CS 536 Announcements for Monday, February 5, 2024

Programming Assignment 1

- symbol table files due Thursday, Feb. 8 by 11:59 pm

Homework 0

- available in schedule
- practice with DFAs, regular expressions

Homework 1

- available tomorrow
- practice with NFA→DFA translation, JLex

Last Time

- non-deterministic FSMs
- equivalence of NFAs and DFAs
- regular languages
- regular expressions

Today

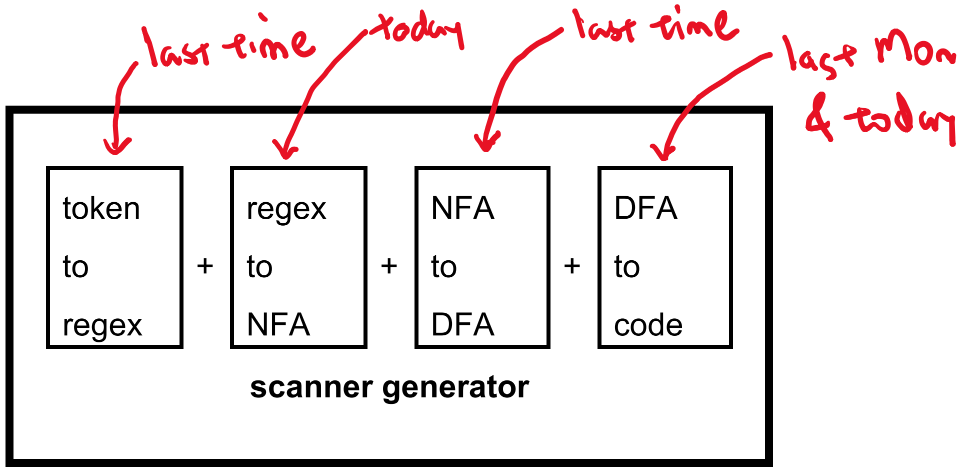
- regular expressions → DFAs
- language recognition → tokenizers
- scanner generators
- JLex

Next Time

- CFGs

Recall

scanner =



token to regex + regex to NFA + NFA to DFA + DFA to code

scanner generator

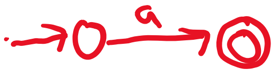
From regular expressions to NFAs

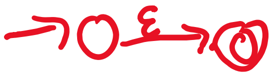
Overview of the process

- Conversion of literals and epsilon → simple FAs
- Conversion of operators
  - convert operands to NFAs
  - join NFAs

Regex to NFA rules

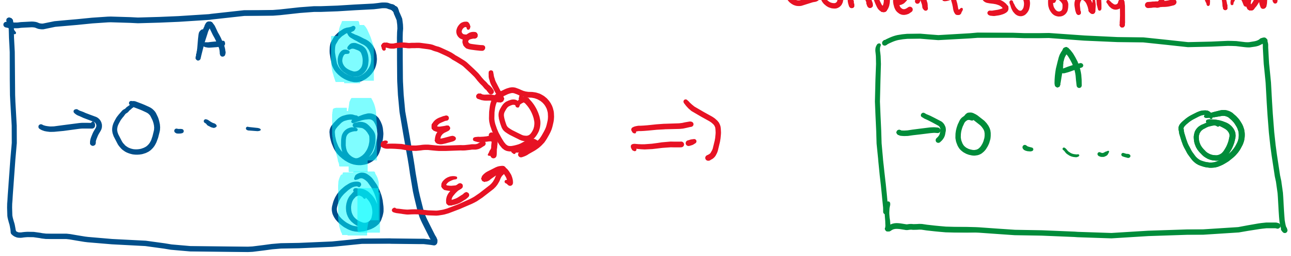
Rules for operands

literal 'a' 

epsilon ε 

Suppose A is a regex with NFA:

