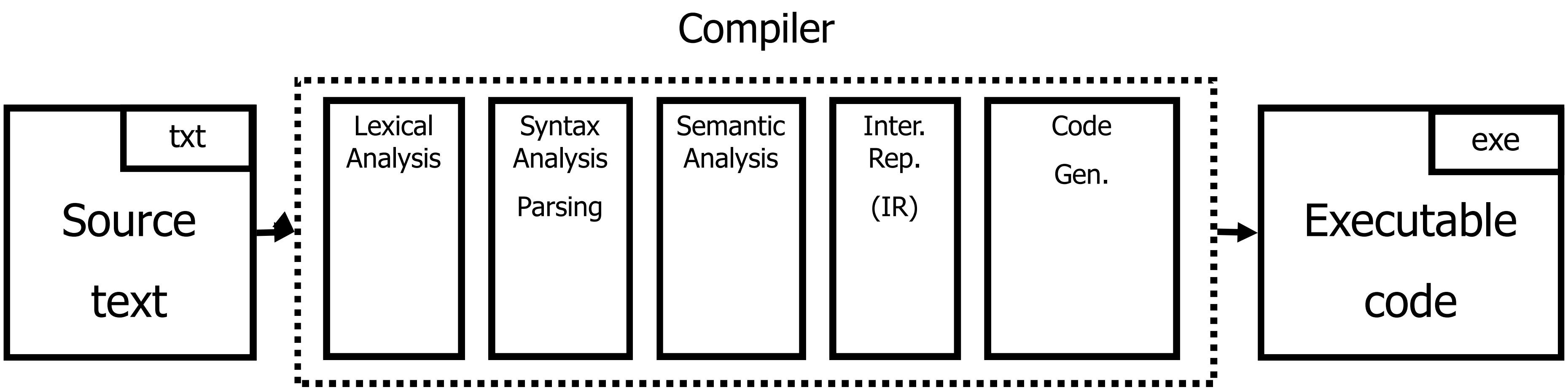
Lecture 02 – Lexical Analysis

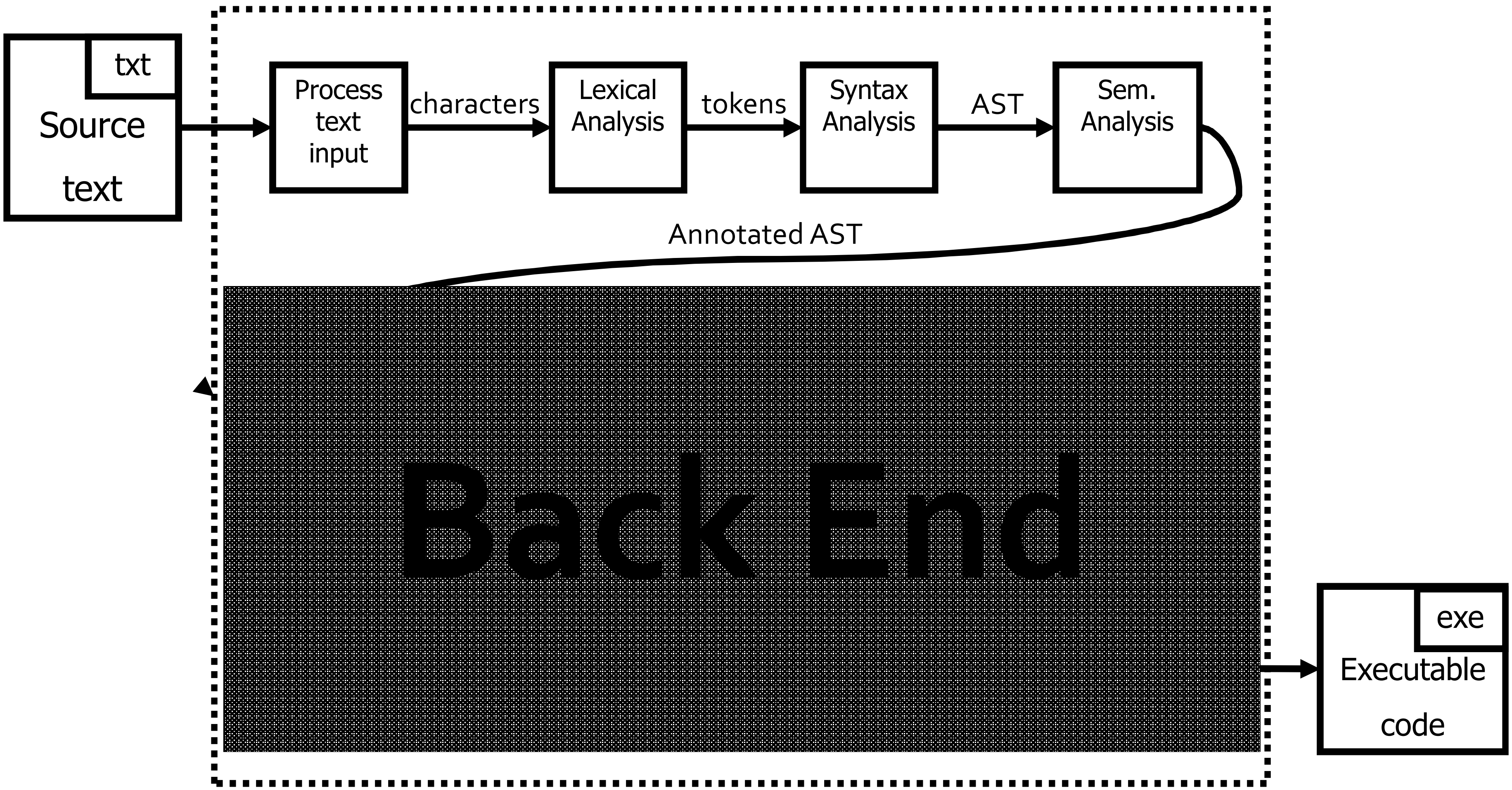
THEORY OF COMPILATION

Eran Yahav

You are here


Compiler
txt
Source
text
Lexical
Analysis
Syntax
Analysis
Parsing
Semantic
Analysis
Inter.
Rep.
(IR)
Code
Gen.
exe
Executable
code

You are here…


txt
Source text
Process text input
characters
Lexical Analysis
tokens
Syntax Analysis
AST
Sem. Analysis
Annotated AST
Back End
exe
Executable code

