# Lexical Analysis

Greta Yorsh

## Admin

- PA0 is **individual submission**
- PA0-PA3:
  - all deadlines are on **Sundays at 23:55**
  - QMUL github required to submit
- Teams for PA1-PA3
  - sign up as soon as possible on QM+
  - to form teams: talk to your classmates, post on QM+ forum…
- Essential information for completing PA1-PA3
  - Tutorials: requirements
  - Labs: implementation
- Cool Reference Manual: see QM+ each week for relevant sections

## FAQ

- Can I use Windows for programming assignments?
  - Not recommended
  - Reference compiler works on CentOS and OS X
  - Marking scripts will run on school's machines with CentOS 7
- I get an error message
  ```
  coolc: command not found
  ```
- How to add coolc to the PATH?
  ```
  export PATH=$PATH:`pwd`/bin
  export PATH=$PATH:~/cool/distro/bin
  ```
- You need to execute this command every time you open a new shell
  Or, you do it automatically by adding this command to your bash profile
- Other useful commands:
  ```
  echo $PATH
  which coolc
  ```

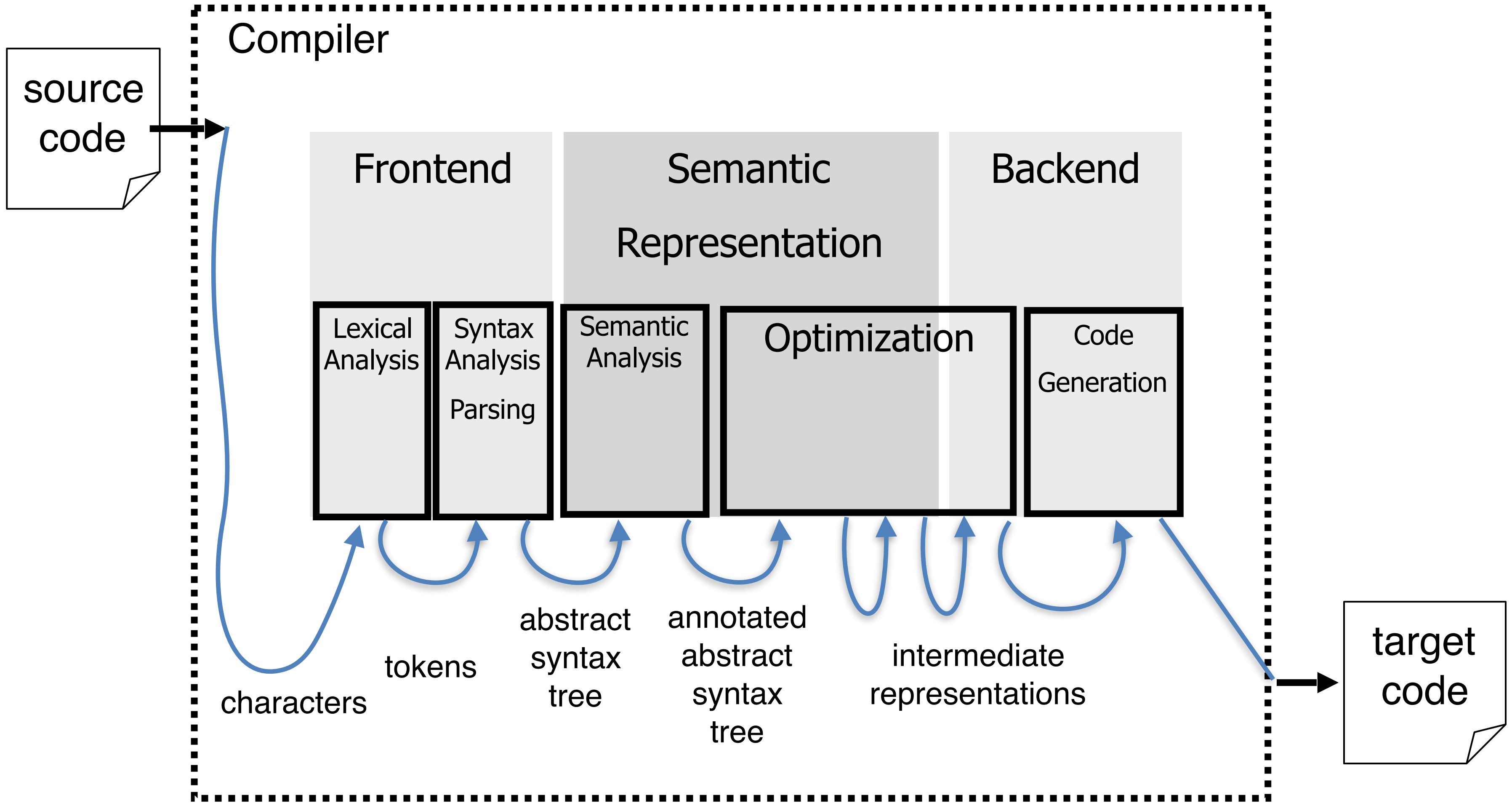
## Today

- Recap: anatomy of a compiler
- Lexical analysis: overview and examples
- Regular expressions and finite state automata
- Identifying tokens
- Lexer generators

# Recap

- Compiler is a **program** that translates code from **source** language to **target** language
- Compilers play a central role
  - bridge from high-level programming languages to machines
  - many useful techniques
  - many useful tools (e.g., lexer/parser generators)
- Compiler vs Interpreter
- Just-In-Time compilation
- Time of events: compiler, linker, loader, runtime
- Bootstrapping a compiler
- Compiler constructed from modular phases

