CS3200 - Scanning & parsing

Note Title 1/22/2016

Announcements

- flex HW - due next Sat.

Last week: Ch 2 of book

- Scanners and regular expressions
  - DFAs + NFAs
  - Flex

- Mentioned parsing, which is the next step (more today)

Ex: Give the regular expression for $\{w \mid w$ begins with a 1 and ends with a 0$\}$

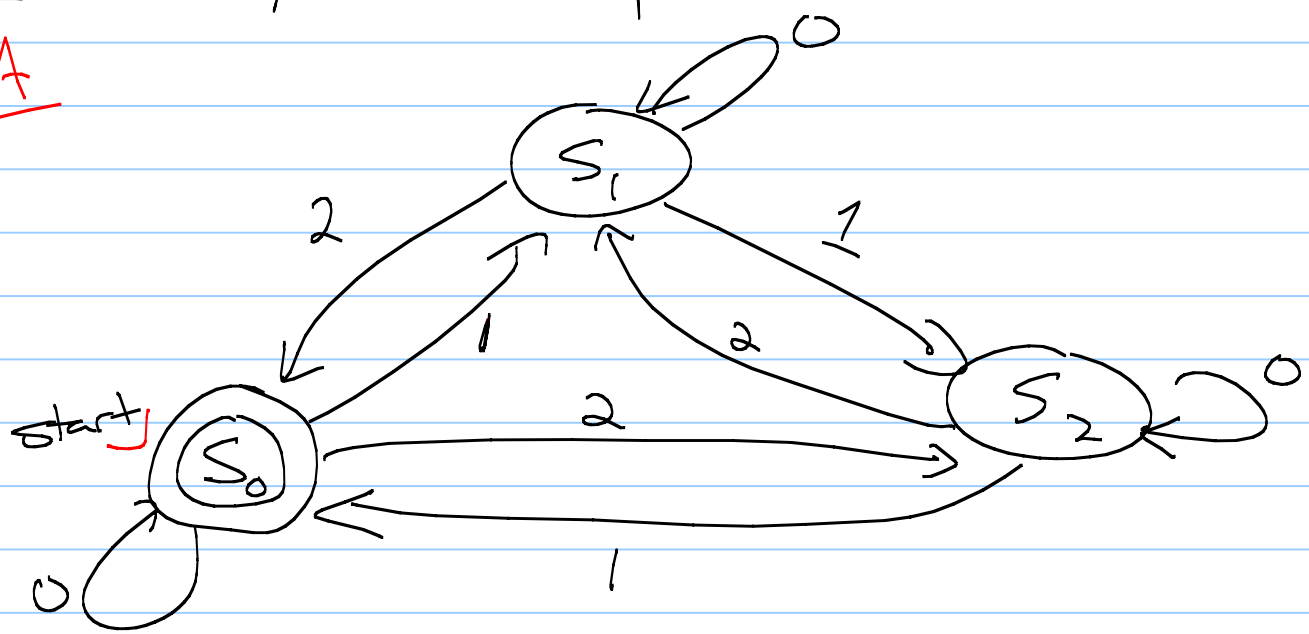
$$1(0|1)^*0$$

Ex: $\{w \mid w$ starts with 0 and has an odd length$\}$

$$0((0|1)(0|1))^*$$

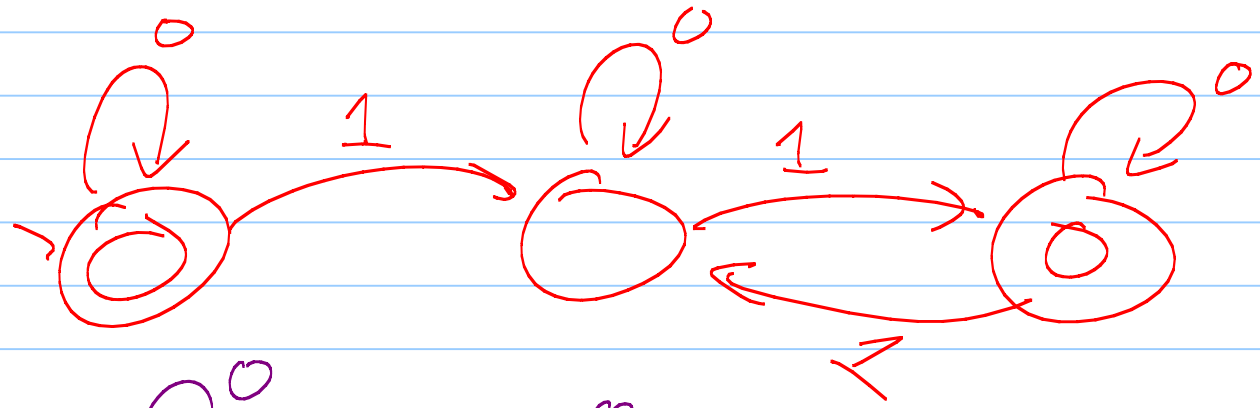
Ex: 3 symbol alphabet: $\{0, 1, 2\}$

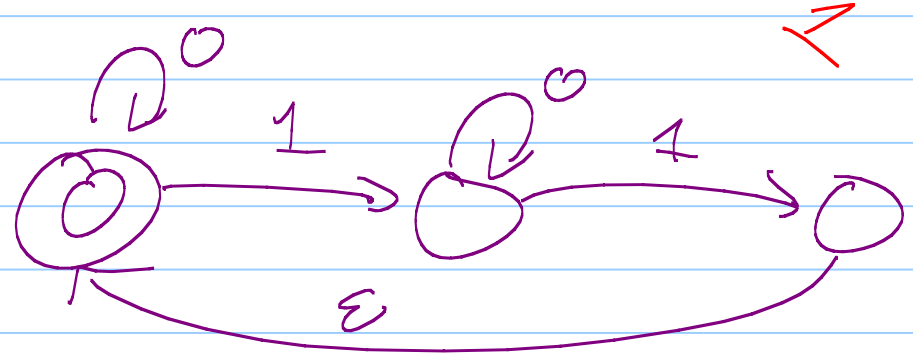
DFA



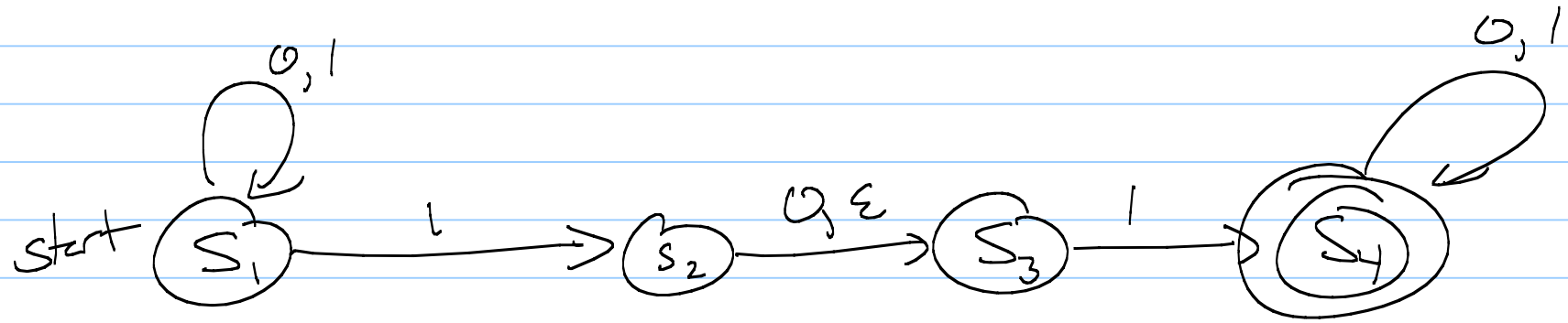
are you $\equiv 0 \bmod 3$?

