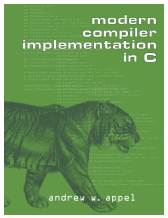
Compilation

0368-3133

Lecture 1b:

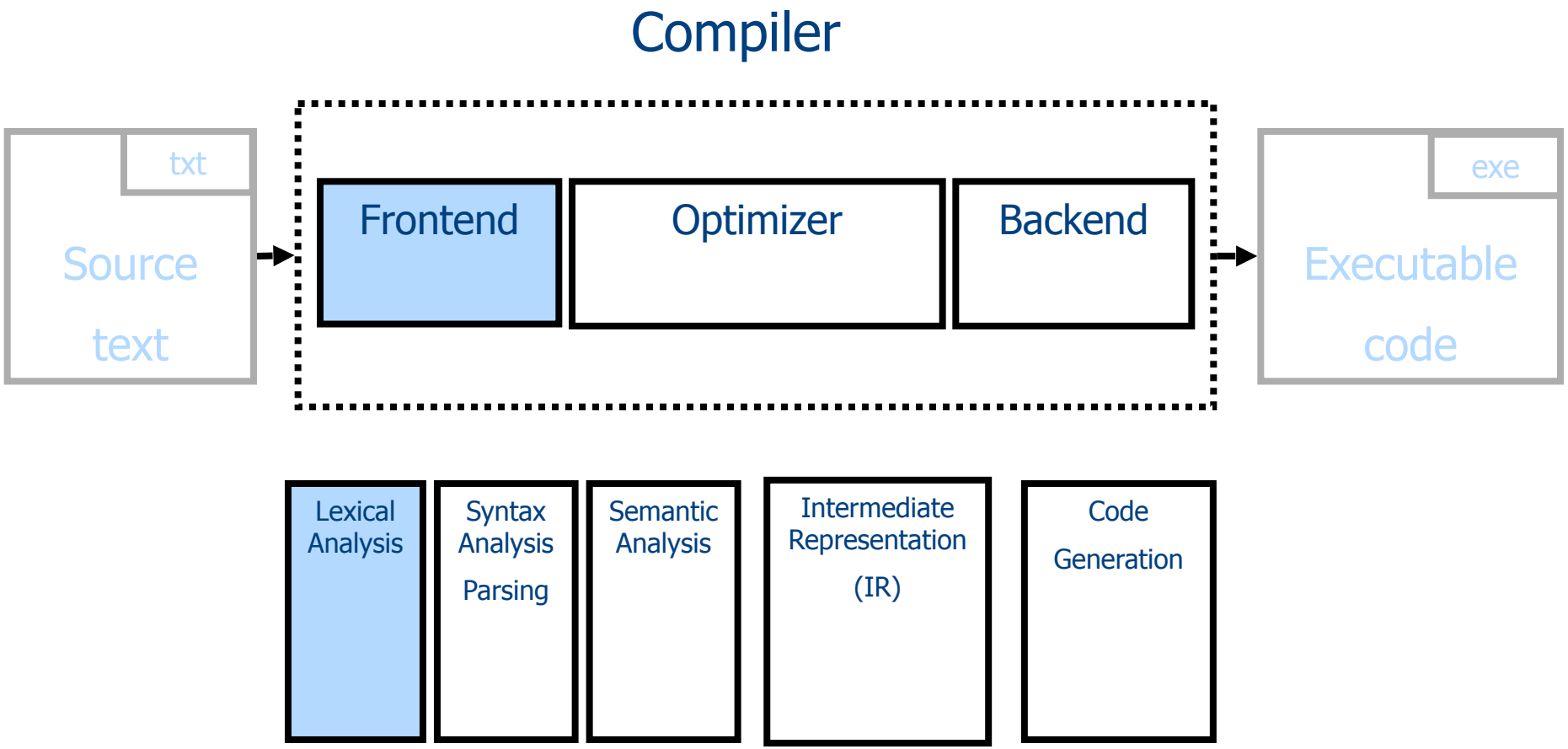
Lexical Analysis



Modern Compiler Implementation in C: Chapter 2

Slides credit: Shachar Itzhaky, Eran Yahav, Mooly Sagiv

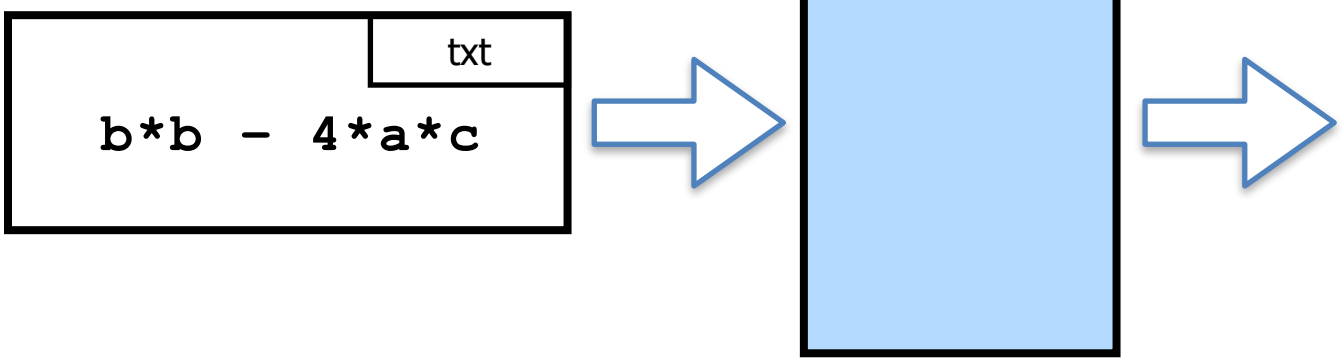
Conceptual Structure of a Compiler



What does Lexical Analysis do?

Stream of characters

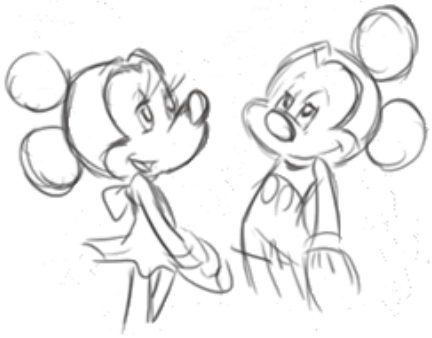
(program source code)



Stream of tokens

(“meaningful words”)

From Characters to Tokens

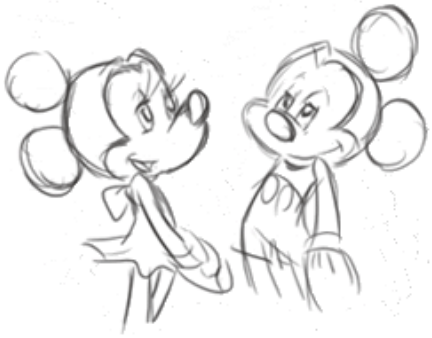


- What is a token?
  - Roughly – a “word” in the source language

Identifiers | Values | Keywords

  - Really – anything that should appear in the input to syntax analysis as a single unit
- Technically
  - Usually a pair of ⟨kind, value⟩

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
- Technically (in "real" compilers)
  - Usually a pair of ⟨kind, value, line, column⟩

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	`-`

Example

- Language: fully parenthesized numerical expressions
  - e.g., ((23 + 7)* 19)

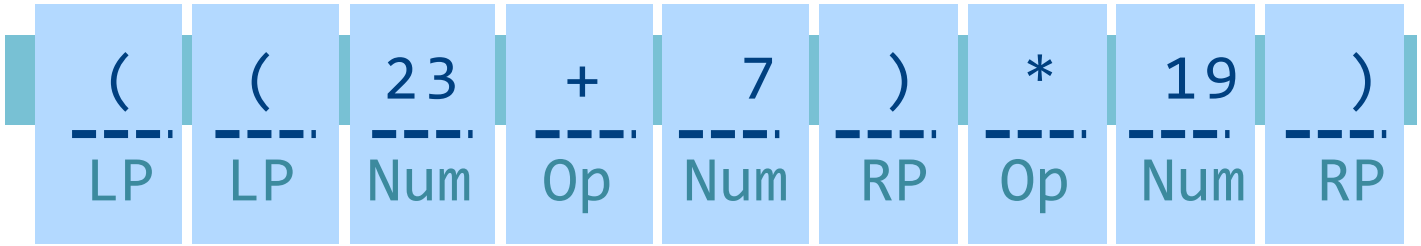
- What are the meaningful “words”?

Example

- Language: fully parenthesized numerical expressions
  - e.g., ((23 + 7)* 19)

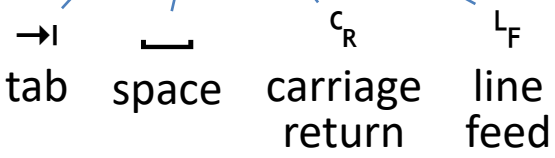
- What are the meaningful "words"?
  - Numbers (e.g., 23, 7, 19)
  - Left (open) parantesis (i.e., ( )
  - Right (close) parenthesis (i.e.))
  - Operators (e.g., + , *)

Output



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`



*In some languages, whitespaces do more than separate words

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...)

