CS 344 - Regular expressions

Note Title 1/23/2012

Announcements

- Ch 2 this week
- Essay due Friday

Regular Expressions

Defined as:

- 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 ocurrances)

Regular Languages

The class of languages described by a regular expression.

Ex: $\underbrace{0^*}1\underbrace{0^*} = L$

language of all strings that contain exactly one 1

$\in$ is a member of

$1 \in L$

$01 \in L$ $\qquad$ $11 \notin L$

$00010 \in L$ $\qquad$ $0 \notin L$

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 $\}$

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

Example : Numbers in Pascal

digit → [0-9]

digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

unsigned_int → digit digit*

unsigned_number → unsigned_int (ε | . unsigned_int) (ε | ((e | E) (+ | - | ε) unsigned_int))

Deterministic Finite Automata (DFA)

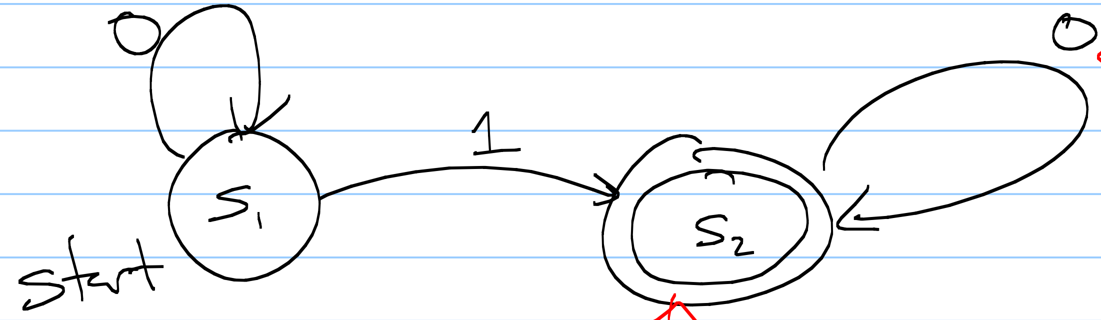
Regular languages are precisely the things recognized by DFAs.

- 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

Example:

0010

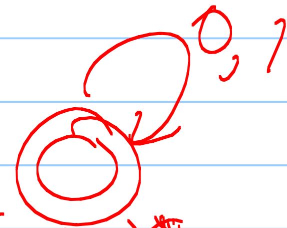
arrows give transition fcn



input

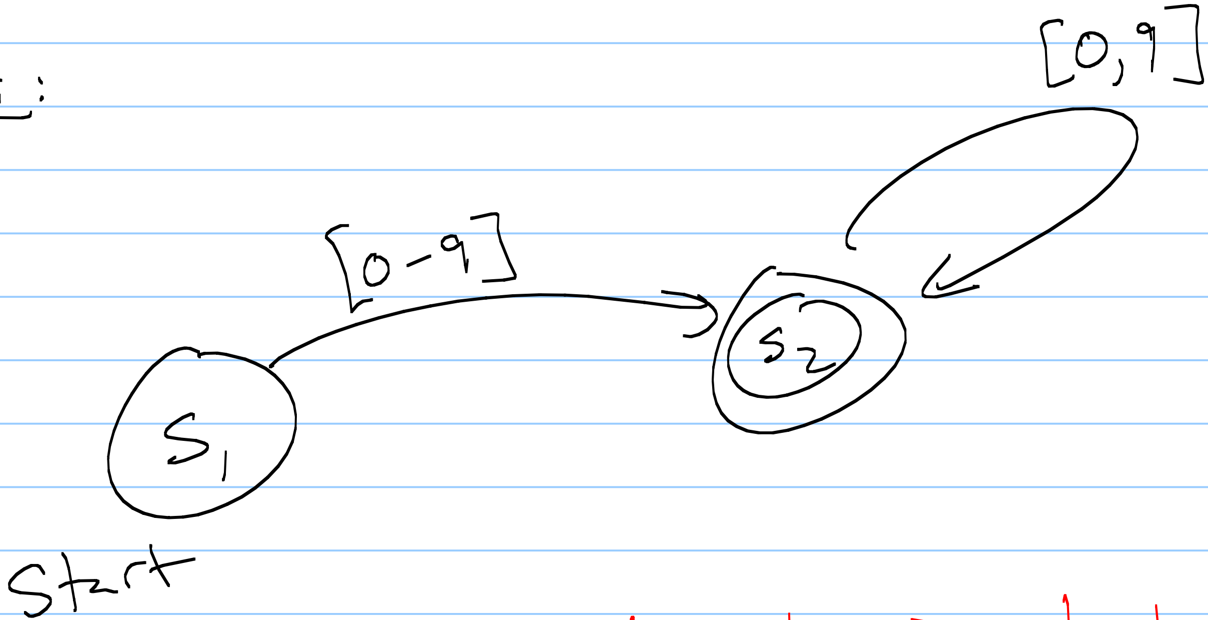
accept state (double circle)