Ex: String of 0's & 1's:
accept if number of 1's is <u>even</u>



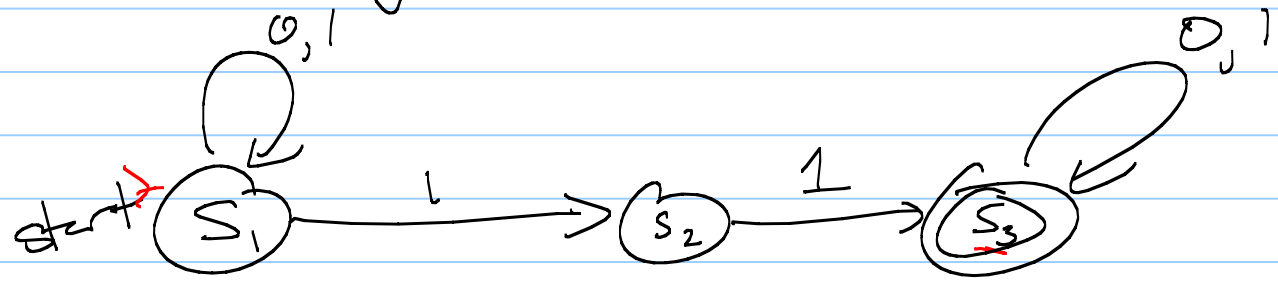


NFAs: DFAs w/ ambiguity

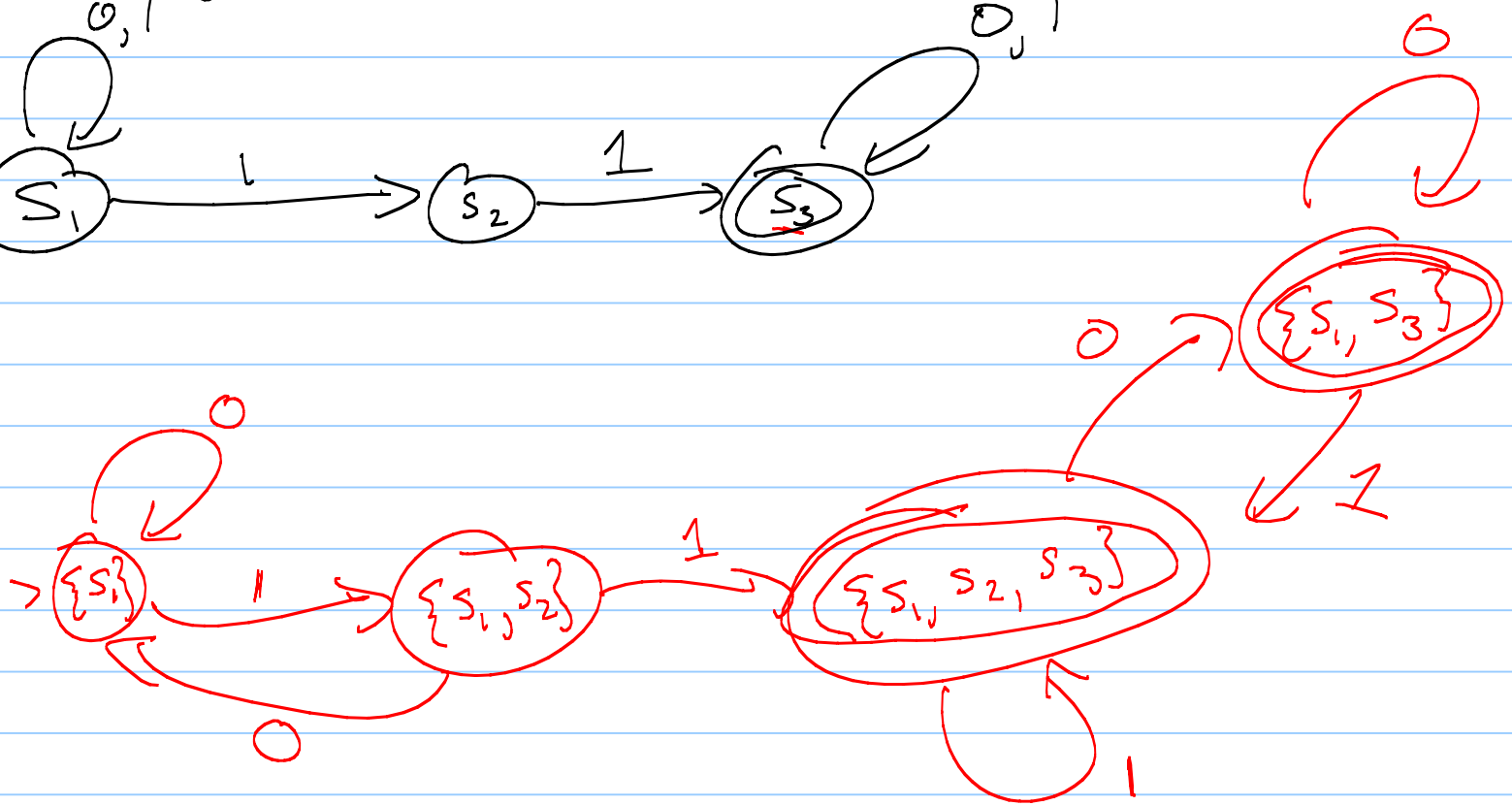


Thm: NFAs & DFAs are equivalent; both accept reg. languages.

Converting NFAs to DFAs (p.57)



DFA:



Generating Scanners

Usually, easier to specify a regular expression.

But code needs to emulate a DFA – NFAs are too complex.

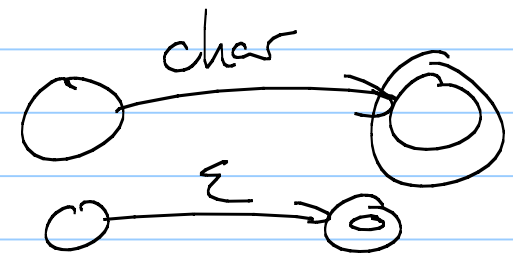
Unfortunately, there's not a great way to convert from reg exp $\longrightarrow$ DFA.

However, NFAs are easy!

Constructing an NFA (p.57ish)

Given a regular expression, we can construct an NFA.

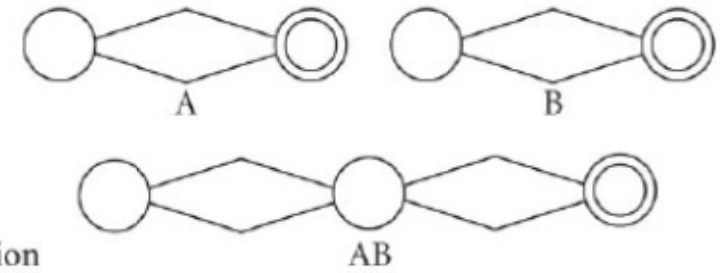
Simple NFA:



(Base case)

