COMPILER (CSE 4120)

(Lecture 2: Lexical Analysis)

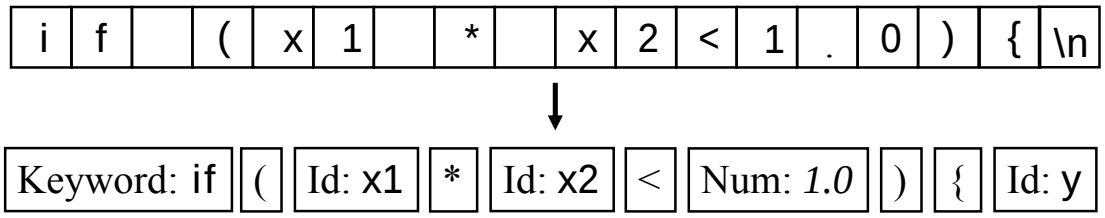
Sungwon Jung

Mobile Computing & Data Engineering Lab
Dept. of Computer Science and Engineering
Sogang University
Seoul, Korea
Tel: +82-2-705-8930
Email : jungsung@sogang.ac.kr

Dept. of Computer Science at Sogang University

What is Lexical Analysis ?

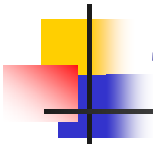
- Converts character stream to token stream
  - Token
    - A sequence of characters that represents a unit of information in the source program
    - "sequence of characters with a collective meanings"
    - e.g., keywords (if, while, …), identifiers, special symbols (+, *, …)
  - Get rid of white spaces, comments
  - if (x1*x2 < 1.0) { y = x1; }



Lexical Analysis

- Tokens – logical entities that are usually defined as an enumerated type
  - typedef enum {IF,THEN,ELSE,PLUS,MINUS,NUM,ID,...} TokenType;
  - C.f., IF vs. if
    - E.g., Token IF represents the strings of characters "if" (c.f., pattern/lexeme)
    - Lexeme (or string value): the string of characters represented by a token
  - Categories of tokens
    - Reserved words, Special symbols, Numbers, Identifiers

Dept. of Computer Science at Sogang University



Token /Pattern /Lexeme

- Token: a name for a class of strings in the input
  - e.g., identifier, keyword, operator, constant, etc.
- Pattern: a rule that describes the set of strings associated wit a token
  - e.g., "a letter followed by zero or more letters, digits, or underscores"
- Lexeme: a sequence of characters matched by the pattern for a token
  - e.g., ***32894*** for NUM token, ***counter*** for ID token
- C.f., Attribute: any value associated to a token
  - e.g., the string value is an example of an attribute

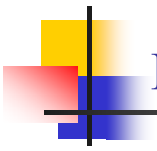
Terminologies (Cont'd)

- A few more examples:

Token	Sample Lexemes	Pattern
while	while	while
integer_constant	32984, 1093, 0	$[0–9]^+$
identifier	buffer_size	$[a–z\ A–Z]^+$

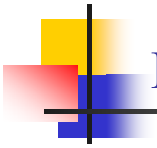
Problems

- How to describe tokens?
  - 2.e0, 20.e-01, 2.000
- How to break text up into tokens?
  - if (x == 0) a = x << 1;
  - iff (x == 0) a = x < 1;
- How to tokenize efficiently?
  - tokens may have similar prefixes
  - want to look at each character 1 time



How to describe Tokens?

- How do we compactly represent the set of all lexemes corresponding to a token?
  - For instance:
    - The token *integer_constant* represents the set of all integers: that is, all sequence of digit (0, 1, …, 9), preceded by an optional sign (+ or -)
- Obviously, we cannot simply enumerate all lexemes
  - Use Regular Expressions
- Programming language tokens can be described as ***regular expressions***
- A regular expression R describes some set of strings L(R)
  - L(abc) = {"abc"}



Regular Expressions

- Notation to represent (potentially) infinite sets of strings over alphabet Σ
  - **a**: stands for the set {**a**} that contains a single string **a**
  - **ab:** stands for the set {**ab**}that contains a single string **ab**
  - **a|b**: stands for the set {**a, b**} that contains two strings **a** and **b**
  - **a***: stands for the set {**ε, a, aa, aaa, …**} that contains all strings of zero or more **a**'s
    - **ε** stands for the empty string

Regular Expressions (Cont'd)