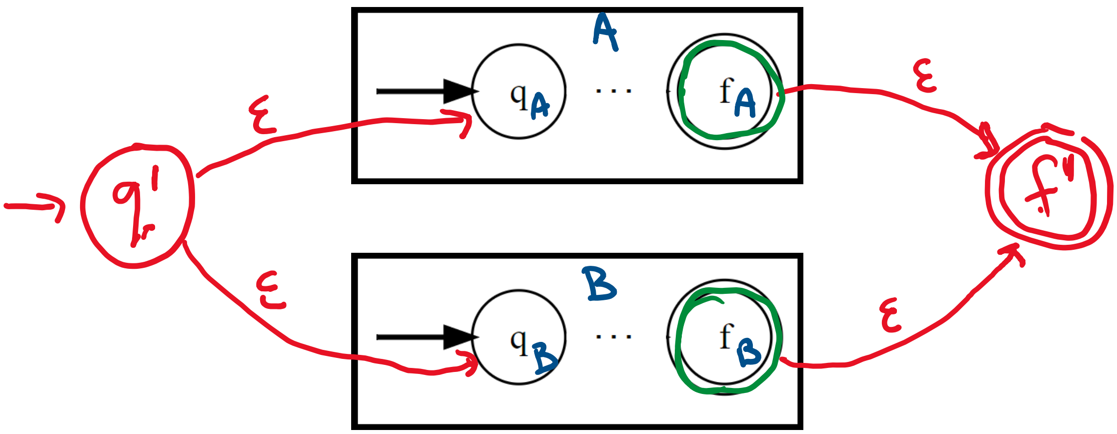
Convert so only 1 final state



make these non-final

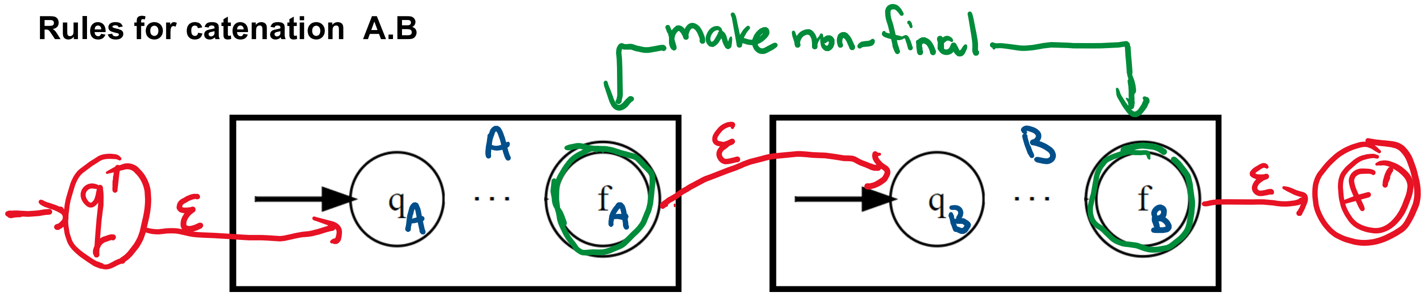
Rules for alternation A|B

make f_A, f_B non-final

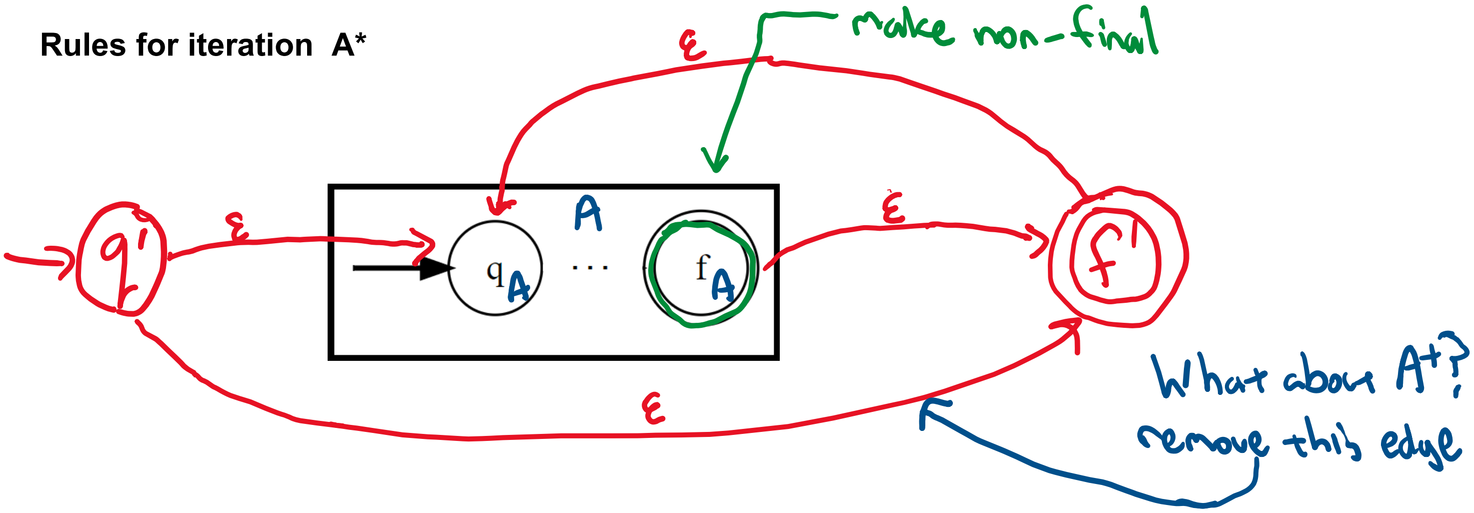


Regex to NFA rules

Rules for catenation A.B

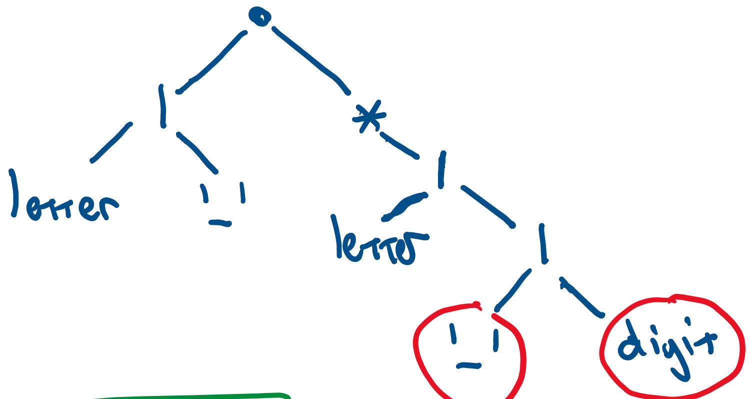


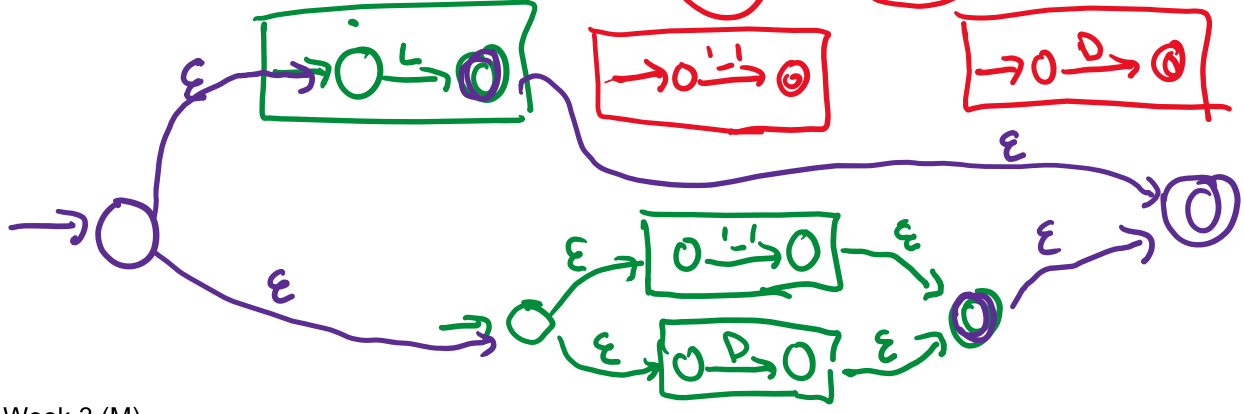
Rules for iteration A*



Tree representation of a regex

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





Regex to DFA

We now can do:

regex → NFA w/ε → NFA w/o ε → DFA

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

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

↳ fewest # of states

To optimize:

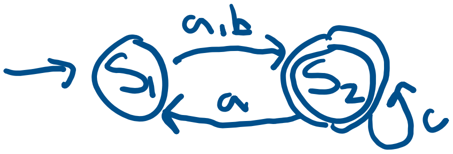
- remove unreachable states → can't get to from start state
- remove dead states → can't get to a final state from it
- merge equivalent states

↳ same transitions (out) w/ same labels

But what's so great about DFAs?

