Chapter 2 Lexical Analysis

Nai-Wei Lin

共勉

子曰：「學而時習之，不亦說乎？」

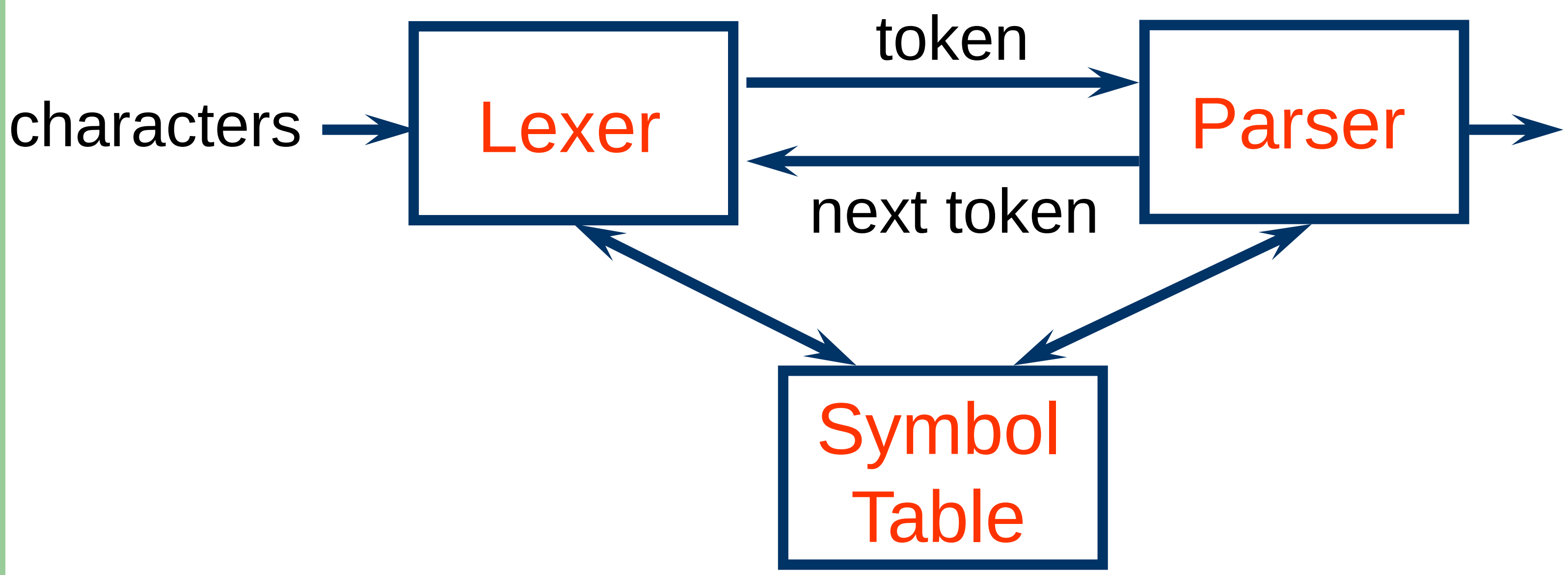
Lexical Analysis

- Lexical analysis recognizes the vocabulary of the programming language and transforms a string of characters into a string of words or tokens
- Lexical analysis discards white spaces and comments between the tokens
- Lexer is the program that performs lexical analysis

Outline

- Lexers
- Tokens
- Regular expressions
- Finite automata
- Automatic conversion from regular expressions to finite automata
- A lexer generator — ANTLR

Lexers



Tokens

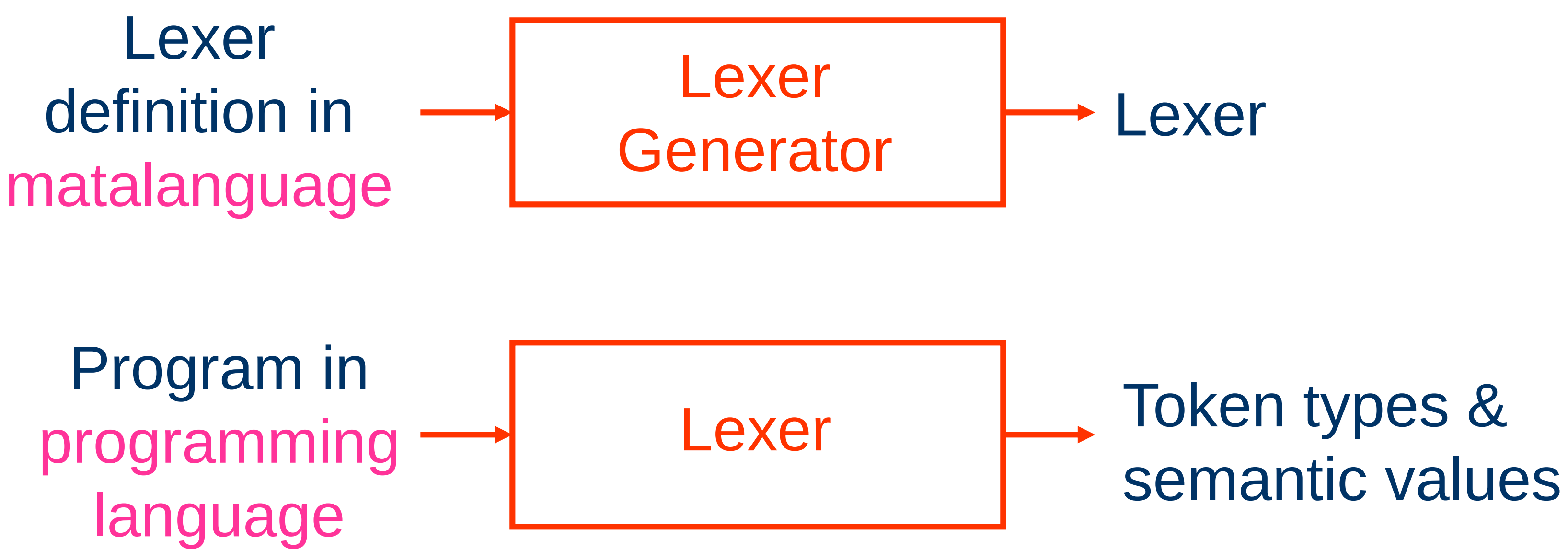
- A token is a sequence of characters that can be treated as a unit in the grammar of a programming language
- A programming language classifies tokens into a finite set of token types

Type	Examples
ID	foo i n
NUM	73 13
IF	if
COMMA	,

Semantic Values of Tokens

- Semantic values are used to distinguish different tokens in a token type
  - < ID, foo>, < ID, i >, < ID, n >
  - < NUM, 73>, < NUM, 13 >
  - < IF, >
  - < COMMA, >
- Token types affect syntax analysis and semantic values affect semantic analysis

Lexer Generators



Languages

- A language is a set of strings
- A string is a finite sequence of symbols taken from a finite alphabet
  - The C language is the (infinite) set of all strings that constitute legal C programs
  - The language of C reserved words is the (finite) set of all alphabetic strings that cannot be used as identifiers in the C programs
  - Each token type is a language

Regular Expressions (RE)

- A language allows us to use a finite description to specify a (possibly infinite) set
- RE is the metalanguage used to define the token types of a programming language

Regular Expressions

- ε is a RE denoting L = {ε}
- If *a* $\in$ alphabet, then *a* is a RE denoting L = {*a*}
- Suppose *r* and *s* are RE denoting L(*r*) and L(*s*)
  - alternation: (*r*) | (*s*) is a RE denoting L(*r*) $\cup$ L(*s*)
  - concatenation: (*r*) • (*s*) is a RE denoting L(*r*)L(*s*)
  - repetition: $(r)^*$ is a RE denoting $(L(r))^*$
  - (*r*) is a RE denoting L(*r*)

Examples

- $a \mid b$ $\{a, b\}$
- $(a \mid b)(a \mid b)$ $\{aa, ab, ba, bb\}$
- a^* $\{\varepsilon, a, aa, aaa, ...\}$
- $(a \mid b)^*$ the set of all strings of a's and b's
- $a \mid a^*b$ the set containing the string a and all strings consisting of zero or more a's followed by a b

Regular Definitions

- Names for regular expressions

$$d_1 \rightarrow r_1$$
$$d_2 \rightarrow r_2$$
$$\ldots$$
$$d_n \rightarrow r_n$$

  where r_i over alphabet $\cup \{d_1, d_2, \ldots, d_{i-1}\}$

- Examples:

  letter $\rightarrow$ A | B | … | Z | a | b | … | z

  digit $\rightarrow$ 0 | 1 | … | 9

  identifier $\rightarrow$ letter (letter | digit)*

Notational Abbreviations

- One or more instances

  $(r)^+$ denoting $(L(r))^+$

  $r^* = r^+ \mid \varepsilon \qquad r^+ = r\, r^*$

- Zero or one instance

  $r? = r \mid \varepsilon$

- Character classes

  [abc] = a | b | c   [a-z] = a | b | ... | z

  [^abc] = any character except a | b | c

- Any character except newline

  .