From characters to tokens

- What is a token?
  - Roughly – a “word” in the source language
  - Identifiers
  - Values
  - Language keywords
  - Really - anything that should appear in the input to syntax analysis
- Technically
  - Usually a pair of (kind,value)

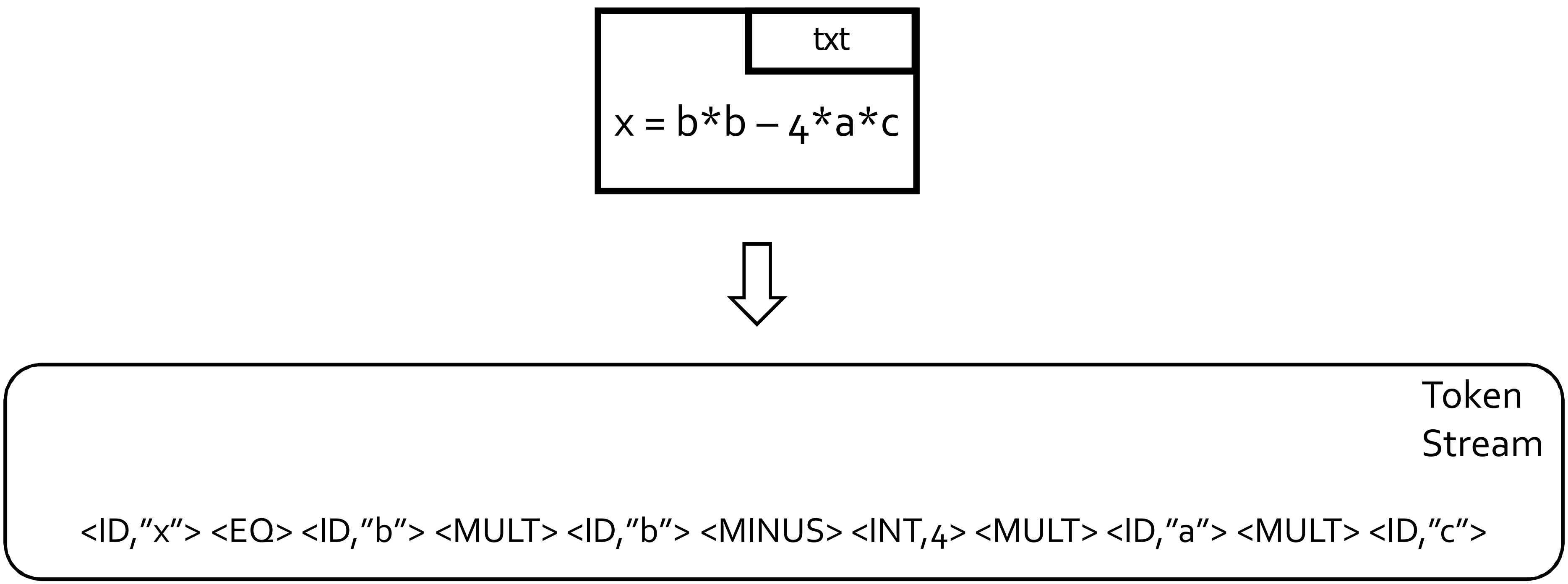
Example Tokens

Type	Examples
Identifier	x, y, z, foo, bar
NUM	42
FLOATNUM	3.141592654
STRING	“so long, and thanks for all the fish”
LPAREN	(
RPAREN	)
IF	if
…	