- Examples of Regular Expressions over {a, b}
  - (**a**|**b**)*: set of strings with zero or more **a**'s and zero or more **b**'s:
    - **{ε, a, b, aa, ab, ba, bb, aaa, aab, …}**
  - (**a***__b__*): set of strings with zero or more **a**'s and zero or more **b**'s such that all **a**'s occur before any **b**:
    - **{ε, a, b, aa, ab, bb, aaa, aab, abb, …}**
  - (**a***__b__*)*: set of strings with zero or more **a**'s and zero or more **b**'s:
    - **{ε, a, b, aa, ab, ba, bb, aaa, aab, …}**

Language of Regular Expressions

- Let R be the set of all regular expressions over Σ. Then
  - Empty String: $\varepsilon \in R$
  - Unit Strings: $\alpha \in \Sigma \Rightarrow \alpha \in R$
  - Concatenation: $r_1, r_2 \in R \Rightarrow r_1 r_2 \in R$
  - Alternative: $r_1, r_2 \in R \Rightarrow (r_1 \mid r_2) \in R$
  - Kleene Closure: $r \in R \Rightarrow r^* \in R$

Regular Definitions

- Assign "names" to regular expressions
  - For example,
    - digit → 0 | 1 | 2 | … | 9
    - natural → digit | digit*
- Shorthands:
  - **a**+: set of strings with one or more occurrences of **a**
  - **a**?: set of strings with zero or one occurrence of **a**
  - Example:
    - integer → (+| –)?digit+

Examples of Regular Definitions

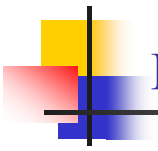
- float → integer . fraction
- integer → (+|–)? natural
- fraction → natural exponent?
- exponent → (E | e) integer
- natural → digit+
- digit → 0 | 1 | . . . | 9

Semantics of Regular Expressions

- Semantic Function £ : Maps regular expressions to sets of strings
  - $£(\varepsilon) = \{\varepsilon\}$
  - $£(\alpha) = \{\alpha\} \quad (\alpha \in \Sigma)$
  - $£(r_1 | r_2) = £(r_1) \cup £(r_2)$
  - $£(r_1 r_2) = £(r_1) \cdot £(r_2)$
  - $£(r^*) = \{\varepsilon\} \cup (£(r) \cdot £(r^*))$

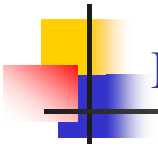
Computing the Semantics

$$
\begin{aligned}
£(ab) &= \{a\} \\
£(a \mid b) &= £(a) \cup £(b) \\
&= \{a\} \cup \{b\} \\
&= \{a, b\} \\
£(ab) &= £(a) \cdot £(b) \\
&= \{a\} \cdot \{b\} \\
&= \{ab\} \\
£((a \mid b)(a \mid b)) &= £(a \mid b) \cdot £(a \mid b) \\
&= \{a, b\} \cdot \{a, b\} \\
&= \{aa, ab, ba, bb\}
\end{aligned}
$$



Regular Definitions and Lexical Analysis

- Regular expression and definitions specify sets of strings over an input alphabet
  - They can hence be used to specify the set of *lexemes* associated with a *token*
    - Used as the pattern language
- How do we decide whether an input string belongs to the set of strings specified by a regular expression?
  - Need *Recognizers* !!!
    - Finite State Automata for Regular Language
    - Push Down Automata for Context-free Language



Recognizers

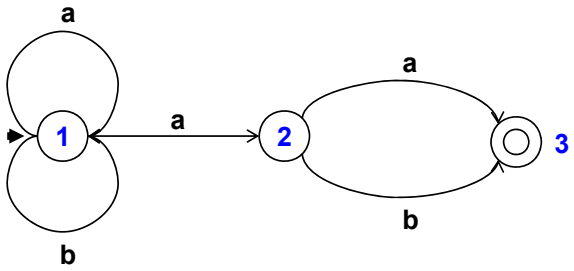
- Construct automata that recognize strings belonging to a language
  - Finite State Automata ⇒ Regular language
    - Finite State → cannot maintain arbitrary counts
  - Push Down Automata ⇒ Context-free language
    - Stack is used to maintain counter, but only one can go arbitrarily high

Finite State Automata

- Represented by a labeled directed graph
  - A finite set of states (vertices)
  - Transition between states (edges)
  - Labels on transitions are drawn from $\Sigma \cup \{\varepsilon\}$
  - One distinguished *start* state
  - One or more distinguished *final* states