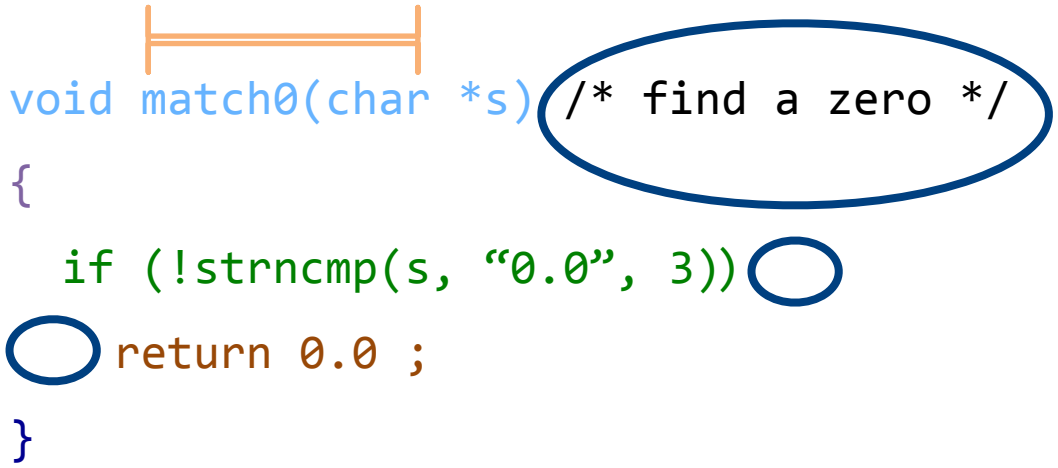
- **Pattern** – a rule specifying a set of strings.
  Example: "*an identifier is a string that starts with a letter, followed by letters and digits*"

- **Token** – a pair of ⟨pattern, attributes (of lexeme)⟩

"kind"
"class"

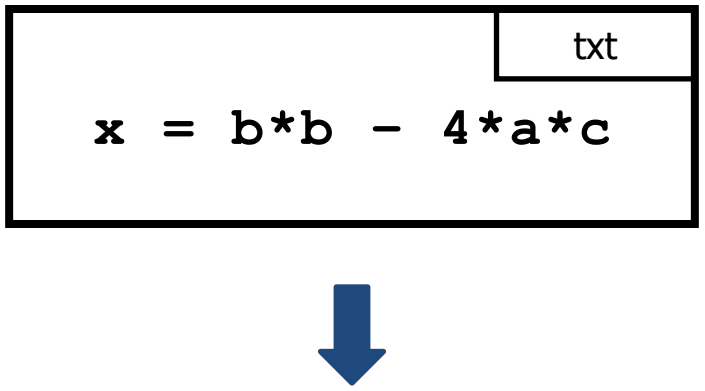
"value"
"representation"

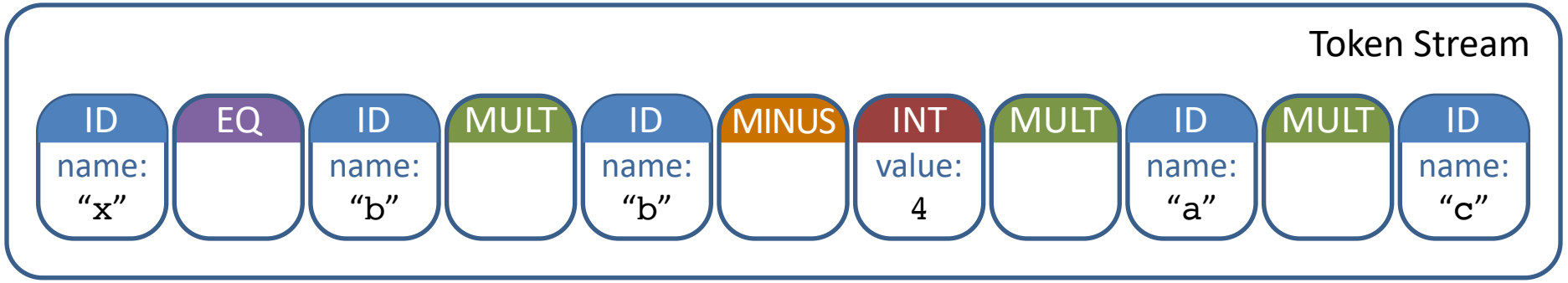
A More Realistic Example




```
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
```

Scanner*: Characters to Patterns





*aka. lexical analyzer, lexer

How can we define

An identifier is a sequence of letters and digits;
the first character must be a letter.
The underscore _ counts as a letter.
Upper- and lowercase letters are different.
If the input stream has been parsed into tokens up to a given character, the next token is taken to include the longest string of characters that could possibly constitute a token.
Blanks, tabs, newlines, and comments are ignored except as they serve to separate tokens.
Some white space is required to separate otherwise adjacent identifiers, keywords, and constants.

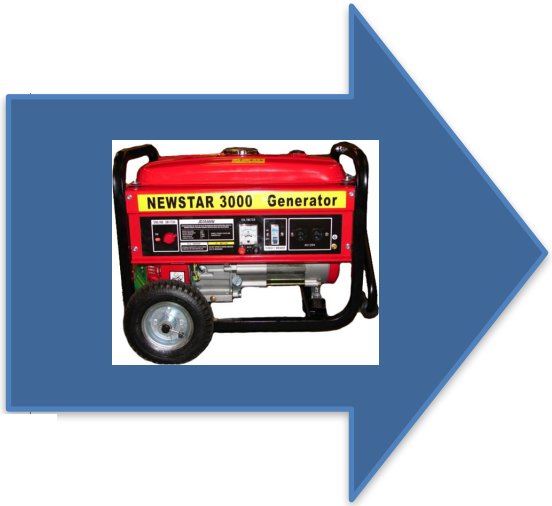
- Keywords — easy!
  - `if`, `then`, `else`, `for`, `while`, …
- Identifiers? `x0`, `b2_b`, `heIsNoOne`, …
- Numerical Values? `0`, `1`, `2`, `3.4`, `5.67`, …
- Strings? `"there are infinitely many"`, …
- Characterize **unbounded** sets of values using a **bounded** description?

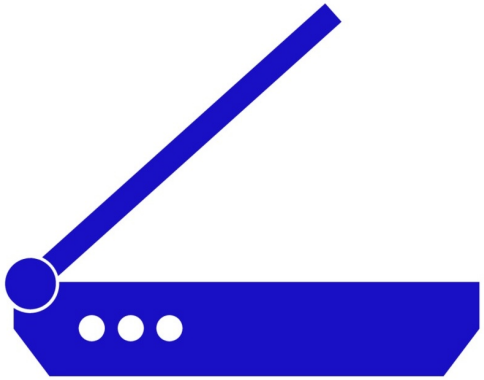
Regular languages & FSMs

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

- Regular languages defined equivalently by
  - Regular expressions (human-readable)
  - Finite-state machines (machine-executable)

Goal: Regular Expressions ⇒ Scanner





* Picture from vecteesy.com

How: Regular Expressions ⇒ Scanner

Describe tokens as regular expressions

Create an automaton

(compiler construction time)

Use autoamton to scan the input

(compilation time)

(Computer-Readable) Regular Expressions

* Picture from vecteesy.com

Regular Expressions over Σ

- 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

Examples

- `(a|b)*` =
- `(0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9)+` =
- `ab*|cd?` =
- `[<>]=?|==|<>` =

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 in groups to denote a range
  - letter = [a-z] | [A-Z] ⇢ [a-zA-Z]
  - digit = [0-9]

Shorthands

- Use names for expressions
  - letter = `a` | `b` | … | `z` | `A` | `B` | … | `Z`
  - letter_ = letter | _
  - digit = 0 | 1 | 2 | … | 9
  - id = letter_ (letter_ | digit)*
- Use caret to denote negation
  - notabc = [^abc]
  - notdigit = [^0-9]

Escape characters

- What is the expression for one or more “+” symbols?
  - `(+)+` won’t work
  - `(\+)+` will
- backslash `\` before an operator turns it into a standard character
  - `\*`, `\?`, `\+`, `\[`, …
- backslash `\` before r / n turns it into carriage return (\r) / line feed (\n)

More Examples

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

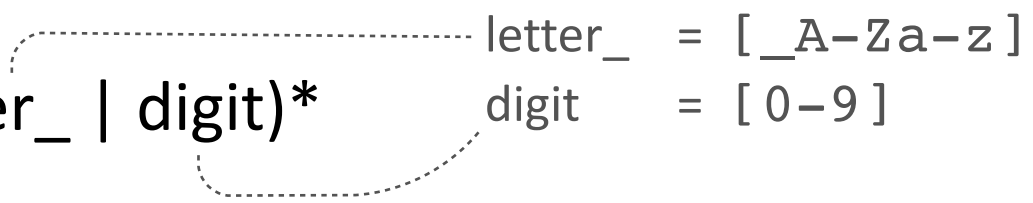
- digit = `[0-9]`
- digits = digit+

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

  E.g., ??

Ambiguity

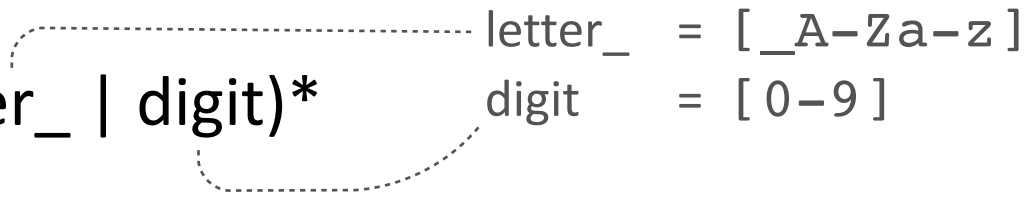
- if = `if`
- id = letter_ (letter_ | digit)*



What's the problem?

Ambiguity

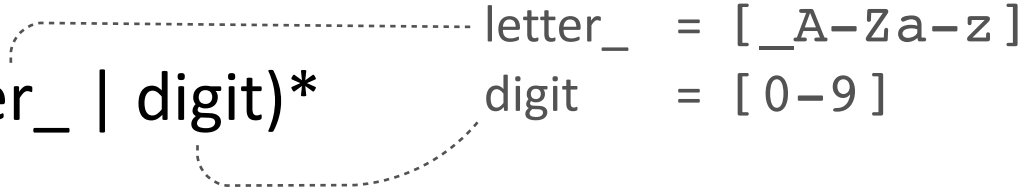
- if = `if`
- id = letter_ (letter_ | digit)*



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

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*)

Building blocks of a lexical analyzer

* Picture from vecteesy.com

Goal: a lexical analyzer

- Given a list of token definitions, write a program such that
  - Conceptually,
    - Input: String of characters to be analyzed
    - Output: List of tokens
  - Practically, implement `get_next_token()`

- How?

Building a Scanner – Take I

```
Token get_next_token()
{
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`: …
 }
```

There must be a better way!

