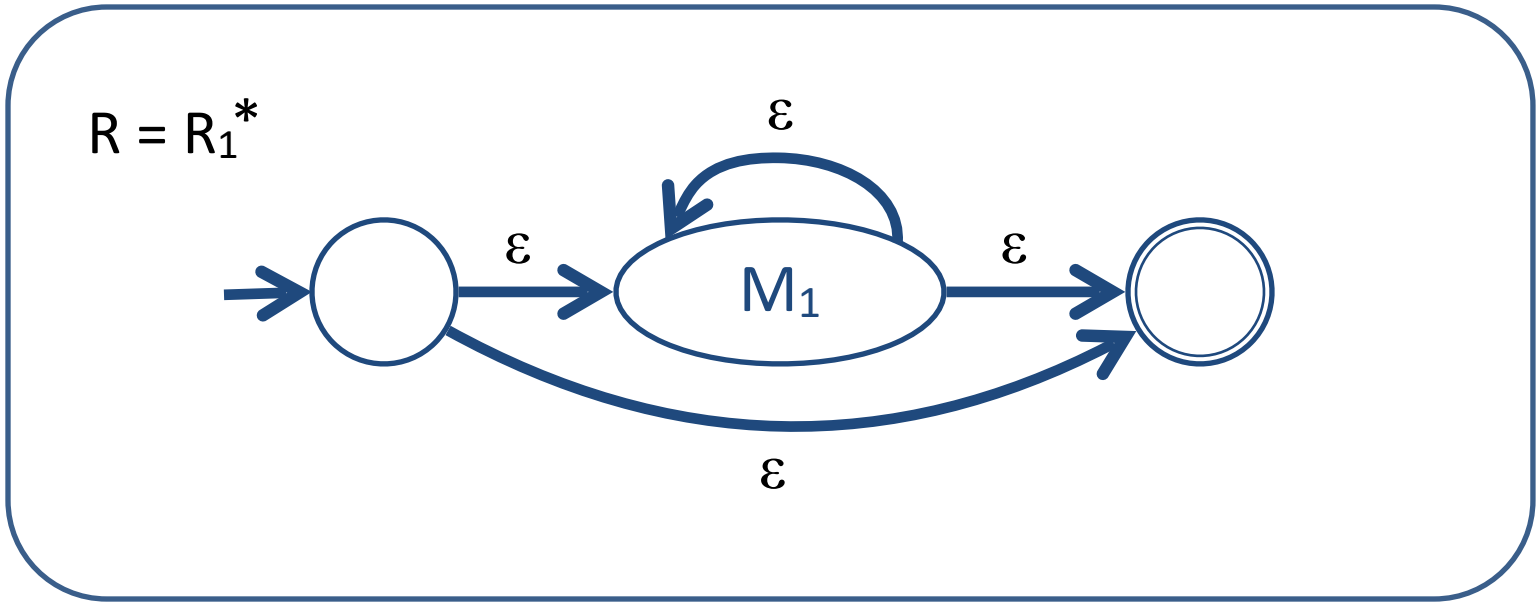
ab



minimize:

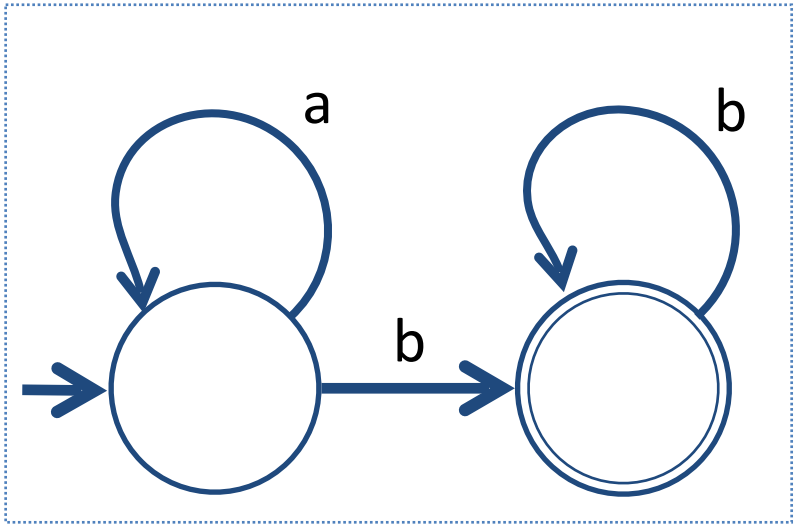


Repetition



$R_1+ = R_1R_1^*$

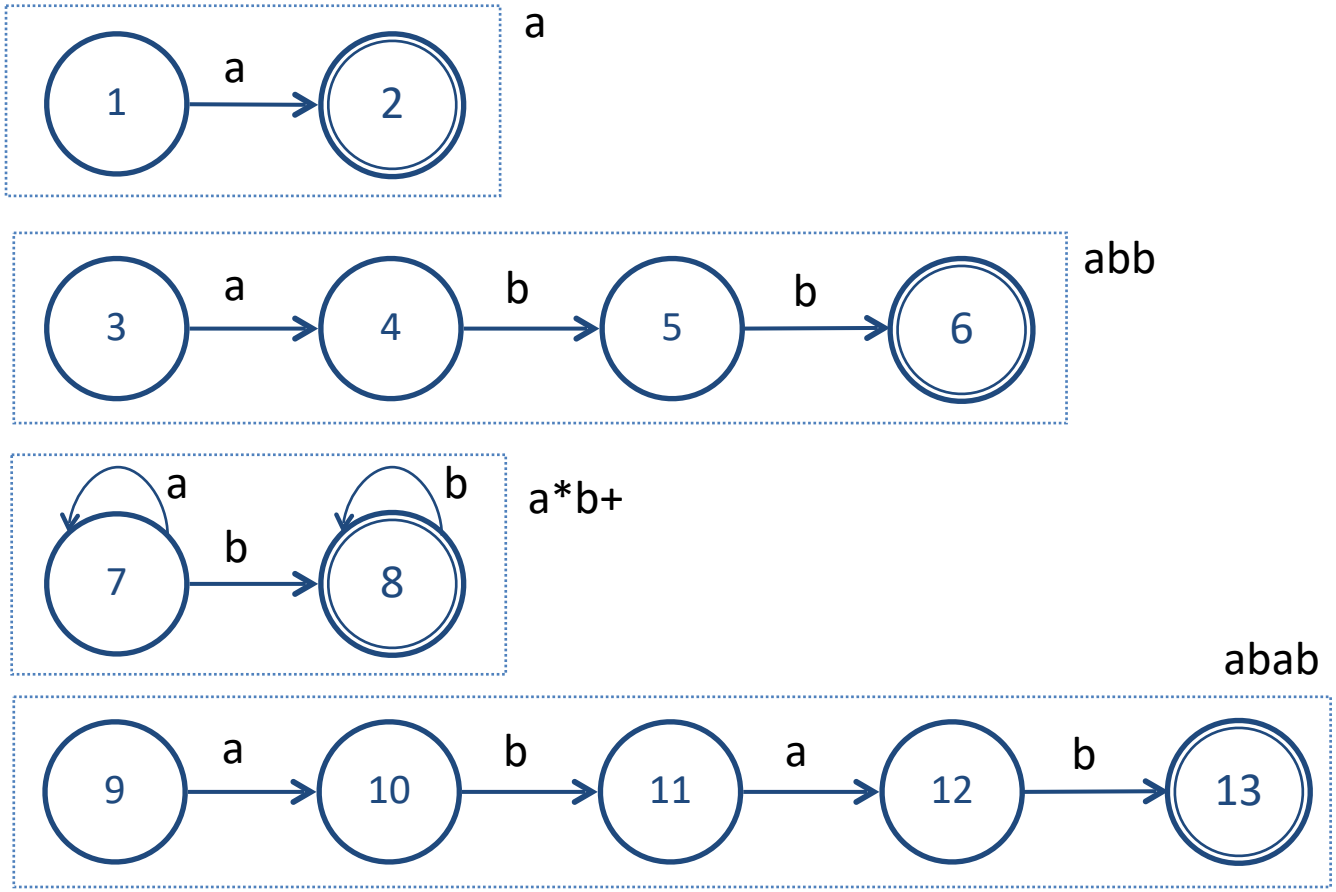