Examples

- 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”)+
  {/*do nothing for white spaces and comments*/}
- . { error(); }

Completeness of REs

- A lexical specification should be complete; namely, it always matches some initial substring of the input

```
…
.                                      /* match any */
```

Disambiguity of REs (1)

- Longest match disambiguation rules: the longest initial substring of the input that can match any regular expression is taken as the next token

([0-9]+“.”[0-9]*)|([0-9]*“.”[0-9]+)   /* REAL */

0.9

Disambiguity of REs (2)

- Rule priority disambiguation rules: for a particular longest initial substring, the first regular expression that can match determines its token type

```
if                      /* IF */
[a-z][a-z0-9]*          /* ID */
```

if

Finite Automata

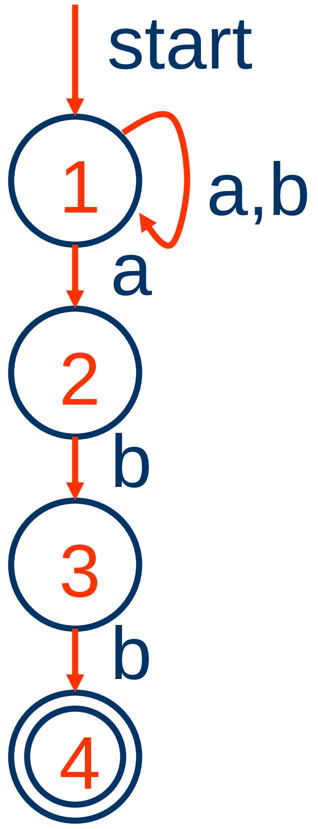
- A finite automaton is a finite-state transition diagram that can be used to model the recognition of a token type specified by a regular expression
- A finite automaton can be a nondeterministic finite automaton or a deterministic finite automaton

Nondeterministic Finite Automata (NFA)

- An NFA consists of
  - A finite set of states
  - A finite set of input symbols
  - A transition function that maps (state, symbol) pairs to sets of states
  - A state distinguished as start state
  - A set of states distinguished as final states

An Example

- RE: $(a \mid b)^{*}abb$
- States: {1, 2, 3, 4}
- Input symbols: {a, b}
- Transition function:
  (1,a) = {1,2}, (1,b) = {1}
  (2,b) = {3}, (3,b) = {4}
- Start state: 1
- Final state: {4}

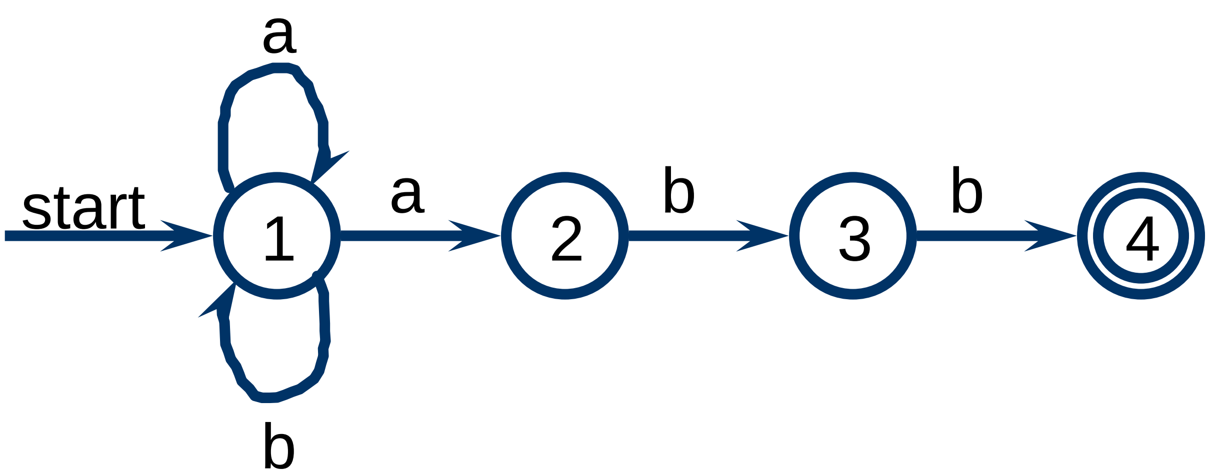


Acceptance of NFA

- An NFA accepts an input string s iff there is some path in the finite-state transition diagram from the start state to some final state such that the edge labels along this path spell out s
- The language recognized by an NFA is the set of strings it accepts

An Example

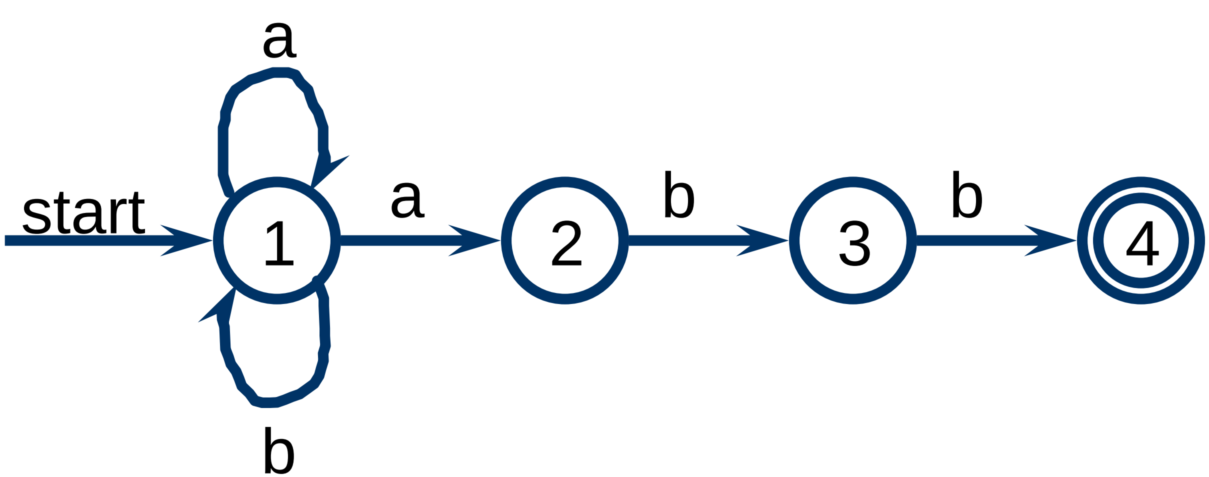
$(a \mid b)^{*}abb$ aabb



$\{1\} \underset{a}{\rightarrow} \{1,2\} \underset{a}{\rightarrow} \{1,2\} \underset{b}{\rightarrow} \{1,3\} \underset{b}{\rightarrow} \{1,4\}$

An Example

$(a \mid b)^{*}abb$ aaba



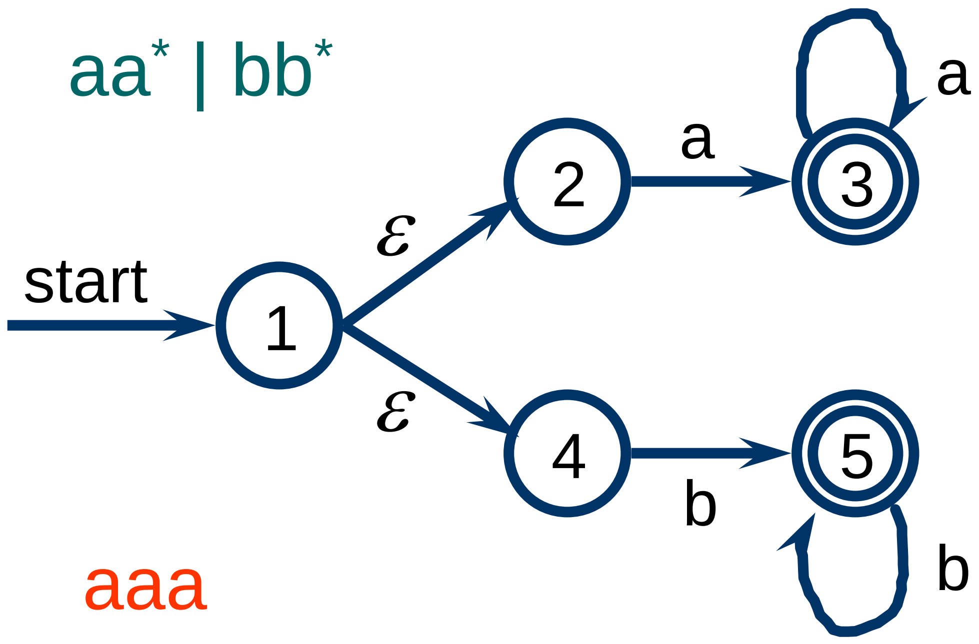
$\{1\} \xrightarrow[a]{} \{1,2\} \xrightarrow[a]{} \{1,2\} \xrightarrow[b]{} \{1,3\} \xrightarrow[a]{} \{1, 2\}$

Another Example

- RE: $aa^* \mid bb^*$
- States: {1, 2, 3, 4, 5}
- Input symbols: {a, b}
- Transition function:
  $(1, \varepsilon) = \{2, 4\}$, $(2, a) = \{3\}$, $(3, a) = \{3\}$,
  $(4, b) = \{5\}$, $(5, b) = \{5\}$
- Start state: 1
- Final states: {3, 5}