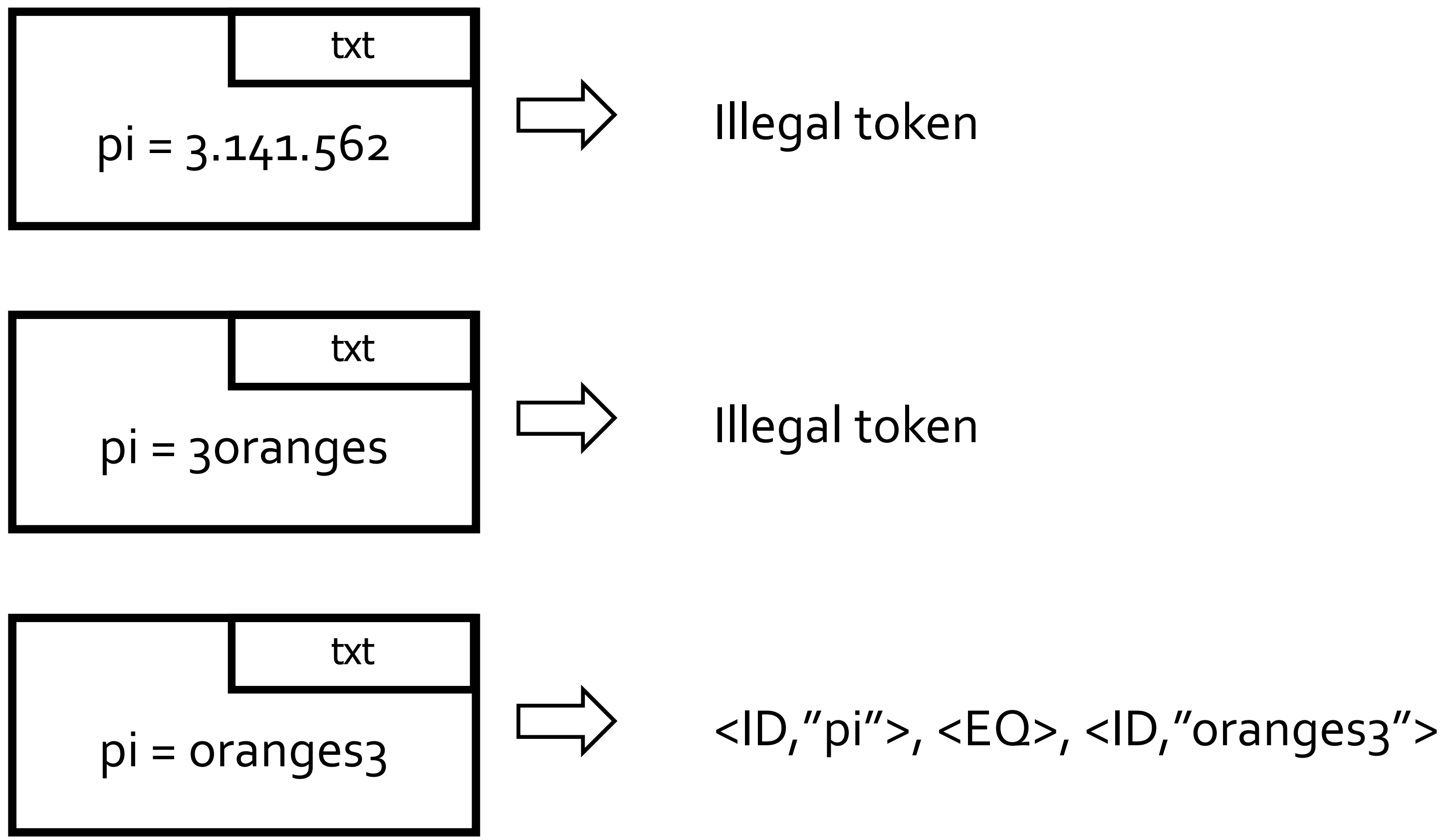
Strings with special handling

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

From characters to tokens



Errors in lexical analysis



How can we define tokens?

- Keywords – easy!
  - if, then, else, for, while, …

- Identifiers?
- Numerical Values?
- Strings?

- Characterize unbounded sets of values using a bounded description?

Regular Expressions

Basic Patterns	Matching
x	The character x
.	Any character, usually except a new line
[xyz]	Any of the characters x,y,z
Repetition Operators	
R?	An R or nothing (=optionally an R)
R*	Zero or more occurrences of R
R+	One or more occurrences of R
Composition Operators	
R1R2	An R1 followed by R2
R1|R2	Either an R1 or R2
Grouping	
(R)	R itself

Examples

- ab*|cd? =
- (a|b)* =
- (0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9)* =

Escape characters

- What is the expression for one or more + symbols?
  - (+)+ won't work
  - (\+)+ will
- backslash \ before an operator turns it to standard character
- *, \?, \+, …

Shorthands

- Use names for expressions
  - letter = a | b | … | z | A | B | … | Z
  - letter_ = letter | _
  - digit = 0 | 1 | 2 | … | 9
  - id = letter_ (letter_ | digit)*
- Use hyphen to denote a range
  - letter = a-z | A-Z
  - digit = 0-9

Examples

- digit = 0-9
- digits = digit+
- number = digits (Є | .digits (Є | e (Є|+|-) digits))
- if = if
- then = then
- relop = < | > | <= | >= | = | <>

Ambiguity

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

- "if" is a valid word in the language of 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 (=> list rules for keywords before identifiers)

Creating a lexical analyzer

- Input
  - List of token definitions (pattern name, regex)
  - String to be analyzed
- Output
  - List of tokens

- How do we build an analyzer?

