Compilation

0368-3133 (Semester A, 2013/14)

Lecture 2: Lexical Analysis

Modern Compiler Design: Chapter 2.1

Noam Rinetzky

Admin

- Slides: *http://www.cs.tau.ac.il/~maon/…*
  - All info: Moodle
- Class room: Trubowicz 101 (Law school)
  - Except 5 Nov 2013
- Mobiles …

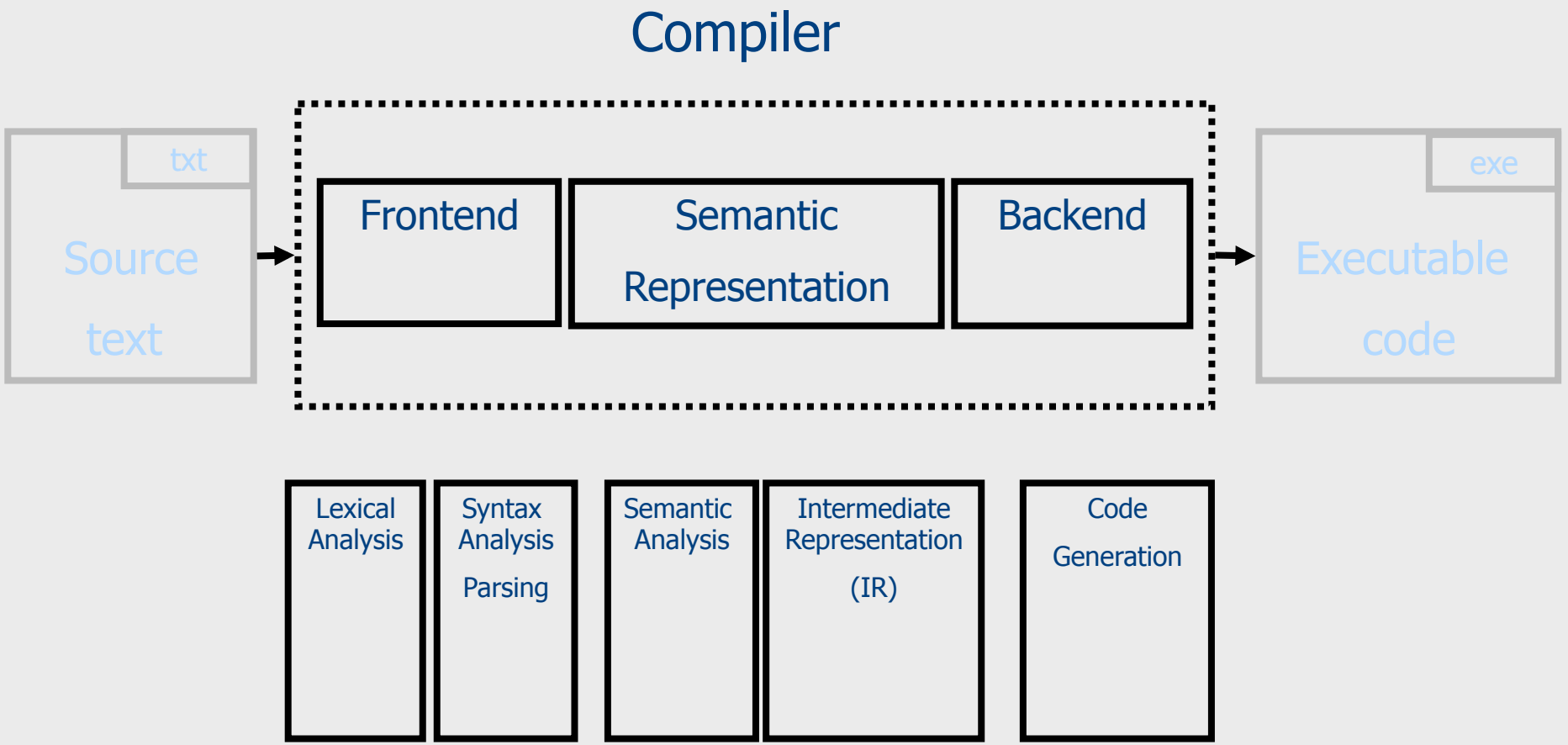
What is a Compiler?

"A compiler is a computer program that transforms source code written in a programming language (source language) into another language (target language).

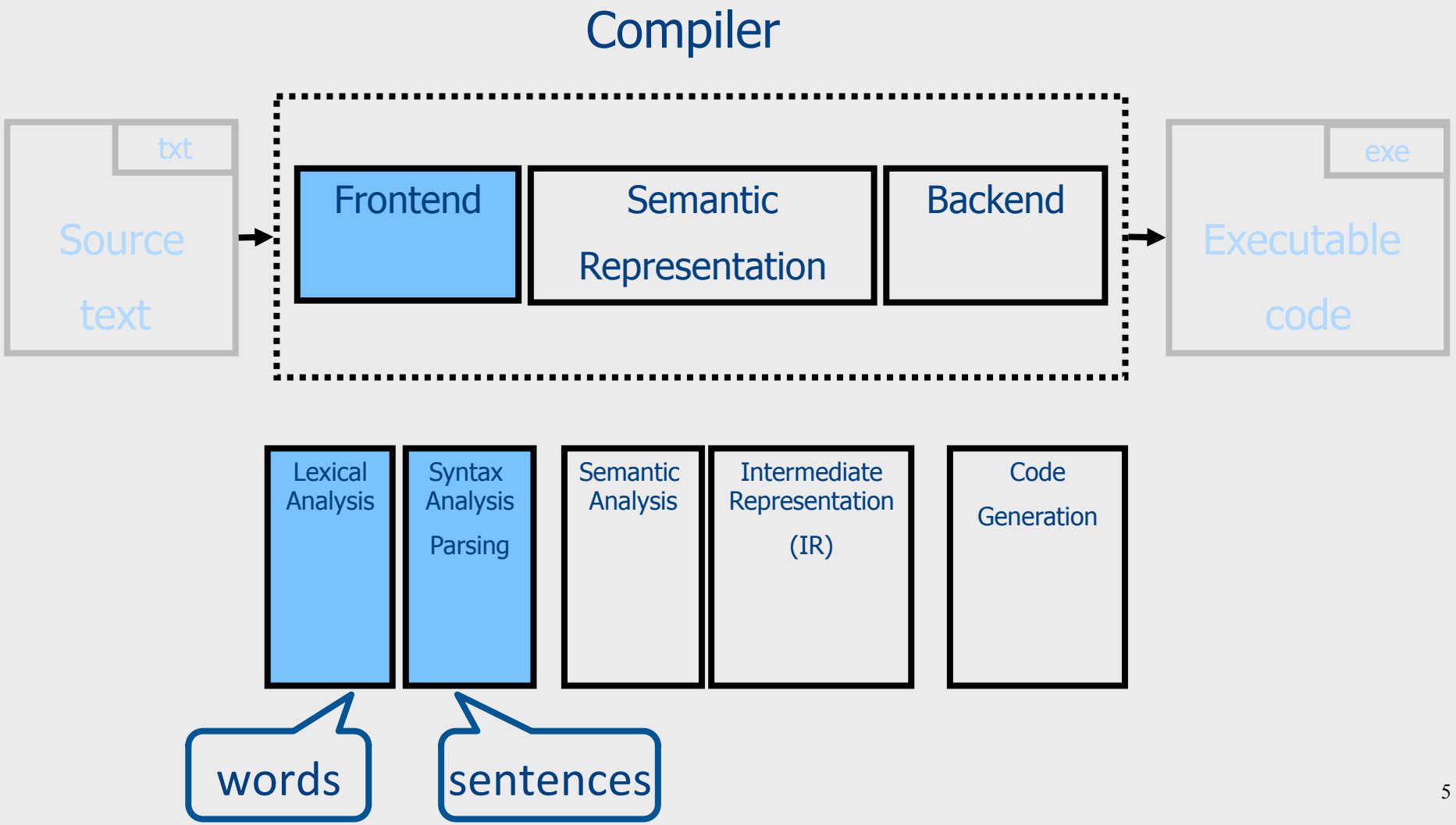
The most common reason for wanting to transform source code is to create an executable program."

--Wikipedia

Conceptual Structure of a Compiler



Conceptual Structure of a Compiler



What does Lexical Analysis do?

- Language: fully parenthesized expressions

Expr → Num | LP Expr Op Expr RP

Num → Dig | Dig Num

Dig → ‘0’ | ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’ | ‘7’ | ‘8’ | ‘9’

LP → ‘(’

RP → ‘)’

Op → ‘+’ | ‘*’

(	(	23	+	7	)	*	19	)

What does Lexical Analysis do?

- Language: fully parenthesized expressions

Context free language

Expr → Num | LP Expr Op Expr RP

Regular languages

Num → Dig | Dig Num

Dig → ‘0’ | ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’ | ‘7’ | ‘8’ | ‘9’

LP → ‘(’

RP → ‘)’

Op → ‘+’ | ‘*’

(	(	23	+	7	)	*	19	)

What does Lexical Analysis do?

- Language: fully parenthesized expressions

Context free language

Expr → Num | LP Expr Op Expr RP

Regular languages

Num → Dig | Dig Num

Dig → ‘0’ | ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’ | ‘7’ | ‘8’ | ‘9’

LP → ‘(’

RP → ‘)’

Op → ‘+’ | ‘*’

(	(	23	+	7	)	*	19	)

What does Lexical Analysis do?

- Language: fully parenthesized expressions

Context free language

Expr → Num | LP Expr Op Expr RP

Regular languages

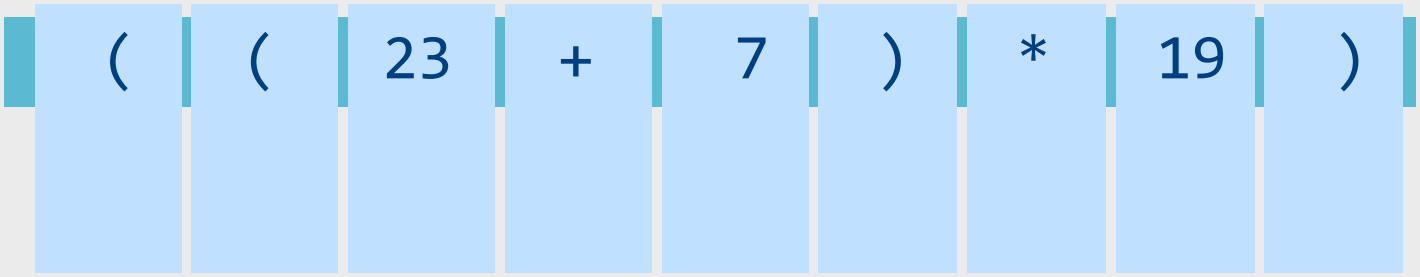
Num → Dig | Dig Num

Dig → ‘0’ | ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’ | ‘7’ | ‘8’ | ‘9’

LP → ‘(’

RP → ‘)’

Op → ‘+’ | ‘*’



What does Lexical Analysis do?

- Language: fully parenthesized expressions

Context free language

Expr → Num | LP Expr Op Expr RP

Regular languages

Num → Dig | Dig Num

Dig → ‘0’ | ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’ | ‘7’ | ‘8’ | ‘9’

LP → ‘(’

RP → ‘)’

Op → ‘+’ | ‘*’

(	(	23	+	7	)	*	19	)
LP	LP	Num	Op	Num	RP	Op	Num	RP

What does Lexical Analysis do?

- Language: fully parenthesized expressions

Context free language

Expr → Num | LP Expr Op Expr RP

Regular languages

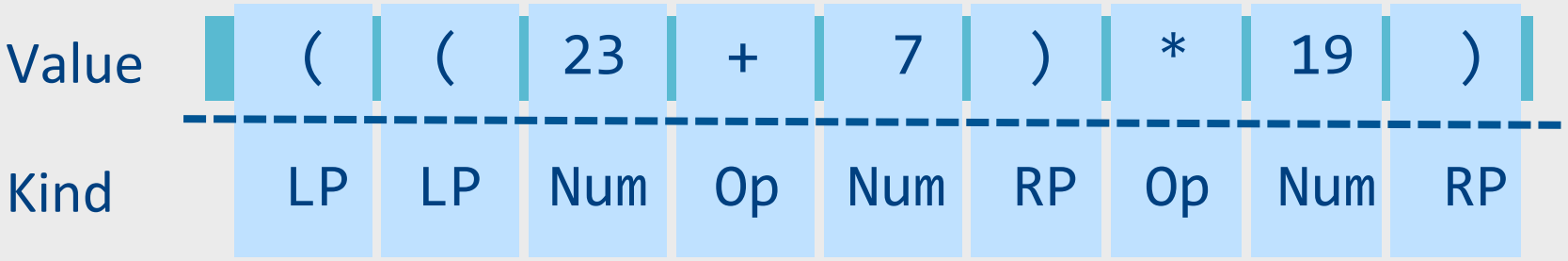
Num → Dig | Dig Num

Dig → ‘0’ | ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’ | ‘7’ | ‘8’ | ‘9’

LP → ‘(’

RP → ‘)’

Op → ‘+’ | ‘*’



What does Lexical Analysis do?

- Language: fully parenthesized expressions

Context free language

Expr → Num | LP Expr Op Expr RP

Regular languages

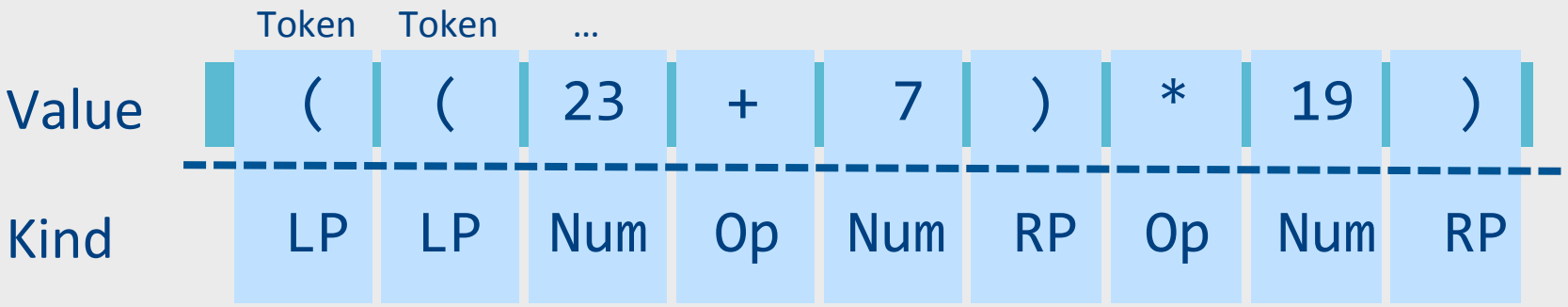
Num → Dig | Dig Num

Dig → ‘0’ | ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’ | ‘7’ | ‘8’ | ‘9’

LP → ‘(’

RP → ‘)’

Op → ‘+’ | ‘*’



What does Lexical Analysis do?