Finite-State Transition Diagram

$aa^* \mid bb^*$



aaa

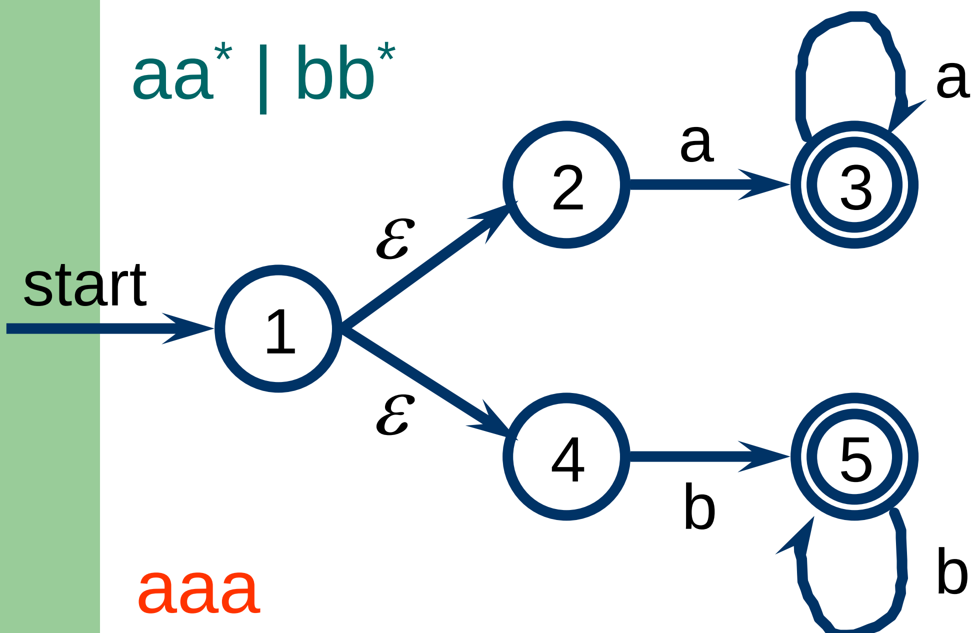
$\{1\} \xrightarrow[\varepsilon]{} \{1,2,4\} \xrightarrow[a]{} \{3\} \xrightarrow[a]{} \{3\} \xrightarrow[a]{} \{3\}$

Operations on NFA states

- ε-*closure*(*s*): set of states reachable from a state *s* on ε-transitions alone
- ε-*closure*(*S*): set of states reachable from some state *s* in *S* on ε-transitions alone
- *move*(*s*, *c*): set of states to which there is a transition on input symbol *c* from a state *s*
- *move*(*S*, *c*): set of states to which there is a transition on input symbol *c* from some state *s* in *S*

An Example

$aa^* \mid bb^*$



aaa

$S_0 = \{1\}$
$S_1 = \varepsilon\text{-}closure(\{1\}) = \{1,2,4\}$
$S_2 = move(\{1,2,4\},a) = \{3\}$
$S_3 = \varepsilon\text{-}closure(\{3\}) = \{3\}$
$S_4 = move(\{3\},a) = \{3\}$
$S_5 = \varepsilon\text{-}closure(\{3\}) = \{3\}$
$S_6 = move(\{3\},a) = \{3\}$
$S_7 = \varepsilon\text{-}closure(\{3\}) = \{3\}$
3 is in {3, 5} $\Rightarrow$ accept

$$\{1\} \xrightarrow[\varepsilon]{} \{1,2,4\} \xrightarrow[a]{} \{3\} \xrightarrow[\varepsilon]{} \{3\} \xrightarrow[a]{} \{3\} \xrightarrow[\varepsilon]{} \{3\} \xrightarrow[a]{} \{3\} \xrightarrow[\varepsilon]{} \{3\}$$

Simulating an NFA

Input: An input string ended with **eof** and an NFA with start state s_0 and final states F.

Output: The answer "yes" if accepts, "no" otherwise.

begin

$S := \varepsilon\text{-}closure(\{s_0\})$; $c := nextchar$;

while c <> **eof do begin**

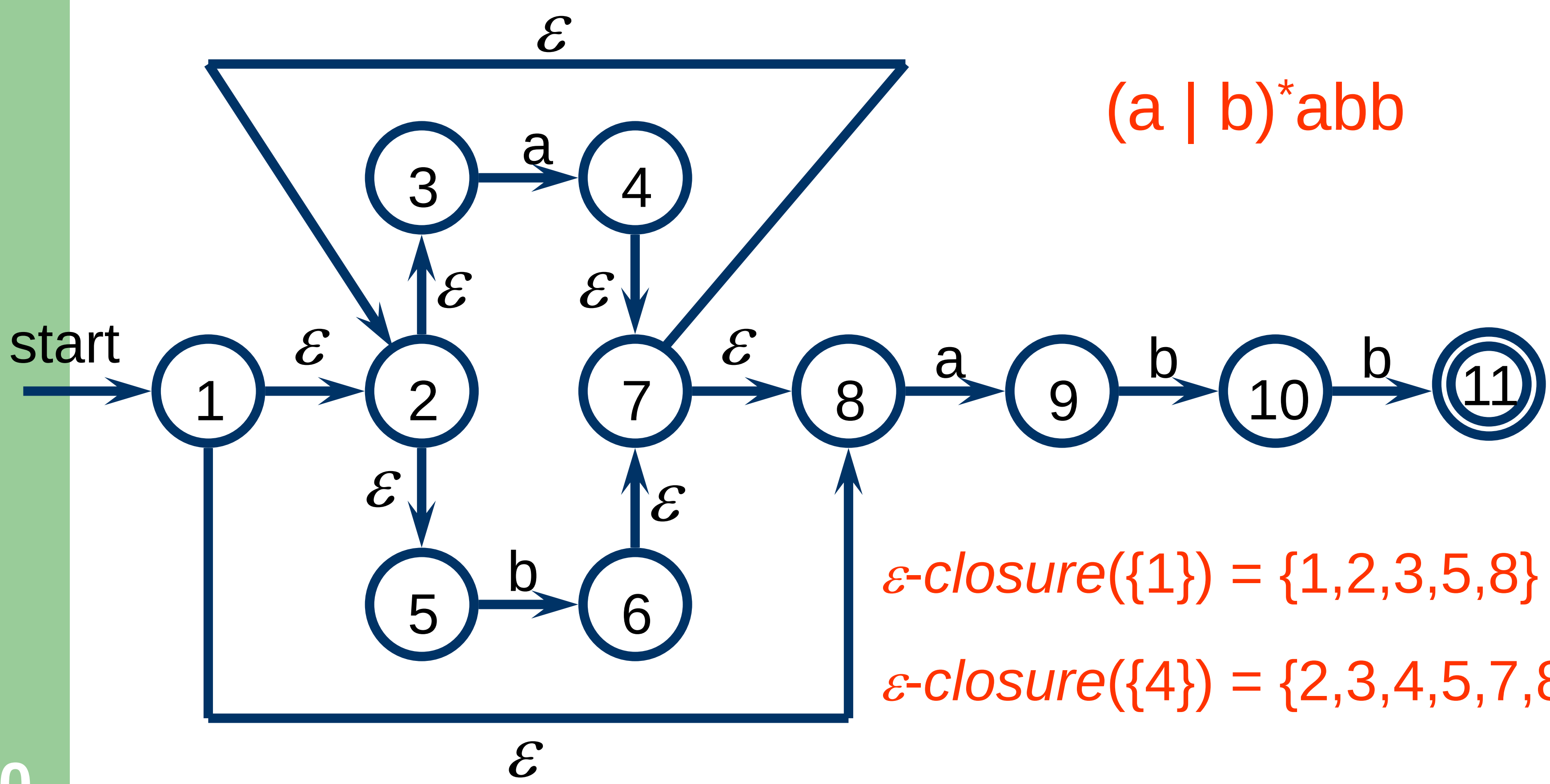
$S := \varepsilon\text{-}closure(move(S, c))$; $c := nextchar$

end;

if $S \cap F \neq \emptyset$ **then return** "yes" **else return** "no"

end.

Computation of ε-closure



(a | b)*abb

ε-*closure*({1}) = {1,2,3,5,8}

ε-*closure*({4}) = {2,3,4,5,7,8}

Computation of ε-closure

Input: An NFA and a set of NFA states S.

Output: $T = \varepsilon\text{-}closure(S)$.

```
begin
  push all states in S onto stack;  T := S;
  while stack is not empty do begin
    pop t, the top element, off of stack;
    for each state u with an edge from t to u labeled ε do
      if u is not in T then begin
        add u to T;   push u onto stack
      end
  end;
  return T
end.
```

Deterministic Finite Automata (DFA)

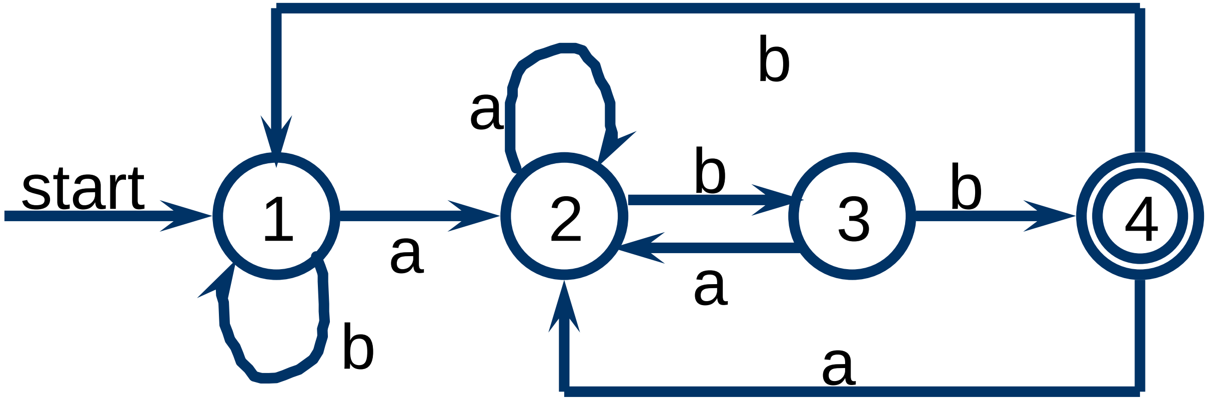
- A DFA is a special case of an NFA in which
- no state has an ε-transition
- for each state s and input symbol a, there is at most one edge labeled a leaving s

