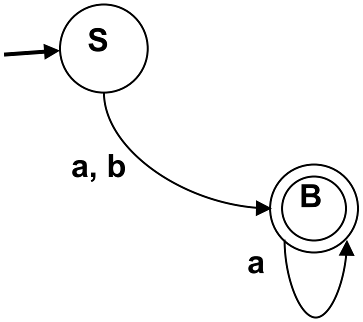
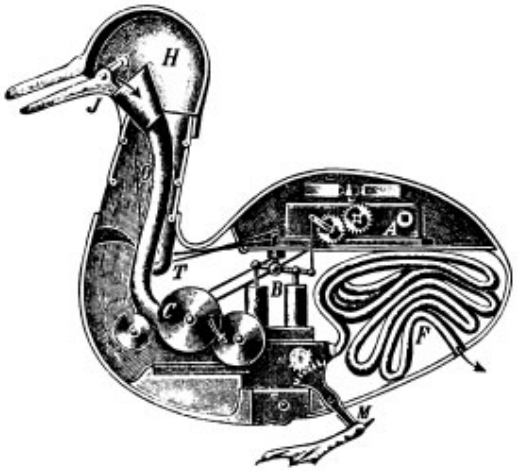
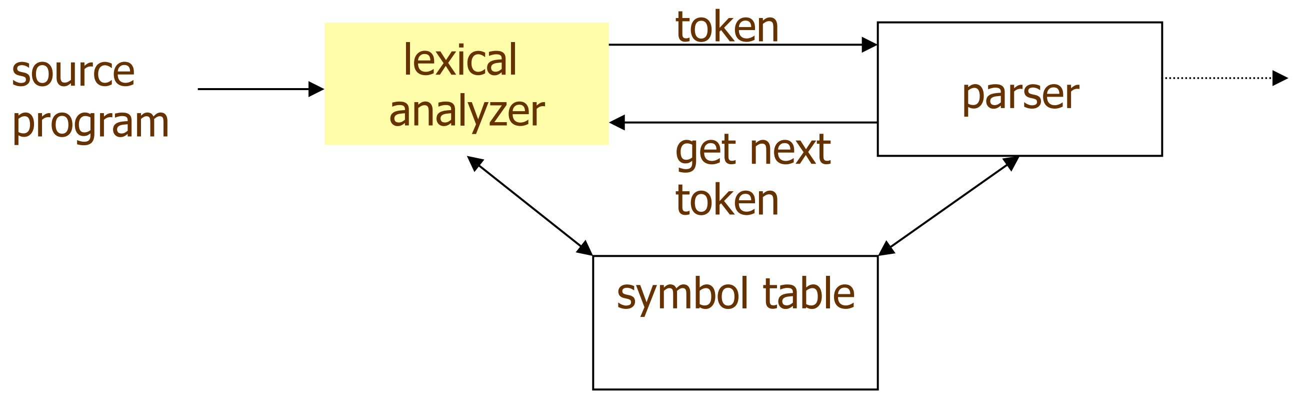
03-60-214 Lexical analysis





Lexical analysis in perspective

- **LEXICAL ANALYZER: Transforms character stream to token stream**
  - Also called scanner, lexer, linear analyzer

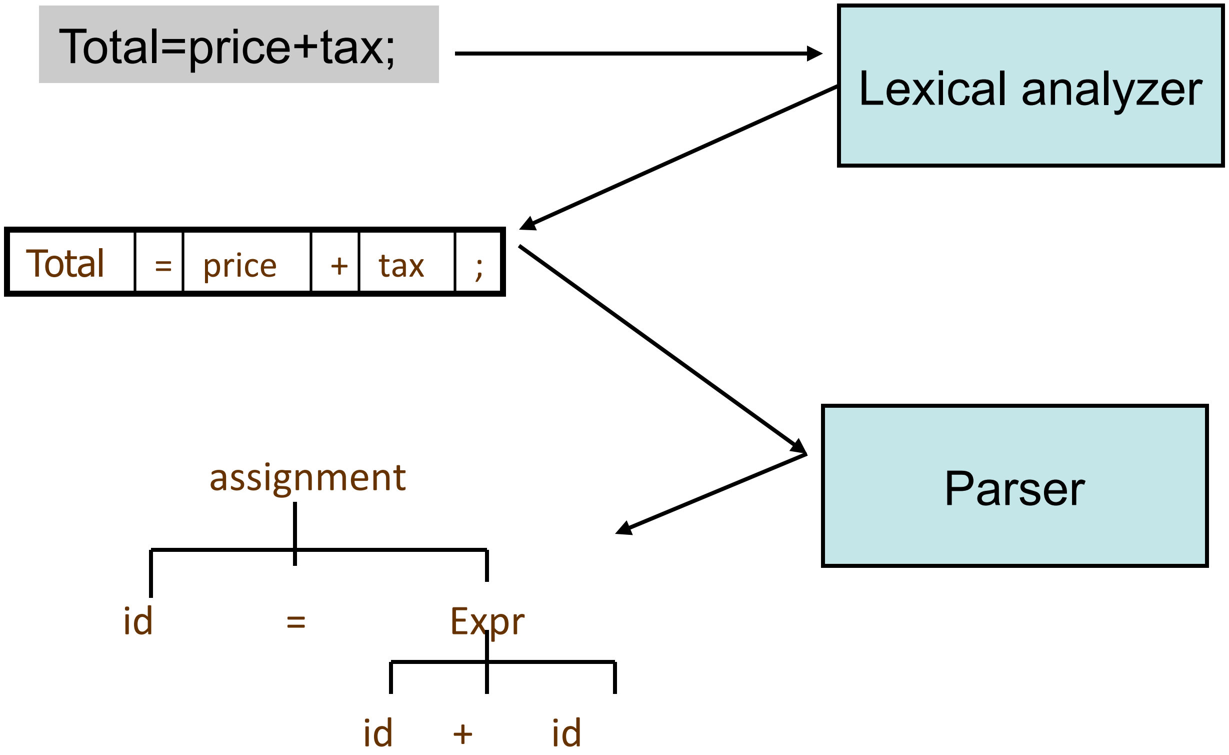


- **LEXICAL ANALYZER**
  - Scan Input
  - Remove White Space, New Line, …
  - Identify Tokens
  - Create Symbol Table
  - Insert Tokens into Symbol Table
  - Generate Errors
  - Send Tokens to Parser

- **PARSER**
  - Perform Syntax Analysis
  - Actions Dictated by Token Order
  - Update Symbol Table Entries
  - Create Abstract Representation of Source
  - Generate Errors

Where we are

Regular expression



Regular expression

Basic terminologies in lexical analysis

- Token
  - A classification for a common set of strings
  - Examples: `if`,`<identifier>`, `<number>` …
- Pattern
  - The rules which characterize the set of strings for a token
  - Recall file and OS wildcards (*.java)
- Lexeme
  - Actual sequence of characters that matches pattern and is classified by a token
  - Identifiers: `if`, `price`, `10.00`, etc…

```
if (price + gst – rebate <= 10.00) gift := false
```

Examples of token, lexeme and pattern

```
if (price + gst – rebate <= 10.00) gift := false
```

Regular expression

Token	lexeme	Informal description of pattern
if	if	If
Lparen	(	(
Identifier	price	String consists of letters and numbers and starts with a letter
operator	+	+
identifier	gst	String consists of letters and numbers and starts with a letter
operator	-	-
identifier	rebate	String consists of letters and numbers and starts with a letter
operator	<=	Less than or equal to
number	10.00	Any numeric constant
rparen	)	)
identifier	gift	String consists of letters and numbers and starts with a letter
operator	:=	Assignment symbol
identifier	false	String consists of letters and numbers and starts with a letter

Regular expression

Regular expression

- Scanner is based on regular expression.
- Remember language is a set of strings.
- Examples of regular expression

– Letter a	a
– Keyword if	if
– All the letters	a|b|c|…|z|A|B|C…|Z
– All the digits	0|1|2|3|4|5|6|7|8|9
– All the Identifiers	letter(letter|digit)*

- Basic operations:

– Set union, e.g.,	`a|b`
– Concatenation, e.g,	`ab`
– Kleene closure, e.g.,	`a*`

Regular expression

- Regular expression: constructing sequences of symbols (strings) from an alphabet.
- Let Σ be an alphabet, r a regular expression then L(r) is the language that is characterized by the rules of r
- Definition of regular expression
  - ε is a regular expression that denotes the language $\{\varepsilon\}$
    - Note that it is not { }
  - If a is in Σ, a is a regular expression that denotes {a}
  - Let r and s be regular expressions with languages L(r) and L(s). Then
    - r | s is a regular expression → $L(r) \cup L(s)$
    - r s is a regular expression →L(r) L(s)
    - r* is a regular expression → (L(r))*
- It is an inductive definition!
- Distinction between regular language and regular expression

Formal language operations

Regular expression

Operation	Notation	Definition	Example L={a, b} M={0,1}
union of L and M	L ∪ M	L ∪ M = {s | s is in L or s is in M}	{a, b, 0, 1}
concatenation of L and M	LM	LM = {st | s is in L and t is in M}	{a0, a1, b0, b1}
Kleene closure of L	L*	L* denotes zero or more concatenations of L	All the strings consists of "a" and "b", plus the empty string. {ε, a, aa, bb, ab, ba, aaa, … }
positive closure	L+	L+ denotes "one or more concatenations of " L	All the strings consists of "a" and "b".

Regular expression

Regular expression example revisited

- Examples of regular expression
  - letter→ a|b|c|...|z|A|B|C...|Z
  - digit→0|1|2|3|4|5|6|7|8|9
  - Identifier → letter(letter|digit)*
- Exercise: why is it a regular expression?

Precedence of operators

- Can the following RE be simplified?

  `(a)| ((b)*(c))`

- `*` is of the highest precedence;
- ts (Concatenation) comes next;
- `|` lowest.

- Example
  - `(a) | ((b)*(c))` is equivalent to `a|b*c`

Regular expression

Properties of regular expressions

<table>
<tr><th>Property</th><th>Description</th></tr>
<tr><td>r|s = s|r</td><td>| is commutative</td></tr>
<tr><td>r|(s|t) = (r|s)|t</td><td>| is associative</td></tr>
<tr><td>(rs)t=r(st)</td><td>Concatenation is associative</td></tr>
<tr><td>r(s|t)=rs | rt
(s|t)r=sr | tr</td><td>Concatenation distributes over |</td></tr>
<tr><td>… …</td><td></td></tr>
</table>

What is why we can write

- either a|b or b|a
- a|b|c, or (a|b)|c
- abc, or (ab)c

Notational shorthand of regular expression

- One or more instance
  - L+ = L L*
  - L* = L+ | ε
  - Example
    - digits→ digit digit*
    - digits→digit+
- Zero or one instance
  - L? = L|ε
  - Example:
    - Optional_fraction→.digits|ε
    - optional_fraction→(.digits)?
- Character classes
  - [abc] = a|b|c
  - [a-z] = a|b|c...|z

RE example

- Strings of length 5
- Suppose the only characters are **a** and **b**
- Solution

  (a|b) (a|b) (a|b) (a|b) (a|b)

- Simplification

  (a|b){5}

RE example

- Strings containing at most one **a**

  b*ab* | b*

  Or

  b*(a|ε) b* | b*

- More concise representation

  b*a?b*

RE for email address

- Valid addresses

  jwho@uwindsor.ca

  j111@cs.uwindsor.ca

  j.111@gmail.com

  …

  letter=[a-zA-Z]

  digit=[0-9]

  w=letter|digit

  w+(.w+)*@w+.w+(.w+)*

RE for odd numbers

Is the following correct?

[13579]+

3

33

43

443

[0-9]*[13579]

Regular expression

More regular expression example

- RE for representing months
  - Example of legal inputs
    - Feb can be represented as 02 or 2
    - November is represented as 11
- First try: (0|1)?[0-9]
  - Matches all legal inputs? Yes
    - 1,2, 11, 12, 01, 02, ...
  - Matches no illegal inputs? No
    - 13, 14, .. etc
- Second try:

  (0|1)? [0-9]
  = (ε|(0|1)) [0-9]
  = [0-9] | (0|1)[0-9]
  = [0-9] | (0 [0-9] | 1[0-9]

  [0-9] | (0 [0-9] | 1[0-2]
  - Matches all legal inputs? Yes
    - 1,2, 11, 12, 01, 02, ...
  - Matches no illegal inputs? No
    - 0, 00

Regular expression

Derive regular expressions

- Solution: [1-9]|(0[1-9])|(1[012])
  - Either 1-9, or 0 followed by 1 to 9, or 1 followed by 0, 1, or 2.
  - Matches all legal inputs
  - Matches no illegal inputs
- More concise solution: 0?[1-9] |1[012]
  - Is it equal to [1-9]|(0[1-9])|(1[012])?