Repetition



a*b+

One Automaton per Pattern

a
abb
a*b+
abab



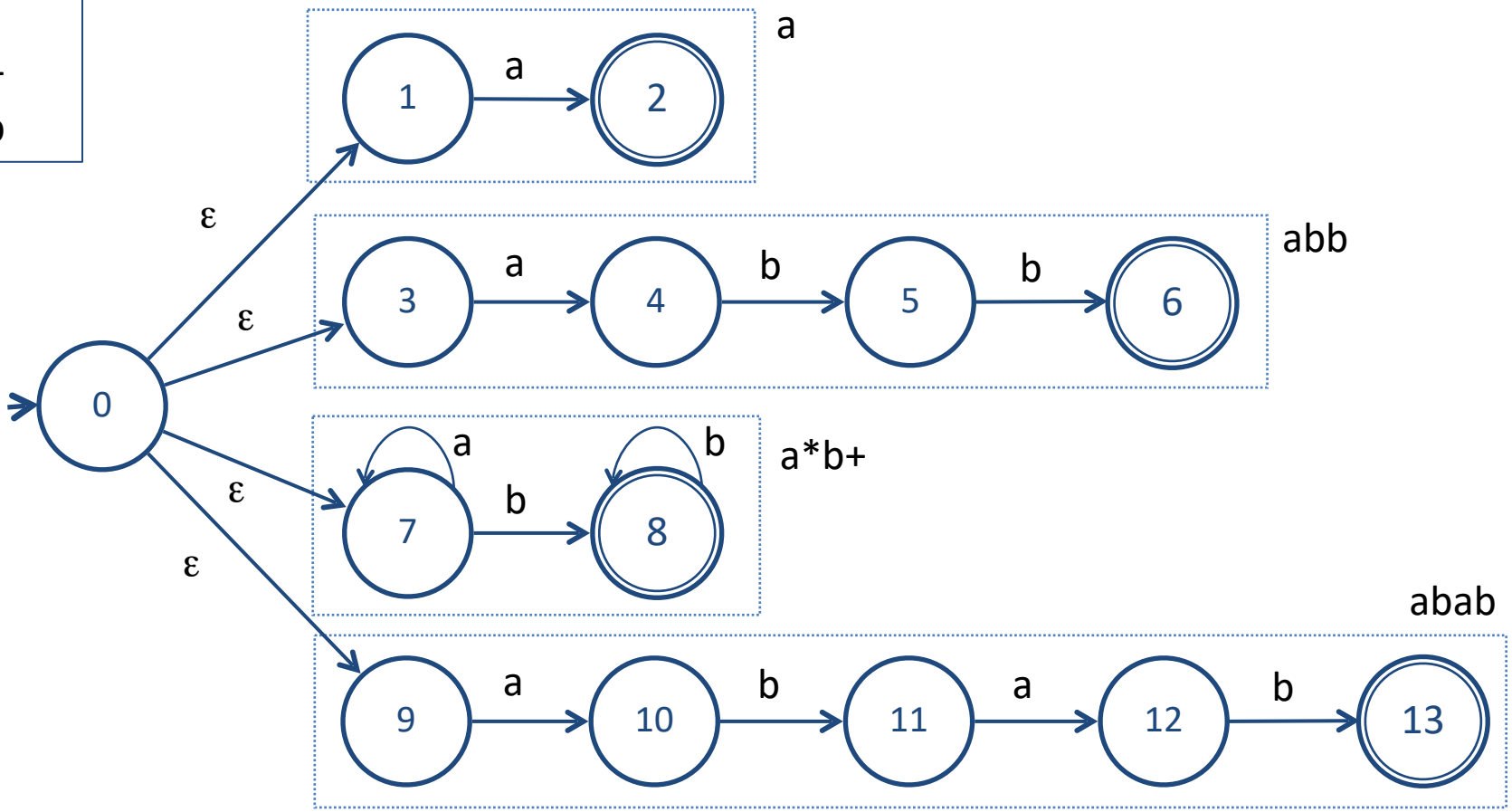
What now?