An Example

- RE: $(a \mid b)^{*}abb$
- States: {1, 2, 3, 4}
- Input symbols: {a, b}
- Transition function:
  (1,a) = 2, (2,a) = 2, (3,a) = 2, (4,a) = 2
  (1,b) = 1, (2,b) = 3, (3,b) = 4, (4,b) = 1
- Start state: 1
- Final state: {4}

Finite-State Transition Diagram

$(a \mid b)^{*}abb$



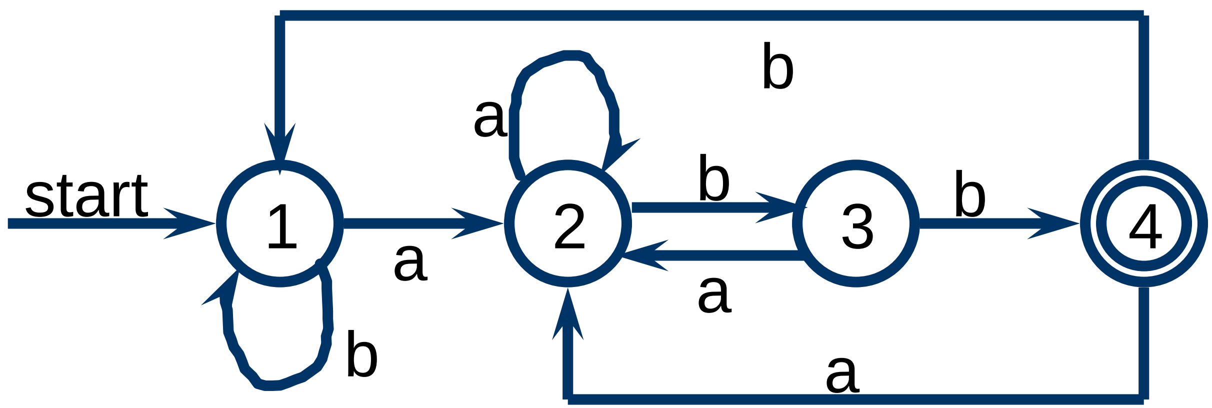
Acceptance of DFA

- A DFA accepts an input string s iff there is one path in the finite-state transition diagram from the start state to some final state such that the edge labels along this path spell out s
- The language recognized by a DFA is the set of strings it accepts

An Example

(a | b)*abb

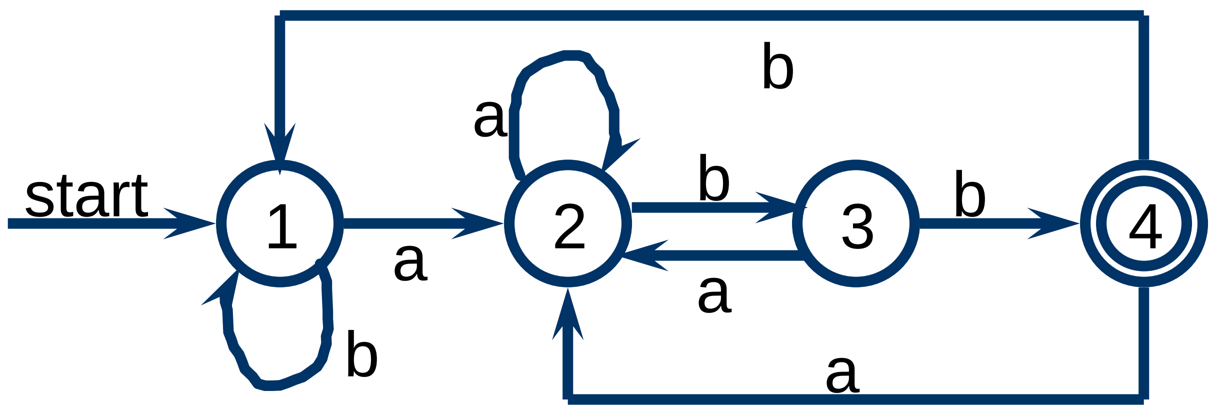
aabb



$$1 \xrightarrow[a]{} 2 \xrightarrow[a]{} 2 \xrightarrow[b]{} 3 \xrightarrow[b]{} 4$$

An Example

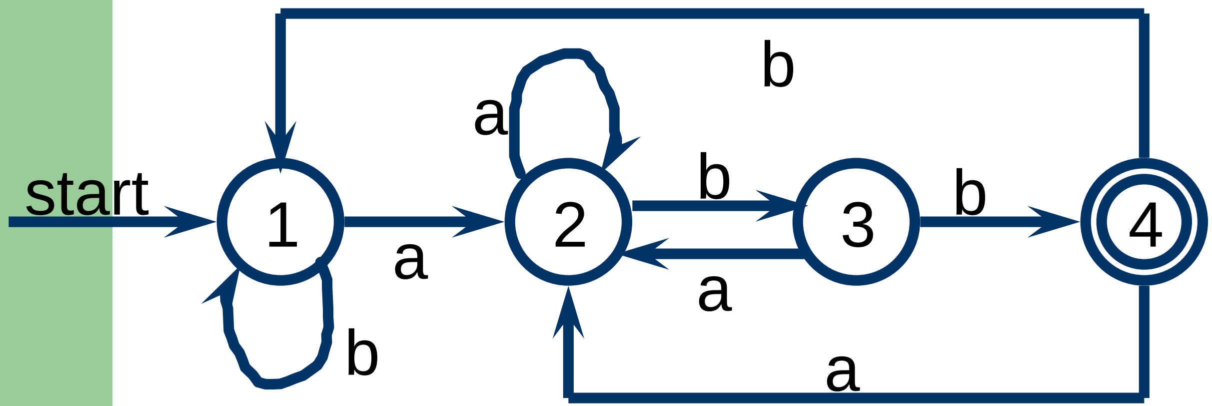
$(a \mid b)^{*}abb$ aaba



$$1 \xrightarrow[a]{} 2 \xrightarrow[a]{} 2 \xrightarrow[b]{} 3 \xrightarrow[a]{} 2$$

An Example

$(a \mid b)^*abb$



bbababb

$s = 1$

$s = move(1, b) = 1$

$s = move(1, b) = 1$

$s = move(1, a) = 2$

$s = move(2, b) = 3$

$s = move(3, a) = 2$

$s = move(2, b) = 3$

$s = move(3, b) = 4$

4 is in {4} $\Rightarrow$ accept

Simulating a DFA

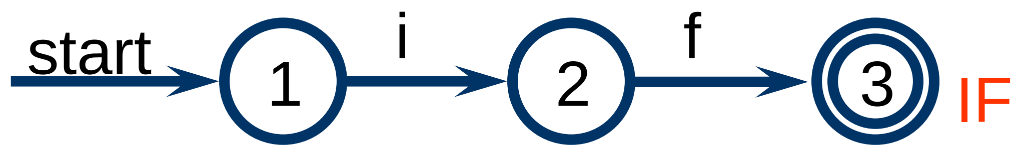
Input: An input string ended with **eof** and a DFA with start state s_0 and final states F.

Output: The answer "yes" if accepts, "no" otherwise.

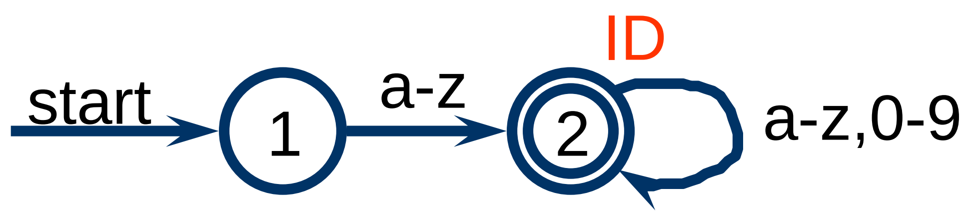
begin
  $s := s_0$; $c := nextchar$;
  while c <> **eof do begin**
    $s := move(s, c)$; $c := nextchar$
  end;
  if s is in F **then return** "yes" **else return** "no"
end.