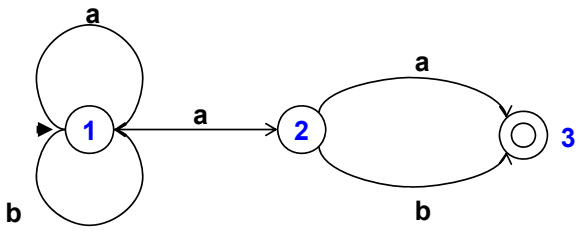
Finite State Automata : An Example

- Consider the regular expression $(a|b)^*a(a|b)$
  - £ $((a|b)^*a(a|b))$ = {aa, ab, aaa, aab, baa, bab, aaaa, aaab, abaa, abab, baaa, …}
  - The following automaton determines whether an input string belong to £ $((a|b)^*a(a|b))$

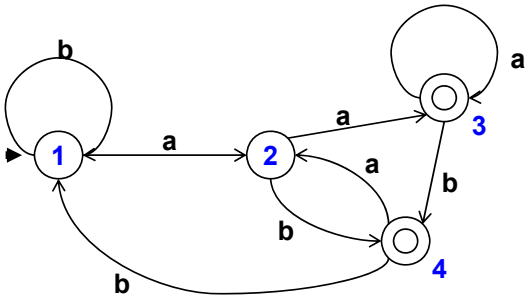


Determinism

- (a|b)*a(a|b):
  - Nondeterministic: (NFA)



  - Deterministic: (DFA)

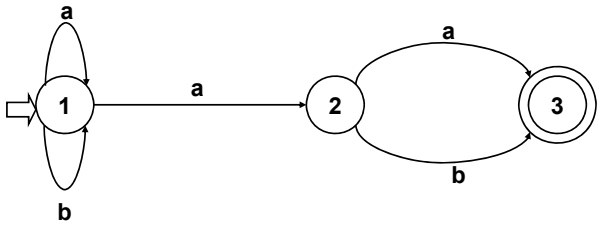


Acceptance Criterion

- A finite state automaton (NFA or DFA) accepts an input string X
  - if beginning from the start state
  - we can trace some path through automaton
  - such that the sequence of edge labels spells X
  - and end if a final state

Recognition with an NFA

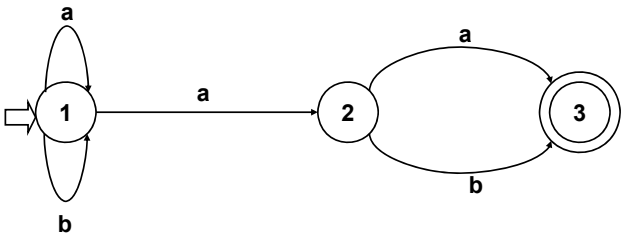
- Is abab ∈ £ ((a|b)*a(a|b)) ?



Input :		a	b	a	b	
Path 1 :	1	1	1	1	1	
Path 2 :	1	1	1	2	3	Accept
Path 3 :	1	2	3	⊥	⊥	
						Accept

Recognition with an NFA

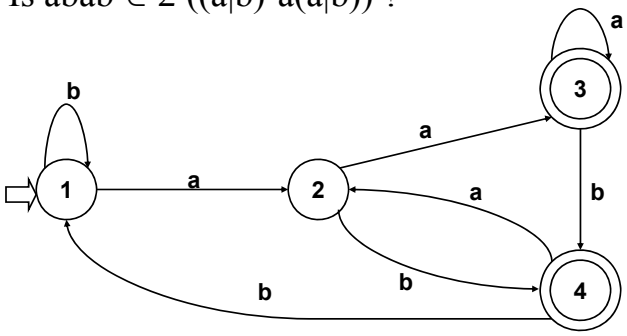
- Is aabb ∈ £ ((a|b)*a(a|b)) ?



Input :		a	b	a	b	
Path 1 :	1	1	1	1	1	
Path 2 :	1	1	2	3	⊥	
Path 3 :	1	2	3	⊥	⊥	
All Paths	{1}	{1,2}	{1,2,3}	{1,3}	{1}	Reject

Recognition with an DFA

- Is abab ∈ £ ((a|b)*a(a|b)) ?