- We have one automaton per pattern.
  Naïve approach: try each automaton separately
- Given a word *w*:
  - Try $M_1(w)$
  - Try $M_2(w)$
  - …
  - Try $M_n(w)$
- Requires “rewinding” after every attempt

Combine Automata

combines

a
abb
a*b+
abab

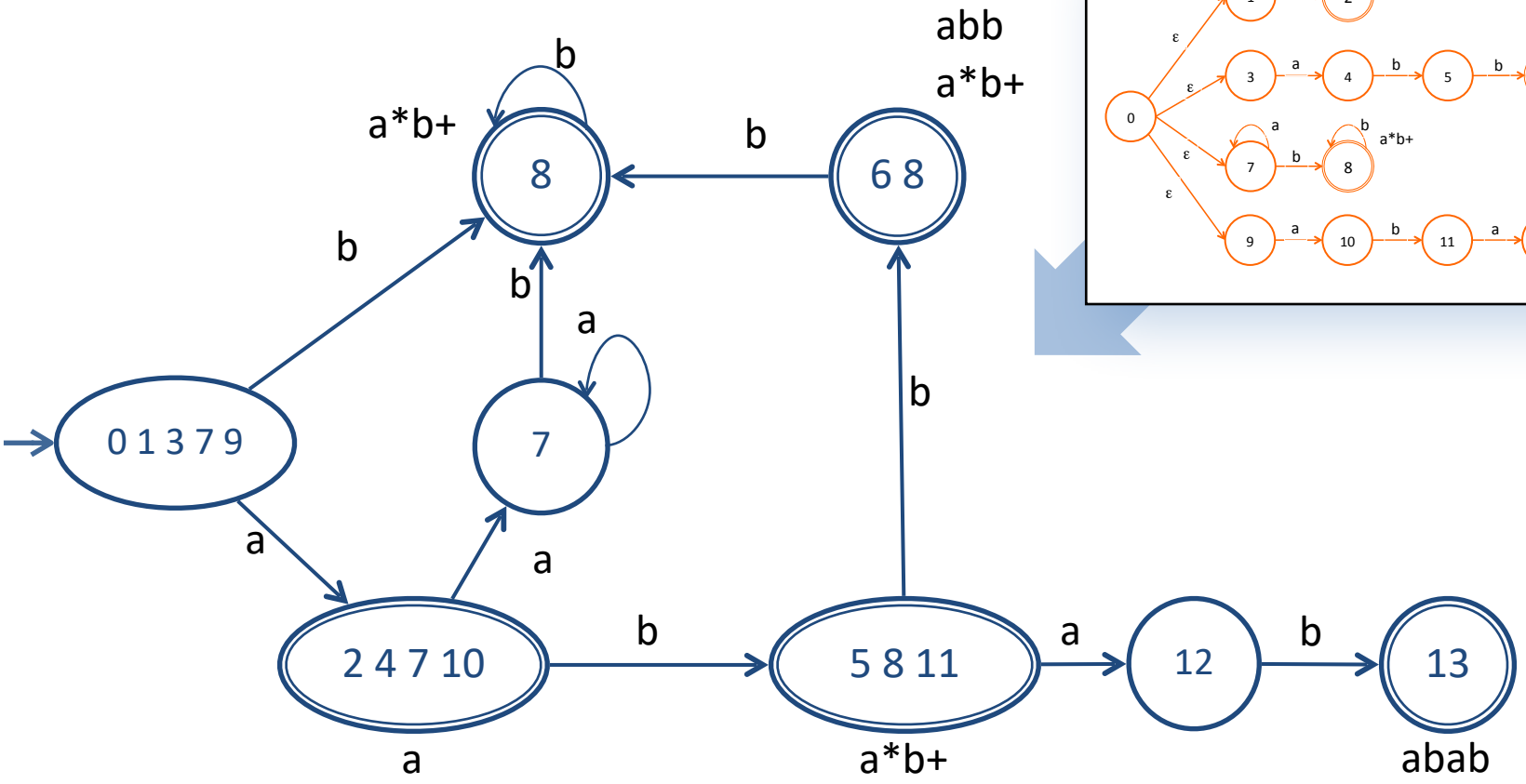


