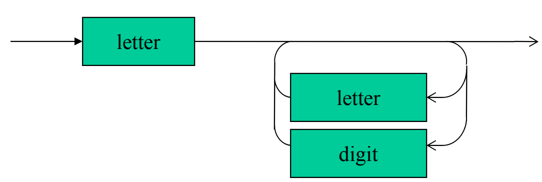
Lecture 3: Lexical Analysis

COSC 4316

(grateful acknowledgement to Robert van Engelen and Elizabeth White for some of the material from which these slides have been adapted)

Lexical Analysis

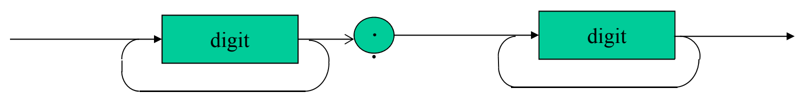
- Simple way to build a lexical analyzer:
  - Construct a structure diagram for tokens
  - Hand translate into a program
    identifier:



  - The techniques we use in lexical analyzers are also used in other applications, for example, any time a pattern in a string triggers some action

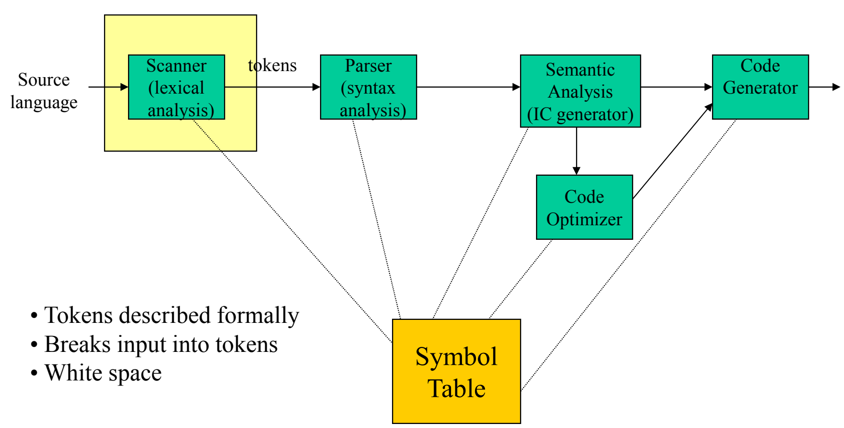
Lexical Analysis

Real constant:



lex is a language for pattern actions that we will use

Lexical Analysis - Scanning

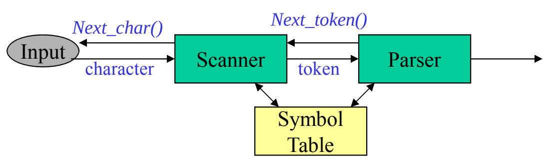


- Tokens described formally
- Breaks input into tokens
- White space

Lexical Analysis

INPUT: sequence of characters

OUTPUT: sequence of tokens



A lexical analyzer is generally a subroutine of parser but separate from it:

- Simpler design
- Efficient
- Portable

Tasks of Lexical Analysis

- **Read input characters and output sequences of tokens**
- Strip out comments and “white space”
- Correlate error messages with the source program
  - Keep track of line numbers
  - Make copy of source with the errors marked
- Implement macro preprocessor functions

Issues in Lexical Analysis

- **Why separate lexical analysis from parsing?**
  - Simpler design
  - Compiler efficiency
  - Compiler portability

Definitions

- **token** – terminal symbol in the grammar
- **pattern** – a rule describing a set of string matching a particular token
- **lexeme** – a sequence of characters with a collective meaning
- **Attributes** of a token: usually only a pointer to the symbol table entry

Examples

Token	Pattern	Sample Lexeme
while	while	while
rel_op	= | != | < | >	<
int_lit	(0-9)*	42
string_lit	Characters between “ ”	“hello”

Input string: size := r * 32 + c

<token,lexeme> pairs:

- < id, size >
- < assign, := >
- < id, r >
- < arith_symbol, * >
- < integer, 32 >
- < arith_symbol, + >
- < id, c >

Lexical Errors

- Not many errors are detectible at the lexical level
- Most often use “panic mode” recovery
  - Delete successive characters until a well-formed token is found
- Other strategies
  - Try to correct input and proceed
  - Usually this is very difficult

Implementing a Lexical Analyzer

Practical Issues:

- Input buffering
- Translating RE into executable form
- Must be able to capture a large number of tokens with single machine
- Interface to parser
- Tools

Review of Formal Languages

- Regular expressions, NFA, DFA
- Translating between formalisms
- Using these formalisms

Specification of Tokens

- **Symbol –** used without definition
  - *e.g.*: letters and digits
- **Alphabet** – finite set of symbols
  - *e.g.*: English alphabet, binary alphabet {0,1}, ASCII, *etc*.
  - A fixed alphabet is usually denoted by Σ
- **String** – finite sequence of symbols from a fixed alphabet Σ
  - Length of a string, s, is |s| = # of symbols in s
  - Length of **abcabc** is 6
  - ε is the empty string (Some texts use Λ as the empty string)
  - |ε| = 0

Specification of Tokens

- **String**
  - Prefix of a string: if s = *abcd,* prefixes are ε, *a, ab, abc, abcd*
  - Suffix is analogously defined
  - ***Concatenation*** of two strings is the string formed by writing the first followed by the second with no intervening space
  - ε is the concatenation identity

Languages

- **Language** – a set of strings over a fixed alphabet
  - can be { } (aka Ø), the empty language.
  - {} ≠ {ε} ***different languages***!
  - *e.g*. set of all palindromes over {0,1},
  - Set of all strings over {0,1}
    = {ε, 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, 100, 101, ... }

Operations on Languages

- **union** – since languages are sets, the union of two languages L_1 and L_2 is already defined:
  - $L_1 \cup L_2 = \{ s \mid s \in L_1 \text{ or } s \in L_2 \}$
  - Concatenation: $L_1L_2 = \{ st \mid s \in L_1 \text{ and } t \in L_2 \}$
  - “Exponentiation”
    - $L^0 = \{\varepsilon\}$ $L^i = L^{i-1}L$
  - *Kleene Closure* $L^* = \bigcup_{i=0}^{\infty} L^i$
  - *Positive Closure* $L^+ = \bigcup_{i=1}^{\infty} L^i$ $L^* = L^+$ iff $\varepsilon \in L$

Suppose $\Sigma = \{a,b,c\}$. Some languages over Σ could be:

- {aa,ab,ac,bb,bc,cc}
- {ab,abc,abcc,abccc,. . .}
- { ε }
- { }
- {a,b,c,ε}
- …

Why do we care about Regular Languages?