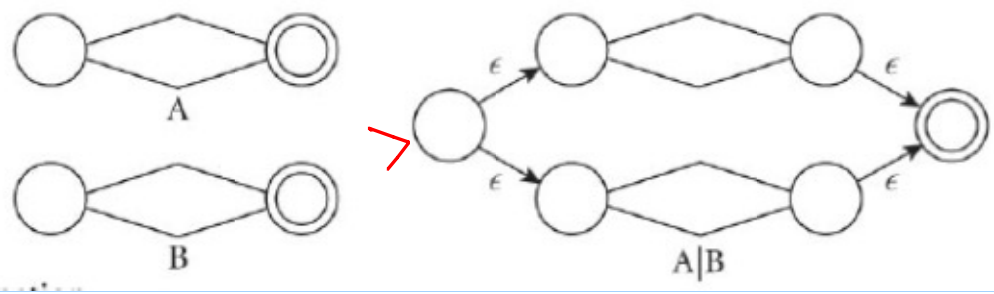
3 operations

Concatenation:

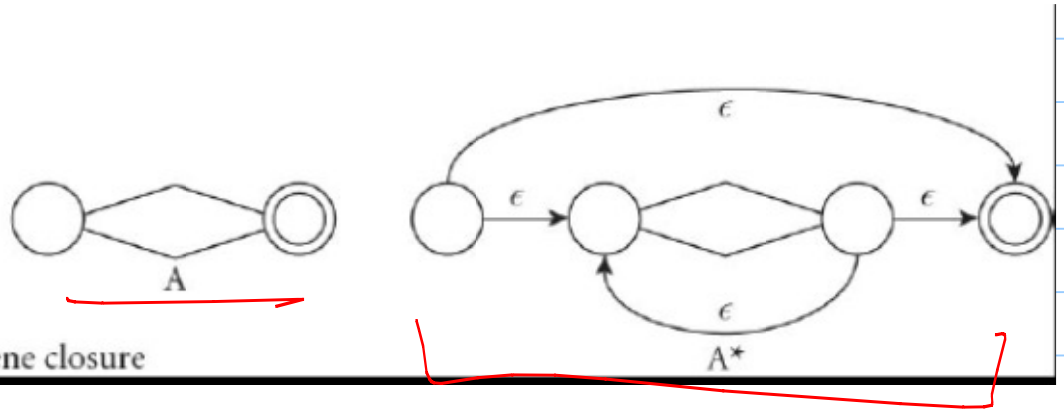


concatenation

Or:



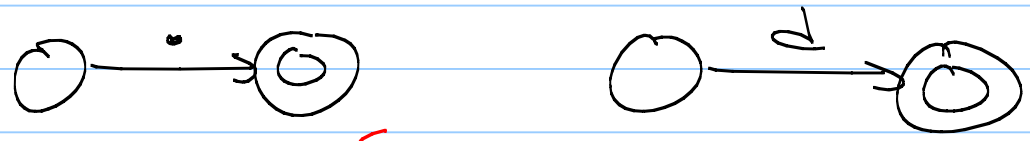
and Kleene closure (*).