Character classification

```
#define is_end_of_input(ch) ((ch) == '\0');
#define is_uc_letter(ch) ('A'<= (ch) && (ch) <= 'Z')
#define is_lc_letter(ch) ('a'<= (ch) && (ch) <= 'z')
#define is_letter(ch) (is_uc_letter(ch) || is_lc_letter(ch))
#define is_digit(ch) ('0'<= (ch) && (ch) <= '9')
…
```

Main reading routine

```
void get_next_token() {
do {
  char c  = getchar();
  switch(c) {
    case is_letter(c) : return recognize_identifier(c);
    case is_digit(c) : return recognize_number(c);
     …
} while (c != EOF);
```

But we have a much better way!

- Generate a lexical analyzer automatically from token definitions

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

Reminder: Finite-State Automaton

- 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

Reminder: Finite-State Automaton

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

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

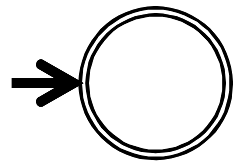
From regular expressions to NFA

- Step 1: assign expression names and obtain pure regular expressions R1…Rm
- Step 2: construct an NFA Mi for each regular expression Ri
- Step 3: combine all Mi into a single NFA

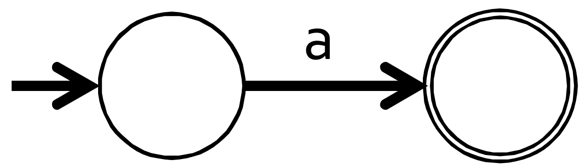
- Ambiguity resolution: prefer longest accepting word

