

Compiler Construction

SMD163

Lecture 3: Lexer generators

Viktor Leijon & Peter Jonsson with slides by Johan Nordlander.

Contains material generously provided by Mark P. Jones

COMPUTER SCIENCE
AND ELECTRICAL ENGINEERING
LULEÅ UNIVERSITY OF TECHNOLOGY

1

But first this ...

2

Homework:

- ◆ Compile and run the quick calculator program (link QuickCalc.java). You can also rewrite it in a language of your own choice if you wish ...
- ◆ Modify your quick calculator to support unary minus:
 - Extend the grammar to support unary minus
 - Add methods to support unary minus (you can use the same methods as for unary plus)
 - Modify the parser to support unary minus (higher precedence than any other operator (try $-2 * 3$, $-(2+3)$, $4 * -5$, and $--23$ should give 1, -5, -20, and 23, respectively))

Remember
this?

3

Quick Unary Minus:

```
static class NegExpr extends Expr {
    private Expr exp;

    NegExpr(Expr exp) {
        this.exp = exp;
    }

    void compile() {
        exp.compile();
        emit("ineg");
    }

    int eval() {
        return -exp.eval();
    }
}
```

4

continued...

```
static Expr atom() {           // Atom = integer
    if (token==integer) {     //      | '(' Expr ')'
        int f = tokval;       //      | '-' Atom
        getToken();
        return new IntExpr(f);
    } else if (token=='(') {
        ...
    } else if (token=='-') {
        getToken();
        return new NegExpr(atom());
    } else {
        calcError("syntax error");
    }
    return null; /* not used */
}
```

5

continued...

```
static Expr atom() {           // Atom = integer
    if (token==integer) {     //      | '(' Expr ')'
        int f = tokval;       //      | '-' Atom
        getToken();
        return new IntExpr(f);
    } else if (token=='(') {
        ...
    } else if (token=='-') {
        getToken();
        return new IntExpr(-atom().eval());
    } else {
        calcError("syntax error");
    }
    return null; /* not used */
}
```

what's wrong
with this?

6

Recap: Handwritten Lexicers:

- ◆ Doesn't require sophisticated programming.
- ◆ Often requires care to avoid non-determinism, or potentially expensive backtracking.
- ◆ Can be fine tuned for performance and for the language concerned.
- ◆ But it might also be something we would want to automate ...

7

Can a Machine do Better?

- ◆ It can be hard to write a (correct) lexer by hand ...
- ◆ But that's not surprising: finite state machines are low level ... an 'assembly language of lexical analysis'
- ◆ Can we build a lexical analyzer generator that will take care of all the dirty details, and let the humans work at a higher level?
- ◆ If so, what would its input look like?

8

Formal Languages:

- ◆ Pick an alphabet A : a set of symbols.
 - For lexical analysis, "symbol" = "character".
 - For parsing, "symbol" = "token".
- ◆ The set of all finite strings of symbols taken from A is written A^* .
- ◆ A language (over A) is a subset of A^*
- ◆ Two issues:
 - How do we describe a certain language over A (subset of A^*) ?
 - How do we check whether a string belongs to the language ?

9

Regular Expressions:

- ◆ A syntax for describing regular languages.
- ◆ A widely used notation for describing patterns in text strings: emacs, vi, grep, awk, perl, ...
- ◆ ... but also good for describing the lexical structure of a programming language ...

10

Regular Expressions:

Expression	Meaning
r_1r_2	Sequencing: text matching r_1 followed by text matching r_2 .
$r_1 \mid r_2$	Alternatives: text matching r_1 or text matching r_2 .
r^*	Repetition: text matching r zero or more times.
(r)	Grouping: text matching r .
c	Constant: text matching the constant c
ϵ	Empty: matches the empty string.