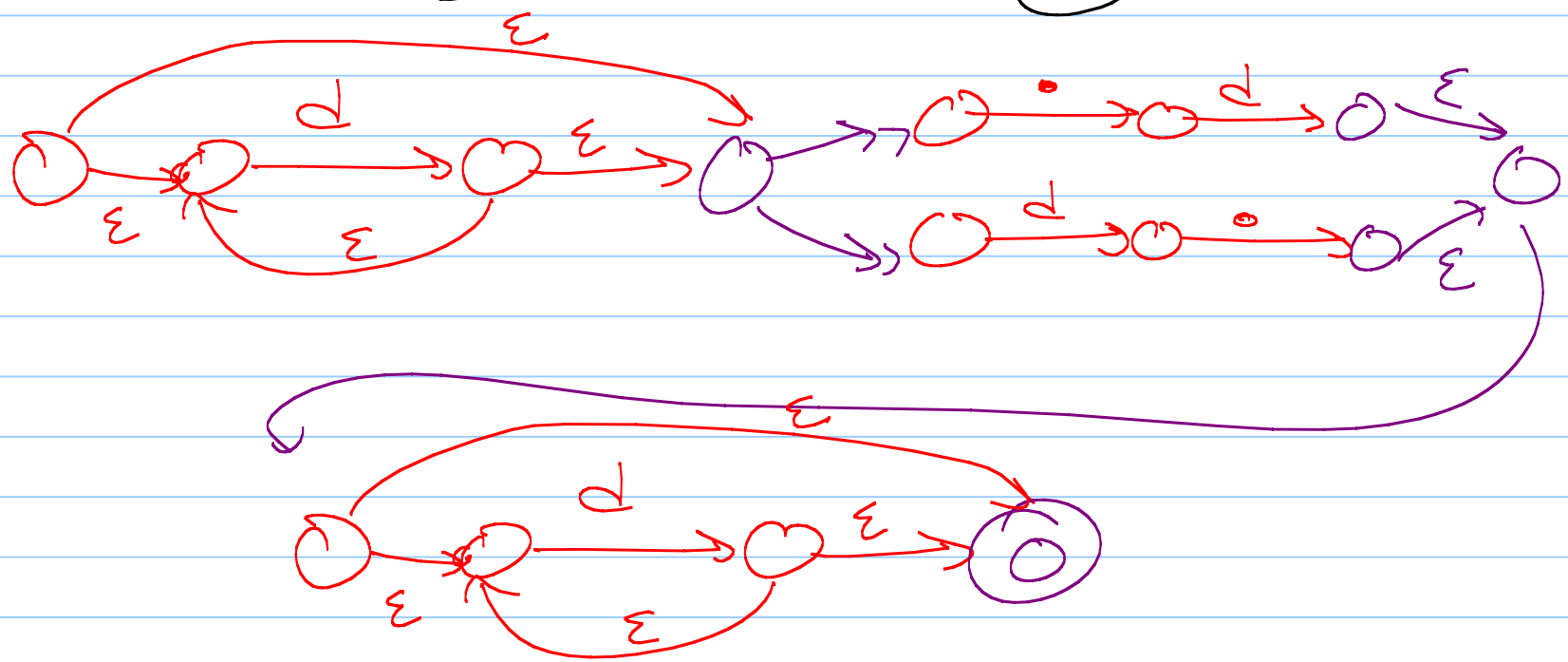


d) Kleene closure

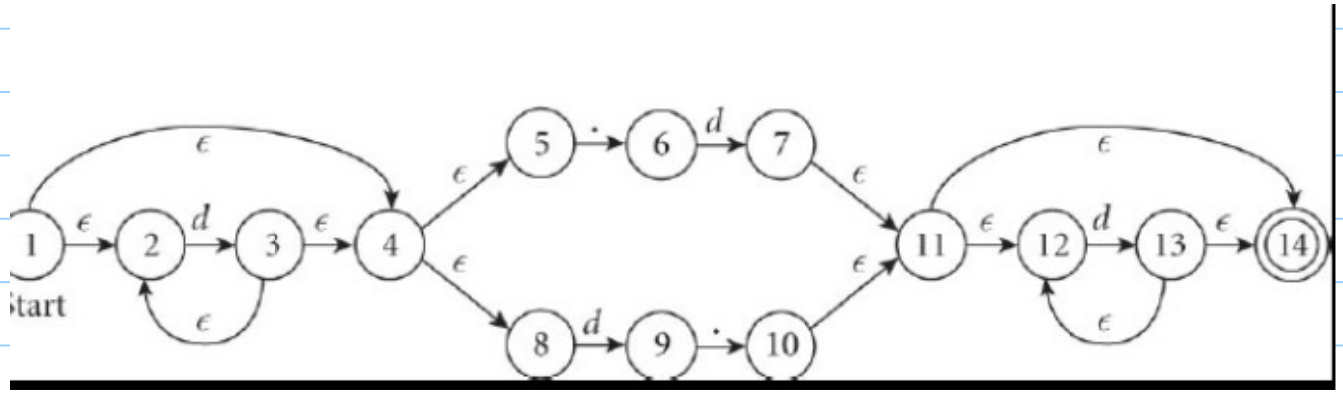
Example: decimals $d^*(.d \mid d.)\,d^*$

Base:



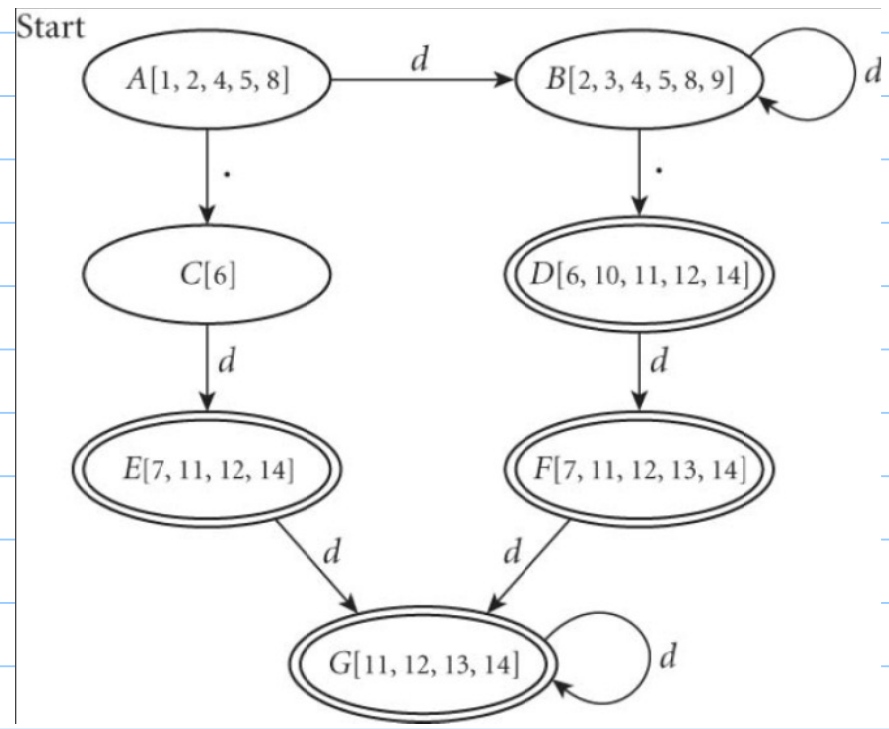


Final product:



Next: Convert to DFA.
(lots of states, but same principle as we saw earlier.)

