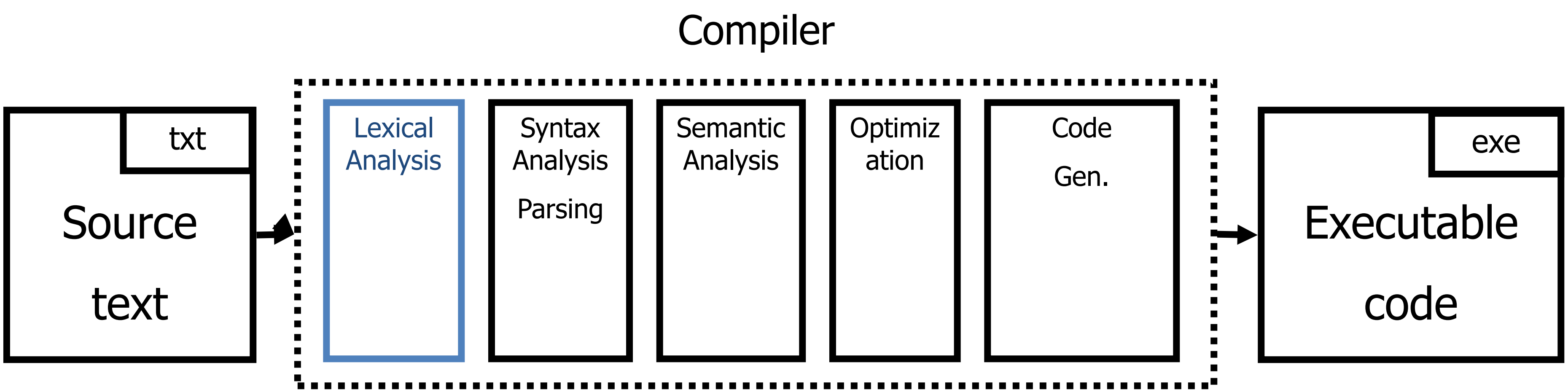
Introduction to Compilation

Lexical Analysis

You are here


Compiler
txt
Source
text
Lexical
Analysis
Syntax
Analysis
Parsing
Semantic
Analysis
Optimiz
ation
Code
Gen.
exe
Executable
code

Role of lexical analyzer

- Recognize tokens and ignore white spaces, comments

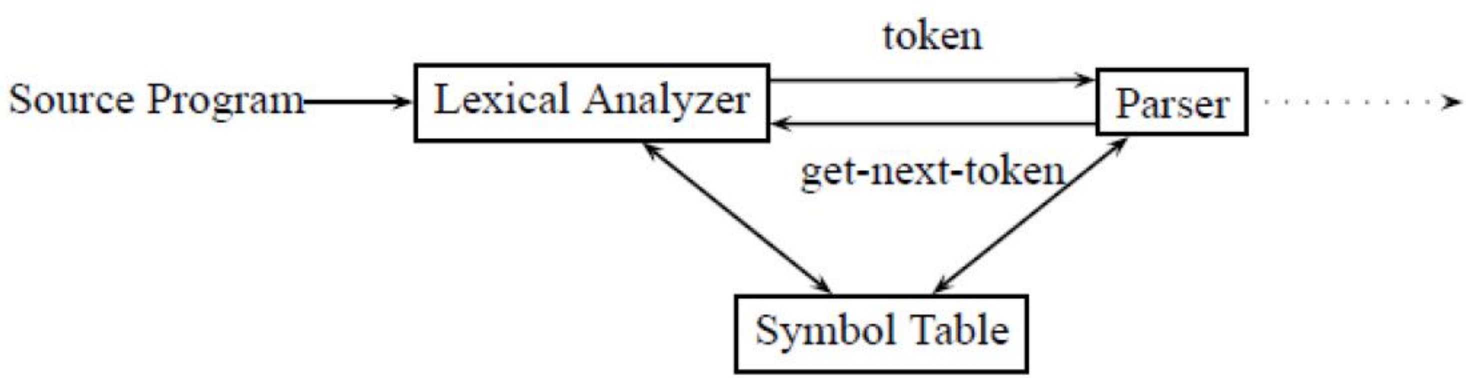
i	f		(	x	1		*	x	2	<	1	.	0	)	{

-
- Generates token stream

if	(	x1	*	x2	<	1.0	)	{

-
- Error reporting

Diagram is here



From characters to tokens

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

Tokens vs. lexemes

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

From characters to tokens

txt

x = b*b – 4*a*c

Token
Stream

<ID,”x”> <EQ> <ID,”b”> <MULT> <ID,”b”> <MINUS> <INT,4> <MULT> <ID,”a”> <MULT> <ID,”c”>

Tokens – using symbol table

Example. Let us consider the following assignment statement:

$$E := M * C * *2$$

then the following pairs $\langle$token,attribute$\rangle$ are passed to the Parser:

$\langle$id, pointer to symbol-table entry for E$\rangle$

$\langle$assign-op, $\rangle$

$\langle$id, pointer to symbol-table entry for M$\rangle$

$\langle$mult-op, $\rangle$

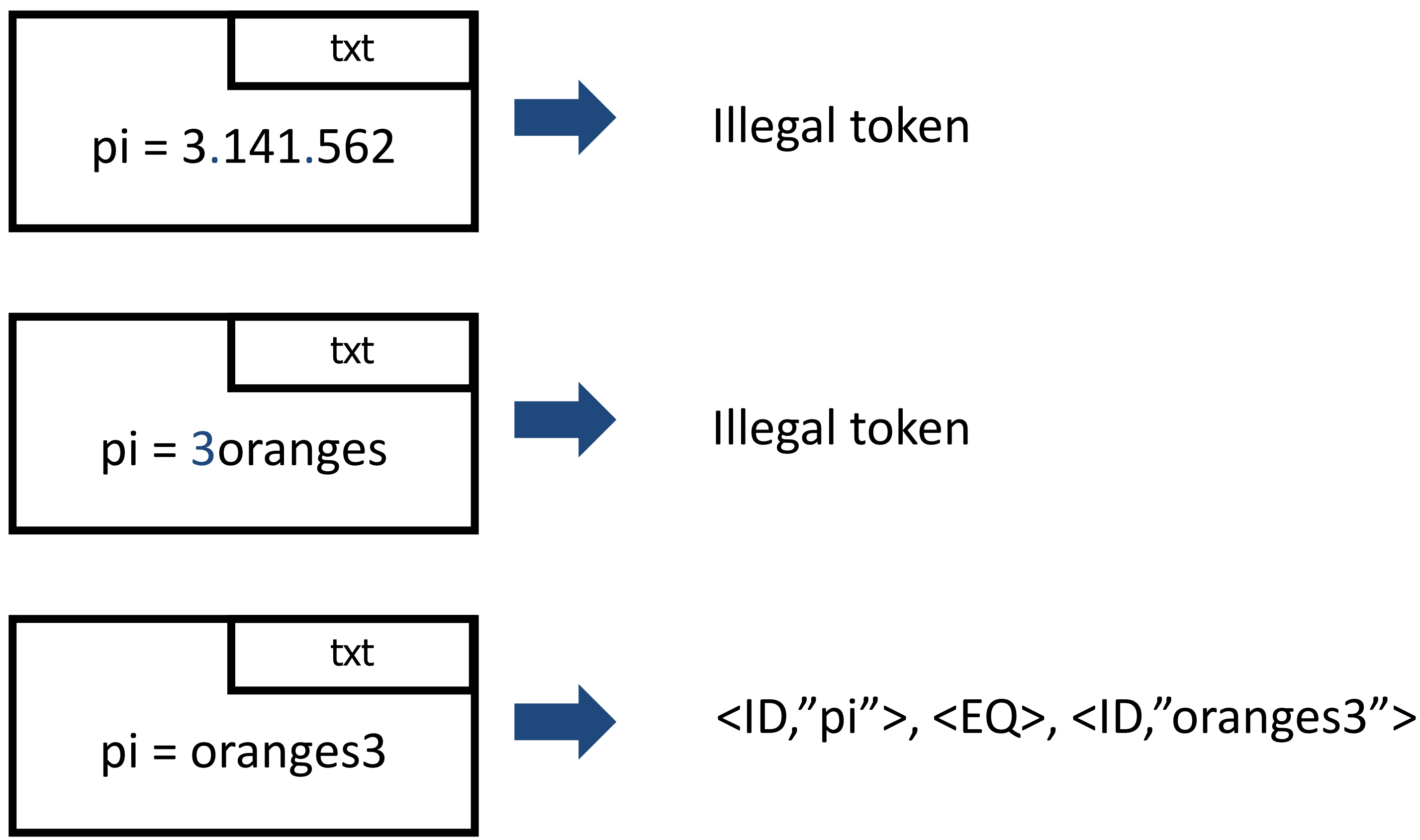
$\langle$id, pointer to symbol-table entry for C$\rangle$

$\langle$exp-op, $\rangle$

$\langle$num, integer value 2$\rangle$.

- Some Tokens have a null attribute: the Token is sufficient to identify the Lexeme.
- From an implementation point of view, each token is encoded as an integer number.

Errors in lexical analysis



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.

How can we define tokens?

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

- Identifiers?
- Numerical Values?
- Strings?

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

Regular Expressions

Basic Patterns	Matching
x	A single letter ‘x’ from the alphabet
.	Any character, usually except a new line
[xyz]	Any of the characters x,y,z
Repetition Operators	
R?	An R or nothing (ε | R)
R*	Zero or more occurrences of R
R+	One or more occurrences of R (RR*)
Composition Operators	
R1R2	An R1 followed by R2
R1|R2	Either an R1 or R2
Grouping	
(R)	R itself

Examples

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

Escape characters

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

Shorthands

- Use names for expressions
  - letter = a | b | … | z | A | B | … | Z
  - letter_ = letter | _
  - digit = 0 | 1 | 2 | … | 9
  - id = letter_ (letter_ | digit)*
- Use “-” 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 = $(0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9)^+$
  $(\varepsilon \mid . (0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9)^+)$
  $(\varepsilon \mid E (0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9)^+)$

- Using shorthands it can be written as (this time with negative exponent as well):

$$
\begin{array}{rcl}
\text{digit} & \rightarrow & 0 \mid 1 \mid \cdots \mid 9 \\
\text{digits} & \rightarrow & \text{digit}^+ \\
\text{optional-fraction} & \rightarrow & (.\text{digits})? \\
\text{optional-exponent} & \rightarrow & (\mathbf{E}(+ \mid -)?\text{digits})? \\
\text{num} & \rightarrow & \text{digits optional-fraction optional-exponent}
\end{array}
$$

Ambiguity

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

- "if" is a valid word in the language of identifiers... so what should it be?
- How about the identifier "iffy"?

- Solution
  - **Always find longest matching token**
  - Break ties using **order of definitions**... first definition wins (=> list rules for keywords before identifiers)

Creating a lexical analyzer

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

- How do we build an analyzer?

Character classification

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

Main reading routine

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

But we have a much better way!

- Generate a lexical analyzer **automatically** from token definitions

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

Overview

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

Reminder: Finite-State Automaton

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

Reminder: Finite-State Automaton

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

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

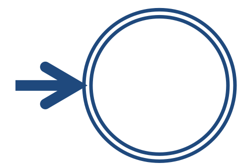
From regular expressions to NFA

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

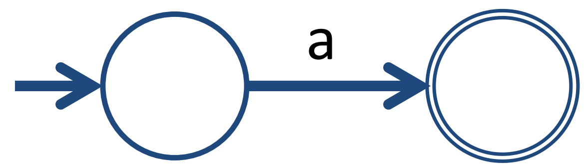
- Ambiguity resolution: prefer longest accepting word