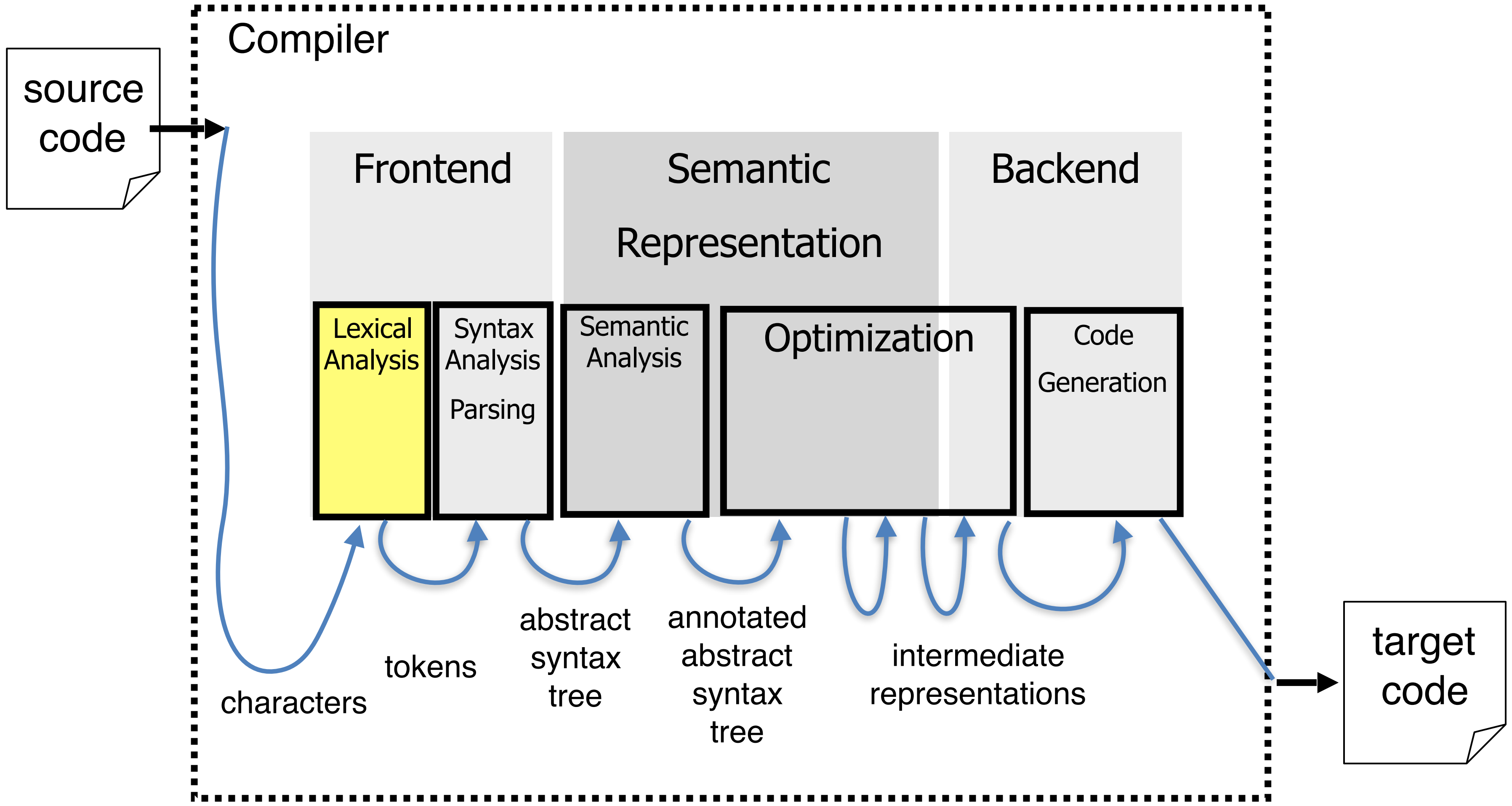
# Anatomy of a modern compiler



## How to precisely define programming language?

- Layered structure of language definition
- Start with the set of letters in language
- **Lexical structure** identifies “words” in language: each word is a sequence of letters
- **Syntactic structure** identifies “sentences” in language: each sentence is a sequence of words
- **Semantics** defines meaning of program: specifies what result should be for each input

# Anatomy of a modern compiler

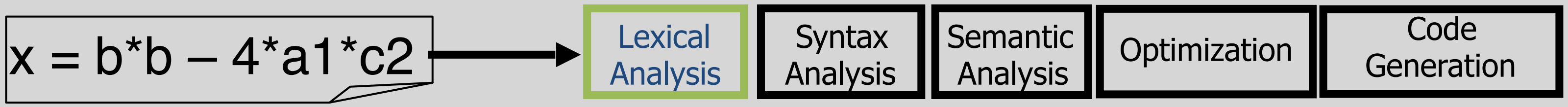


# From characters to tokens

x = b*b – 4*a1*c2

$\downarrow$

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **ID,”x”** | EQ | **ID,”b”** | MULT | **ID,”b”** | MINUS | INT,4 | MULT | **ID,”a1”** | MULT | **ID,”c2”** |



# What is a token?

- Intuitively, a “word” in the source language
- Usually a pair of name and value
- Anything that should appear in the input to syntactic analysis
- Examples

```
if (x*4) return;
```

| | |
|---|---|
| • numbers | INT(4) |
| • identifiers | **ID(x)** |
| • keywords | IF RETURN |
| • punctuation | LPAREN RPAREN SEMI |
| • operators | BINOP("*") |

# Typical lexer

- Identify language keywords
- Recognize standard identifiers
- Remove whitespaces
- Report illegal symbols
- Count line numbers
- Handle include files and macros
- Produce symbol table

## Design decisions

- How to describe tokens unambiguously
- How to break text up into tokens
    - `if (x==0) a = x << 1;`
    - `if (x==0) a = x < 1;`
- How to  tokenize efficiently
    - tokens may have similar prefixes
    - look at each character only about once

## Some basic terminology

- **Lexeme** - **a sequence of characters** separated from the rest of the program according to a convention (space, semi-column, comma, ...)
- **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”
- **Token** - pattern name and its attributes

# Example Tokens

| **Pattern name** | **Example lexeme** |
|---|---|
| Identifier | foo |
| NUM | 42 |
| FLOATNUM | 3.141592654 |
| STRING | “so long, and thanks for all the fish” |
| LPAREN | ( |
| RPAREN | ) |
| IF | if |
| … | |

# Strings with special handling

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

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

## Example

```
1     void match0(char *s) /* find a zero */
2     {
3       if (!strncmp(s, “0.0”, 3))
4         return 0. ;
5     }
6
```

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

## Error Handling

- Many errors cannot be identified during lexical analysis
- 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 for "fi"

# Error Handling

- Sometimes the lexeme does not match any pattern
- Goal: allow the compilation to continue
- Easiest: eliminate letters until the beginning of a legitimate lexeme
- Alternatives: eliminate/add/replace one letter, reorder two adjacent letters…
- Problem: errors that spread all over

## How can we define tokens?

- Keywords – easy!
  - **if**, **then**, **else**, **for**, **while**, …
- Identifiers?
- Numerical Values?
- Strings?

- Provide a formal language for patterns

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

# Regular Expressions over Σ

| **Basic Patterns** | **Matching** |
|---|---|
| x | A single letter ‘x’ from the alphabet Σ |
| . | Any character from Σ, 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** | |
| $R_1R_2$ | An R1 followed by R2 |
| $R_1 \mid R_2$ | Either an R1 or R2 |
| **Grouping** | |
| (R) | R itself |

# Examples

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

# Examples

- ab*|cd? = { ab, abb, abbb, ... , c, cd }
- (a|b)* = {a, b, aa, ab, ba, bb, aaa, aab, ...}
- (0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9)* = natural

# 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

- IF = if
- THEN = then
- RELOP = < | > | <= | >= | = | <>

- DIGIT = 0-9
- DIGITS = DIGIT+

# Example: floating point numbers

- Examples
  - 2.1
  - 2.1**E**+3 represents $2*10^{3}$ = 2100
  - 2.1**E**-3 represents $2.1*10^{-3}$= 0.0021

- NUMBER = ( 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 )+
  ( . ( 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 )+
  ( **E** (+|-)? ( 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 )+)?)?

- Using shorthands it can be written as
  NUMBER = DIGITS (.DIGITS ( **E** (+|-)? DIGITS )?)?
  DIGITS = DIGIT+
  DIGIT = 0-9

## Example: decimal representation of rationals

- Rational numbers in decimal representation **no leading zeros, no ending zeros**

DIGIT = 1|2|…|9
DIGIT0 = 0 | DIGIT
NUM = DIGIT  DIGIT0*
FRAC = DIGIT0* DIGIT
POS = NUM | .FRAC | NUM.FRAC
R =  0 | POS | -POS

| legal | illegal |
|---|---|
| 0 | 007 |
| 123.757 | 1.30 |
| .93333 | 0.301 |
| 10 | 10.0 |
| -34.6 | 0.0 |

## Example: integers without leading zeros

DIGIT = 1|2|…|9

DIGIT0 = 0 | DIGIT
POS = DIGIT DIGIT0*
INT = 0 | POS | -POS

| legal | illegal |
|---|---|
| 0 | 007 |
| 123 | -078 |
| -120 | 1.3 |

## Ambiguity

- if = if
    - ID = LETER_ (LETTER_ | DIGIT)*
    - "if" is a valid word in the language of identifiers
    - "if" is also a keyword
    - what should the token stream 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?