- Partitions the input into stream of tokens
  - Numbers
  - Identifiers
  - Keywords
  - Punctuation

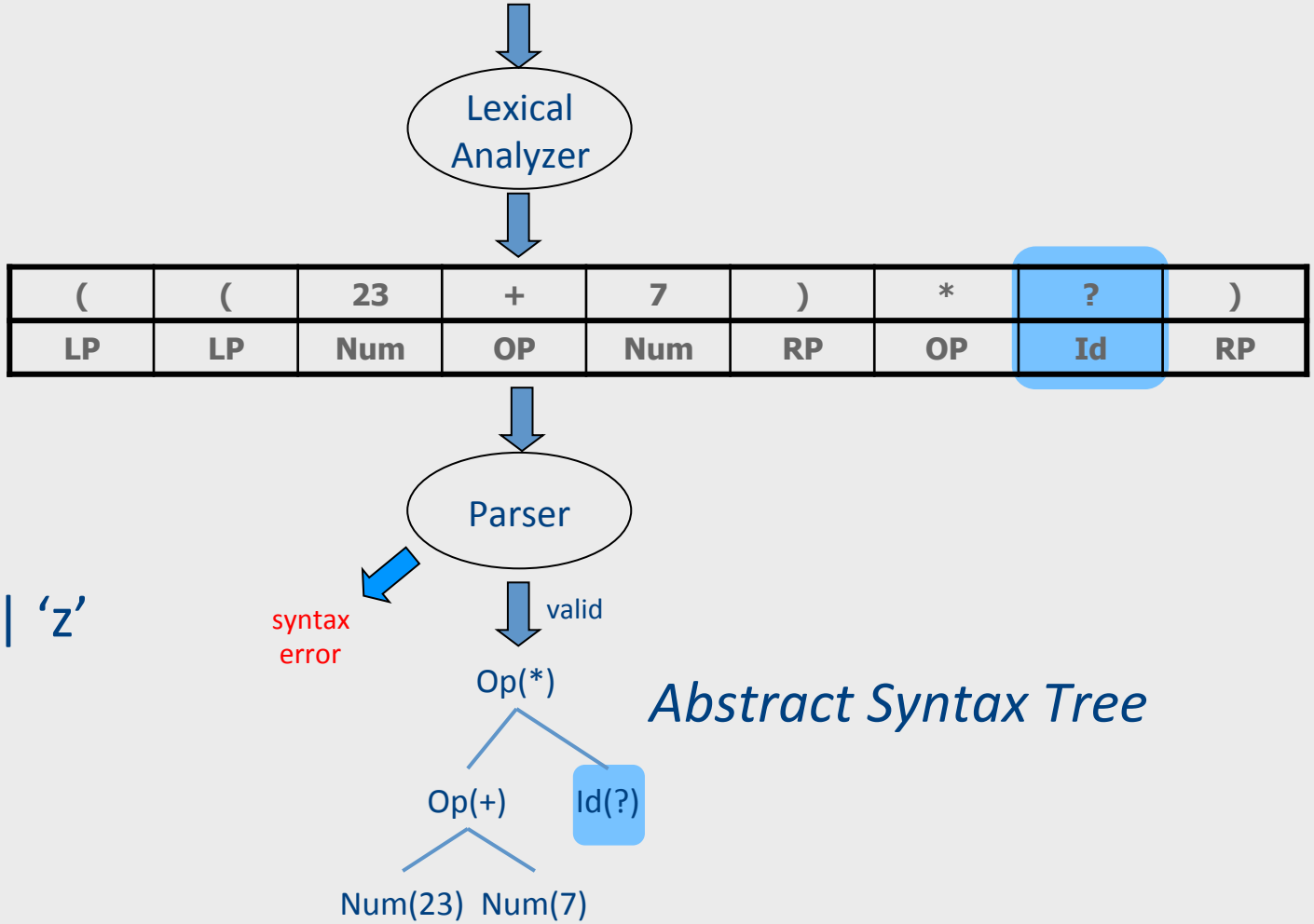
- "word" in the source language
- "meaningful" to the syntactical analysis

- Usually represented as (kind, value) pairs
  - (Num, 23)
  - (Op, '*')

From scanning to parsing

program text

((23 + 7) * x)



token stream

Grammar:

E → ... | Id

Id → ‘a’ | ... | ‘z’

Abstract Syntax Tree

Why Lexical Analysis?

- Simplifies the syntax analysis (parsing)
  - And language definition
- Modularity
- Reusability
- Efficiency

Lecture goals

- Understand role & place of lexical analysis
- Lexical analysis theory
- Using program generating tools

Lecture Outline

- ✓ Role & place of lexical analysis
- What is a token?
- Regular languages
- Lexical analysis
- Error handling
- Automatic creation of lexical analyzers

What is a token? (Intuitively)

- A “word” in the source language
  - Anything that should appear in the input to syntax analysis
    - Identifiers
    - Values
    - Language keywords
- Usually, represented as a pair of (kind, value)

Example Tokens

Type	**Examples**
ID	`foo, n_14, last`
NUM	`73, 00, 517, 082`
REAL	`66.1, .5, 5.5e-10`
IF	`if`
COMMA	`,`
NOTEQ	`!=`
LPAREN	`(`
RPAREN	`)`

Example Non Tokens

Type	**Examples**
comment	`/* ignored */`
preprocessor directive	`#include <foo.h>`
	`#define NUMS 5.6`
macro	`NUMS`
whitespace	`\t, \n, \b, ‘ ‘`

Some basic terminology

- Lexeme (aka symbol) - a series of letters separated from the rest of the program according to a convention (space, semi-column, comma, etc.)
- Pattern - a rule specifying a set of strings. Example: “an identifier is a string that starts with a letter and continues with letters and digits”
  - (Usually) a regular expression
- Token - a pair of (pattern, attributes)

Example

```
void match0(char *s) /* find a zero */
{
  if (!strncmp(s, “0.0”, 3))
    return 0. ;
}
```

```
VOID ID(match0) LPAREN CHAR DEREF ID(s) RPAREN
LBRACE
IF LPAREN NOT ID(strncmp) LPAREN ID(s) COMMA STRING(0.0)
COMMA NUM(3) RPAREN RPAREN
RETURN REAL(0.0) SEMI
RBRACE
EOF
```

Example Non Tokens

Type	**Examples**
comment	`/* ignored */`
preprocessor directive	`#include <foo.h>`
	`#define NUMS 5.6`
macro	`NUMS`
whitespace	`\t, \n, \b, ‘ ‘`

- Lexemes that are recognized but get consumed rather than transmitted to parser
  - If
  - i/*comment*/f

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?

Lecture Outline

- ✓ Role & place of lexical analysis
- ✓ What is a token?
- Regular languages
- Lexical analysis
- Error handling
- Automatic creation of lexical analyzers

Regular languages

- Formal languages
  - Σ = finite set of letters
  - Word = sequence of letter
  - Language = set of words

- Regular languages defined equivalently by
  - Regular expressions
  - Finite-state automata

Common format for reg-exps

Basic Patterns	Matching
x	The character x
.	Any character, usually except a new line
[xyz]	Any of the characters x,y,z
^x	Any character except x
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`
  - `\*, \?, \+, a\(b\+\*, (a\(b\+\*)+, …`
- backslash double quotes surrounds text
  - "a(b+*", "a(b+*"+

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

- if = if
- then = then
- relop = < | > | <= | >= | = | <>

- digit = 0-9
- digits = digit+

Example

- A number is

```
number = ( 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 )⁺
     ( ε | . ( 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 )⁺
          ( ε | E ( 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 )⁺
          )
)
```

- Using shorthands it can be written as

```
number = digits (ε | .digits (ε | E (ε|+|-) digits ) )
```

Exercise 1 - Question

- Language of Java identifiers
  - Identifiers start with either an underscore '_' or a letter
  - Continue with either underscore, letter, or digit

Exercise 1 - Answer

- Language of Java identifiers
  - Identifiers start with either an underscore '_' or a letter
  - Continue with either underscore, letter, or digit

- (_|a|b|…|z|A|…|Z)(_|a|b|…|z|A|…|Z|0|…|9)*
- Using shorthand macros

  First = _|a|b|…|z|A|…|Z
  Next = First|0|…|9
  R = First Next*

Exercise 1 - Question

- Language of rational numbers in decimal representation (no leading, ending zeros)
  - 0
  - 123.757
  - .933333
  - Not 007
  - Not 0.30

Exercise 1 - Answer

- Language of rational numbers in decimal representation (no leading, ending zeros)
  - Digit = 1|2|…|9
    Digit0 = 0|Digit
    Num = Digit Digit0*
    Frac = Digit0* Digit
    Pos = Num | .Frac | 0.Frac| Num.Frac
    PosOrNeg = (Є|-)Pos
    R = 0 | PosOrNeg

Exercise 2 - Question

- Equal number of opening and closing parenthesis: $[^n]^n$ = [], [[]], [[[]]], …

Exercise 2 - Answer

- Equal number of opening and closing parenthesis: $[^n]^n$ = [], [[]], [[[]]], …
- Not regular
- Context-free
- Grammar: S ::= [] | [S]

Challenge: 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

- Given a list of token definitions (pattern name, regex), write a program such that
  - Input: String to be analyzed
  - Output: List of tokens

- How do we build an analyzer?

Lecture Outline

- ✓ Role & place of lexical analysis
- ✓ What is a token?
- ✓ Regular languages
- Lexical analysis
- Error handling
- Automatic creation of lexical analyzers

A Simplified Scanner for C

```
Token nextToken()
{
char c ;
loop: c = getchar();
switch (c){
  case ` `: goto loop ;
  case `;`: return SemiColumn;
  case `+`:
    c = getchar() ;
    switch (c) {
      case `+': return PlusPlus ;
      case '=’  return PlusEqual;
      default:  ungetc(c); return Plus;
    };
  case `<`: …
  case `w`: …
 }
```

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
  - Regular languages defined equivalently by
    - Regular expressions
    - Finite-state automata

Reg-exp vs. automata

- Regular expressions are declarative
  - Offer compact way to define a regular language *by humans*
  - Don't offer direct way to check whether a given word is in the language
- Automata are operative
  - Define an *algorithm* for deciding whether a given word is in a regular language
  - Not a natural notation for humans

Overview

- Define tokens using regular expressions
- Construct a nondeterministic finite-state automaton (NFA) from regular expression
- Determinize the NFA into a deterministic finite-state automaton (DFA)
- DFA can be directly used to identify tokens

Finite-State Automata

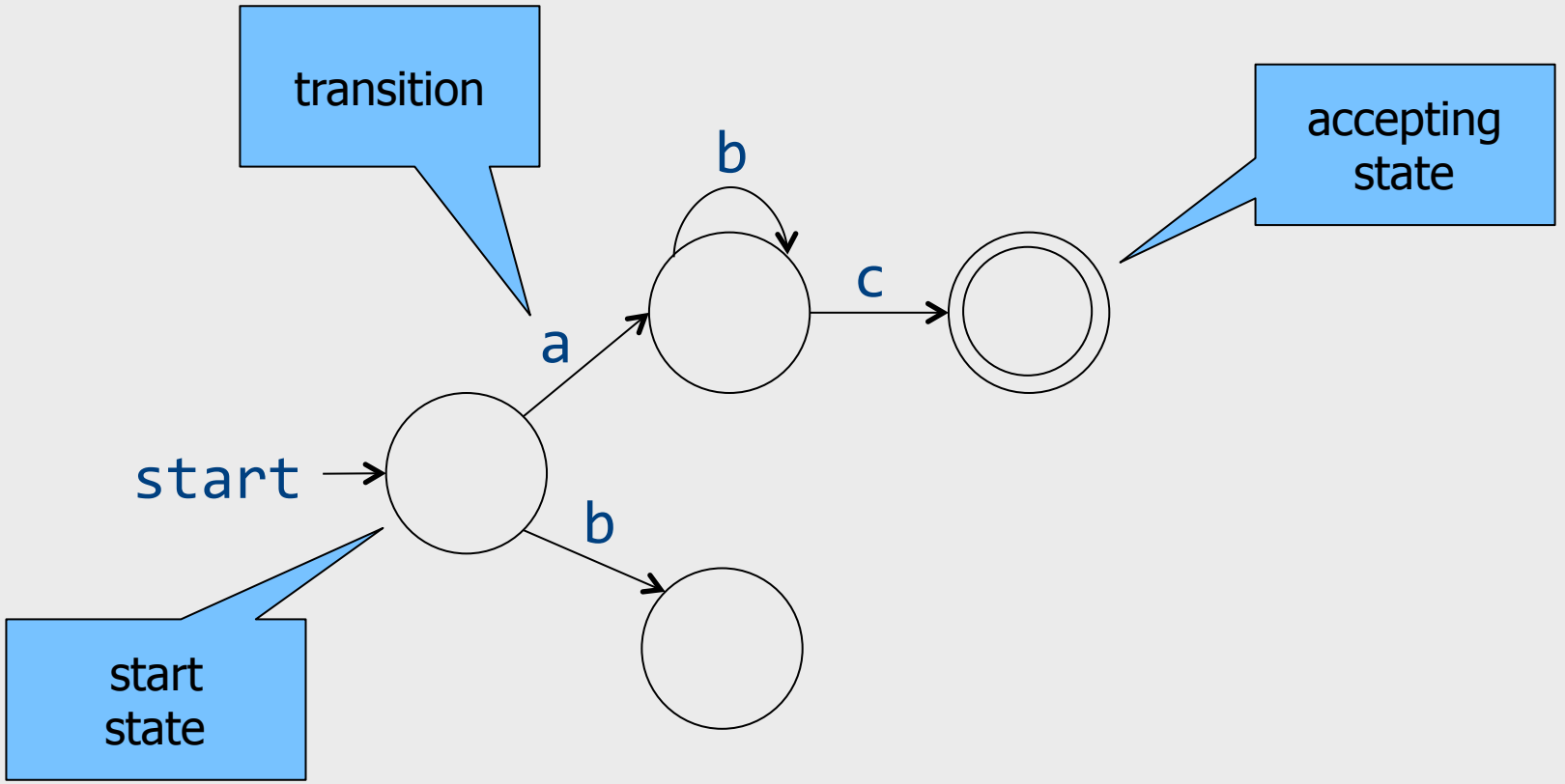
- Deterministic automaton
- $M = (\Sigma, Q, \delta, q_0, F)$
  - Σ - alphabet
  - Q – finite set of state
  - $q_0 \in Q$ – initial state
  - $F \subseteq Q$ – final states
  - $\delta : Q \times \Sigma \rightarrow Q$ - transition function

Finite-State Automata

- 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
- 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.

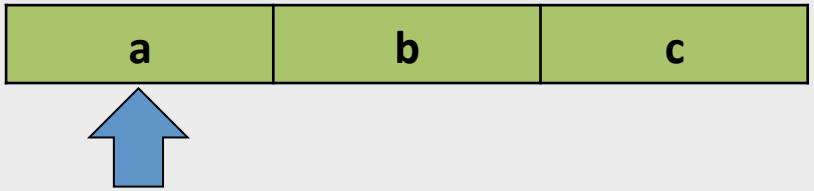
Deterministic Finite automata

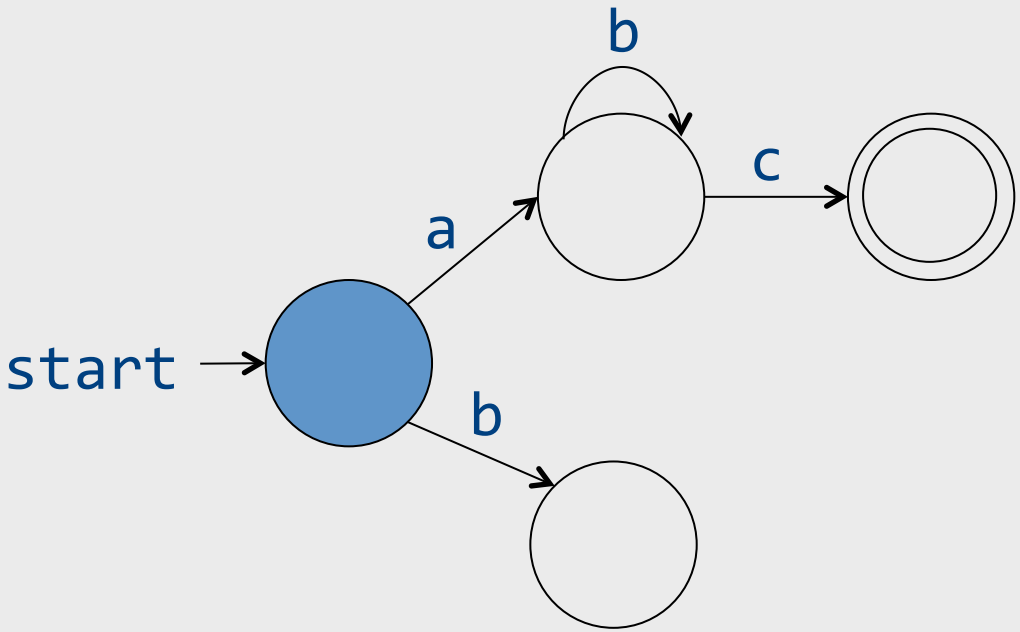
- An automaton is defined by states and transitions