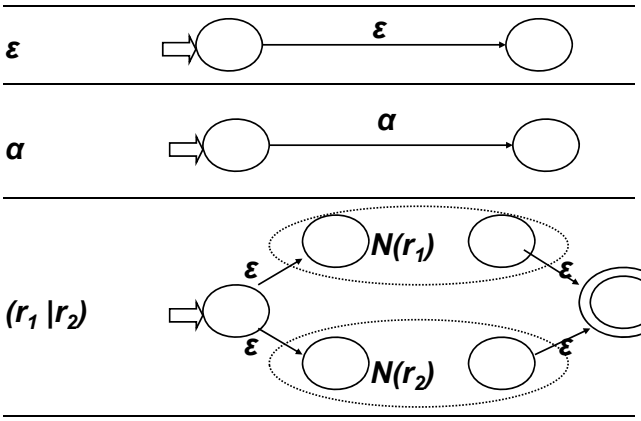
Input	**:**		**a**	**b**	**a**	**b**
Path	**:**	**1**	**2**	**4**	**2**	**4** **Accept**

NFA vs. DFA

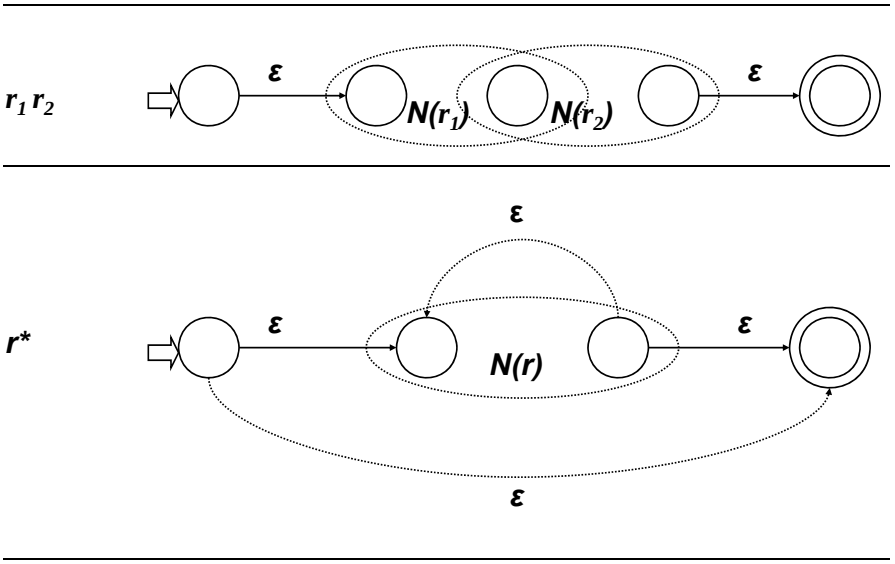
- For every NFA, there is a DFA that accepts the same set of strings
  - NFA may have transitions labeled by ε (*spontaneous transition*)
  - All transition labels in a DFA belong to Σ
  - For some string X, there may be many accepting path in an NFA
  - For all string X, there is one unique accepting path in a DFA
  - Usually, an input string can be recognized *faster* with a DFA
  - NFAs are typically *smaller* than the corresponding DFAs

Regular Expressions to NFA

- Thompson's Construction:
  - For every regular expression r, derive an NFA $N(r)$ with unique start and final states

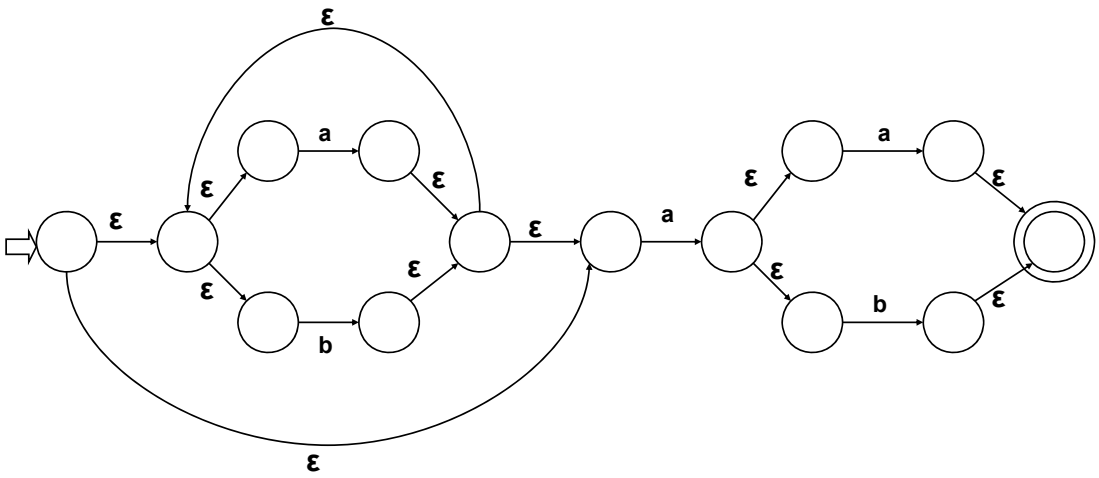


Regular Expressions to NFA (Cont'd)