# Main reading routine

```
Token nextToken() {
  while(c = getchar())
  switch (c){
    case ` `: continue;
    case `;`: return SemiColumn;
    case `+`:
    c = getchar() ;
    switch (c) {
      case `+': return PlusPlus ;
      case '='  return PlusEqual;
      default:  ungetc(c); return Plus;
    };
    case is_letter(c) : return recognize_identifier(c);
    case is_digit(c)  : return recognize_number(c);
…
 }
```

```
bool is_uc_letter(char c) { return ('A'<= (c) && (c) <= 'Z') }
bool is_lc_letter(char c) ('a'<= (c) && (c) <= 'z')
bool is_letter(char c) (is_uc_letter(c) || is_lc_letter(c))
bool is_digit(char c) ('0'<= (c) && (c) <= '9')
```

## 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

## Overview

- 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

# Reminder: Finite-State Automaton

- **Deterministic**
- **DFA** $M = (\Sigma, Q, \delta, q_0, F)$
  - $\Sigma$ is an alphabet
  - $Q$ is a **finite** set of states
  - $q_0 \in Q$ is the **initial** state
  - $F \subseteq Q$ is the set of **final** states
  - $\boldsymbol{\delta : Q \times \Sigma \rightarrow Q}$ is a transition function

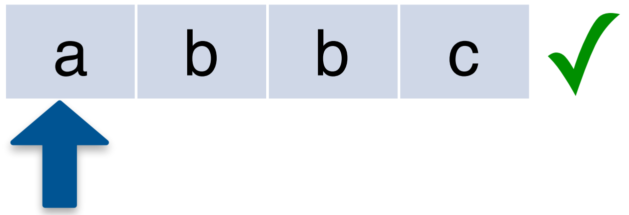
# Reminder: Finite-State Automaton

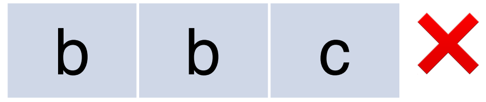
- **Non-Deterministic**
- **NFA** M = ($\Sigma$,Q,$\delta$,$q_0$,F)
  - $\Sigma$ is an alphabet
  - Q is a **finite** set of states
  - $q_0 \in Q$ is the **initial** state
  - $F \subseteq Q$ is the set of **final** states
  - $\boldsymbol{\delta : Q \times (\Sigma \cup \{\varepsilon\}) \rightarrow 2^Q}$ is a transition function

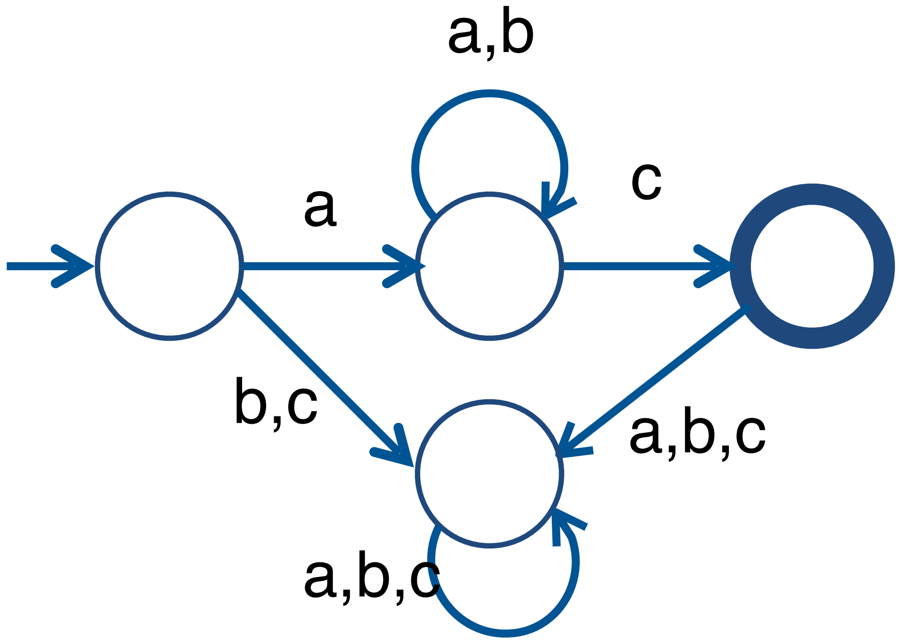
- Possible $\varepsilon$-transitions
- For word w, M can reach a number of states or get stuck.
- If **some state** reached is final, M accepts w.

# Example DFA

- Σ = {a,b,c}
- Missing transition means **stuck** and leads to reject
- Words are scanned left-to-right
- Maintain the **current state** during scan
- Accept iff the current state is final upon reaching end of input





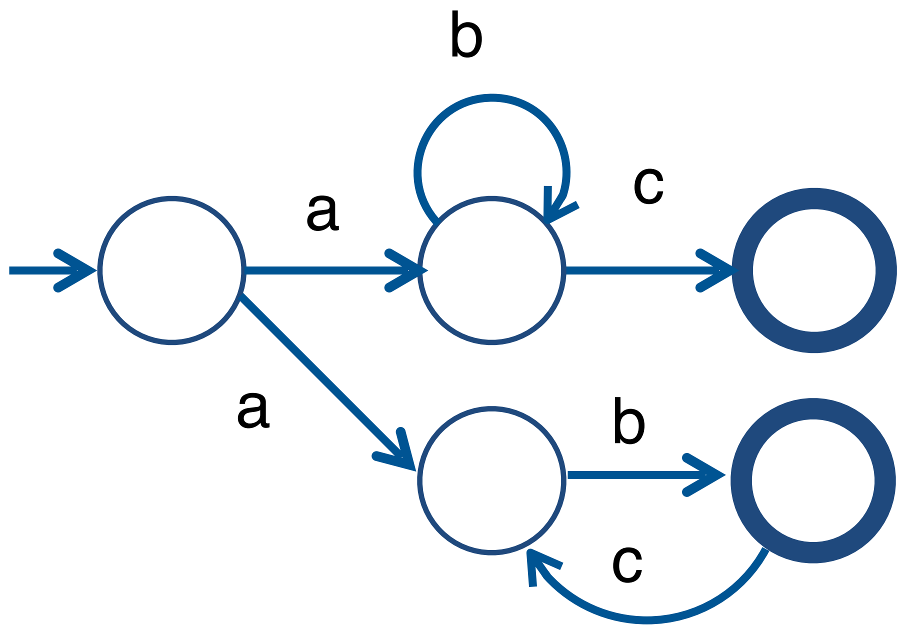


# Example NFA

- Allow multiple transitions from given state with the same label
- Maintain a **set of current states**
- Accept iff one of current states is final upon reaching end of input