A better way

- Define tokens using regular expressions
- Convert regular expressions to a finite-state automaton
- Use finite-state automaton for detection

Reg-exp vs. automata

- Regular expressions are declarative
  - Good for humans
  - Not “executable”

- Automata are operative
  - Define an *algorithm* for deciding whether a given word is in a regular language
  - Not a natural notation for humans

Construction

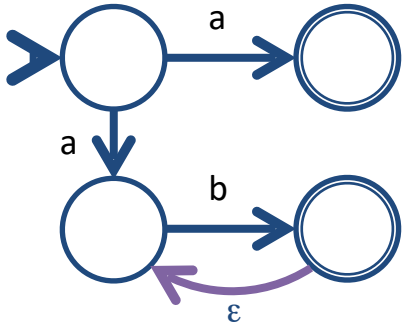
txt

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

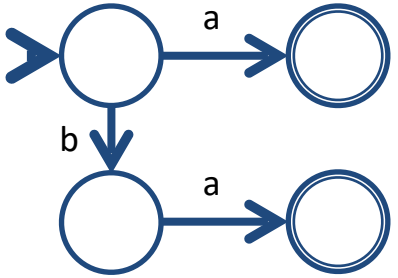
Token Definitions

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

Non-deterministic
"Finite Automaton"



Deterministic
"Finite Automaton"



Token Stream

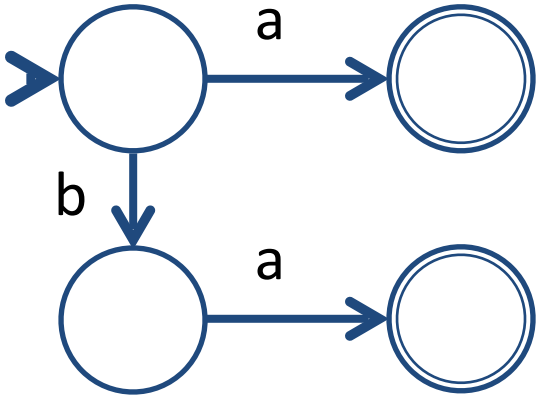
⟨ID,"**x**"⟩ ⟨EQ⟩ ⟨ID,"**b**"⟩ ⟨MULT⟩ ⟨ID,"**b**"⟩ ⟨MINUS⟩
⟨INT,**4**⟩ ⟨MULT⟩ ⟨ID,"**a**"⟩ ⟨MULT⟩ ⟨ID,"**c**"⟩

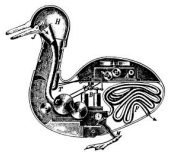
* Actually, to handle ambiguity, a slight variant of finite-state automata is used

Reminder: Finite-State Automata

Deterministic automaton

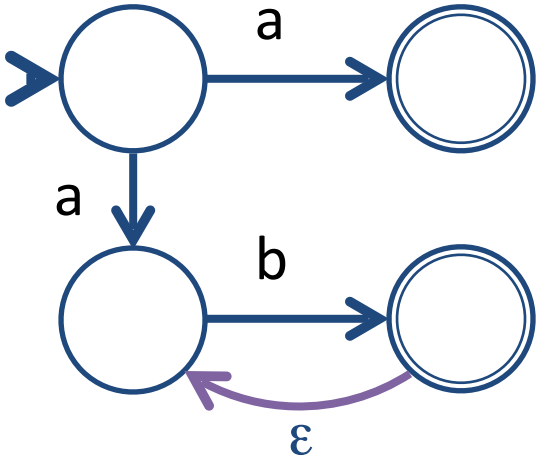
- $M = (\Sigma, Q, \delta, q_0, F)$
  - Σ – alphabet
  - Q – finite set of states
  - $q_0 \in Q$ – initial state
  - $F \subseteq Q$ – final/accepting states
  - $\delta : Q \times \Sigma \rightarrow Q$ – transition function



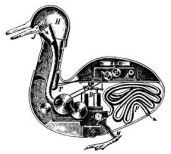


Non-Deterministic automaton

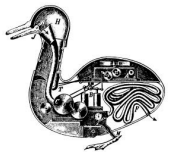
- $M = (\Sigma, Q, \delta, q_0, F)$
  - Σ - alphabet
  - Q – finite set of states
  - $q_0 \in Q$ – initial state
  - $F \subseteq Q$ – final/accepting states
  - $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \rightarrow 2^Q$ – transition function