Basic constructs

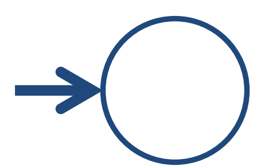
R = ε



R = a

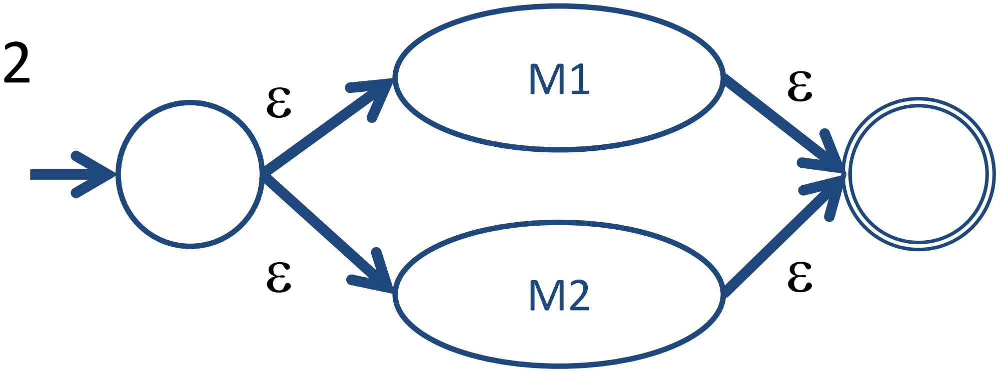


R = ϕ

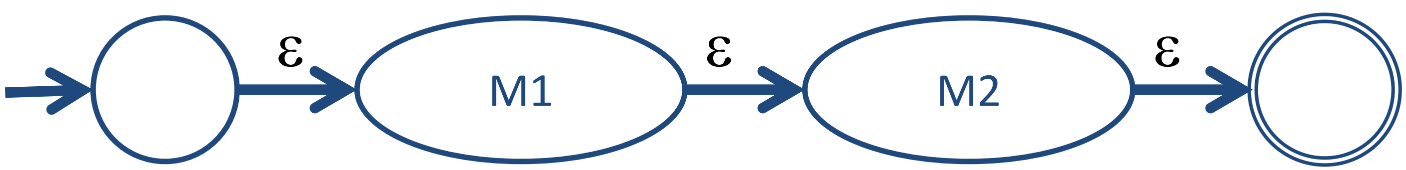


Composition

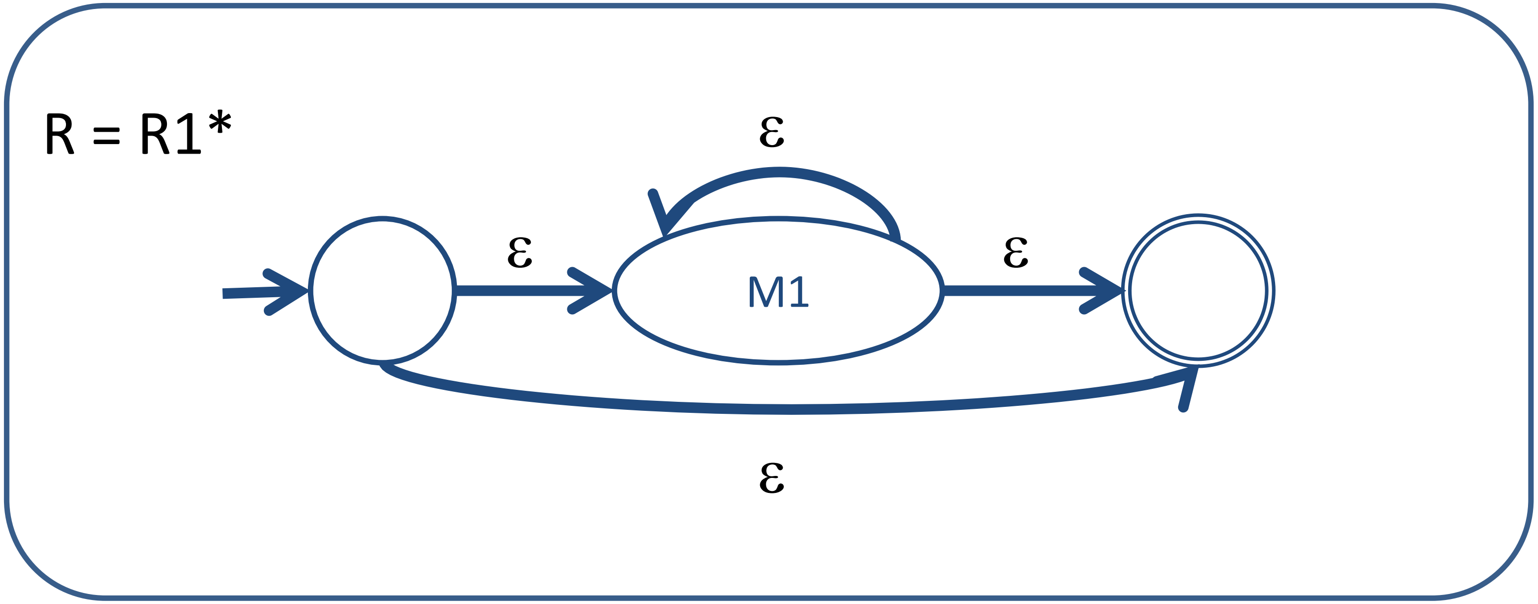
R = R1 | R2



R = R1R2



Repetition



What now?