Regular Expressions to NFA: An Example

$(a|b)^*a(a|b)$



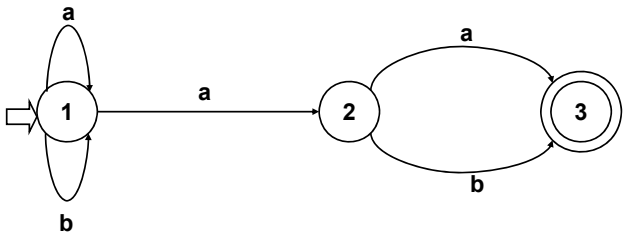
Converting NFA to DFA

- Subset construction
  - Given a set S of NFA states
    - compute S_ε = ε-clousure(S): S_ε is the set of all NFA states reachable by zero or more ε-transitions from S
    - Compute S_α = goto(S,α) :
      - S' is the set of all NFA states reachable from S by taking a transition labeled α
      - S_α = ε-clousure(S')

Converting NFA to DFA (Cont'd)

- Each state in DFA corresponds to a set of states in NFA
- Start state of DFA = ε-closure(start state of NFA)
- From a state s in DFA that corresponds to a set of states S in NFA:
  - add a transition labeled α to state s' that corresponds to a non-empty S' in NFA such that S' = goto(S, α)
- s is a state in DFA such that the corresponding set of states S in NFA contains a final state NFA
  - s becomes a final state of DFA

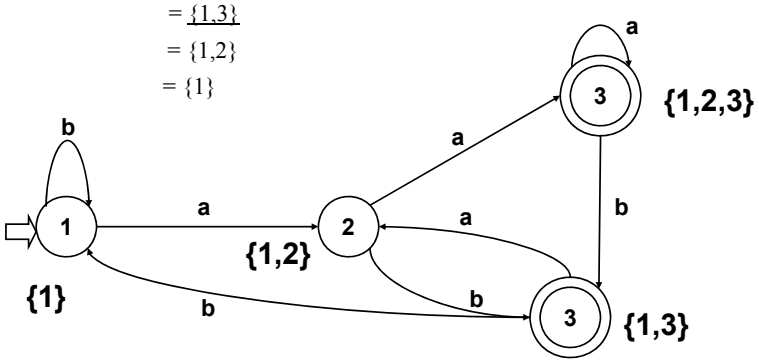
NFA → DFA: An Example



ε-closure({1})	= {1}
goto({1},a)	= {1,2}
goto({1},b)	= {1}
goto({1,2},a)	= {1,2,3}
goto({1,2},b)	= {1,3}
goto({1,2,3},a)	= {1,2,3}
goto({1,2,3},b)	= {1,3}
goto({1,3},a)	= {1,2}
goto({1,3},b)	= {1}

NFA → DFA: An Example (Cont'd)

ε-closure({1}) = {1}
goto({1},a) = {1,2}
goto({1},b) = {1}
goto({1,2},a) = {1,2,3}
goto({1,2},b) = {1,3}
goto({1,2,3},a) = {1,2,3}
goto({1,2,3},b) = {1,3}
goto({1,3},a) = {1,2}
goto({1,3},b) = {1}



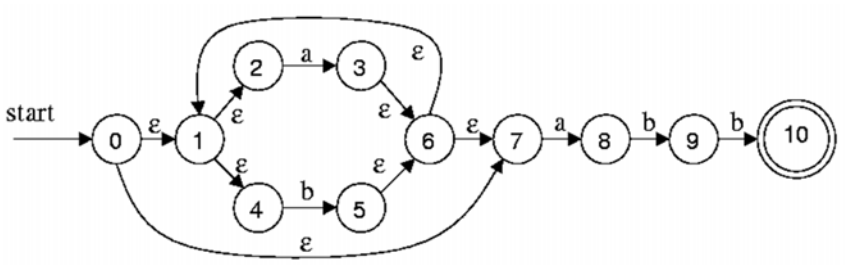
NFA vs. DFA

- R = Size of regular expression
- N = Length of input string

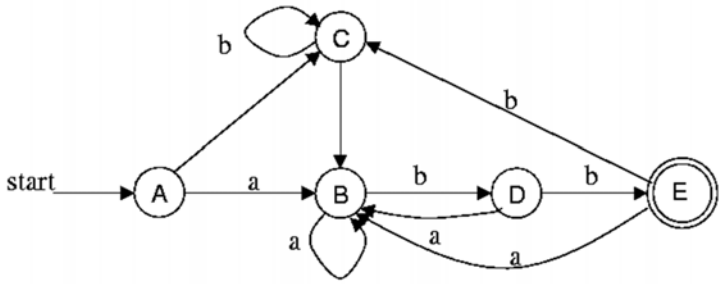
	NFA	DFA
Size of Automaton	**O(R)**	**$O(2^R)$**
Recognition time per input string	**O(N×R)**	**O(N)**

