Compiler Techniques

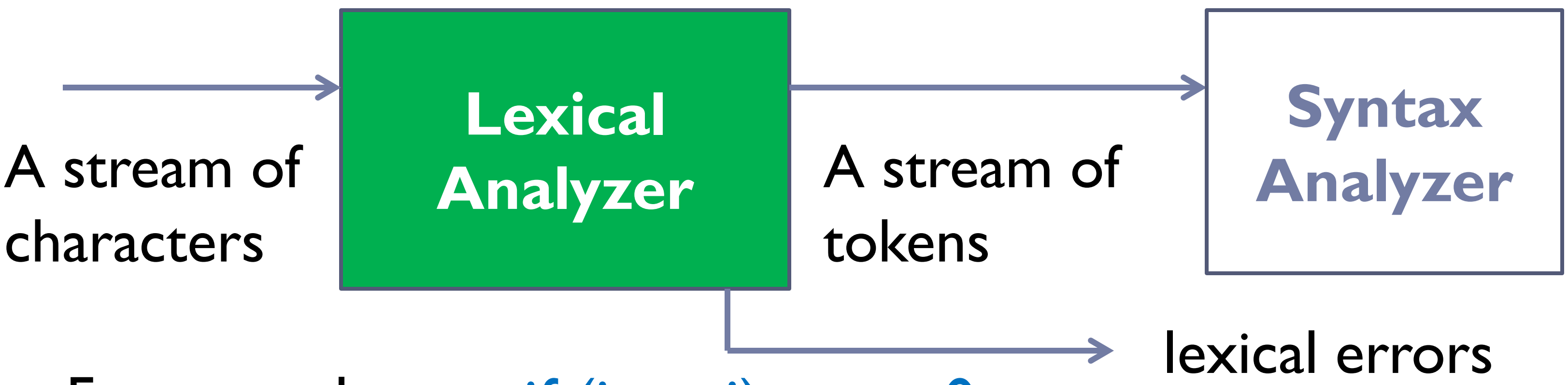
2. Lexical Analysis

Huang Shell Ying

Lexical Analysis

back

- The lexical analyzer (a.k.a. "lexer" or "scanner") transforms the input program from a sequence of characters into a sequence of tokens.



For example, if (i == j) z = 0;

\t	i	f		(	i	=	=	j	)	\t	z		=		0	;	…

IF	LPAR	ID	EQ	ID	RPAR	ID	ASSIGN	INTLITERAL	…

A Lexer in Java

- In Java, a lexer could be implemented as a class like this:

```
public class Lexer {
    public class Token {    // class for representing tokens
        int type;    // integer coded token type
        String lexeme;
    }

    private final String source; // input string to be lexed

    private final int curpos; // current position inside input

    / / called by the parser, return token starting at position curpos
    public Token nextToken() { ... }
}
```

back

Tokens

- A token is given by its type (such as IDENTIFIER) and its lexeme.
- A lexeme is a particular instance of a token type.

Token type	lexeme	Token type	lexeme
WHILE	"While"	MINUS	"-"
IF	"if"	ASSIGN	"="
LPAREN	"("	NEQ	"!="
RPAREN	")"	INTLITERAL	"10", "123","0"
IDENTIFIER	"a2", "i", "f"	WHITESPACE	"\t", " ", "\n"

back

- Some token types (like LPAREN) have a single lexeme, i.e. a single string is recognized as this token type.

Tokens

- Some token types (like IDENTIFIER) have many lexemes, i.e. a set of strings are recognized as this token type.
- What is a string?
  - An ***alphabet*** is a finite set of characters.
  - A ***string over an alphabet Σ*** is a finite sequence of characters drawn from Σ.
- WHITESPACE and COMMENT tokens are discarded by the lexer (not output by the lexer).
- Different programming languages have different tokens.

Building a Lexical Analyser

Step 1: Define a finite set of tokens and describe which strings belong to each token

- Tokens describe all items of interest
- Choice of tokens depends on language, design of parser

Step 2: Implement the lexer: An implementation must do two things:

1. Recognize substrings corresponding to tokens
2. Return the type and the lexeme of the token

Automatically Generating a Lexer

- Implementing a lexer by hand is tedious and error-prone.
- A better alternative is to use a lexer generator that automatically generates a lexer implementation from a high-level specification:

Slide 3

Token specification → **Lexer Generator** → A lexical analyzer

back

In this course, we use the JFlex lexer generator, which generates a lexer in Java. The specification describes tokens by *regular expressions*.

Regular Expressions

Slide 4

- Regular expressions are a convenient way to specify various simple (possibly infinite) **sets of strings**.
- Regular expressions are widely used in computer applications other than compilers, e.g. Unix utility ‘grep’ uses them to define search patterns in files.
- A regular expression **defines the structure** of a set of strings. This set of strings forms a token class.
- What do we need in order to define the structures of various token classes?

For example, how do we define the following token classes?

Regular Expressions

Input text string	Token type output	Input text string	Token type output
"("	LPAREN	"-"	MINUS
")"	RPAREN	"="	ASSIGN
"\t", " ", "\n"	WHITESPACE	"+", "-"	SIGN
"!="	NEQ	"While"	WHILE
"a2", "i", "f", "x0y0"	IDENTIFIER	"10", "123", "0"	INTLITERAL

- The definition of regular expressions starts with **a finite character set**, or **vocabulary** (denoted Σ).
  E.g. The Σ of C programming language is the set of ASCII characters.

Alphabet in slide 5

Regular Expressions

Definition of regular expressions:

- $\varnothing$ is a regular expression denoting the empty set, i.e. the set containing no strings.
- λ is a regular expression denoting the empty string, i.e. "".
- The symbol ***s*** is a regular expression denoting {*s*}: a set containing the single symbol $s \in \Sigma$.
  For example, **'<'** specifies {"<"};
- If ***A*** and ***B*** are regular expressions, then ***A|B*** are regular expressions denoting the set of strings which are either in *A* or in *B*. **|** is the **alternation operator**.
  For example, **'+' | '-'** specifies {"+", "-"};

Regular Expressions

- If ***A*** and ***B*** are regular expressions, then ***A·B*** are regular expressions denoting the set of strings which are the **concatenation** of one string from *A* and one string from *B*.
  For example, **‘:’ · ‘=’** specifies {“:=”}

- If ***A*** is a regular expression then ***A**** is a regular expression representing all strings formed by the concatenation of **zero or more** selections from *A*. The operator * is called the ***Kleene closure operator***.
  For example, **a*** specifies {“”, “a”, “aa”, “aaa”, …}

Additional forms or operators

- To save ink, we usually omit the dot for concatenation.
  For example, **'<' '='** specifies {"<="}
- For single characters, we often omit the quotation marks.
- A^+ is a regular expression representing all strings formed by the concatenation of **one or more** selections from A.
  $A^* = A^+ | \lambda$ and $A^+ = AA^*$.