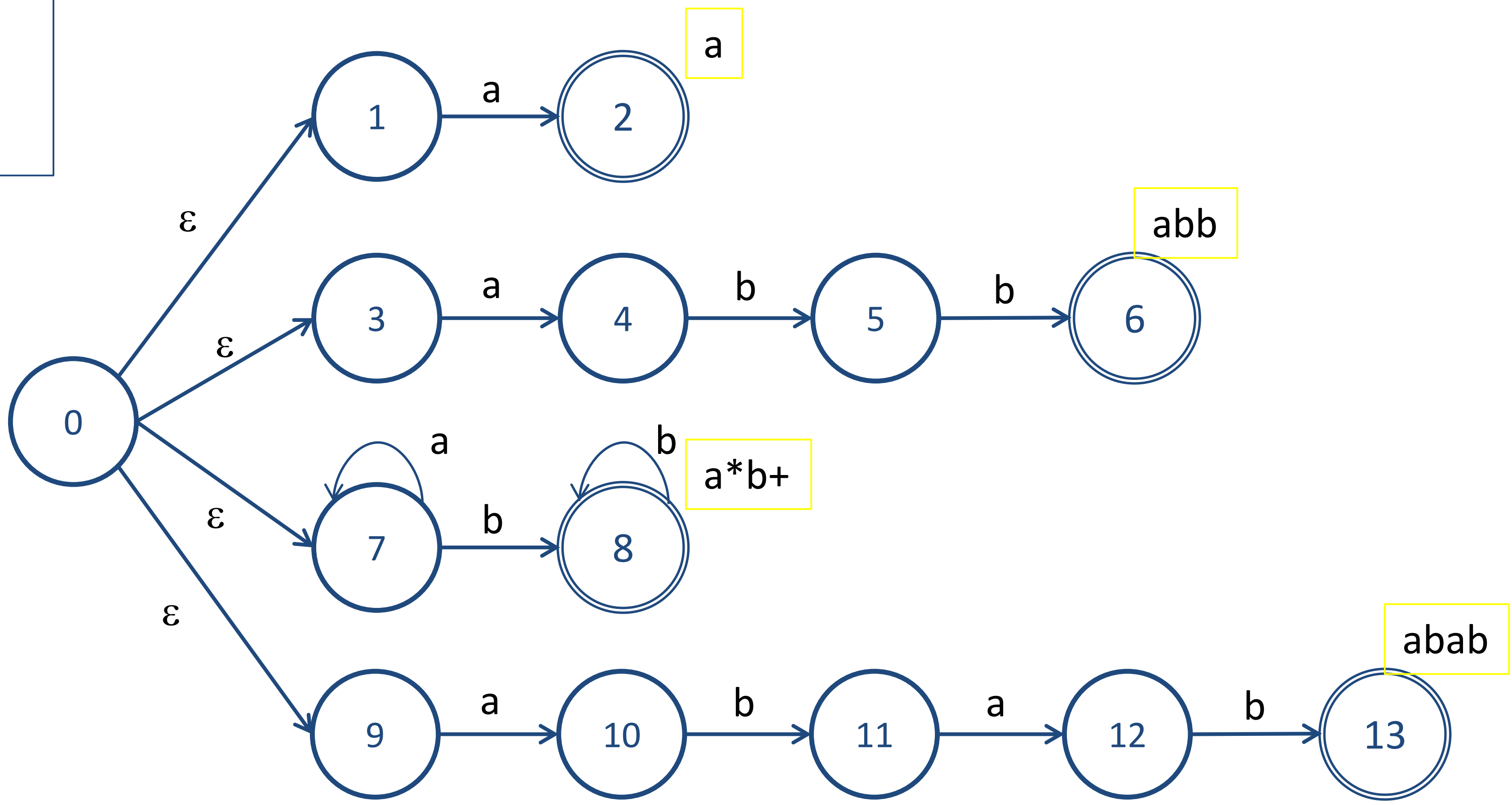
- Naïve approach: try each automaton separately
- Given a word w:
  - Try M1(w)
  - Try M2(w)
  - …
  - Try Mn(w)

