CS 344 – Scanning

Note Title 1/24/2012

Announcements

- HW2 is due in 1 week

Last time: Regular expressions

- A character
- The empty string, ε
- 2 regular expressions concatenated
- 2 regular expressions separated by an or (written |)
- A regular expression followed by * (Kleene star – 0 or more occurrances)

} base cases

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$\}$

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

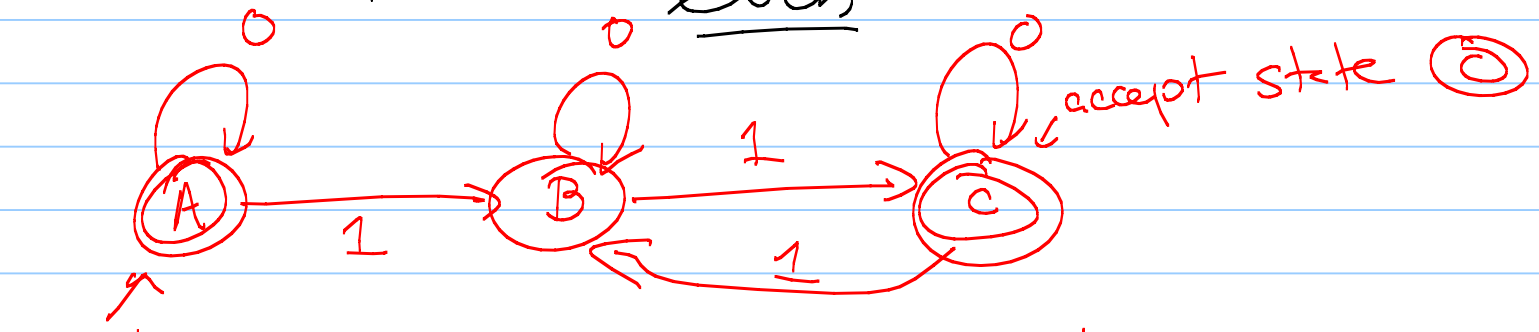
Deterministic Finite Automata (DFA)

Regular languages are precisely the things recognized by ~~DFAs~~.

- A set of states
- input alphabet
- A start state
- A set of accept states
- A transition function: given a state & an input, output a new state