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

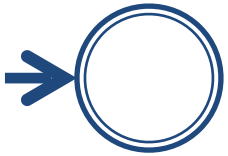


From Regular Expressions to NFA

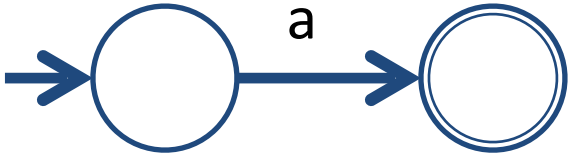


Basic constructs

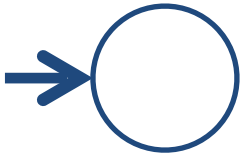
R = ε

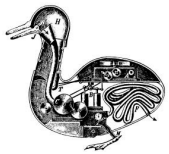


R = a



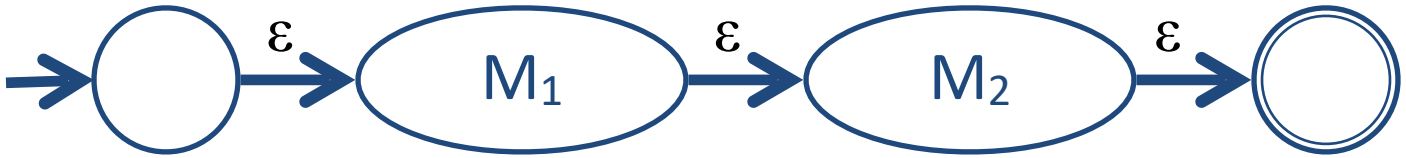
R = ∅



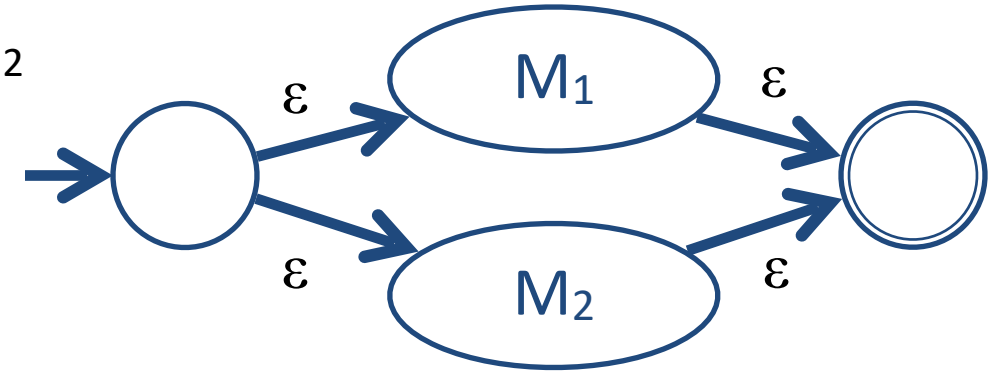


Composition

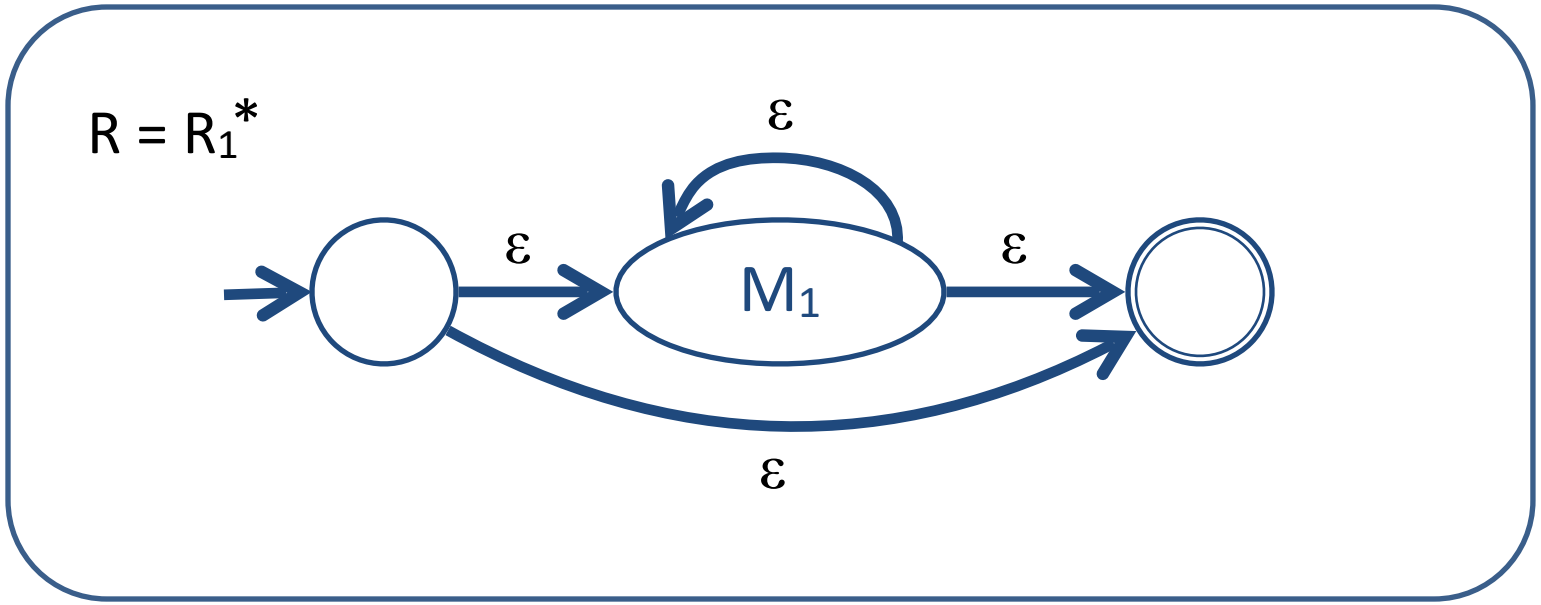
$R = R_1 R_2$

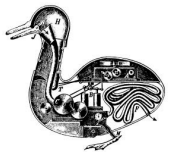


$R = R_1 \mid R_2$

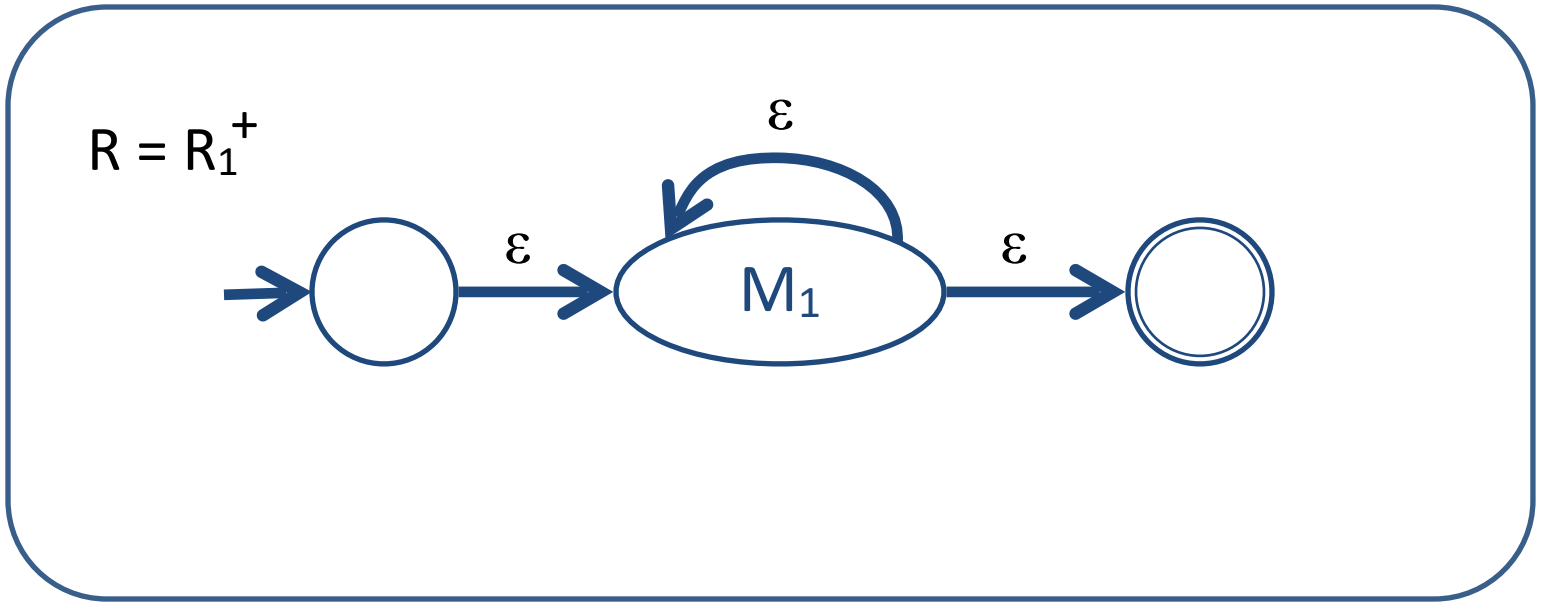


Repetition

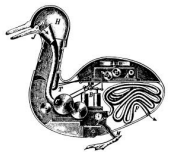




Repetition



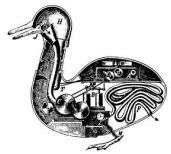
$R_1^+ = R_1R_1^*$



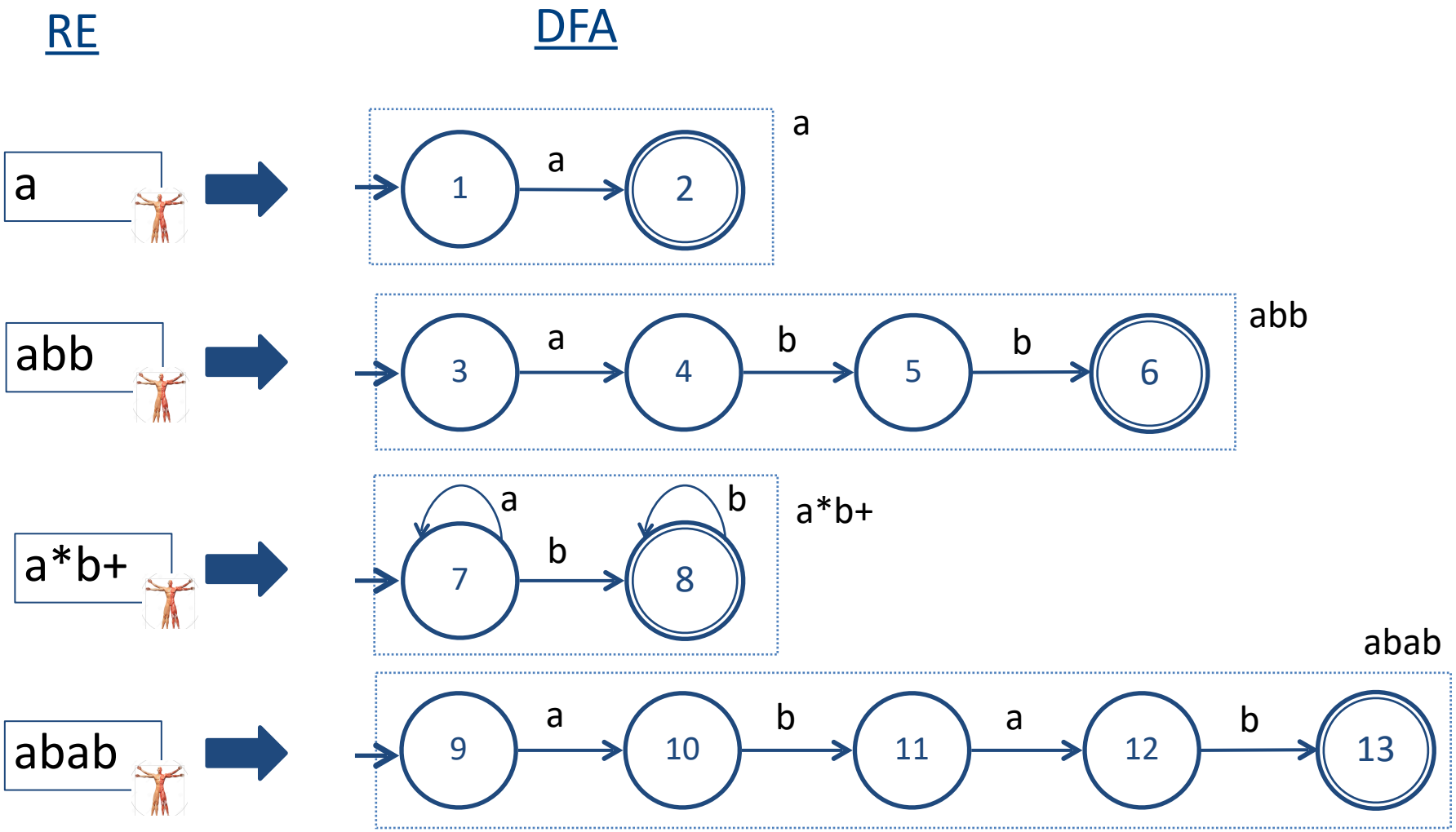
From NFA to DFA

- Theorem: *there is an algorithm that transforms an NFA+Є automaton into a DFA that accepts the same language*

- *Recall "Modelim"*



Example: Regular Expressions to Automata



What now?

We have one DFM per pattern ($RE_i \Rightarrow M_i$).

Naïve approach: try each automaton separately

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

What now?

We have one DFM per pattern ($RE_i \Rightarrow M_i$).

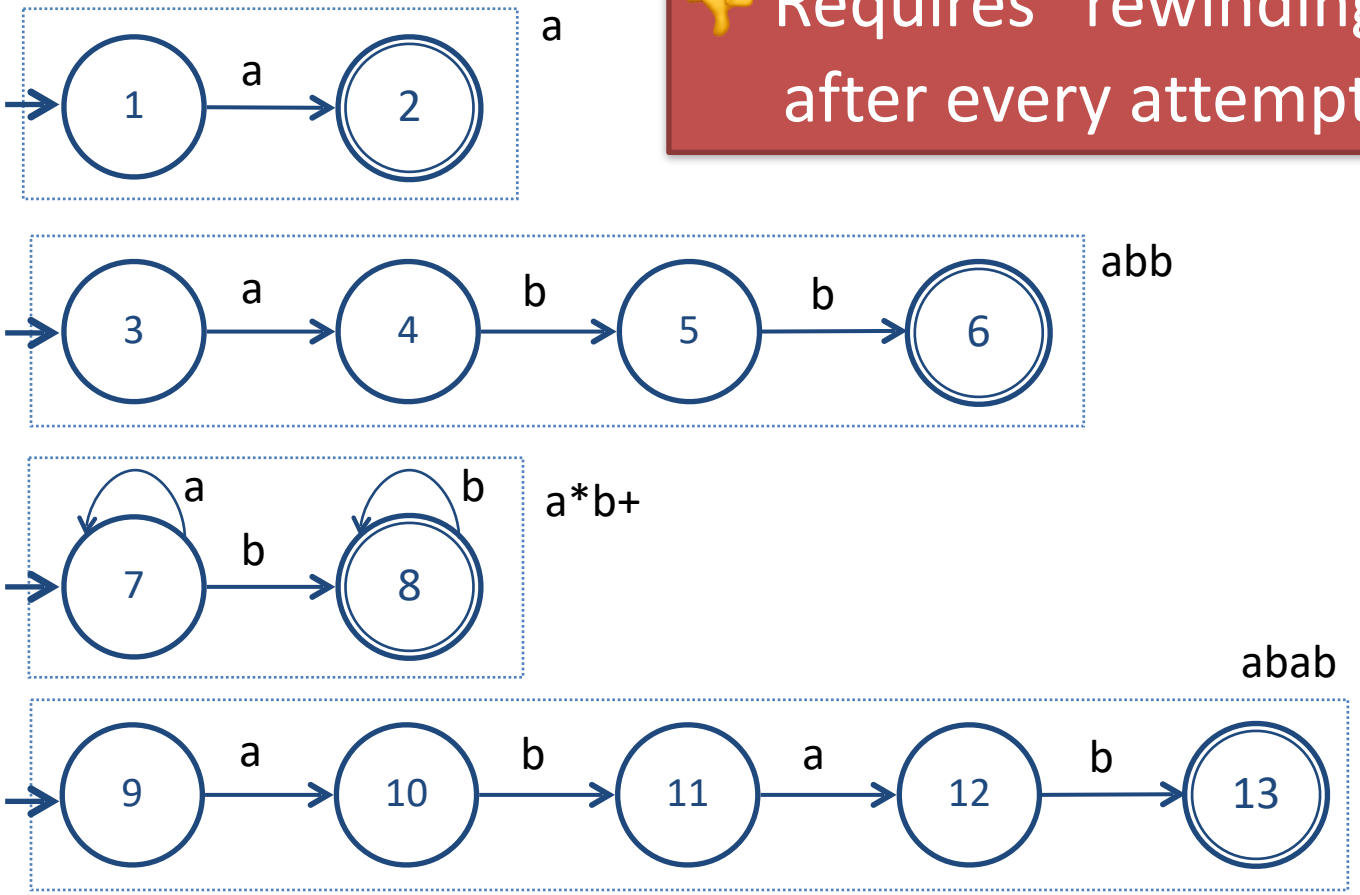
Naïve approach: try each automaton separately