- Requires resetting after every attempt

Combine automata

combines

a
abb
a*b+
abab

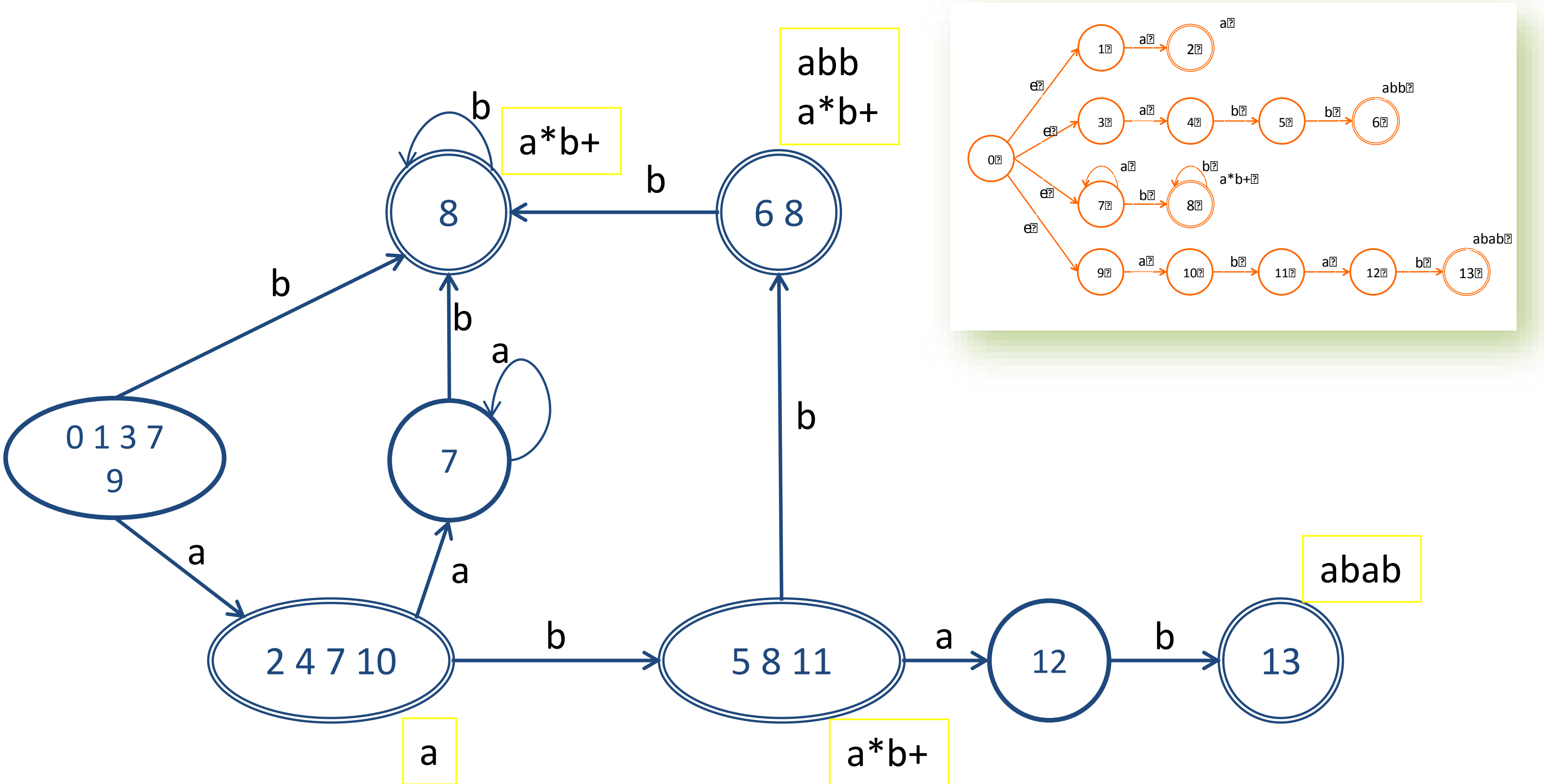


Ambiguity resolution

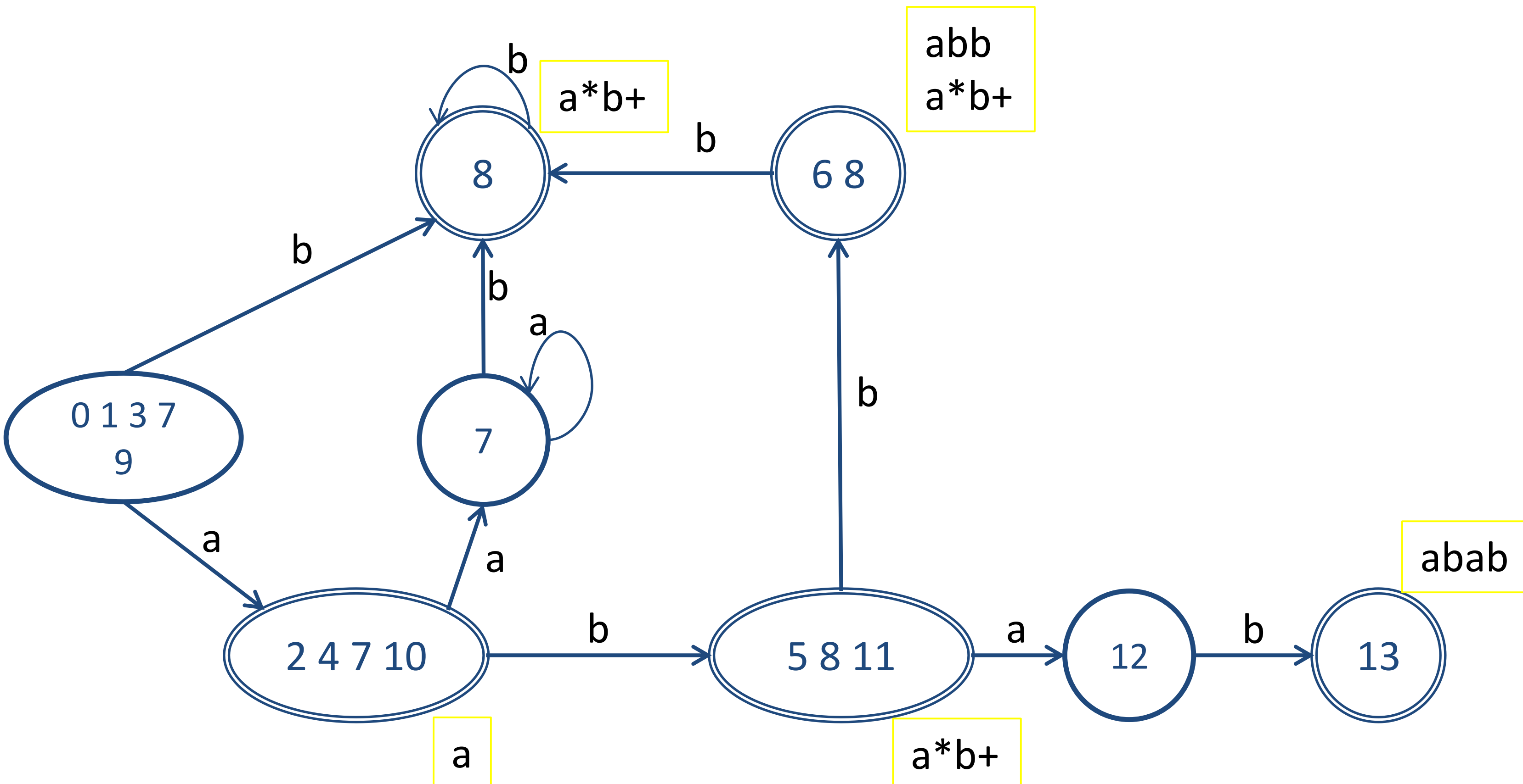
- Recall…
- Longest word
- Tie-breaker based on order of rules when words have same length

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

Corresponding DFA



Examples



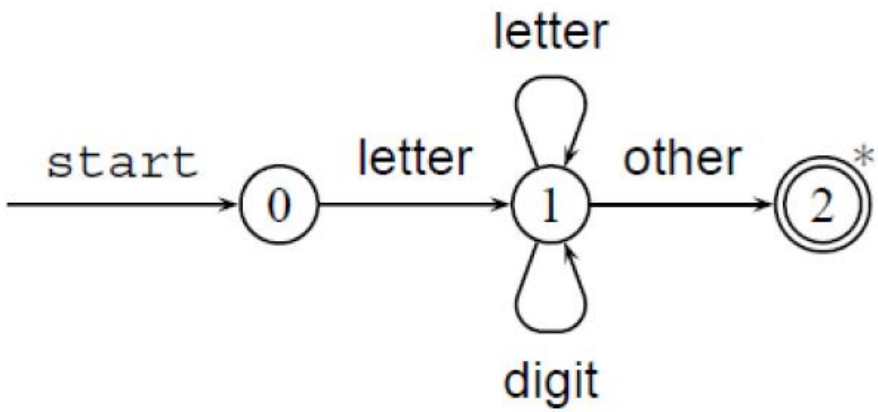
abaa: gets stuck after aba in state 12, backs up to state (5 8 11) pattern is a*b+, token is ab

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

Maximal lexemes strategy

Solution 1. **Use Automata with *Lookahead*.**

- *Example*: Automaton for id with lookahead.



- The label other refers to any character not indicated by any other edges living the node;
- The $*$ indicates that we read an extra character and we must retract the *forward* input pointer by one character.

From NFA to DFA

- NFA are hard to simulate with a computer program.
  1. There are many possible paths for a given input string caused by the nondeterminism;
  2. The acceptance condition says that there must be *at least one* path ending with a final state;
  3. We may need to find all the paths before accepting/excluding a string.
- The algorithm to map an NFA to a DFA is called the *Subset Construction*.
- *Main Idea:* Each DFA state corresponds to a set of NFA states, thus encoding all the possible states an NFA could reach after reading an input symbol.
- The algorithm makes use of the operation ϵ-closure: Given a set of states, T, ϵ-closure(T) returns the set of NFA states reachable from some $s \in T$ on ϵ-transition only.

