CS 536 Announcements for Thursday, January 30, 2025

Course websites:

`pages.cs.wisc.edu/~hasti/cs536/epic`
`www.piazza.com/wisc/spring2025/compsci536`

Programming Assignment 1

- test code due Sunday, Feb. 2 by 11:59 pm
- other files due Thursday, Feb. 6 by 11:59 pm

Last Time

- intro to CS 536
- compiler overview
- start scanning
- finite state machines
    - formalizing finite state machines
    - coding finite state machines
    - deterministic vs non-deterministic FSMs

Today

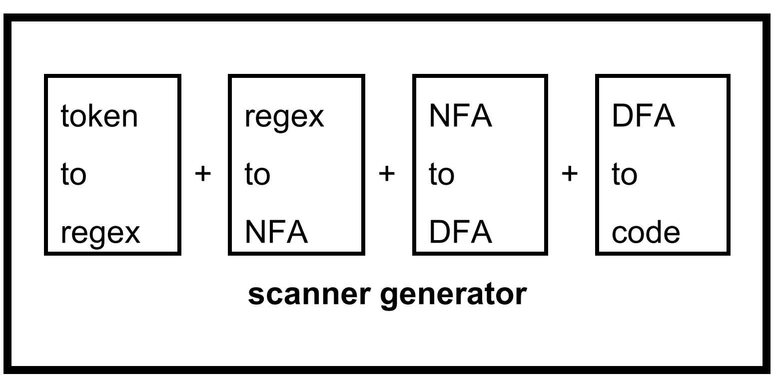
- non-deterministic FSMs
- equivalence of NFAs and DFAs
- regular languages
- regular expressions
- regular expressions → DFAs
- language recognition → tokenizers
- scanner generators
- JLex

Recall

- scanner : converts a sequence of characters to a sequence of tokens
- scanner implemented using FSMs
- FSMs can be DFA or NFA

Creating a scanner

scanner =



NFAs, formally

finite state machine *M* = (*Q*, Σ, δ, *q*, *F*)

L(*M*) = the language of FSM *M* = set of all strings *M* accepts

Example:

"Running" an NFA

To check if a string is in L(*M*) of NFA *M*, simulate set of choices it could make.

The string is in L(*M*) iff there is at least one sequence of transitions that

- consumes all input (without getting stuck) and
- ends in one of the final states

NFA and DFA are equivalent

Two automata *M* and *M** are equivalent iff L(*M*) = L(*M**)

Lemmas to be proven:

Lemma 1: Given a DFA *M*, one can construct an NFA *M** that recognizes the same language as *M*, i.e., L(*M**) = L(*M*)

Lemma 2: Given an NFA *M*, one can construct a DFA *M** that recognizes the same language as *M*, i.e., L(*M**) = L(*M*)

Proving Lemma 2

Lemma 2: Given an NFA *M*, one can construct a DFA *M** that recognizes the same language as *M*, i.e., L(*M**) = L(*M*)

Part 1: Given an NFA *M* without ε-transitions, one can construct a DFA *M** that recognizes the same language as *M*

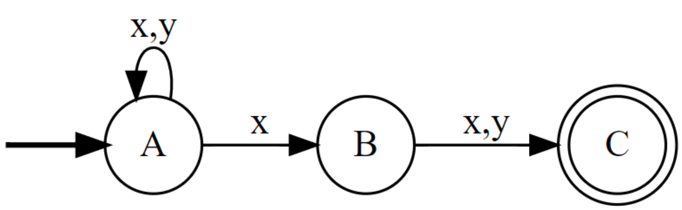
Part 2: Given an NFA *M* with ε-transitions, one can construct a NFA *M** without ε-transitions that recognizes the same language as *M*

NFA without ε-transitions to DFA

Observation: we can only be in finitely many subsets of states at any one time

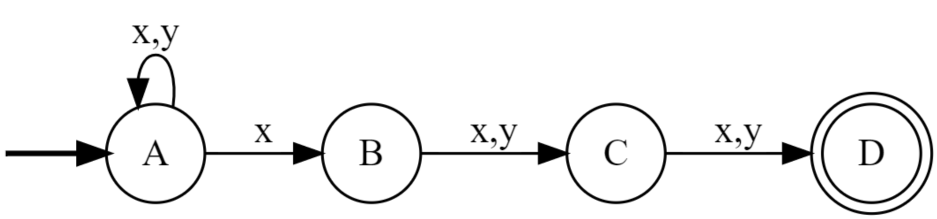
Idea: to do NFA *M* → DFA *M**, use a single state in *M** to simulate sets of states in *M*

Suppose *M* has |*Q*| states. Then *M** can have only up to states.
Why?