0? [1-9] |1[012]
= (ε|0) [1-9] | 1[012] (by shorthand notation)
= ε[1-9] | 0[1-9] | 1[012] (by distribution over |)
= [1-9] | 0[1-9] | 1[012]

Regular expression example (real number)

- Real number such as 0, 1, 2, 3.14
  - Digit: [0-9]
  - Integer: [0-9]+
  - First try: [0-9]+(.[0-9]+)?
    - Want to allow ".25" as legal input?
  - Second try: [0-9]+ | ([0-9]*.[0-9]+)
- Optional unary minus:

```
-? ([0-9]+ | ([0-9]*.[0-9]+))
```

```
-? (\d+ | (\d*.\d+))
```

Regular expression exercises

- Can the string `baa` be created from the regular expression `a*b*a*b*` ?

- Describe the language (in words) represented by `(a*a)b|ab`.

  a+b

- Write the regular expression that represents:
  - All strings over Σ={a, b} that end in a.

    [ab]*a

    (a|b)*a
  - All strings over Σ={0,1} of even length.

    ((0|1)(0|1))*

    (00|01|10|11)*

Regular expression

Regular grammar and regular expression

- They are equivalent
  - Every regular expression can be expressed by regular grammar
  - Every regular grammar can be expressed by regular expression
  - Different ways to express the same thing
- Why two notations
  - Grammar is a more general concept
  - RE is more concise
- How to translate between them
  - Use automata
  - Will introduce later

Regular expression

What we learnt last class

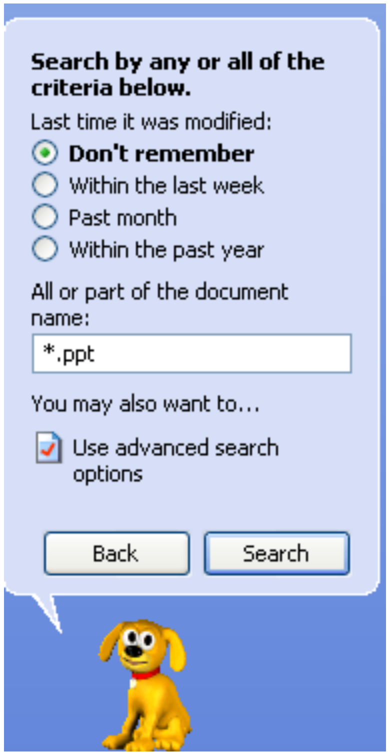
- Definition of regular expression
  - ε is a regular expression that denotes the language $\{\varepsilon\}$
    - Note that it is not { }
  - If a is in Σ, a is a regular expression that denotes {a}
  - Let r and s be regular expressions with languages L(r) and L(s). Then
    - (r) | (s) is a regular expression → $L(r) \cup L(s)$
    - (r)(s) is a regular expression →L(r) L(s)
    - (r)* is a regular expression → (L(r))*

Applications of regular expression

- In Windows
  - In windows you can use RE to search for files or texts in a file
- In unix, there are many RE relevant tools, such as Grep
  - Stands for Global Regular Expressions and Print (or Global Regular Expression and Parser …);
  - Useful UNIX command to find patterns of characters in a text file;
- XML DTD content model
  - <!ELEMENT student (name, (phone|cell)*, address, course+) >

```
<student>
  <name> Jianguo </name>
   <phone> 1234567 </phone>
   <phone> 2345678 </phone>
   <address>  401 sunset ave </address>
    <course> 214 </course>
 </student>
```

- Java Core API has regex package!
- Scanner generation



Search by any or all of the criteria below.

Last time it was modified:

● **Don't remember**
○ Within the last week
○ Past month
○ Within the past year

All or part of the document name:

*.ppt

You may also want to…

☑ Use advanced search options

Regular expression

- RE in XML Schema

```
<xsd:simpleType name="TelephoneNumber">
   <xsd:restriction base="xsd:string">
      <xsd:length value="8"/>
      <xsd:pattern value="\d{3}-\d{4}"/>
   </xsd:restriction>
</xsd:simpleType>
```

Regular expressions used in Scanner, String etc

- A sample problem
  - Develop a program that, given as input three points P1, P2, and P3 on the cartesian coordinate plane, reports whether P3 lies on the line containing P1 and P2.
  - In order to input the three points, the user enters six integers x1, y1, x2, y2, x3, and y3. The three points are P1 = (x1, y1), P2 = (x2, y2), and P3 = (x3, y3).
  - The program should repeat until the user's input is such that P1 and P2 are the same point.
- Sample input
  - Enter x1: 0
  - Enter y1: 0
  - Enter x2: 2
  - Enter y2: 5
  - Enter x3: 1
  - Enter y3: 3
- Output
  - The point (1, 3) IS NOT on the line constructed from points (2, 5) and (0, 0).

Regular expression in Java

How to read and process the input

- First Try

```
Scanner sc=new Scanner(System.in);
int x1=sc.nextInt();

java.util.InputMismatchException
```

- We want to capture the values for x and y, and discard everything else

- Describe everything else as delimiters

```
Scanner sc=new Scanner(System.in).useDelimiter("…");
```

- The delimiters can be any regular expression

Sample input

```
Enter x1: 2
Enter y1: 8
Enter x2: 0
Enter y2: 0
Enter x3: 1
Enter y3: 4
```

useDelimiter in Scanner class

- useDelimiter(“Enter x1:”)
  - This will throw away “Enter x1:” only. To discard “Enter x2:” as well, you may want to add the following
- useDelimiter(“Enter x1:| Enter x2:”)
  - Vertical bar means “OR”—either “`Enter x1:`” OR “`Enter x2:`”
  - It is called Regular expression;
  - Now you know how to expand to the case for x3--
- useDelimiter(“Enter x1:| Enter x2:| Enter x3”)
  - We can simplified the above using other notations in REGUALR EXPRESSION.

Use regular expression to capture the input

- useDelimiter(“Enter x\\d”)
  - Where \\d means any digit. Now how to read in the values for y axis?
- useDelimiter(“Enter \\w\\d”)
  - \\w means any letter or digit
- useDelimiter(“Enter \\w{2}”)
  - What if there are leading and trailing spaces around the sample input?
- useDelimiter(“(|\\t)Enter \\w{2}(|\\t)”)
- useDelimiter(“\\sEnter \\w{2}\\s”)
  - \\s stands for all kinds of white space.
- useDelimiter(\\s*Enter \\w{2}:\\s*)
  - * means that zero or more spaces can occur.

Code fragment to read data

```
//read from file for testing purpose
Scanner sc=new Scanner(new File("online.txt")).
        useDelimiter("\\s*Enter \\w{2}:\\s*");
int x1=sc.nextInt();
int y1=sc.nextInt();
int x2=sc.nextInt();
int y2=sc.nextInt();
int x3=sc.nextInt();
int y3=sc.nextInt();
```

Regular expression in Java

Regex package in Java

- Java has regular package java.util.regex,

- A simple example:
  - Pick out the valid dates in a string
  - E.g. in the string “final exam 2008-04-22, or 2008-4-22, but not 2008-22-04”
  - Valid dates: 2008-04-22, 2008-4-22
- First we need to write the regular expressions.

```
\d{4}-(0?[1-9]|1[012])-\d{2}
```

Regular expression in Java

Regex package

- First, you must compile the pattern

```
import java.util.regex.*;
Pattern p = Pattern.compile(“\\d{4}-(0?[1-9]|1[012])-\
  \d{2}");
```

  - Note that in java you need to write \\d instead of \d
- Next, you must create a matcher for a specific piece of text by sending a message to your pattern
  - Matcher m = p.matcher(“…your text goes here….");
- Points to notice:
  - Pattern and Matcher are both in java.util.regex
  - Neither Pattern nor Matcher has a public constructor; you create these by using methods in the Pattern class
  - The matcher contains information about both the pattern to use and the text to which it will be applied

Regex in java

- Now that we have a matcher m,
  - `m.matches()` returns true if the pattern matches the entire text string, and false otherwise
  - `m.lookingAt()` returns true if the pattern matches at the beginning of the text string, and false otherwise
  - `m.find()` returns true if the pattern matches any part of the text string, and false otherwise
    - If called again, m.find() will start searching from where the last match was found
    - m.find() will return true for as many matches as there are in the string; after that, it will return false
    - When m.find() returns false, matcher m will be reset to the beginning of the text string (and may be used again)

Regular expression in java

Regex example

```
import java.util.regex.*;
public class RegexTest {
public static void main(String args[]) {
    String pattern = "\\d{4}-(0?[1-9]|1[012])-\\d{2}";
    String text = "final exam 2008-04-22, or 2008-4-22, but not
       2008-22-04";
    Pattern p = Pattern.compile(pattern);
    Matcher m = p.matcher(text);
    while (m.find()) {
       System.out.println("valid date:"+text.substring(m.start(),  m.end()));
    }
    }
}
```

Printout:

```
– valid date:2008-04-22
– valid date:2008-4-22
```

More shorthand notation in specific tools, like regex package in Java

Regular expression in java

- Different software tools have slightly different notations (e.g. regex, grep, JLEX);
- Shorthand notations from regex package

.	any one character except a line terminator
\d	a digit: [0-9]
\D	a non-digit: [^0-9]
\s	a white space character: [\t\n\r]
\S	a non-whitespace character: [^\s]
\w	a word character: [a-zA-Z_0-9]
\W	a non-word character: [^\w]

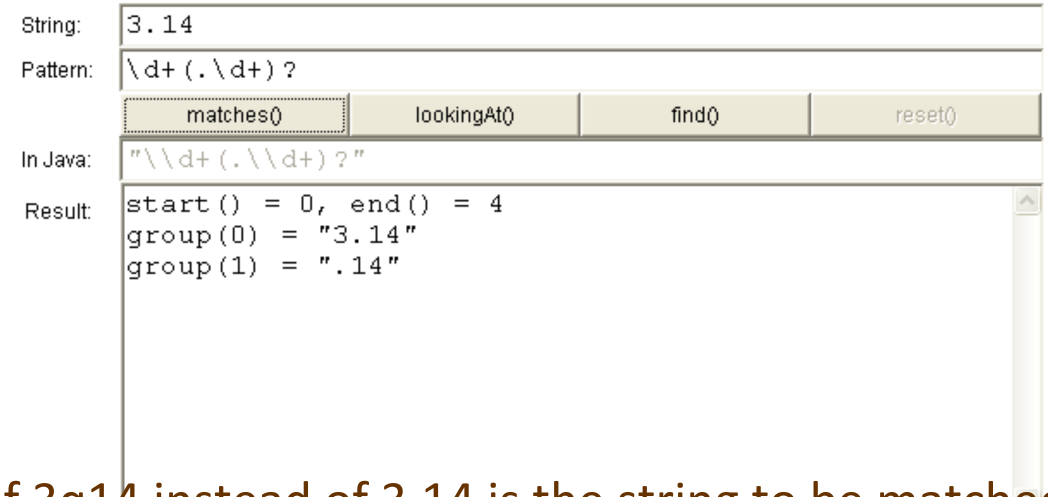
- Get familiar with regular expression using the regexTester Applet.

- Note that String class since Java1.4 provides similar methods for regular expression

Regular expression

Try RegexTester

- Running at course web site as an applet;
  - http://cs.uwindsor.ca/~jlu/214/regex_tester.htm
- Write regular expressions and try the match(), find() methods;



```
String:   3.14
Pattern:  \d+(.\d+)?
In Java:  "\\d+(.\\d+)?"
Result:   start() = 0, end() = 4
          group(0) = "3.14"
          group(1) = ".14"
```

- What if 3q14 instead of 3.14 is the string to be matched? Why?
- groups are numbered by counting their opening parentheses from left to right.
  - ((A)(B(C))) has four groups:
  - **1** ((A)(B(C)))
  - **2** (A)
  - **3** (B(C))
  - **4** (C)

Practice regular expression using grep

Use grep to search for certain pattern in html files;

- Search for Canadian postal code in a text file;
- Search for Ontario car plate number in a text file.

- use tcsh. Type
  - % tcsh
- Prepare text file, say “test”, that consists of sample postal code etc.
- Type
  - grep ‘[a-z][0-9][a-z] [0-9][a-z][0-9]’ test

  - grep –i ‘[a-z][0-9][a-z] [0-9][a-z][0-9]’ test

Regular expression in unix

Practice the following grep commands:

```
grep 'cat' grepTest
```

-you will find both "cat" and "vacation”

```
grep '\<cat\>' grepTest
```

--word boundary

```
grep -i '\<cat\>' grepTest
```

- ignore the case

```
grep '\<ega\.att\.com\>' grepTest
```

-meta character

```
grep '"[^"]*"' grepTest
```

--find quoted string

```
egrep '[a-z][0-9][a-z] ?[0-9][a-z][0-9]' grepTest
```

- find postal code, only if it is in lower case

```
egrep -i '[a-z][0-9][a-z] ?[0-9][a-z][0-9]' grepTest
```

- ignore the case

```
grep 'q[^u]' grepTest
```

--won't find "iraq", but will find "iraqi"

```
grep '^cat' grepTest
```

find only lines start with cat

- egrep is similar to grep, but some patterns only work in egrep.