- Given a sequence of tokens as *input**:
  - Try M_1(*input*)
  - Try M_2(*input*)
  - …
  - Try M_n(*input*)

- Run M_i on *input* from start state until **stuck**
  - Remember ***position***$_i$ after character read at the last final state and the accepted ***prefix***$_i$
- Set m = max{*poisition*$_1$, … , *position*$_n$}
  - if m=0 return ERROR
- Set j = min{ i | *position*$_i$ == m}
- Advance input to position$_j$
- return (*pattern*$_j$, *prefix*$_j$)

*Requires "rewinding" input after every attempt

Naïve Approach: Example Run



👎 Requires "rewinding" after every attempt

abaa: Recognizes

abba: Recognizes

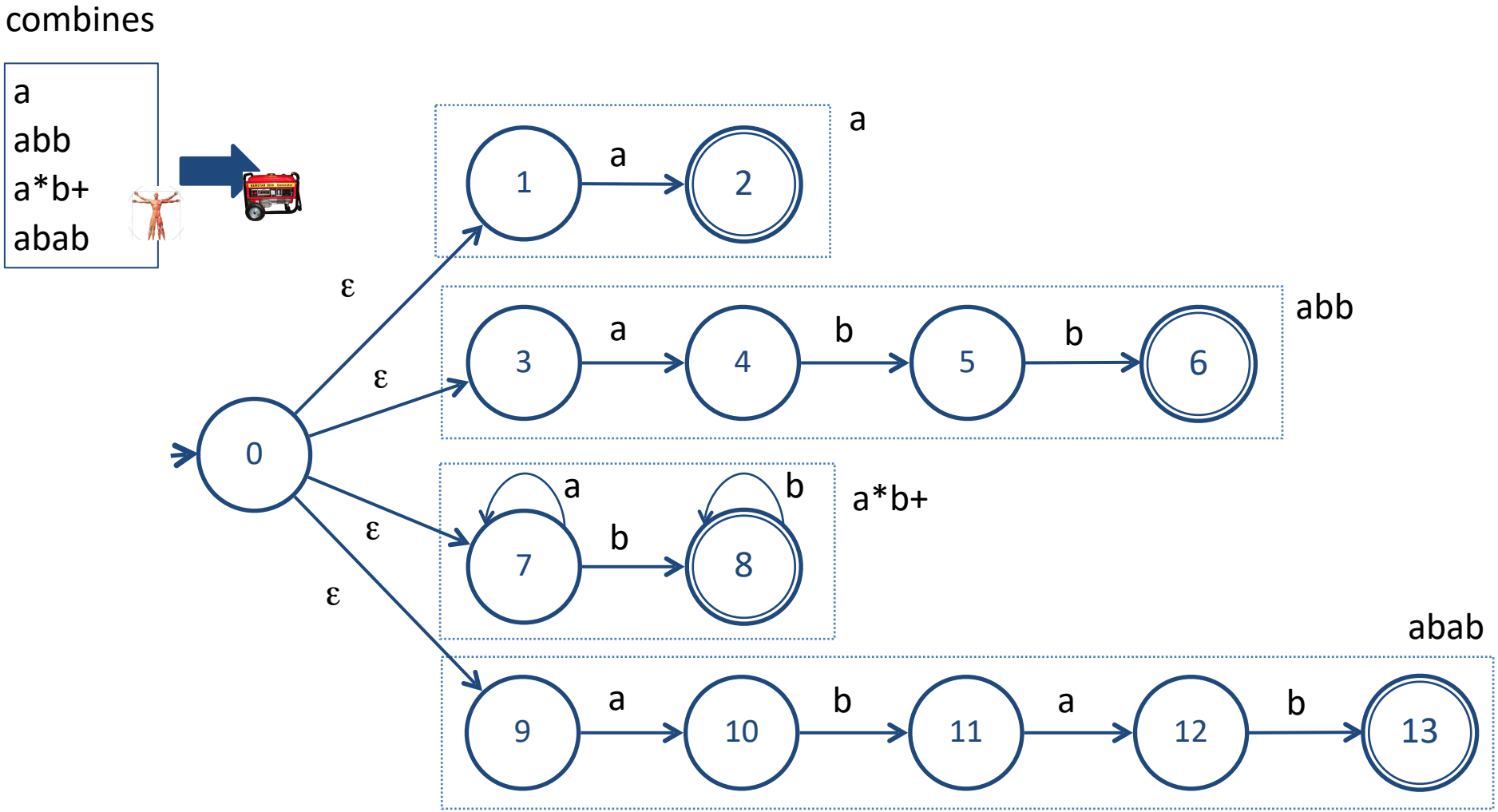
A Better Solution

Construct a single DFM for all the patterns

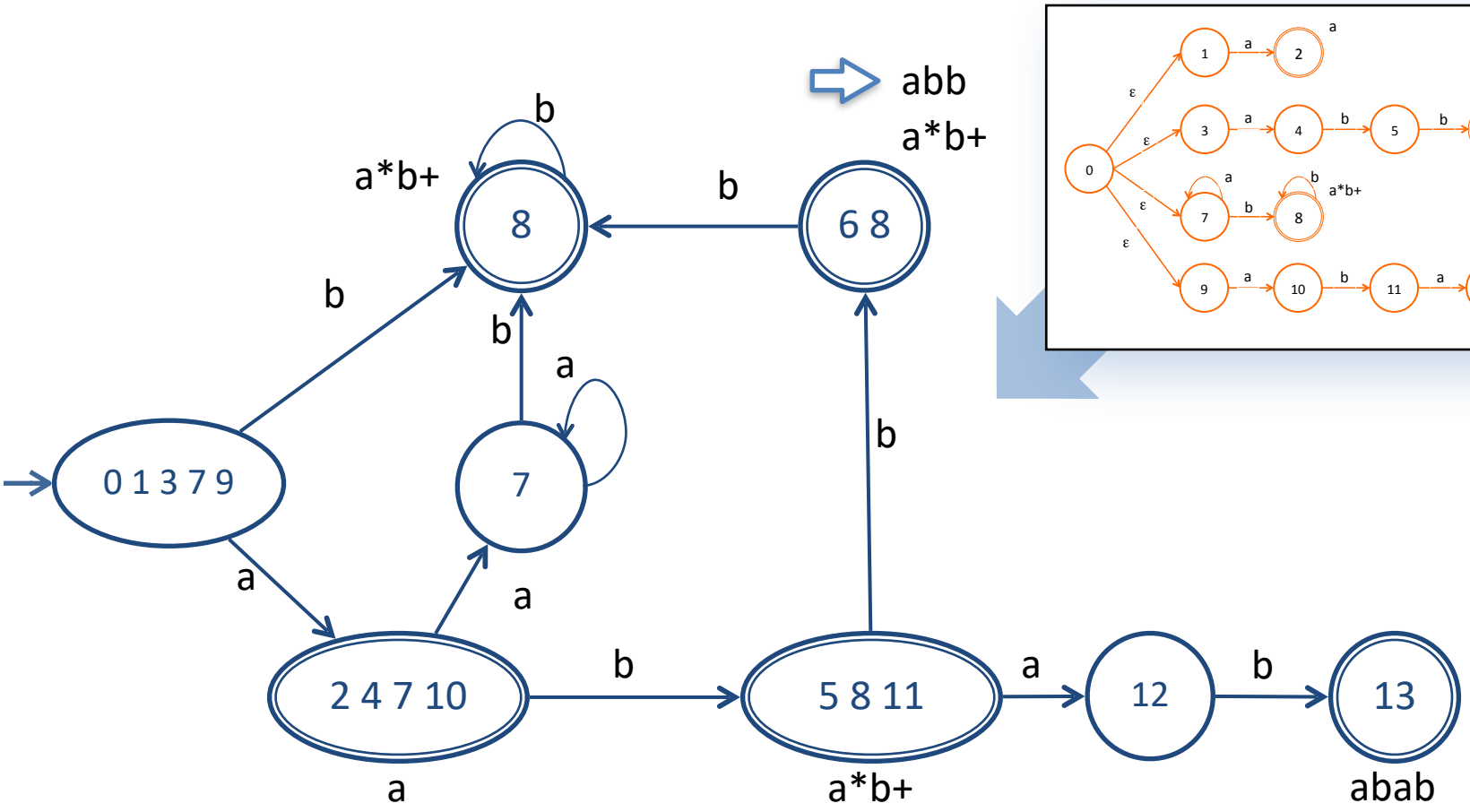
- Record the origin patterns of the final states

- Run the DFM on input from start state until stuck as before
  - Again, ambiguity resolution based on length and order of rules
  - Again, requires rewinding, but only once!