- Formally describe tokens in the language
  - Regular Expressions
  - NFA
  - DFA
- Regular Expressions ➔ finite automata
- Tools assist in the process

Definition of Regular Expressions

- Each regular expression **r** denotes a language L(r) (i.e., a set of strings)

Defn: The **regular expressions** over finite Σ are the strings over the alphabet Σ + {), (, |, * } such that:

1. **Ø** is a regular expression denoting the empty set
2. ε is a regular expression denoting the set { ε }
3. ***a*** is a regular expression denoting set { *a* } for any *a* in Σ

(*definition continued on next slide*)

Definition of Regular Expressions (2)

4. If r denotes L(r) and s denotes L(s), (regular expressions over Σ), then:
   - **r | s** is a regular expression denoting $L(r) \cup L(s)$ **(<u>union</u>)**
   - **rs** is a regular expression denoting L(r)L(s) **(<u>concatenation</u>)**
   - **r*** is a regular expression denoting $(L(r))^*$ **(<u>closure</u>)**

Precedence Rules for REs

- * has highest precedence and is left associative
- Concatenation has 2nd highest precedence and is left associative
- | has lowest precedence and is left associative
- Parentheses are used to change order of operations
- Example: a | b*c ⇔ (a) | ((b)* (c))

Examples

If $\Sigma = \{a,b\}$

- (a | b)(a | b) denotes $\{aa,ab,ba,bb\}$
- (a | b)*b denotes all strings of *a*'s and *b*'s ending in *b*
- a*b*a*
- a*a (also known as a^+)
- (ab*)|(a*b)
- (a|b)* = (a*b*)* (denote the same language – all strings of *a*'s and *b*'s)

Properties of regular expressions

- r | s = s | r (commutativity of |)
- r | (s | t) = (r | s) | t (associativity of |)
- (rs)t = r(st)
- r(s|t) = rs | rt
- (s|t)r = sr | tr
- rε | εr = r
- r** = r*

Regular Definitions

- Like productions in a CFG (but no recursion)
- *digit* → 0 | 1 | 2 | … | 9
- *letter*→ A | B | C | … | Z
- *id* → *letter (letter | digit)**

Regular Definitions (2)

- Unsigned numbers:

  digits → *digit* (*digit*)*

  fraction → **.** *digits* | ε

  exponent → **E (+ | - |** ε) *digits* | ε

  unsigned_number → *digits fraction exponent*

Notational Shorthands

- $r^+ \equiv rr^*$ (one or more)
- $r? \equiv r \mid \varepsilon$ (zero or one)
- $[abc] \equiv a \mid b \mid c$ where $a, b, c \in \Sigma$
- $[a\text{-}z] \equiv a \mid b \mid c \mid \ldots \mid z$
- Examples
  - *id* → [A-Za-z][A-Za-z0-9]*
  - *unsignednumber* → $[0\text{-}9]^+(.[0\text{-}9]^+)?(E(+|-)?[0\text{-}9]^+)?$

A little extreme

Non-regular Sets Exist

- Matched parentheses cannot be represented by a regular expression
- They can, however, be easily represented by the CFG
  - S → S (S) S | ε
- Repeating strings are not regular, nor even context free, for example: $\{ wcw \mid w \in (a|b)^+ \}$
- Regular expressions denote
  - Fixed number of repetitions, or
  - An unspecified number of repetitions

Regular Definitions and Grammars

Grammar fragment

stmt → **if** *expr* **then** *stmt*
  | **if** *expr* **then** *stmt* **else** *stmt*
  | ε
expr → *term* **relop** *term*
  | *term*
term → **id**
  | **num**

Regular definitions

if → `if`
then → `then`
else → `else`
relop → `<` | `<=` | `<>` | `>` | `>=` | `=`
id → **letter** (**letter** | **digit**)*
num → **digit**+ (. **digit**+)? (**E** (+ | -)? **digit**+)?

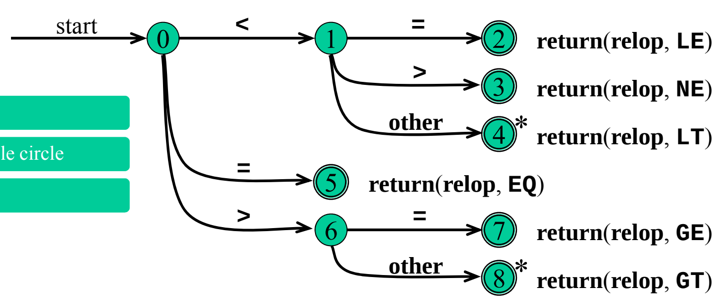
Assume lexemes are separated by white space
delim → ‘ ’ | ‘\t’ | ’\n’
ws → **delim+**
The lexical analyzer does not return a token if a match for ws is found. It instead goes on to find another token.