Ambiguity resolution

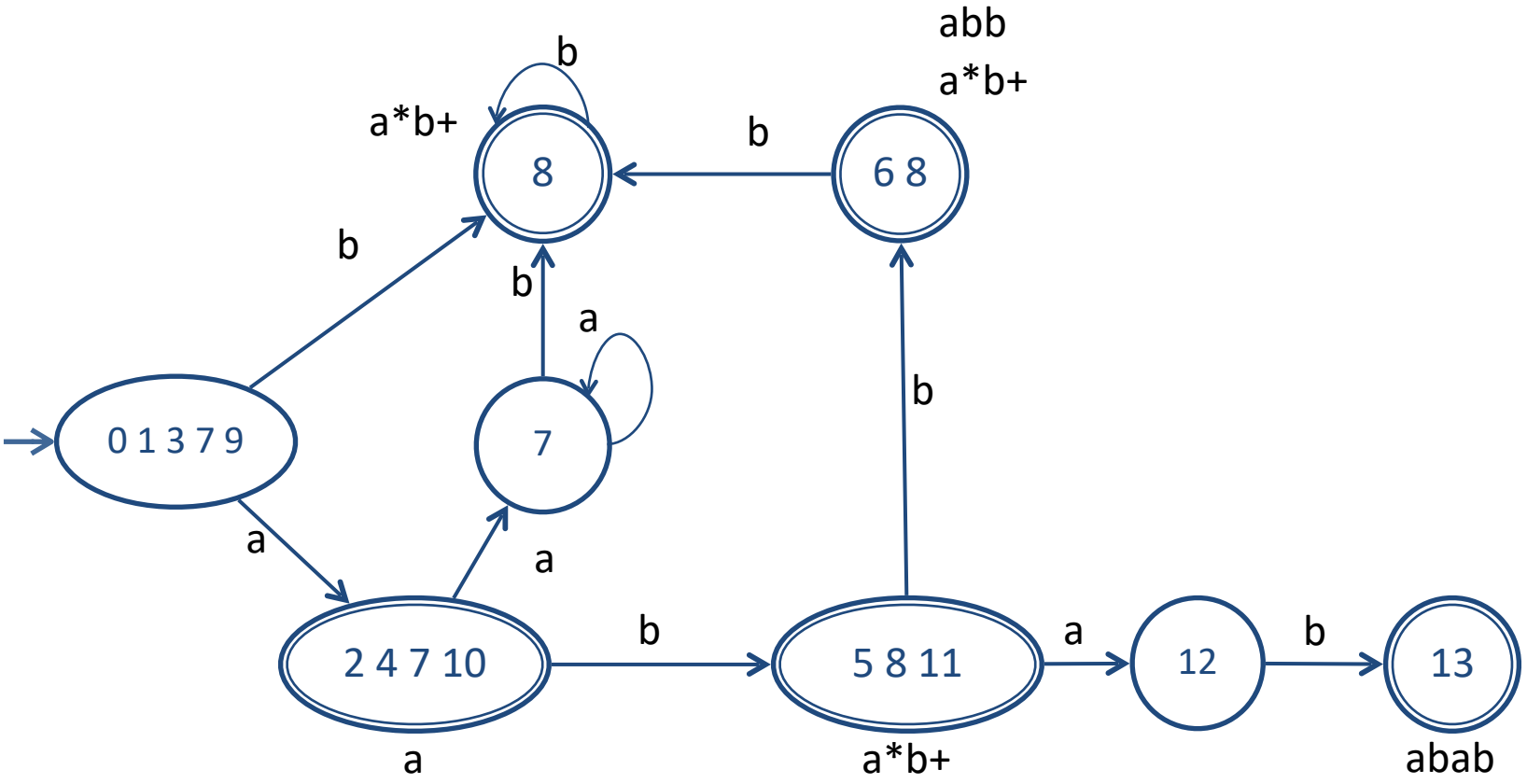
- Recall…
  - ✳ Longest word
  - ✳ Tie-breaker based on order of rules when words have same length

- Recipe
  - Turn NFA to DFA
  - Run until stuck, remember ***last accepting state*** — this is the token to be returned