Accepting Words

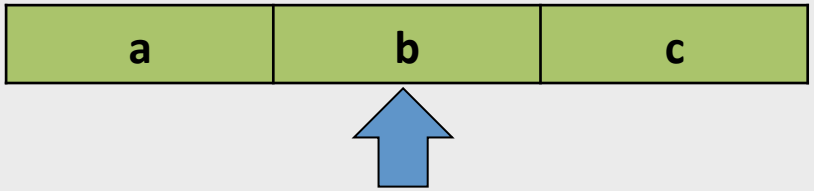
- Words are read left-to-right

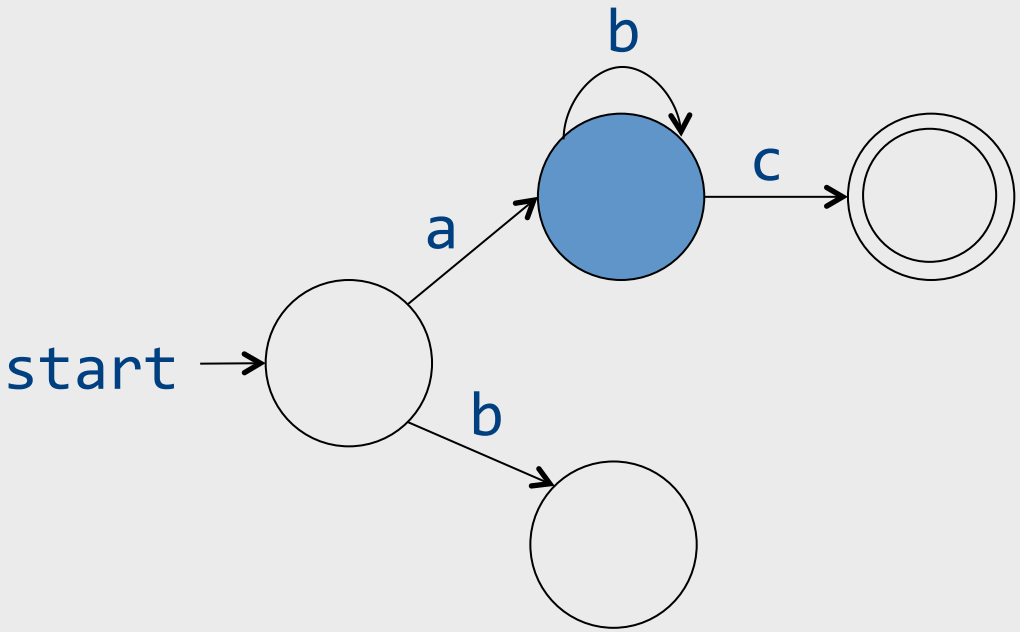




Accepting Words

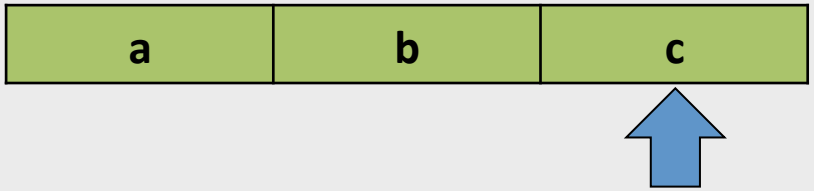
- Words are read left-to-right

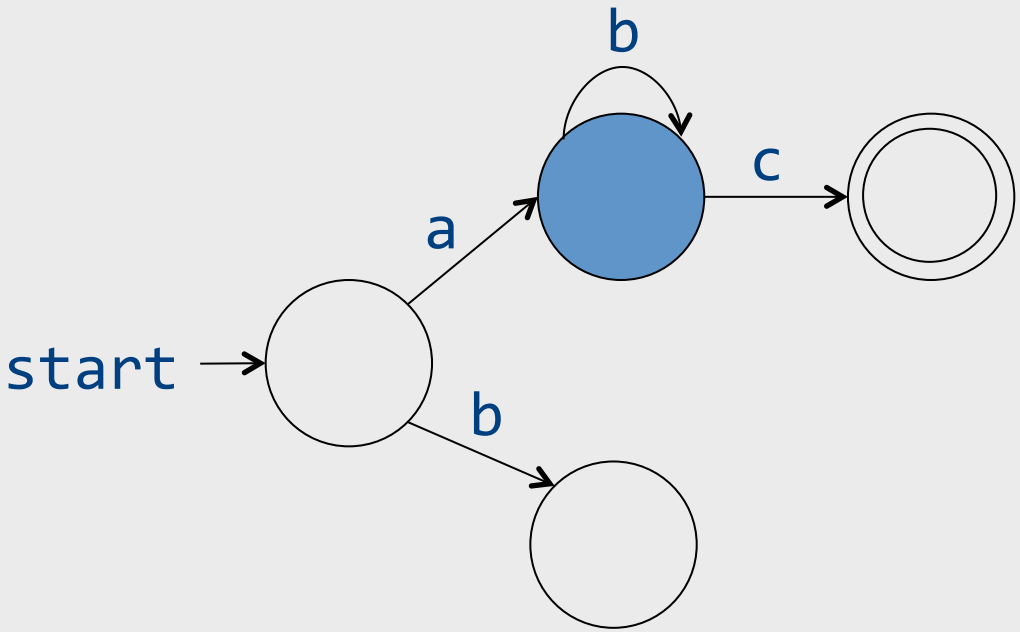




Accepting Words

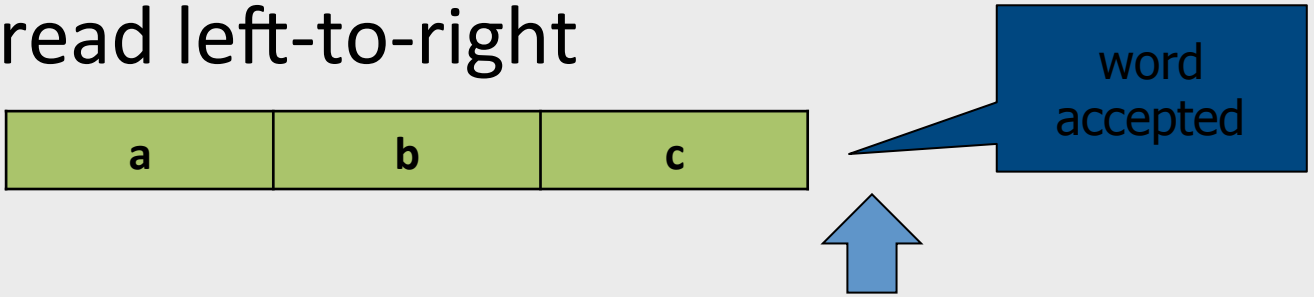
- Words are read left-to-right

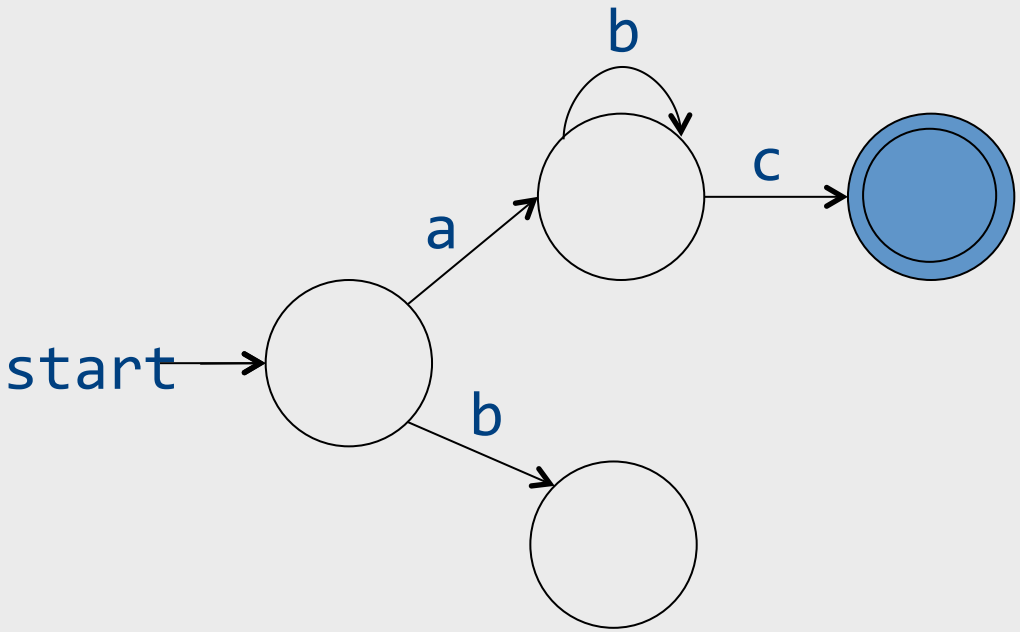




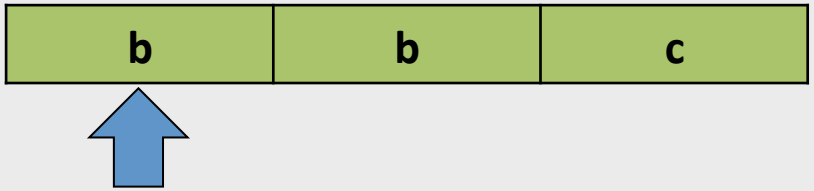
Rejecting Words

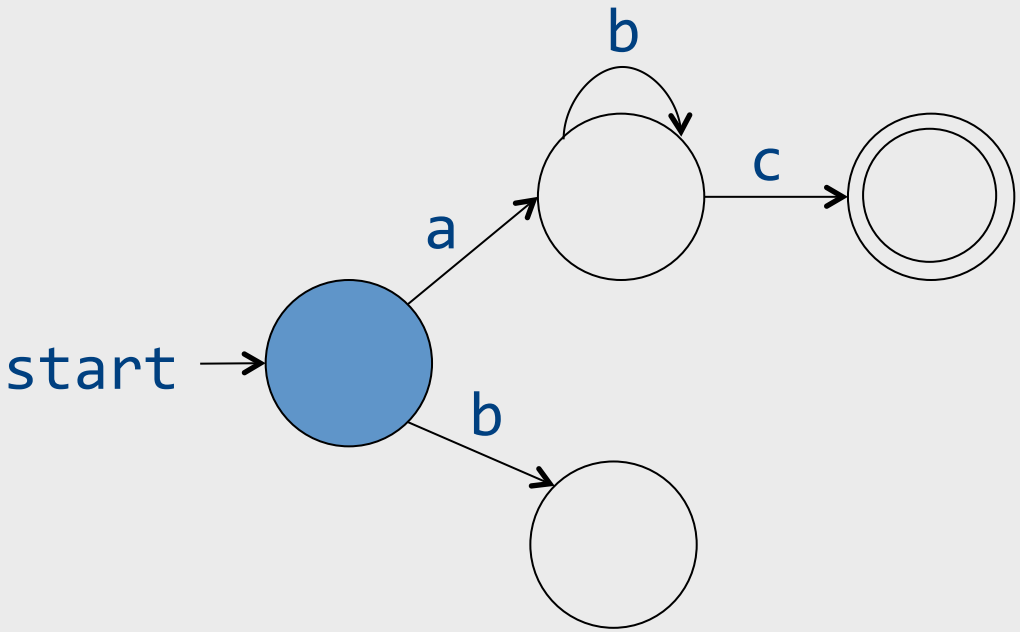
- Words are read left-to-right





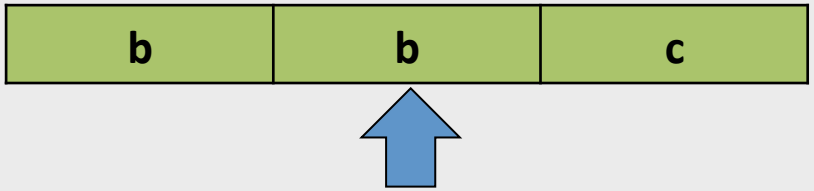
Rejecting Words

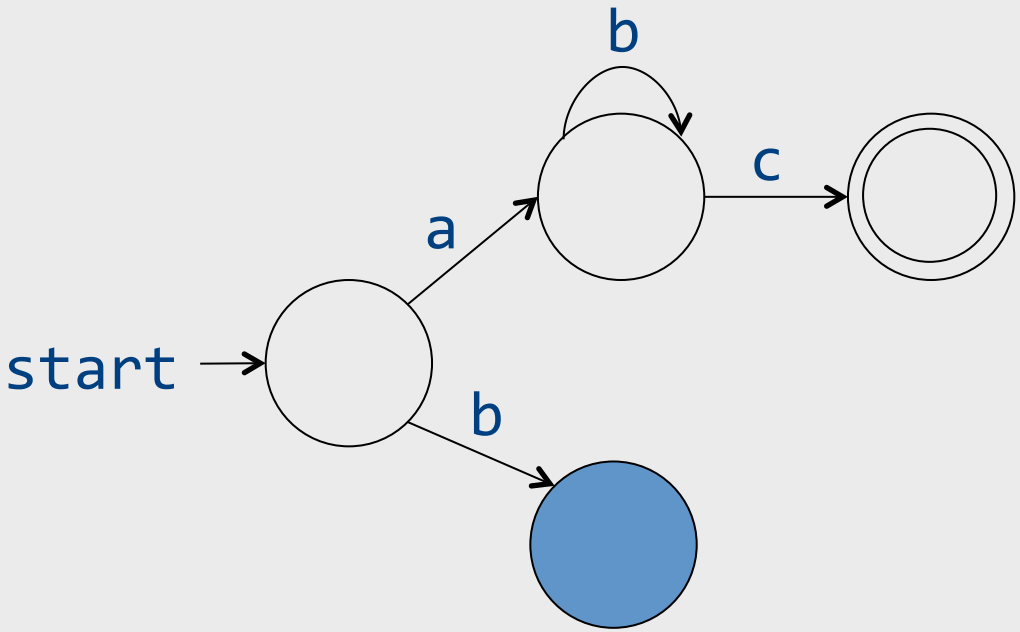




Rejecting Words

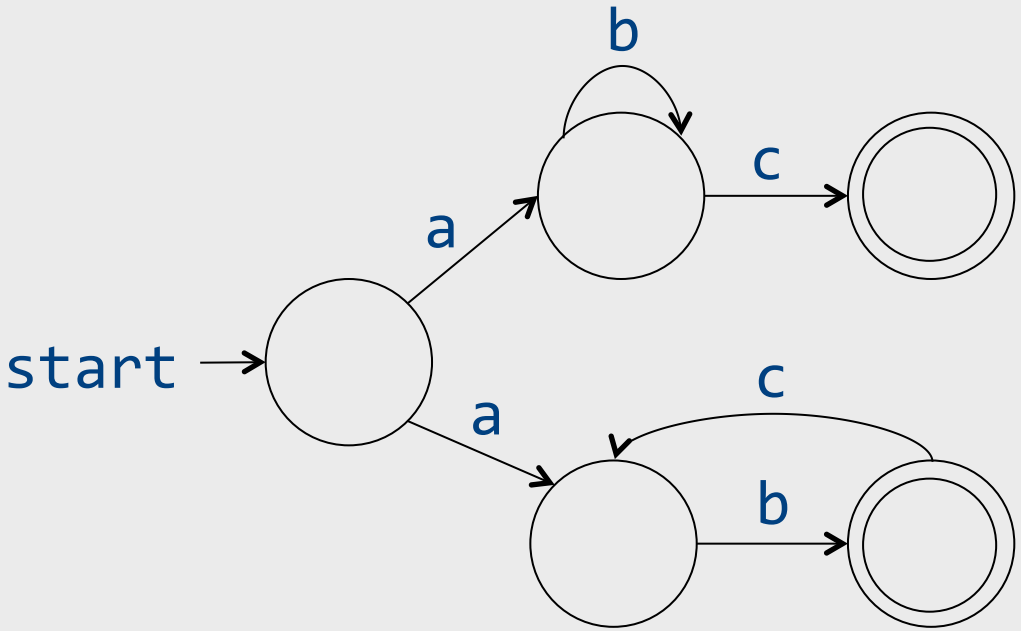
- Missing transition means non-acceptance