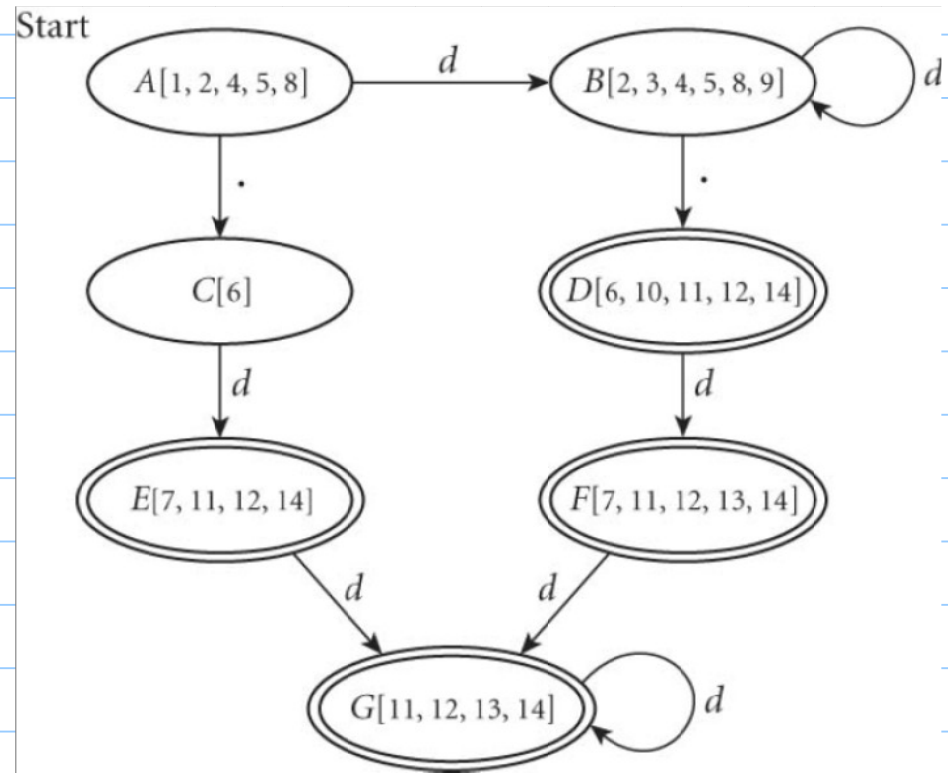
Result:
(see p. 57-58)



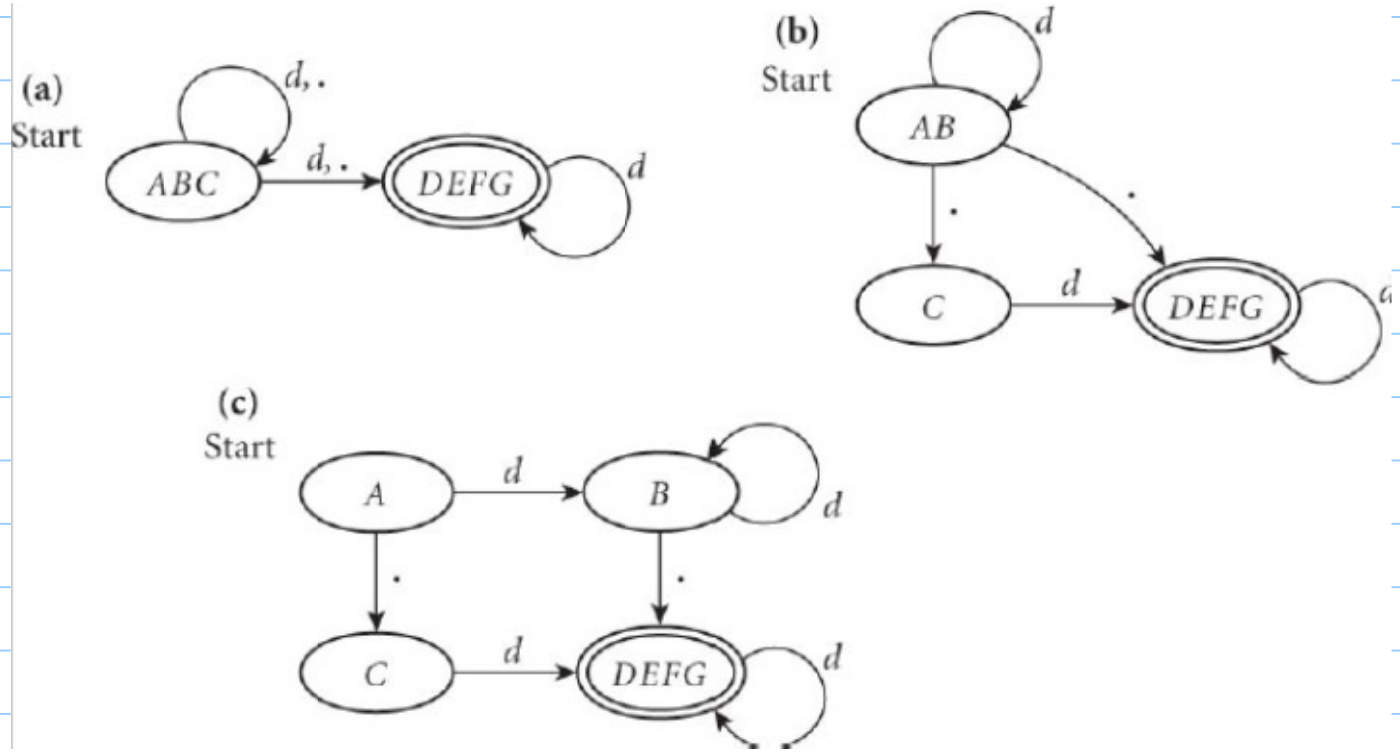
Note: This DFA is a bit redundant.