Converting an NFA to DFA

- Just one more example
  - $(a|b)^*abb$

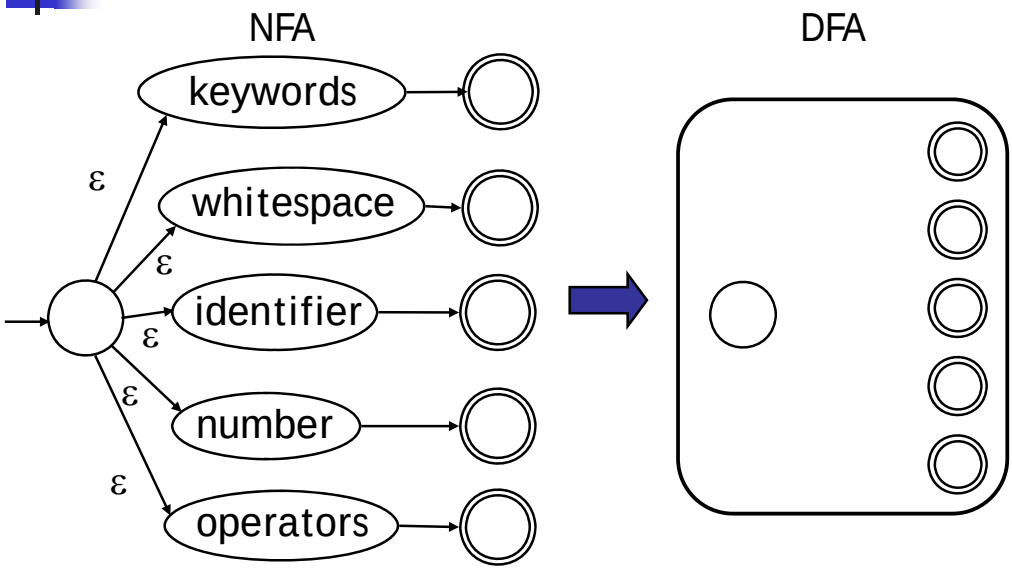


Converting an NFA to DFA (Cont'd)



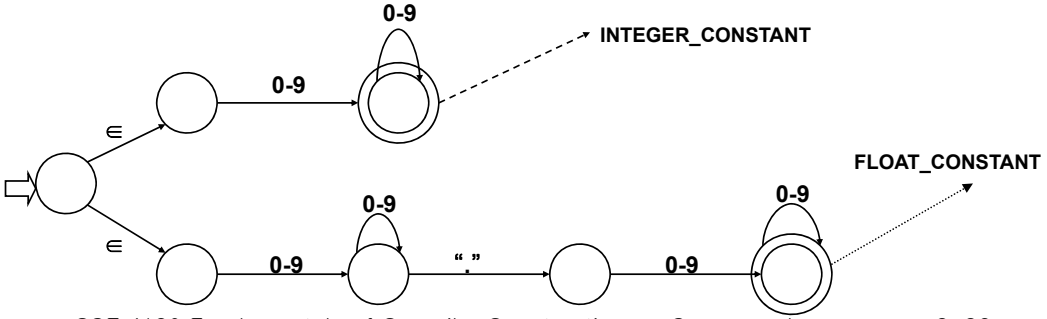
- A = {0, 1, 2, 4, 7}
- B = {1, 2, 3, 4, 6, 7, 8}
- C = {1, 2, 4, 5, 6 ,7}
- D = {1, 2, 4, 5, 6, 7, 9}
- E = {1, 2, 4, 5, 6, 7, 10}