Corresponding DFA



Example Inputs

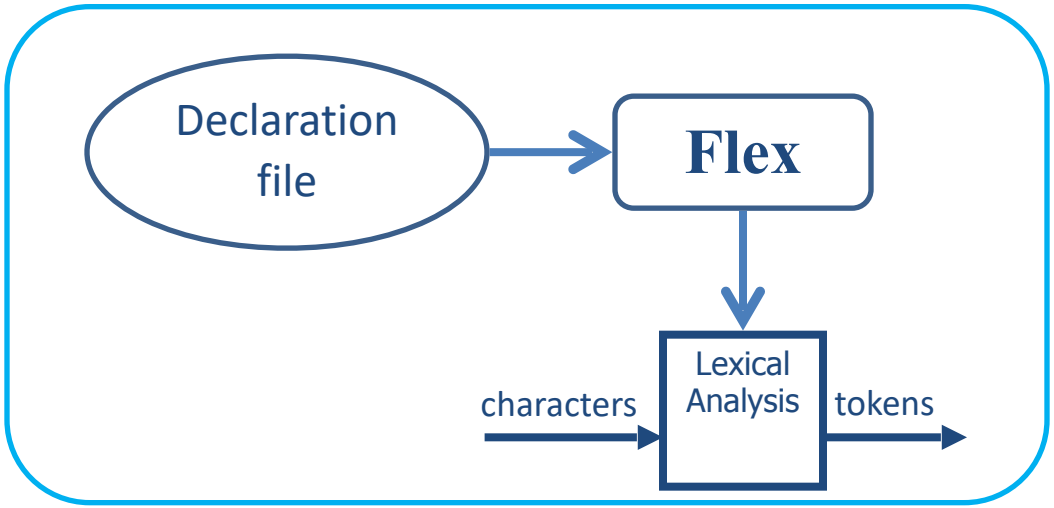


abaa: gets stuck after **aba** in state 12, backs up to state (5 8 11) pattern is a*b+, token is **ab**

abba: stops after second **b** in (6 8), pattern is abb because it comes first in spec

Good News

- Construction is done automatically by common tools
- **Flex** is your friend
  - Automatically generates a lexical analyzer from declaration file
- Advantages: short declaration file, easily checked, easily modified and maintained



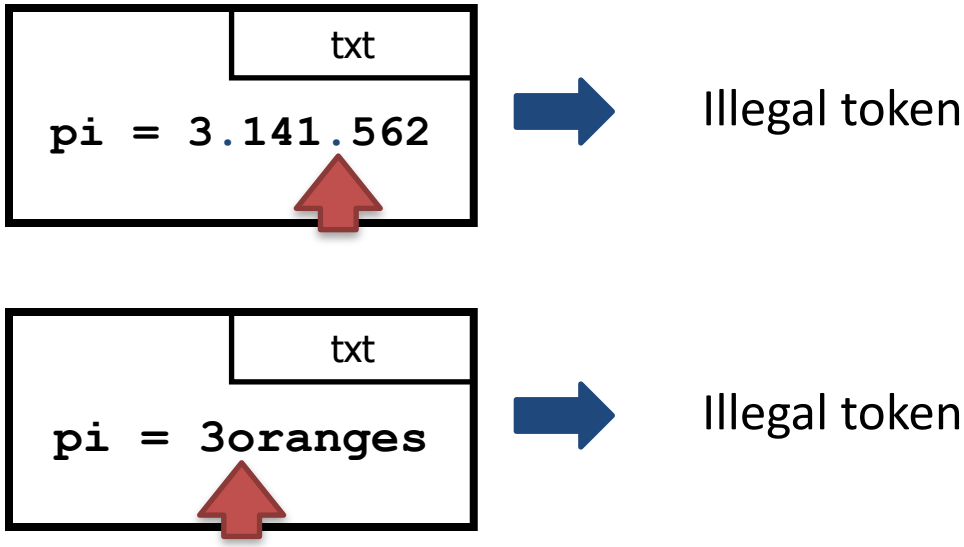
Intuitively:

- **Flex** builds DFA table
- Analyzer simulates (runs) the DFA on a given input

Flex declarations file

```
%{
#include <math.h>
int line_number = 1;
%}

WS      [ \t]
LETTER  [a-zA-Z]
DIGIT   [0-9]
ID      {LETTER}({LETTER}|{DIGIT})*

%%

{DIGIT}+     { printf("number: %d\n", atoi(yytext)); return 1; }
{ID}         { return 2;}
{WS}         { /* ignore */ }
\n           { line_number++; /* and ignore */ }
.            { return -1; /* ERROR */ }

%%

int main() { return yylex(); }
```

Errors in Lexical Analysis



Note: How can we make this an error?

Error Handling

- Sometimes the lexeme does not match any pattern
  - Instead of aborting, we try to recover
  - Easiest fix: skip characters until the beginning of a “legitimate” lexeme
  - Alternatives: eliminate/add/replace one letter, replace order of two adjacent letters, etc.

#@!?&

👍 *Pro*: allow the compilation to continue

👎 *Con*: errors that spread all over

Summary

- **Lexical analyzer**
  - ✓ Turns character stream into token stream
  - ✓ Tokens defined using regular expressions
  - ✓ Regex ⇾ NFA ⇾ DFA construction for identifying tokens
  - ✓ Automated constructions of lexical analyzer using **Flex**

Coming Up