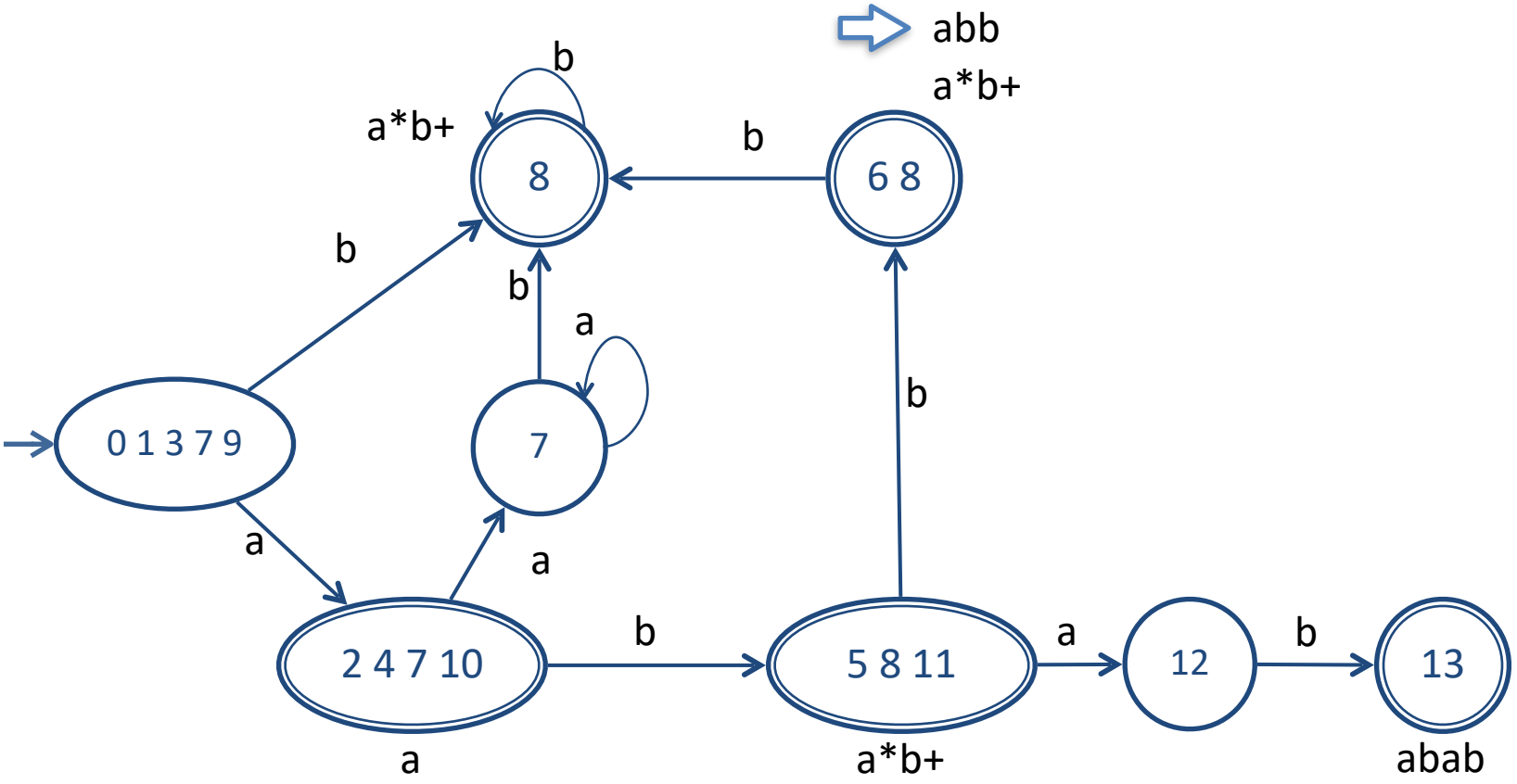
Example: Combine Automata



Example: Corresponding DFA

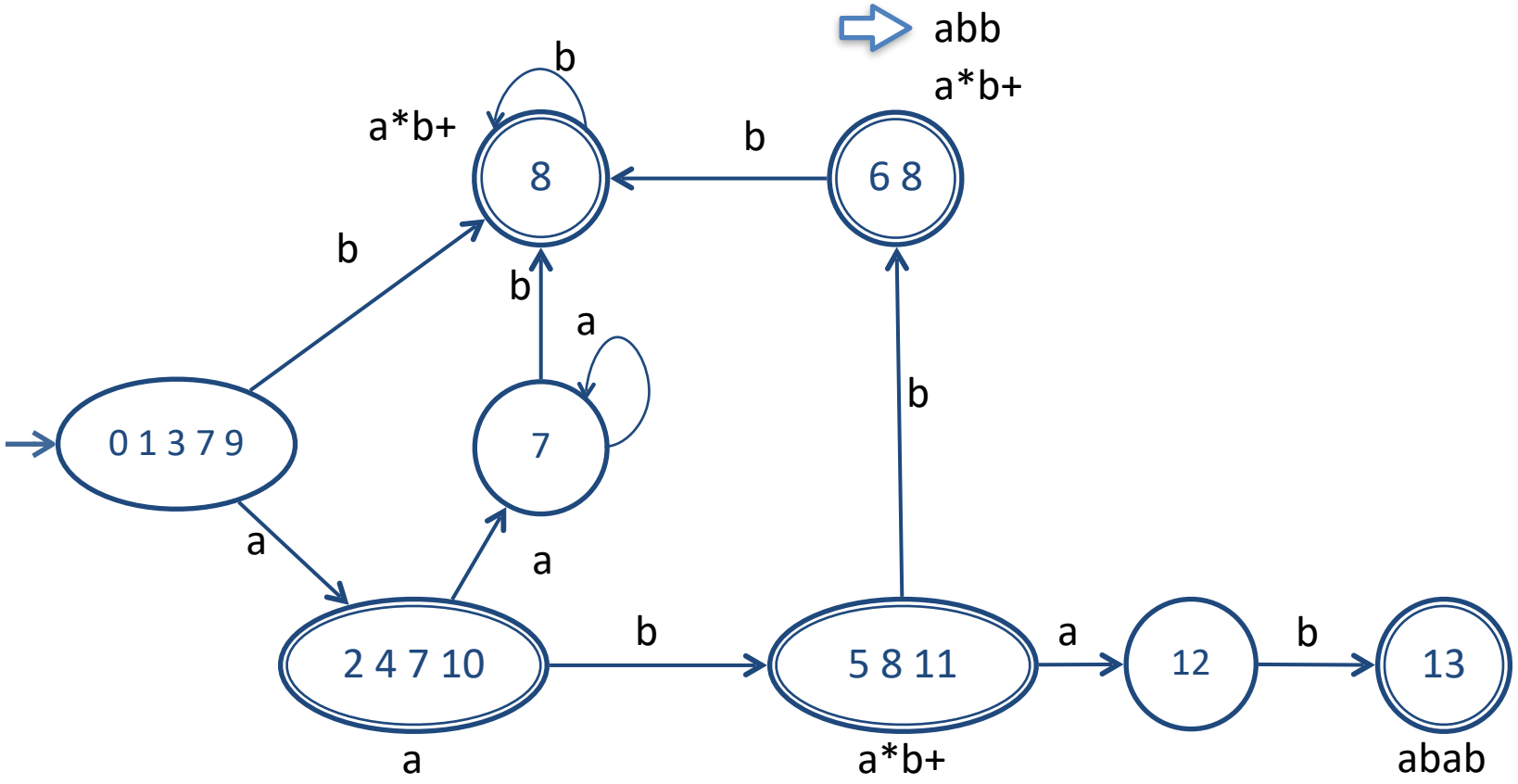


Example Runs



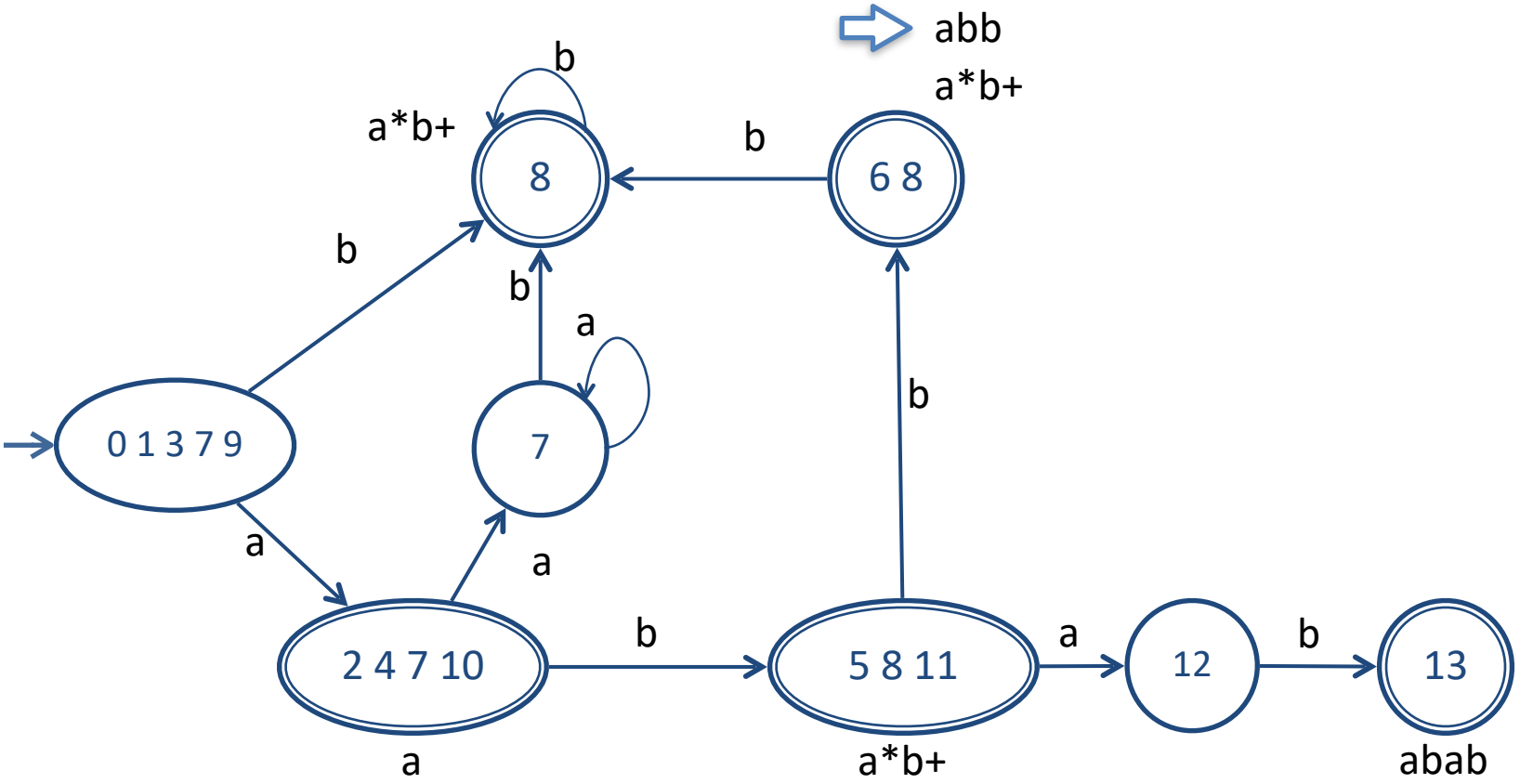
abaa: gets stuck after **aba** in state 12, backs up to state (5 8 11) pattern is a*b+, token is **ab**