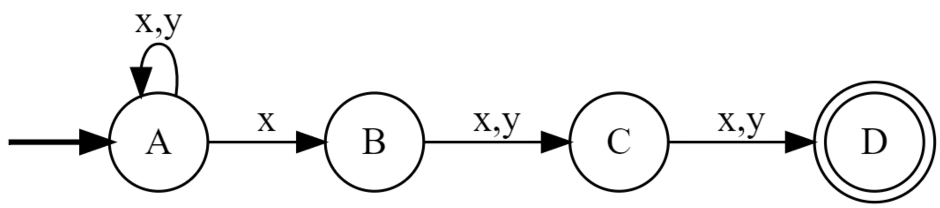
A	B	C
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

Example



NFA without ε-transitions to DFA

Given NFA *M*:

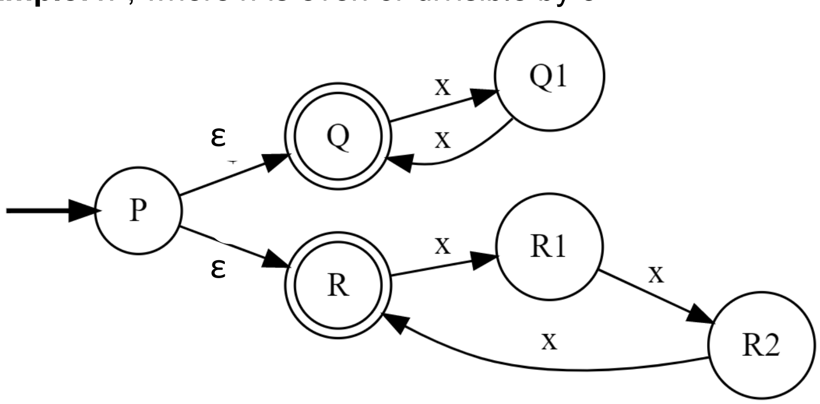


Build new DFA *M**

To build DFA: Add an edge in *M** from state *S** on character *c* to state *T** if *T** represents the set of all states that a state in *S** could possibly transition to on input *c*

ε-transitions

Example: x^n, where *n* is even or divisible by 3

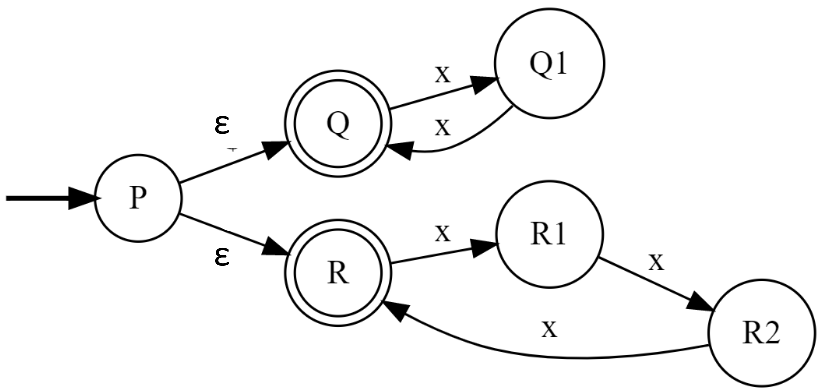


Eliminating *ε*-transitions

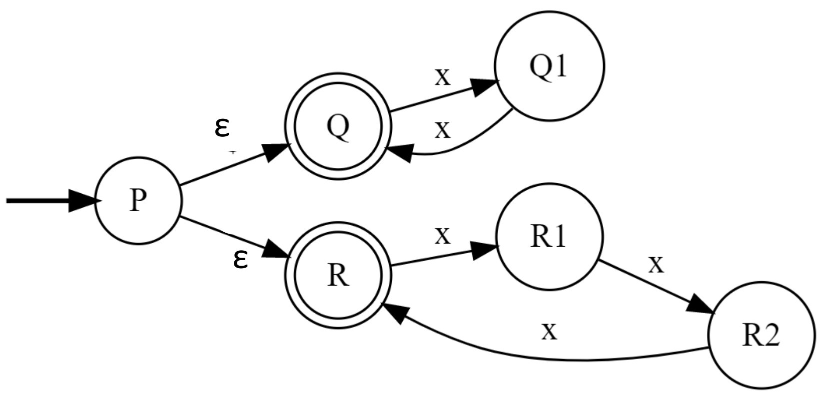
Goal: given NFA *M* with ***ε***-transitions, construct an ***ε***-free NFA *M** that is equivalent to *M*