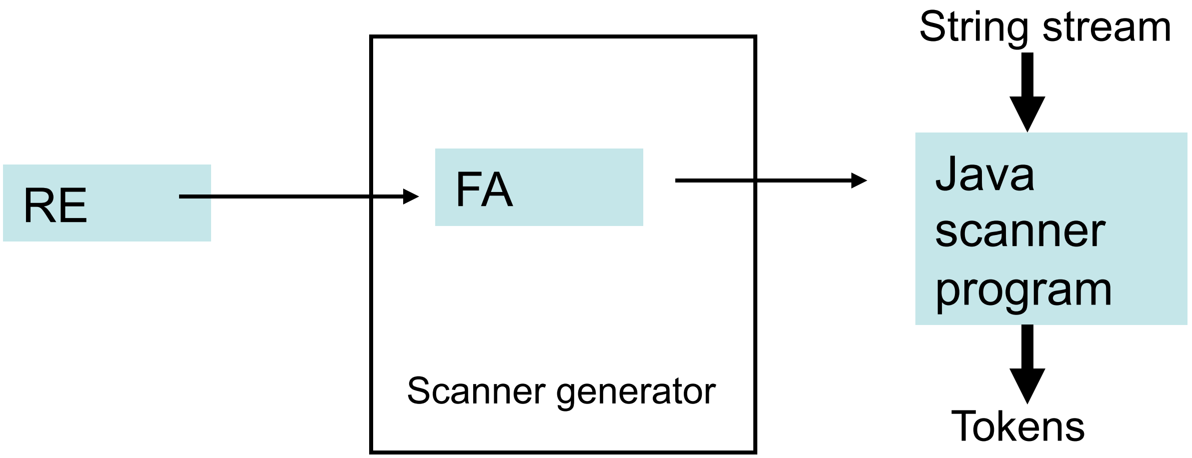
```
megawatt computing
ega.att.com
Line with uppercase Cat and Dog.
214 cat vacation Cat Dog
vacation only
capital Cat
the cat likes the dog
"quoted string "
iraq
iraqi
Qantas
ADBB 123
ADBB12
N9B 3P5
N9B3P5
01-16-2004
14-14-2004
```

Unix machine account

- Apply for a unix account:
  - Write to accounts@cs.uwindsor.ca
- Access unix machines at home:
  - You need to use SSH;
  - One place to download:
    - www.uwindsor.ca/its --> services/downloads

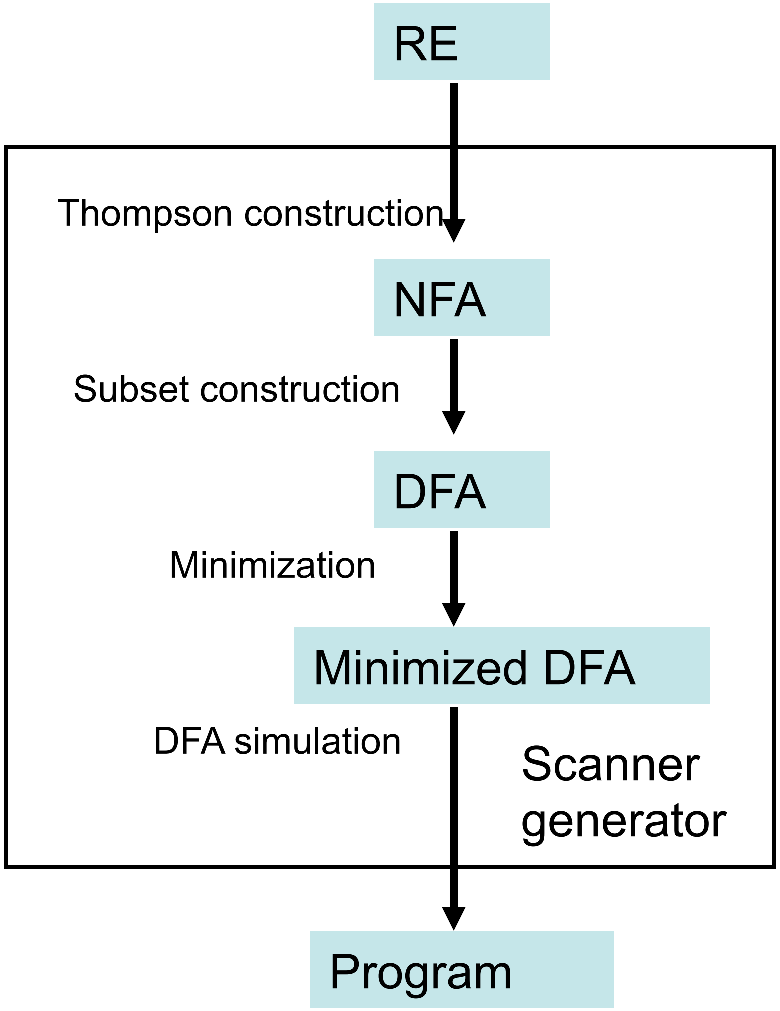
RE and Finite state Automaton (FA)

- Regular expression is a declarative way to describe the tokens
  - It describes *what* is a token, but not *how* to recognize the token.
- FA is used to describe *how* the token is recognized
  - FA is easy to be simulated by computer programs;
- There is a 1-1 correspondence between FA and regular expression
  - Scanner generator (such as JLex) bridges the gap between regular expression and FA.

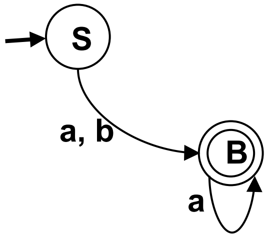


Inside scanner generator

- Main components of scanner generation
  - RE to NFA
  - NFA to DFA
  - Minimization
  - DFA simulation

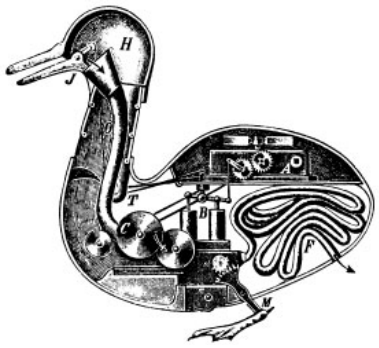


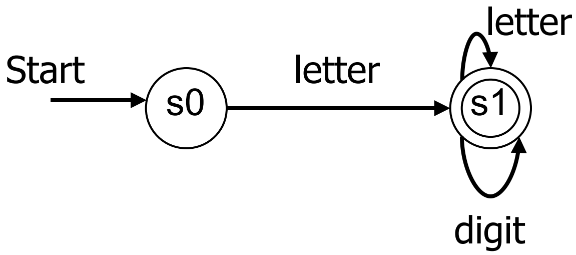
Finite automata



Automata

- FA also called Finite State Machine (FSM)
  - Abstract model of a computing entity;
  - Decides whether to accept or reject a string.
- Two types of FA:
  - Non-deterministic (NFA): Has more than one alternative action for the same input symbol.
  - Deterministic (DFA): Has at most one action for a given input symbol.
- Example: how do we write a program to recognize java identifiers?



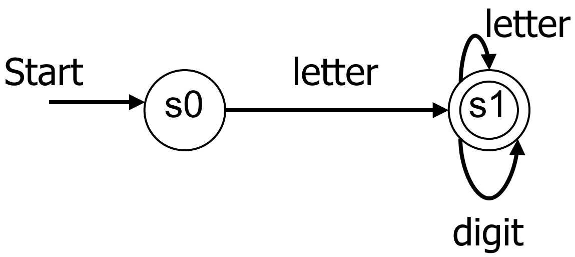


```
S0:
          if (getChar() is letter) goto S1;
S1:    if (getChar() is letter or digit) goto S1;
```

Automaton: a mechanism that is relatively self-operating -- webster

Non-deterministic Finite Automata (FA)

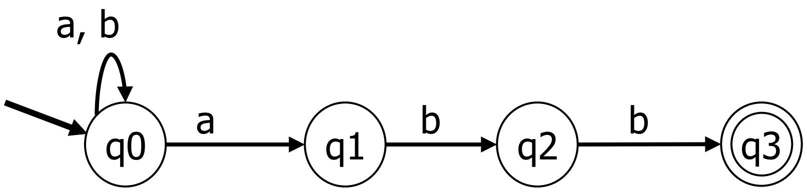
- NFA (Non-deterministic Finite Automaton) is a 5-tuple (S, Σ, δ, S0, F):
  - **S**: a set of states;
  - **Σ**: the symbols of the input alphabet;
  - **δ** : a transition function;
    - move(state, symbol) → a set of states
  - **S0**: s0 $\in$S, the start state;
  - **F**: F $\subseteq$ S, a set of final or accepting states.
- Non-deterministic -- a state and symbol pair can be mapped to a set of states.
  - It is deterministic if the result of transition consists of only one state.
- Finite—the number of states is finite.



Transition Diagram

- FA can be represented using transition diagram.
- Corresponding to FA definition, a transition diagram has:
  - **States** : Represented by circles;
  - **Σ**: Alphabet, represented by labels on edges;
  - **Moves** : Represented by labeled directed edges between states. The label is the input symbol;
  - **Start State** : arrow head;
  - **Final State** (s) : represented by double circles.

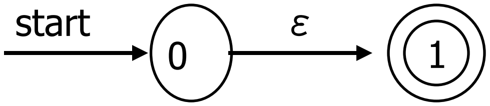
- Example transition diagram to recognize `(a|b)*abb`



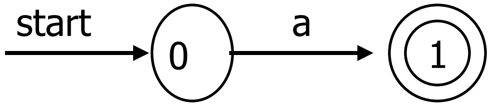
Simple examples of FA

Automata

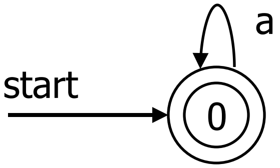
- Epsilon



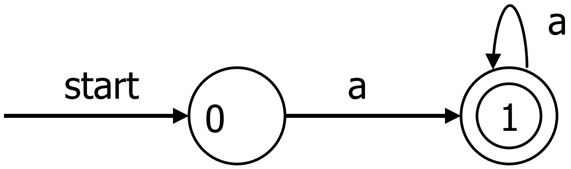
- a



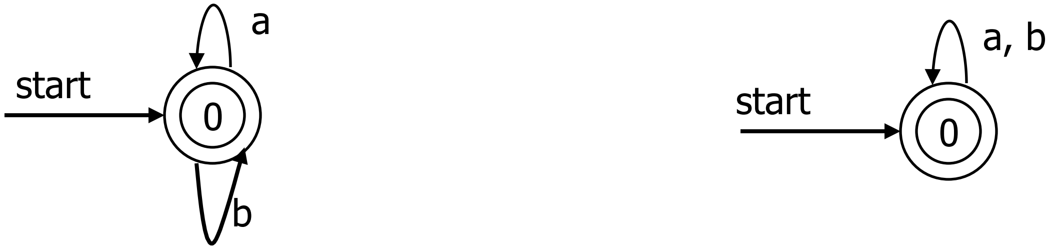
- a*



- a+



- (a|b)*



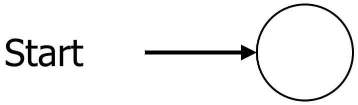
Procedures of defining a DFA/NFA

- Define input alphabet and initial state
- Draw the transition diagram
- Check
  - all states have out-going arcs labeled with all the input symbols (DFA).
  - Are there any missing final states?
  - Are there any duplicate states?
  - all strings in the language can be accepted.
  - all strings not in the language can not be accepted.
- Name all the states
- Define (S, Σ, δ, q_0, F)

Example of constructing a FA

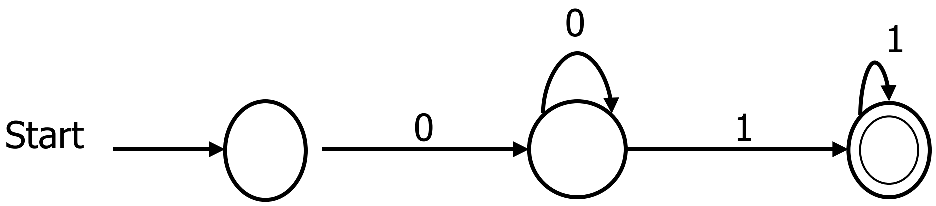
Automata

- Construct a DFA that accepts a language L over Σ = {0, 1} such that L is the set of all strings with any number of "0"s followed by any number of "1"s.
- Regular expression: 0*1*
- Σ = {0, 1}
- Draw initial state of the transition diagram

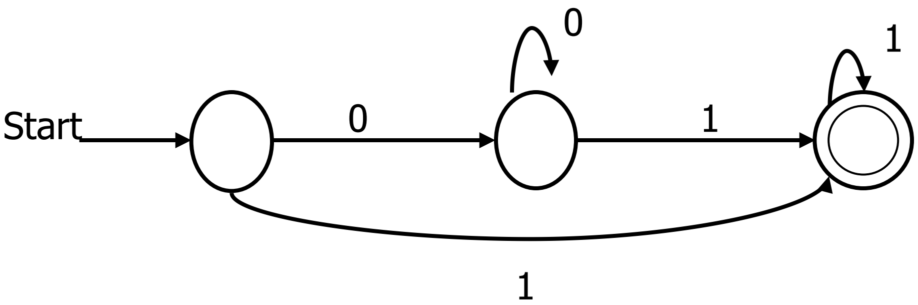


Example of constructing a FA (cont.)