Example Runs



abaa →

Example Runs



abaa → (a*b+, ab) (a, a) (a, a)

abaa →

Summary of Construction

✓ Describe tokens as regular expressions

✓ Regular expressions "compiled" into a single DFA

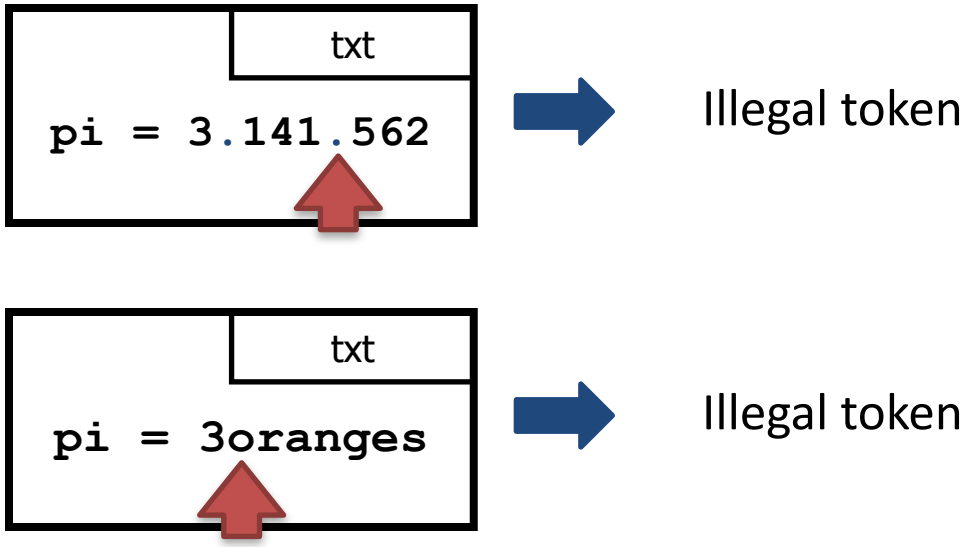
(compiler construction time)

✓ Lexical analyzer simulates the run of an automaton with the given transition table on any input string

- Use the following ambiguity resolution algorithm:
  - Prefer ***longest*** accepting word
  - Respect order of definitions

(compilation time)

Errors in Lexical Analysis



Error Handling

- Many errors cannot be identified at this stage
- Example: 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

```
fi (a==f(x))
```

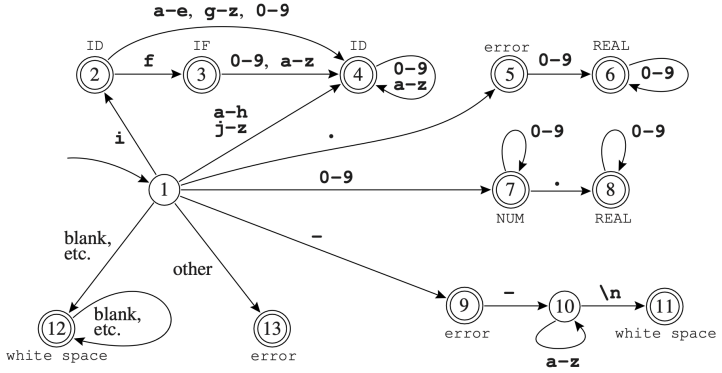
- Sometimes the lexeme does not match any pattern
  - Easiest fix: skip characters until the beginning of a legitimate lexeme
  - Alternatives: eliminate/add/replace one letter, replace order of two adjacent letters, etc.

- Pro: allow the compilation to continue
- Con: errors that spread all over

- Record source location of token (line/column) for error reporting

Implementing a Scanner

Pseudocode: Encoding DFA



"stuck" state

initial state

```
int edges[14][ * …, 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  */      ,  …, 4, 4, 4, 4,  ..., 0,  4, 3, 4, 4, 4, 4, …, 0, 0},
/* state 3  */      , …,  4, 4, 4, 4,  …, 0,  4, 4, 4, 4, 4, 4, , 0, 0},
/* state 4  */      , …,  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}
};

bool isFinal[14] = {false, false, true, … , true, … };

Kind pattern[14] = { Error, null, ID, IF, … };  // Final state →  Token Kind
```

Pseudocode: Scanning routine

```
input = " … "; startPos = 0;

Token get_next_token() {
    lastFinal = 0;      // Stuck state
    currentState = 1;  // Initial state
    currentPos = posAtLastFinal = startPos;

    while (currentState != 0)  {
       currentChar = input[currentPos]
       nextState = edges[currentState][currentChar];
       if (isFinal[nextState]) {
              lastFinal = nextState ;
              posAtLastFinal = currentPos;
       }
       currentState = nextState;
       currentPos++;
    }
    lexeme = input[startPos .. posAtLastFinal];
    startPos = posAtLastFinal + 1;
    return (pattern[LastFinal], lexeme);
}
```

Pseudocode: Scanning routine

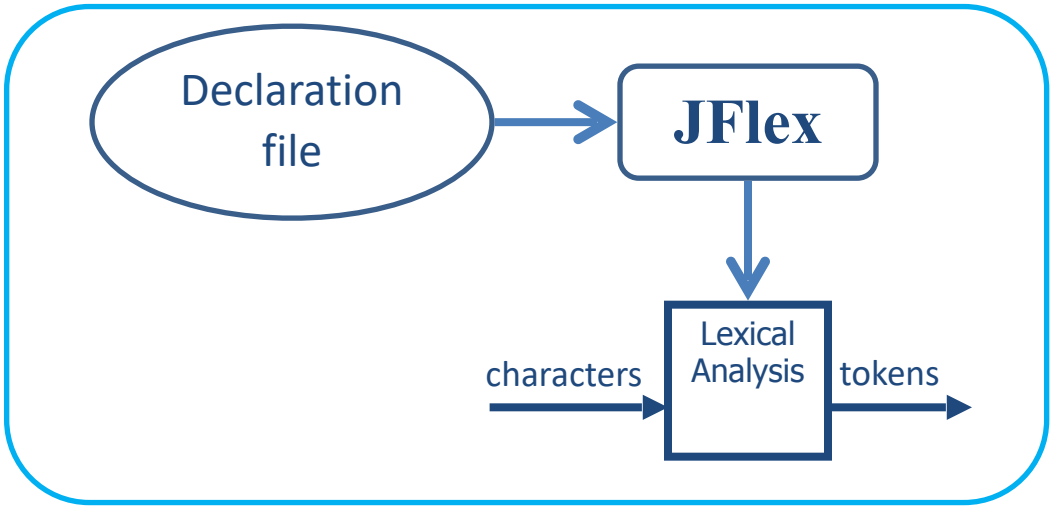
```
input = " … "; startPos = 0;

Token get_next_token() {
    lastFinal = 0;      // Stuck state
    currentState = 1;   // Initial state
    currentPos = posAtLastFinal = startPos;

    while (currentState != 0)  {
       currentChar = input[currentPos]
       nextState = edges[currentState][currentChar];
       if (isFinal[nextState]) {
              lastFinal = nextState ;
              posAtLastFinal = currentPos;
       }
       currentState = nextState;
       currentPos++;
    }
    lexeme = input[startPos .. posAtLastFinal];
    startPos = posAtLastFinal + 1;
    return (pattern[LastFinal], attr[lastFinal](lexeme), initPos);
}
```

NUM: "18" ➔ 0x12

ID: "a18" ➔ "a18"

Good News: Compiler Compiler

- Construction is done **automatically** by common tools
- **JFLex/Flex/Lex** are your friends
  - Automatically generate a lexical analyzer from a declaration file
- Advantages: short declaration file, easily checked, easily modified and maintained



Intuitively:

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

JFLex Spec File

Java code: Copied verbatim to generated Java file
(before the lexer class)

%%

%{

Java code: Copied verbatim into the lexer class

%}

%line, %column, %jcup

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

JFlex directives: options, shorthand macros, state names

%%

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

{return new Symbol(Symbol.IF); }
{return new Symbol(Symbol.ID, yytext()); }

Lexical analysis rules: Optional state, pattern { action }

whitespace counter

Creating a Scanner using JFlex

```
import java_cup.runtime.*;

%%

%cup
%{ // Copied into the generated Lexer class source code
   private int wsCounter = 0;
   public int getWSCounter() { return wsCounter; }
%}

LineTerminator  = \r | \n | \r\n
WhiteSpace      = {LineTerminator} | [\s\t]

%%

{WhiteSpace} { wsCounter++; }
.            { }
```

tab

space

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

rules ...

```
.                { error("Illegal Token"); }
```

to include any character + newlines:

.|[\r\n]

or

[^]


conclusion

Constructing A Scanner

get_next_toekn()

+

Define Tokens using regular expression

Define attribute extracting functions

DFA

Scanner

From Regular Expressions to DFA

(compiler construction time)

- 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, determinize
  - Associate every accepting state with a single pattern, respecting the order of definitions

Using DFM to Recognize Tokens
(compilation time)

- Step 1: Read input (program) character by character while simulating the run of the DFM until getting stuck
  - Record last accepting state + corresponding prefix
- Step 2: Rewind input to the longest accepted prefix
- Step 3: Return the token associated with the last accepting state

DFA Construction

- Turning an NFA into a DFA is expensive, but
  - Exponential in the worst-case
  - In practice, it works fine

- The construction is done **once per language**
  - At Compiler construction time
  - Not at compilation time

- Minimize the automaton
  - Separate accepting states by accepted token kinds

Missing

- Efficient Scanners
  - Transition table compression
  - Input buffering
  - Using `switch` and `goto`s instead of tables
- Symbol Tables
- Nested Comments
- Handling Macros
- Creating a lexical analysis by hand

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 **JFlex**

Exercise (From Exam)

שאלה 1 (20 נקודות)

נתונה רשימת מאורעות. לכל אחד מהמאורעות הסבירו בקצרה (1-3 שורות) האם הוא מתרחש במהלך בניית הקומפיילר / הקומפילציה / assembly / linking / loading / ריצת התוכנית. אם בחרתם בזמן קומפילציה, הסבירו מהו השלב המסוים בקומפיילר בו מתרחש המאורע ומהם מבני הנתונים הרלבנטיים. אם מאורע יכול להתרחש במספר שלבים, ציינו את כולם.

Possible answers (at this point):

- Compiler construction time
- Lexical Analysis
- Later stages

Exercise (From Exam)

שאלה 1 (20 נקודות)

נתונה רשימת מאורעות. לכל אחד מהמאורעות הסבירו בקצרה (1-3 שורות) האם הוא מתרחש במהלך בניית הקומפיילר / הקומפילציה / assembly / linking / loading / ריצת התוכנית. אם בחרתם בזמן קומפילציה, הסבירו מהו השלב המסוים בקומפיילר בו מתרחש המאורע ומהם מבני הנתונים הרלבנטיים. אם מאורע יכול להתרחש במספר שלבים, ציינו את כולם.