29

Coding Regular Definitions in *Transition Diagrams*

Transition diagrams can be used as an aid in the construction of a lexical analyzer

relop → `<` | `<=` | `<>` | `>` | `>=` | `=`



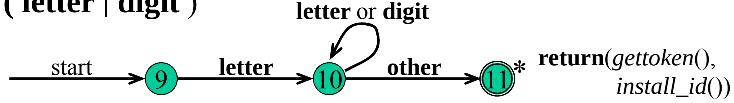
initial state – labeled the *start* state.

accepting state – indicated by a double circle

* Indicates input retraction occurs

Gettoken() looks in symbol table for a keyword, if there then returns token for keyword, otherwise returns **id**
installid() returns 0 if lexeme is a keyword; otherwise returns symbol table ptr.

id → **letter** (**letter** | **digit**)*



30

Coding Regular Definitions in Transition Diagrams: Code

```
token nexttoken()
{ while (1) {
    switch (state) {
    case 0: c = nextchar();
       if (c==blank || c==tab || c==newline) {
         state = 0;
         lexeme_beginning++;
       }
       else if (c=='<') state = 1;
       else if (c=='=') state = 5;
       else if (c=='>') state = 6;
       else state = fail();
       break;
     case 1:
       …
     case 9: c = nextchar();
       if (isletter(c)) state = 10;
       else state = fail();
       break;
     case 10: c = nextchar();
       if (isletter(c)) state = 10;
       else if (isdigit(c)) state = 10;
       else state = 11;
       break;
     …
```

Decides the next start state to check

```
int fail()
{ forward = token_beginning;
  swith (start) {
  case  0: start =  9; break;
  case  9: start = 12; break;
  case 12: start = 20; break;
  case 20: start = 25; break;
  case 25: recover(); break;
  default: /* error */
  }
  return start;
}
```

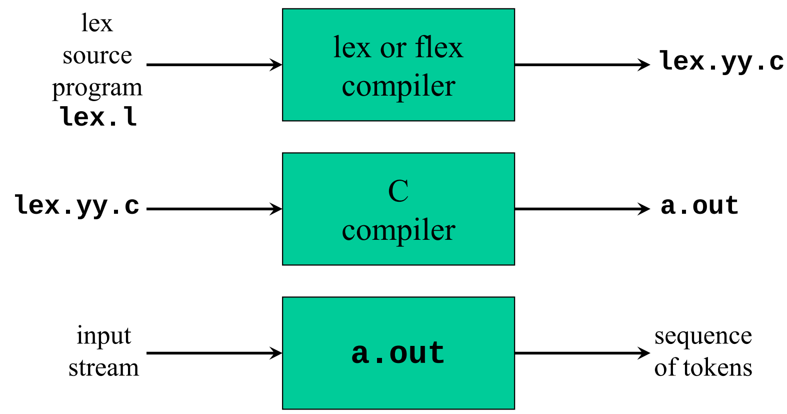
31

The Lex and Flex Scanner Generators

- *Lex* and its newer cousin *flex* are scanner generators
- Systematically translate regular definitions into C source code for efficient scanning
- Generated code is easy to integrate in C applications

32

Creating a Lexical Analyzer with Lex and Flex



33

Lex Specification

- A *lex specification* consists of three parts:

  regular definitions, C declarations in **`%{ %}`**
  %% ← required
  translation rules
  %% ← Optional if user procedures ommitted
  user-defined auxiliary procedures ← optional

- The *translation rules* are of the form:

  p_1 { $action_1$ }
  p_2 { $action_2$ }
  …
  p_n { $action_n$ }

34

Lex Specification

- Absolute minimum Lex program:
  %%

(no definitions, no rules)

What does it do? Copies the input to the output unchanged.

Translation rules in the form:

regular expression actions (program fragments)

e.g.

```
integer            {printf("keyword INT found")}
```

This looks for the string "integer" in the input stream and prints the message "keyword INT found" whenever it appears.

35

Regular Expressions in Lex

`x`	match the character `x`
`\.`	match the character `.`
"*string*"	match contents of string of characters
`.`	match any character except newline
`^`	match beginning of a line
`$`	match the end of a line
`[xyz]`	match one character `x`, `y`, or `z` (use `\` to escape `-`)
`[^xyz]`	match any character except `x`, `y`, and `z`
`[a-z]`	match one of `a` to `z`
r`*`	closure (match zero or more occurrences)
r`+`	positive closure (match one or more occurrences)
r`?`	optional (match zero or one occurrence)
r_1r_2	match r_1 then r_2 (concatenation)
r_1`|`r_2	match r_1 or r_2 (union)
`(` r `)`	grouping
r_1`\`r_2	match r_1 when followed by r_2
`{`d`}`	match the regular expression defined by d

36

Example Lex Specification 1

Translation rules

Contains the matching lexeme

Invokes the lexical analyzer

```
%{
#include <stdio.h>
%}
%%
[0-9]+  { printf("%s\n", yytext); }
.|\n    { }
%%
main()
{ yylex();
}
```

```
lex spec.l
gcc lex.yy.c -ll
./a.out < spec.l
```

37

Example Lex Specification 2

Translation rules

Regular definition

```
%{
#include <stdio.h>
int ch = 0, wd = 0, nl = 0;
%}
delim     [ \t]+
%%
\n        { ch++; wd++; nl++; }
^{delim}  { ch+=yyleng; }
{delim}   { ch+=yyleng; wd++; }
.         { ch++; }
%%
main()
{ yylex();
  printf("%8d%8d%8d\n", nl, wd, ch);
}
```

yyleng contains the number of characters in **yytext**

38

Example Lex Specification 3

Translation rules

Regular definitions

```
%{
#include <stdio.h>
%}
digit     [0-9]
letter    [A-Za-z]
id        {letter}({letter}|{digit})*
%%
{digit}+  { printf("number: %s\n", yytext); }
{id}      { printf("ident: %s\n", yytext); }
.         { printf("other: %s\n", yytext); }
%%
main()
{ yylex();
}
```

39

Example Lex Specification 4

Return token to parser

Token attribute

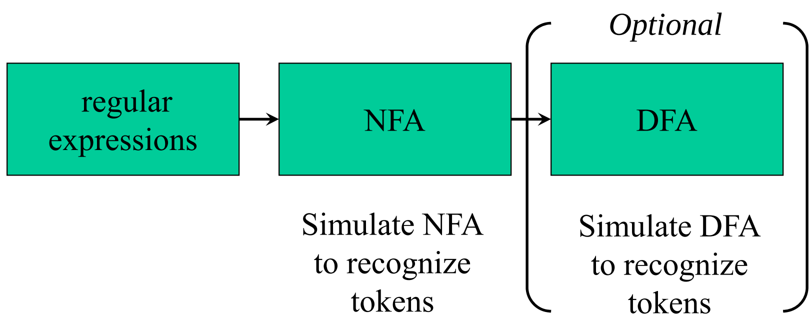
Install **yytext** as identifier in symbol table

```
%{ /* definitions of manifest constants */
#define LT (256)
…
%}
delim     [ \t\n]
ws        {delim}+
letter    [A-Za-z]
digit     [0-9]
id        {letter}({letter}|{digit})*
number    {digit}+(\.{digit}+)?(E[+\-]?{digit}+)?
%%
{ws}      { }
if        {return IF;}
then      {return THEN;}
else      {return ELSE;}
{id}      {yylval = install_id(); return ID;}
{number}  {yylval = install_num(); return NUMBER;}
"<"       {yylval = LT; return RELOP;}
"<="      {yylval = LE; return RELOP;}
"="       {yylval = EQ; return RELOP;}
"<>"      {yylval = NE; return RELOP;}
">"       {yylval = GT; return RELOP;}
">="      {yylval = GE; return RELOP;}
%%
int install_id()
…
```