  For example, $(0|1|2|3|4|6|7)^+$ specifies octal integers
- If k is a constant, A^k is a regular expression representing all strings formed by the **concatenation of k** (possibly different) strings from A.
  For example, $(0|1)^8$ specifies strings of 8 binary digits

Additional forms or operators

- A character class, delimited by **[** and **]**, represents **a single character** from the class. Ranges of characters are separated by a **-**.

  For example, **([0-9])$^+$** specifies decimal integers

- $\overline{A}$ or **Not(A)** represents $(\Sigma$ - $A)$, i.e. all characters in Σ not included in A.

  For example, **'\' '\' (Not('\n'))* '\n'** specifies single line comments

- **?** Is the **optional** choice operator.

  For example, **(+ | -)? ([0-9])$^+$** specifies signed integers

Operator precedence

- Operator precedence in decreasing order:

  (R), R^*, R_1R_2, $R_1|R_2$

Examples:

Regular expression	Set of strings defined
$\varnothing$	{ }
λ	{ “” }
0	{“0”}
0 | 1	{“0”, “1”}
0 1	{“01”}

Examples

Regular expression	Set of strings defined
(0 | 1) 0	{"00","10"}
0 | 10	{"0","10"}
0*	{"","0","00","000", ….}
$(ab)^+$	{"ab","abab","ababab", …}
ab^+	{"ab","abb","abbb", …}
[abc] or [a-c] or a|b|c	{"a","b","c"}
[a-z] ([a-z] | [0-9])*	{"a", … "z", "aa", …, "a0", …} –alphanumeric sequences starting with a lower case letter

Automata

- An **automaton** (plural: **automata** or **automatons**) is a self-operating machine.

Finite Automata

- A finite automaton we study is a ***machine*** that reads a string and decides whether it is a token specified by a regular expression.

- A finite automaton is essentially a graph, with nodes and transition edges.

- Nodes represent states and transition edges represent transitions between states.

- A finite automaton consists of the following:

1. A finite set of ***states***

a state:

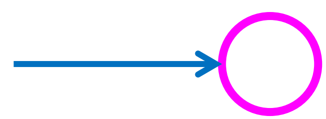
Finite Automata

2. A finite ***vocabulary***, denoted Σ

3. A set of ***transitions*** from one state to another, labeled with characters in Σ or λ

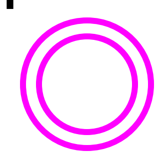
$$\xrightarrow{c \in \Sigma \text{ or } \lambda}$$

4. A special state with no predecessor state, called the ***start*** state



a start state

5. A subset of the states called the ***accepting***, or ***final*** states

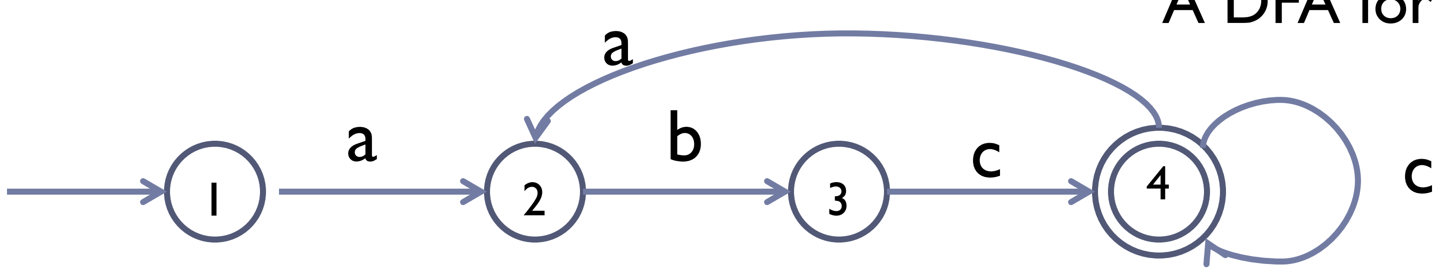
an accepting state: 

Finite Automata

- Finite automata come in two flavours:
  - **Deterministic finite automata** (DFA)
  - **Nondeterministic finite automata** (NFA)
- **Deterministic finite automata** (DFA)
  - do not allow λ to label a transition.
  - do not allow the same character to label transitions from one state to several different states.

Finite Automata

- We can represent either an NFA or DFA by a **transition graph**:



A DFA for $(abc^+)^+$

- or by a **transition table**:

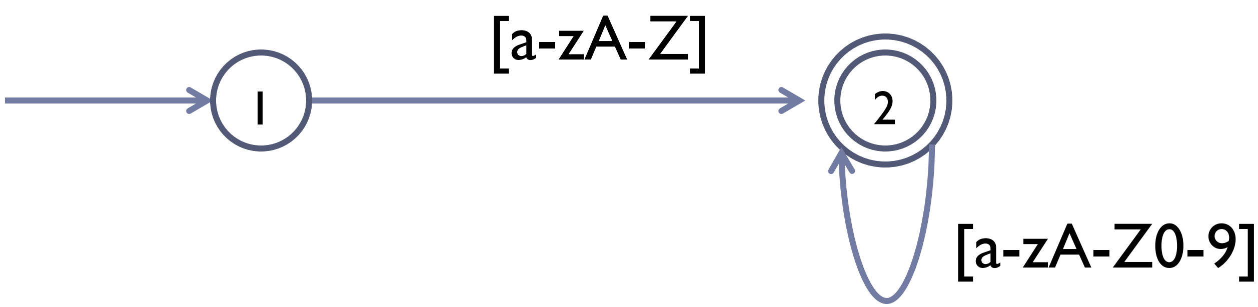
	a	b	c	others
1, start	2	0	0	0
2	0	3	0	0
3	0	0	4	0
4, final	2	0	4	0

State 0: error

Deterministic Finite Automata (DFA)