א. (4 נקודות) נמצא כי נקראת מתודה המוגדרת במחלקה D.

ב. (4 נקודות) נמצא כי אין ריצה בה המשתנה x מכיל ערכים אי זוגיים.

ג. (4 נקודות) נמצא כי אין אף ריצה בה יש לכתוב את הערך של המשתנה הלוקאלי x במחסנית.

ד. (4 נקודות) הוחלט כי הארגומנט הראשון בפעולות כפל יהיה ברגיסטר %eax.

ה. (4 נקודות) נמצא כי יש identifier בשם myid בתוכנית.

Exercise (From Exam)

שאלה 1 (20 נקודות)

נתונה רשימת מאורעות. לכל אחד מהמאורעות הסבירו בקצרה (1-3 שורות) האם הוא מתרחש במהלך בניית הקומפיילר / הקומפילציה / assembly / linking / loading / ריצת התוכנית. אם בחרתם בזמן קומפילציה, הסבירו מהו השלב המסוים בקומפיילר בו מתרחש המאורע ומהם מבני הנתונים הרלבנטיים. אם מאורע יכול להתרחש במספר שלבים, ציינו את כולם.

א. (4 נקודות)נקבע כי השדה f שניגשים אליו במתודה של מחלקה D נורש מה superclass של D.

ב. (4 נקודות) נמצא כי אין ריצה של התוכנית בה אין גישה מחוץ לגבולות מערך.

ג. (4 נקודות) הוחלט כי יש לקבל את רצף התווים else כ token מסוג ELSE ולא identifier.

ד. (4 נקודות) הוחלט לקודד גישה לזיכרון מסוימת באמצעות הפקודה movl 4(%ebx,%ecx,8), %eax .

ה. (4 נקודות) נקבעה ה loading address של executable.

Exercise (From Exam)

שאלה 1 (20 נקודות)

נתונה רשימת מאורעות. לכל אחד מהמאורעות הסבירו בקצרה (1-3 שורות) האם הוא מתרחש במהלך בניית הקומפיילר / הקומפילציה / ריצת התוכנית. אם בחרתם בזמן קומפילציה, הסבירו מהו השלב המסוים בקומפיילר בו מתרחש המאורע ומהם מבני הנתונים הרלבנטיים. אם מאורע יכול להתרחש במספר שלבים, ציינו את כולם.

א. (4 נקודות) ערך ה callee-save register משוחזר מהמחסנית.

ב. (4 נקודות) יש בדקדוק רקורסיה שמאלית.

ג. (4 נקודות) אין רגיסטר פנוי עבור המשתנה הזמני t170 .

ד. (4 נקודות) השם r1$ אינו שם חוקי.

ה. (4 נקודות) יש פונקציה בשם r1.

Exercise (From Exam)

נתונה רשימת מאורעות. לכל אחד מהמאורעות הסבירו בקצרה (1-3 שורות)

1. האם הוא מתרחש במהלך בניית הקומפיילר/הקומפילציה / Linking / Loading /ריצת התוכנית.
2. אם בחרת בזמן קומפילציה ,הסבר מהו השלב המסוים בקומפיילר בו מתרחש המאורע ומהם מבני הנתונים הרלבנטיים.
3. אם מאורע יכול להתרחש במספר שלבים, ציינו את כולם.

א. (4 נקודות) נקבע כי הפרמטרים יוכנסו למחסנית בסדר הפוך מסדרם ברשימת הארגומנטים..
ב. (4 נקודות) נמצאה כי יש בתוכנית מחרוזת קבועה.
ג. (4 נקודות) נתגלה כי הקוד של הפרוצדורה printf מהספריה הסטנדארטית אינו נמצא.
ד. (4 נקודות) נתגלה כי **בכל** פעם שמתבצעת הפקודה x = 42 + y ערכו של y הינו 0.
ה. (4 נקודות) נמצא כי שמות כל המשתנים בתוכנית מתחילים באות x.

identifiers הוחלט

Exercise (From Exam)

שאלה 2 (20 נקודות)

נתונה רשימת מאורעות. לכל אחד מהמאורעות הסבירו בקצרה (1-3 שורות)

1. האם הוא מתרחש בזמן בניית הקומפיילר, בזמן קומפילציה או בזמן ריצה
2. אם בחרת בזמן קומפילציה, הסבר מהו השלב המסוים בקומפיילר בו מתרחש המאורע ומהם מבני הנתונים הרלוונטים.
3. אם בחרת בזמן ריצה, הסבר האם מתרחש לפני ביצוע התוכנית, בזמן הביצוע, או לאחריו.

א. (4 נקודות) התגלה כי המתכנת השתמש במילת מפתח (keyword) כשם משתנה.
ב. (4 נקודות) התגלתה בקוד פעולת חלוקה באפס.
ג. (4 נקודות) התגלתה מחלקה שיש בה שדה מטיפוס לא מוגדר.
ד. (4 נקודות) התגלה כי משתנה מסוים ערכו תמיד חיובי.
ה. (4 נקודות) התגלה כי הדקדוק של שפת התכנות הינו רב משמעי.

Exercise (From Exam)

שאלה 1 (20 נקודות)

נתונה רשימת מאורעות. לכל אחד מהמאורעות הסבירו בקצרה (3-1 שורות)

1. האם הוא מתרחש במהלך בניית הקומפיילר/הקומפילציה/loading/linking/ריצת התוכנית.
2. אם בחרת בזמן קומפילציה, הסבר מהו השלב המסוים בקומפיילר בו מתרחש המאורע ומהם מבני הנתונים הרלבנטיים.

א. (4 נקודות) התגלה כי ישנה שגיאה בקוד הבא: X+++++Y

- הניחו כי אופרטורים ++ ו + מותרים, וכי X ו Y הם מסוג int. (מותר להפעיל ++ רק ישירות על משתנים)

ב. (4 נקודות) התגלה כי אין די רגיסטרים במכונה עבור קוד הביניים הנתון.

ג. (4 נקודות) התגלה כי המתודה הנקראת לא מוגדרת.

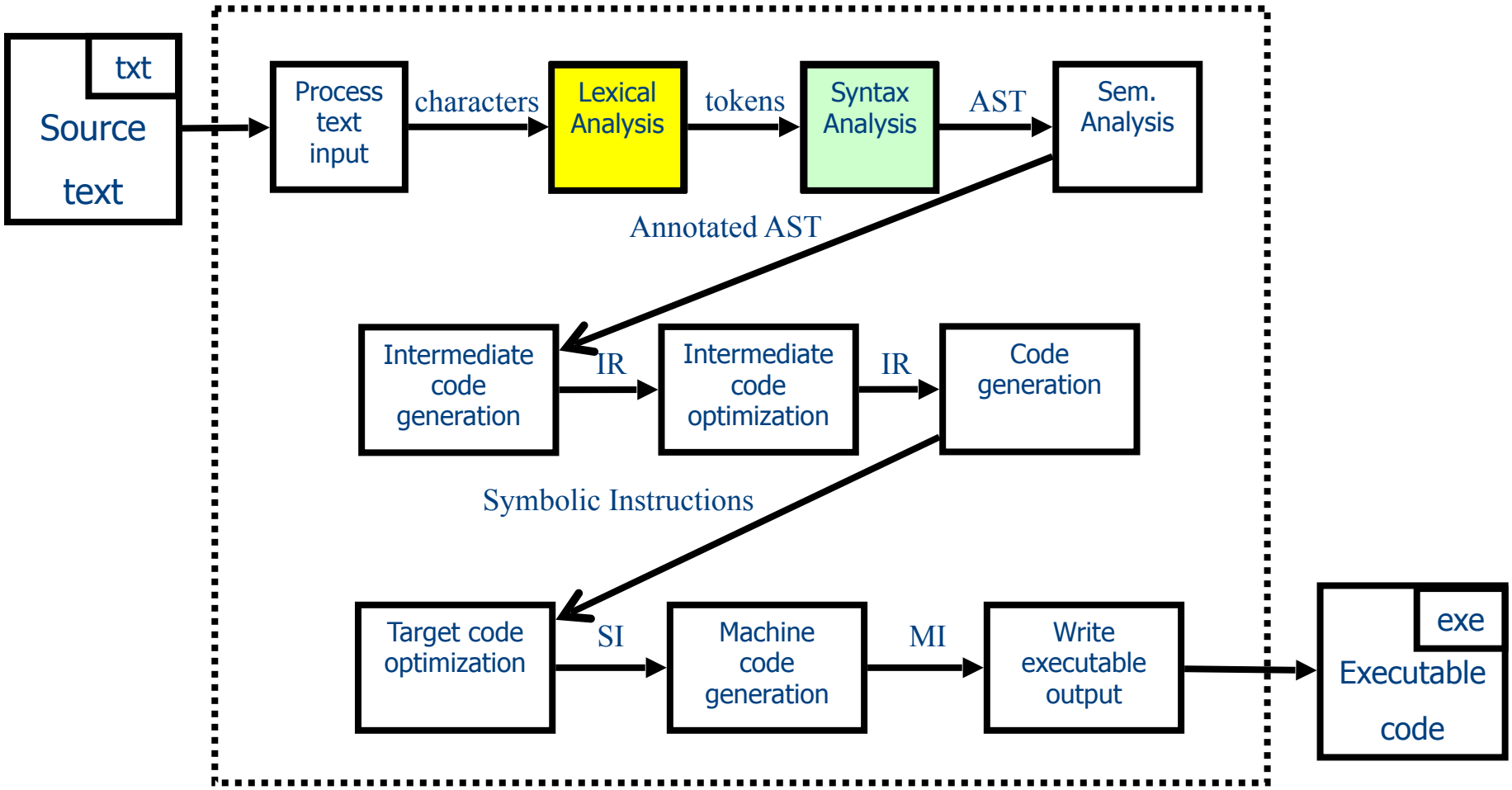
ד. (4 נקודות) התגלה כי אין ספריה סטנדרטית נדרשת.

ה. (4 נקודות) התגלה כי תמיד המתודה foo הנקראת שייכת למחלקת האב.

```
If (...) // ... = random choice
  x = new MyClass()
else
  x = new MySuperClass()
x.foo()
```

מה יקרה אם הקוד הוא X++ + ++Y? (רווחים מפרידים בין Tokens)

The Real Anatomy of a Compiler



The end