- Draft the transition diagram



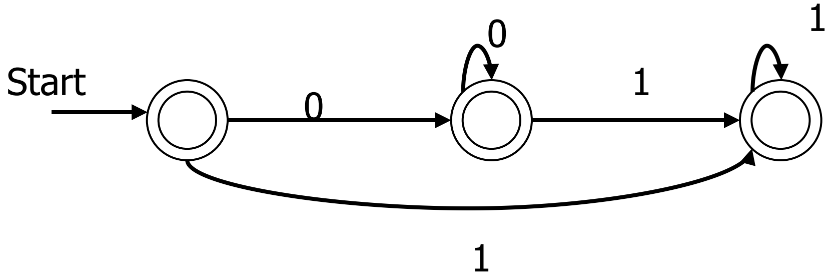
- Is “111” accepted?
- The leftmost state has missed an arc with input “1”



- Is “00” accepted?

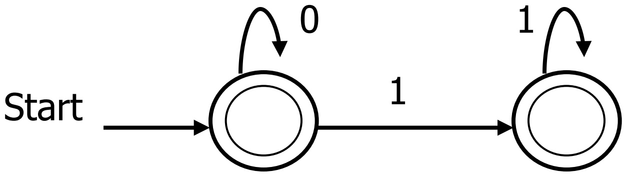
Example of constructing a FA (cont.)

- Is “00” accepted?
- The leftmost two states are also final states
  - First state from the left: ε is also accepted
  - Second state from the left:
    strings with “0”s only are also accepted

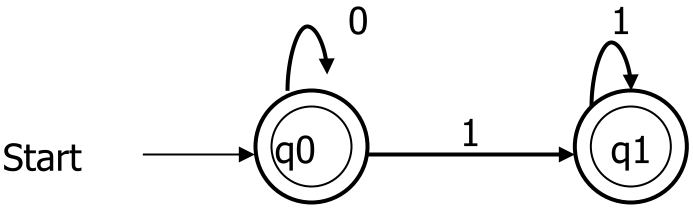


Example of constructing a FA (cont.)

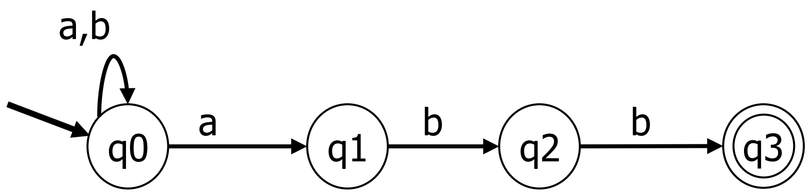
- The leftmost two states are duplicate
  - their arcs point to the same states with the same symbols



- Check that they are correct
  - All strings in the language can be accepted
    - ε is accepted
    - strings with “0”s / “1”s only are accepted
  - All strings not belonged to the language can not be accepted
- Name all the states



How does FA work



- NFA definition for (a|b)*abb
  - S = {q0, q1, q2, q3 }
  - Σ = { a, b }
  - Transitions: move(q0,a)={q0, q1}, move(q0,b)={q0}, ...
  - s0 = q0
  - F = { q3 }
- Transition diagram representation
  - Non-determinism:
    - exiting from one state there are multiple edges labeled with same symbol, or
    - There are epsilon edges.
  - How does FA work? Input: ababb

move(q0, a) = q1
move(q1, b) = q2
move(q2, a) = ? (undefined)

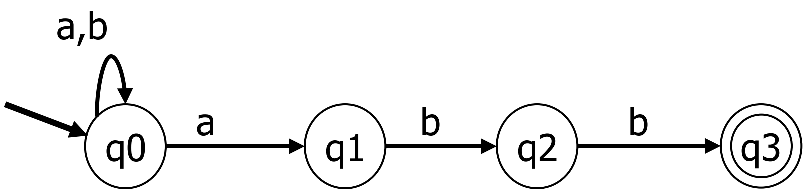
REJECT !

move(q0, a) = q0
move(q0, b) = q0
move(q0, a) = q1
move(q1, b) = q2
move(q2, b) = q3

ACCEPT !

Automata

FA for (a|b)*abb



- What does it mean that a string is accepted by a FA?
  - An FA accepts an input string *x* iff there is a path from the start state to a final state, such that the edge labels along this path spell out *x*;
- A path for "aabb": $q0 \rightarrow^{a} q0 \rightarrow^{a} q1 \rightarrow^{b} q2 \rightarrow^{b} q3$
- Is "aab" acceptable?

  $q0 \rightarrow^{a} q0 \rightarrow^{a} q1 \rightarrow^{b} q2$

  $q0 \rightarrow^{a} q0 \rightarrow^{a} q0 \rightarrow^{b} q0$

  - The answer is no;
  - Final state must be reached;
  - In general, there could be several paths.
- Is "aabbb" acceptable?

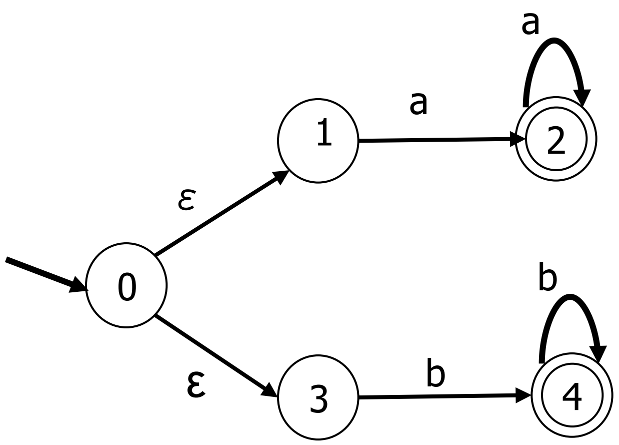
  $q0 \rightarrow^{a} q0 \rightarrow^{a} q1 \rightarrow^{b} q2 \rightarrow^{b} q3$

  - The answer is no.
  - Labels on the path must spell out the entire string.

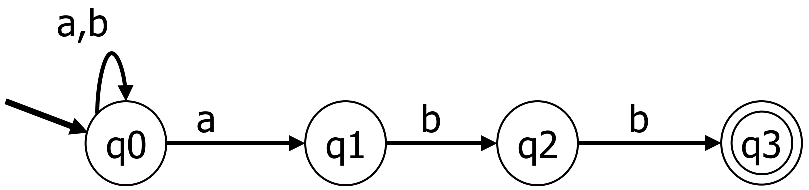
Example of NFA with epsilon symbol

Automata

- NFA accepting `aa*|bb*`
  - Is “aaab” acceptable?
  - Is “aaa” acceptable?

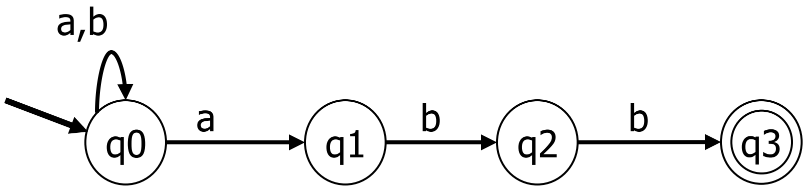


Simulating an NFA



- Keep track of a set of states, initially the start state and everything reachable by ε-moves.
- For each character in the input:
  - Maintain a set of next states, initially empty.
- For each current state:
  - Follow all transitions labeled with the current letter.
  - Add these states to the set of new states.
- Add every state reachable by an ε-move to the set of next states.
- Complexity: O(mn^2) for strings of length m and automata with n states

Transition table



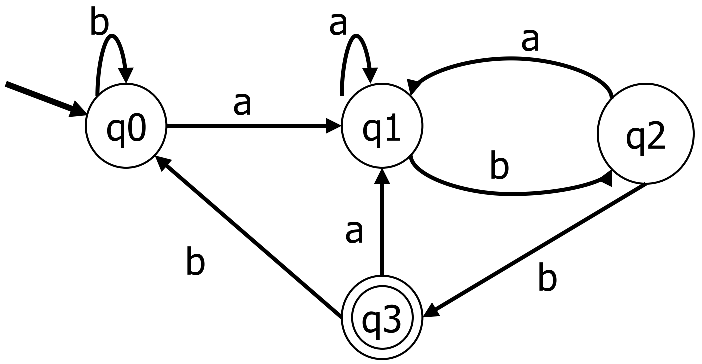
- It is one of the ways to implement the transition function
  - There is a row for each state;
  - There is a column for each symbol;
  - Entry in (state s, symbol a) is the set of states can be reached from state s on input a.
- Nondeterministic:
  - The entries are sets instead of a single state

States	Input	
	a	b
>q0	{q0, q1}	{q0}
q1		{q2}
q2		{q3}
*q3		

Automata

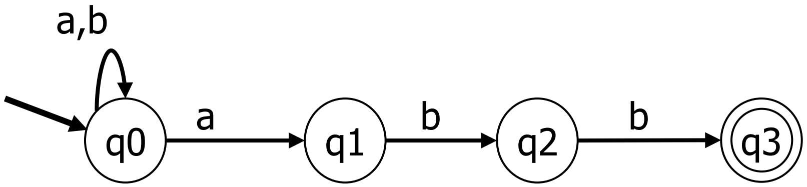
DFA (Deterministic Finite Automaton)

- A special case of NFA
  - The transition function maps the pair (state, symbol) to one state.
    - When represented by transition diagram, for each state *S* and symbol *a,* there is at most one edge labeled *a* leaving *S;*
    - When represented transition table, each entry in the table is a single state.
  - There is no ε-transition
- Example: DFA for (a|b)*abb



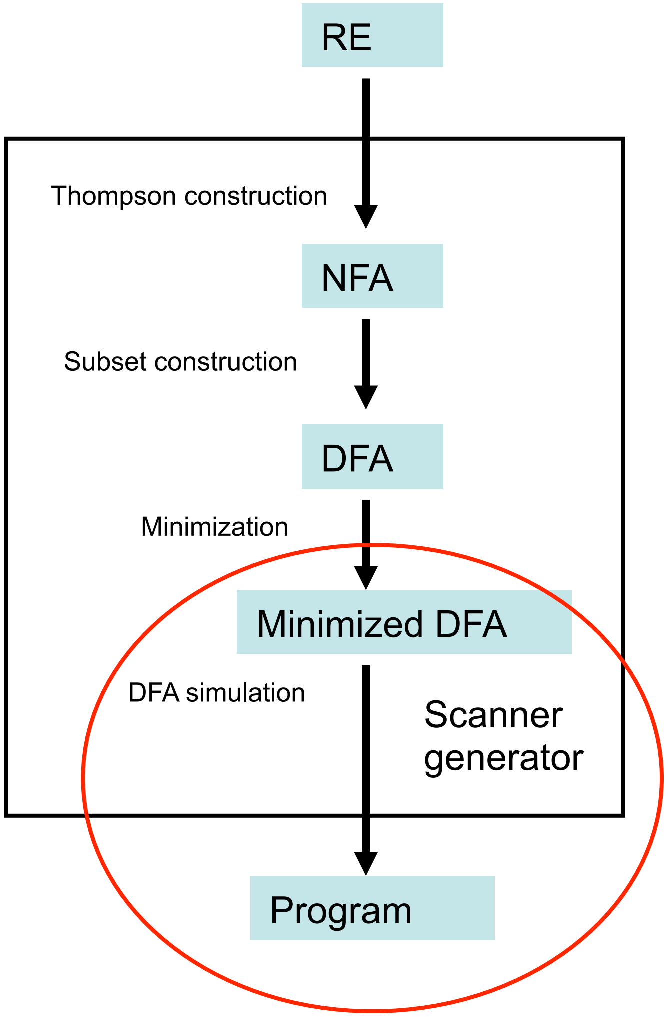
States	Input	
	a	b
Q0	Q1	q0
Q1	Q1	Q2
Q2	Q1	Q3
Q3	Q1	Q0

- Recall the NFA:



DFA to program

- NFA is more concise, but not easy to implement;
- In DFA, since transition tables don't have any alternative options, DFAs are easily simulated via an algorithm.



Simulate a DFA

- Algorithm to simulate DFA

  Input: String x, DFA D.
  - Transition function is move(s,c);
  - Start state is S0;
  - Final states are F.