11

Derived Forms:

Expression	Meaning
r^+	Repetition: text matching r one or more times. $r^+ = rr^*$
$r?$	Optional: Optional text matching r . $r? = (r \mid \epsilon)$.
$[abc]$	Character Classes: enumerations or ranges. E.g., $[a-zA-Z]$.
$.$	Wildcard: Matches any character (except newline).
\wedge	Matches the empty string, but only at the beginning of a line.
$\$$	Matches the empty string, but only at the end of a line.

12

Relevant Examples:

Expression	Possible Use
<code>[0-9]+</code>	Integer literals
<code>if</code>	The keyword <code>if</code>
<code>[A-Za-z][A-Za-z0-9_]*</code>	Identifier in C++
<code>[\t\n]*</code>	Whitespace
<code>//.*\$</code>	C++ comment
<code>{int}{frac}?{exp}?</code> where <code>int</code> = <code>[0-9]+</code> <code>frac</code> = <code>"\."</code> {int} <code>exp</code> = <code>[Ee](+ -)?{int}</code>	Floating point literal

13

The lex Family:

- ◆ `lex` is a tool for generating C programs to implement lexical analyzers.
- ◆ It can also be used as a quick way of generating simple text processing utilities.
- ◆ `lex` dates from the mid-seventies and has spawned a family of clones: `flex`, `ML lex`, `JLex`, `JFlex`, etc...
- ◆ `Lex` is based on ideas from the theory of formal languages and automata

14

Variations on the Theme:

- ◆ There are quite a few `lex`-like tools each catering to particular
 - programming languages
 - operating system/environments
- ◆ They vary in minor details of syntax etc., but the `lex` heritage is usually pretty clear.
- ◆ Most of them have a lot of features; read the manuals!
- ◆ For C/C++, I recommend `flex`
- ◆ For Java, I recommend `JFlex` (www.jflex.de)

15

The Ins and Outs of JFlex:

- ◆ The input to `JFlex` is a set of rules, each of which specifies:
 - A pattern, given by a regular expression.
 - An action to be performed when the pattern is matched.
- ◆ The output of `JFlex` is a Java program (a class `Yylex` with a public method `yylex()` in a file `Yylex.java`) that searches for patterns in its input, and carries out the associated actions if/when they are found.

16

Format of a JFlex Input File:

```
... Java package and imports section ...
%%
%option ...
...
macro = regexpr
...
%%
... rules ...
```

A rule is of the form

```
    regexpr { Java statements }
```

At least one rule is required; other sections are optional

17

Simple JFlex Programs:

- ◆ Option `%standalone` adds a default main method to the generated class
- ◆ Running standalone, input that does not match any pattern is copied unchanged to the output
- ◆ A program that copies its input to its output without any change:

```
%%
%standalone
%%
[] {}
```

18

Simple JFlex Programs:

- ◆ Text matching rules with empty actions is ignored
- ◆ A program that deletes trailing tabs and spaces at the end of each line:

```
%%
%standalone
%%
[ \t]+$ {}
```

19

Simple JFlex Programs:

- ◆ Text matching rules with empty actions is ignored
- ◆ A program that deletes trailing carriage returns from the end of a line --- useful for converting DOS files to Unix format:

```
%%
%standalone
%%
\r$ {}
```

20

Simple JFlex Programs:

- ◆ Output produced by an action replaces input matched by a pattern
- ◆ A global search and replace utility:

```
%%
%standalone
%%
compiler { System.out.print("thing"); }
```

21

Simple JFlex Programs:

- ◆ There is no need for actions to produce output:

```
%%
%{
    int lines=0, chars=0;
}%
%eof{
    System.out.println("Lines: " + lines);
    System.out.println("Chars: " + chars);
}%eof
%%
\n { lines++; chars++; }
. { chars++; }
```

22

Simple JFlex Programs:

- ◆ Output for a given pattern may depend on context:

```
%%
%standalone
%{
    boolean flag = false;
}%
%%
^BEGIN\n { flag = true; }
(.|\n) { if (flag)
        System.out.print(yytext()); }
^END\n { flag = false; }
```

23

Simple JFlex Programs:

- ◆ The maximal munch rule applies
- ◆ Strips out one line comments, and places each non-whitespace sequence of characters on a separate line:

```
%%
%standalone
%%
[ \t\n]+ { System.out.print('\n'); }
"//".* { }
```

24

Simple JFlex Programs:

- ◆ If two rules match the same lexeme, the first one wins:

```
%%
%standalone
%%
if          { System.out.print("IF"); }
[A-Za-z]+  { System.out.print(yytext()); }
.          { }
```

25

Towards a JFlex Lexer:

```
%%
%integer
%implements MyTokens
%%

if          { return IF; }
else       { return ELSE; }
[A-Za-z][A-Za-z0-9]* { return IDENT; }
[ \n\t]    { }
"//".*     { }
...
```

26

Towards a JFlex Lexer:

- ◆ Calling the lexer repeatedly:

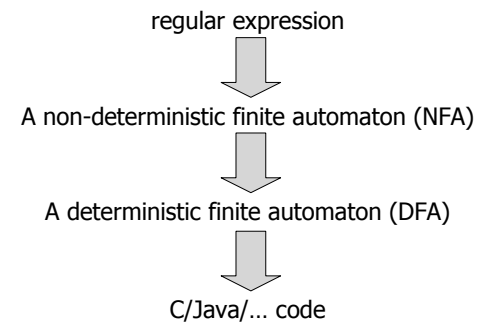
```
...
Yylex lexer = new Yylex(System.in);
try
while (lexer.yylex() != lexer.YYEOF) {
    /* ok */
}
System.out.println("Success");
```

- ◆ Failures throw an exception

27

How lex & friends work:

- ◆ Lex tackles the generation of lexers via two intermediate stages:



28

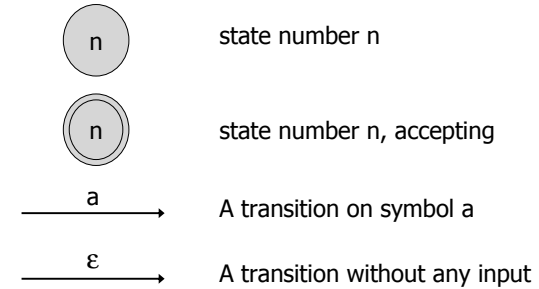
Finite Automata

- ◆ Abstract devices for recognizing (checking) whether a string belongs to some regular language
- ◆ For a given string, a finite automaton either accepts it (the string belongs to the corresponding language), or rejects it
- ◆ Intuitive way of operation: based on the current state and the next input character, either reject the character, or accept it and go to the next state

29

Finite Automata:

- ◆ We will use the following symbols to describe automata:



30

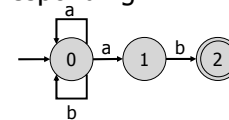
(Non) Deterministic?

- ◆ A machine is non-deterministic (an NFA) if there is a state with:
 - more than one transition on the same symbol; or
 - with an ϵ transition.
- ◆ Otherwise, the machine is deterministic (a DFA).

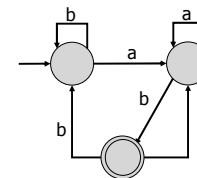
31

Example:

- ◆ Given a regular expression $(a|b)^*ab$, we can build a corresponding NFA:



- ◆ An equivalent DFA:



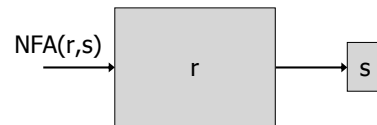
32

Building NFAs:

◆ We will describe the construction of NFAs as a process that takes:

- A regular expression r ; and
- A pointer s to another machine.

◆ The result is a machine:

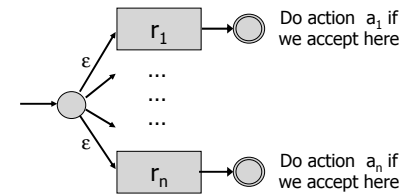


◆ ... which recognizes r , and then goes to s .