Basic constructs

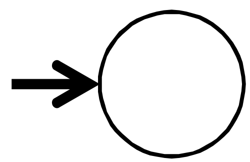
R = ε



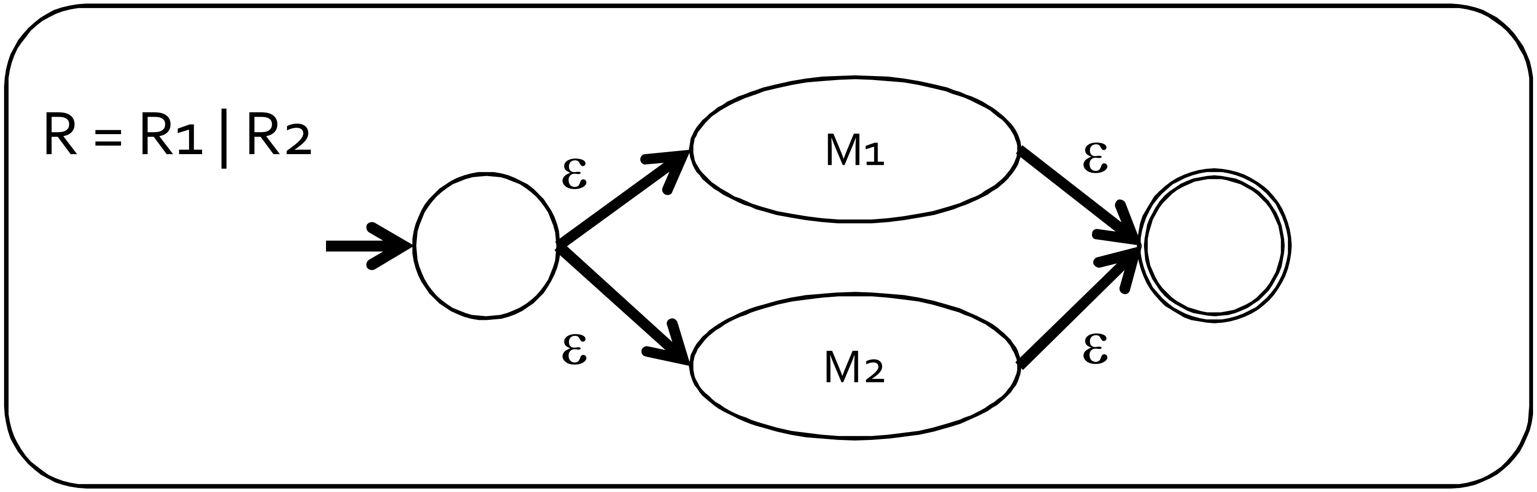
R = a

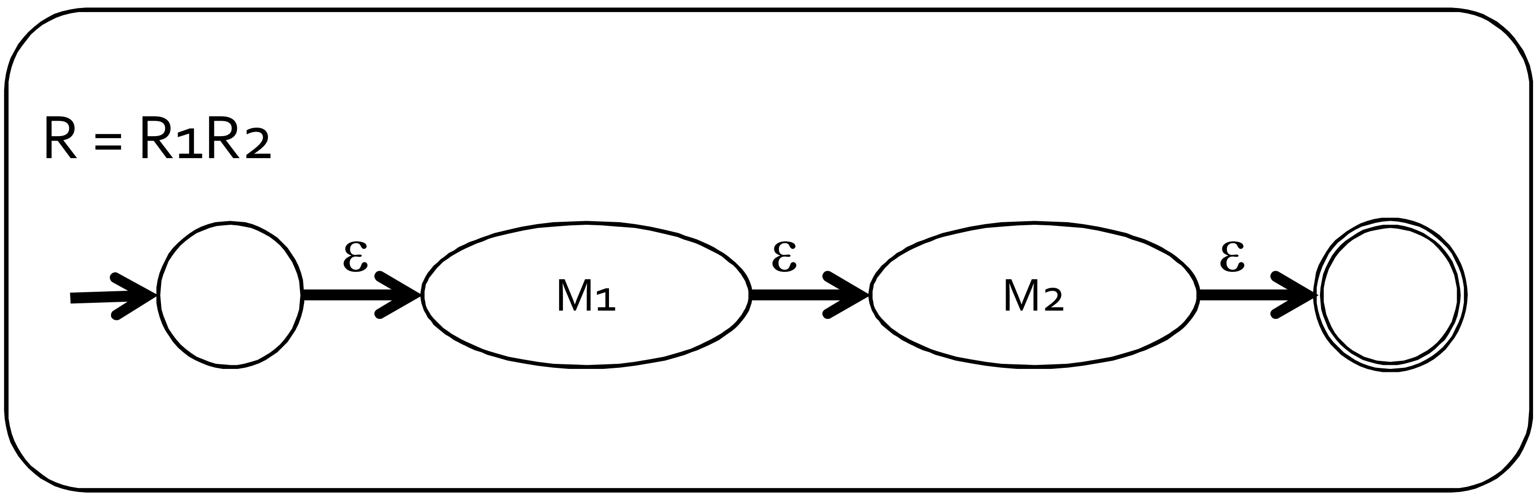


R = ϕ

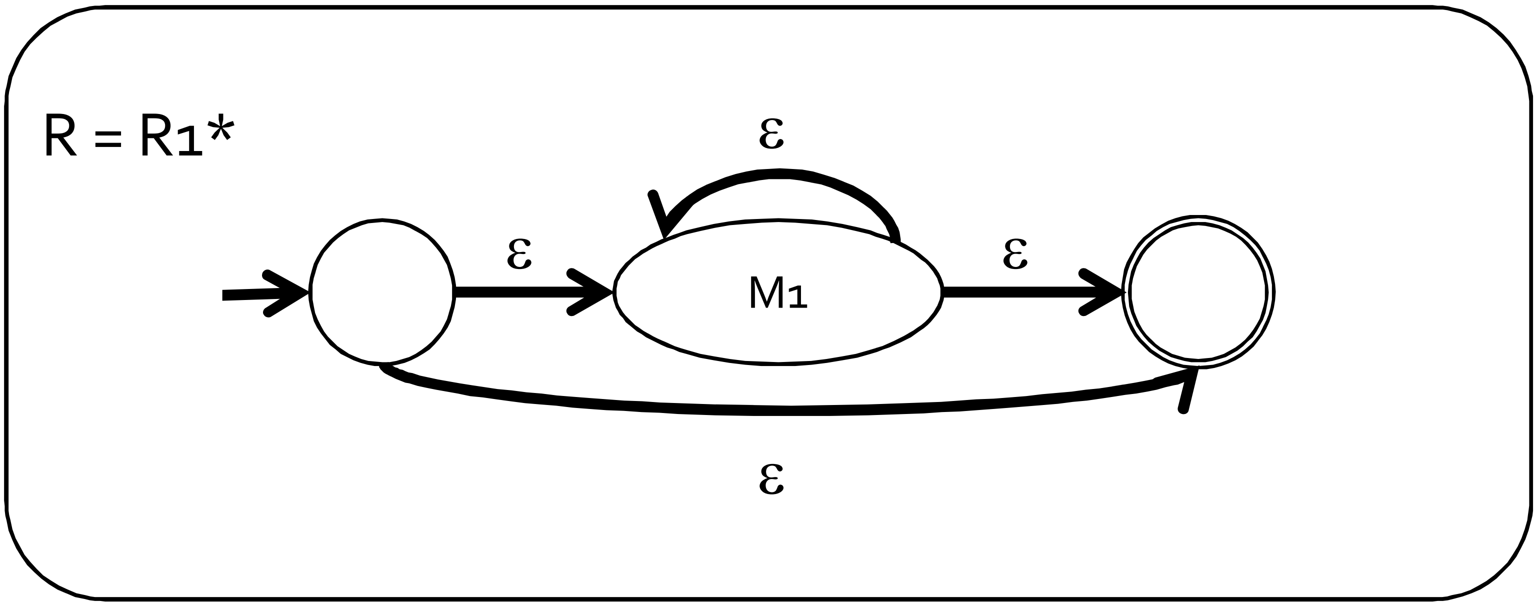