00101 ∈ L

Ex: String of 0's & 1's:
L = accept if number of 1's is even

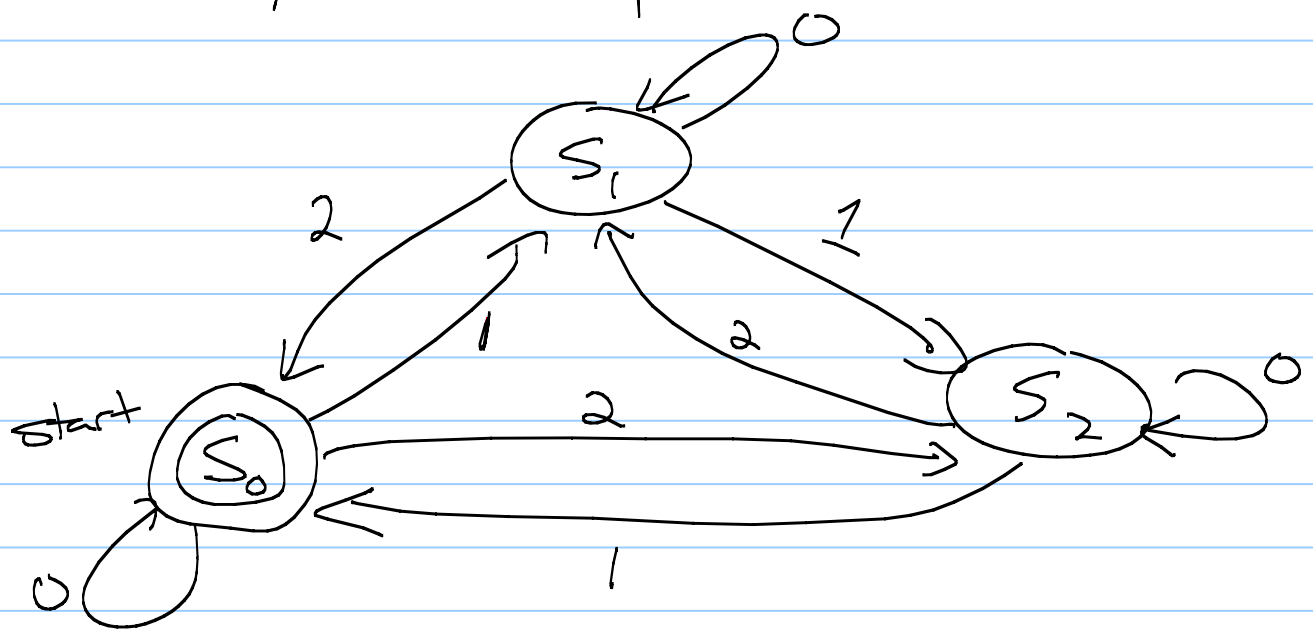


111 ∉ L

1111 ∈ L

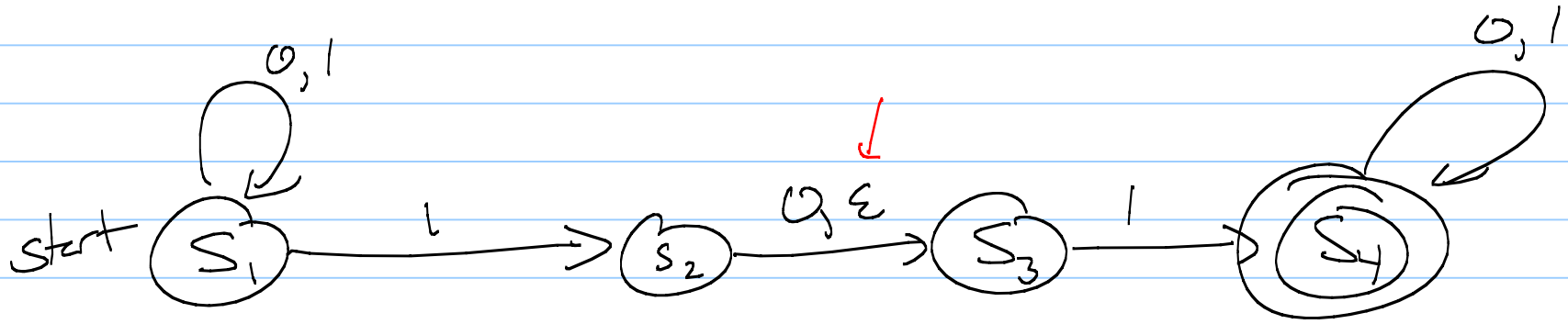
reg exp: $(0^*10^*10^*)^*$

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



Counts modulo 3
accepts words w/ sum $\equiv 0 \bmod 3$

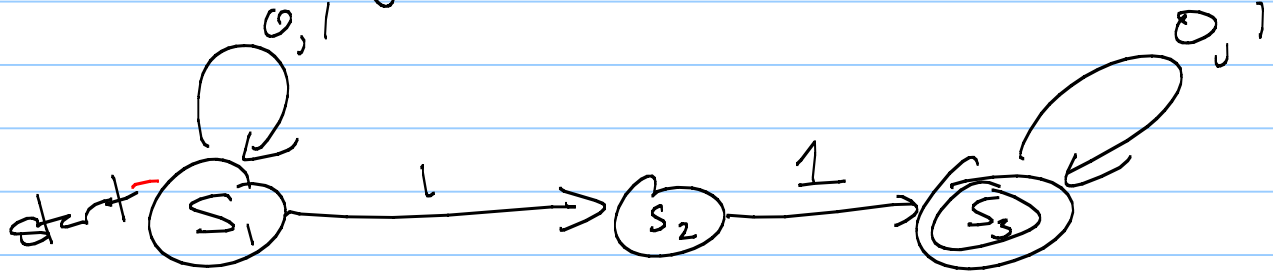
NFAs: DFAs w/ ambiguity



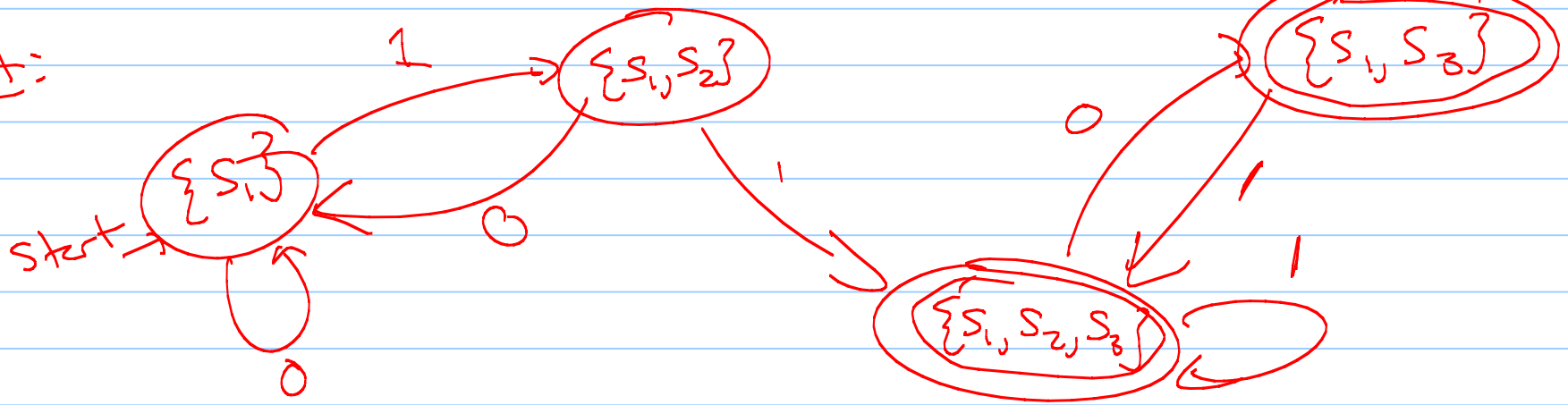
→ 10

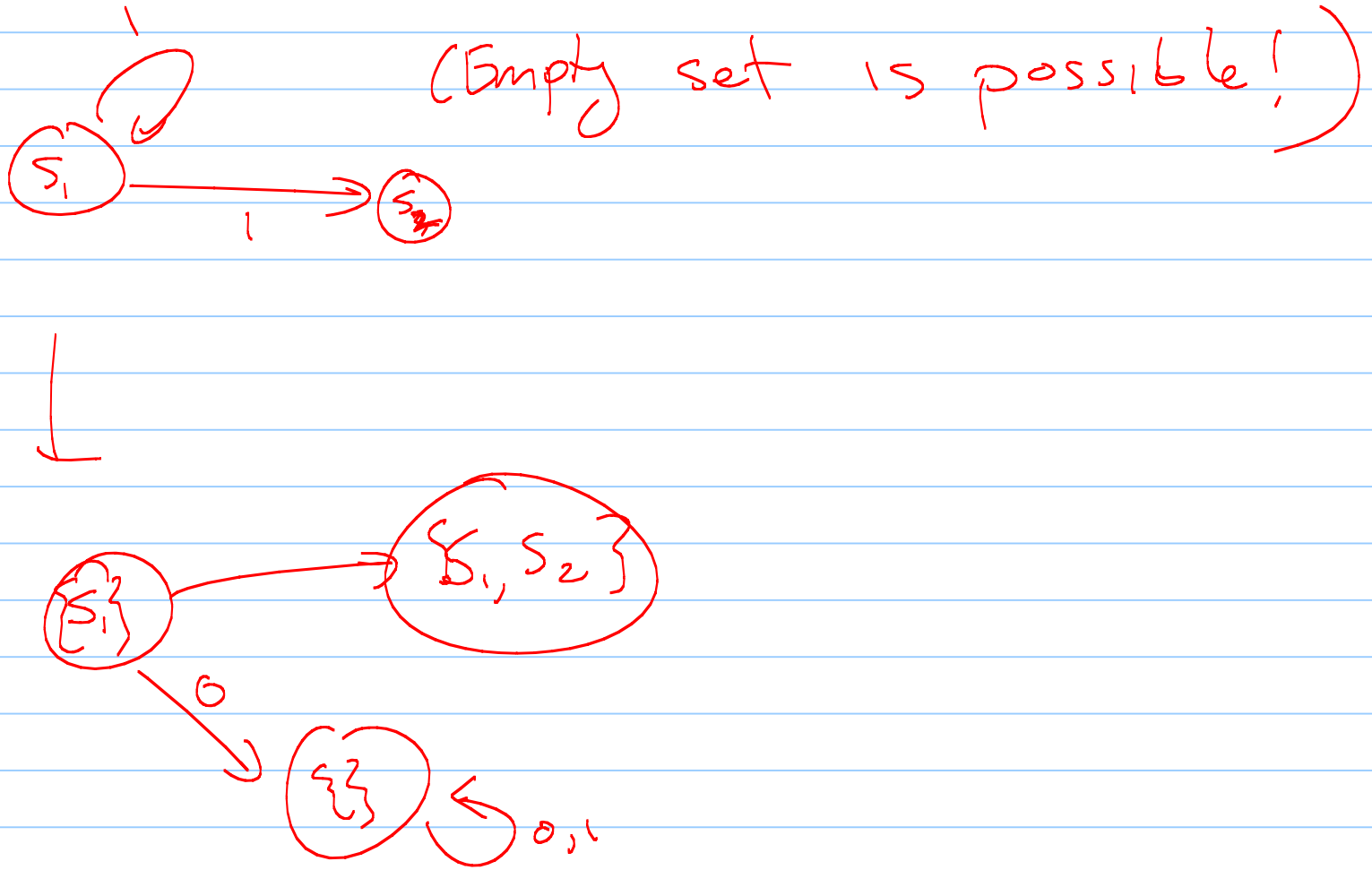