Definition: *epsilon closure*

eclose(*S*) = set of all states reachable from *S* using 0 or more epsilon transitions



	eclose
P	
Q	
R	
Q1	
R1	
R2	



Summary of FSMs

DFAs and NFAs are equivalent

- an NFA can be converted into a DFA, which can be implemented via the table-driven approach

ε-transitions do not add expressiveness to NFAs

- algorithm to remove ε-transitions

Regular Languages and Regular Expressions

Regular language

Any language recognized by an FSM is a *regular language*

Examples:

- single-line comments beginning with //
- hexadecimal integer literals in Java
- C/C++ identifiers
- {ε, ab, abab, ababab, abababab, …}

Regular expression

= a pattern that defines a regular language

regular language: (potentially infinite) set of strings

regular expression: represents a (potentially infinite) set of strings by a single pattern

Example: {$\boldsymbol{\varepsilon}$, ab, abab, ababab, abababab, …} ←→ (ab)*

Why do we need them?

- Each token in a programming language can be defined by a regular language
- Scanner-generator input = one regular expression for each token to be recognized by the scanner

→

Formal definition

A **regular expression** over an alphabet Σ is any of the following:

- ∅ (the empty regular expression)
- ε
- *a* (for any $a \in \Sigma$)

Moreover, if R_1 and R_2 are regular expressions over Σ, then so are: $R_1 \mid R_2$, $R_1 \cdot R_2$, R_1^*

Regular expressions (as an expression language)

regular expression = pattern describing a set of strings

operands: single characters, epsilon

operators:

alternation ("or"): a | b

concatenation ("followed by"): a.b ab

iteration ("Kleene star"): a*

Conventions

aa is a.a

a+ is aa*

letter is a|b|c|d|…|y|z|A|B|…|Z

digit is 0|1|2|…|9

not(x) is all characters except x

parentheses for grouping and overriding precedence, e.g., (ab)*

Example: single-line comments beginning with //

Example: hexadecimal integer literals in Java

- must start 0x or 0X
- followed by at least one hexadecimal digit (hexdigit)
  - hexdigit = 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f, A, B, C, D, E, F
- optionally can add long specifier (`l` or `L`) at end

Example: C/C++ identifiers (with one added restriction)

- sequence of letters/digits/underscores
- cannot begin with a digit
- cannot end with an underscore

From regular expressions to NFAs

Overview of the process

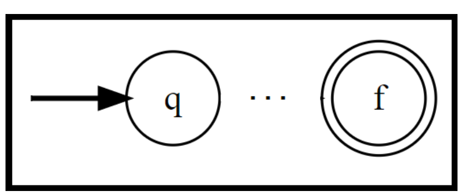
- Conversion of literals and epsilon
- Conversion of operators

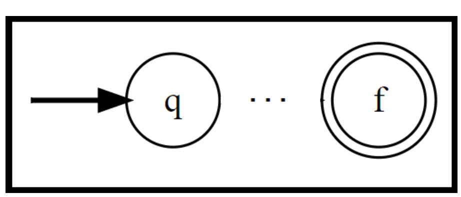
Regex to NFA rules

Rules for operands

Suppose A is a regex with NFA:

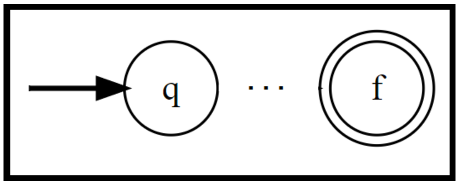
Rules for alternation A|B

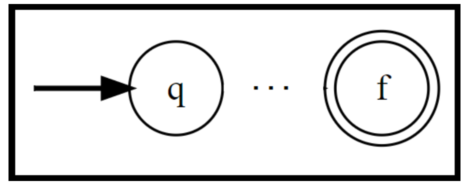




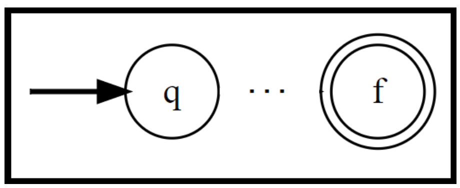
Regex to NFA rules

Rules for catenation A.B





Rules for iteration A*



Tree representation of a regex

Consider regex: (letter | '_') (letter | '_' | digit)*

Regex to DFA

We now can do:

We can add one more step: **optimize DFA**

Theorem: For every DFA *M*, there exists a unique equivalent smallest DFA *M** that recognizes the same language as *M***.**

To optimize:

- remove unreachable states
- remove dead states
- merge equivalent states

But what's so great about DFAs?