| a | b | c | ✓ |
|---|---|---|---|

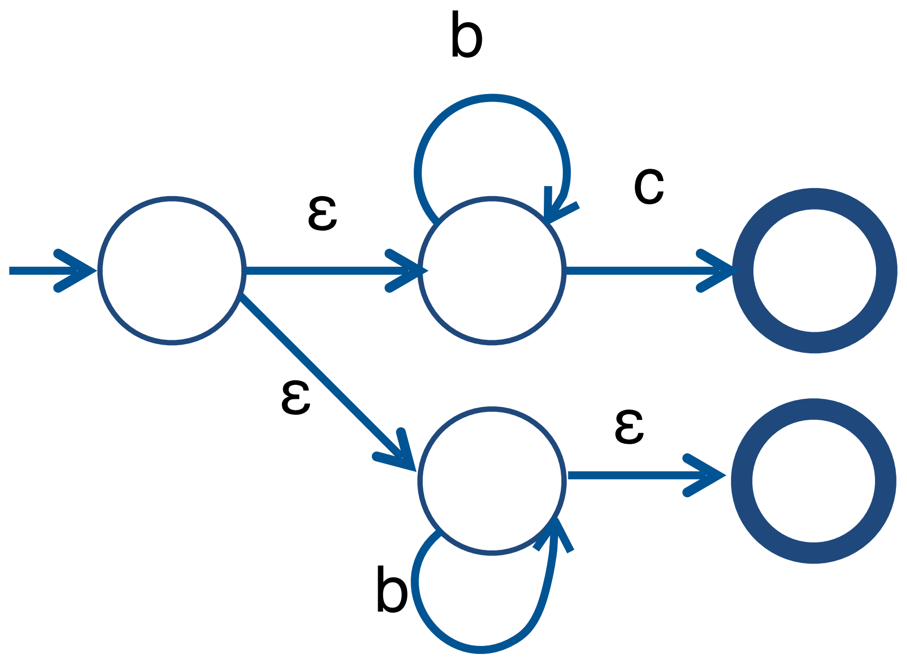
| a | b | c | b | c | ✗ |
|---|---|---|---|---|---|



# Example: NFA with ε transitions

- Make ε transition without reading input
- Maintain a **set of current states**
- Accept iff one of current states is final upon reaching end of input

| | | | | | |
|---|---|---|---|---|---|
| b | b | c | ✓ | | |
| b | b | b | b | b | ✓ |
| b | b | b | b | a | ✗ |



# From regular expressions to NFA

- Definitions
  - L(R) maps regular expression R
    to the set of words over alphabet Σ **described by R**
  - L(M) maps a finite state automaton M
    to the set of words over alphabet Σ **accepted by M**
- Theorem
  - For every R, there exists M such that L(M)=L(R)

## Semantics of regular expressions

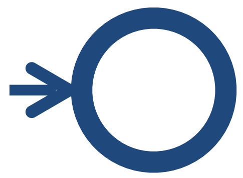
- $L(\varepsilon)=\{\text{“”}\}$
- $L(a)=\{\text{“a”}\}$
- $L(R_1|R_2)=L(R_1)\cup L(R2)$
- $L(R_1R_2)=\{w_1w_2 \mid w_1\in L(R_1),\ w_2\in L(R_2)\}$
- $L(R^*)=\cup\{\ L(R^k) \mid k\geq 0\ \}$

## From regular expressions to NFA

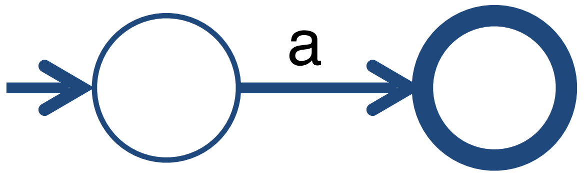
- Associate each regular expression R with an NFA with the following properties
  - exactly one final state
  - no transitions out of the final state
  - no transitions into the inital state
  - accepts the same language L(R)
- Bottom-up construction on the syntax of R