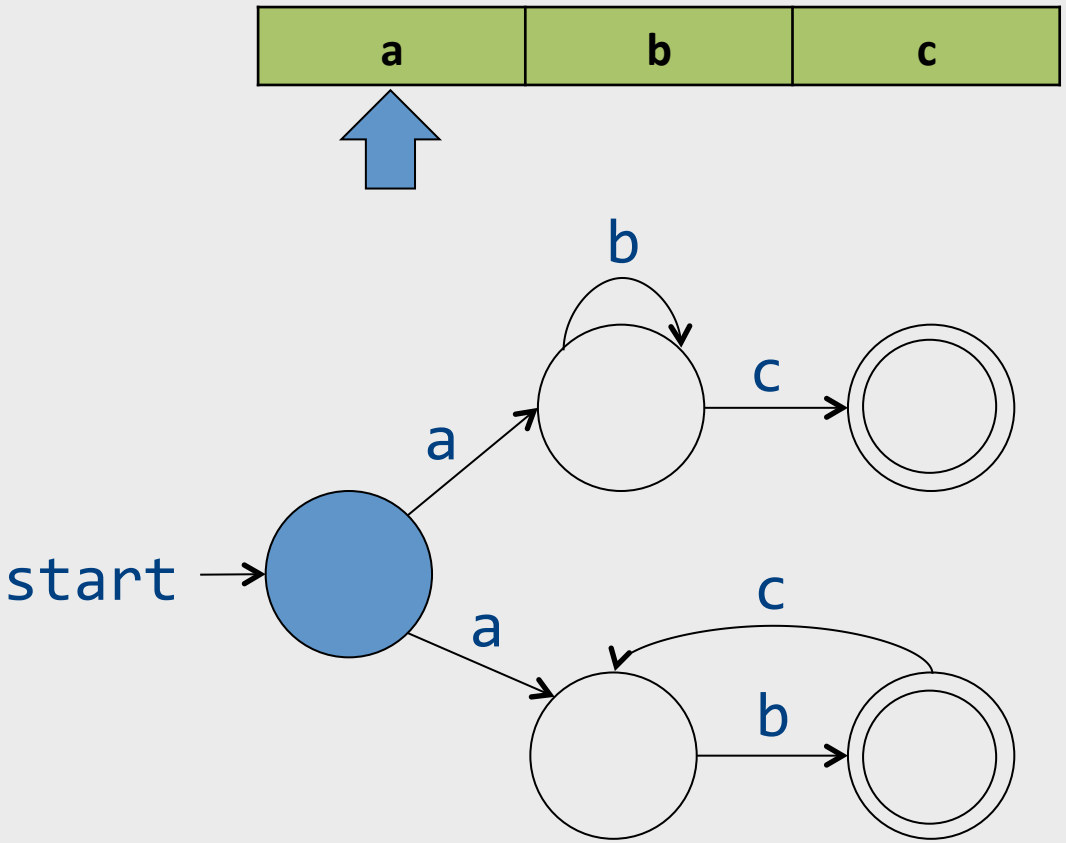


Non-deterministic automata

- Allow multiple transitions from given state labeled by same letter

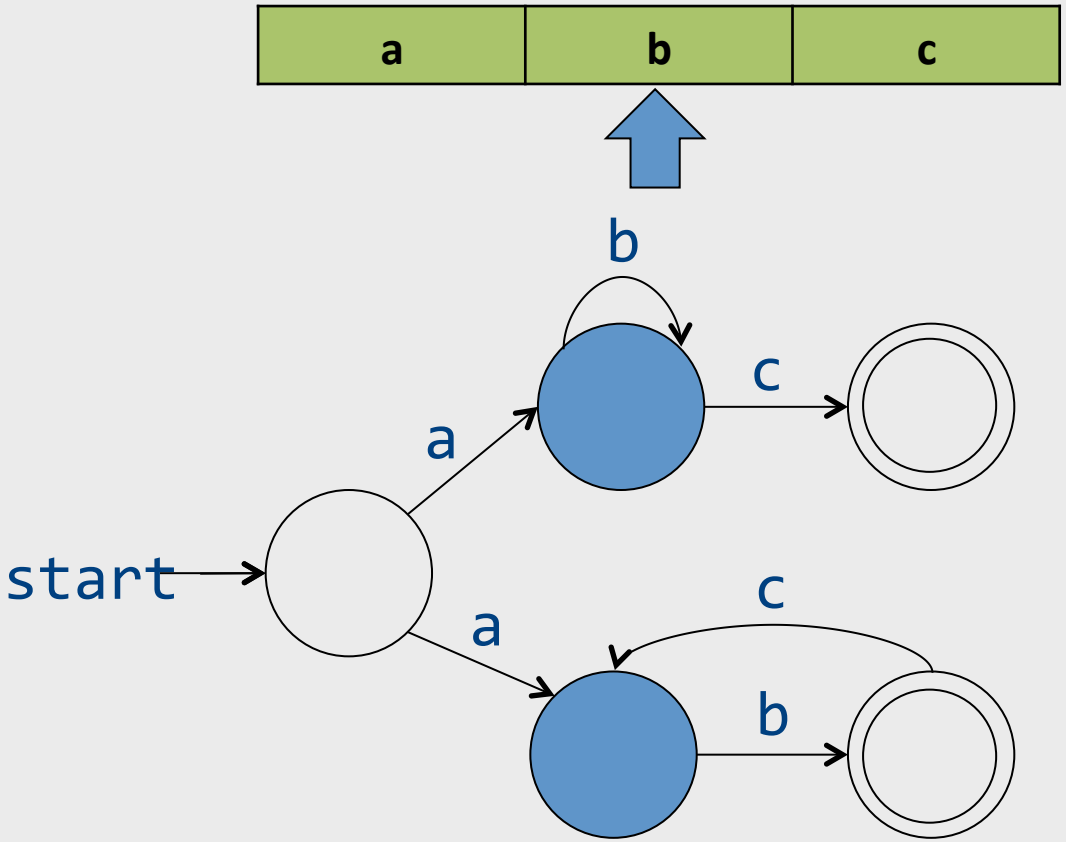


NFA run example

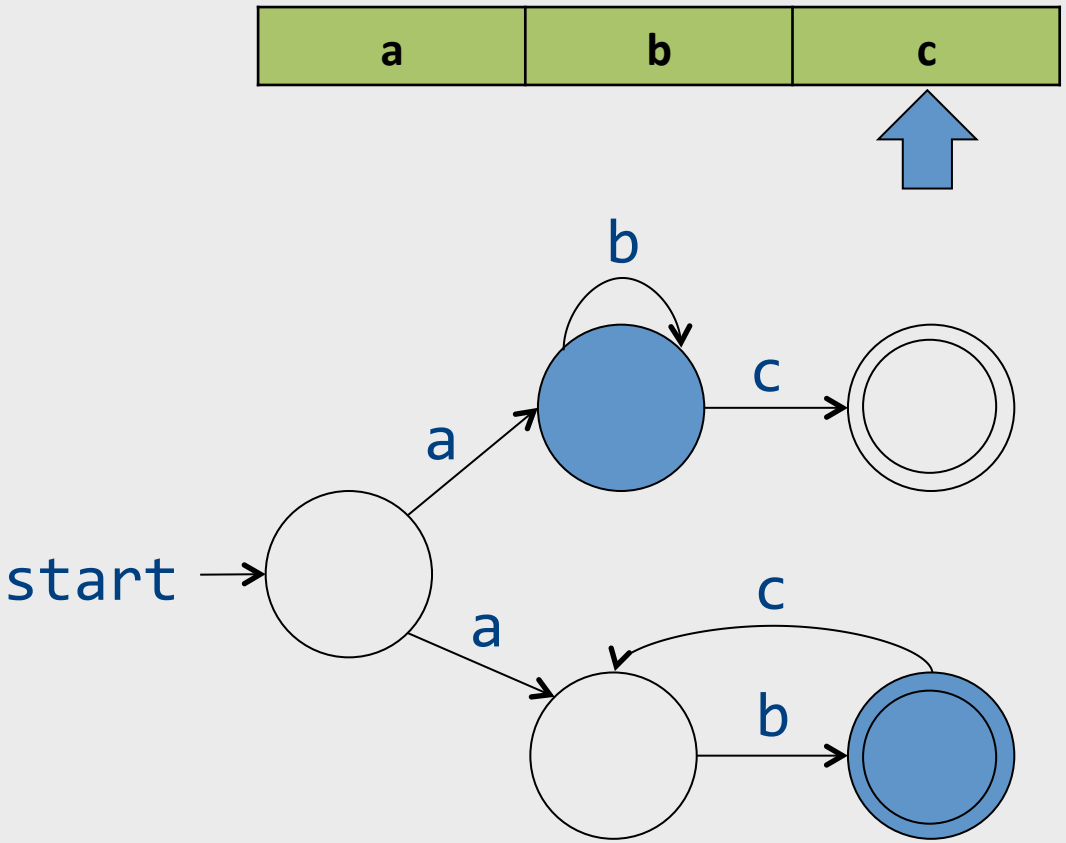


NFA run example

- Maintain set of states

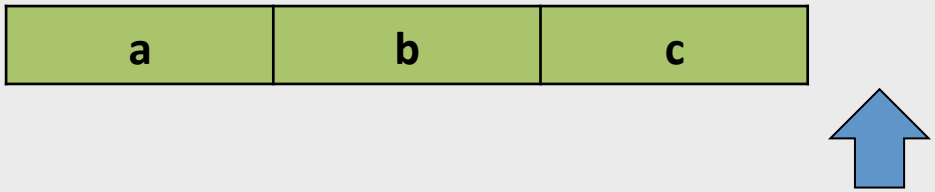


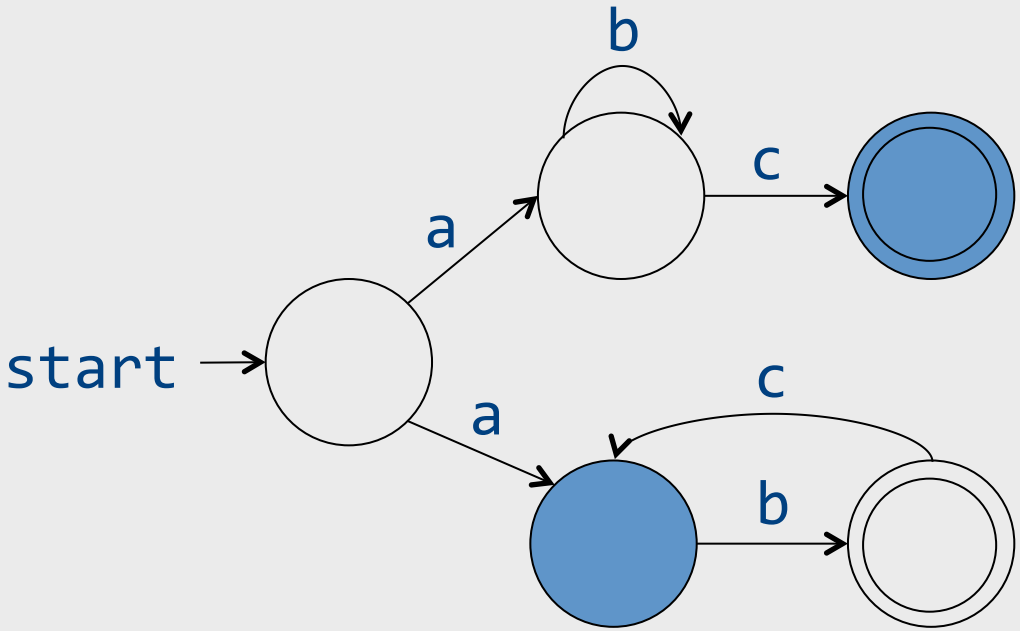
NFA run example



NFA run example

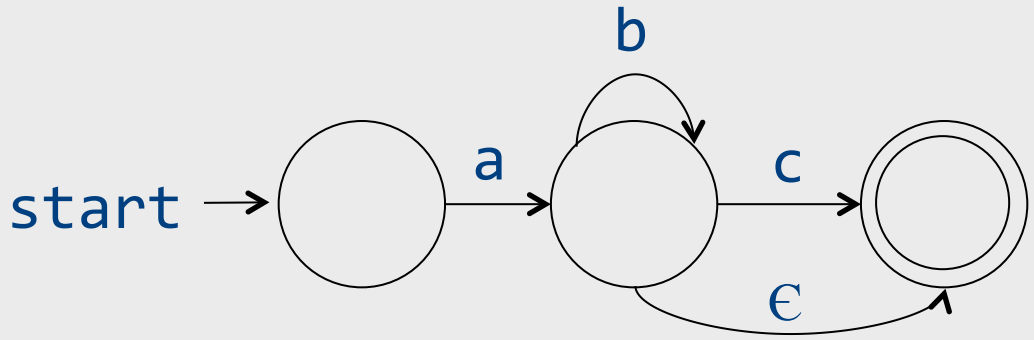
- Accept word if any of the states in the set is accepting



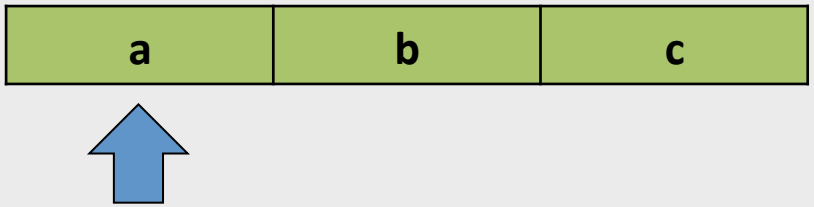


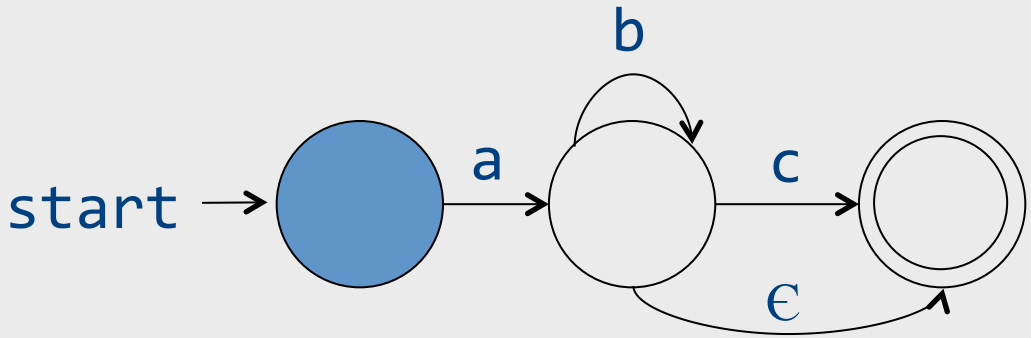
NFA+Є automata

- Є transitions can “fire” without reading the input



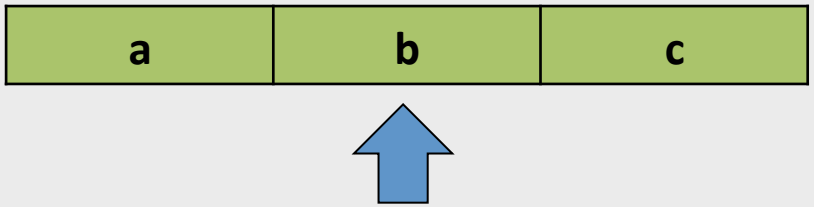
NFA+Є run example

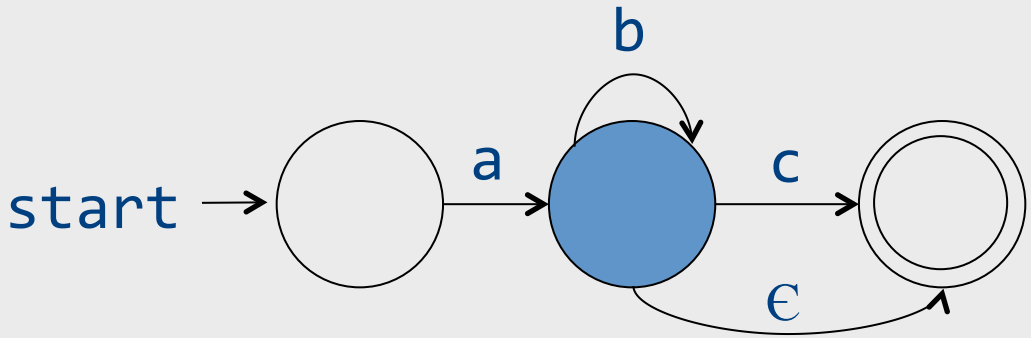




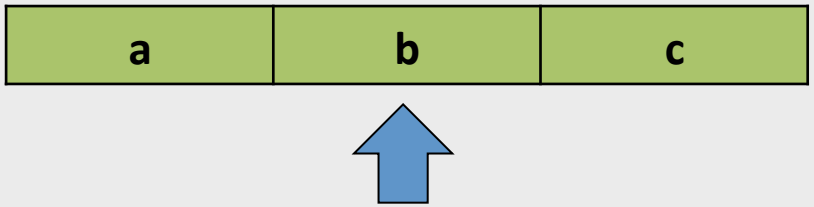
NFA+Є run example

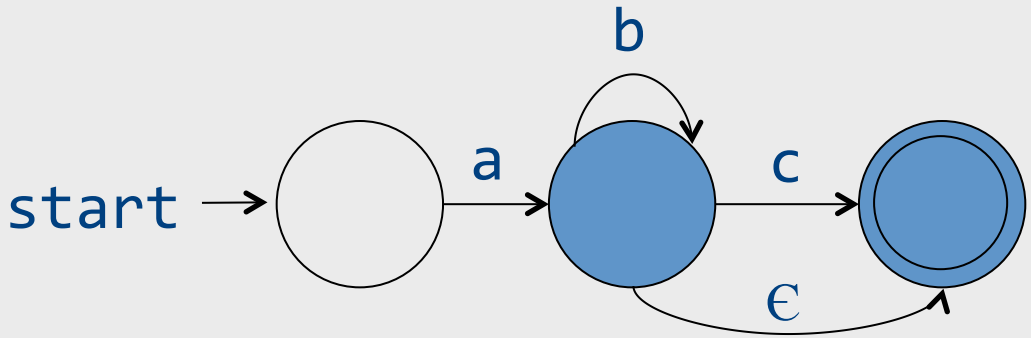
- Now Є transition can non-deterministically take place