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

➔ very efficient array representation



	a	b	c
S1	S2	S2	
S2	S1		S2

➔ 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

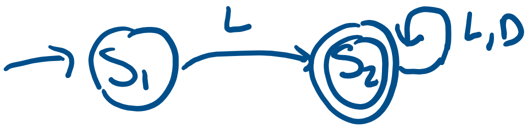
scanner needs to

- 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

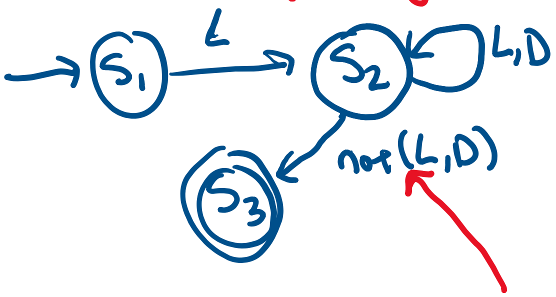
Consider: (letter)(letter | digit)*



state	action
S2	return ID ← token

Problem: Don't get longest match

avg=7



state	action
S3	~~return ID~~

To fix: put back 1 char, return ID

avg=7

Problem: maybe we need this char

Actions needed:
- return a token
- put back a character
- report an error

Also add EOF token, EOF symbol to alphabet Σ

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:

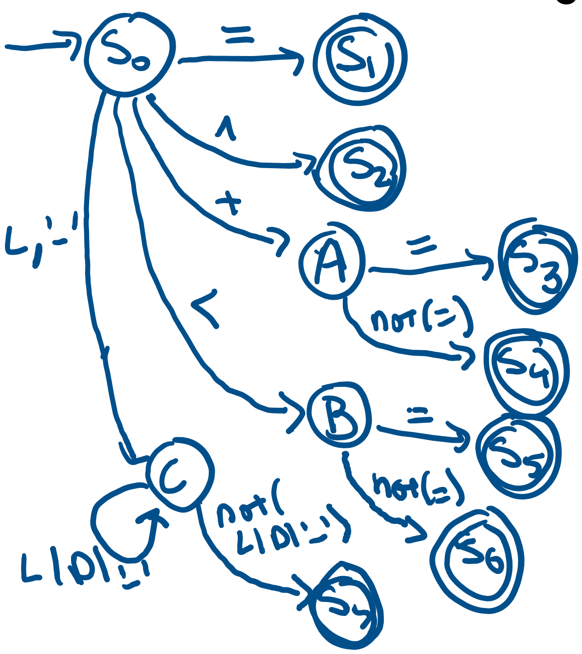
- `ID + ID`
- `ID ^ ID`
- `ID < ID`
- `ID <= ID`

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	(letter | '_')(letter | '_' | digit)*

Combined DFA



Actions

S_1: return ASSIGN

S_2: return EXP

S_3: return ING

S_4: put 1 back, return PLUS

S_5: return LEQ

S_6: put 1 back, return LESSTHAN

S_7: put 1 back, return ID

State-transition table

	=	+	^	<	_	letter	digit	EOF	none of these
S0	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 (non-procedure)

- 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,

JLex specification

— .jlex eg xyz.jlex

Output: Java source code for a scanner

↳ xyz.jlex.java — compile to get Yylex.class

- ctor: takes input stream as arg

- next_token: return next token of input

Format of JLex specification

3 sections separated by %%

- user code section
- directives
- regular expression rules

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: `-` `^` `\`

JLex example

```
// This file contains a complete JLex specification for a very
// small 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"); }
```

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);
}
```