# Basic constructs

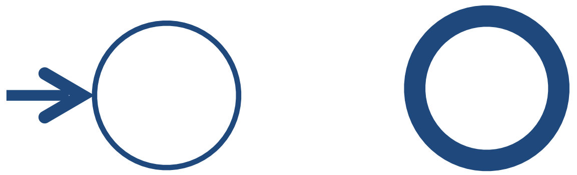
R = ε



R = a

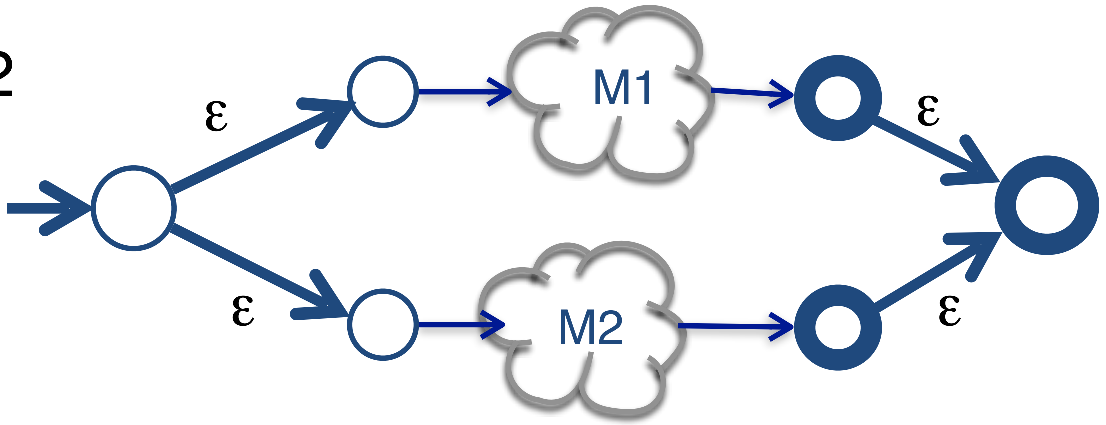


R = ϕ

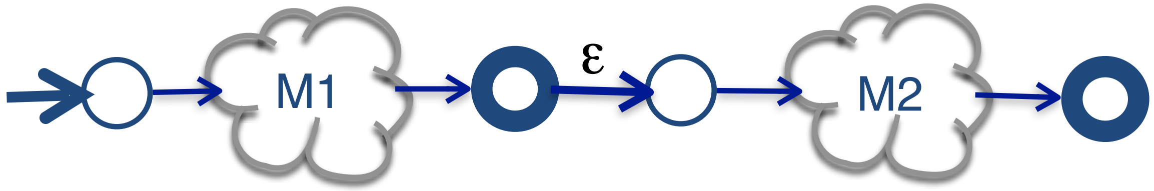


# Composition

R = R1 | R2

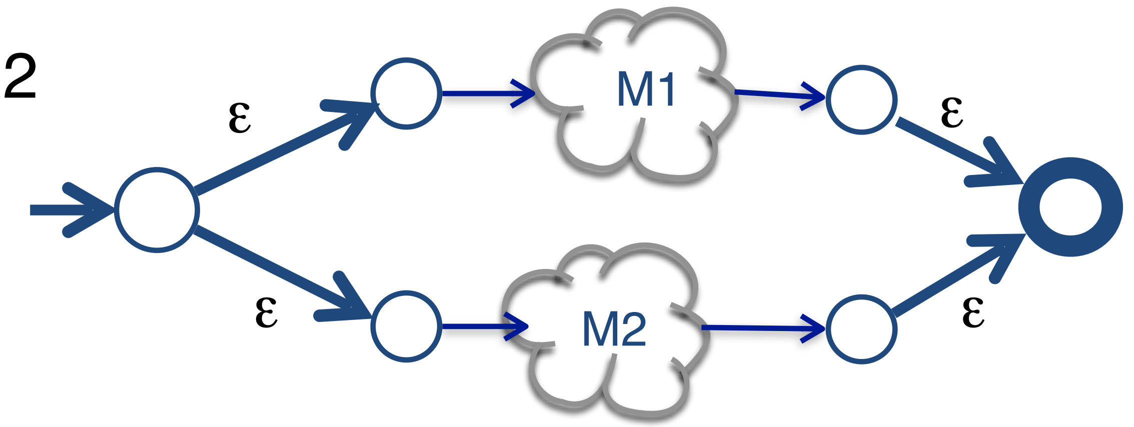


R = R1R2

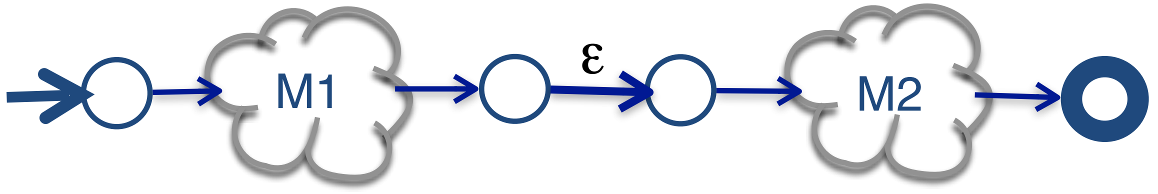


# Composition

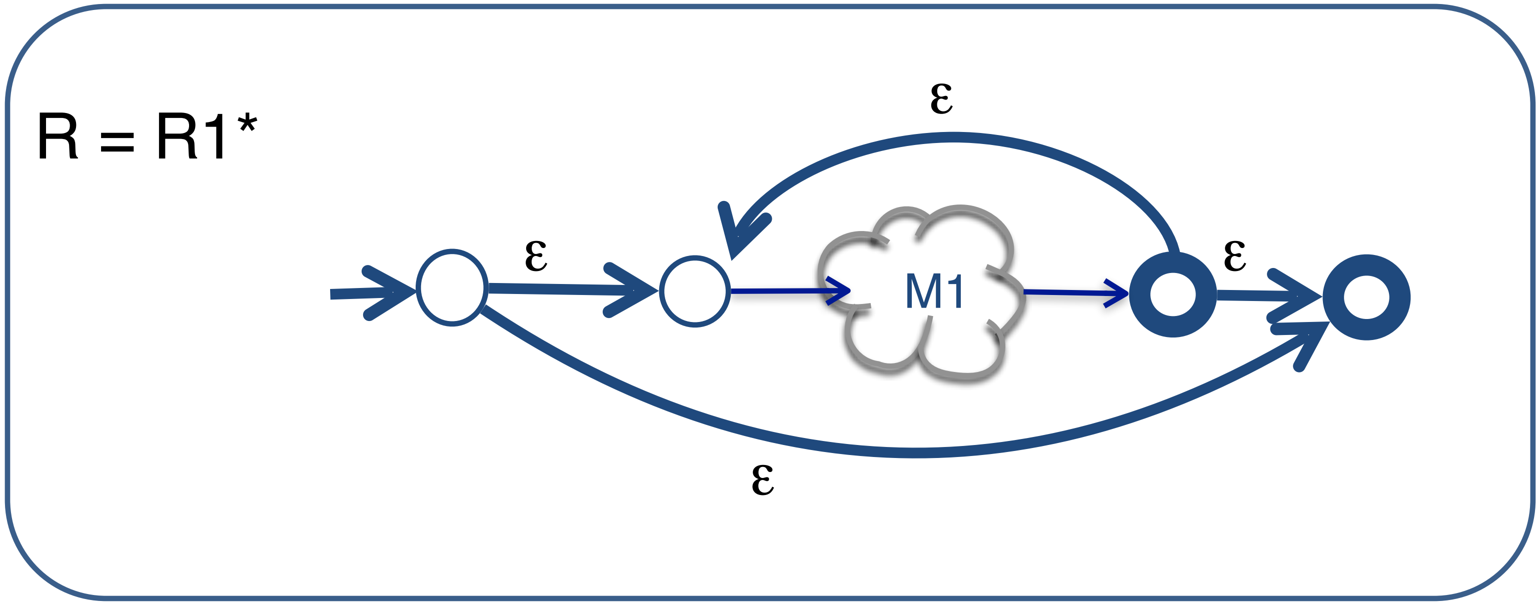
R = R1 | R2



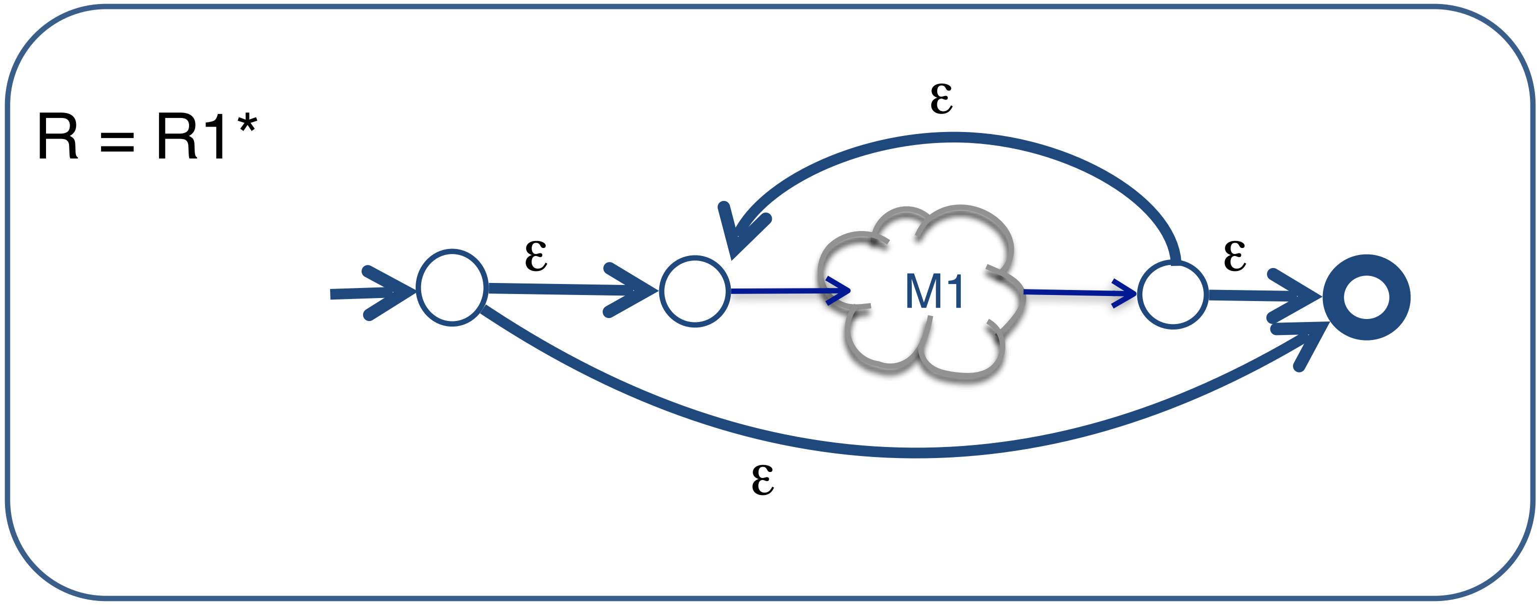
R = R1R2