ε-closure algorithm

```
ϵ-closure(T)
    CL_T := T;                    /* Current Closure of T */
    NOT_EX := CL_T;               /* Not Examined States */
    repeat                        /* Updates the Closure of T */
        NEW_CL := ∅;
        for each t in NOT_EX do
            Q := δ(t, ϵ);
            if Q ⊄ CL_T then
                NEW_CL := NEW_CL ∪ (Q - CL_T);
        end
        CL_T := CL_T ∪ NEW_CL;
        NOT_EX := NEW_CL;
    until NEW_CL == ∅
return CL_T                       /* ϵ-Closure of T */
```

NFA to DFA definition

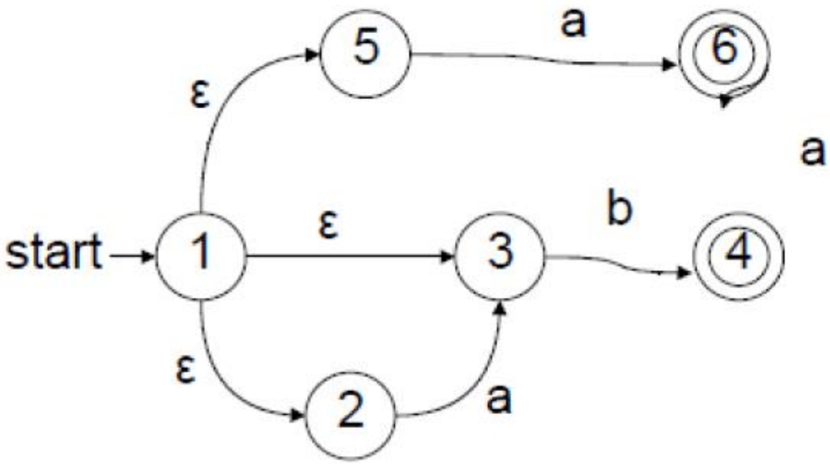
Let NFA = $(S, \mathsf{V}, \delta, s_0, F)$ then the equivalent DFA = $(S', \mathsf{V}, \delta', s_0', F')$ where:

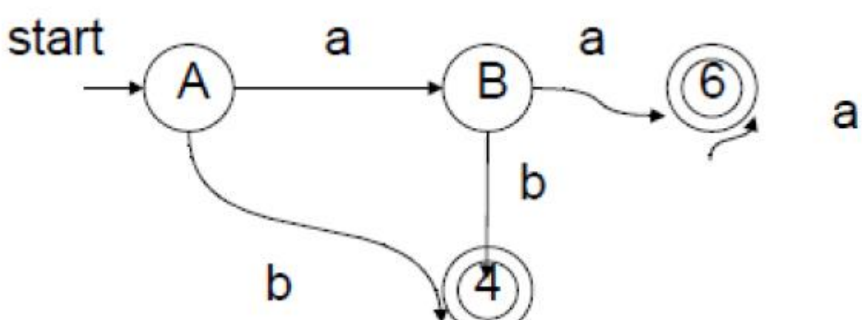
- $S' \subseteq 2^S$. With $[s_1, \ldots, s_n]$ we denote an element in S', which stands for the set of states $\{s_1, ..., s_n\}$.
- $s_0' = [\epsilon\text{-closure}(\{s_0\})]$;
- F' is the set of states in S' containing some element in F;
- $\delta'([s_1, \ldots, s_n], a) = [q_1, \ldots, q_m]$ iff $\{q_1, \ldots, q_m\} = \epsilon\text{-closure}(\delta(\{s_1, \ldots, s_n\}, a))$, where $\delta(\{s_1, \ldots, s_n\}, a) = \bigcup_{i=1}^{n} \delta(s_i, a)$.

NFA to DFA – Subset construction algorithm

```
Subset_Construction()                      /* Subset Construction algorithm */
  DS := {ε-closure({s0})};                  /* Current Deterministic states */
  NOT_EX := DS;                             /* Not Examined States */
  repeat                                    /* DFA Construction */
    NEW_DS := ∅;
    for each T in NOT_EX do
      for each symbol a ∈ V do
        Q := ε-closure(δ(T, a));
        if Q ∉ DS then NEW_DS := NEW_DS ∪ Q;
        δ'(T, a) := Q;                      /* DFA's Transition Function update */
      end
    end
    DS := DS ∪ NEW_DS;                      /* DFA's States update */
    NOT_EX := NEW_DS;
  until NEW_DS == ∅
return DS, δ'                               /* DFA's States and Transition Function */
```