  Output: "yes" if D accepts x; "no" otherwise;

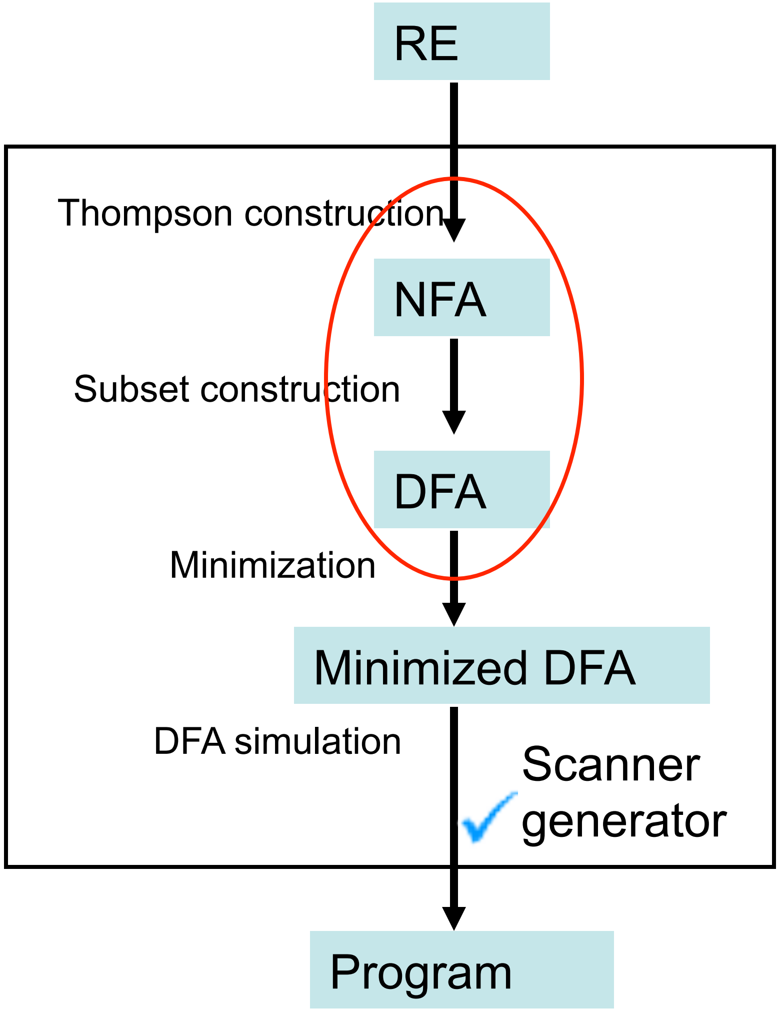
  Algorithm:

```
currentState ← s0
currentChar ← nextchar;
while currentChar ≠ eof  {
    currentState ← move(currentState, currentChar);
    currentChar ← nextchar;
}
if currentState is in F then return "yes"
  else return "no"
```

- Run the FA simulator!
- Write a simulator.

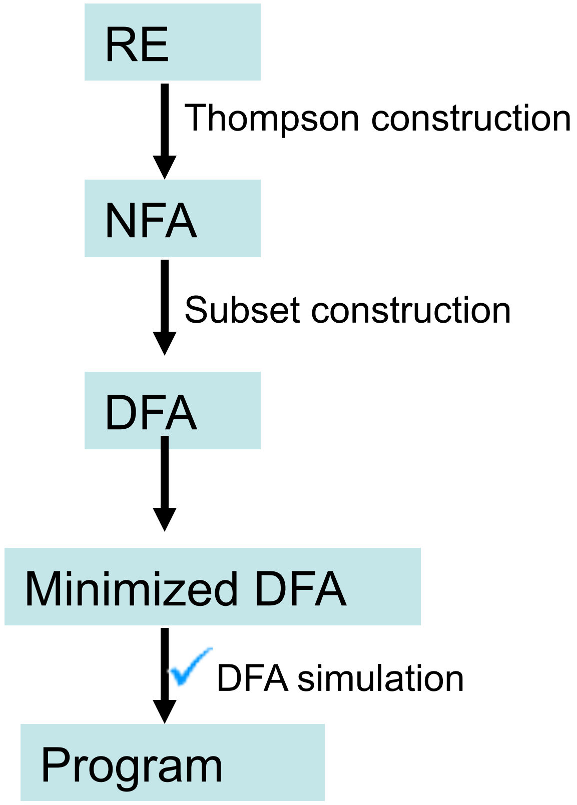
NFA to DFA

- Where we are: we are going to discuss the translation from NFA to DFA.
- Theorem: A language L is accepted by an NFA iff it is accepted by a DFA
- Subset construction is used to perform the translation from NFA to DFA.



Summarize

- We have covered many concepts
  - RE, Regular grammar, FA(NFA,DFA), Transition Diagram, Transition Table.
- What is the relationship between them?
  - RE, Regular grammar, NFA, DFA, Transition Diagram are all of the same expressive power;
  - RE is a declarative description, hence easier for us to write;
  - DFA is closer to machine;
  - Transition Diagram is a graphic representation of FA;
  - Transition Table is one of the methods to implement the transition functions in FA.
- What about regular grammar?
  - We will see its relevance in syntax analysis.
- Another path: how to derive RE from DFA?



Regular expression and regular grammar

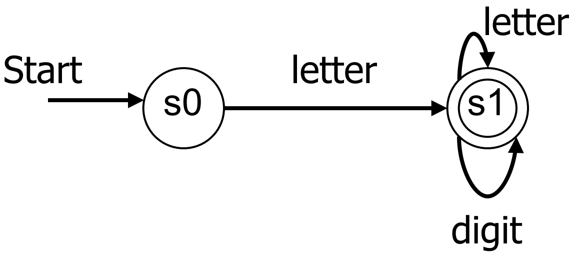
- RE to regular grammar:
  - Draw an automata for the RE
  - Construct regular grammar from automaton
  - Example

S0→letter S1

S1→letter S1

S1→digit S1

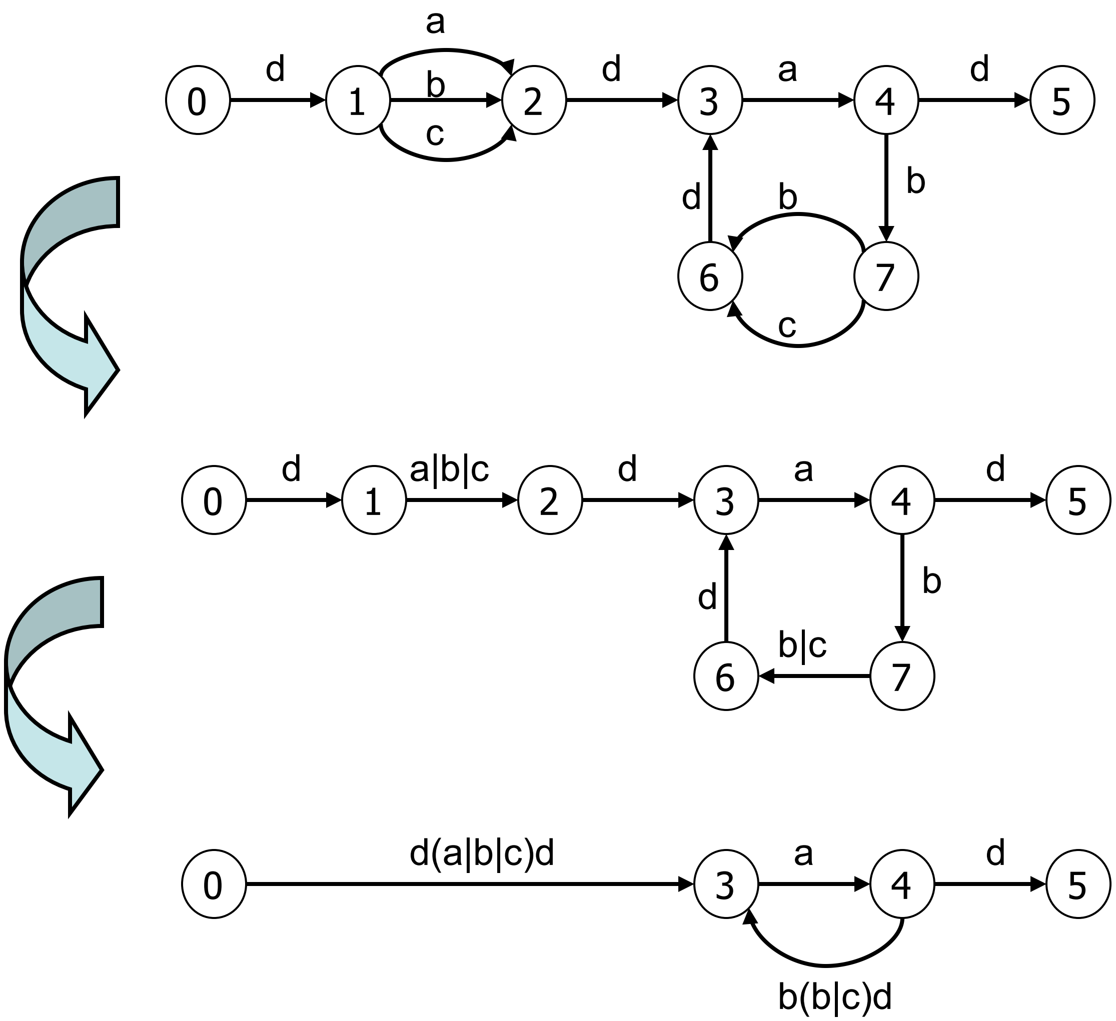
S1→ε



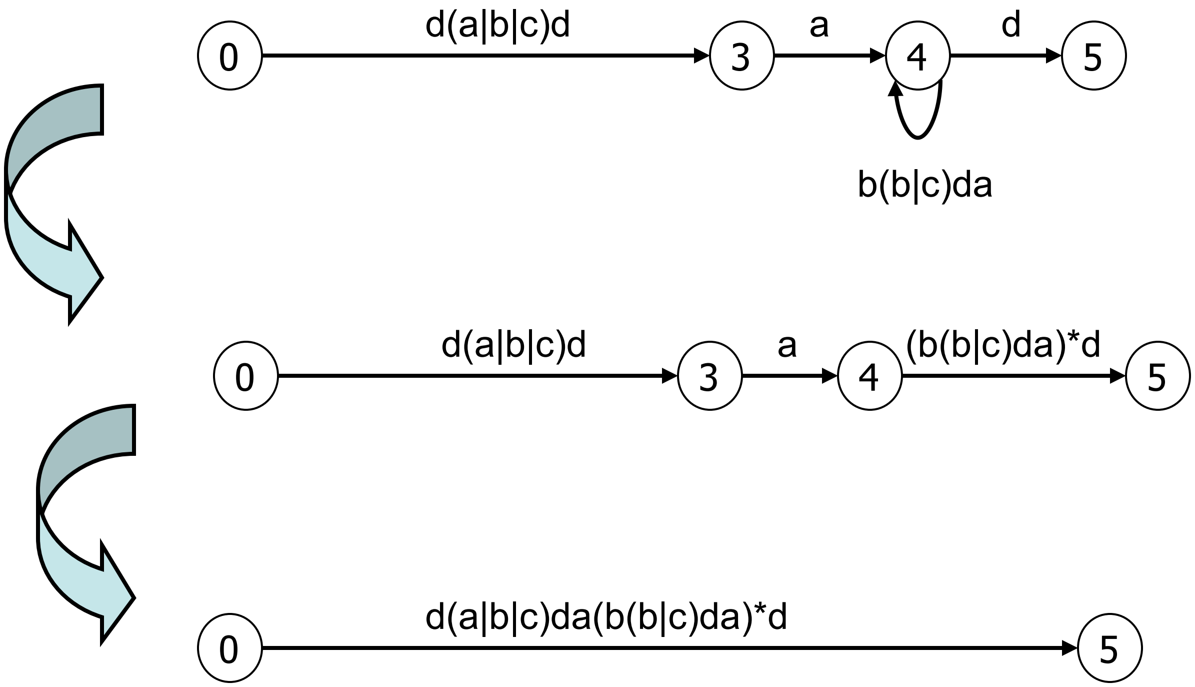
Converting DFAs to REs

1. Combine serial links by concatenation
2. Combine parallel links by alternation
3. Remove self-loops by Kleene closure
4. Select a node (other than initial or final) for removal. Replace it with a set of equivalent links whose path expressions correspond to the in and out links
5. Repeat steps 1-4 until the graph consists of a single link between the entry and exit nodes.