# Repetition



# Repetition



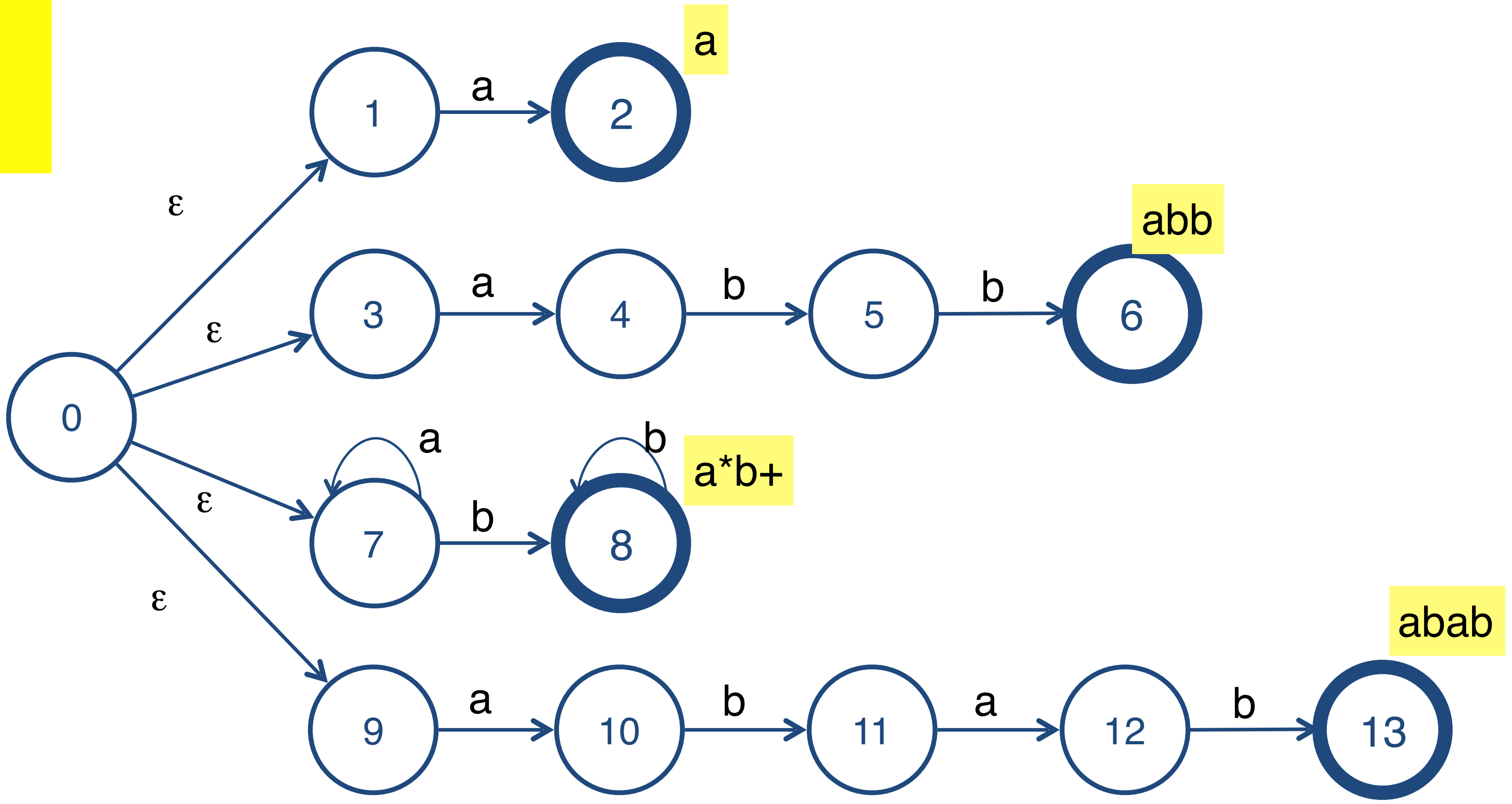
## What now?

- Tokens are defined by regular expressions $R_1 \ldots R_n$
- Construct an NFA $M_i$ for each regular expression $R_i$
- Naïve approach: try each automaton separately
  - given a word w
    - try $M_1(w)$
    - try $M_2(w)$
    - …
    - try $M_n(w)$
  - requires resetting w after every try
- Better approach: combine all $M_1..M_n$ into a single NFA