40

Design of a Lexical Analyzer Generator

- Translate regular expressions to NFA
- Translate NFA to an efficient DFA



Nondeterministic Finite Automata

A **nondeterministic finite automaton** (NFA) is a mathematical model denoted by the 5-tuple $(S, \Sigma, \delta, s_0, F)$ where

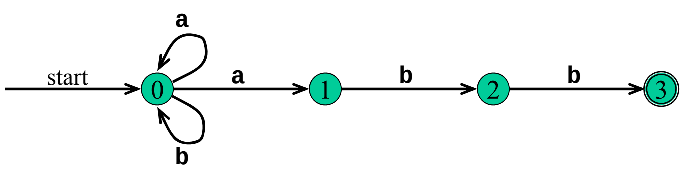
1. S is a <u>set of states</u>
2. Σ is an input <u>alphabet</u>
3. δ is a <u>transition function</u> that maps state/symbol pairs to a set of states:
   $S \times \{\Sigma + \varepsilon\} \rightarrow$ set of S
4. $s_0 \in S$ is a special state called the <u>start state</u>
5. $F \subseteq S$ is a set of <u>accepting states</u>

INPUT: string

OUTPUT: yes or no

Transition Graph

- An NFA can be diagrammatically represented by a labeled directed graph called a *transition graph*



$S = \{0,1,2,3\}$
$\Sigma = \{\mathbf{a},\mathbf{b}\}$
$s_0 = 0$
$F = \{3\}$

43

Transition Table

- The mapping δ of an NFA can be represented in a *transition table*

$\delta(0,\mathbf{a}) = \{0,1\}$
$\delta(0,\mathbf{b}) = \{0\}$
$\delta(1,\mathbf{b}) = \{2\}$
$\delta(2,\mathbf{b}) = \{3\}$

→

State	*Input* **a**	*Input* **b**
0	{0, 1}	{0}
1		{2}
2		{3}

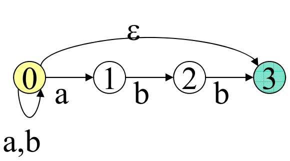
44

The Language Defined by an NFA

- An NFA *accepts* an input string *x* if and only if there is some path with edges labeled with symbols from *x* in sequence from the start state to some accepting state in the transition graph
- A state transition from one state to another on the path is called a *move*
- The *language defined by* an NFA is the set of input strings it accepts, such as (**a**|**b**)***abb** for the example NFA

45

Example NFA



S = {0,1,2,3}
$S_0 = 0$
Σ = {a,b}
F = {3}

Transition Table:

STATE	a	b	ε
0	0,1	0	3
1		2	
2		3	
3			

- The language accepted by an NFA: the set of strings it accepts
- For example, the language accepted here is (a|b)*(abb | ε)

NFA Execution

An NFA says 'yes' for an input string if there is some path from the start state to some final state where all input has been processed.

```
NFA(int s0, int input_element) {
   if (all input processed and s0 is a final state) return Yes;
   if (all input processed and s0 is not a final state) return No;

   for all states s1 where transition(s0,table[input_element]) = s1
        if (NFA(s1,input_element+1) = = Yes) return Yes;

   for all states s1 where transition(s0,ε) = s1
        if (NFA(s1,input_element) = = Yes) return Yes;
    return No;
}
```

Uses backtracking to search all possible paths

Deterministic Finite Automata

- A *deterministic finite automaton* is a special case of an NFA
  - No state has an ε-move
  - For each state *s* and input symbol *a* there is at most one edge labeled *a* leaving *s*
- Each entry in the transition table is a single state
  - At most one path exists to accept a string
  - Simulation algorithm is simple

An Algorithm Simulating a DFA

$s \leftarrow s_0$
$ch \leftarrow getchar()$
while (ch ≠ end_of_string) **or** (s ≠ error_state) **do**
    $s \leftarrow \delta(s, ch)$
    $ch \leftarrow getchar()$
endwhile
accept ← $s \in F$

DFA Execution

```
DFA(int start_state) {
   state current = start_state;
   input_element = next_token();
   while (input to be processed) {
       current = transition(current,table[input_element])
       if current is an error state return No;
       input_element = next_token();
    }
    if current is a final state return Yes;
    else return No;
}
```

Regular Languages

1. There is an algorithm for converting any RE into an NFA.
2. There is an algorithm for converting any NFA to a DFA.
3. There is an algorithm for converting any DFA to a RE.

These facts tell us that REs, NFAs and DFAs have equivalent expressive power. All three describe the class of regular languages.