Thm: NFAs + DFAs are equivalent.

Converting NFAs to DFAs (p.57)



DFA:




(Empty set is possible!)
S_1
1
S_2
$\{S_1\}$
$\{S_1, S_2\}$
0
$\{\}$
0,1

Context free Grammars (aka BNF)

Ex:

nonterminals

Expr → Expr Op Expr | (Expr) | - Expr | id | number

terminals

Op → + | - | * | /

A derivation: derive ½ * x + intercept

variables (ids)

Expr ⇒ Expr Op Expr

⇒ Expr + Expr

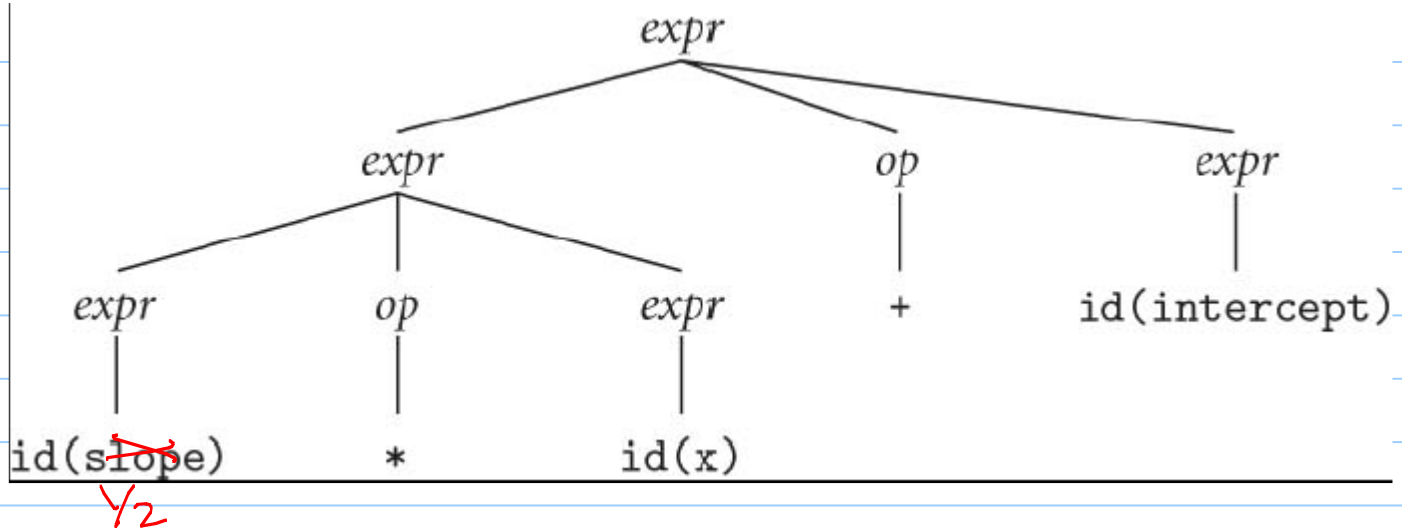
⇒ Expr + id(intercept)