Handling multiple token REs



Specifying Lexical Analysis

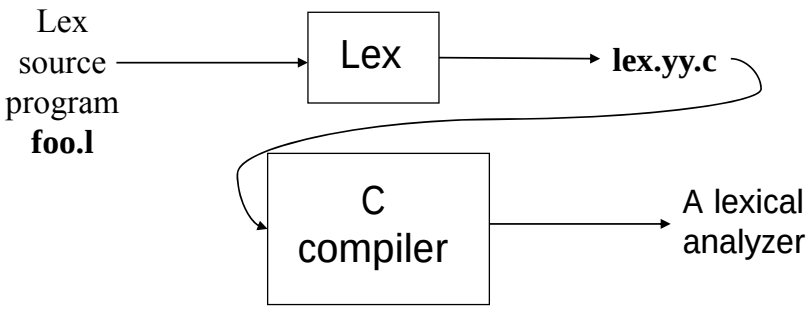
- Each final state in an automaton (DFA/NFA) is associated with an action: ⇒ *emit the corresponding token*
- Consider a recognizer for integers (sequence of digits) and floats (sequence of digits separated by a decimal point)
  - $[0\text{-}9]^+$ {emit(INTEGER_CONSTANT);}
  - $[0\text{-}9]^+$“.”$[0\text{-}9]^+$ {emit(FLOAT_CONSTANT);}



Lex

- A tool for building lexical analyzers or a tool for generating a scanner written in C from a lexical specification for the scanner
  - flex (Fast Lex): the most popular version of Lex
- Input: lexical specifications (*.l* file)
- Output: C function (yylex) that returns a token on each invocation
  - yylex(): table-driven implementation of a DFA : lex.yy.c
- Tokens are simply integers
  - Usually defined as an enumerated type
    - typedef enum
      {IF, THEN, ELSE, PLUS, NUM, ID, ... } TokenType;

Constructing Lexical Analyzer Using (f)lex



Lex Convention for Regular Expression

- Matching strings
  - Example: if or "if "
    - To match the reserved word ***if*** for an ***if*** statement
  - " required for metacharacters
    - "(" required for metacharacters
    - \ for single metacharacters: e.g., \(*
  - Examples: Equivalent expressions
    - (aa|bb)(a|b)*c?
    - ("aa"|"bb")("a"|"b")*"c"?
    - (aa|bb)[ab]*c?

Lex Convention for Regular Expression

- Ranges of characters
  - e.g., [0-9] : any digits zero through nine
- . (period) – any character except a newline
- Complementary sets – sets that do not contain certain characters
  - ^ as the first character inside the brackets
  - e.g., [^0-9abc] : any character that is not a digit and is not one of the letters a, b, or c
- Example: RE for the set of signed numbers
  - ("+"|"-")?[0-9]+("."[0-9]+)?(E("+"|"-")?[0-9]+)?
  - c.f., [-+] vs. [+-]

Lex Convention for Regular Expression

- Lex uses the convention that previously defined names are surrounded by { } (curly brackets)
- Definition of "signed natural number"
  - *nat* = [0-9]+
  - *signedNat* = ("+"|"–")? *nat*
- In Lex conventions,
  - nat [0-9]+
  - signedNat (+| –)?{nat}
  - [^*/]+ Sequence of characters except * and /
  - \" [^"]*\" Sequence of non-quote characters beginning and ending with a quote
  - ({letter}|"_")({letter}|{digit}|"_")* C style identifiers

Basic Metacharacter Conventions in Lex

Pattern	Meaning
a	The character a
"a"	The character a, even if a is a metacharacter
\a	The character a when a is a metacharacter
a*	Zero or more repetitions of a
a+	One or more repetitions of a
a?	An optional a
a|b	a or b
(a)	a itself
[abc]	Any of the characters a, b, or c
[a-d]	Any of the characters a, b, c, or d
[^ab]	Any character except a or b
.	Any character except a newline
{xxx}	The regular expression that the name xxx represents

The Format of a Lex Input File

```
{definitions}
%%
{rules}
%%
{auxiliary routines}
```

- Lex input file
  - Definitions- including
    - C code that must be inserted external to any function should appear in this section between the delimiters %{ and %}
    - Names for regular expressions must also be defined in this section
  - Rules
    - Sequence of regular expressions followed by the C code (i.e., action) that is to be executed when the corresponding regular expression is matched
  - Auxiliary routines or user routines
    - Auxiliary routines that are called in the second section
    - May also contain a main program – for standalone program

Example 1

```
%{
#include <stdio.h>
#include “tokens.h”
%}
digit       [0-9]
hexdigit    [0-9a-f]
%%
"+" { return(PLUS); }
"–"         { return(MINUS); }
{digit}+                { return(INTEGER_CONSTANT); }
{digit}+ "."{digit}+    { return(FLOAT_CONSTANT); }
.                       { return(SYNTAX_ERROR); }
%%
```