L accepted is $0^* 1 0^*$



$L = (0|1)^*$

Ex:

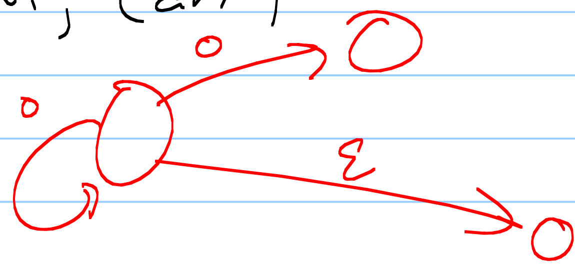


unsigned_int $\rightarrow$ digit digit*

Non-deterministic Finite Automata

Note: No ambiguity is allowed in DFA's.

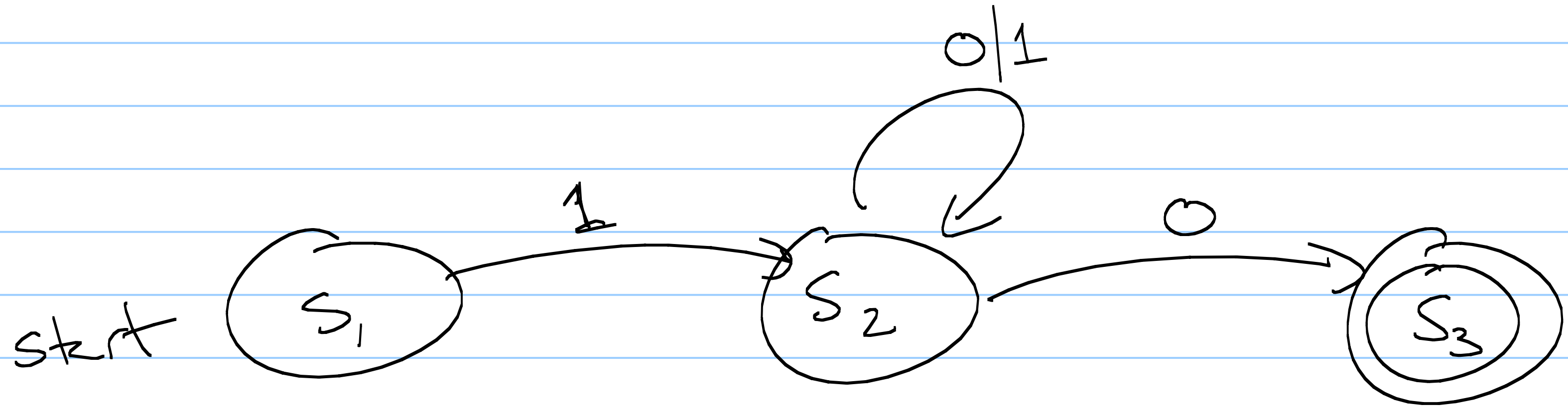
So given a state & input, can't be multiple options.



Also – no ε – transitions.

If we allow several choices to exist, this is called an NFA.