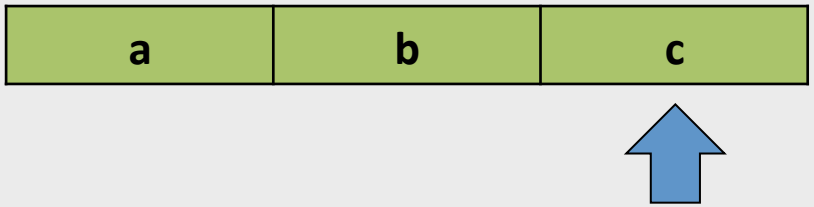


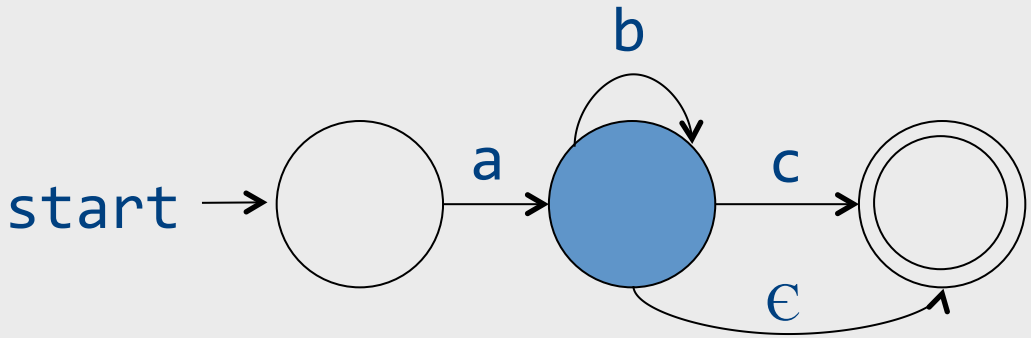
NFA+Є run example



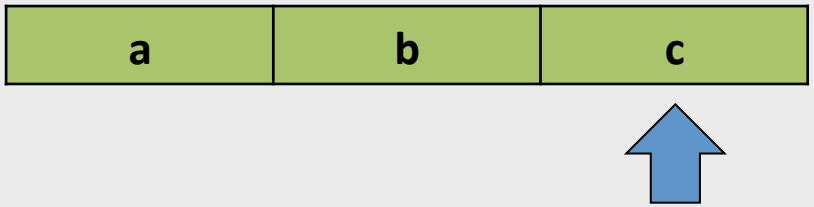


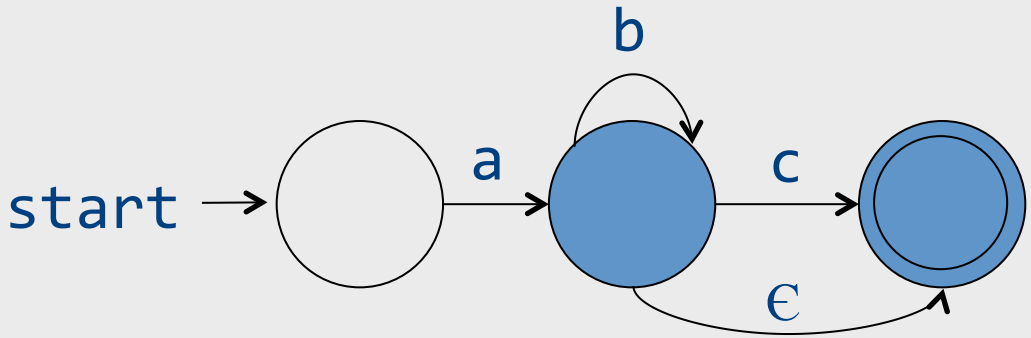
NFA+Є run example





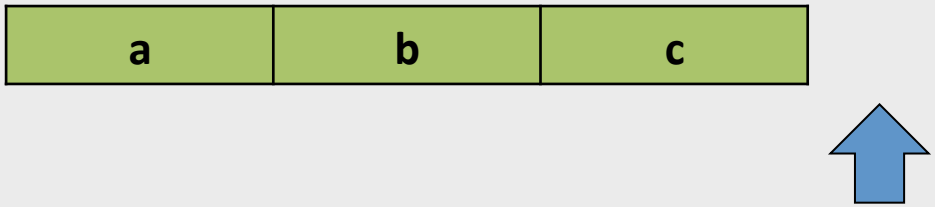
NFA+Є run example

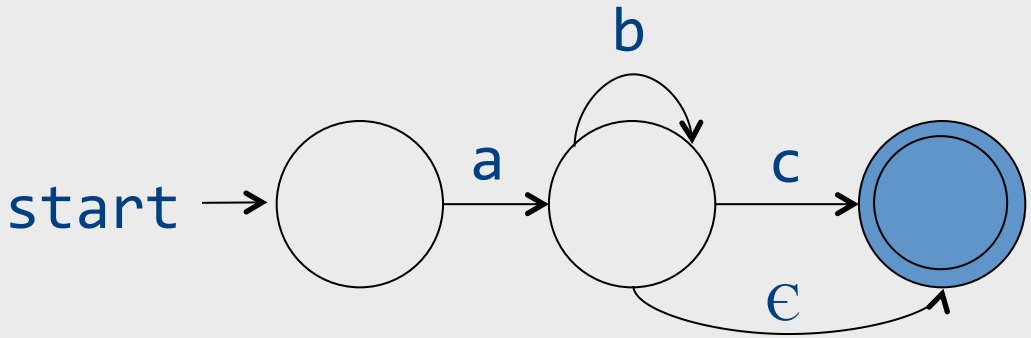




NFA+Є run example

- Word accepted





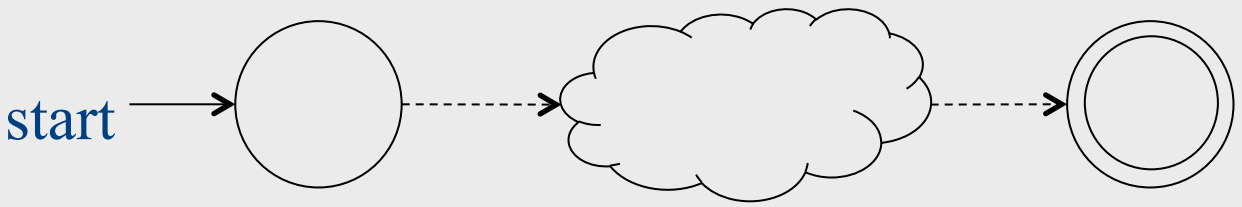
From regular expressions to NFA

- 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

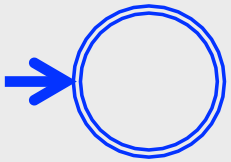
From reg. exp. to automata

- Theorem: *there is an algorithm to build an NFA +Є automaton for any regular expression*
- Proof: *by induction on the structure of the regular expression*

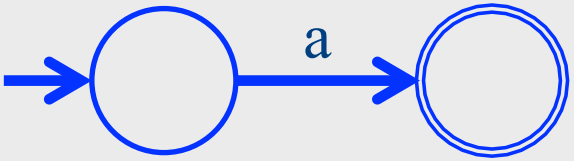


Basic constructs

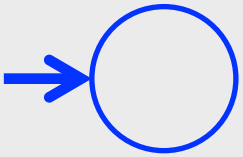
$R = \varepsilon$



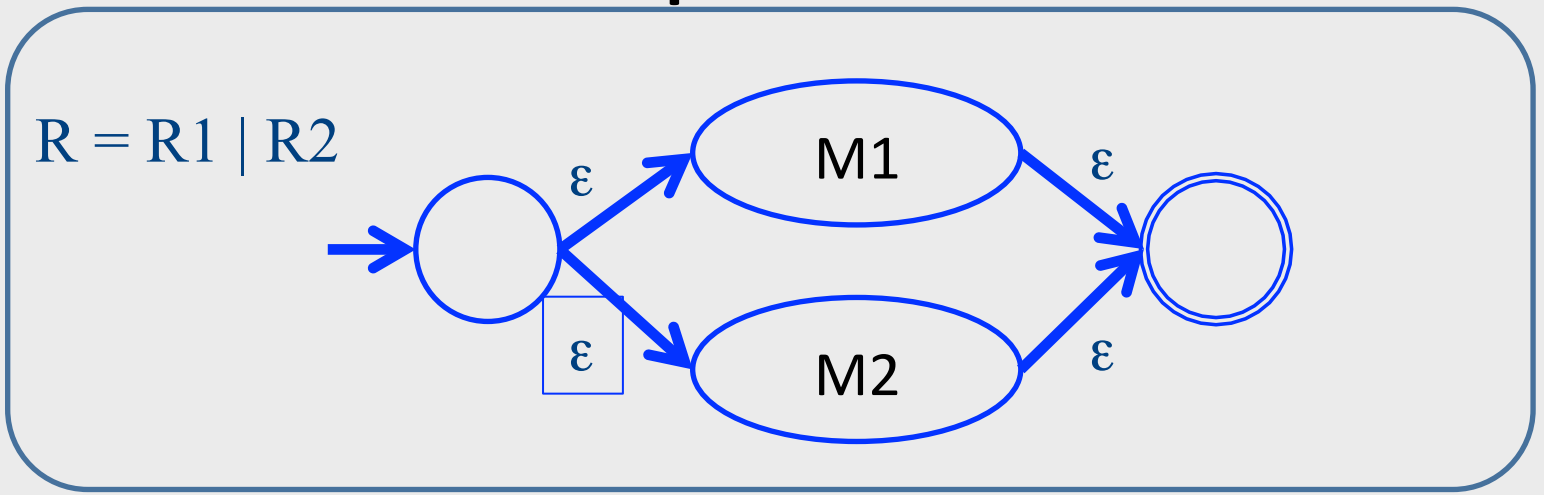
$R = a$

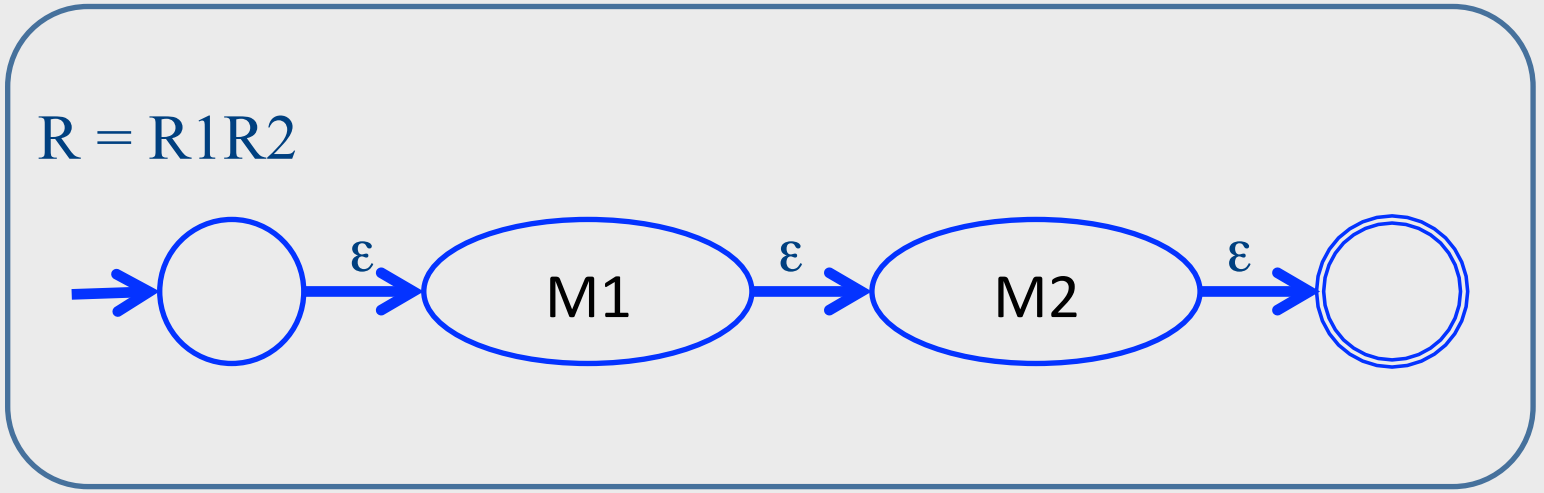


$R = \phi$

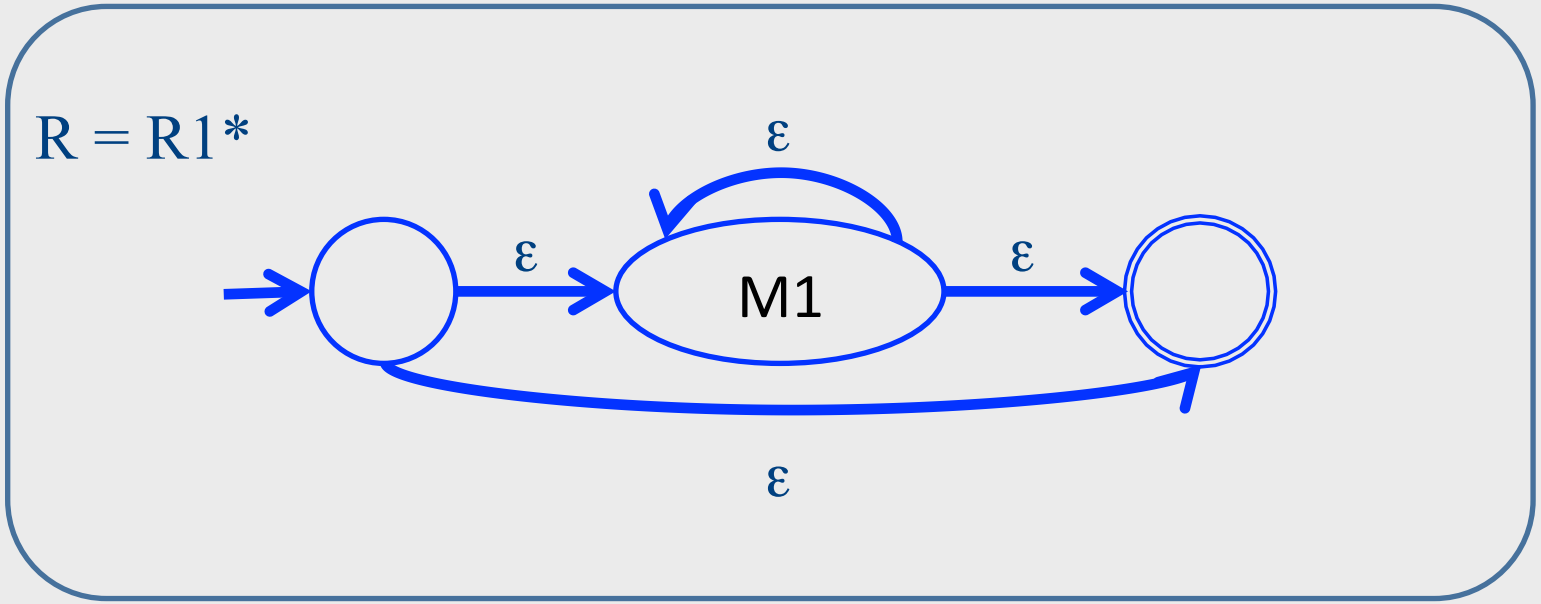


Composition





Repetition



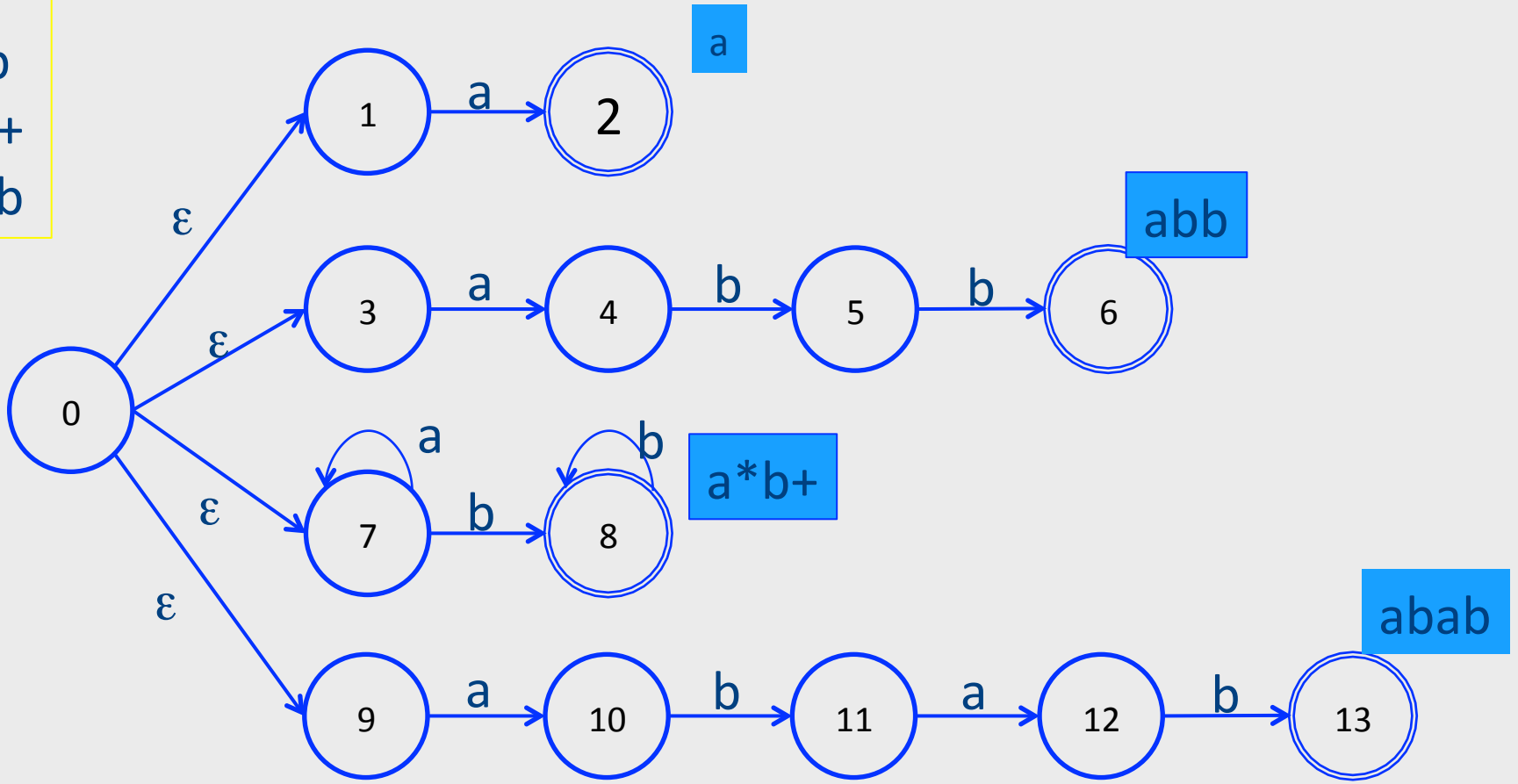
What now?