A DFA for identifiers: [a-zA-Z][a-zA-Z0-9]*

By a transition diagram



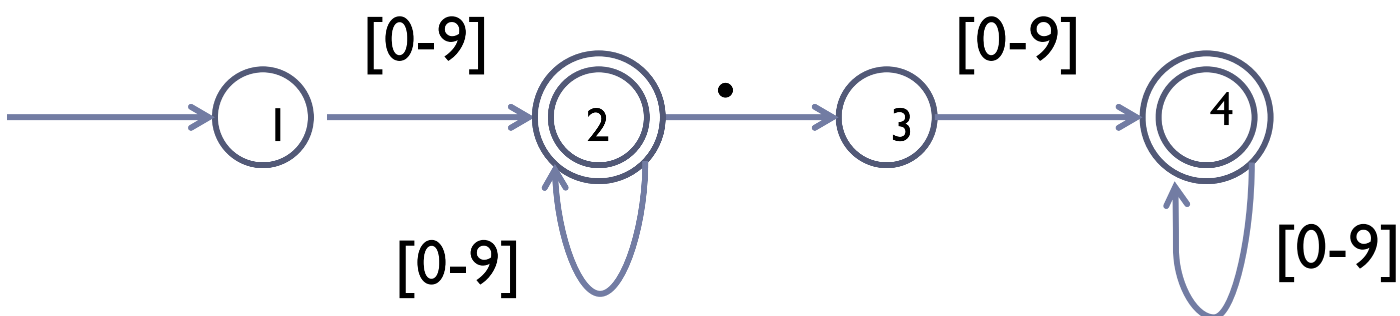
By a transition table

	a …	z	A…	Z	0…	9	others
1, start	2…	2	2…	2	0…	0	0
2, final	2…	2	2…	2	2…	2	0

Deterministic Finite Automata (DFA)

A DFA for numbers: $([0\text{-}9])^+ \mid ([0\text{-}9])^+ \text{'.'} ([0\text{-}9])^+$

By a transition diagram



By a transition table

	[0-9]	.	others
1, start	2	0	0
2, final	2	3	0
3	4	0	0
4, final	4	0	0

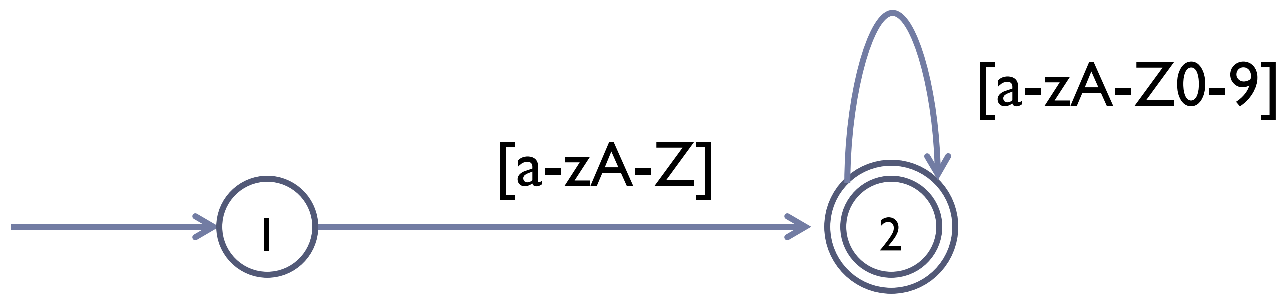
Coding the Deterministic Finite Automata

- A DFA can be coded in a **table-driven** form:

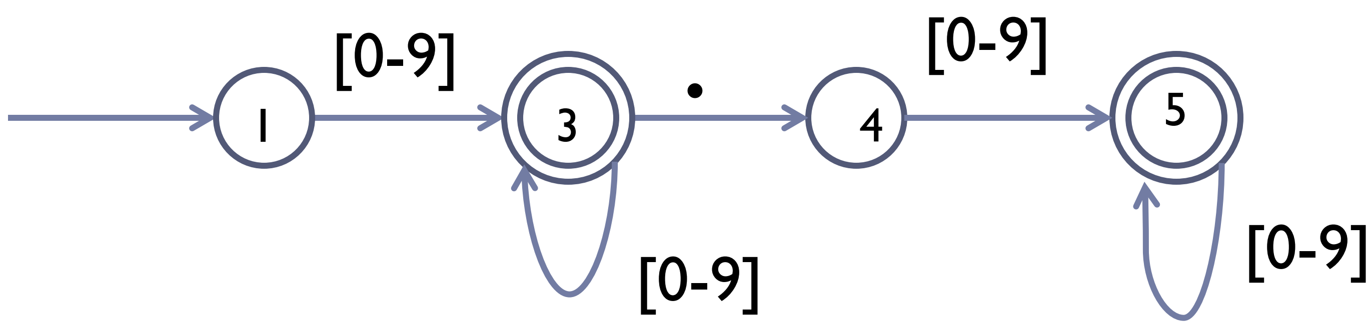
```
currentChar = read();
state = startState;
while true do
    nextState = T[state, currentChar];
    if (nextState == error)  then break;
    state = nextState;
    currentChar = read()
if (state in acceptingStates)
    then /* return or process the valid token */
    else /* signal a lexical error */
```

back

Example



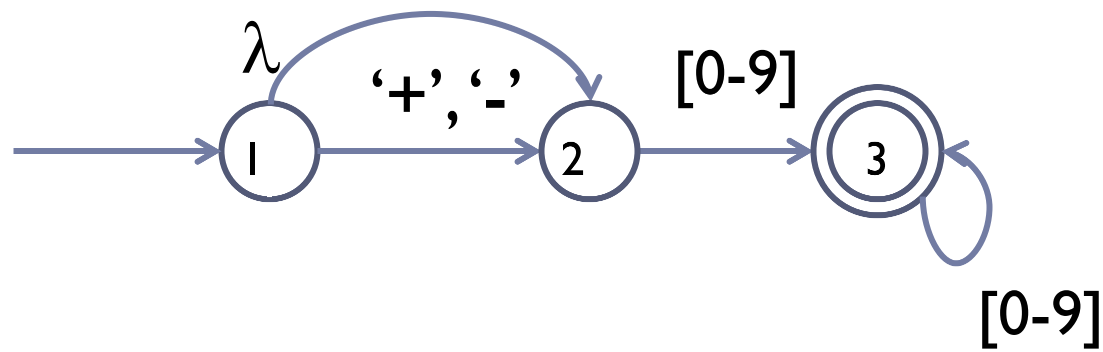
Input: “max2”. What happens for the input “m”, for “2m”?