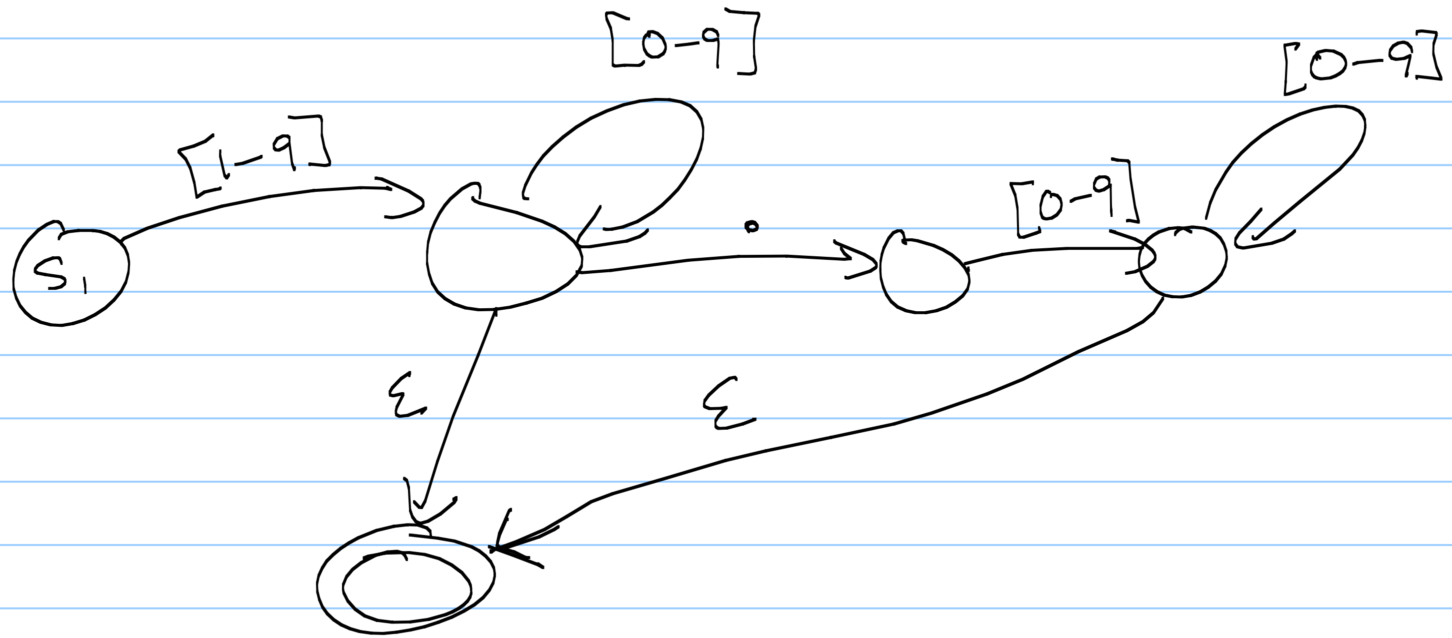
Ex:



$1(0|1)^* 0$

Ex:

unsigned_number ⟶ unsigned_int (ε | . unsigned_int)



Essentially, we can think of an NFA as modeling a parallel set of possibilities (or a tree of them).

<u>Thm:</u> Every NFA has an equivalent DFA.

Both recognize reg. languages.

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.

$0^n 1^n$

Why do we need this?

Need to "nest" expressions.

Ex: expr → id | number | - expr | (expr) | expr op expr

op → + | - | / | *

Regular expressions can't quite do this.

Context Free Languages - CFLs

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: $\{0^n 1^n \mid n > 0\}$

Nonterminals $= \{0, 1\}$

$$\left.\begin{aligned} S &\longrightarrow 0S1 \\ S &\longrightarrow \varepsilon \end{aligned}\right\} \text{productions}$$

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

$S \rightarrow 0S1$ ☆

$S \rightarrow 1S0$

$S \rightarrow \varepsilon$

Show 001011 is in this lang:

$S \Rightarrow 0S1 \Rightarrow 00S11 \Rightarrow 001S011$

use ε

☺

Expression grammars : Simple calculator

expr → term | expr add_op term

term → factor | term mult_op factor

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

variable

terminals

add_op → + | -

mult_op → * | /

x + y * z + a

Example: Show how rules can generate 3 + 4 * 5

expr ⇒ expr addop term

expr → term
term → factor
factor → num

⇒ number + term

⇒ number + (term * factor)