- Naïve approach: try each automaton separately
- Given a word w:
  - Try $M_1(w)$
  - Try $M_2(w)$
  - …
  - Try $M_n(w)$
- Requires resetting after every attempt

Combine automata

combines

a
abb
a*b+
abab

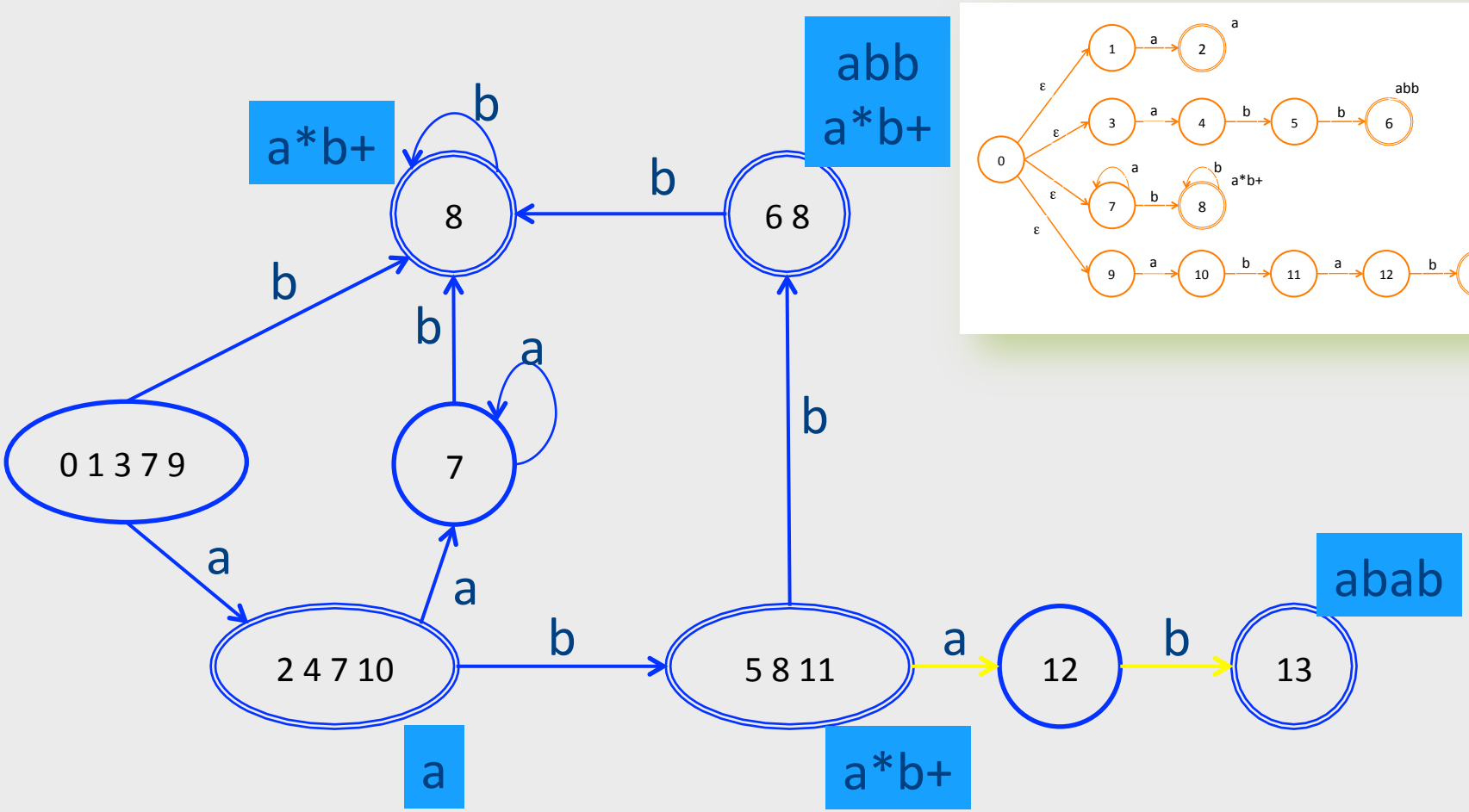


Ambiguity resolution

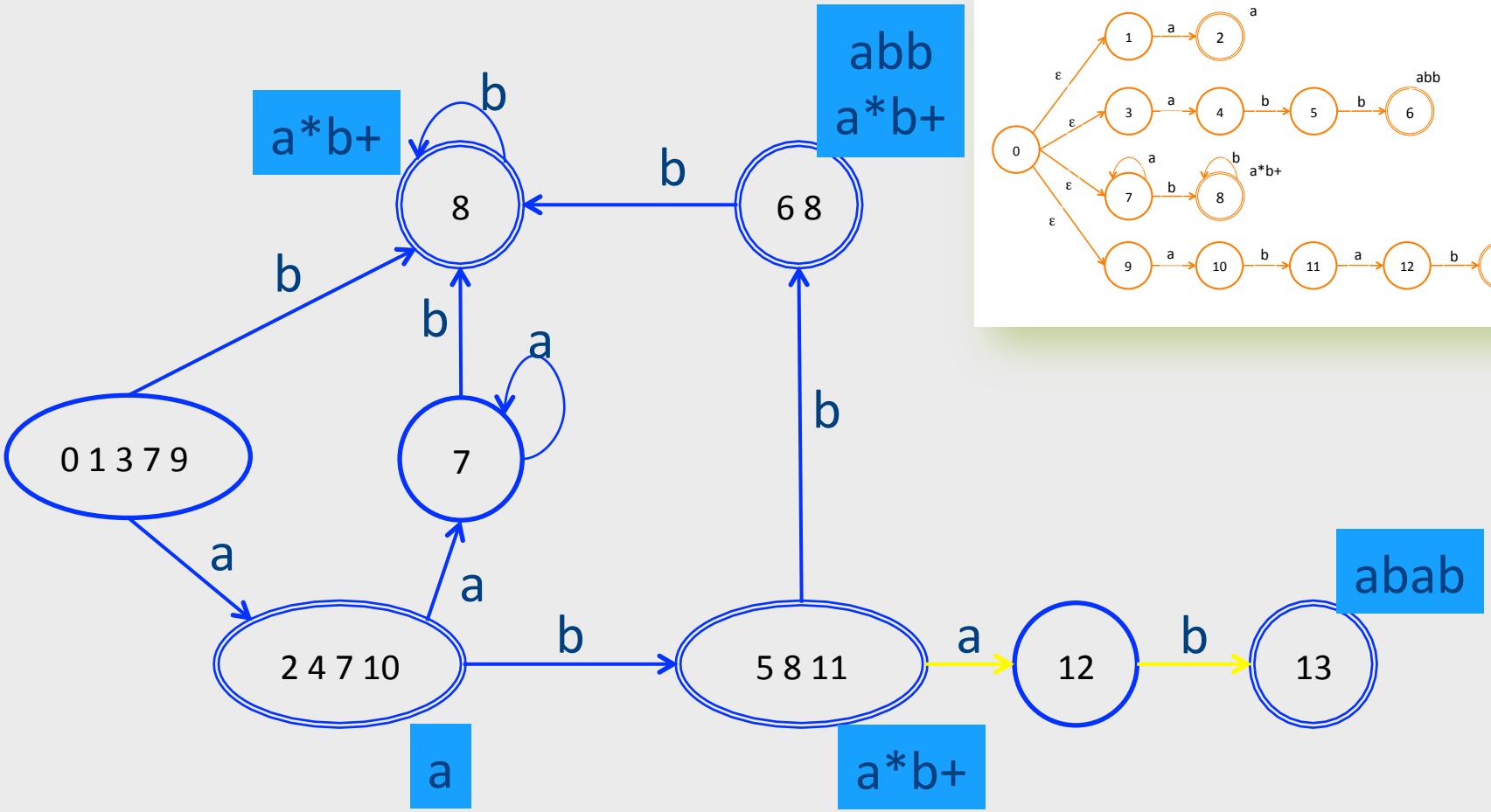
- Recall…
- Longest word
- Tie-breaker based on order of rules when words have same length

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

Corresponding DFA



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

Summary of Construction

- Describes tokens as regular expressions
  - and decides which attributes (values) are saved for each token
- Regular expressions turned into a DFA
  - describes expressions and specifies which attributes (values) to keep
- Lexical analyzer simulates the run of an automata with the given transition table on any input string

A Few Remarks

- Turning an NFA to a DFA is expensive, but
  - Exponential in the worst case
  - In practice, works fine
- The construction is done once per-language
  - At Compiler construction time
  - Not at compilation time

Implementation by Example

```
if                                  { return IF; }
[a-z][a-z0-9]*                      { return ID; }
[0-9]+                              { return NUM; }
[0-9]”.”[0-9]*|[0-9]*”.”[0-9]+       { return REAL; }
(\-\-[a-z]*\n)|(“  “|\n|\t)         { ; }
.                                   { error(); }
```

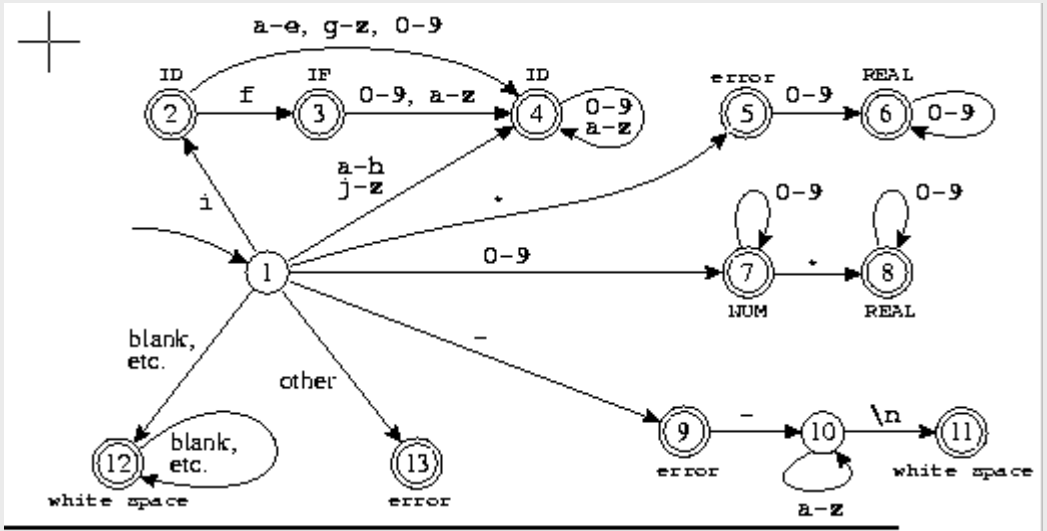


FIGURE 2.4. Combined finite automaton.
From *Modern Compiler Implementation in ML*,
Cambridge University Press, ©1998 Andrew W. Appel

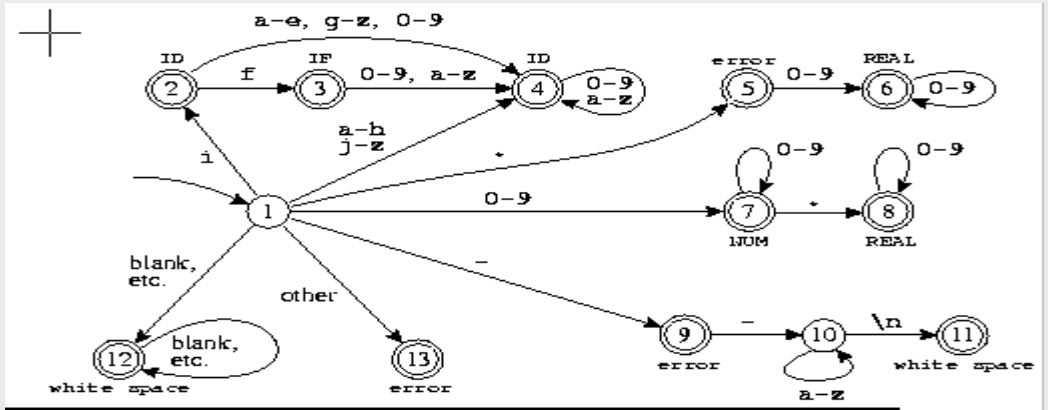


FIGURE 2.4. Combined finite automaton.
From *Modern Compiler Implementation in ML*,
Cambridge University Press, ©1998 Andrew W. Appel

```
int edges[][256]= {
                    /* …, 0, 1, 2, 3, ..., -, e, f, g, h, i, j,  ... */
/* state 0  */   {0, …, 0, 0, …,  0, 0, 0, 0, 0, ..., 0, 0, 0, 0, 0, 0},
/* state 1  */   {13, … , 7, 7, 7, 7,  …,  9,  4, 4, 4, 4, 2, 4, …, 13, 13},
/* state 2  */   {0,  …, 4, 4, 4, 4,  ..., 0,  4, 3, 4, 4, 4, 4, …, 0, 0},
/* state 3  */   {0, …,  4, 4, 4, 4,  …, 0,  4, 4, 4, 4, 4, 4, , 0, 0},
/* state 4  */   {0, …,  4, 4, 4, 4,   …, 0,  4, 4, 4, 4, 4, 4, …, 0, 0},
/* state 5  */   {0, …,  6, 6, 6, 6,   …, 0,  0, 0, 0, 0, 0, 0, …,  0, 0},
/* state 6  */   {0,  …,  6, 6, 6, 6,  …, 0,  0, 0, 0, 0, 0, 0, ..., 0, 0},
/* state 7  */
/* state …  */                        ...
/* state 13 */   {0,  …,  0, 0, 0, 0,  …, 0,  0, 0, 0, 0, 0, 0, …,  0, 0}
};
```

Pseudo Code for Scanner

```
Token nextToken()
{
lastFinal = 0;
currentState = 1 ;
inputPositionAtLastFinal = input;
currentPosition = input;
while (not(isDead(currentState)))  {
       nextState = edges[currentState][*currentPosition];
       if  (isFinal(nextState)) {
              lastFinal = nextState ;
              inputPositionAtLastFinal = currentPosition; }
       currentState = nextState;
       advance currentPosition;
       }
input = inputPositionAtLastFinal ;
return action[lastFinal];
}
```