Priority of matching

- What if an input string matches more than one RE?

if	{return(TOKEN_IF);}
{letter}$^+$	{return{TOKEN_ID);}
while	{return(TOKEN_WHILE);}

- Two important disambiguation rules
  - a RE that matches the longest string is chosen
    - e.g., **if1** is matched with an identifier, not the keyword **if**
  - of REs that match strings of same length, the first (from the top of file) is chosen
    - e.g., **while** is matched as an identifier, not the keyword **while**

Example 2

- To add line numbers to text

```
%{
/* a Lex program that adds line numbers
   to lines of text
*/
#include <stdio.h>
int lineno = 1;
%}
line .*\n
%%
{line} { printf("%5d %s",lineno++,yytext); }
%%
main()
{ yylex(); return 0; }
yywrap()
{ /* actions for reaching EOF */ }
```

Some Lex Internal Names

- Additional information about a token's lexeme
  - **yytext**: lexeme (actual text string)
    - Internal name Lex gives to the string matched by the regular expression
  - **yyleng**: length of string in yytext
  - **yylineno**: current line number (number of '\n' seen thus far)
    - Enabled by % option yylineno

Lex Internal Name	Meaning/Use
`lex.yy.c`	Lex output filename
`yylex`	Lex scanning routine
`yytext`	String matched on current action
`yyin`	Lex input file (default: stdin)
`yyout`	Lex output file (default: stdout)
`input`	Lex buffered input routine
`ECHO`	Lex default action (print `yytext` to `yyout`)

Example 3

- To change all numbers from decimal to hexadecimals notation

```
%{
/* a Lex program that changes all numbers
   from decimal to hexadecimal notation */
#include <stdlib.h>
#include <stdio.h>
int count = 0;
%}
digit [0-9]
number {digit}+
%%
{number} { int n = atoi(yytext);
           printf("%x", n);
           if (n > 9) count++; }
%%
main()
{ yylex();
  fprintf(stderr, "number of replacements = %d",
                  count);
  return 0;
}
```

Advanced Metacharacter Conventions in Lex

Pattern	Meaning
x	match the character 'x'
.	any character (byte) except newline
[xyz]	a "character class"; in this case, the pattern matches either an 'x', a 'y', or a 'z'
[abj-oZ]	a "character class" with a range in it; matches an 'a', a 'b', any letter from 'j' through 'o', or a 'Z'
[^A-Z]	a "negated character class", i.e., any character but those in the class. In this case, any character except an uppercase letter.
[^A-Z\n]	any character except an uppercase letter or a newline
r*	zero or more r's, where r is any regular expression
r+	one or more r's
r?	zero or one r's (that is, "an optional r")
r{2,5}	anywhere from two five r's
r{2,}	two or more r's
r{4}	exactly 4 r's

Advanced Metacharacter Conventions in Lex

Pattern	Meaning
{name}	the expansion of the "name" definition (see above)
"[xyz]\"foo"	the literal string: '[xyz]"foo"
\x	if x is an `a', `b', `f', `n', `r', `t', or `v', then the ANSI-C interpretation of \x. Otherwise, a literal `x' (used to escape operators such as`*')
\0	a NUL character (ASCII code 0)
\123	the character with octal value 123
\x2a	the character with hexadecimal value 2a
(r)	match an r; parentheses are used to override precedence (see below)
rs	the regular expression r followed by the regular expression s; called "concatenation"
r|s	either an r or an s
r/s	an r but only if it is followed by an s
^r	an r, but only at the beginning of a line
r$	Equivalent to"r/\n".

To handle C comments

- **input()** reads the next character from the input stream.

```
%%
"/*" { register int c;
       for ( ; ; )
       { while ( (c = input()) != '*' && c !=EOF)
           ; /* eat up text of comment */
         if( c =='*' )
         { while ( (c = input()) == '*' )
             ;
           if ( c == '/' )
             break; /* found the end */
         }
         if( c == EOF )
         { error( "EOF in comment" );
           break;
         }
       }
     }
```


Lexical Analysis: A Summary

- Convert a stream of characters into a stream of tokens
  - Make rest of compiler independent of character set
  - Strip off comments
  - Recognize line numbers
  - Ignore white space characters
  - Process macros (definitions and uses)
  - Interface with symbol (name) table
- Things to do
  - Read (f)lex manual
  - Read C- language reference manual in Appendix A of the textbook