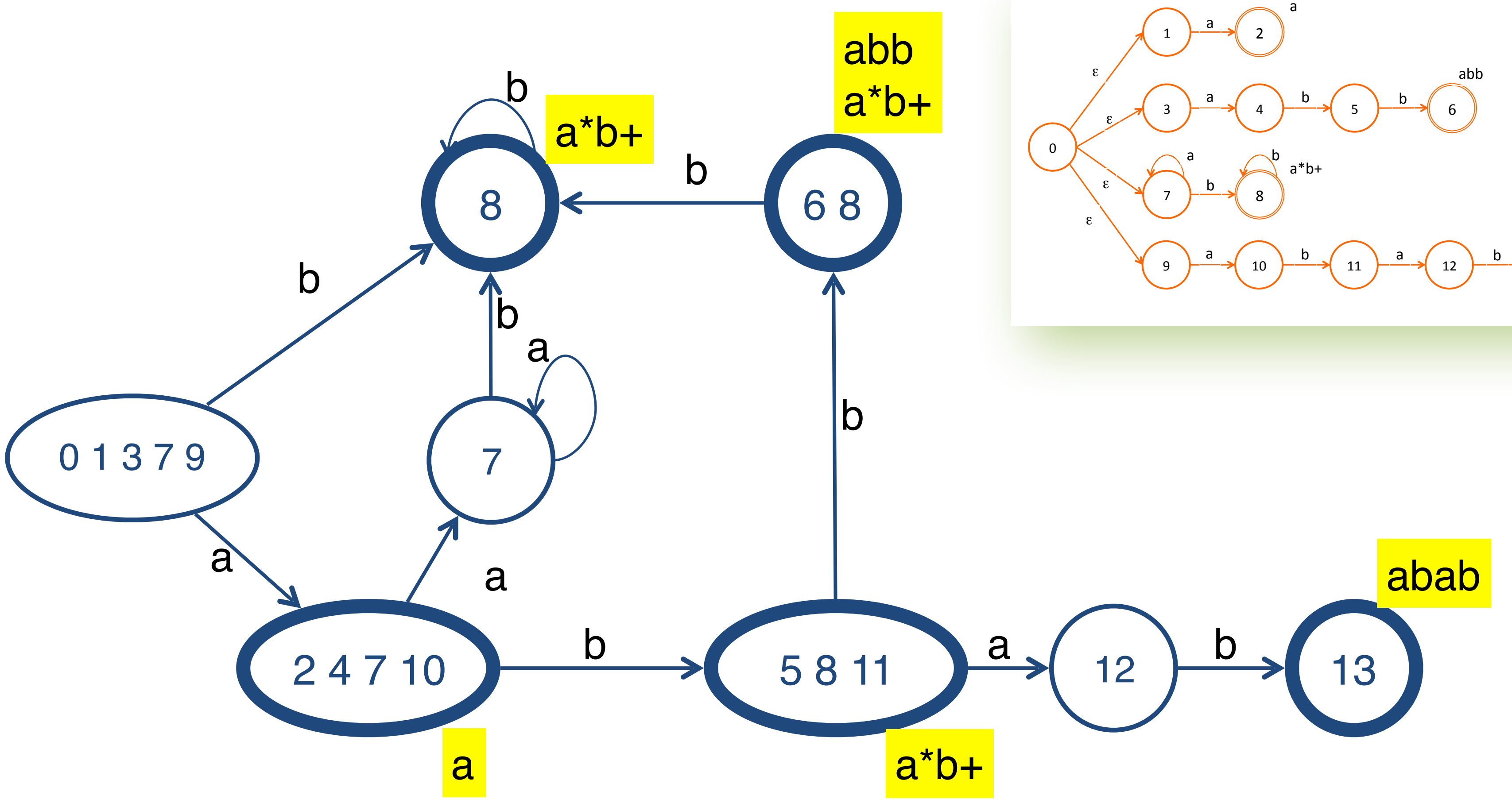
# Combine automata

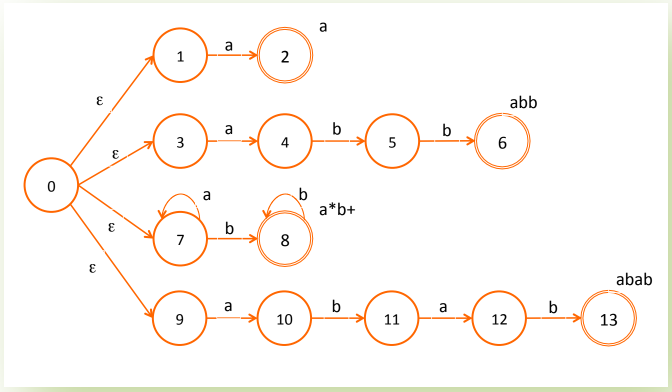
combines

a
abb
a*b+
abab



# Corresponding DFA

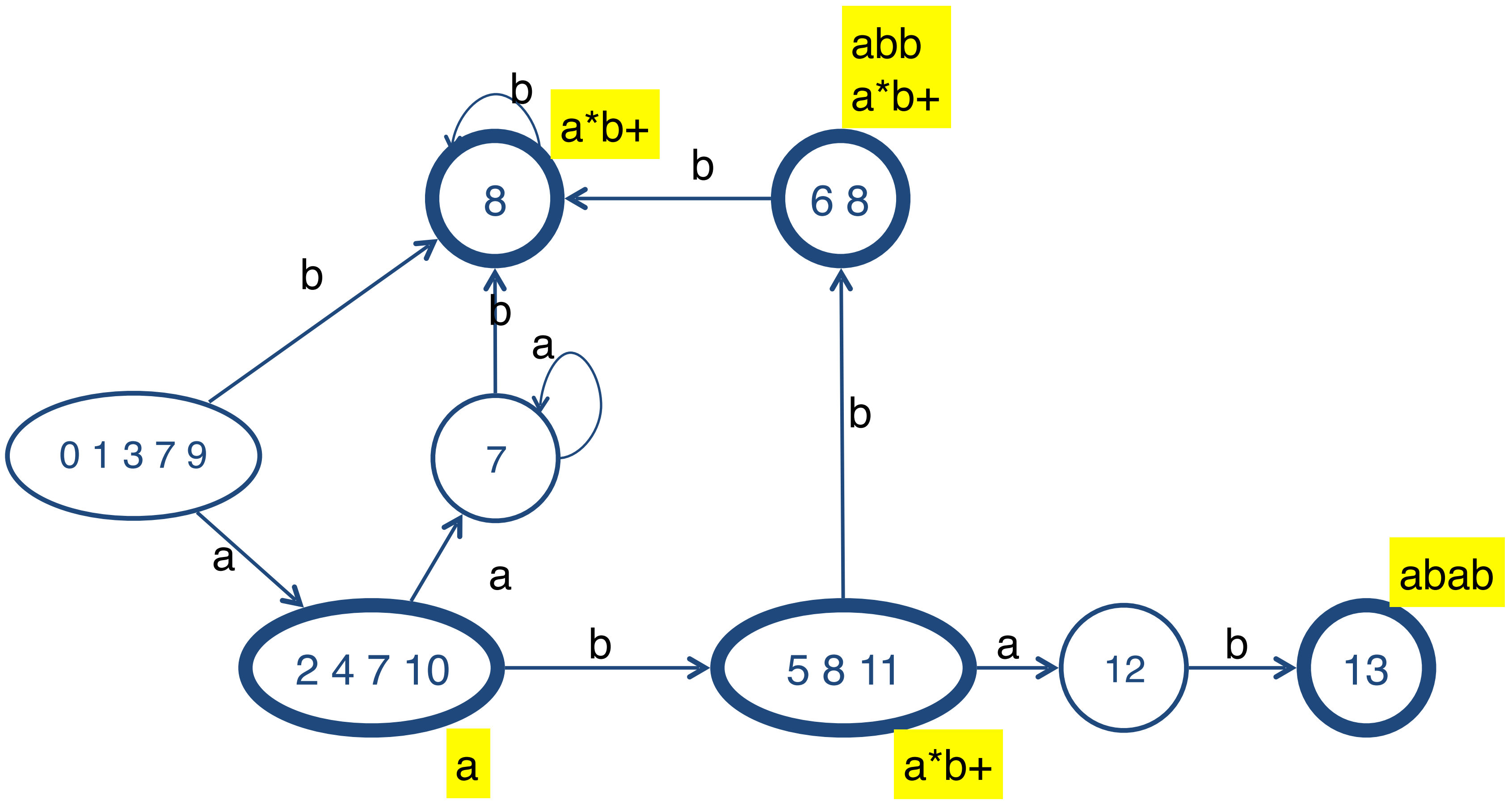




## Scanning with DFA

- Run DFA on the input
- Remember **last-seen final state** and the corresponding position in the input
- When **stuck**
  - return the token corresponding to last-seen final state
  - restart DFA to scan from the corresponding position

# Example



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

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

## Ambiguity resolution

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

## Overview

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

# Scanning with NFA vs. DFA

| **Automaton** | **SPACE** | **TIME** |
|---|---|---|
| NFA | O(\|R\|) | O(\|R\|*\|w\|) |
| DFA | O($2^{\|R\|}$) | O(\|w\|) |

- input word w
- regular expressions R
- worst-case input: $(a|b)^*a\underbrace{(a|b)(a|b)\ldots(a|b)}_{\text{n times}}$

## Efficient implementation

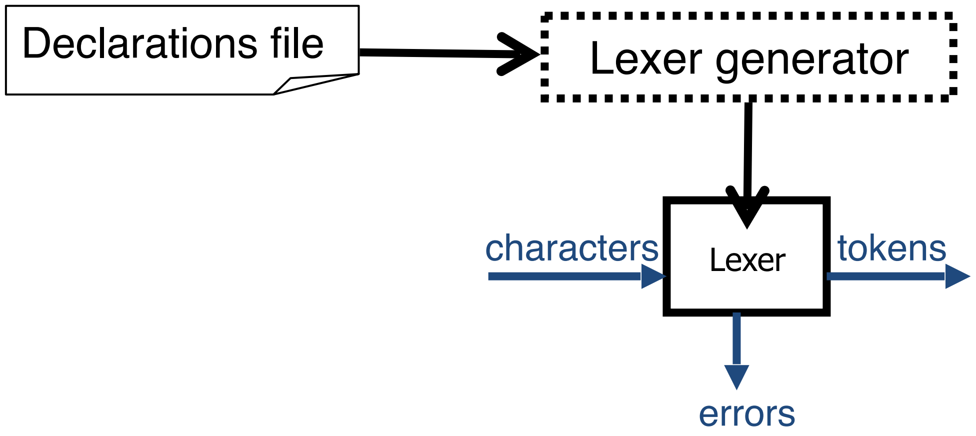
- Minimize DFA
- Efficient representation of states and transitions
- Using switch and gotos
- Input buffering
- …

# Summary of lexer construction

- Describe tokens as regular expressions
- Decide which attributes to save for each token
- Construct a non-deterministic finite-state automaton (NFA) from regular expressions and label final states with corresponding tokens
- Determinize the NFA into a deterministic finite-state automaton (DFA)
- DFA can be directly used to identify tokens
- Lexer simulates the run of the DFA on any input