Combined Finite Automata

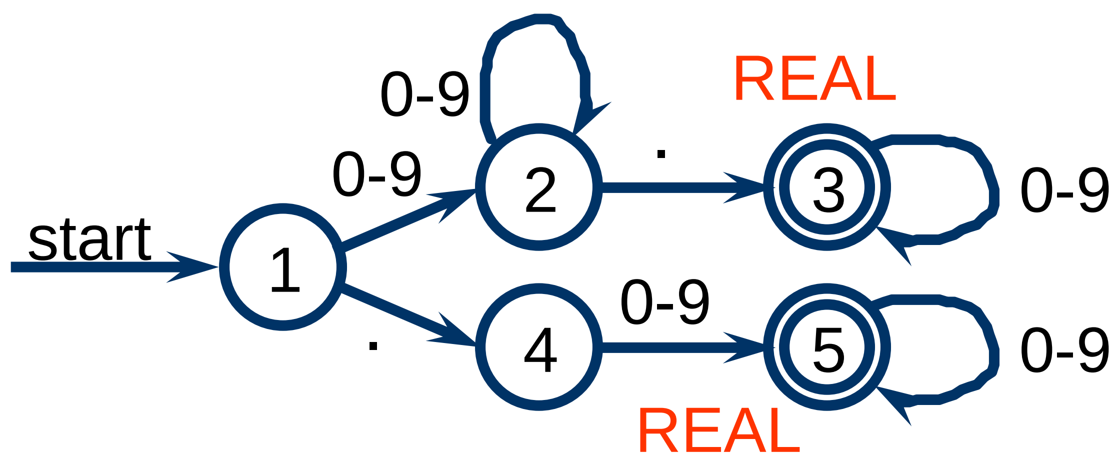
if



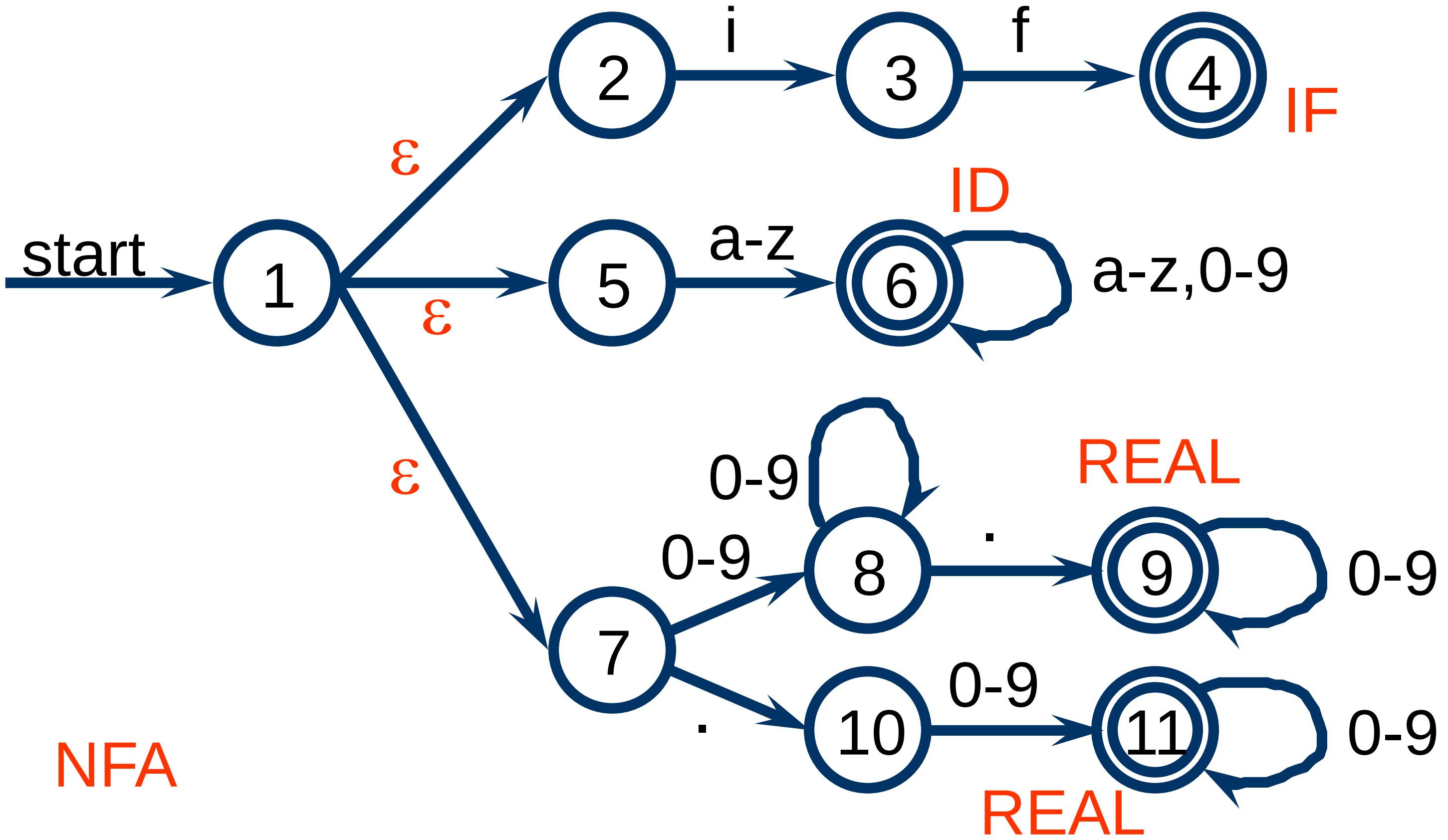
[a-z][a-z0-9]*



([0-9]+“.”[0-9]*)
|
([0-9]*“.”[0-9]+)