⇒ Expr Op Expr + id(intercept)

⇒ Expr * Expr + id(intercept)

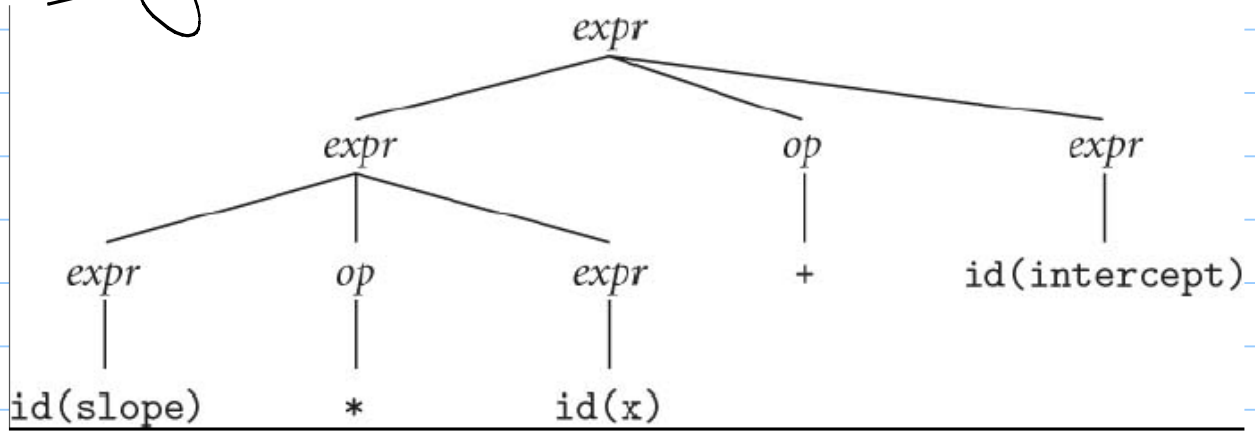
⇒ number(½) * id(x) + id(intercept)

Derivation tree

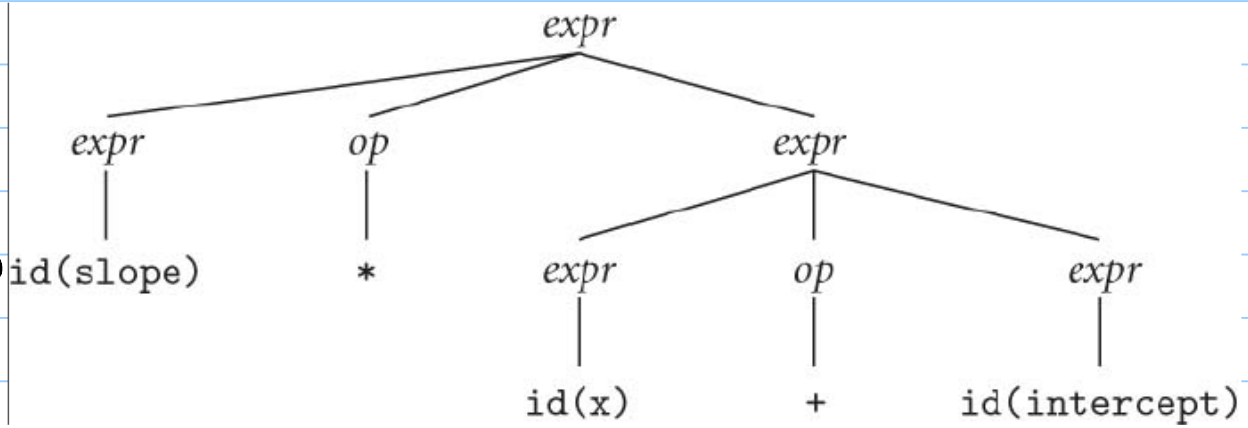


(rightmost derivation)

Ambiguous grammars



leftmost derivation →



Grammars

There are infinitely many ways to make a grammar for any context free language.

Problem in the pasing stage:
which is better?

(Try to define unambiguous grammars.)

Another example (from last time)

goal: avoid ambiguity from last ex.

Expression grammars: Simple calculator

Expr → Term | Expr Add_op Term

Term → Factor | Term Mult_op Factor

Factor → id | number | -Factor | (Expr)