Example

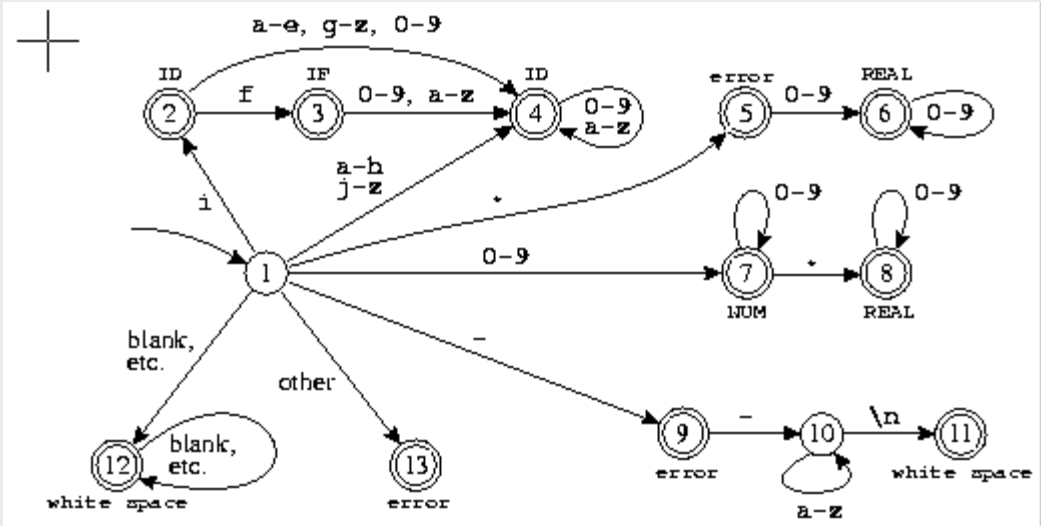


FIGURE 2.4. Combined finite automaton.
From *Modern Compiler Implementation in ML*,
Cambridge University Press, ©1998 Andrew W. Appel

Input: “if --not-a-com”

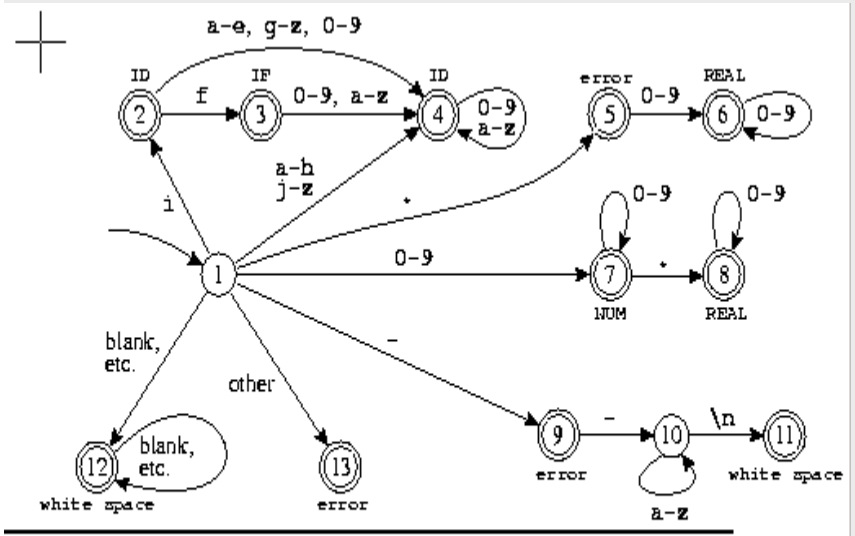


FIGURE 2.4. Combined finite automaton.
From *Modern Compiler Implementation in ML*,
Cambridge University Press, ©1998 Andrew W. Appel

return IF

final	state	input
0	1	if --not-a-com
2	2	if --not-a-com
3	3	if --not-a-com
3	0	if --not-a-com

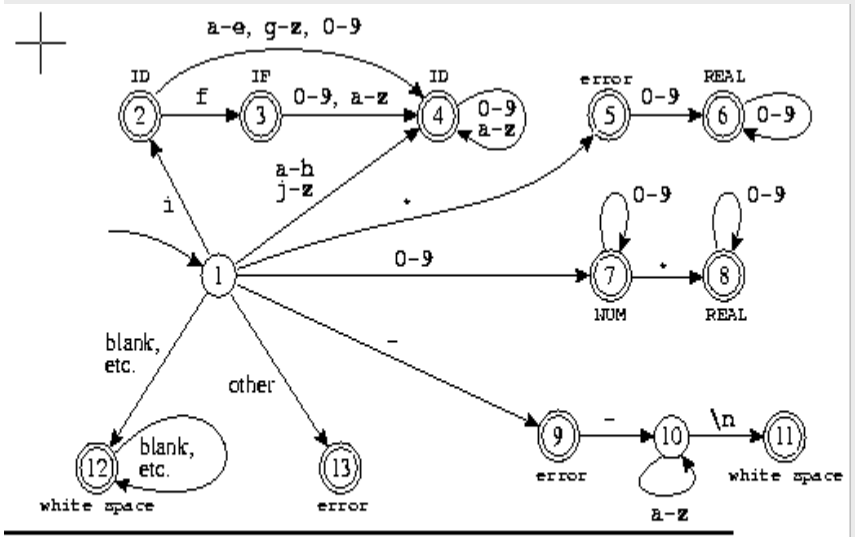


FIGURE 2.4. Combined finite automaton.
From *Modern Compiler Implementation in ML*,
Cambridge University Press, ©1998 Andrew W. Appel

final	state	input
0	1	--not-a-com
12	12	--not-a-com
12	0	--not-a-com

found whitespace

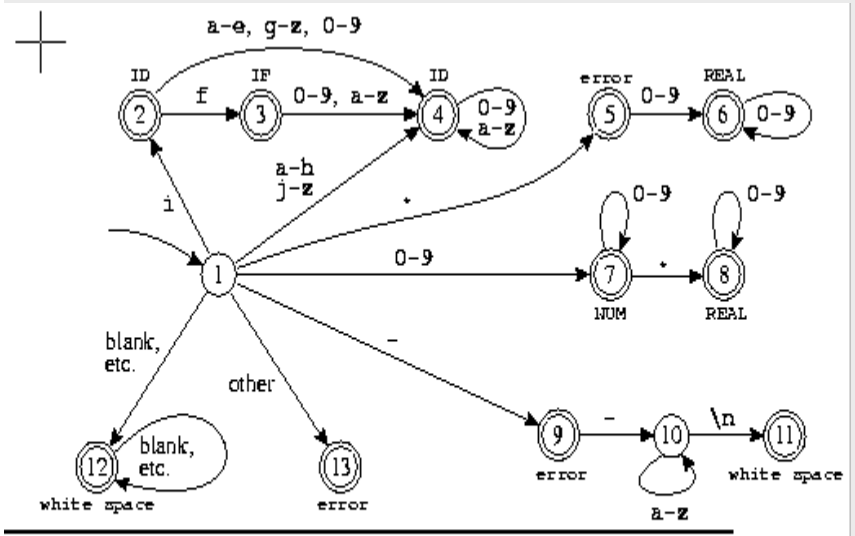


FIGURE 2.4. Combined finite automaton.
From *Modern Compiler Implementation in ML*,
Cambridge University Press, ©1998 Andrew W. Appel

error

final	state	input
0	1	--not-a-com
9	9	--not-a-com
9	10	--not-a-com
9	10	--not-a-com
9	10	--not-a-com
9	0	--not-a-com

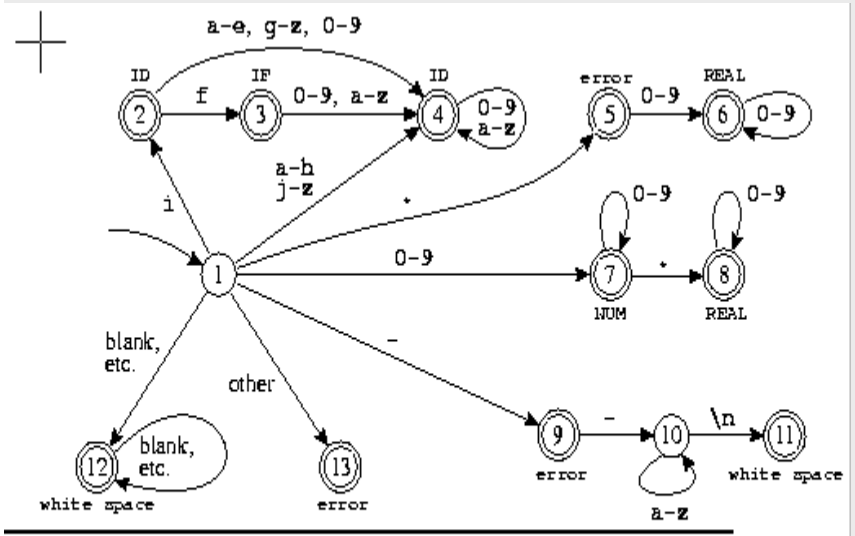


FIGURE 2.4. Combined finite automaton.
From *Modern Compiler Implementation in ML*,
Cambridge University Press, ©1998 Andrew W. Appel

final	state	input
0	1	-not-a-com
9	9	-not-a-com
9	0	-not-a-com

error

Efficient Scanners

- Efficient state representation
- Input buffering
- Using switch and gotos instead of tables

DFSM from Specification

- Create a non-deterministic automaton (NDFA) from every regular expression
- Merge all the automata using epsilon moves (like the | construction)
- Construct a deterministic finite automaton (DFA)
  - State priority
- Minimize the automaton starting
  - separate accepting states by token kinds

NDFA Construction

```
if                      { return IF; }
[a-z][a-z0-9]*          { return ID; }
[0-9]+                  { return NUM; }
```

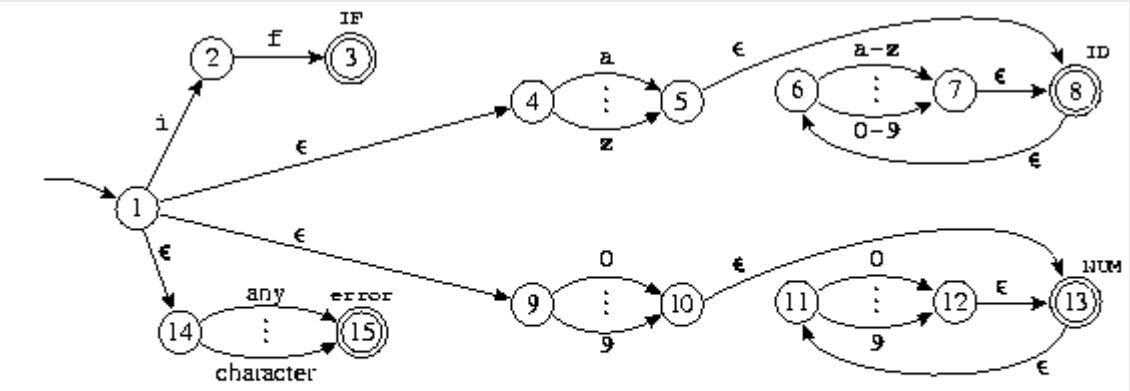


FIGURE 2.7. Four regular expressions translated to an NFA.
From *Modern Compiler Implementation in ML*,
Cambridge University Press, ©1998 Andrew W. Appel

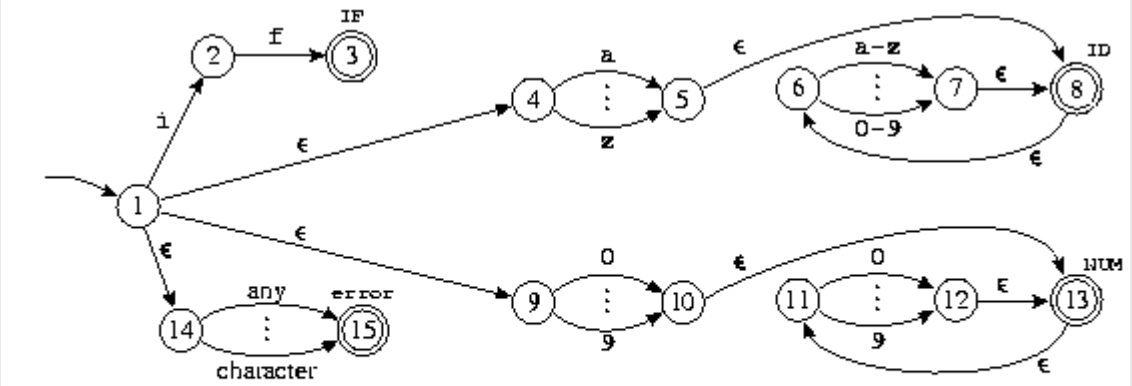


FIGURE 2.7. Four regular expressions translated to an NFA.
From *Modern Compiler Implementation in ML,*
Cambridge University Press, ©1998 Andrew W. Appel

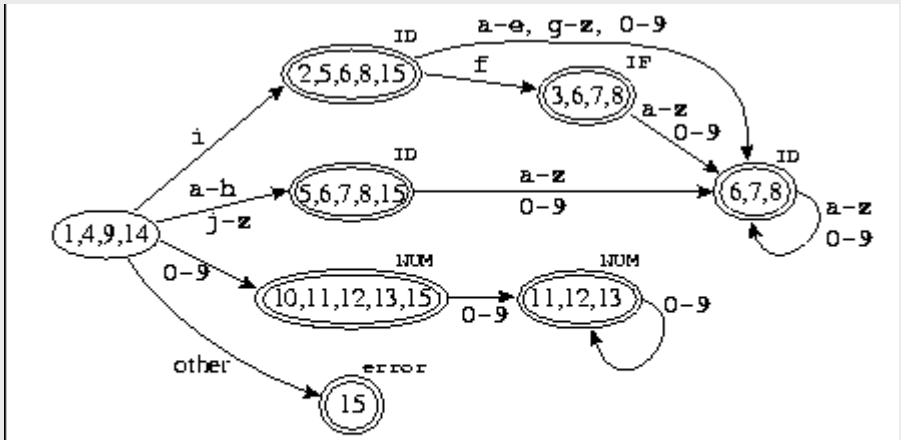


FIGURE 2.8. NFA converted to DFA.
From *Modern Compiler Implementation in ML,*
Cambridge University Press, ©1998 Andrew W. Appel

Minimization

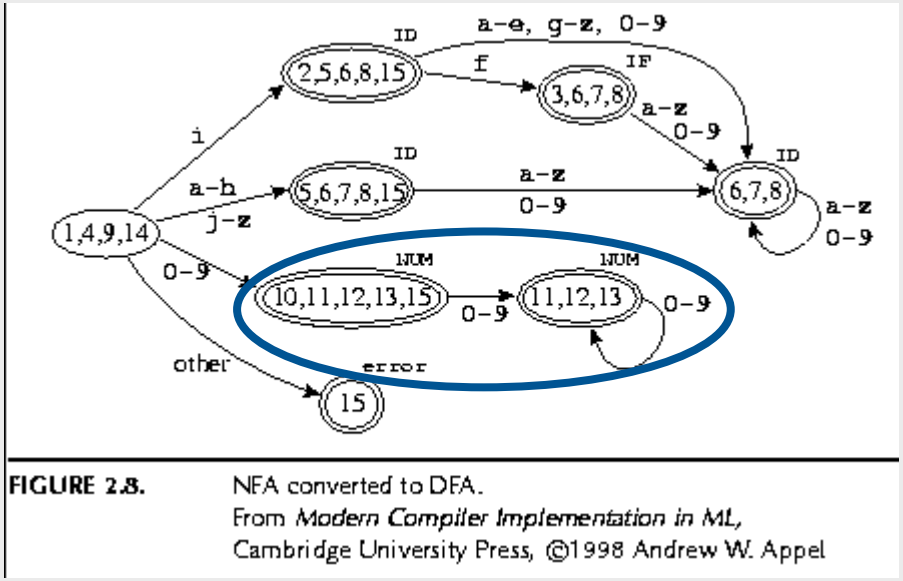
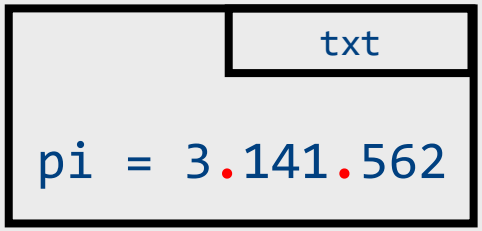


FIGURE 2.8. NFA converted to DFA.
From *Modern Compiler Implementation in ML*,
Cambridge University Press, ©1998 Andrew W. Appel

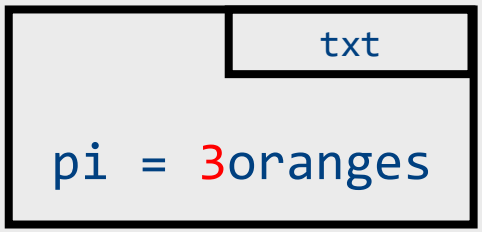
Lecture Outline

- ✓ Role & place of lexical analysis
- ✓ What is a token?
- ✓ Regular languages
- ✓ Lexical analysis
- Error handling
- Automatic creation of lexical analyzers

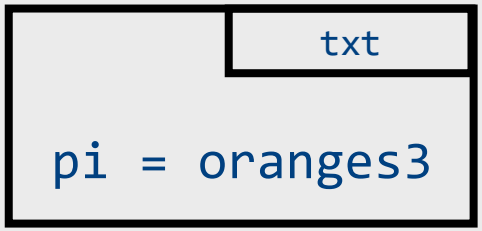
Errors in lexical analysis



Illegal token



Illegal token



<ID,”pi”>, <EQ>, <ID,”oranges3”>

Error Handling

- Many errors cannot be identified at this stage
- Example: “fi (a==f(x))”. Should “fi” be “if”? Or is it a routine name?
  - We will discover this later in the analysis
  - At this point, we just create an identifier token

- Sometimes the lexeme does not match any pattern
  - Easiest: eliminate letters until the beginning of a legitimate lexeme
  - Alternatives: eliminate/add/replace one letter, replace order of two adjacent letters, etc.