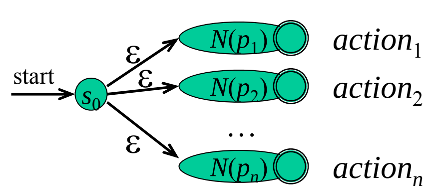
Design of a Lexical Analyzer Generator: RE to NFA to DFA

Overview

Lex specification with regular expressions

p_1	{ $action_1$ }
p_2	{ $action_2$ }
…	
p_n	{ $action_n$ }

NFA



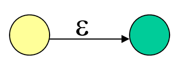
Subset construction

DFA

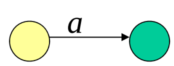
Converting Regular Expressions to NFAs

The **regular expressions** over finite Σ are the strings over the alphabet Σ + {), (, |, *} such that:

- Empty string ε is a regular expression denoting { ε }



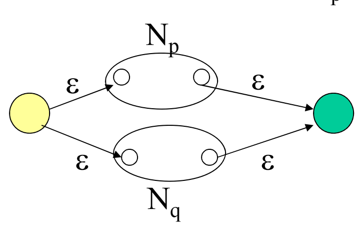
- *a* is a regular expression denoting {*a* } for any *a* in Σ



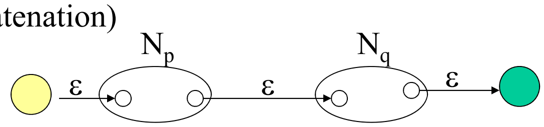
Converting Regular Expressions to NFAs

If P and Q are regular expressions with NFAs N_p, N_q:

P | Q (union)



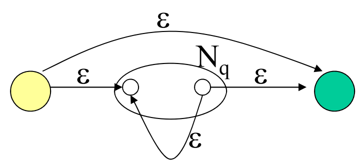
PQ (concatenation)



Converting Regular Expressions to NFAs

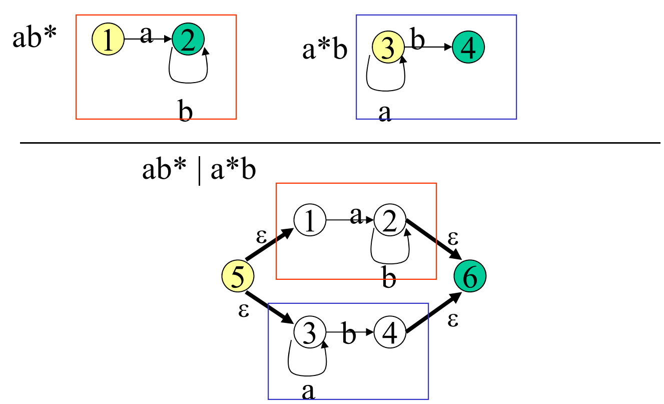
If Q is a regular expression with NFA N_q:

Q* (closure)



Example (ab* | a*b)*

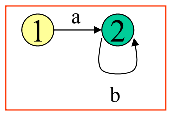
Starting with:



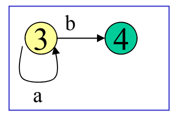
Example (ab* | a*b)*

Starting with:

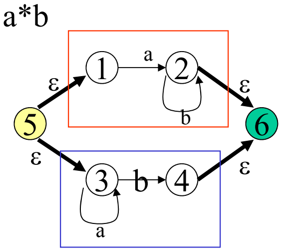
ab*



a*b

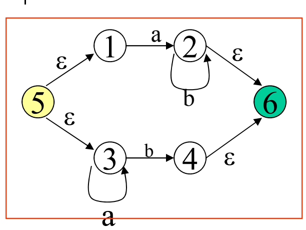


ab* | a*b

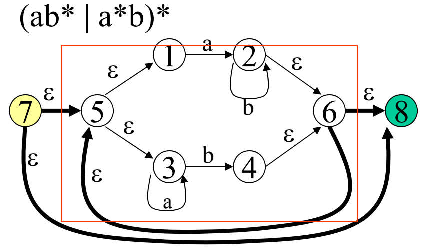


Example (ab* | a*b)*

ab* | a*b



(ab* | a*b)*



Converting NFAs to DFAs

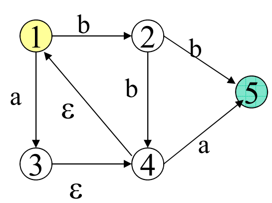
- **Idea**: Each state in the new DFA will correspond to some set of states from the NFA. The DFA will be in state $\{s_0, s_1, \ldots\}$ after input if the NFA could be in *any* of these states for the same input.

- **Input**: NFA N with state set S_N, alphabet Σ, start state s_N, final states F_N, transition function T_N: $S_N \times \Sigma + \{\varepsilon\} \rightarrow$ set of S_N
- **Output**: DFA D with state set S_D, alphabet Σ, start state

$s_D = \varepsilon\text{-closure}(s_N)$, final states F_D, transition function T_D: $S_D \times \Sigma \rightarrow S_D$

ε-closure()

Defn: ε-closure(T) = T + all NFA states reachable from any state in T using only ε transitions.



ε-closure({1,2,5}) = {1,2,5}
ε-closure({4}) = {1,4}
ε-closure({3}) = {1,3,4}
ε-closure({3,5}) = {1,3,4,5}

Algorithm: Subset Construction

$s_D \leftarrow \varepsilon$-closure($s_N$) -- create start state for DFA

$S_D = \{s_D\}$ (unmarked)

while there is some unmarked state **R** in S_D

mark state **R**

for all a in Σ do

s = ε-closure(T_N(**R**,a));

if s not already in S_D then add it (unmarked)

T_D(**R**,a) = s;