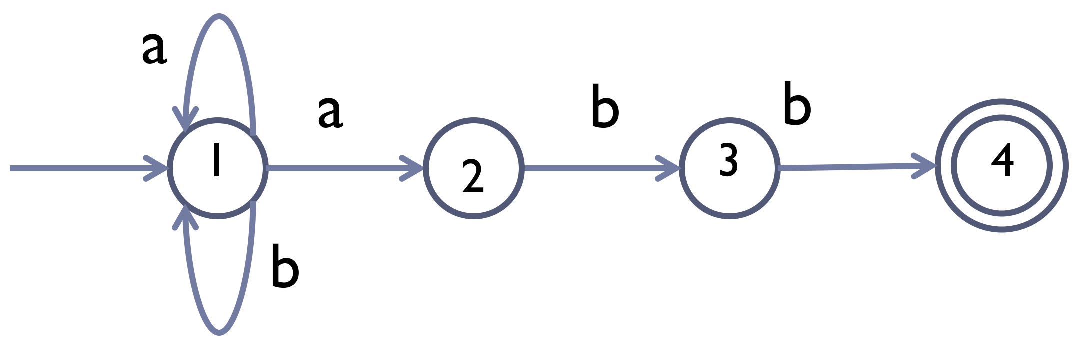
Input: “0.5”. What happens for the input “6”, for “6.”?

Nondeterministic Finite Automata (NFA)

An NFA for integer constants: (‘+’ | ‘-’ | λ) [0-9]$^+$

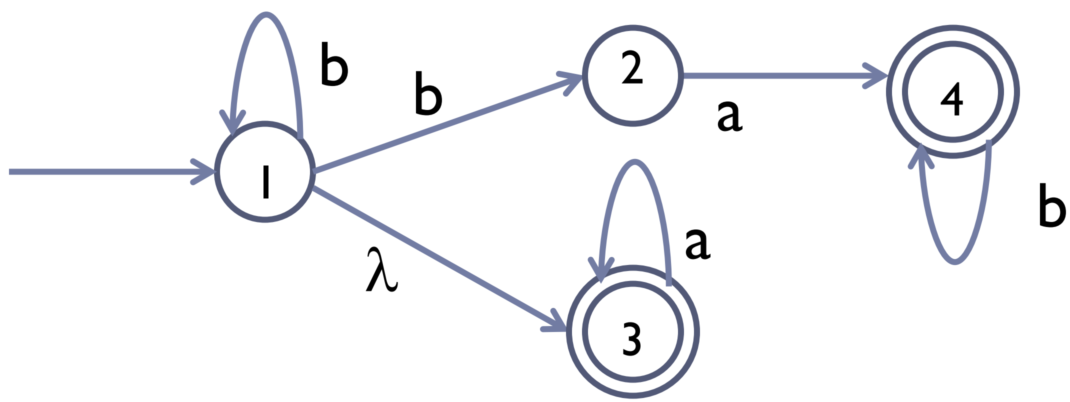


An NFA for (a|b)*abb



Nondeterministic Finite Automata (NFA)

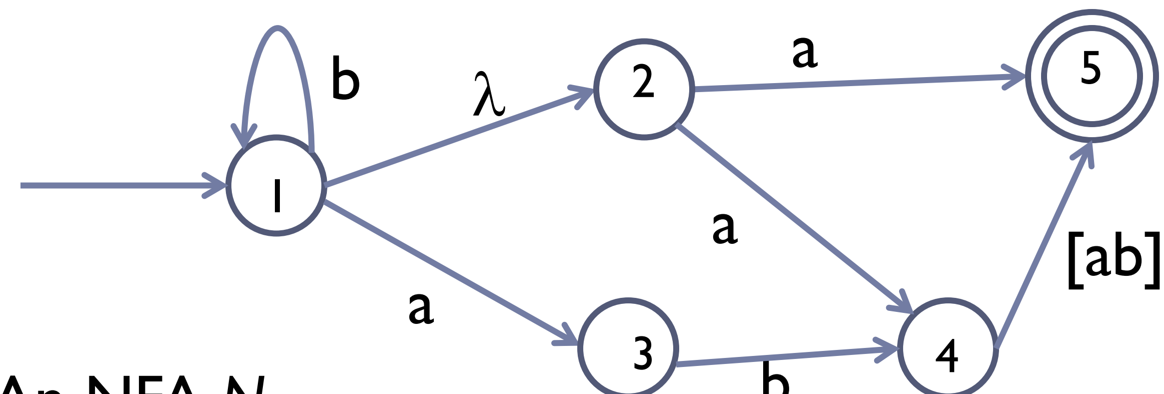
An NFA for $b^*a^* \mid b^+ab^*$



What is the transition table like for this NFA?

Conversion of an NFA to a DFA

- The transformation from an NFA ***N*** to a DFA ***D*** can be done by a **subset construction** algorithm.
- The algorithm associates each state of *D* with a ***set*** of states of *N*.
- The start state of *D* is the set of all states to which *N* can **transition without reading** any character.