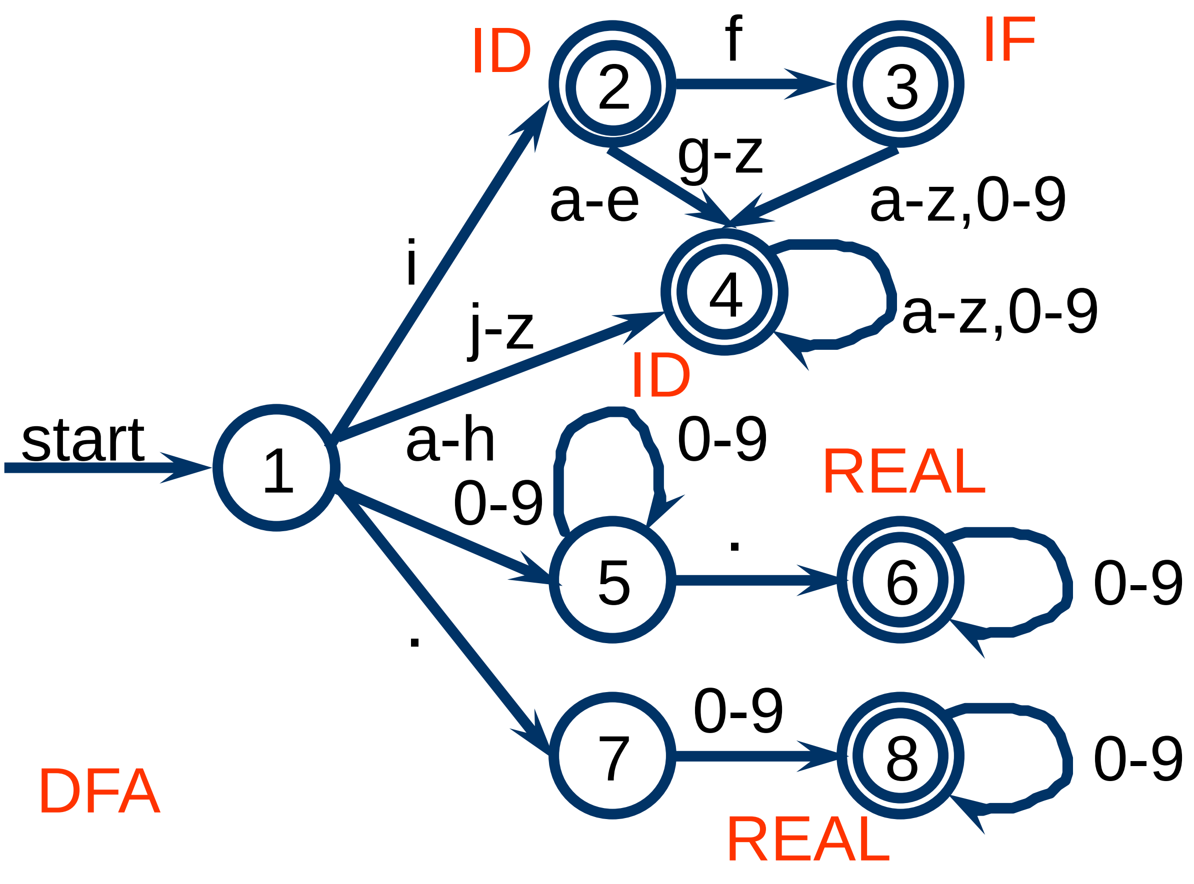


Combined Finite Automata



NFA

Combined Finite Automata

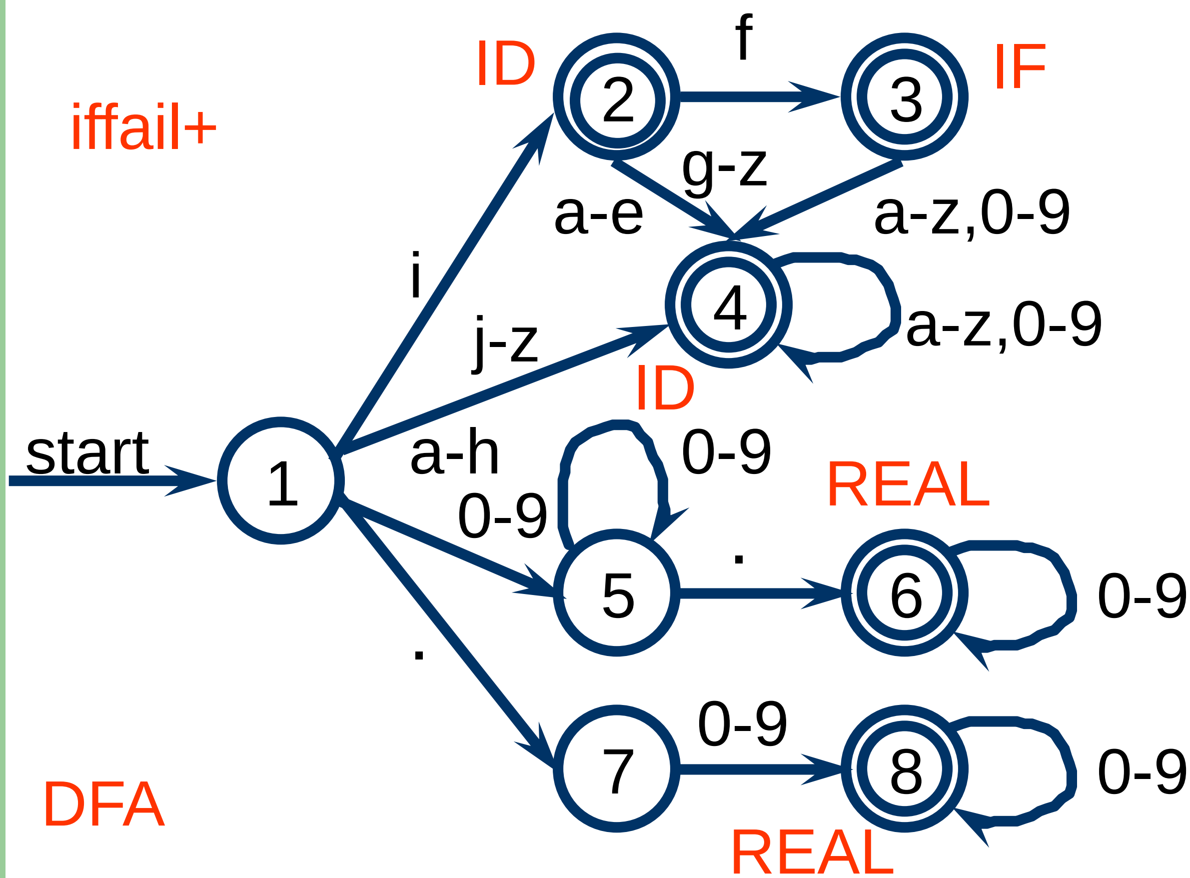


DFA

Recognizing the Longest Match

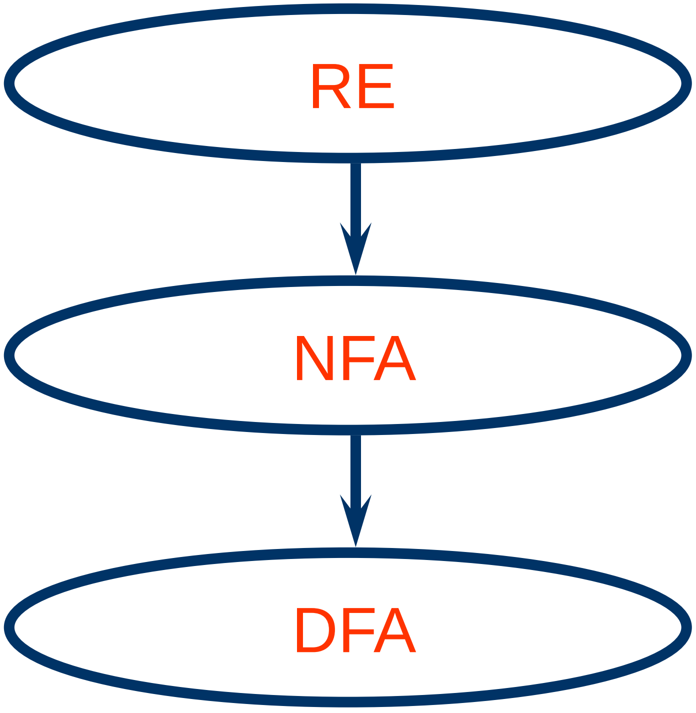
- The automaton must keep track of the longest match seen so far and the position of that match until a dead state is reached
- Use two variables Last-Final (the state number of the most recent final state encountered) and Input-Position-at-Last-Final to remember the last time the automaton was in a final state

An Example



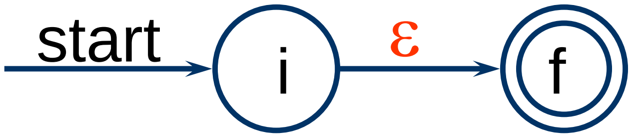
S	C	L	P
	1	0	0
i	2	2	1
f	3	3	2
f	4	4	3
a	4	4	4
i	4	4	5
l	4	4	6
+	?		

Automatic Conversion from RE to FA

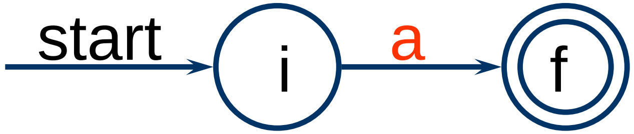


From a RE to an NFA

- Thompson's construction algorithm
  - For ε , construct

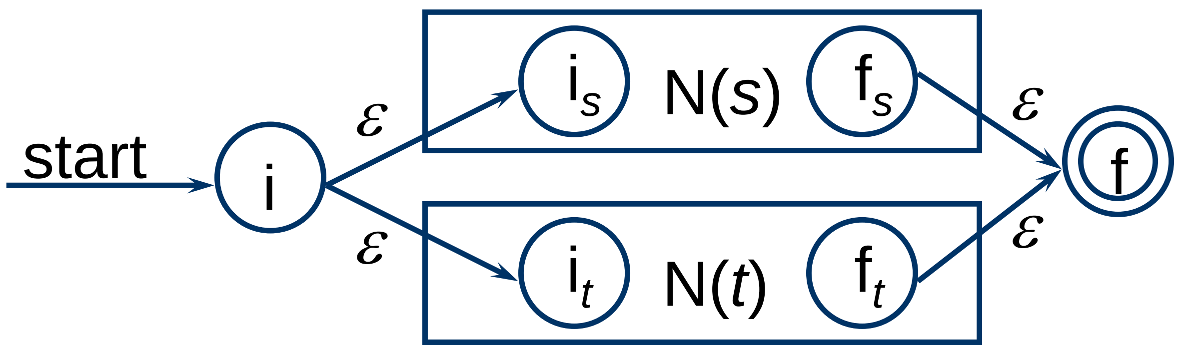


  - For a in alphabet, construct

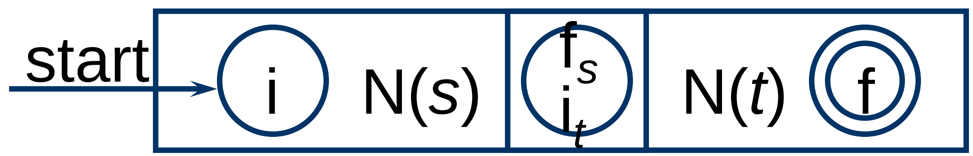


From a RE to an NFA

- Suppose N(*s*) and N(*t*) are NFA for RE *s* and *t*
  - for *s* | *t*, construct