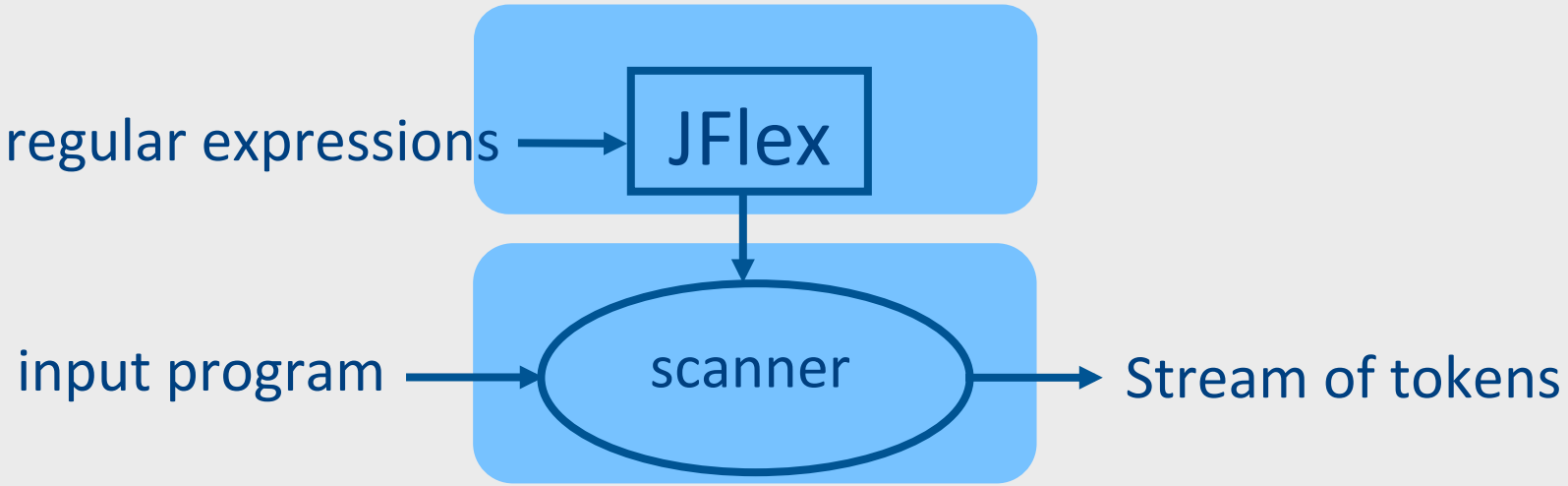
- Goal: allow the compilation to continue
- Problem: errors that spread all over

Lecture Outline

- ✓ Role & place of lexical analysis
- ✓ What is a token?
- ✓ Regular languages
- ✓ Lexical analysis
- ✓ Error handling
- • Automatic creation of lexical analyzers

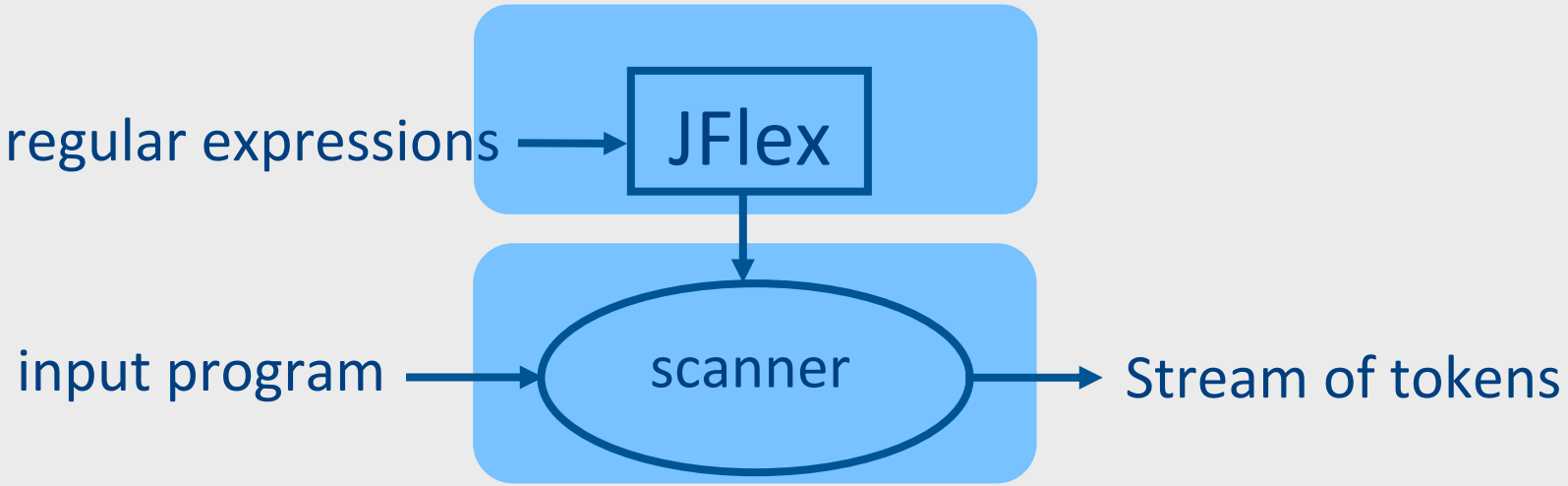
Use of Program-Generating Tools

- Automatically generated lexical analyzer
  - Specification ➔ Part of compiler
  - Compiler-Compiler



Use of Program-Generating Tools

- Input: regular expressions and actions
  - Action = Java code
- Output: a scanner program that
  - Produces a stream of tokens
  - Invoke actions when pattern is matched



Line Counting Example

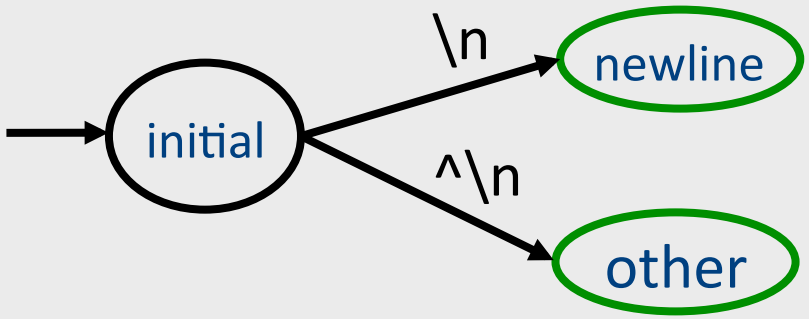
- Create a program that counts the number of lines in a given input text file

Creating a Scanner using Flex

```
int num_lines = 0;
%%
\n      ++num_lines;
.       ;
%%
main() {
  yylex();
  printf( "# of lines = %d\n", num_lines);
}
```

Creating a Scanner using Flex

```
int num_lines = 0;
%%
\n      ++num_lines;
.       ;
%%
main() {
  yylex();
  printf( "# of lines = %d\n", num_lines);
}
```



JFLex Spec File

User code: Copied directly to Java file

%%

JFlex directives: macros, state names

%%

Lexical analysis rules:

- Optional state, regular expression, action
- How to break input to tokens
- Action when token matched

Possible source of javac errors down the road

DIGIT= [0-9]
LETTER= [a-zA-Z]

YYINITIAL

{LETTER}
({LETTER}|{DIGIT})*

Creating a Scanner using JFlex

```
import java_cup.runtime.*;
%%
%cup
%{
  private int lineCounter = 0;
%}

%eofval{
  System.out.println("line number=" + lineCounter);
  return new Symbol(sym.EOF);
%eofval}

NEWLINE=\n
%%
{NEWLINE}        { lineCounter++; }
[^{NEWLINE}]     { }
```

Catching errors

- What if input doesn't match any token definition?
- Trick: Add a "catch-all" rule that matches any character and reports an error
  - Add after all other rules

A JFlex specification of C Scanner

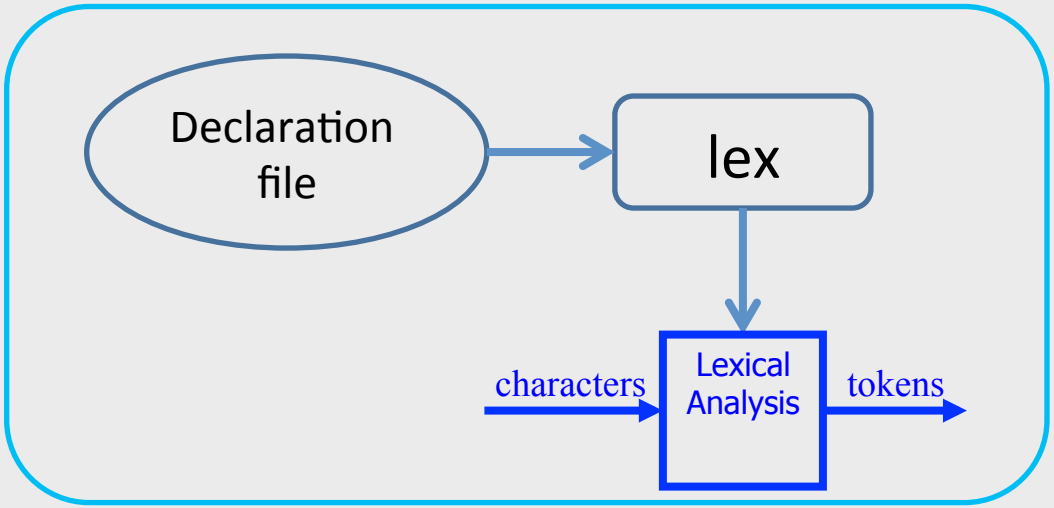
```
import java_cup.runtime.*;
%%
%cup
%{
  private int lineCounter = 0;
%}
Letter= [a-zA-Z_]
Digit= [0-9]
%%
”\t”     { }
”\n”     { lineCounter++; }
“;”      { return new  Symbol(sym.SemiColumn);}
“++”     { return new  Symbol(sym.PlusPlus); }
“+=”     { return new  Symbol(sym.PlusEq); }
“+”      { return new  Symbol(sym.Plus); }
“while” { return new  Symbol(sym.While); }
{Letter}({Letter}|{Digit})*
         { return new  Symbol(sym.Id, yytext() ); }
“<=”     { return new  Symbol(sym.LessOrEqual); }
“<”      { return new  Symbol(sym.LessThan); }
```

A Simplified Scanner for C

```
Token nextToken()
{
char c ;
loop: c = getchar();
switch (c){
  case ` `: goto loop ;
  case `;`: return SemiColumn;
  case `+`:
    c = getchar() ;
    switch (c) {
      case `+': return PlusPlus ;
      case '=’  return PlusEqual;
      default:  ungetc(c); return Plus;
    };
  case `<`: …
  case `w`: …
 }
```

Automatic Construction of Scanners

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



Intuitively:

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

Lecture Outline

- ✓ Role & place of lexical analysis
- ✓ What is a token?
- ✓ Regular languages
- ✓ Lexical analysis
- ✓ Error handling
- ✓ Automatic creation of lexical analyzers

Start States

- It may be hard to specify regular expressions for certain constructs
  - Examples
    - Strings
    - Comments
- Writing automata may be easier
- Can combine both
- Specify partial automata with regular expressions on the edges
  - No need to specify all states
  - Different actions at different states

Missing

- Creating a lexical analysis by hand
- Table compression
- Symbol Tables
- Nested Comments
- Handling Macros

What does Lexical Analysis do?

- Input: program text (file)
- Output: sequence of tokens

- Read input file
- Identify language keywords and standard identifiers
- Handle include files and macros
- Count line numbers
- Remove whitespaces
- Report illegal symbols

- [Produce symbol table]

Lexical Analysis – Impl.

- 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

Automatic Construction of Scanners

- For most programming languages lexical analyzers can be easily constructed automatically
  - Exceptions:
    - Fortran
    - PL/1
- Lex/Flex/Jlex/JFlex are useful tools

Next Week

- Syntax analysis (aka parsing)

- Lecture in **this room**
  - Trubowicz 101 (Law school)

NFA vs. DFA

Automaton	SPACE	TIME
NFA	O(|r|)	O(|r|*|w|)
DFA	O(2^|r|)	O(|w|)

- (a|b)*a(a|b)(a|b)…(a|b)

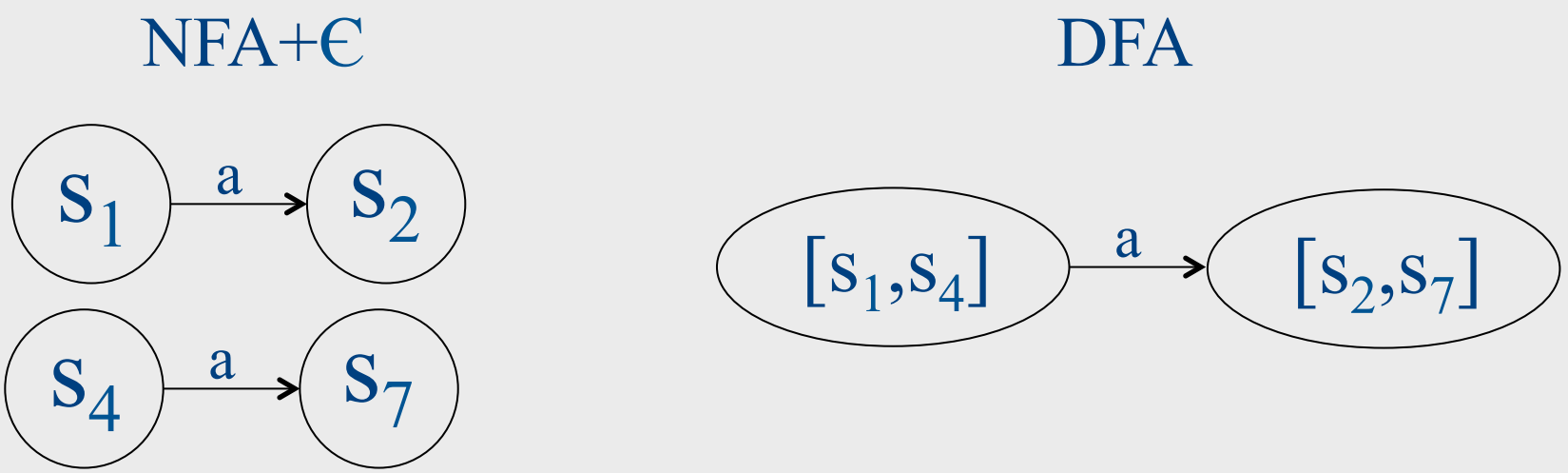
n times

From NFA+Є to DFA

- Construction requires O(*n*) states for a reg-exp of length *n*
- Running an NFA+Є with *n* states on string of length *m* takes O($m \cdot n^2$) time
- Solution: determinization via subset construction
  - Number of states worst-case exponential in *n*
  - Running time O(*m*)

Subset construction

- For an NFA+Є with states M={s_1,…,s_k}
- Construct a DFA with one state per set of states of the corresponding NFA
  - M'={ [], [s_1], [s_1,s_2], [s_2,s_3], [s_1,s_2,s_3], …}
- Simulate transitions between individual states for every letter



Subset construction

- For an NFA+Є with states M={s_1,…,s_k}
- Construct a DFA with one state per set of states of the corresponding NFA
  - M'={ [], [s_1], [s_1,s_2], [s_2,s_3], [s_1,s_2,s_3], …}
- Extend macro states by states reachable via Є transitions