Example

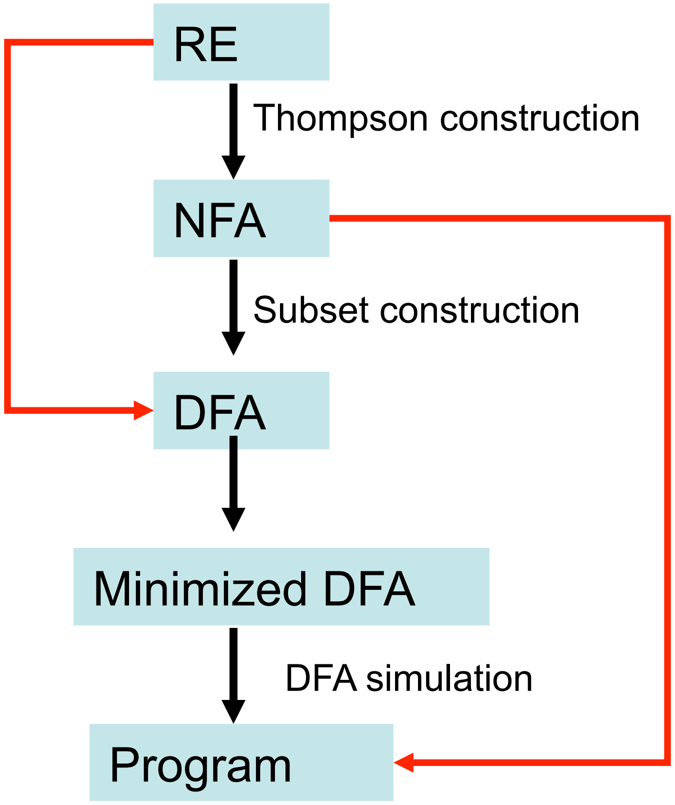

0
d
1
a
b
c
2
d
3
a
4
d
5
b
7
b
c
6
d
0
d
1
a|b|c
2
d
3
a
4
d
5
b
7
b|c
6
d
0
d(a|b|c)d
3
a
4
d
5
b(b|c)d

Example (cont.)



Issues not covered

- Regular expression to DFA directly;

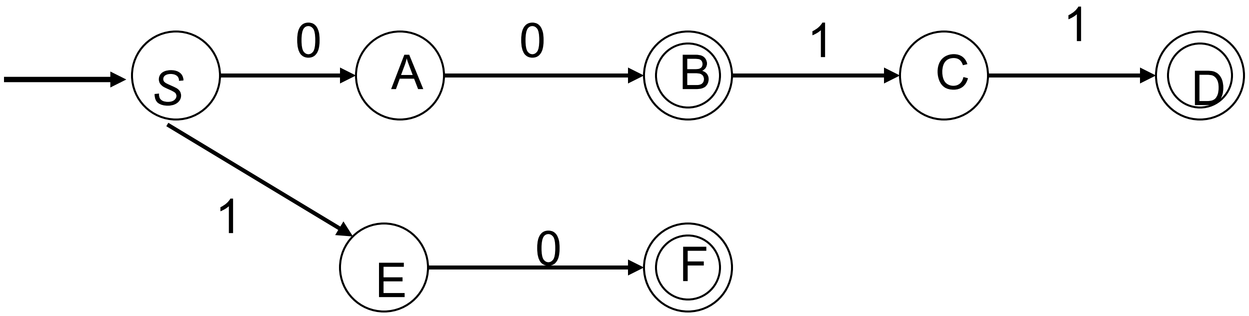


Lexical acceptors and Lexical analyzers

- DFA/NFA accepts or rejects a string;
  - They are called lexical acceptors;
- But the purpose of a lexical *analyzer* is not just to accept or reject string. There are several issues:
  - *Multiple matches:* One regular expression may match several substrings.
    - e.g., ID=letter+, String="abc", ID can match with a, ab, abc.
    - We should find the longest matches, i.e., longest substring of the input that matches the regular expression;
  - *Multiple REs*: What if one string can match several REs?
    - e.g., ID=letter+, INT="int",
    - String "int" can be both a reserved word INT, and an identifier. How can we decide it is a reserved word instead an usual identifier?
  - *Actions:* Once a token is recognized, we want to perform different tasks on them, instead of simply return the string recognized.

Longest match

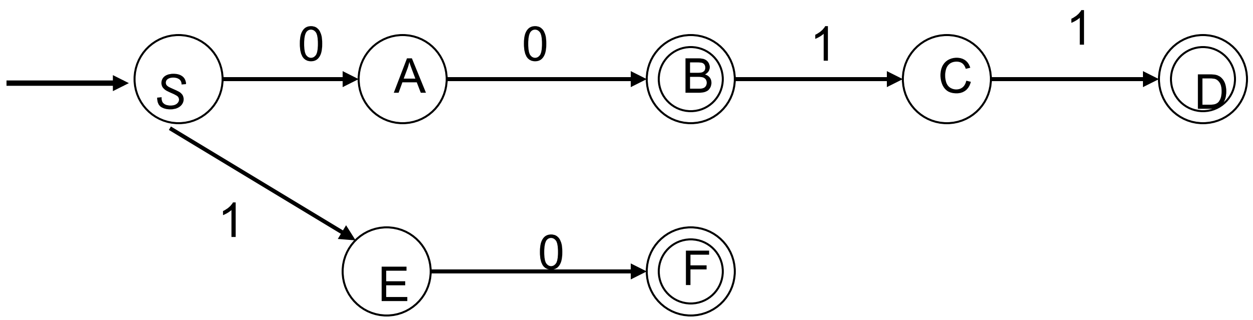
- When several substrings can match the same RE, we should return the longest one.
  - e.g., ID=letter+, String="abc", ID can match with a, ab, abc.
- Problem: what if a lexer goes past a final state of a shorter token, but then doesn't find any other matching token later?
- Example: Consider R=00|10|0011 and input w=0010.



- We reach state C with no transition on input 0.
- Solution: Keeping track of the longest match just means remembering the last time the DFA was in a final state;

Longest match (cont.)

- This is done by introducing the following variables:
  - LastFinal: final state most recently encountered;
  - InpputPositionAtLastFinal: most recent position in the input string in which the execution of the DFA was in a final state;
  - Yytext: Text of the token being matched, i.e., substring between initialInputPosition and inputPositionAtLastFinal.
- This way a longest match is recognized when the execution of the DFA reaches a dead-end, i.e., a state with no transitions.
- Each time a token is recognized, the execution of the DFA resumes in the initial state to recognize the next token.
- In general, when a token is recognized, currentInputPosition may be far beyond inputPositionAtLastFinal.



Handling multiple REs

- Combine the NFAs of all the REs into a single finite automaton.
- What if two REs matches the same string?
  - E.g., for a string “abb”, both REs “a*bb” and “ab*” matches the string. Which RE is intended?
  - It is important because different actions may take depending on the RE being matched;
  - Solution: Order REs: the RE precedes will match first.
- How about reserved words?
  - For string “int”, should we return token INT or token ID?
  - Two solutions:
    - Construct a reserved word table and look up the table every time an identifier is encountered;
    - Put “int” as an RE, and put that RE before the identifier RE. So whenever the string “int” is met, RE “int” will be matched first and the token *INT* will be returned (instead of the token *ID*).

Actions

- Actions can be added for final states;
- Actions can be described in a usual programming language. In JLex, action is described in Java.

Build a scanner for a simple language

- The language of assignment statements:

  LHS = RHS
  int LHS = RHS

  …

  - left-hand side of assignment is an identifier, with optional type declaration;
  - Identifier is a letter followed by one or more letters or digits
  - right-hand side is one of the following:
    - ID + ID
    - ID * ID
    - ID == ID
- Example statement

  int x3=x1+x2

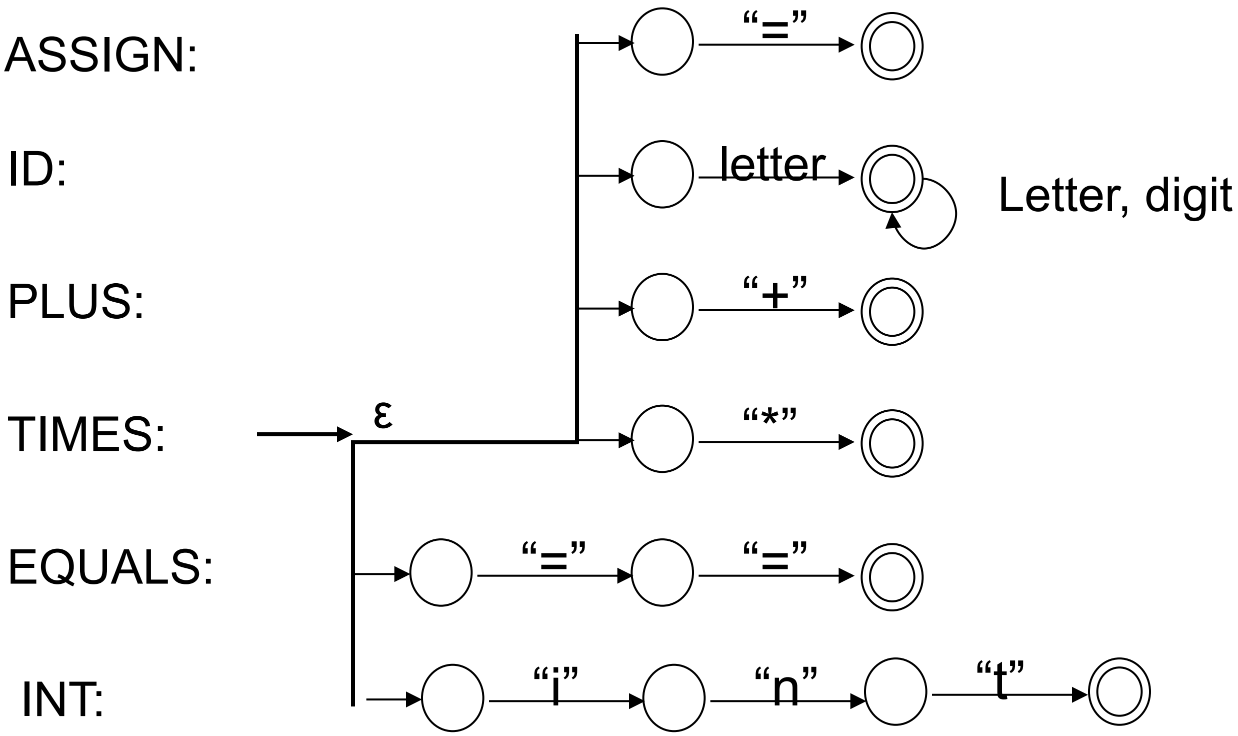
Step 1: Define tokens

Our language has six type of tokens.

- they can be defined by six regular expressions:

token	Regular expression
ASSIGN	"="
ID	letter (letter | digit)*
INT	“int”
PLUS	"+"
TIMES	"*"
EQUALS	"=="