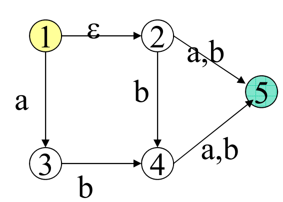
end for

end while

F_D = any element of S_D that contains a state in F_N

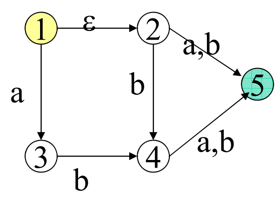
Example 1: Subset Construction

NFA



Example 1: Subset Construction

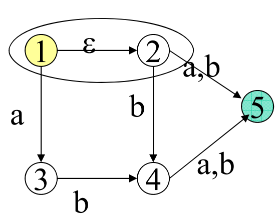
NFA

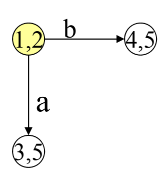


	a	b
{1,2}		

Example 1: Subset Construction

NFA

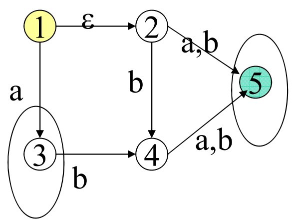


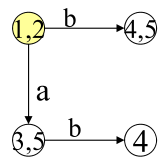


	a	b
{1,2}	{3,5}	{4,5}
{3,5}		
{4,5}		

Example 1: Subset Construction

NFA

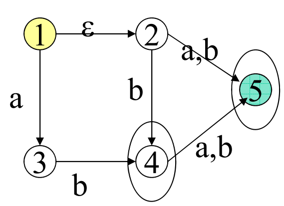


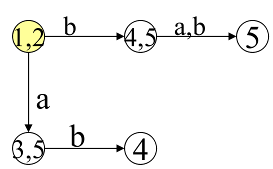


	a	b
{1,2}	{3,5}	{4,5}
{3,5}	-	{4}
{4,5}		
{4}		

Example 1: Subset Construction

NFA

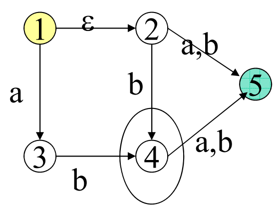


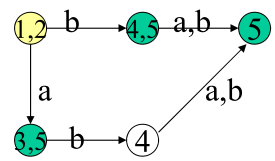


	a	b
{1,2}	{3,5}	{4,5}
{3,5}	-	{4}
{4,5}	{5}	{5}
{4}		
{5}		

Example 1: Subset Construction

NFA



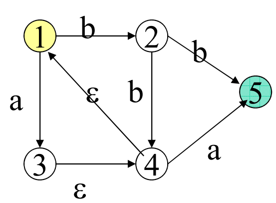


	a	b
{1,2}	{3,5}	{4,5}
{3,5}	-	{4}
{4,5}	{5}	{5}
{4}	{5}	{5}
{5}	-	-

All these states are final states since the NFA final state is included

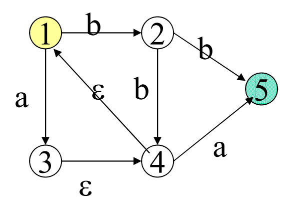
Example 2: Subset Construction

NFA

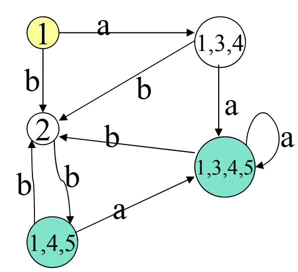


Example 2: Subset Construction

NFA

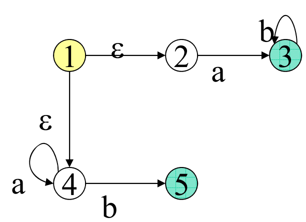


DFA

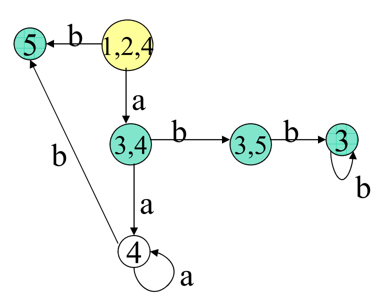


Example 3: Subset Construction

NFA



DFA

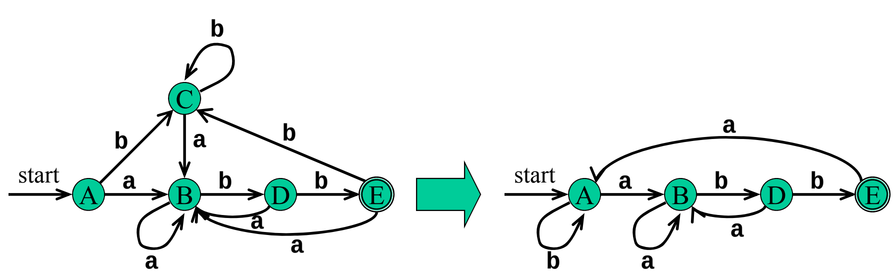


Example: re → NFA → DFA → minimized DFA

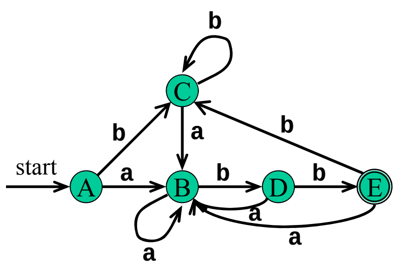
- a {$action_1$}| abb {$action_2$}| a*b+ {$action_3$}
- Convert to NFA to DFA and then minimize

Minimizing the Number of States of a DFA

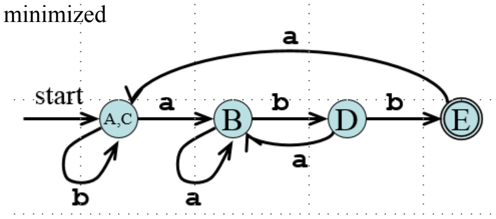
- How do we do this?



Minimizing the Number of States of a DFA



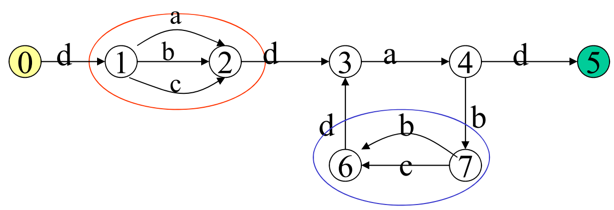
- First, split into nonfinal/final states
- {A, B, C, D} {E}
- {A, B, C} {D} {E}
  - (since D on input b goes to E, so it is different)
- {A, C} {B} {D} {E}
  - (since B on input b goes to D, so it is different than A&C)
- At this point each of the original states in a subset go to the same subset on the same input, so we are minimized



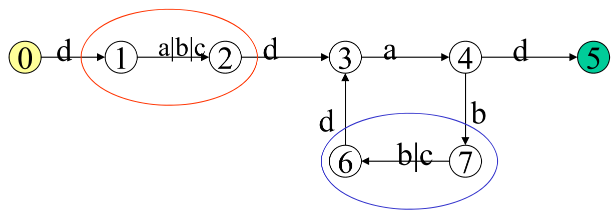
Converting DFAs to REs (optional topic)

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.