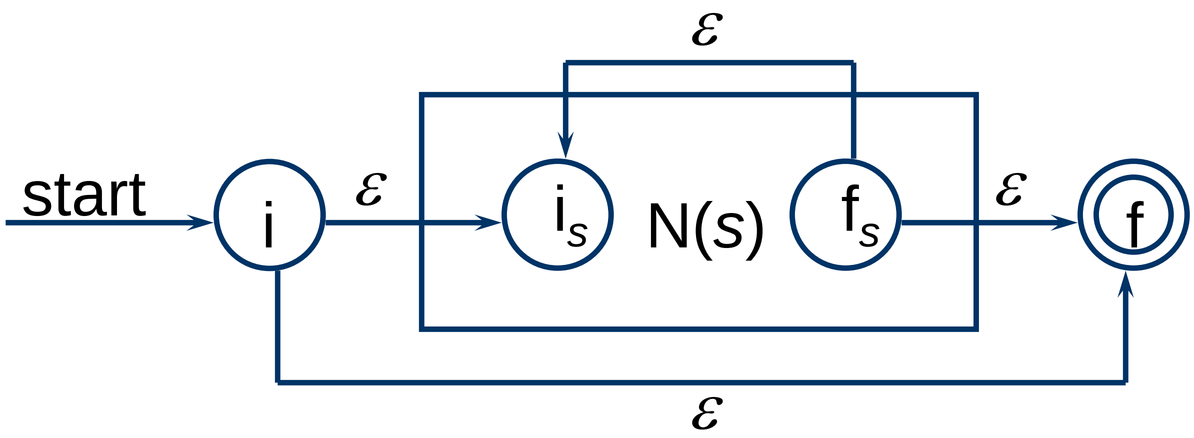


  - for *s t*, construct



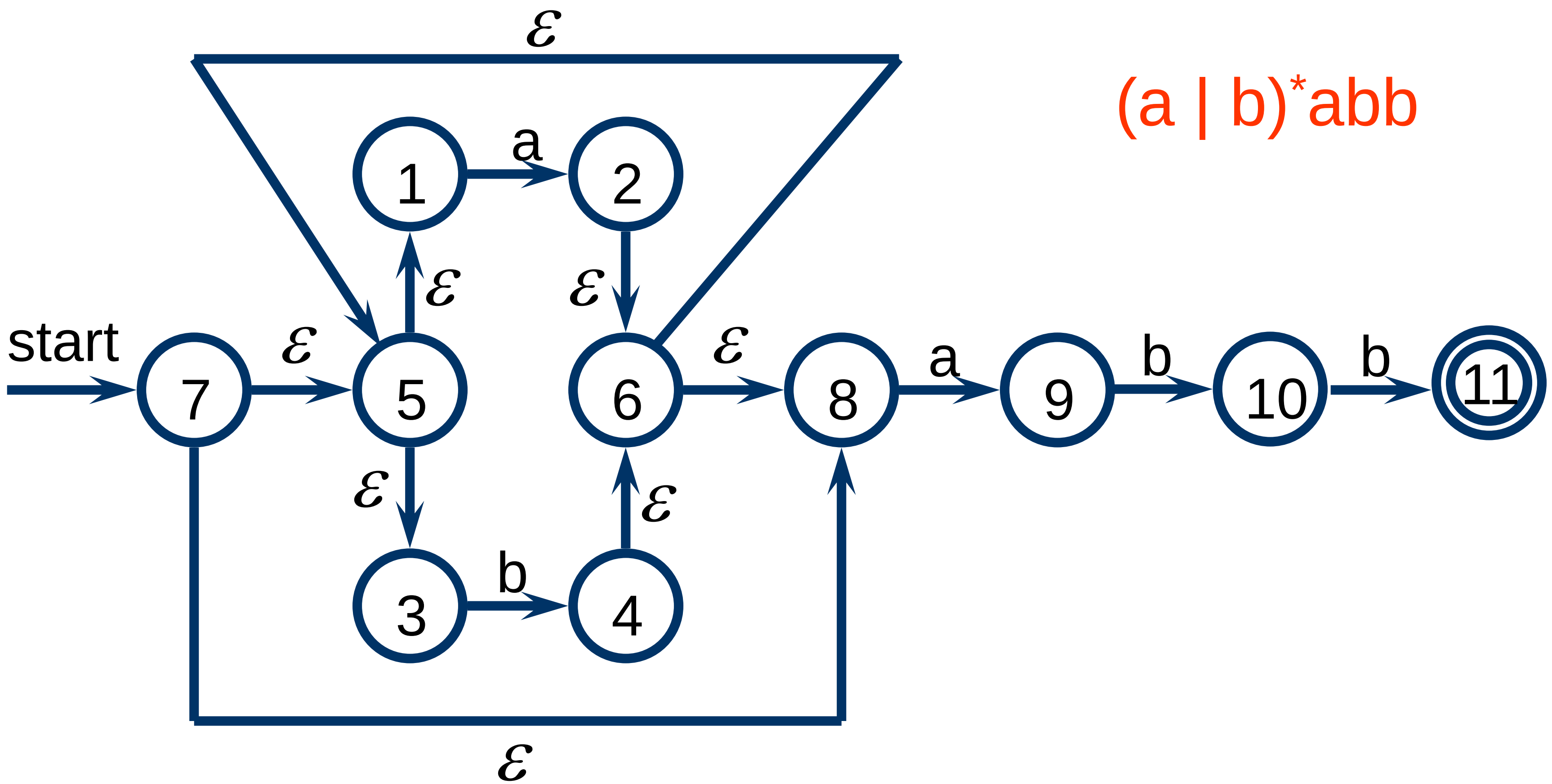
From a RE to an NFA

– for s^*, construct



– for (s), use $N(s)$

An Example



$(a \mid b)^{*}abb$

From an NFA to a DFA

Subset construction Algorithm.

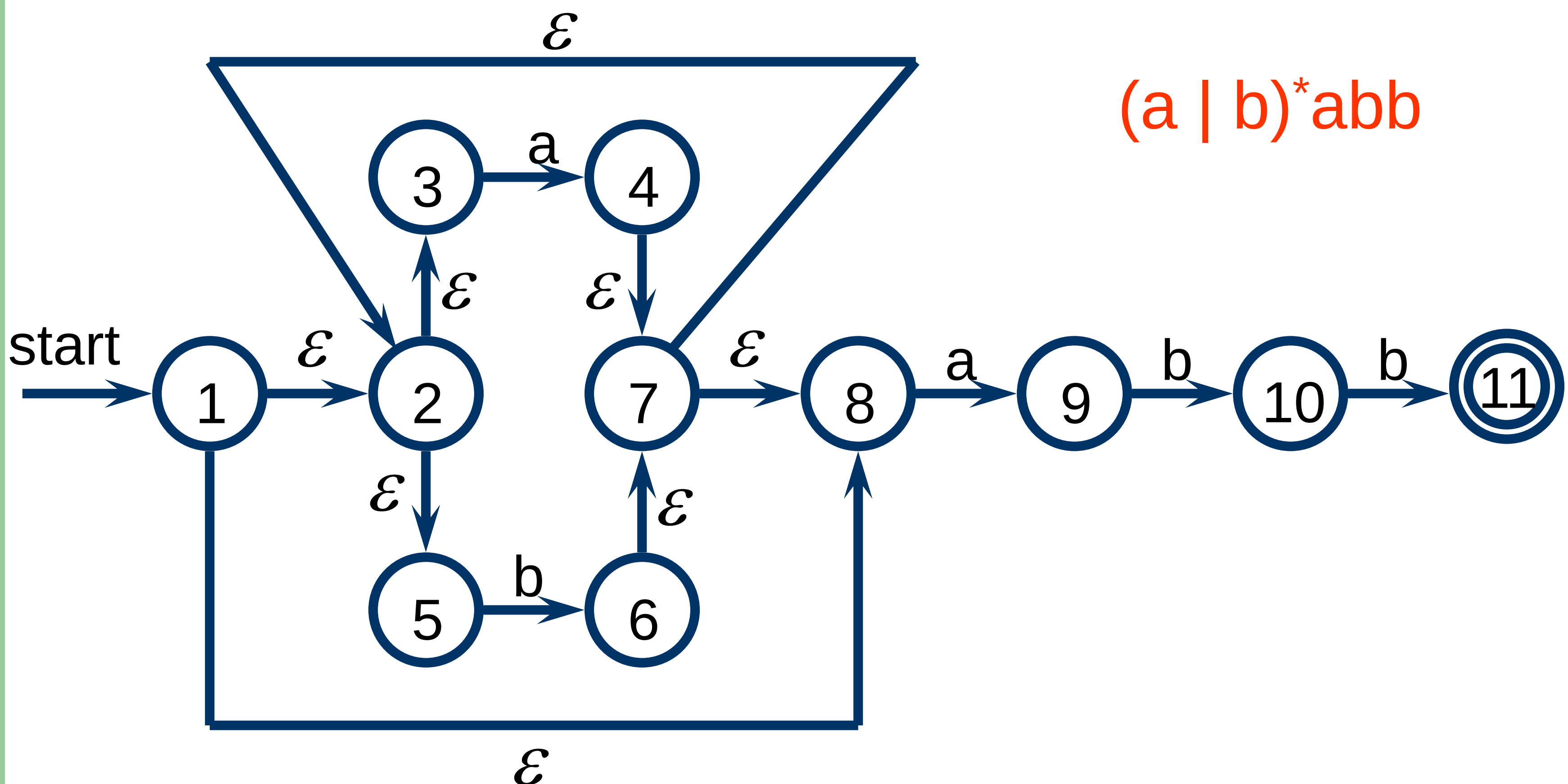
Input: An NFA *N*.

Output: A DFA *D* with states *Dstates* and trasition table *Dtran*.

```
begin
  add ε-closure(s0) as an unmarked state to Dstates;
  while there is an unmarked state T in Dstates do begin
    mark T;
    for each input symbol a do begin
      U := ε-closure(move(T, a));
      if U is not in Dstates then
        add U as an unmarked state to Dstates;
      Dtran[T, a] := U
    end
end.
```