# Good News

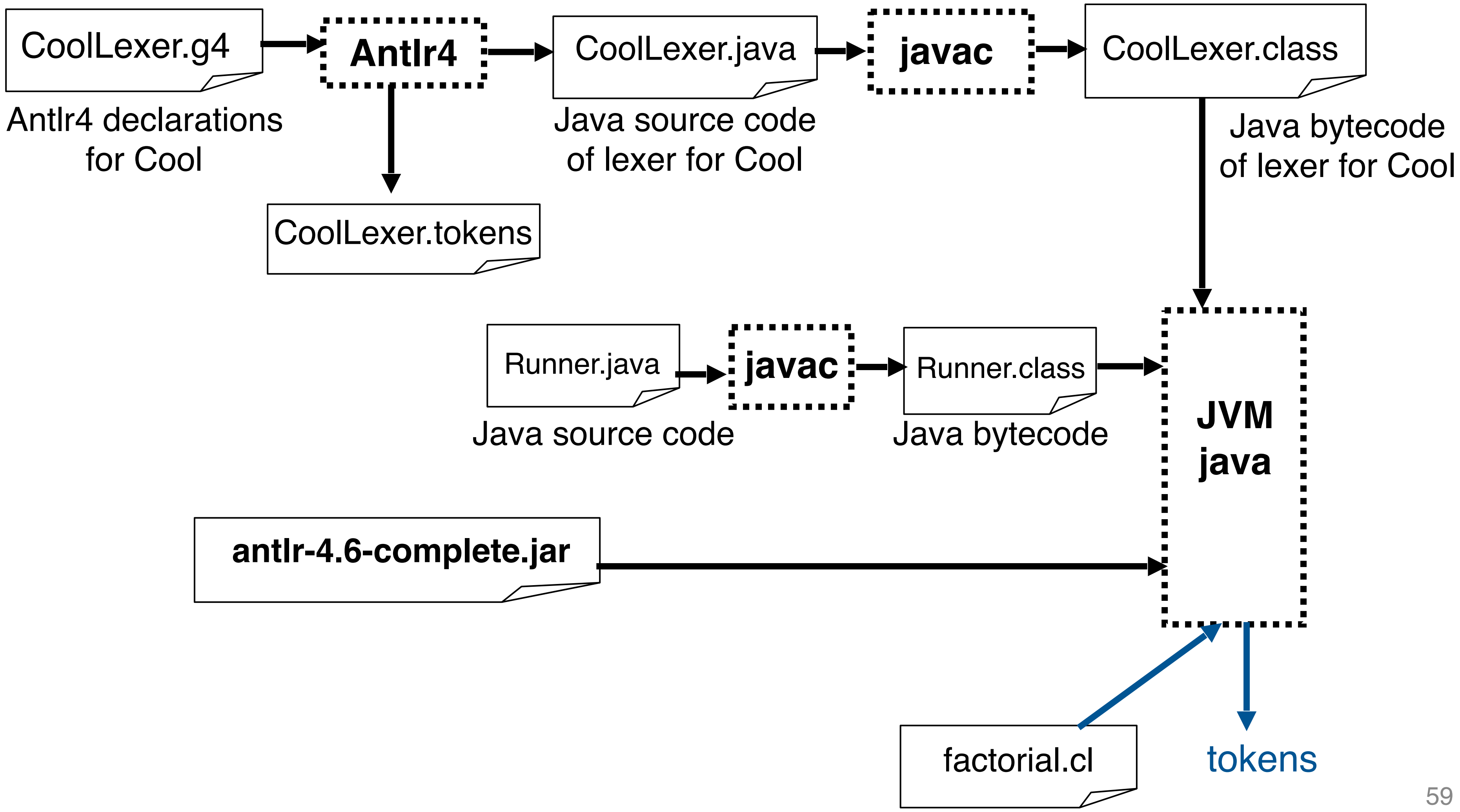
- Construction is done automatically by common tools
- **Lexer generator** automatically creates **lexer** from **declarations file**
  - lexer generator builds DFA table
  - lexer simulates (runs) the DFA on a given input
- Short declarations file is easily checked, modified, maintained
- We will use **Antlr** lexer generator



# Declarations file in Antlr4

```
lexer grammar ExampleLexer;

INT              : DIGIT+ ;
fragment
DIGIT            : [0-9] ;

ID               :  [a-z] IDENTIFIER* ;
fragment
LETTER            :  [a-zA-Z];

COMMENT    : '/*' .*? '*/'   -> skip;

WHITESPACE : (' ' | '\n' | '\r' | '\t' | '\u000B')+ -> skip;

ERROR              : . ;
```

# Antlr: build and run a lexer



# Runner

```
import org.antlr.v4.runtime.*;
import org.antlr.v4.runtime.tree.*;
public class Runner {
  public static void main(String[] args) throws Exception {
    ANTLRInputStream input = new ANTLRInputStream(System.in);
    CoolLexer lexer = new CoolLexer(input);
    CommonTokenStream tokens = new CommonTokenStream(lexer);
    for (Token t : tokens.getTokens()) {
          System.out.println(t.toString());
    }
  }
}
```

## Terminology

- lexical analysis
- lexical analyzer
- lexer
- scanner

- **Not** the same as lexer generator

# Lexical analysis can be hard

- C++
  - nested templates: `list<vector<int>> x`
  - streams: `cin>> x`
- Fortran, Algol68
  - whitespace not significant: `Char lie` and `Charlie`
- PL/I
  - keywords not reserved
  - `if else then else=then;  else then=else;`

## Limitations of regular languages

- Not all languages are regular
- Regular languages cannot count
- DFAs cannot recognize matched parenthesis
  - $L = \{ \ (^k \ )^k \ \mid k \geq 0 \ \}$

# Summary: lexical analysis

- Turns character stream into token stream
- Tokens defined using regular expressions
- Construction for identifying tokens:
  Regular expressions -> NFA -> DFA
- Exponential worst case, not a problem in practice
- Automated construction of lexical analyzer
- Antlr4 generates more powerful lexers
  - predicated context-free grammars (not just regular expressions)
  - can recognize context-free tokens such as nested comments
  - can handle context-sensitive lexing such as merging C and SQL

## How to precisely define programming language?

- Layered structure of language definition
- Start with the set of letters in language
- **Lexical structure** identifies “words” in language: each word is a sequence of letters
- **Syntactic structure** identifies “sentences” in language: each sentence is a sequence of words
- **Semantics** defines meaning of program: specifies what result should be for each input

## Regular languages

- Regular expressions:
  L(R) maps regular expression R to a set of words over Σ that match R
- Finite state automata:
  L(A) maps a finite state automaton A to the set of words over Σ accepted by A
- For every R, there exists A such that L(R)=L(A) and vice versa

# Context-free languages

- Context free grammars:
  L(G) maps grammar G to a set of words over Σ derived by G
- Pushdown automata:
  L(P) maps a pushdown automaton P to the set of words over Σ accepted by P
- For every G, there exists P such that L(G)=L(P) and vice versa

# Approaches to formal languages

- **Generative**: regular expression or grammar
  - generate all strings in language
  - declarative, good for humans, specification
- **Recognition**: automaton
  - recognize if a specific string is in the language or not
  - operative, good for automation, implementation
- Theorems about **equivalence**
- Automatic **conversion** between representations

# Specifying formal languages

- Huge success in computer science
  - beautiful theoretical results
  - practical techniques and applications
- Standard practice
  - use regular expressions or grammars to **define** a language
  - translate automatically into corresponding automata for **implementation**