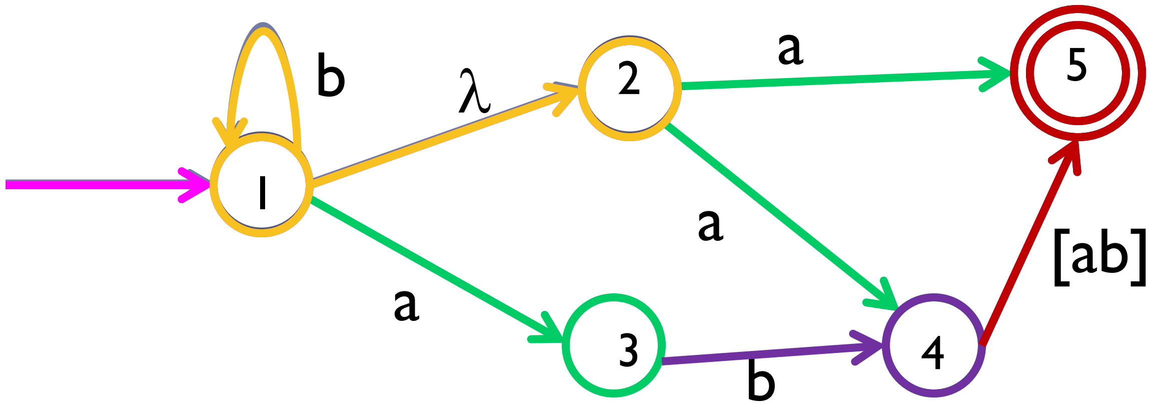
An NFA *N*
Start state: 1

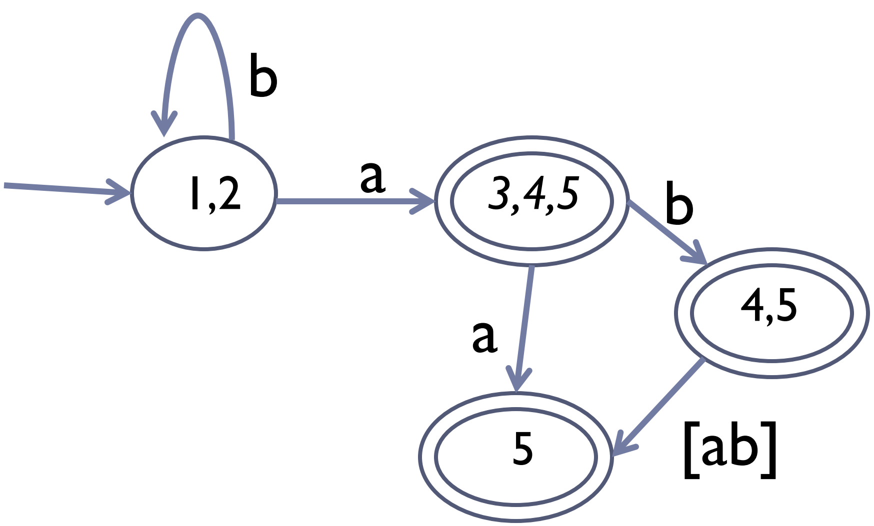
The DFA *D*
Start state: {1,2}

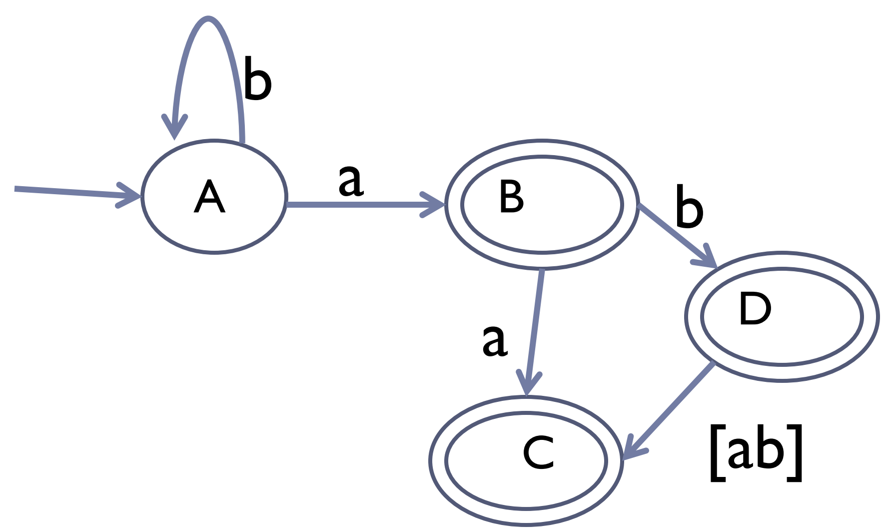
Conversion of an NFA to a DFA

- Put each state ***S*** of ***D*** on a work list when ***S*** is created.
- For each state ***S*** = **{*n*1, *n*2, …}** on the work list and each character ***c*** $\in \Sigma$, we compute the successor states of ***n*1, *n*2, …** under ***c*** in ***N*** and obtain a set **{*m*1, *m*2, …}**. Then we include the λ-successors of ***m*1, *m*2, …** in ***N***.
- The resulting set of NFA states is included as a state ***T*** in ***D***, and a transition from ***S*** to ***T***, labeled with c is added to ***D***.
- We continue to add states and transitions to ***D*** until all possible successors to existing states are added.
- An accepting (final) state of ***D*** is any set that contains an accepting (final) state of ***N***.

Conversion of an NFA to a DFA







Conversion of an NFA to a DFA

Function makeDFA(NFA N) **// returns DFA *D***

{ $D.startState$ = recordState({$N.startState$});

for each S in $workList$

$workList = workList - \{S\}$;

for each c in Σ

$D.T(S, c)$ = recordState($\bigcup_{s \in S} N.T(s, c)$);

The set of NFA states to transition to under c from states in *S*

D.acceptStates = $\{S \in D.\text{states} \mid S \cap N.\text{acceptStates} \neq \varnothing\}$

}

e.g. $D.T(\{1,2\}, \text{'a'})$ = recordState($N.T(1, \text{'a'}) \cup N.T(2, \text{'a'})$)

Conversion of an NFA to a DFA

Function recordState(S)

{ S = close(S, $N.T$);

if ($S \notin D.states$) then

$D.states = D.states \cup \{S\}$;

$workList = workList \cup \{S\}$;

return S;

}

// the input parameter S is the set of NFA states

// the output S is the set of NFA states which is the

// λ-closure of the input – a new DFA state

e.g. S = {1}

At the end of recordState({1})

$D.states$ = { {1, 2} }

$workList$ = { {1, 2} }

recordState({1}) returns {1, 2}

Conversion of an NFA to a DFA

Function close(*S*,*T*) **// compute states that can be reached**

{ *ans* = *S*; **// after only λ transitions**

repeat

changed = false;

for each $s \in ans$

for each $t \in T(s, \lambda)$

if ($t \notin ans$) then

$ans = ans \cup \{t\}$; *changed* = true;

until not *changed*;

return *ans*; **// the λ-closure of *S***

}

	λ	a	b	.
1, start	2	3	1	0
2, final	\	4, 5	0	0
3	\	0	4	0
4	\	5	5	0
5, final	\	0	0	0

e.g. *S* = {1}
Close({1}, *T*) returns {1, 2}

Steps from regular expressions to DFAs

Regular expression

Translating a Regular Expression into an NFA

NFA

Conversion of an NFA to a DFA

DFA

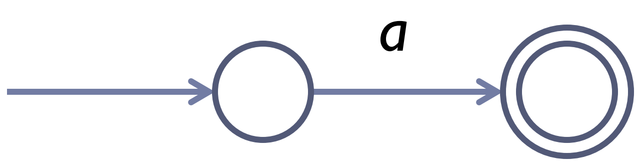
Translating Regular Expressions into NFAs