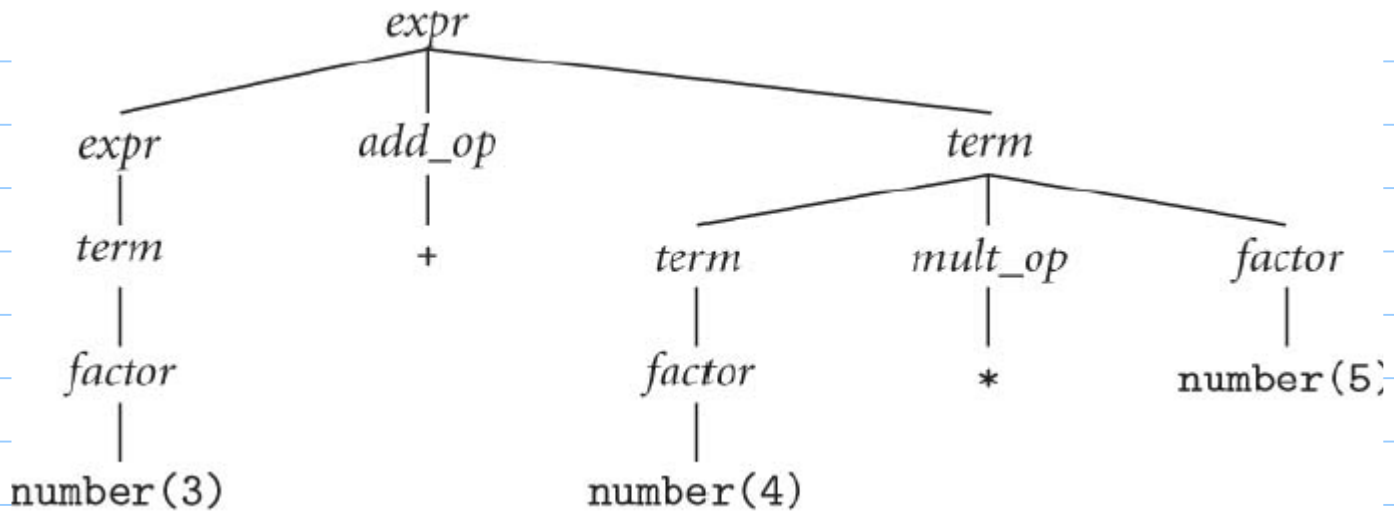
Add_op → + | -

Mult_op → * | /

Parse Tree

Ex: 3 + 4 * 5

(avoids rightmost derivation)



Scanners: do this in code

Find the syntax (not semantics) of code.

Output tokens.

A few types:

- Ad-hoc
- Finite automata ← DFA/NFA
  - nested case statements
  - table + driver
    ↖ Simulates DFA

① Ad-hoc: case based code

:= variable assignment

```
if current ∈ { "(", ")", "+", "-", "*" }
      return that symbol
if current = ":"
      read next
        if it is =, announce "assign"
              else announce error
if   current = "/"
      read next
        if it is "*" or "/"
              read until "*/" or "newline" (resp.)
        else return divide
etc.
```

Ad-hoc approach

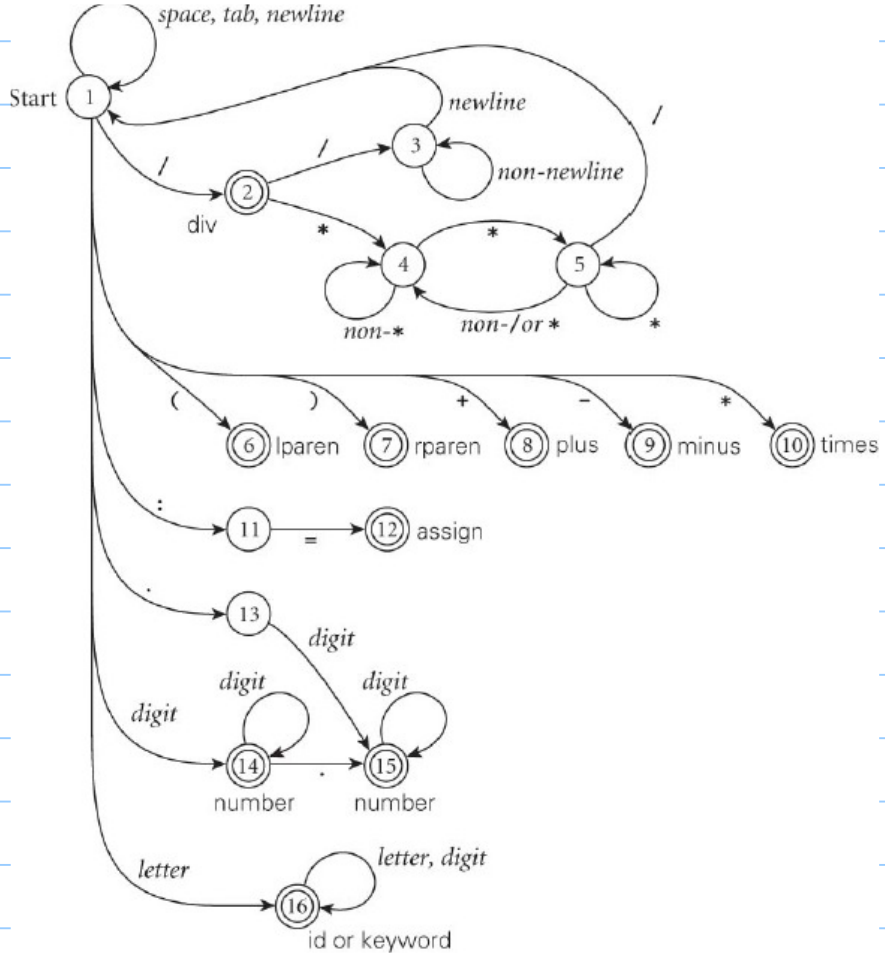
Advantage:
code is fast & compact

Disadvantage:
very ad-hoc!
- hard to debug
- no explicit representation

DFA approach

Recall our simple calculator language.

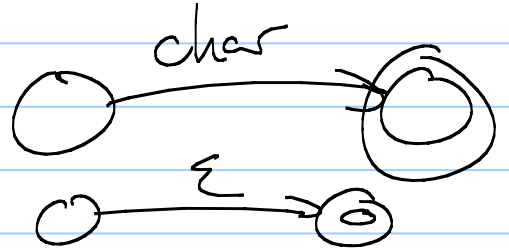
But how to get this DFA, & then how to actually model it?