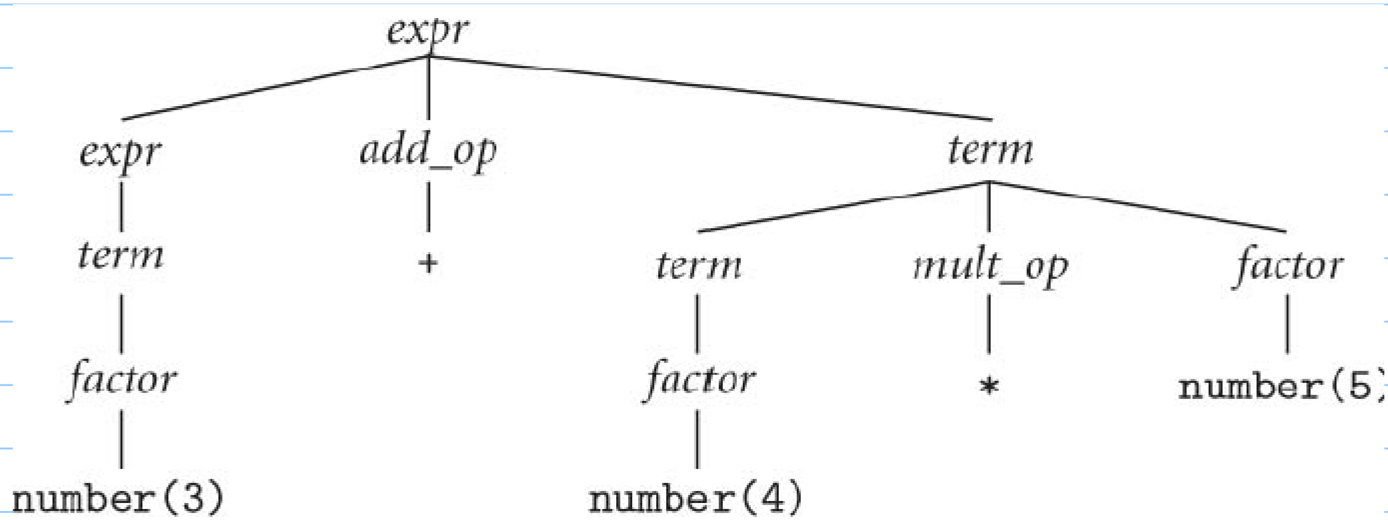
term → factor
factor → num

factor → num

num + (num * factor)