Not minimal.

Can easily find the equivalence classes and minimize.



Process to minimize (p. 59)



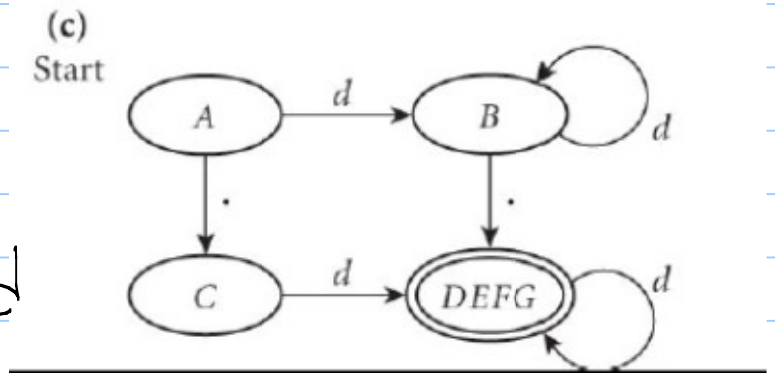
Now:

Given DFA, generate case statements to simulate it.

```
state = 1
repeat;
    read curr_char
    case state is:

        A:   case curr_char = d
                 state = B
             case curr_char = .
                 state = C

        B:
```



<u>So</u>: End result:

Automatic generation of a DFA in some language given reg. expression input.

(<u>This</u> is what flex is doing for you!)

Limitations of Regular Expressions

Certain languages are <u>not</u> regular.

<u>Ex:</u> $\{w \mid w$ has an equal number of 0's and 1's$\}$

Somehow, this needs a type of memory, which regular expressions do not have.

<u>Ex:</u> $0^n 1^n$

Why do we need this?

Need to "nest" expressions.

<u>Ex:</u> expr → id | number | -expr | (expr) | expr op expr

op → + | - | / | *

Regular expressions can't quite do this.

non-regular

Context Free Languages

Described in terms of productions
(Called Backus-Naur Form, or BNF)

- A set of terminals T
- A set of non-terminals N
- A start symbol S (a non-terminal)
- A set of productions

Ex: $\{0^n 1^n \mid n > 0\} = L$

terminals

non-terminals (capital letters)

$S \rightarrow 0S1$

$S \rightarrow 01$

Q: Is $0011 \in L$?

yes: $S \rightarrow 0S1 \rightarrow 0011$ ✓

Is $00011 \in L$? No

won't do palindromes!

Ex: $\{ w \mid w$ has an equal number of 0's & 1's $\}$

$S \to 0S1$

$S \to 1S0$

$S \to \varepsilon$

$S \to 10S$

$S \to 01S$

Q: is this enough?

$S \to S01$

$S \to S10$

$\underbrace{1010}$

$S \to 1S0$

$\to 1\underbrace{0S1}0$

$\to 1010$

Expression grammars: Simple calculator

expr → term | expr add_op term

term → factor | term mult_op factor

factor → id | number | -factor | (expr)