Constructing a DFA (p57)

Given a regular expression, we can construct an NFA.

Simple NFA:

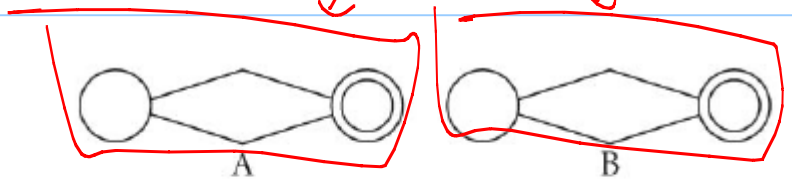


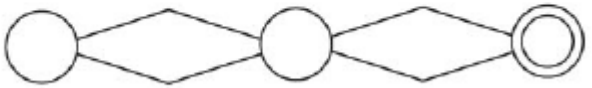
(Base case)

3 operators

Concatenation:

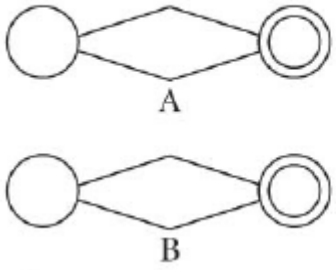
could be huge

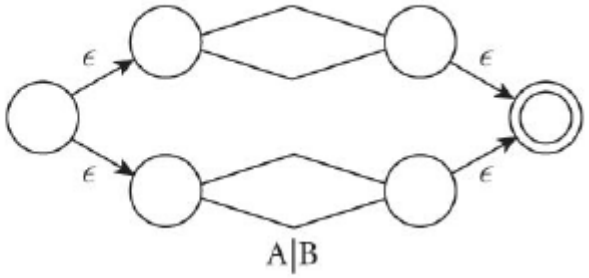




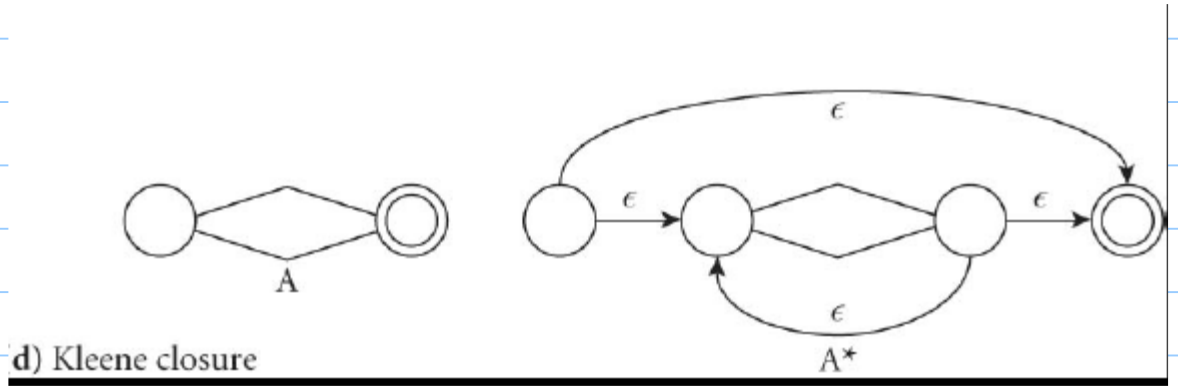
concatenation

Or:





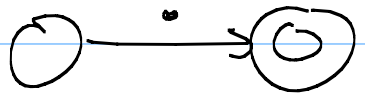
and Kleene closure (*).

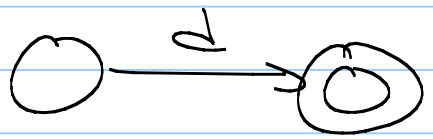


d) Kleene closure

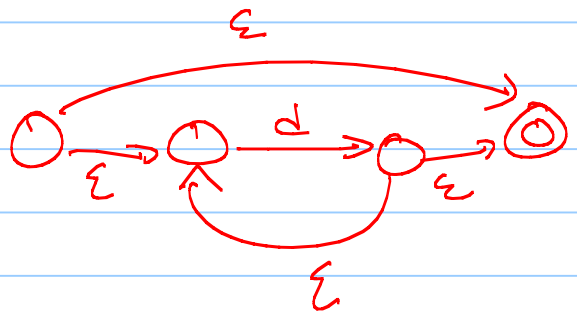
Example : decimals $d^*(.d \mid d.)\,d^*$

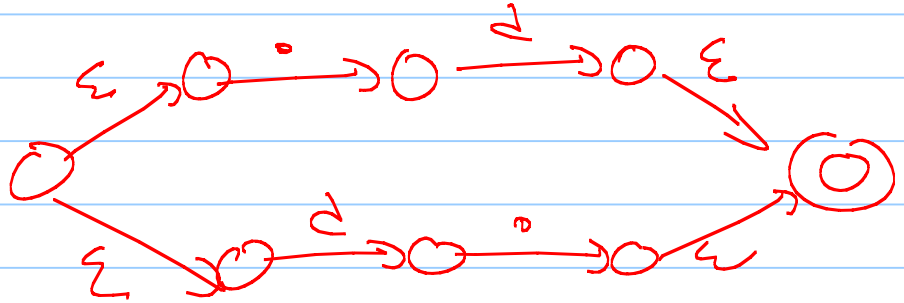
Base:



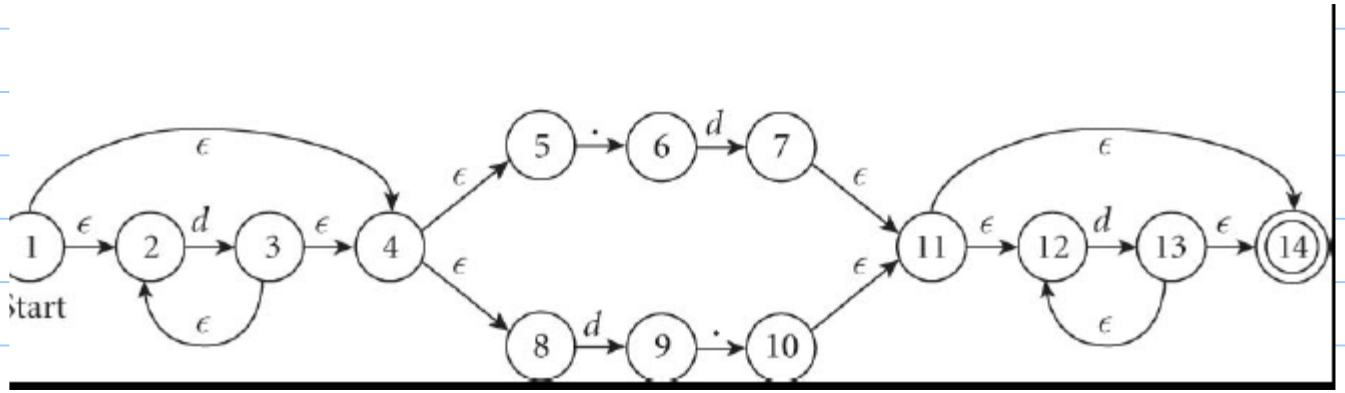


d^* :



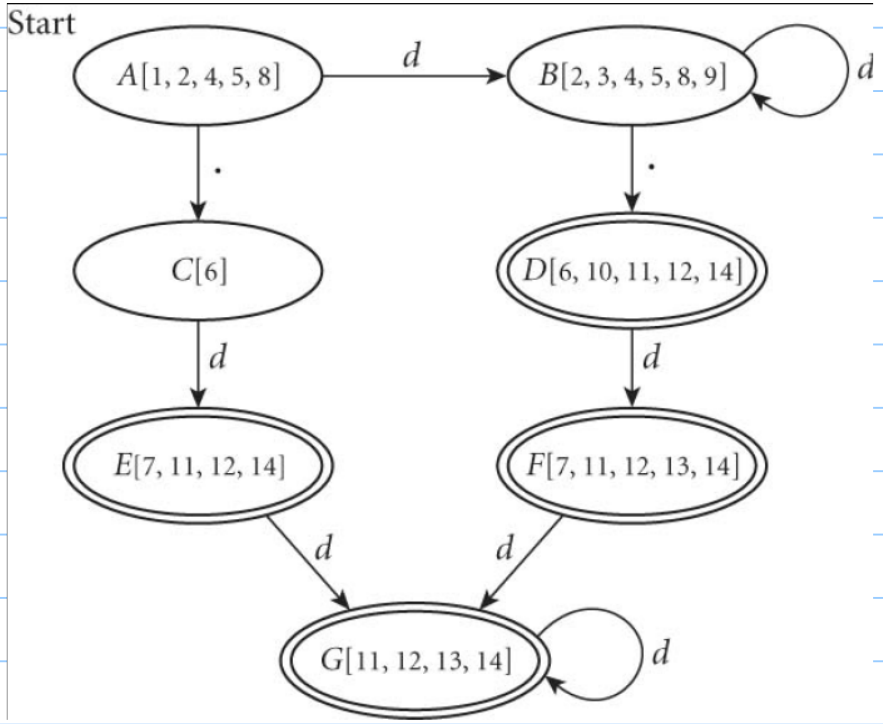


Final product :



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

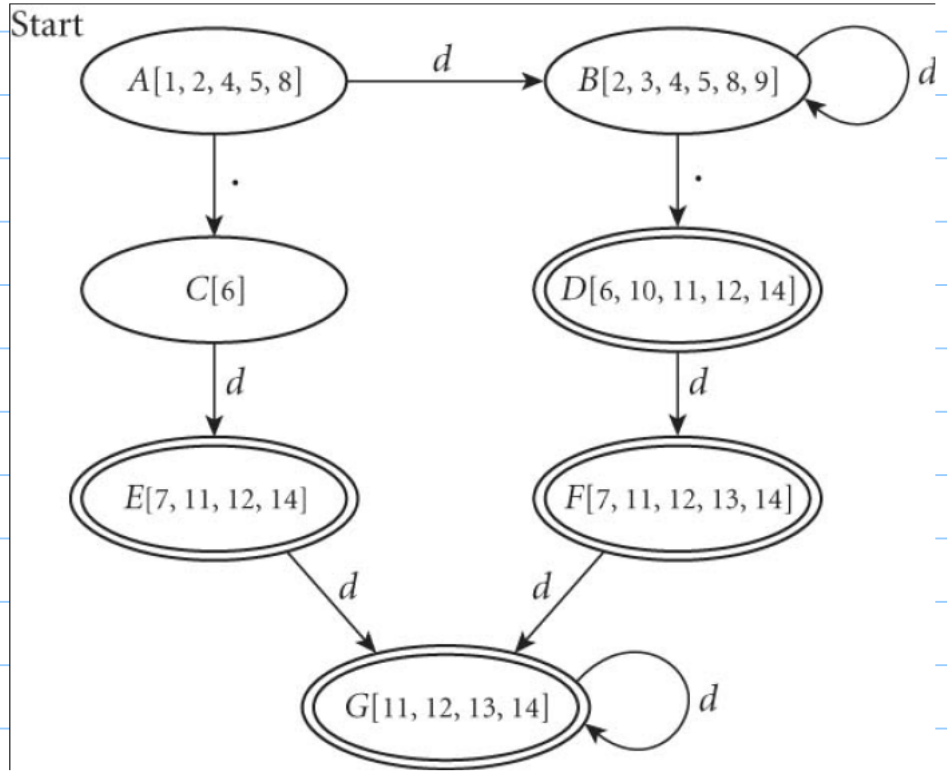
Result:
(see p. 57-58)