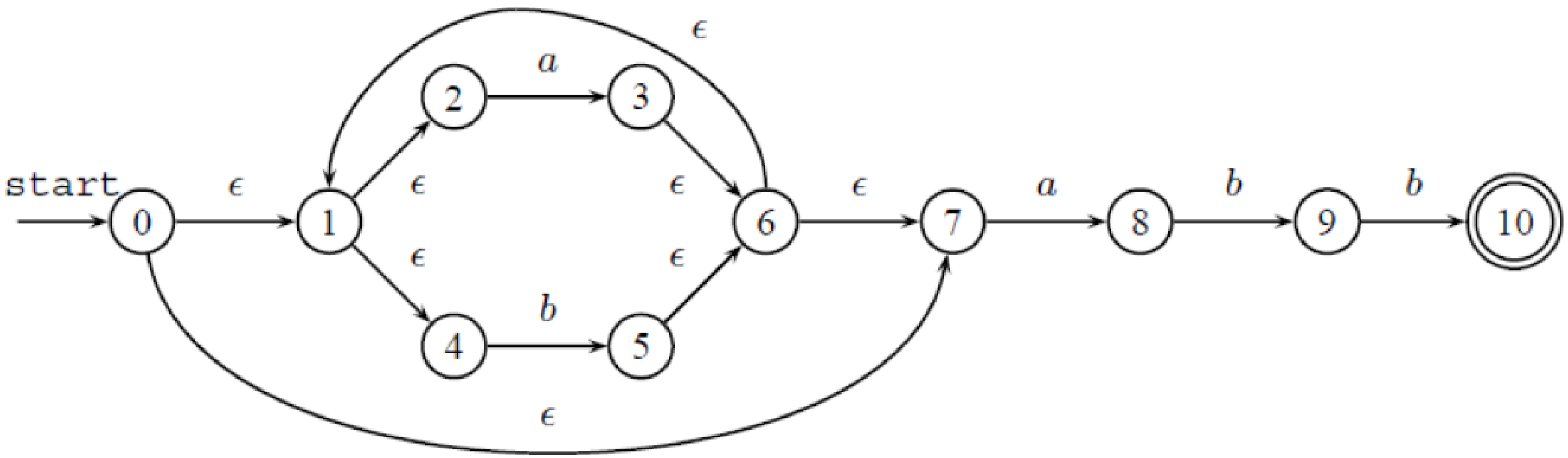
NFA to DFA example

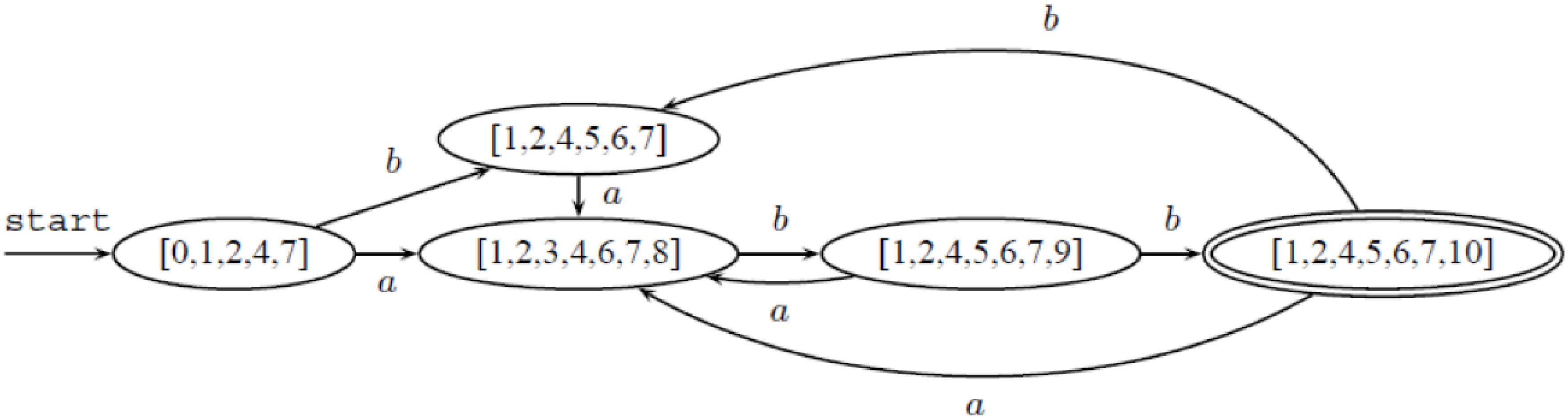




- ε-closure(1) = {1, 2, 3, 5}
- Create a new state A = {1, 2, 3, 5} and examine transitions out of it
- move(A, a) = {3, 6}
- Call this a new subset state = B = {3, 6}
- move(A, b) = {4}
- move(B, a) = {6}
- move(B, b) = {4}
- Complete by checking move(4, a); move(4, b); move(6, a); move(6, b)

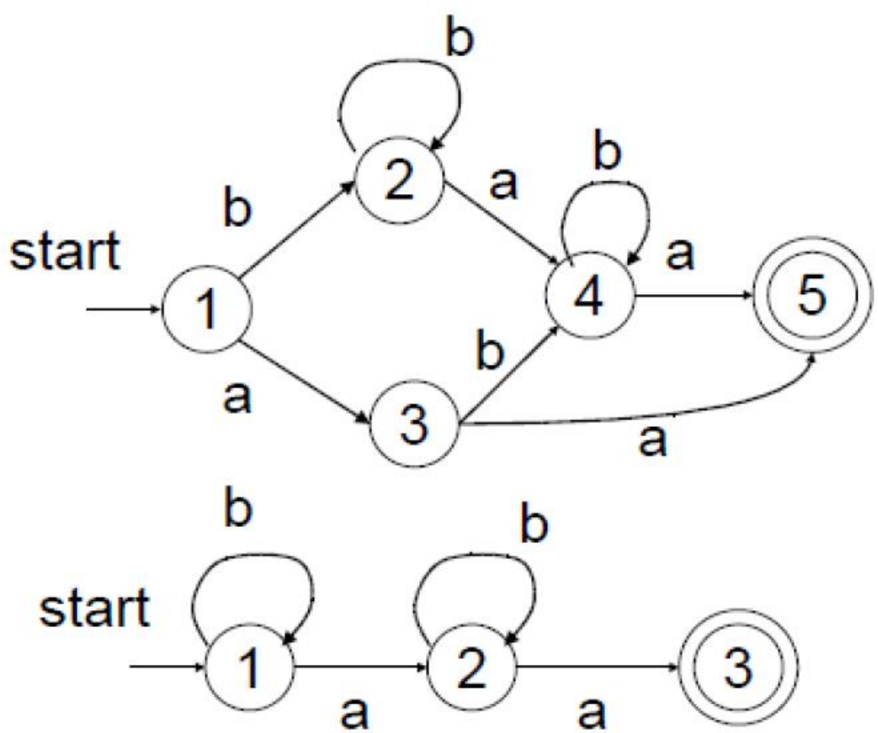
NFA to DFA example 2





State minimization

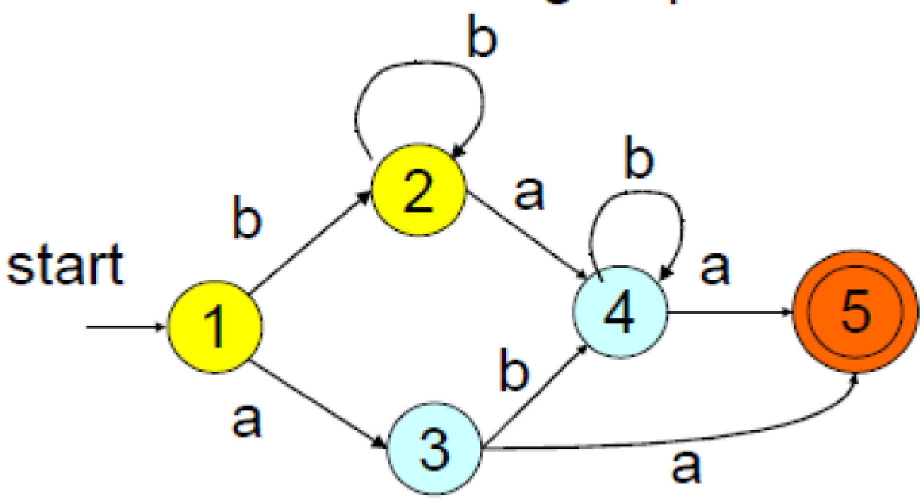
- Resulting DFA can be quite large
  - Contains redundant or equivalent states