add_op → + | -

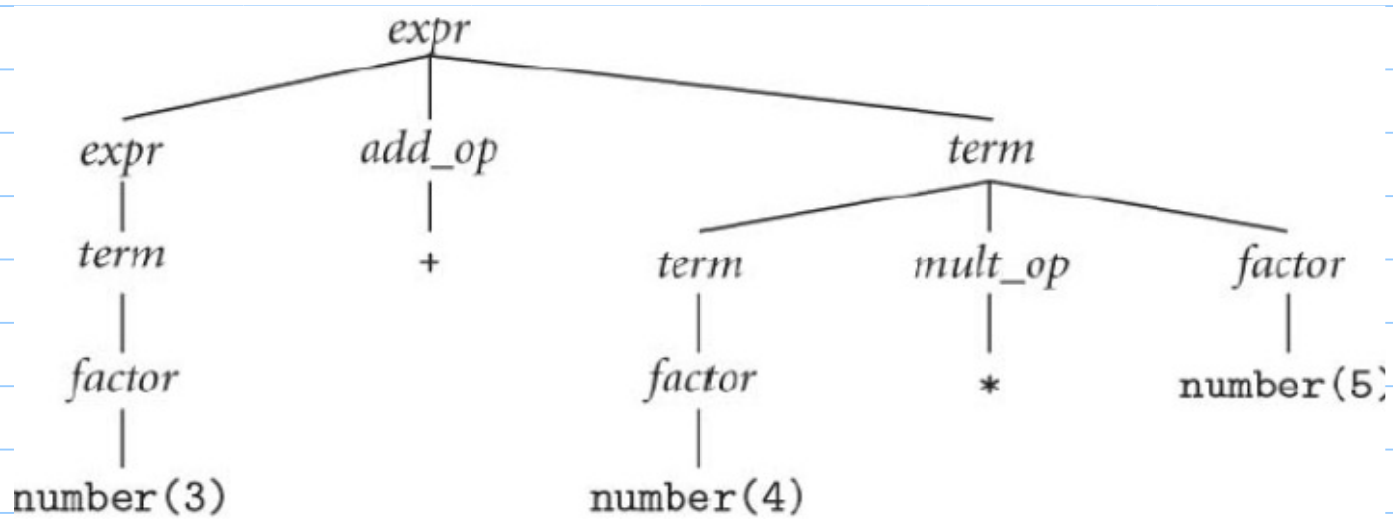
mult_op → * | /

<u>Example</u>: Show how rules can generate $3 + 4 * 5$

Parse Tree

Ex: 3 + 4 * 5

derivations



Parsers:

There are also tools to do this for you.

For flex, the complementary tool is bison:

Example:

Consider a simple calculator. Needs to accept valid numerical expressions, & then compute results.

Flex: a tokenizer for a calculator:

```
/* recognize tokens for the calculator and print them out */
%{
   enum yytokentype {
     NUMBER = 258,
     ADD = 259,
     SUB = 260,
     MUL = 261,
     DIV = 262,
     ABS = 263,
     EOL = 264
   int yylval;
%}

%%
"+"    { return ADD; }
"-"    { return SUB; }
"*"    { return MUL; }
"/"    { return DIV; }
"|"    { return ABS; }
[0-9]+ { yylval = atoi(yytext); return NUMBER; }
\n     { return EOL; }
[ \t]  { /* ignore whitespace */ }
.      { printf("Mystery character %c\n", *yytext); }
%%
main(int argc, char **argv)
{
  int tok;

  while(tok = yylex()) {
    printf("%d", tok);
    if(tok == NUMBER) printf(" = %d\n", yylval);
    else printf("\n");
  }
}
```

In Action:

```
$ flex fb1-4.l
$ cc lex.yy.c -lfl
$ ./a.out
a / 34 + |45
Mystery character a
262
258 = 34
259
263
258 = 45
264
```

Bison accepts these tokens:

```
/* simplest version of calculator */
%{
#include <stdio.h>
%}

/* declare tokens */
%token NUMBER
%token ADD SUB MUL DIV ABS
%token EOL

%%

calclist: /* nothing */
 | calclist exp EOL { printf("= %d\n", $1); }
 ;

exp: factor          default $$ = $1
 | exp ADD factor { $$ = $1 + $3; }
 | exp SUB factor { $$ = $1 - $3; }
 ;

factor: term         default $$ = $1
 | factor MUL term { $$ = $1 * $3; }
 | factor DIV term { $$ = $1 / $3; }
 ;

term: NUMBER   default $$ = $1
 | ABS term    { $$ = $2 >= 0? $2 : - $2; }
;
%%
main(int argc, char **argv)
{
  yyparse();
}

yyerror(char *s)
{
  fprintf(stderr, "error: %s\n", s);
}
```

CFL

Building:

```
# part of the makefile
fb1-5:  fb1-5.l fb1-5.y
        bison -d fb1-5.y
        flex fb1-5.l
        cc -o $@ fb1-5.tab.c lex.yy.c -lfl
```

Running:

```
$ ./fb1-5
2 + 3 * 4
= 14
2 * 3 + 4
= 10
20 / 4 - 2
= 3
20 - 4 / 2
= 18
```

Back to what Bison is:

```
exp: factor          default $$ = $1
  | exp ADD factor { $$ = $1 + $3; }
  | exp SUB factor { $$ = $1 - $3; }
  ;

factor: term         default $$ = $1
  | factor MUL term { $$ = $1 * $3; }
  | factor DIV term { $$ = $1 / $3; }
  ;
```

nice

Essentially, this is a CFG!

But only works on a particular type of grammar...