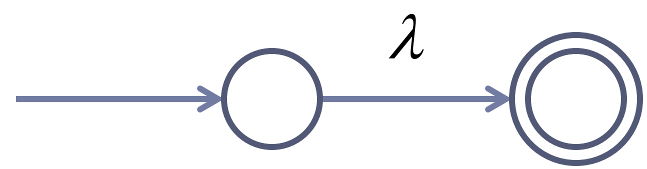
- The McNaughton-Yamada-Thompson method to construct NFAs from regular expressions produces NFAs that accept the same languages.
- An NFA constructed has the following properties:
  1) The NFA has at most twice as many states as there are operators and operands in the regular expression;
  2) The NFA has one start state and one accepting state. The accepting state has no outgoing transitions and the start state has no incoming transitions.
  3) Each state of the NFA other than the accepting state has either one outgoing edge on a symbol in $\Sigma \cup \{\lambda\}$ or two outgoing edges, both on λ.

back

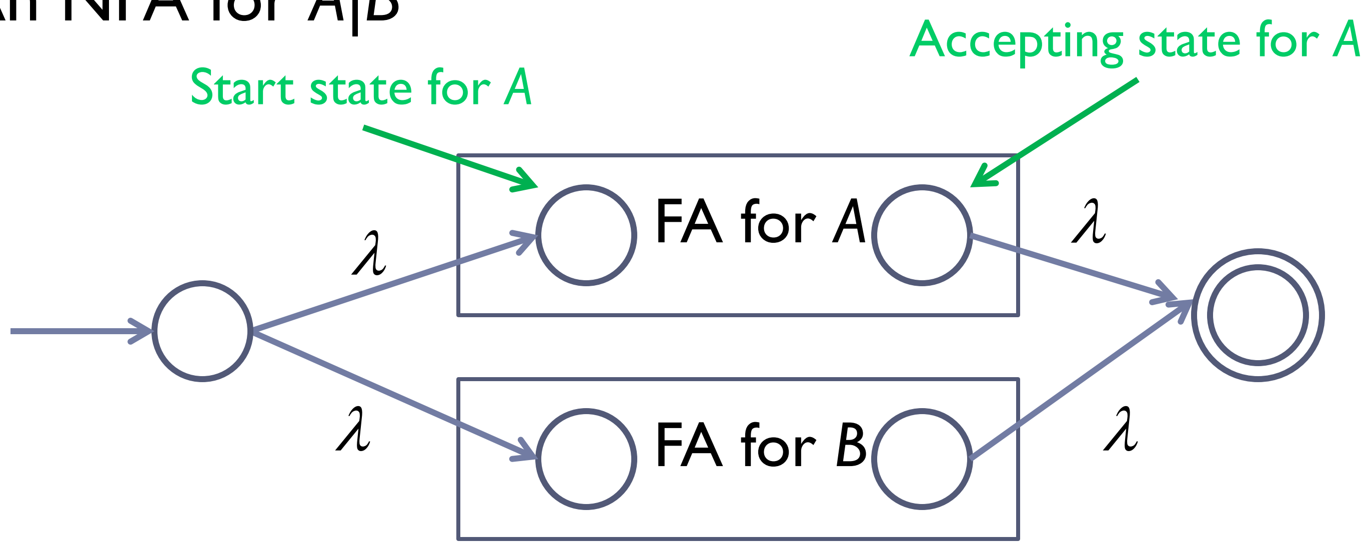
Translating Regular Expressions into NFAs