An Example

$(a \mid b)^{*}abb$



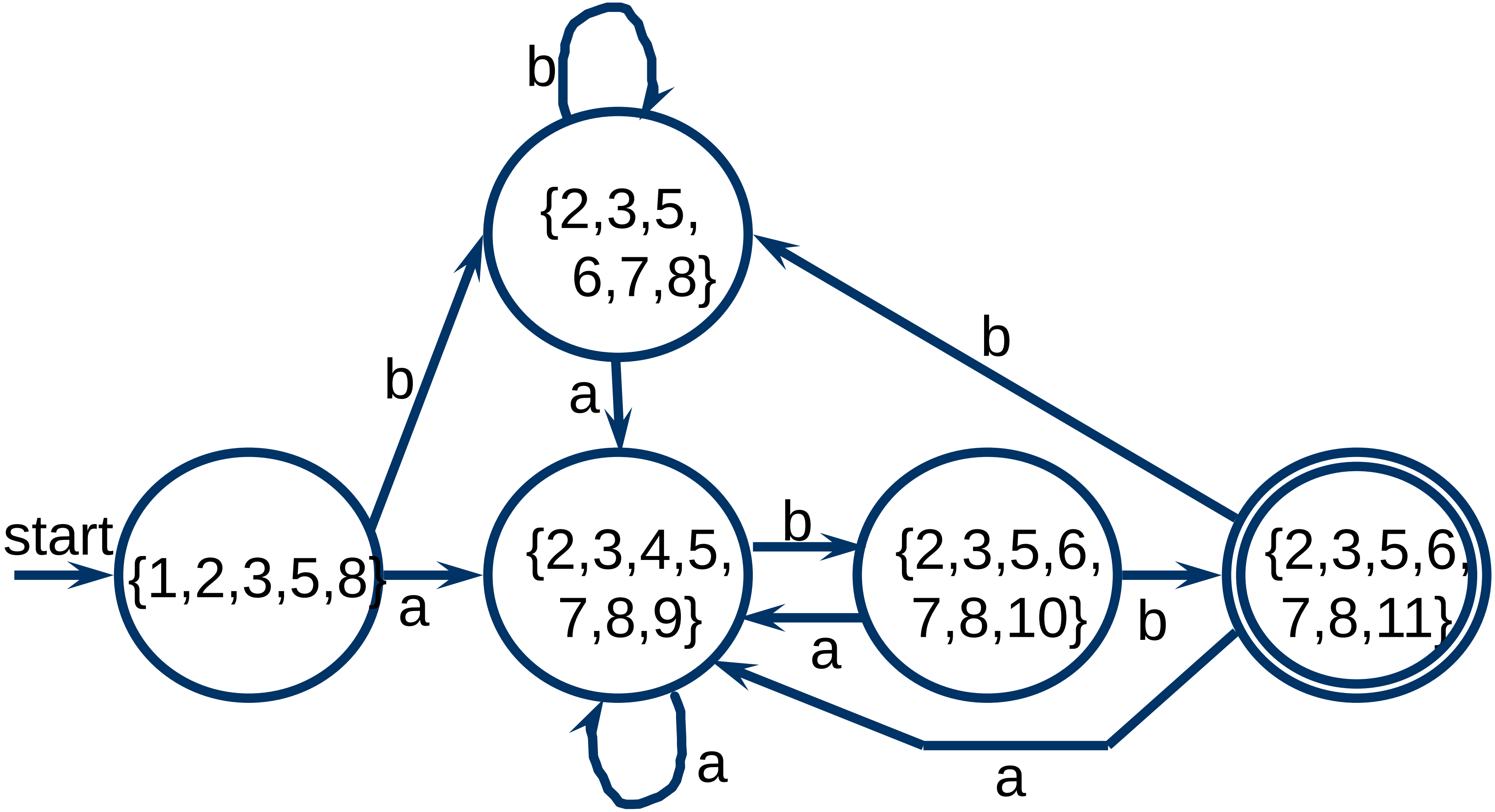
An Example

ε-*closure*({1}) = {1,2,3,5,8} = A
ε-*closure*(*move*(A, *a*))=ε-*closure*({4,9}) = {2,3,4,5,7,8,9} = B
ε-*closure*(*move*(A, *b*))=ε-*closure*({6}) = {2,3,5,6,7,8} = C
ε-*closure*(*move*(B, *a*))=ε-*closure*({4,9}) = B
ε-*closure*(*move*(B, *b*))=ε-*closure*({6,10}) = {2,3,5,6,7,8,10} = D
ε-*closure*(*move*(C, *a*))=ε-*closure*({4,9}) = B
ε-*closure*(*move*(C, *b*))=ε-*closure*({6}) = C
ε-*closure*(*move*(D, *a*))=ε-*closure*({4,9}) = B
ε-*closure*(*move*(D, *b*))=ε-*closure*({6,11}) = {2,3,5,6,7,8,11} = E
ε-*closure*(*move*(E, *a*))=ε-*closure*({4,9}) = B
ε-*closure*(*move*(E, *b*))=ε-*closure*({6}) = C

An Example

State	Input Symbol	
	a	*b*
A = {1,2,3,5,8}	B	C
B = {2,3,4,5,7,8,9}	B	D
C = {2,3,5,6,7,8}	B	C
D = {2,3,5,6,7,8,10}	B	E
E = {2,3,5,6,7,8,11}	B	C

An Example



A Lexer Generator — ANTLR

- ANTLR (ANother Tool for Language Recognition) is a powerful compiler generator for reading, processing, executing, or translating structured text or binary files.
- It's widely used to build languages, tools, and frameworks.

ANTLR Download

- The latest version of ANTLR is 4.7.2, released December 18, 2018. As of 4.7.2, we have a Java, C#, JavaScript, Python2, Python3, Go, C++, Swift targets.
- ANTLR is really two things: a tool that translates your grammar to a parser/lexer in Java and the runtime needed by the generated parsers/lexers.
- The file antlr-4.7.2-complete.jar contains the tool and the runtime for Java.

ANTLR Windows Installation

- Download https://www.antlr.org/download/antlr-4.7.2-complete.jar to C:\JavaLib.
- Add C:\JavaLib\antlr-4.7.2-complete.jar to CLASSPATH, Using System Properties dialog > Environment variables > Create or append to CLASSPATH variable
- Create batch commands for ANTLR Tool, TestRig in dir in PATH
  antlr4.bat: java org.antlr.v4.Tool %*
  grun.bat: java org.antlr.v4.gui.TestRig %*

Grammar Lexicon

- Comments
- Keywords
- Identifiers
- Literals
- Actions

Comments

```
/** This grammar is an example illustrating
 *  the three kinds of comments.
 */
grammar T;
/* a multi-line
   comment
 */
/** This rule matches a declarator */
decl : ID ; // match a variable name
```

Keywords

- The reserved words in ANTLR:
- import, fragment, lexer, parser, grammar, returns, locals, throws, catch, finally, mode, options, tokens.
- Also, although it is not a keyword, do not use the word rule as a rule name.
- Further, do not use any keyword of the target language as a token, label, or rule name.

Identifiers

- Token names or lexer rule names always start with a capital letter.
- Parser rule names always start with a lowercase letter.
- The initial character can be followed by uppercase and lowercase letters, digits, and underscores.

Identifiers

```
/* token names or lexer rule names
ID, LPAREN, RIGHT_CURLY

// parser rule names
expr, simpleDeclarator, d2, header_file
```

Literals

- ANTLR does not distinguish between character and string literals.
- All literal strings one or more characters in length are enclosed in single quotes such as ';', 'if', '>=', and '\''.
- ANTLR understands the usual special escape sequences: '\n', '\r', '\t', '\b', and '\f'.
- Literals can contain Unicode escape sequences of the form \uXXXX, where XXXX is the hexadecimal Unicode character value.

Grammar Structure

grammar *Name*;
options {...}
import ... ;
tokens {...}
channels {...}
rules

Grammar Options

- ANTLR options may be set either within the grammar file using the options syntax or when invoking ANTLR on the command line, using the -D option.
- E.g.,
  options { language = java; }

Grammar imports

- Grammar imports let you break up a grammar into logical and reusable chunks.

```
grammar X;
import Y;
expr : INT | ID;
INT : [0-9]+ ;
```

```
grammar Y;
ID : [a-z]+ ;
```

```
grammar X;
expr : INT | ID;
INT : [0-9]+ ;
ID : [a-z]+ ;
```

Tokens Section

- The purpose of the tokens section is to define token types needed by a grammar for which there is no associated lexical rule.
- The basic syntax is:
  tokens { Token1, ..., TokenN }
- E.g.
  tokens { BEGIN, END, IF, THEN, WHILE }

Lexer Rules

- Lexer rule names must begin with an uppercase letter.
  TokenName :
      alternative1 | ... | alternativeN ;

- You can also define rules that are not tokens but rather aid in the recognition of tokens.
  fragment HelperTokenRule :
      alternative1 | ... | alternativeN ;

An Example

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

Lexer Rule Elements

- ’literal’: Match that character or sequence of characters. E.g., ’while’ or ’=’.
- ’x’..’y’: Match any single character between range x and y, inclusively. E.g., ’a’..’z’.
- .: The dot is a single-character wildcard that matches any single character. E.g.,
  ESC : '\\' . ;

Lexer Rule Elements

- [char set]: Match one of the characters specified in the character set. Interpret x-y as set of characters between range x and y, inclusively. The following escaped characters are interpreted as single special characters: \n, \r, \b, \t, and \f. To get], \, or - you must escape them with \. You can also use Unicode character specifications: \uXXXX.
- [a-z] is identical to 'a'..'z'.

Lexer Rule Elements

- ~x: Match any single character not in the set described by x. Set x can be a single character literal, a range, or a subrule set like ~('x'|'y'|'z') or ~[xyz].

Lexer Rule Elements

- T: Invoke lexer rule T; recursion is allowed in general, but not left recursion. T can be a regular token or fragment rule.
- E.g.,
  ID : LETTER (LETTER | DIGIT)* ;
  fragment LETTER : [a-zA-Z_] ;
  fragment DIGIT : [0-9] ;

Lexer Commands

- To avoid tying a grammar to a particular target language, ANTLR supports lexer commands.
- Lexer commands appear at the end of the outermost alternative of a lexer rule definition.
- A lexer command consists of the -> operator followed by one or more command names that can optionally take parameters:
  TokenName : «alternative» -> command-name
  TokenName : «alternative» -> command-name («identifier or integer»)

Lexer Commands

- A 'skip' command tells the lexer to get another token and throw out the current text.
  WS : [\t]+ -> skip ;
- A ‘channel(x)’ command sends the token type to the x channel. HIDDEN channel is not connected to the parser.
  WS : [\t]+ -> channel(HIDDEN) ;

Nongreedy Lexer Subrules

- Subrules like (...)* and (...)+ are greedy—They consume as much input as possible.
- Constructs like .* consume until the end of the input in the lexer.
- We can make any subrule that has a *, or + suffix nongreedy by adding another ? suffix.
- E.g.,
  COMMENT : '/*' .*? '*/' -> skip ;

Parser Rules

- Parser rule names must begin with a lowercase letter.

  ```
  parserRuleName :
      alternative1 | ... | alternativeN ;
  ```

An Example

```
// File Rose.g4
grammar Rose;
token : (BEGIN | ELSE | … )* ;
BEGIN : ‘begin’ ;
ELSE : ‘else’ ;
  …
```

An Example

```
// edit Rose.g4
> antlr4 Rose.g4
// generate Rose.tokens Rose*.java
> javac Rose*.java
// generate Rose*.class
// edit input_file
> grun Rose token –tree < input_file
(token begin else … )
```