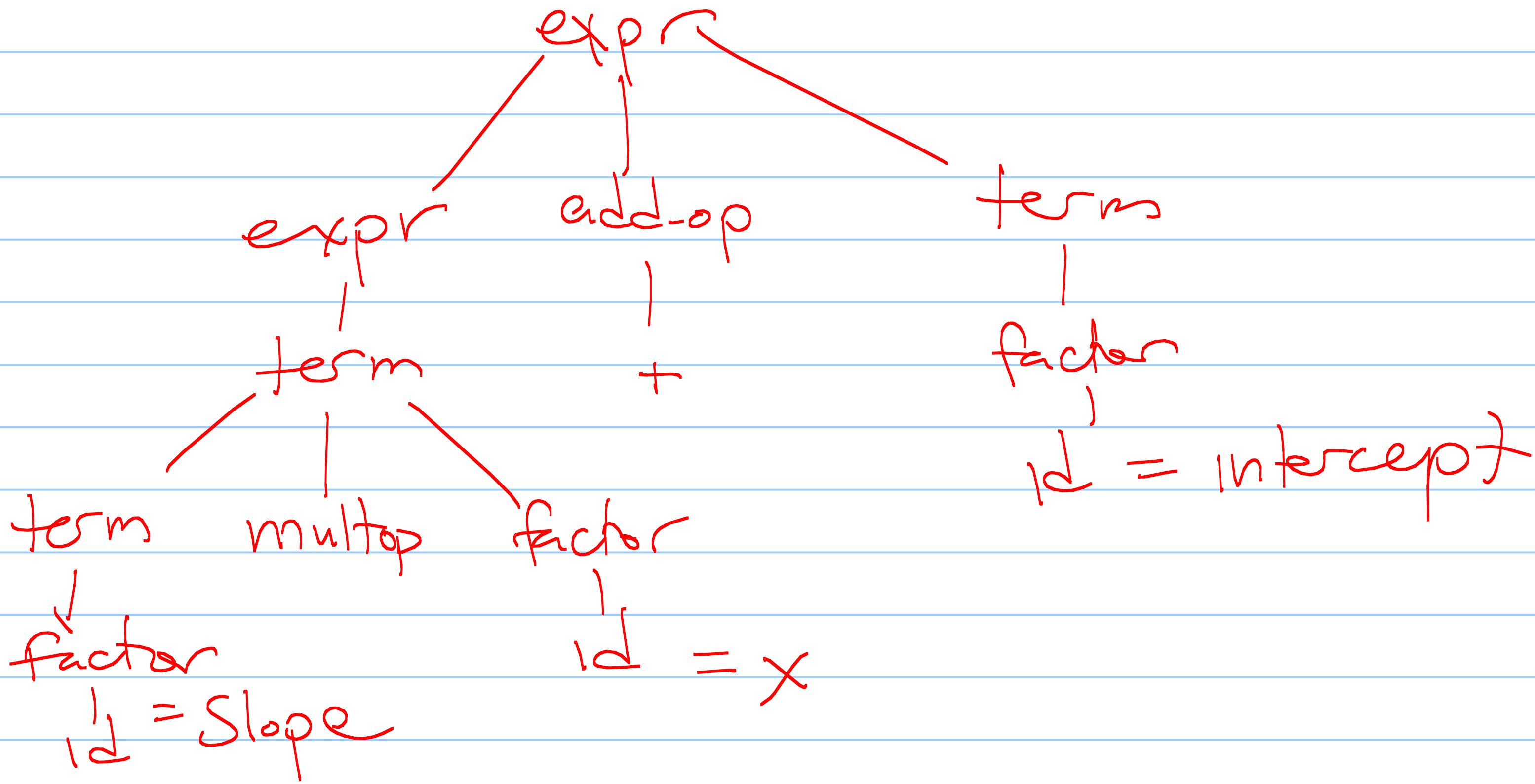
Parse Tree

Ex: 3 + 4 * 5



Another example:

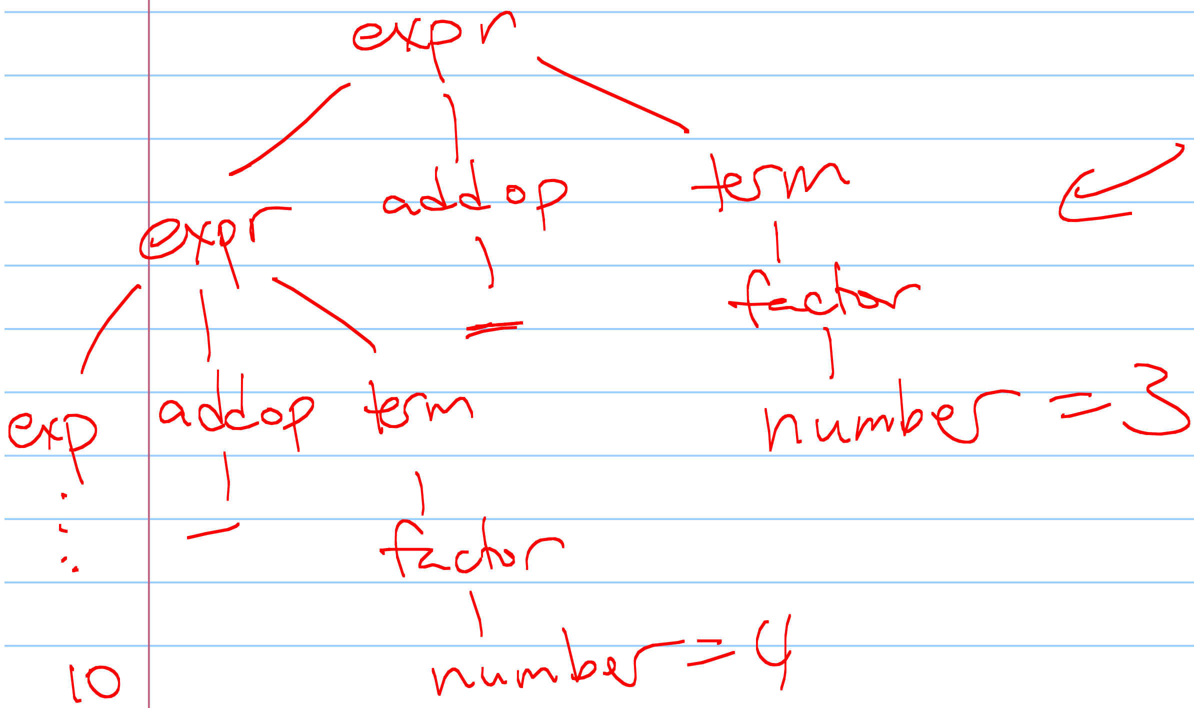
Generate slope * x + intercept



Question: Can these be ambiguous?

Ex: 10 - 4 - 3

Yes



not this one ☺

Scanning

Recall that the scanner is responsible for

- tokenizing source code
- removing comments
- save text of identifiers, #s, strings
- saving source locations for error messages

Ex: Calculator Scanner

expr → term | expr add_op term
term → factor | term mult_op factor
factor → id | number | - factor | (expr)
add_op → + | -
mult_op → * | /

Add:
id → letter (letter | digit)*
comment → /* (non-* | * non-/)* */
| // (non-newline)* newline
assign → :=

How to scan / recognize?