- NFAs for *a* and λ



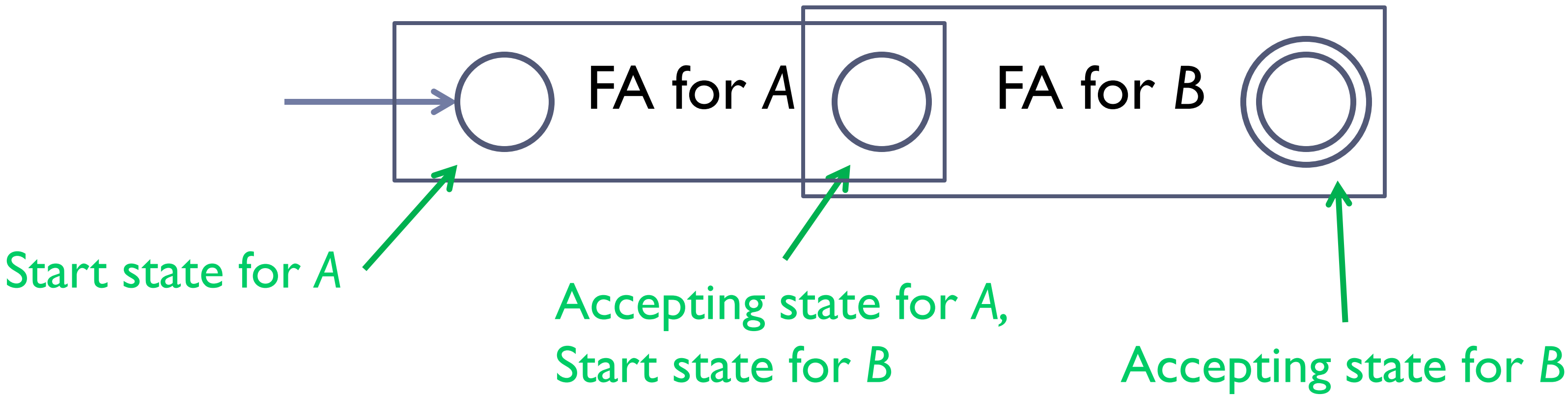


- An NFA for *A*|*B*

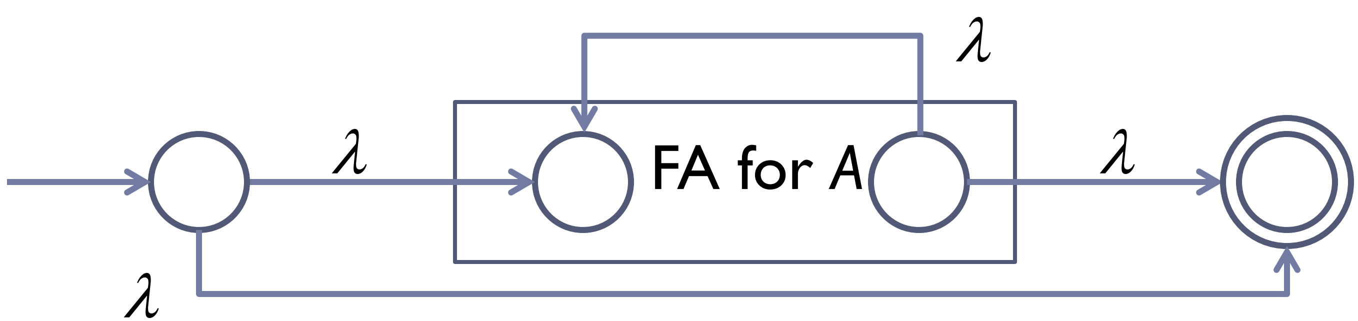


Translating Regular Expressions into NFAs

- An NFA for *AB*



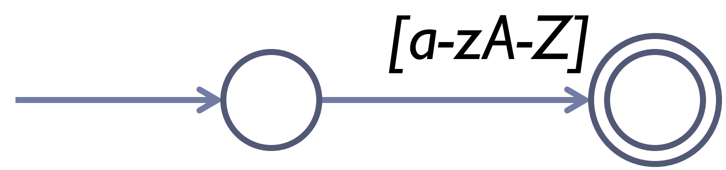
- An NFA for *A**



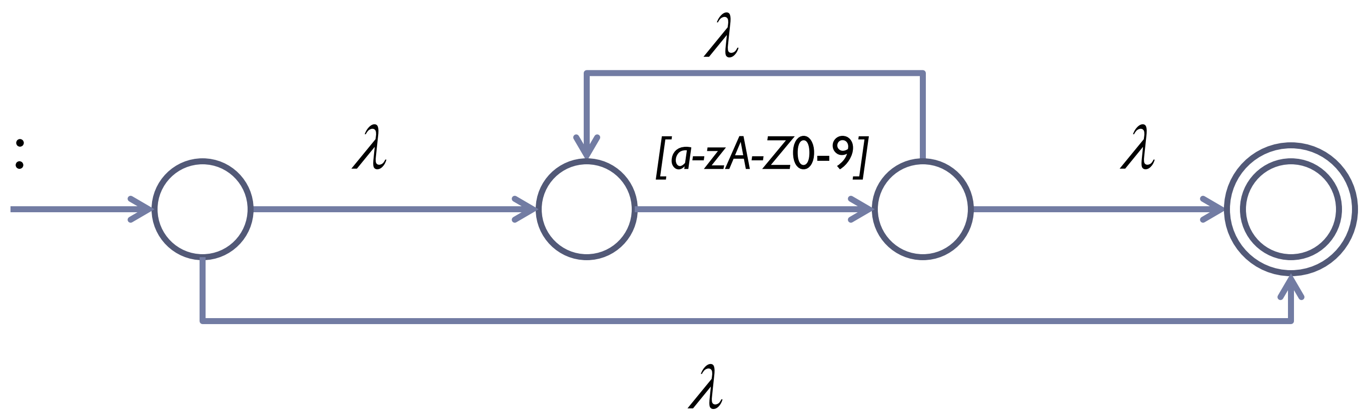
Translating Regular Expressions into NFAs

Example: [a-zA-Z] [a-zA-Z0-9]*

[a-zA-Z] :

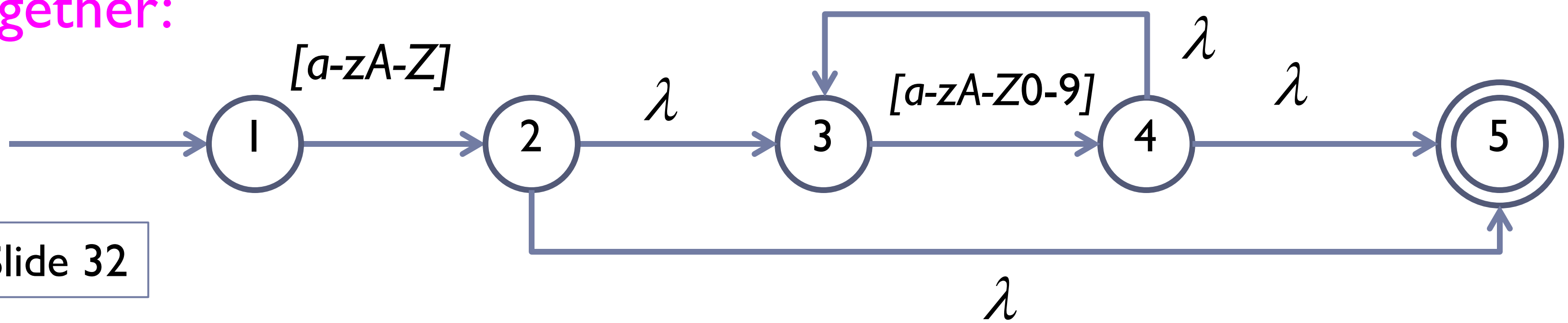


[a-zA-Z0-9]* :



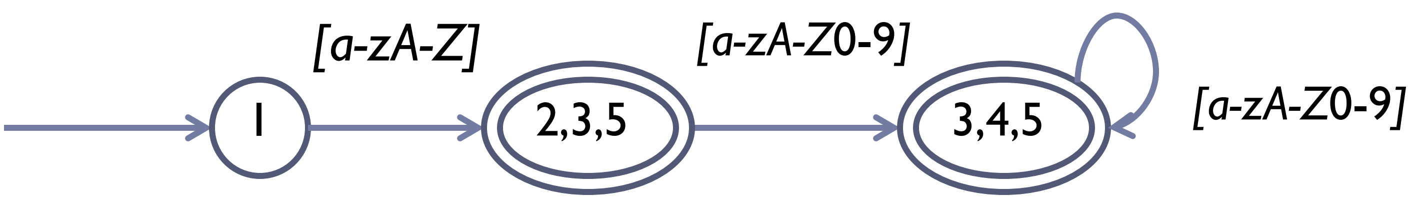
Translating Regular Expressions into NFAs

Together:

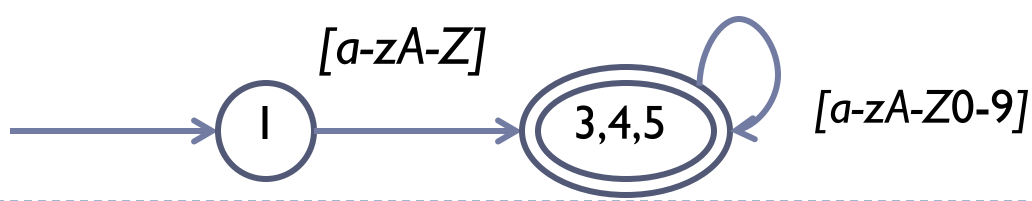


Slide 32

After subset construction:



After optimization (not covered by the course):



Lexer Generators

- A very popular scanner generator, **Lex**, was developed by M.E. Lesk and E.Schmidt of AT&T Bell Laboratories. It is distributed as part of the Unix system.
- It was used primarily with programs written in C or C++ running under Unix.
- **Flex** is a widely used, freely distributed reimplementation of Lex that produces faster and more reliable scanners.
- **JFlex** is a similar tool for use with Java.