Both DFAs accept b*ab*a

State minimization - strategy

- Idea – find groups of equivalent states and merge them
  - All transitions from states in group G1 go to states in another group G2
  - Construct minimized DFA such that there is 1 state for each group of states



Basic strategy: identify distinguishing transitions

State minimization - algorithm

$P \leftarrow \{ F, \{Q\text{-}F\}\}$

while (P is still changing)

  $T \leftarrow \{\}$

  for each set $s \in P$

    for each $\alpha \in \Sigma$

      partition s by α

        into $s_1, s_2, \ldots, s_k$

      $T \leftarrow T \cup s_1, s_2, \ldots, s_k$

  if $T \neq P$ *then*

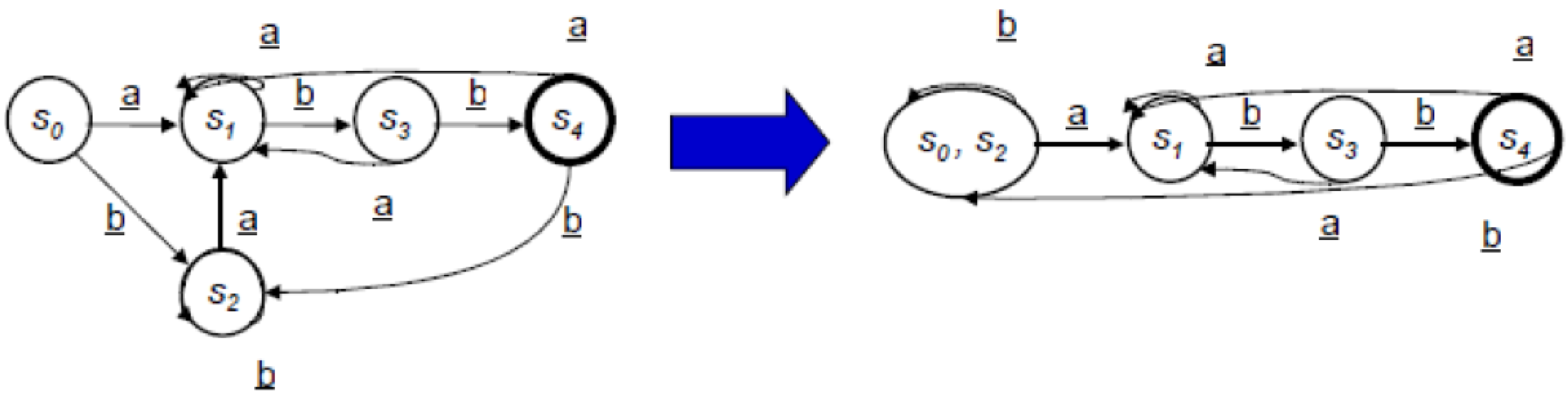
    $P \leftarrow T$

State minimization - example

example: (a | b)* abb

	Current Partition	*Split on a*	*Split on b*
P_0	$\{s_4\}\{s_0, s_1, s_2, s_3\}$	none	$\{s_0, s_1, s_2\}\{s_3\}$
P_1	$\{s_4\}\{s_3\}\{s_0, s_1, s_2\}$	none	$\{s_0, s_2\}\{s_1\}$
P_2	$\{s_4\}\{s_3\}\{s_1\}\{s_0, s_2\}$	none	None

final state

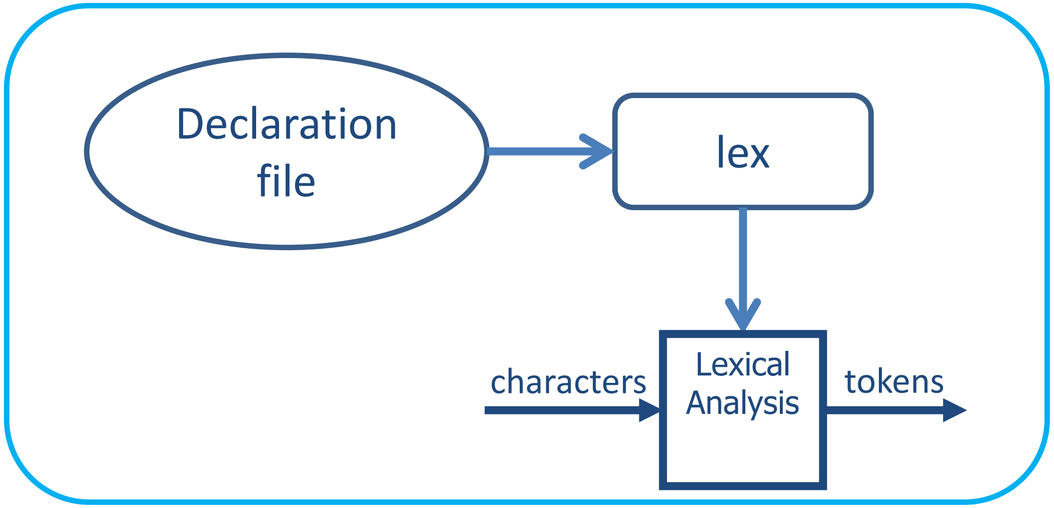


Summary of Construction

- Describe tokens as regular expressions.
- Regular expressions are turned into NFA.
- NFA is turned to DFA via subset constriction.
- State minimization of DFA.
- Lexical analyzer simulates the run of an automata with the given transition table on any input string.

Good News

- 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