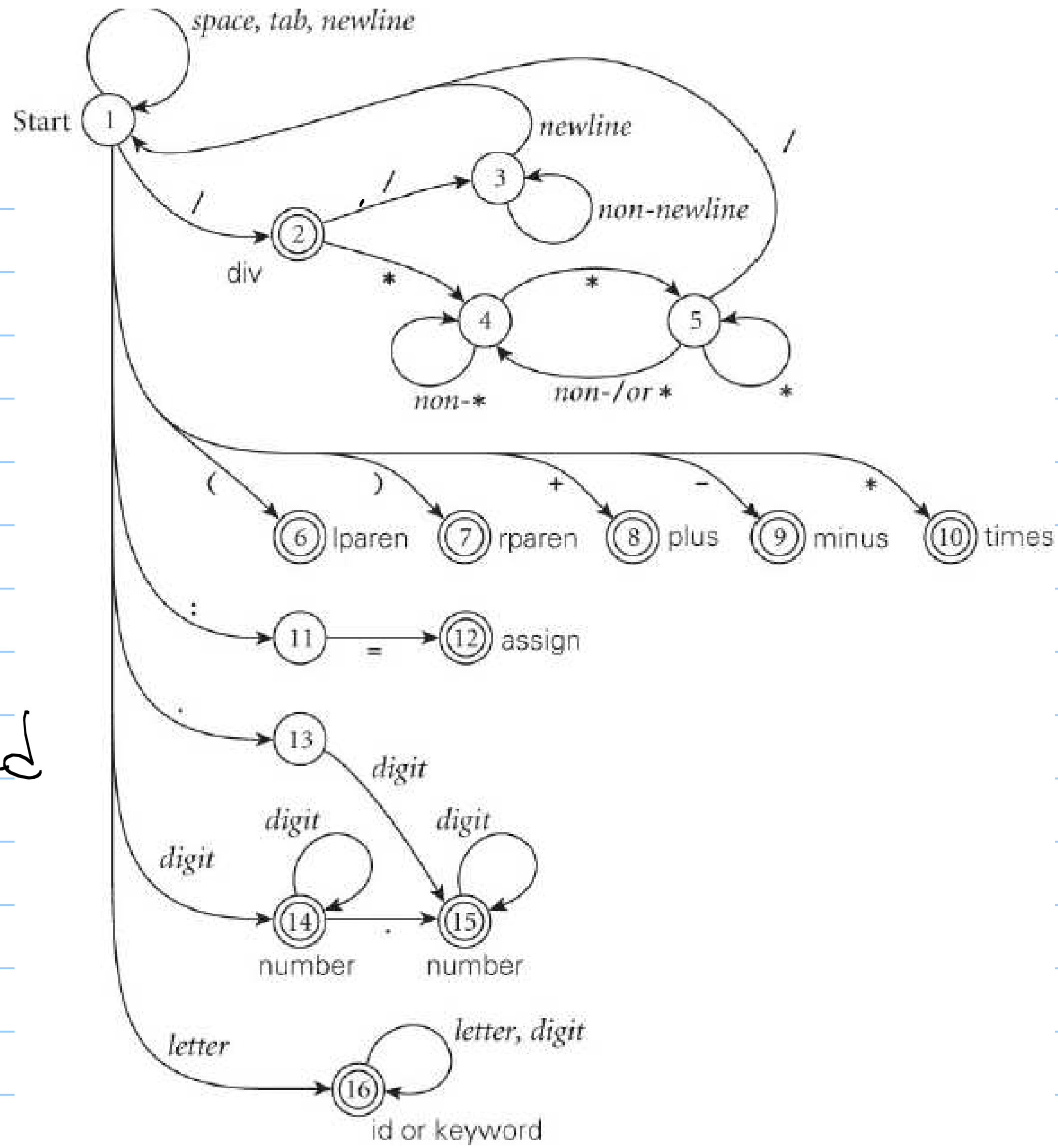
Ad Hoc Approach: Go char by char!

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

Another Way:

We are essentially running a/ DFA!

Accept states are just places we have reached a token.



Getting tokens (with DFA)

- Run the machine over & over to get our tokens

Rule: Always take longest possible token

Why? Ex: 3.14159

Ex: foobar