Lexer Generators

- A scanner specification that defines the tokens and how they are to be processed is presented to JFlex.
- JFlex generates a complete scanner coded in Java.
- This scanner is combined with other compiler components (syntax analyzer, etc) to create a complete compiler.

- A lexer generator takes as its input a list of rules:

  R_1 $\{A_1\}$
  R_2 $\{A_2\}$
  R_3 $\{A_3\}$
  ……

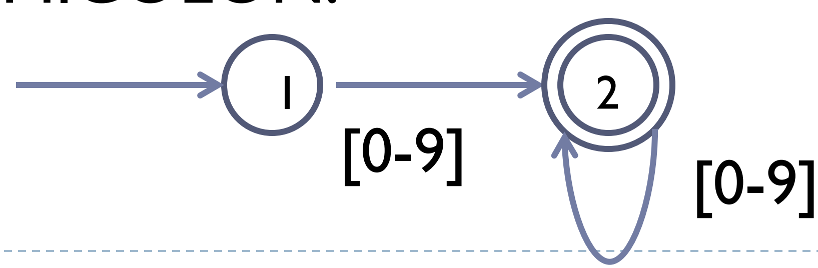
  where Ri are regular expressions and Ai are snippets of Java code.

Lexer Generators

- Intended meaning:
  - read input string one character at a time;
  - whenever the input read so far matches some R_i, execute the corresponding action A_i;
  - then continue reading the input.

Slide 23

- The longest possible match between the input stream and R_i is chosen when matching R_i. For example, "123;" will be matched as one token with type INTEGER and lexeme "123" and another token SEMICOLON.



JFlex Regular Expression Syntax

- Concatenation is written without the dot ·
- alternation, repetition and non-empty repetition are written as a|b, a* and a+
- negation Not(a) is written as !a
- single characters can be written without quotes, but only if they don't have special meaning (like *, +)
- character classes:
  - [0-9] is a character range (no quotes around 0 and 9!)
  - [123] means '1' | '2' | '3'
  - can be combined: [0-9abc]
  - if the character class starts with ˆ, it is a negated character class: [ˆ0-9]

JFLex definition file

- This is the input file to the scanner generator.
- The general structure of JFLex definition files has three sections:

Slide 3

User code -- **copied to lexer.java before the lexer class declaration**

%%

Declarations -- **macro declarations: abbreviations to make lexical specifications easier to read and understand**

%%

Regular expression rules

Example of a simple arithmetic expression

In JFlex, when we want to use an abbreviation like Digit we have to surround it by curly braces

Slide 3

```
%%
Digit = [0-9]
Alpha = [a-zA-Z_]
%%
"+"                     { return new Token(PLUS, yytext()); }
"-"                     { return new Token(MINUS, yytext()); }
"*"                     { return new Token(MULT, yytext()); }
"/"                     { return new Token(DIV, yytext()); }
"("                     { return new Token(LPAREN, yytext()); }
")"                     { return new Token(RPAREN, yytext()); }
("+"|"-")?{Digit}+      { return new Token(INTLIT, yytext()); }
{Alpha}({Alpha}|{Digit})* { return new Token(IDENTIFIER, yytext()); }
[ \t\n]                 { /* skip blank, tab and end of line chars */}
```

Example of a simple arithmetic expression

- Note that the method **yytext** (provided by JFlex) returns the text matched by the regular expression.
- From this specification, JFlex generates a class Lexer whose skeleton is:

```
public class Lexer {
        public Lexer(InputStream in) {
        ...
        }
        public Token nextToken() {
        ...
        }
}
```

Resolving ambiguities and error handling

- There often is more than one way to partition a given input string into tokens. For example "-12".
- JFlex disambiguates this case using the longest match rule: if two different regular expressions R_i and R_j both match the start of the input, it chooses the one that matches the longer string. E.g. "breaker" is an identifier
- If both R_i and R_j match the same number of characters, it prefers the one that occurs earlier in the specification. This is useful to introduce "catch all" rules for error reporting. E.g. "if" matches both a keyword and an identifier

Resolving ambiguities and error handling

- For instance, we could add the following rule at the end of our specification:

```
.    { System.err.println("Unexpected character " +
                     yytext() + " " + "at line " +
               yyline + ", column " + yycolumn); }
```

- The dot character matches any input symbol.
- This rule will only fire if none of the other rules do.
- The action prints an explanatory error message and skips over the offending character.
- **yyline** and **yycolumn** are provided by JFlex, which contain information about the current source position.

Processing Reserved Words

- Virtually all programming languages have **keywords** which are reserved. They are called ***reserved words***.
- Keywords also match the lexical syntax of ordinary identifiers.
- One possible approach: create distinct regular expressions for each reserved word before the rule for identifiers. For example,

```
%%
…
"if"                      {return new Token(IF,"");}
"then"                    {return new Token(THEN,""):}
…
{Alpha}({Alpha}|{Digit})* { return new Token(IDENTIFIER,
                            yytext()); }
```

Processing Reserved Words

- This approach increases the size of the transition table significantly.
- An alternative approach: after an apparent identifier is recognized, look up the lexeme in a keyword table to see if it is a keyword. For example,

```
{Alpha}({Alpha}|{Digit})* { word = yytext();
                            code = lookUP(keywordTable, word);
                            if (code < keywordTable.size)
                              return new Token(code, “”);
                            else return(IDENTIFIER, word);
                          }
```

Slide 3

Assuming 0 to keywordTable.size-1 are the token type codes

Some Review Questions/tasks

1. What does a lexer (lexical analyzer) do?
2. What are regular expressions used for in a compiler?
3. What are the differences between Deterministic Finite Automata and Nondeterministic Finite Automata?
4. Draw the transition table for the two DFAs on slide 24.
5. Answer the questions on slide 24. Give the sequence of states when the two DFAs process the various input strings respectively.
6. What is the Subset Construction algorithm used for?
7. What is the McNaughton-Yamada-Thompson method used for?
8. What is a lexer generator used for?