33

Building the Main Recognizer:

For an input file with patterns r_1, \dots, r_n and corresponding actions a_1, \dots, a_n , we construct the following NFA:



34

Simple Cases:

$$\text{NFA}(\epsilon, \rightarrow \boxed{s}) = \rightarrow \boxed{s}$$

$$\text{NFA}(c, \rightarrow \boxed{s}) = \rightarrow \text{circle} \xrightarrow{c} \boxed{s}$$

$$\begin{aligned} \text{NFA}(r_1 r_2, \rightarrow \boxed{s}) &= \text{NFA}(r_1, \text{NFA}(r_2, \rightarrow \boxed{s})) \\ &= \rightarrow \boxed{r_1} \rightarrow \boxed{r_2} \rightarrow \boxed{s} \end{aligned}$$

35

Alternatives and Repetition:

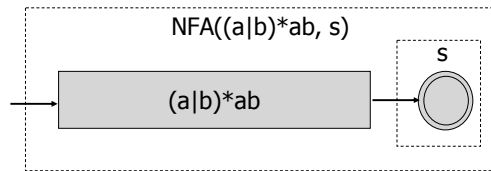
$$\text{NFA}(r_1 | r_2, \rightarrow \boxed{s}) = \rightarrow \text{circle} \begin{cases} \xrightarrow{\epsilon} \boxed{r_1} \rightarrow \boxed{s} \\ \xrightarrow{\epsilon} \boxed{r_2} \rightarrow \boxed{s} \end{cases}$$

$$\text{NFA}(r^*, \rightarrow \boxed{s}) = \rightarrow \text{circle} \begin{cases} \xrightarrow{\epsilon} \boxed{r} \rightarrow \text{circle} \xrightarrow{\epsilon} \boxed{r} \rightarrow \text{circle} \dots \\ \xrightarrow{\epsilon} \boxed{s} \end{cases}$$

36

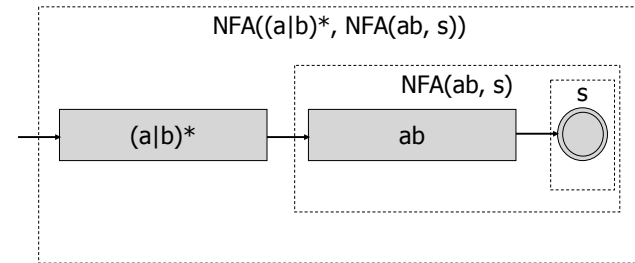
Following the Algorithm:

Let's use the algorithm to build an NFA for the regular expression $(a|b)^*ab$:



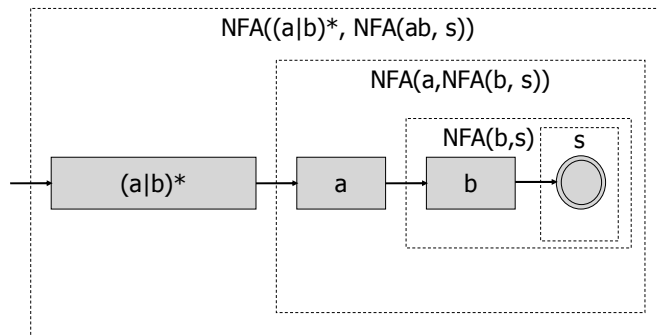
37

Following the Algorithm:



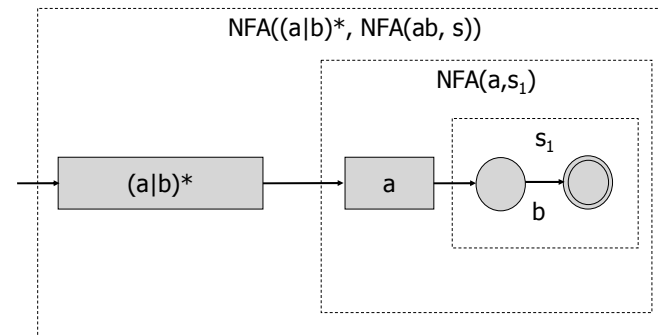
38

Following the Algorithm:



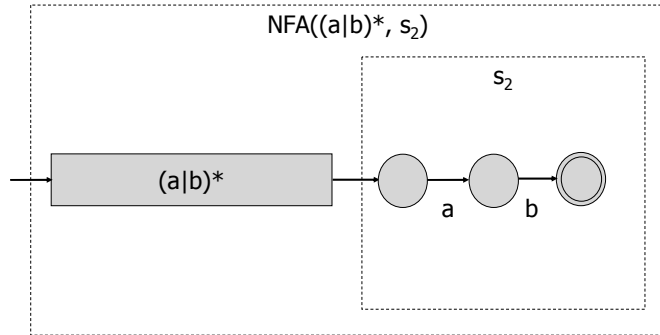
39

Following the Algorithm:



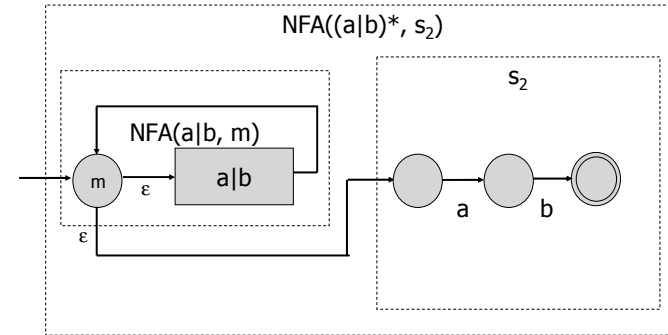
40

Following the Algorithm:



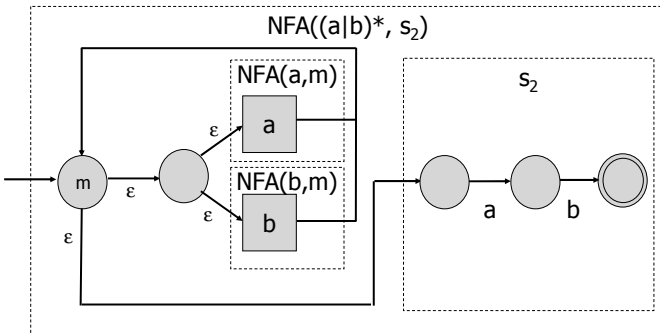
41

Following the Algorithm:



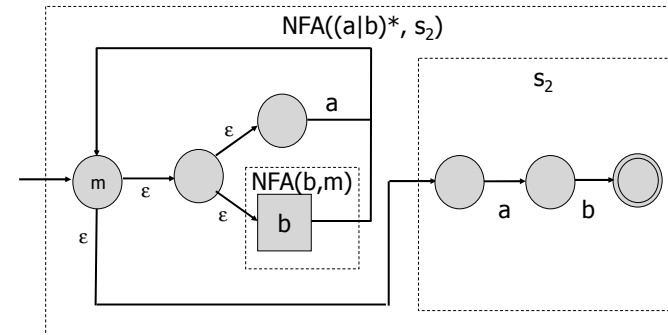
42

Following the Algorithm:



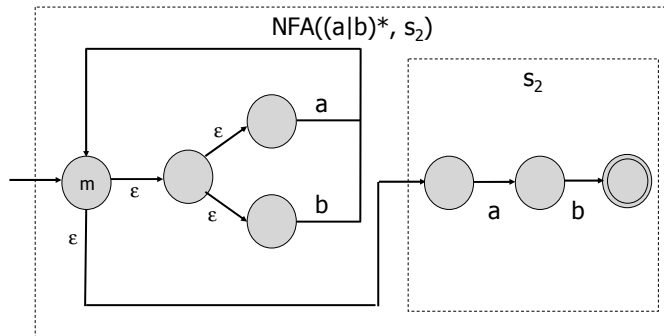
43

Following the Algorithm:



44

Following the Algorithm:

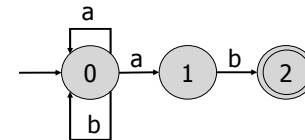


The right structure, but more complex than we might have hoped ... we'll fix this soon ...