Recall: state-transition function (δ) can be expressed as a table

➔ very efficient array representation

➔ efficient algorithm for running (any) DFA

```
s = start state
while (more input){
    c = read next char
    s = table[s][c]
}
if s is final, accept
else reject
```

What else do we need?

FSMs – only check for language membership of a string

scanner needs to

- recognize a stream of many different tokens using the longest match
- know what was matched

Table-driven DFA → tokenizer

Idea: augment states with actions that will be executed when state is reached

Consider: (letter)(letter | digit)*

Problem:

Problem:

Scanner Generator Example

Language description:
consider a language consisting of two statements

- assignment statements: `ID = expr`
- increment statements: `ID += expr`

where `expr` is of the form:

- `ID + ID`
- `ID ^ ID`
- `ID < ID`
- `ID <= ID`

and `ID` are identifiers following C/C++ rules (can contain only letters, digits, and underscores; can't start with a digit)

Tokens:

Token	Regular expression
ASSIGN	
INCR	
PLUS	
EXP	
LESSTHAN	
LEQ	
ID	

Combined DFA

State-transition table

	=	+	^	<	_	letter	digit	EOF	none of these
S_0	ret ASSIGN	A	ret EXP	B	C	C		ret EOF	
A	ret INC	put 1 back, ret PLUS							
B	ret LEQ	put 1 back, ret LESSTHAN							
C	put 1 back, ret ID				C	C	C	put 1 back, ret ID	

```
do {
    read char
    perform action / update state
    if (action was to return a token)
        start again in start state
} while not(EOF or stuck)
```

Lexical analyzer generators
(aka scanner generators)

Formally define transformation from regex to scanner

Tools written to synthesize a lexer automatically

- Lex : UNIX scanner generator, builds scanner in C
- Flex : faster version of Lex
- JLex : Java version of Lex

JLex

Declarative specification

- you don't tell JLex how to scan / how to match tokens
- you tell JLex what you want scanned (tokens) & what to do when a token is matched

Input: set of regular expressions + associated actions

Output: Java source code for a scanner

Format of JLex specification

3 sections separated by %%

- user code section
- directives
- regular expression rules

Example

```
// User Code section:  For right now, we will not use it.
%%

DIGIT=       [0-9]
LETTER=      [a-zA-Z]
WHITESPACE=  [\040\t\n]

%state SPECIALINTSTATE

%implements java_cup.runtime.Scanner
%function next_token
%type java_cup.runtime.Symbol

%eofval{
System.out.println("All done");
return null;
%eofval}

%line
```

```
%%

({LETTER}|"_")({DIGIT}|{LETTER}|"_")* {
                          System.out.println(yyline+1 + ": ID "
                                     + yytext()); }

"="            { System.out.println(yyline+1 + ": ASSIGN"); }
"+"            { System.out.println(yyline+1 + ": PLUS"); }
"^"            { System.out.println(yyline+1 + ": EXP"); }
"<"            { System.out.println(yyline+1 + ": LESSTHAN"); }
"+="           { System.out.println(yyline+1 + ": INCR"); }
"<="           { System.out.println(yyline+1 + ": LEQ"); }
{WHITESPACE}*  { }
.              { System.out.println(yyline+1 + ": bad char"); }
```

Regular expression rules section

Format: `<regex>{code}` where `<regex>` is a regular expression for a single token

- can use macros from Directives section – surround with curly braces `{ }`
- characters represent themselves (except special characters)
- characters inside `" "` represent themselves (except `\"`)
- `.` matches anything

Regular expression operators: `|` `*` `+` `?` `( )`

Character class operators: `-` `^` `\`

Using scanner generated by JLex in a program

```
// inFile is a FileReader initialized to read from the
// file to be scanned
Yylex scanner = new Yylex(inFile);
try {
    scanner.next_token();
} catch (IOException ex) {
    System.err.println(
              "unexpected IOException thrown by the scanner");
    System.exit(-1);
}
```