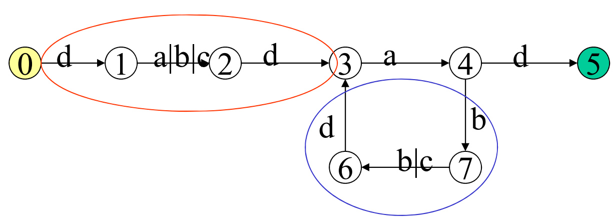
Example



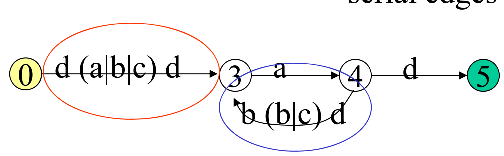
parallel edges become alternation



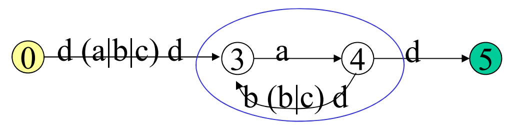
Example



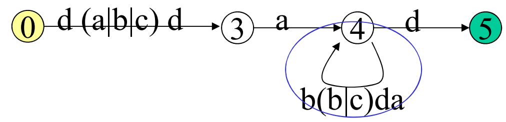
serial edges become concatenation



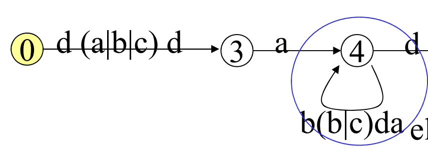
Example



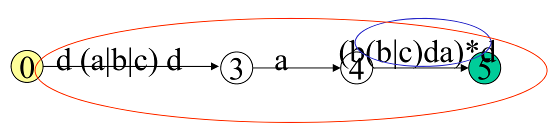
Find paths that can be “shortened”



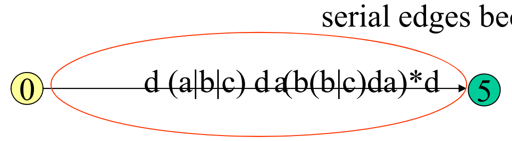
Example



eliminate self-loops



serial edges become concatenation



Describing Regular Languages

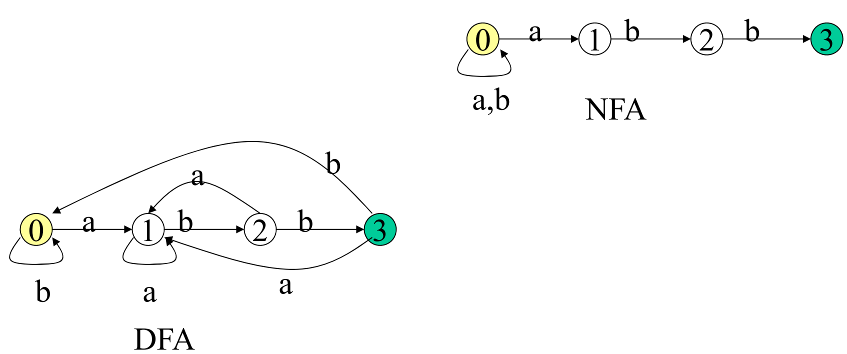
- Generate ***all*** strings in the language
- Generate ***only*** strings in the language

Try the following:

– Strings of {*a*,*b*} that end with ‘*abb*’
– Strings of {*a*,*b*} that don’t end with ‘*abb*’
– Strings of {*a*,*b*} where every *a* is followed by at least one *b*

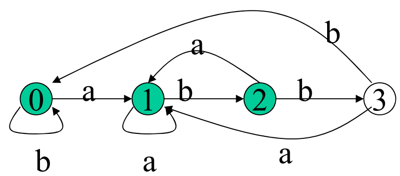
Strings of (a|b)* that end in abb

re: (a|b)*abb



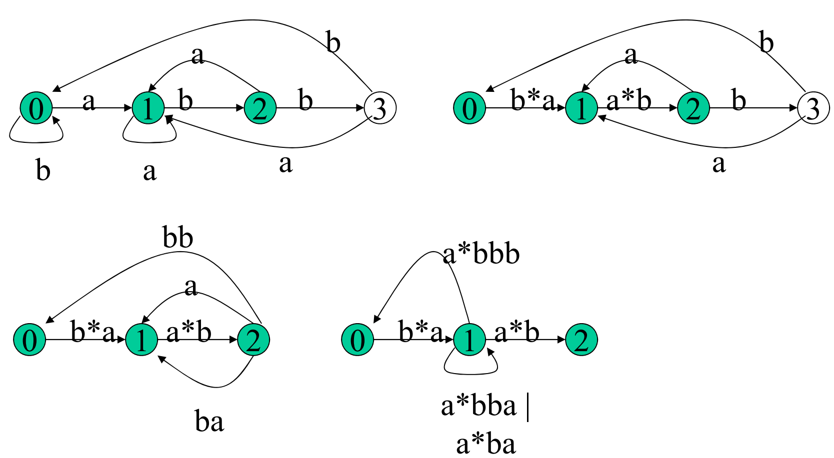
Strings of (a|b)* that don’t end in abb

re: ??



DFA/NFA

Strings of (a|b)* that don’t end in abb



Suggestions for writing NFA/DFA/RE

- Typically, one of these formalisms is more natural for the problem. Start with that and convert if necessary.
- In NFA/DFAs, each state typically captures some partial solution
- Be sure that you include all relevant edges (ask – does every state have an outgoing transition for all alphabet symbols?)

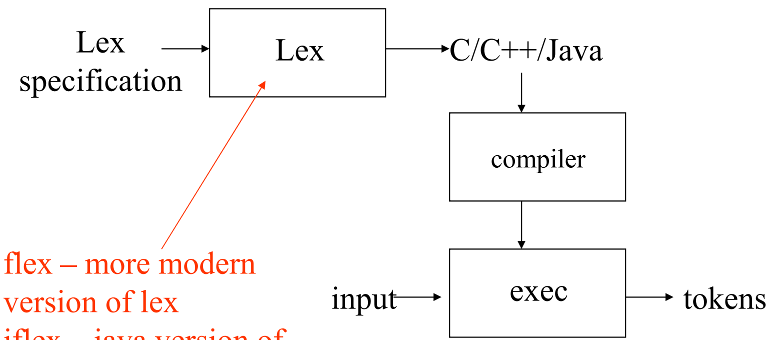
Non-Regular Languages

Not all languages are regular"

- The language *ww* where *w*=(a|b)*

Non-regular languages cannot be described using REs, NFAs and DFAs.