45

From NFAs to DFAs:

◆ Consider the simple NFA for (a|b)*ab:



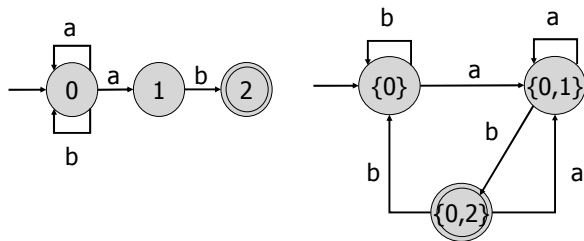
◆ How do we know which branch to take in state 0 when we see an a?

- We might end up in state 0 or in state 1.
- Let's make this alternative explicit!

46

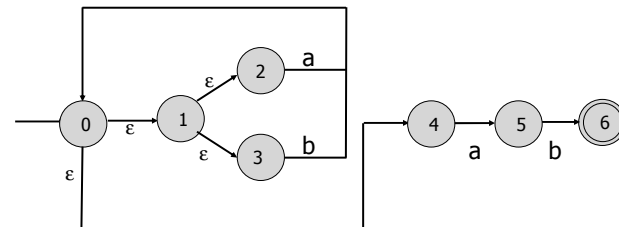
Illustrating NFA to DFA:

◆ Key idea: label every state in the NFA with a **set** of DFA states:



47

Making a DFA for the Example:



- ◆ Start state is {0}
- ◆ From {0}, we can reach {0,1,2,3,4} without any input (this is the ϵ -closure of {0}).
- ◆ {0,1,2,3,4} goes to {0,5}, hence {0,1,2,3,4,5} on a.
- ◆ {0,1,2,3,4} goes to {0}, hence {0,1,2,3,4} on b.

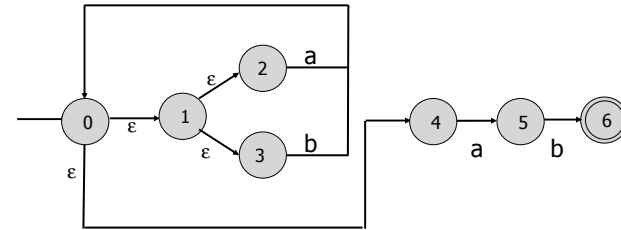
48

ϵ -closure:

- ◆ If S is a set of NFA states, the ϵ -closure of S is the set of all states that can be reached from S by an ϵ -transition.
- ◆ If $s \in S$, then $s \in \epsilon$ -closure(S).
- ◆ If $s \in \epsilon$ -closure(S), and there is an ϵ -transition from s to t , then $t \in \epsilon$ -closure(S).

49

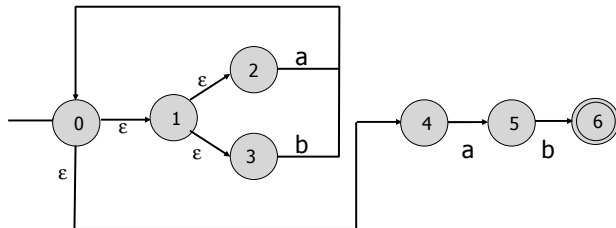
Making a DFA for the Example:



- ◆ $\{0,1,2,3,4,5\}$ goes to $\{0,5\}$, hence $\{0,1,2,3,4,5\}$ on a .
- ◆ $\{0,1,2,3,4,5\}$ goes to $\{0,6\}$, hence $\{0,1,2,3,4,6\}$ on b .