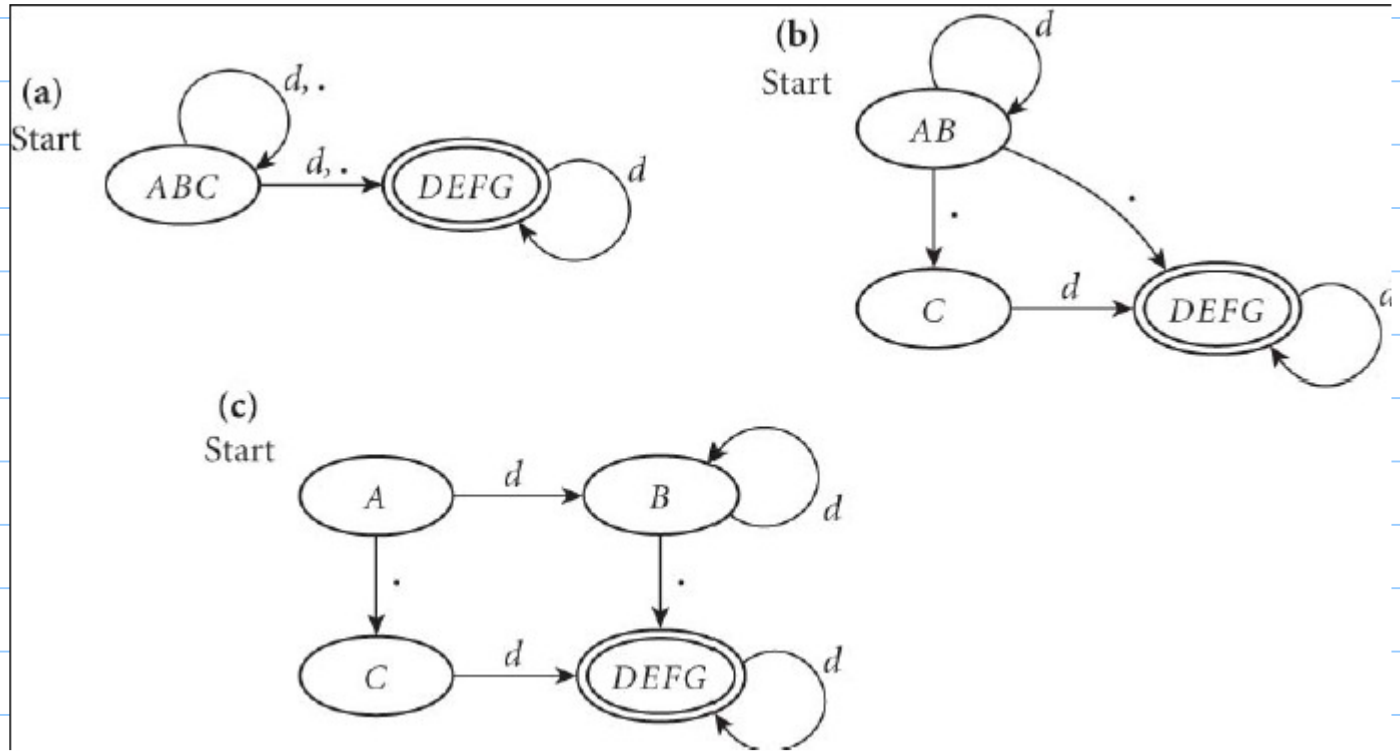
Note: This DFA is a bit redundant.

Not minimal.

Can easily find the equivalence classes. and minimize.



Process to minimize

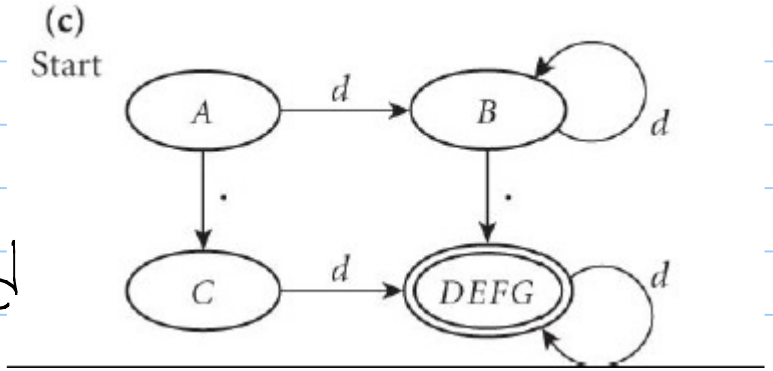


Now:

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:

Scanner Tools

In reality, this DFA is often done automatically.

Specify the rules of regular language, & the program does this for you.

Many such examples:

Lex (flex), JLex / JFlex, Quex, Ragel, ...

Next time:

Lex / flex : C-style driver

Look for HW on regular expressions, NFA / DFA, & context free languages

Next programming assignment will use flex.