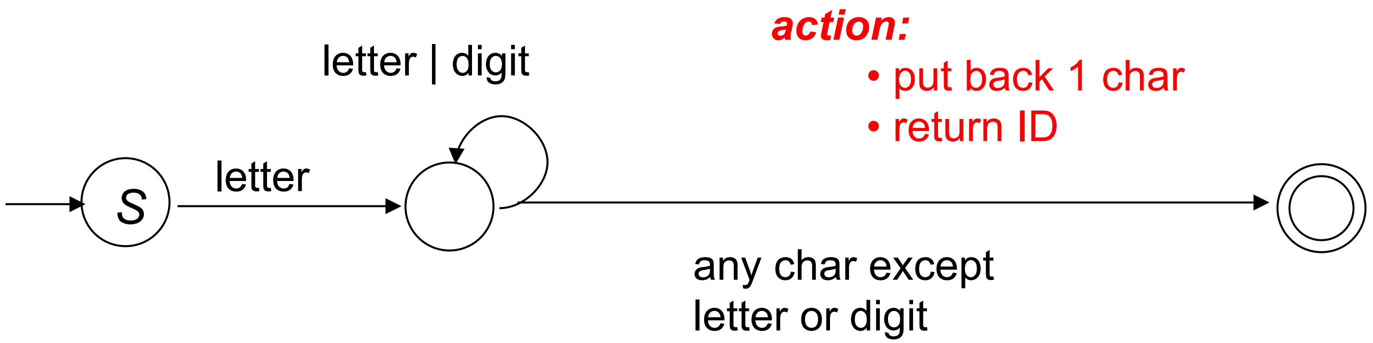
Step 2: Convert REs to NFAs



Step 3: Combine the NFAs, Convert NFAs to DFAs, minimize the DFAs

Step 4: Extend the DFA

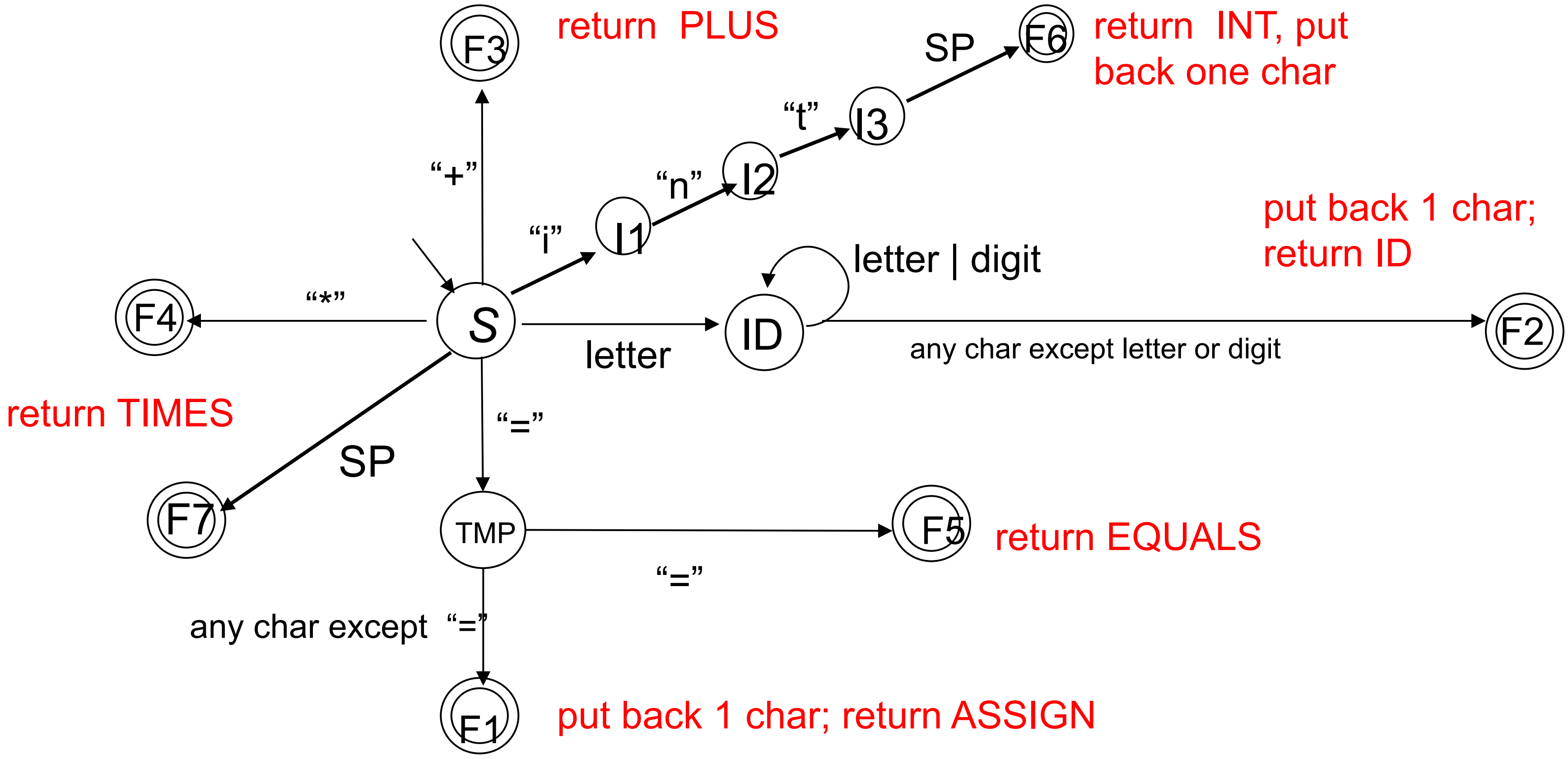
- Modify the DFA so that a final state can have
  - an associated action, such as: "put back one character" or "return token XXX“.
- For example, the DFA that recognizes identifiers can be modified as follows
  - recall that scanner is called by a parser (one token is returned per each call)
  - hence action *return* puts the scanner into state S



action:
- put back 1 char
- return ID

Step 5: Combined FA for our language

- combine the DFAs for all of the tokens in to a single FA.



- It is not a DFA. Just for illustration purpose.

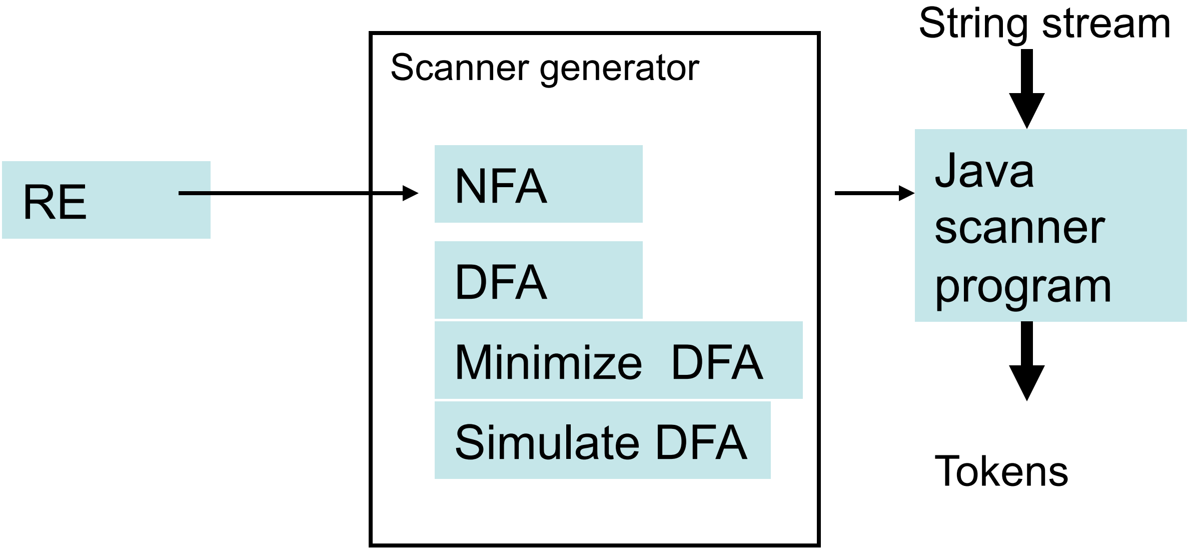
Example trace for “int x3=x1+x2”

<table>
<tr><th>Input</th><th>Last final state</th><th>Current state</th><th>Input position at lastFinalState</th><th>Current input position</th><th>Initial input position</th><th>action</th></tr>
<tr><td>int x3=x1+x2</td><td>0</td><td>S</td><td>0</td><td>0</td><td>0</td><td></td></tr>
<tr><td>nt x3=x1+x2</td><td>0</td><td>I1</td><td>1</td><td>1</td><td>0</td><td></td></tr>
<tr><td>t x3=x1+x2</td><td>0</td><td>I2</td><td>2</td><td>2</td><td>0</td><td></td></tr>
<tr><td>[sp]x3=x1+x2</td><td>0</td><td>I3</td><td>3</td><td>3</td><td>0</td><td></td></tr>
<tr><td>x3=x1+x2</td><td>0</td><td>F6</td><td>4</td><td>4</td><td>0</td><td>Action 1</td></tr>
<tr><td colspan="7">putBackOneChar(); Yytext=substring(0, 4-1); intialInputPosition=3; currentSate = S;</td></tr>
<tr><td>[sp]x3=x1+x2</td><td>0</td><td>S</td><td>3</td><td>3</td><td>3</td><td></td></tr>
<tr><td>x3=x1+x2</td><td>0</td><td>SP</td><td>4</td><td>4</td><td>3</td><td>Action 2</td></tr>
<tr><td colspan="7">Yytext=substring(3, 3); initialInputPosition=4; currentState=S;</td></tr>
<tr><td>x3=x1+x2</td><td>0</td><td>S</td><td>4</td><td>4</td><td>4</td><td></td></tr>
<tr><td>3=x1+x2</td><td>0</td><td>ID</td><td>5</td><td>5</td><td>4</td><td></td></tr>
<tr><td>=x1+x2</td><td>0</td><td>ID</td><td>6</td><td>6</td><td>4</td><td></td></tr>
<tr><td>x1+x2</td><td>0</td><td>F2</td><td>7</td><td>7</td><td>4</td><td>Action 3</td></tr>
<tr><td colspan="7">putBackOneChar; Yytext=substring(4,7-1);initialInputPosition=6; currentState=S;</td></tr>
<tr><td>=x1+x2</td><td>0</td><td>S</td><td>6</td><td>6</td><td>6</td><td></td></tr>
<tr><td>… …</td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
</table>

Scanner generator: history

- LEX
  - A lexical analyzer generator, written by Lesk and Schmidt at Bell Labs in 1975 for the UNIX operating system;
  - It now exists for many operating systems;
  - LEX produces a scanner which is a C program;
  - LEX accepts regular expressions and allows actions (i.e., code to executed) to be associated with each regular expression.
- JLex
  - Lex that generates a scanner written in Java;
  - Itself is also implemented in Java.
- There are many similar tools, for most programming languages

Overall picture



Inside lexical analyzer generator

JLex

- How does a lexical analyzer work?
  - Get input from user who defines tokens in the form that is equivalent to regular grammar
  - Turn the regular grammar into a NFA
  - Convert the NFA into DFA
  - Generate the code that simulates the DFA

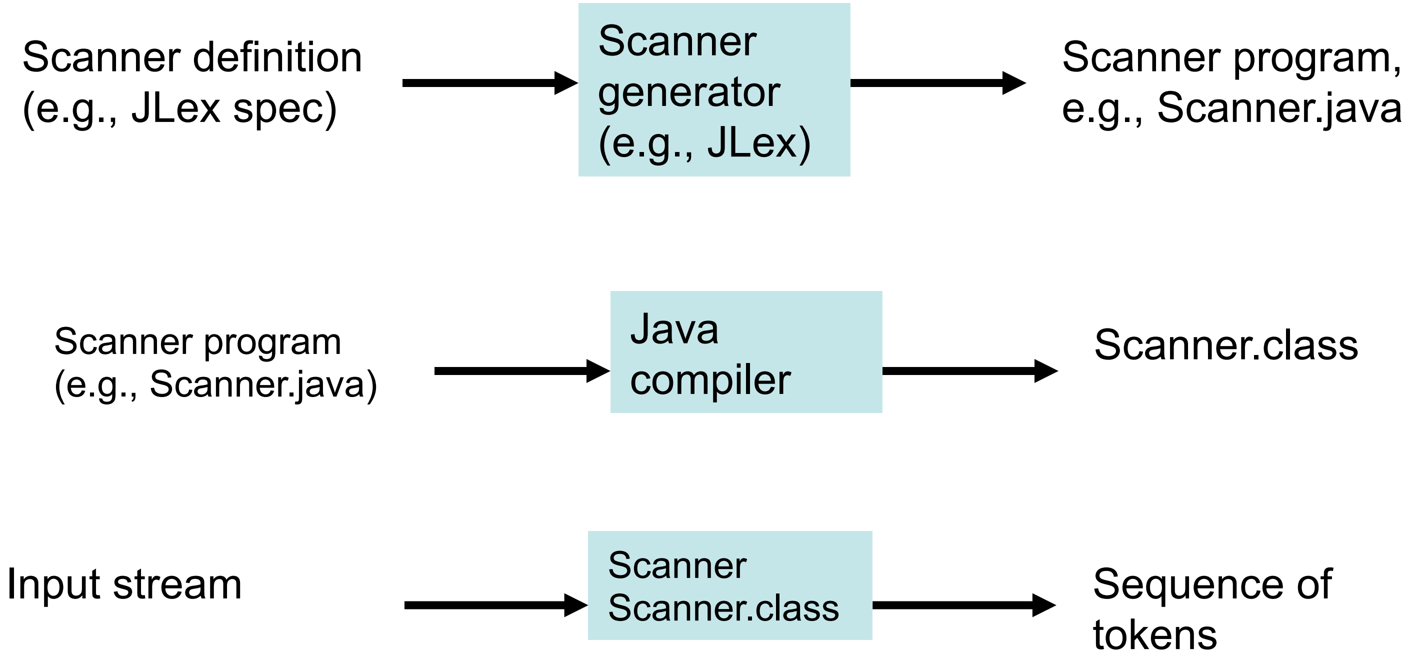
Classes in JLex:

CAccept
CAcceptAnchor
CAlloc
CBunch
CDfa
CDTrans
CEmit
CError
CInput
CLexGen
CMakeNfa
CMinimize
CNfa
CNfa2Dfa
CNfaPair
CSet
CSimplifyNfa
CSpec
CUtility
Main
SparseBitSet
ucsb

JLex

How scanner generator is used

- Write the scanner specification;
- Generate the scanner program using scanner generator;
- Compile the scanner program;
- Run the scanner program on input streams, and produce sequences of tokens.