Composition





Repetition



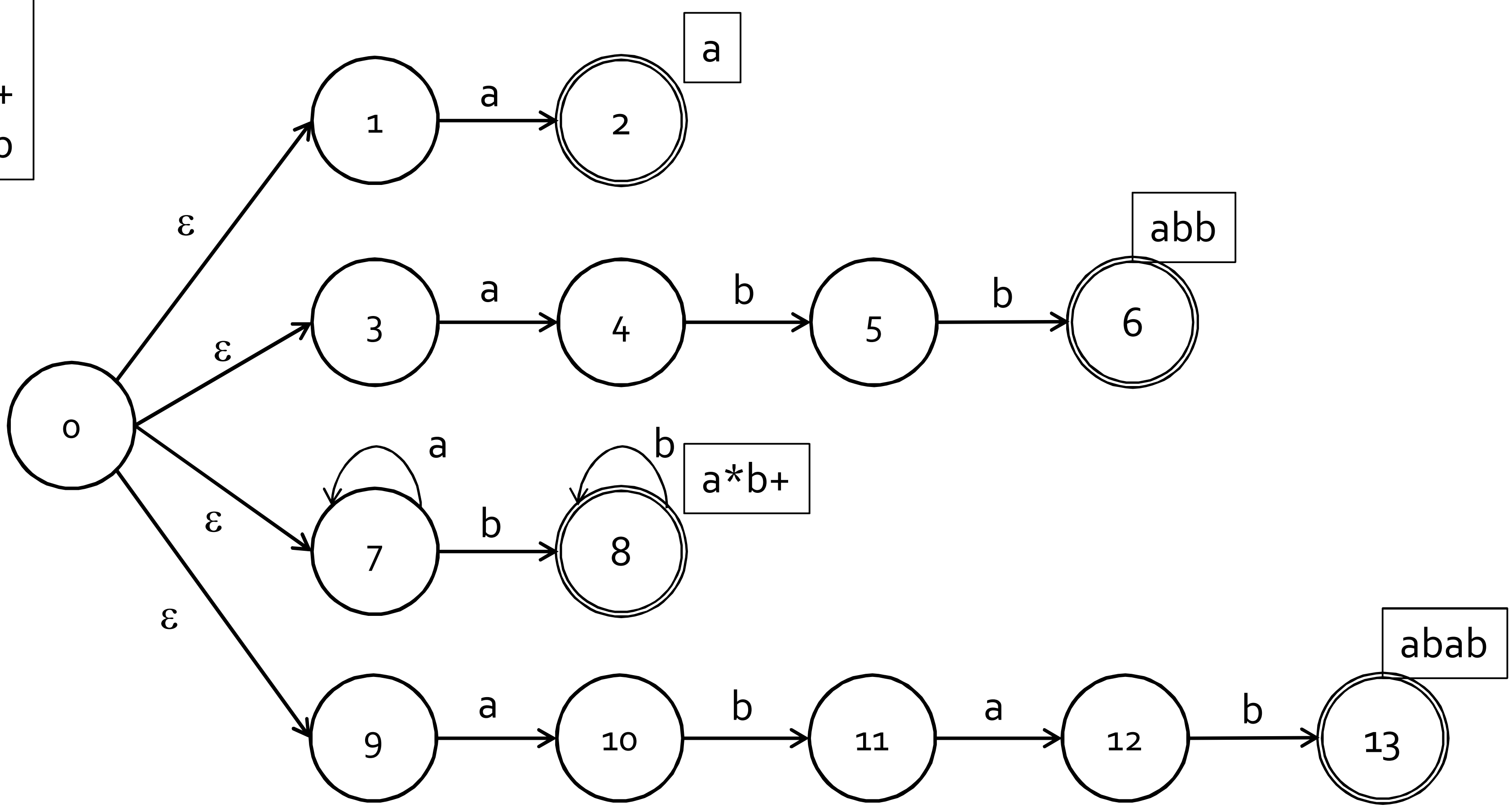
What now?

- Naïve approach: try each automaton separately
- Given a word w:
  - Try M1(w)
  - Try M2(w)
  - …
  - Try Mn(w)