50

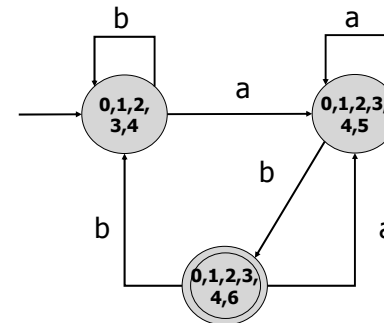
Making a DFA for the Example:



- ◆ $\{0,1,2,3,4,6\}$ goes to $\{0,5\}$, hence $\{0,1,2,3,4,5\}$ on a .
- ◆ $\{0,1,2,3,4,6\}$ goes to $\{0\}$, hence $\{0,1,2,3,4\}$ on b .
- ◆ $\{0,1,2,3,4,6\}$ is an accept state in the DFA because 6 is an accept in the NFA.

51

The Final DFA:



52

More Formally:

- ◆ If the start state of the NFA is labeled n , then the start state of the DFA is labeled $\{n\}$.
- ◆ For each character $a \in A$ and for each state X , there is a single transition from X on a which goes to the state:
 $Y = \epsilon\text{-closure}(\{y \mid x \in X \text{ and } x \text{ goes to } y \text{ on } a \text{ in the NFA}\})$
- ◆ A state X in the DFA is an accept state if any $x \in X$ is an accept state in the NFA.
- ◆ We can delete/ignore any transitions to the empty state, $\{\}$, because a transition to $\{\}$ will never lead to an accept state.

53

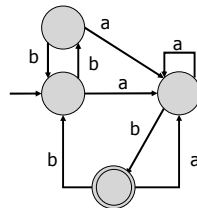
State Explosion?

- ◆ In theory, given an NFA with n distinct states, the corresponding DFA might have as many as 2^n distinct states!
- ◆ In practice, we need only consider those states that are reachable from the initial start state; this is usually much lower than 2^n .

54

DFA Minimization:

- ◆ Is this DFA the only DFA that will recognize $(a|b)^*ab$?
- ◆ Clearly not:



- ◆ But every regular set is recognized by a minimum state DFA that is unique up to state names.
- ◆ DFA minimization, however, is beyond the scope of this class ...

55

From DFA to Implementation:

- ◆ The lex approach – use a general, table driven algorithm:

```

for (;;) {
    state = table[state][getchar()];
    if (state==ACCEPT)
        break;
    if (state == X)
        throw (new ScanError());
}
/* user code */
return TOKEN23;

```

State	'a'	'b'	'c'	'd'	'e'	...
...
34	37	14	ACCEPT	X	X	...
...

56

From DFA to Implementation:

- ◆ The flex approach – generate custom executable code:

```
...
state34 : switch(getChar()) {
    case 'a' : goto state37;
    case 'b' : goto state14;
    case 'c' : goto accept;
    default : goto stuck;
}
...
accept: /* user code */ { return TOKEN23; }
```

57

Handwritten or Machine Generated?

- ◆ A mildly controversial topic!
- ◆ Issues include:
 - How efficient is the generated lexer?
 - How easily does it interface to other code ?
 - How natural is the input? If the language you are compiling has some awkward features, the lexers produced by a tool might need some massaging to do “the right thing”.
 - How good are the error messages?

58

Summary:

- ◆ Regular expressions provide a high-level language for describing lexemes.
- ◆ lex is a useful tool for text processing ... and also for writing lexers ...
- ◆ lex works by mapping regular expressions to NFAs, and then mapping them to DFAs.
 - Automates the task of removing non-determinism.
- ◆ By hand or by machine, the choice is yours!

59

Lab Project (step 1):

- ◆ Implement a lexer for MiniJava, using either JFlex or a handwritten solution.
- ◆ A short main method shall call the lexer repeatedly, and print “Lexical analysis succeeded/failed” depending on whether lexical errors were found
- ◆ The lexer shall handle the MiniJava test suite correctly (see course web page)
- ◆ The MiniJava syntax is described in Appel, Appendix 1, but we add one exception:
- ◆ No /* nested /* comments */ necessary! */

60

Homework assignment 2:

- ◆ See course web page
- ◆ Deadline: January 30, 2.45pm

- ◆ Recall: homework assignments should be completed individually!
- ◆ Might generate 1 bonus point on the final exam
- ◆ Report via email

61