Lex – Lexical Analyzer Generator



flex – more modern version of lex
jflex – java version of flex

Lex Specification (flex)

optional → Definitions – Code, RE

```
%{
int charCount=0, wordCount=0, lineCount=0;
%}
word   [^ \t\n]+
```

required →

```
%%
```

Rules – RE/Action pairs

```
{word}      {wordCount++; charCount += strlen(yytext); }
[\n]        {charCount++; lineCount++;}
.           {charCount++;}
```

optional

```
%%
```

User Routines

```
main() {
    yylex();
    printf("Characters %d, Words: %d, Lines: %d\n",
charCount, wordCount, lineCount);
}
```

Lex Specification (jflex)

```
import java.io.*;
%%
%class ex1
%unicode
%line
%column
%standalone
%{
static int charCount = 0, wordCount = 0, lineCount = 0;
public static void main(String [] args) throws IOException
{
     ex1 lexer = new ex1(new FileReader(args[0]));
     lexer.yylex();
     System.out.println("Characters: " + charCount +
                " Words: " + wordCount +" Lines: " +lineCount);
}
%}
%type Object //this line changes the return type of yylex into Object
word = [^ \t\n]+
%%
{word}    {wordCount++; charCount += yytext().length(); }
[\n]              {charCount++; lineCount++; }
.         {charCount++; }
```

Definitions – Code, RE

Rules – RE/Action pairs

Lex definitions section

```
%{
int charCount=0, wordCount=0, lineCount=0;
%}
word   [^ \t\n]+
```

- C/C++/Java code:
  - Surrounded by **%{… %}** delimiters
  - Declare any variables used in actions
- RE definitions:
  - Define shorthand for patterns:

    digit [0-9]

    letter [a-z]

    ident {*letter*}({*letter*}|{*digit*})*
  - Use shorthand in RE section: {*ident*}

Lex Regular Expressions

```
{word} {wordCount++; charCount += strlen(yytext); }
[\n]   {charCount++; lineCount++;}
.      {charCount++;}
```

- Match explicit character sequences
  - integer, “+++”, \<\>
- Character classes
  - [abcd]
  - [a-zA-Z]
  - [^0-9] – matches non-numeric

- Alternation
  - twelve | 12
- Closure
  - * - zero or more
  - + - one or more
  - ? – zero or one
  - {*number*}, {*number*,*number*}

- Other operators
  - . – matches any character except newline
  - ^ - matches beginning of line
  - $ - matches end of line
  - / - trailing context
  - () – grouping
  - {} – RE definitions

Lex Operators

Highest: closure
concatenation
alternation

Special lex characters:
- \ / * + > “ { } . $ ( ) | % [ ] ^

Special lex characters inside [ ]:
- \ [ ] ^

Examples

- a.*z
- (ab)+
- [0-9]{1,5}
- (ab|cd)?ef = *abef,cdef,ef*
- -?[0-9]\.[0-9]

Lex Actions

Lex actions are C (C++, Java) code to implement some required functionality

- Default action is to echo to output
- Can ignore input (empty action)
- ECHO – macro that prints out matched string
- *yytext* – matched string
- *yyleng* – length of matched string (not all versions have this)

In Java:
yytext() and
yytext().length()

User Subroutines

```
main() {
   yylex();
  printf("Characters %d, Words: %d, Lines: %d\n",charCount,
          wordCount, lineCount);
}
```

- C/C++/Java code
- Copied directly into the lexer code
- User can supply 'main' or use default

How Lex works

Lex works by processing the file one character at a time, trying to match a string starting from that character.

1. Lex *always* attempts to match the longest possible string.
2. If two rules are matched (and match strings are same length), the first rule in the specification is used.

Once it matches a string, it starts from the character after the string

Lex Matching Rules

1. Lex *always* attempts to match the longest possible string.

```
beg      {…}
begin    {…}
in       {…}
```

Input ‘begin’ can match either of the first two rules.
The second rule will be chosen because of the length.

Lex Matching Rules

2. If two rules are matched (the matched strings are same length), the first rule in the specification is used.

```
begin    {… }
[a-z]+   {…}
```

Input ‘begin’ can match both rules – the first one will be chosen

Lex Example: Extracting white space

```
%{
#include <stdio.h>
%}
%%
[ \t\n]          ;
.           {ECHO;}
%%
```

To compile and run above (simple.l):

flex simple.l
gcc lex.yy.c –ll
a.out < input

Input:

This is a file

of stuff we want   to extract all

white space from

Output:

Thisisafileofstuffwewantoextractallwhitespacefrom

Lex (C/C++)

- Lex always creates a file ‘lex.yy.c’ with a function yylex()
- -ll directs the compiler to link to the lex library
- The lex library supplies external symbols referenced by the generated code
- The lex library supplies a default main:

```
main(int argc,char *argv[]) {return yylex(); }
```


Lex Example 2: Unix wc

```
%{ int charCount=0, wordCount=0, lineCount=0;
%}
word   [^ \t\n]+
%%
{word} {wordCount++; charCount += strlen(yytext); }
[\n]   {charCount++; lineCount++;}
.      {charCount++;}
%%
main() {
    yylex();
    printf(“Characters %d, Words: %d, Lines: %d\n”,
              charCount, wordCount, lineCount);
}
```

Lex Example 3: Extracting tokens

```
%%
and                     return(AND);
array                   return(ARRAY);
begin                   return(BEGIN);
.
.
.
\[                      return('[');
":="                    return(ASSIGN);
[a-zA-Z][a-zA-Z0-9_]*   return(ID);
[+-]?[0-9]+             return(NUM);
[ \t\n]                 /* nothing */ ;
%%
```


Uses for Lex

- **Transforming Input** – convert input from one form to another (example 1). *yylex*() is called once; return is not used in specification
- **Extracting Information** – scan the text and return some information (example 2). *yylex*() is called once; return is not used in specification.
- **Extracting Tokens** – standard use with compiler (example 3). Uses return to give the next token to the caller.