- Requires resetting after every attempt

Combine automata

a
abb
a*b+
abab

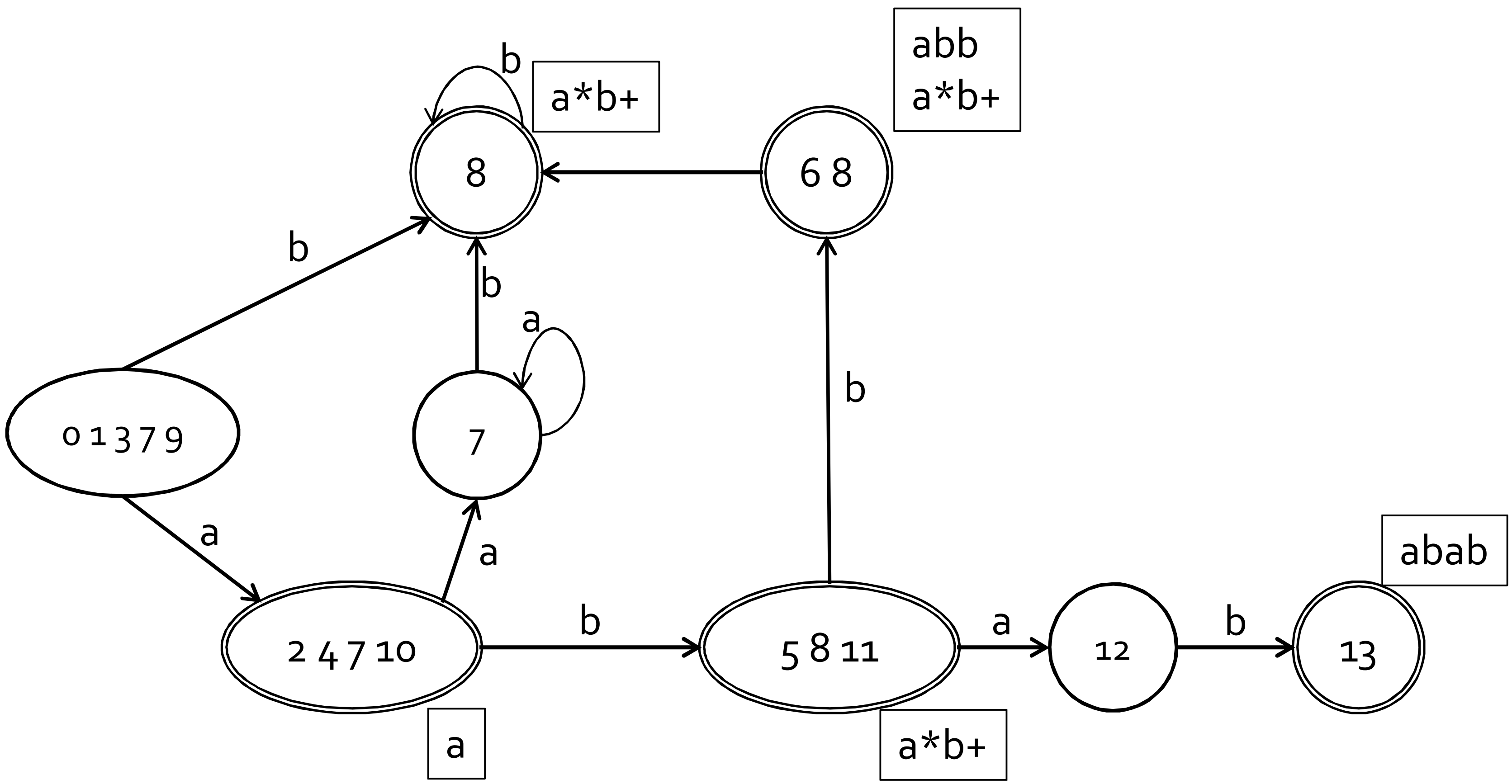


Ambiguity resolution

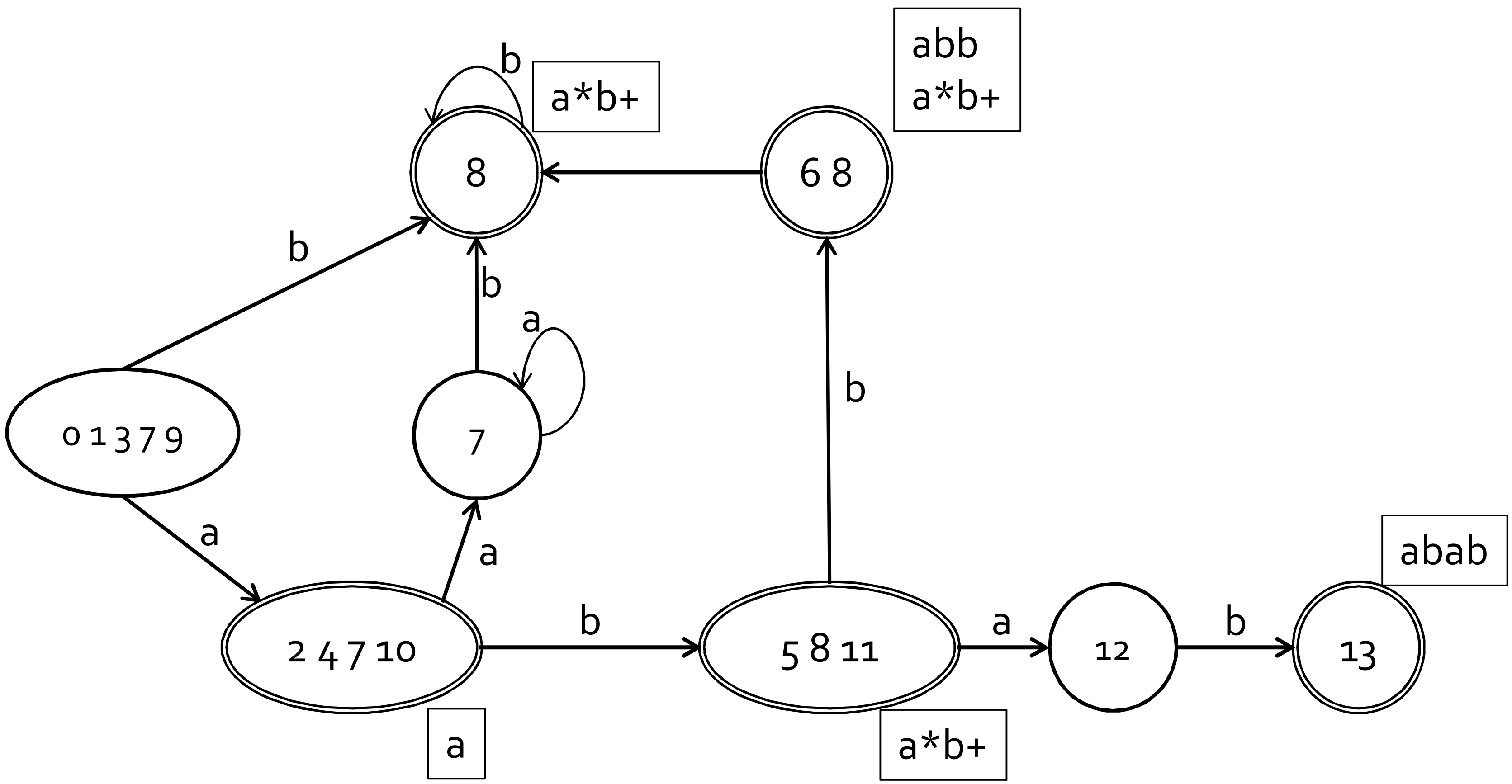
- Recall…
- Longest word
- Tie-breaker based on order of rules when words have same length

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

Corresponding DFA



Examples



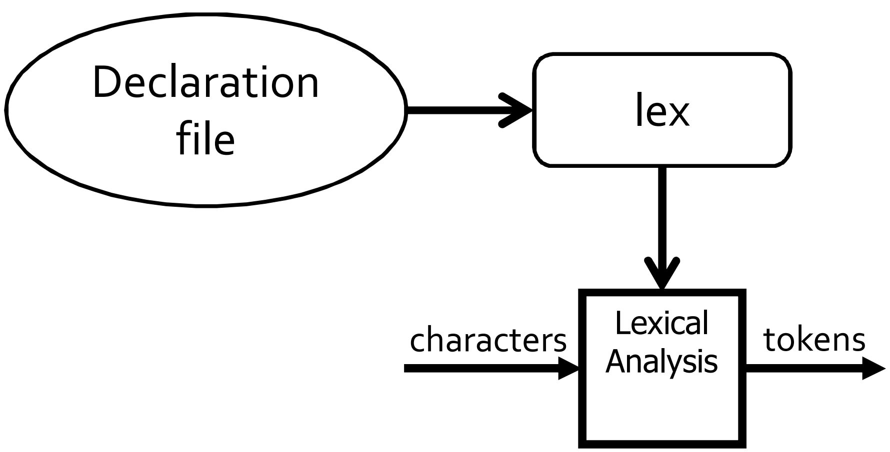
abaa: gets stuck after aba in state 12, backs up to state (5 8 11) pattern is a*b+, token is ab

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

Good News

- All of this construction is done automatically for you by common tools

- lex is your friend
  - Automatically generates a lexical analyzer from declaration file



Lex declarations file

```
%{
#include “lex.h”
Token_Type Token;
int line_number=1
%}
whitespace [ \t]
letter [a-zA-Z]
digit [0-9]
…
%%
{digit}+ {return INTEGER;}
{identifier} {return IDENTIFIER;}
{whitespace} { /* ignore whitespace */ }
\n           { line_number++;}
.            { return ERROR; }
…
%%
void start_lex(void){}
void get_next_token(void) {…}
```

Summary

- Lexical analyzer
  - Turns character stream into token stream
  - Tokens defined using regular expressions
  - Regular expressions -> NFA -> DFA construction for identifying tokens
  - Automated constructions of lexical analyzer using lex

Coming up next time

- Syntax analysis

NFA vs. DFA

<table>
<tr><th>Automaton</th><th>SPACE</th><th>TIME</th></tr>
<tr><td>NFA</td><td>O(|r|)</td><td>O(|r|*|w|)</td></tr>
<tr><td>DFA</td><td>O(2^|r|)</td><td>O(|w|)</td></tr>
</table>

- $(a|b)^*a\underbrace{(a|b)(a|b)\ldots(a|b)}_{\text{n times}}$