Scanner definition (e.g., JLex spec)

Scanner generator (e.g., JLex)

Scanner program, e.g., Scanner.java

Scanner program (e.g., Scanner.java)

Java compiler

Scanner.class

Input stream

Scanner Scanner.class

Sequence of tokens

JLex specification

JLex

- JLex specification consists of three parts, separated by “%%”

  User Java code, to be copied verbatim into the scanner program, placed before the lexer class;

  %%

  JLex directives,

  macro definitions, commonly used to specify letters, digits, whitespace;

  %%

  Regular expressions and actions:

  - Specify how to divide input into tokens;
  - Regular expressions are followed by actions;
    - Print error messages; return token codes;

First JLex example simple.lex

JLex

- Recognize int and identifiers.

```
%%
%{   public static void main(String argv[]) throws java.io.IOException {
        MyLexer yy = new MyLexer(System.in);
        while (true){
          yy.yylex();
         }
        }
%}
%notunix
%type void
%class MyLexer
%eofval{ return;
%eofval}

IDENTIFIER = [a-zA-Z][a-zA-Z0-9]*

%%

"int" { System.out.println("INT recognized");}
{IDENTIFIER} { System.out.println("ID is ..." + yytext());}
\r|\n {}
. {}
```

JLex

Code generated will be in simple.lex.java

```
class MyLexer {
    public static void main(String argv[]) throws java.io.IOException {
        MyLexer yy = new MyLexer(System.in);
        while (true){
          yy.yylex();
       }
     }
    public void yylex(){
     ... ...
      case 5:{ System.out.println("INT recognized"); }
      case 7:{ System.out.println("ID is ..." + yytext()); }
      ... ...
    }
}
```

Copied from internal code directive

JLex

Running the JLex example

- Steps to run the JLex

```
D:\214>java JLex.Main simple.lex
Processing first section -- user code.
Processing second section -- JLex declarations.
Processing third section -- lexical rules.
Creating NFA machine representation.
NFA comprised of 22 states.
Working on character classes.::::::::.
NFA has 10 distinct character classes.
Creating DFA transition table.
Working on DFA states...........
Minimizing DFA transition table.
9 states after removal of redundant states.
Outputting lexical analyzer code.

D:\214>move simple.lex.java MyLexer.java

D:\214>javac MyLexer.java

D:\214>java MyLexer       // it is waiting for keyboard input
int myid0
INT recognized
ID is ...myid0
```

Exercises

- Try to modify JLex directives in the previous JLex spec, and observe whether it is still working. If it is not working, try to understand the reason.
  - Remove “%notunix” directive;
  - Change “return;” to “return null;”;
  - Remove “%type void”;
  - ... ...
- Move the Identifier regular expression before the “int” RE. What will happen to the input “int”?
- What if you remove the last line (line 19, “. {}”) ?

Change simple.lex: read input from file

```
import java.io.*;
%%
%{   public static void main(String argv[]) throws java.io.IOException {
       MyLexer yy = new MyLexer( new FileReader(“input”) );
       while (yy.yylex()>=0);
       }
%}
%integer
%class MyLexer

%%

"int" { System.out.println("INT recognized");}
[a-zA-Z_][a-zA-Z0-9_]* { System.out.println("ID is ..." + yytext());}
\r|\n|. {}
```

- %integer: to make the returning type of yylex() as int.

Extend the example: add returning and use classes

JLex

- When a token is recognized, in most of the case we want to return a token object, so that other programs can use it.

```
class UseLexer {
  public static void main(String [] args) throws java.io.IOException {
    Token t;   MyLexer2 lexer=new MyLexer2(System.in);
     while ((t=lexer.yylex())!=null) System.out.println(t.toString());
  }
}
class Token {
  String type;   String text;     int line;
  Token(String t, String txt, int l) { type=t; text=txt; line=l; }
  public String toString(){ return text+" " +type + " " +line; }
}
%%
%notunix
%line
%type Token
%class MyLexer2
%eofval{ return null;
%eofval}
IDENTIFIER = [a-zA-Z_][a-zA-Z0-9_]*
%%
"int" { return(new Token("INT", yytext(), yyline));}
{IDENTIFIER} { return(new Token("ID", yytext(), yyline));}
\r|\n {}
. {}
```

JLex

Code generated from mylexer2.lex

```
class UseLexer {
  public static void main(String [] args) throws java.io.IOException {
    Token t;   MyLexer2 lexer=new MyLexer2(System.in);
     while ((t=lexer.yylex())!=null) System.out.println(t.toString());
  }
}
class Token {
  String type;   String text;     int line;
  Token(String t, String txt, int l) { type=t; text=txt; line=l; }
  public String toString(){ return text+" " +type + " " +line; }
}

Class MyLexer2 {
  public Token  yylex(){
     ... ...
      case 5: { return(new Token("INT", yytext(), yyline)); }
      case 7: { return(new Token("ID", yytext(), yyline)); }
                ... ...
   }

}
```

Running the extended lex specification mylexer2.lex

JLex

```
D:\214>java JLex.Main mylexer2.lex
Processing first section -- user code.
Processing second section -- JLex declarations.
Processing third section -- lexical rules.
Creating NFA machine representation.
NFA comprised of 22 states.
Working on character classes.::::::::.
NFA has 10 distinct character classes.
Creating DFA transition table.
Working on DFA states..........
Minimizing DFA transition table.
9 states after removal of redundant states.
Outputting lexical analyzer code.

D:\214>move mylexer2.lex.java MyLexer2.java

D:\214>javac MyLexer2.java

D:\214>java UseLexer
int
int INT 0
x1
x1 ID 1
```

Another example

JLex

```
import java.io.IOException;
 %%
 %public
 %class  Numbers_1
 %type   void
 %eofval{ return;
 %eofval}

  %line
  %{      public static void main (String args []) {
    Numbers_1 num = new Numbers_1(System.in);
    try {
      num.yylex();
    } catch (IOException e) { System.err.println(e); }
   }
  %}

  %%
  \r\n      { System.out.println("--- " + (yyline+1));  }
  .*\r\n    { System.out.print  ("+++ " + (yyline+1)+"\t"+yytext());  }
```

User code (first section of JLex)

JLex

- User code is copied verbatim into the lexical analyzer source file that JLex outputs, at the top of the file.
  - Package declarations;
  - Imports of an external class
  - Class definitions
- Generated code

```
package declarations;
import packages;
Class definitions;
class Yylex {
  ... ...
}
```

- Yylex class is the default lexer class name. It can be changed to other class name using *%class* directive.

JLex directives (Second section)

- Internal code to lexical analyzer class
- Marco definition
- State declaration
- Character/line counting
- Lexical analyzer component title
- Specifying the return value on end-of-file
- Specifying an interface to implement

Internal Code to Lexical Analyzer Class

JLex

- %{ …. %} directive permits the declaration of variables and functions internal to the generated lexical analyzer
- General form:

```
%{
    <code >
%}
```

- Effect: <code > will be copied into the Lexer class, such as MyLexer.

```
class MyLexer{
  ….. <code> ……
}
```

- Example

```
public static void main(String argv[]) throws java.io.IOException {
    MyLexer yy = new MyLexer(System.in);
      while (true){   yy.yylex();      }
}
```

- Difference with the user code section
  - It is copied inside the lexer class (e.g., the MyLexer class)

Macro Definition

JLex

- Purpose: define once and used several times;
  - A must when we write large lex specification.
- General form of macro definition:
  - <name> = <definition>
  - should be contained on a single line
  - Macro name should be valid identifiers
  - Macro definition should be valid regular expressions
  - Macro definition can contain other macro expansions, in the standard {<name>} format for macros within regular expressions.
- Example
  - Definition (in the second part of JLex spec):

    ```
    IDENTIFIER = [a-zA-Z_][a-zA-Z0-9_]*
    ALPHA=[A-Za-z_]
    DIGIT=[0-9]
    ALPHA_NUMERIC={ALPHA}|{DIGIT}
    ```
  - Use (in the third part):

    ```
    {IDENTIFIER} {return new Token(ID, yytext()); }
    ```

State directive

- Same string could be matched by different regular expressions, according to its surrounding environment.
  - String “int” inside comment should not be recognized as a reserved word, not even as an identifier.
- Particularly useful when you need to analyze mixed languages;
- For example, in JSP, Java programs can be imbedded inside HTML blocks. Once you are inside Java block, you follow the Java syntax. But when you are out of the Java block, you need to follow the HTML syntax.
  - In java “int” should be recognized as a reserved word;
  - In HTML “int” should be recognized just as a usual string.
- States inside JLex

```
<HTMLState> %{   { yybegin(JavaState); }
<HTMLState>   “int” {return string; }
<JavaState>   %}   { yybegin(HTMLState); }
<JavaState>    “int” {return keyword; }
```

State Directive (cont.)

- Mechanism to mix FA states and REs
- Declaring a set of “start states” (in the second part of JLex spec)

  %state state0 [, state1, state2, ….]
- How to use the state (in the third part of JLex spec):
  - RE can be prefixed by the set of start states in which it is valid;
- We can make a transition from one state to another with input RE
  - yybegin(STATE) is the command to make transition to STATE;
- YYINITIAL : implicit start state of yylex();
  - But we can change the start state;
- Example (from the sample in JLex spec):

```
%state COMMENT
%%
<YYINITIAL>if          {return new tok(sym.IF,″IF″);}
<YYINITIAL>[a-z]+      {return new tok(sym.ID, yytext());}
<YYINITIAL>″/*″        {yybegin(COMMENT);}
<COMMENT>″*/″          {yybegin(YYINITIAL);}
<COMMENT>.             {}
```

JLex

Character and line counting

- Sometimes it is useful to know where exactly the token is in the text. Token position is implemented using line counting and char counting.
- Character counting is turned off by default, activated with the directive “%char”
  - Create an instance variable `yychar` in the scanner;
  - zero-based character index of the first character on the matched region of text.
- Line counting is turned off by default, activated with the directive “%line”
  - Create an instance variable yyline in the scanner;
  - zero-based line index at the beginning of the matched region of text.♪
- Example

  “int” { return (new Yytoken(4,yytext(),yyline,yychar,yychar+3)); }

JLex

Lexical analyzer component titles

- Change the name of generated

lexical analyzer class	%class <name>
the tokenizing function	%function <name>
the token return type	%type <name>

- Default names

```
class Yylex {        /* lexical analyzer class */
  public Yytoken          /* the token return type */
          yylex() { …}   /* the tokenizing function */
  ==> Yylex.yylex()  returns  Yytoken  type
```

JLex

Specifying an Interface to implement

- Form: %implements <InterfaceName>
- Allows the user to specify an interface which the Yylex or your lexer class will implement.
- The generated parser class declaration will look like:

```
class MyLexer implements InterfaceName {
 …….
}
```

Regular expression rules

JLex

General form:	regularExpression	{ action}
Example:	{IDENTIFIER}	{ System.out.println("ID is ..." + yytext());}
Interpretation:	Patten to be matched	code to be executed when the pattern is matched

- Code generated in MyLexer:

  “ case 2: { System.out.println("ID is ..." + yytext());} “

Regular Expression Rules

JLex

- Specifies rules for breaking the input stream into tokens
- Regular Expression + Actions (java code)

  [<states>] <expression>   { <action>}

- When matched with more than one rule,
  - choose the rule that is given first in the Jlex spec.
    - Refer the “int” and IDENTIFIER example.
- The rules given in a JLex specification should match all possible input.
- An error will be raised if the generated lexer receives input that does not match any of its rules
  - E.g., the rules only listed the case for Identifiers, and said nothing about numbers, but your input has numbers.
  - This is the most common error (more than 50%)
  - put the following rule at the bottom of RE spec

    - `{java.lang.System.out.println(“Error:” + yytext());}`

dot(.) will match any input except for the newline.

Available lexical values within action code

JLex

- java.lang.String yytext()
  - matches portion of the character input stream;
  - always active.
- Int yychar
  - Zero-based character index of the first character in the matched portion of the input stream;
  - activated by *%char* directive.
- Int yyline
  - Zero-based line number of the start of the matched portion of the input stream;
  - activated by *%line* directive.

Regular expression in JLex

- Special characters: ? + | ( ) ˆ $ / ; . = < > [ ] { } ” \ and blank
  - After \ the special characters lose their special meaning.
  - Example: \+
- Between double quotes ” all special characters but \ and ” lose their special meaning.
  - Example: ”+”
- The following escape sequences are recognized: \b \n \t \f \r.
- With [ ] we can describe sets of characters.
  - [abc] is the same as (a|b|c). Note that it is not equivalent to abc
  - With [ˆ] we can describe sets of characters.
  - [ˆ\n\”] means anything but a newline or quotes
  - [ˆa–z] means anything but ONE lower-case letter
- We can use . as a shortcut for [ˆ\n]
- $: denotes the end of a line. If $ ends a regular expression, the expression matched only at the end of a line.

JLex

Concluding remarks

- Focused on Lexical Analysis Process, Including
  - Regular Expressions
  - Finite Automaton
  - Conversion
  - Lex
- Regular grammar=regular expression
- Regular expression→ NFA→ DFA→ lexer
- The next step in the compilation process is Parsing:
  - Context free grammar;
  - Top-down parsing and bottom up parsing.