Parsing

Grammars
Parse trees
left vs right

Recap:

- HW due Friday
  covering CFG + scanning
- Next HW - over flex
  due next Friday
- Next topic:
  parsing + CFG
- I am gone next Mon + Tuesday
  (no office hours, but you'll have class)

Parsing:

- Given string of input tokens, a parser must determine if the tokens generate a valid program

The basis of these are context free grammars (CFGs):

- terminals: for, +, { → lowercase
- nonterminals (one a start symbol) S
  typically uppercase or underlined
- production rules
  ↳ tell transition

Notation:

start
expr → expr op expr
| (expr)
| id (variable)

op → + | - | * | /

Ex: (capital, so non-terminal)

$$E \to E\ A\ E$$
$$\to (E)$$
$$\to -E$$
$$\to id$$

(terminals)

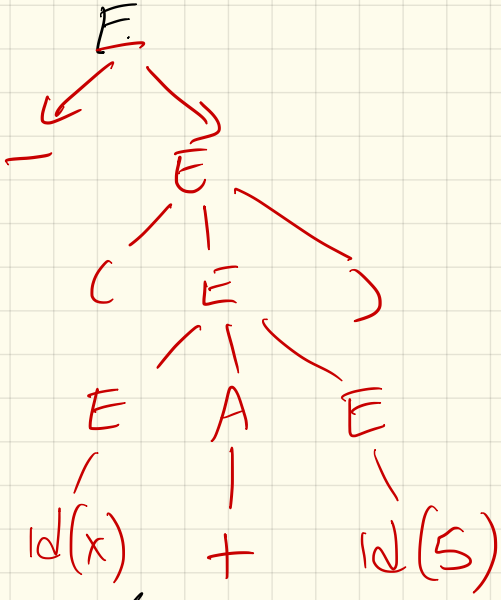
$$A \to +$$
$$\to -$$
$$\to *$$
$$\to /$$
$$\to \uparrow$$

Derivation: The process by which a grammar parses & defines a language.

Ex: Show $-(x+5)$ is accepted by the above grammar:

$$E \Rightarrow -E \Rightarrow -(E)$$
$$\Rightarrow -(E\ A\ E) \Rightarrow -(id(x)\ A\ E)$$
$$\Rightarrow -(id(x) + E)$$
$$\Rightarrow -(id(x) + id(5))$$

Parse tree: A graphical representation of this derivation:



Each parent/child shows one step of the derivation

- leaves are terminals
- root is start non-terminal

Leftmost vs rightmost

Leftmost derivation: one where the leftmost nonterminal is replaced in each step

Ex: $-(id_x + id_5)$

$E \Rightarrow -E \Rightarrow -(E)$
$\Rightarrow -(E A E) \Rightarrow -(E A id(5))$
$\Rightarrow -(E + id(5)) \Rightarrow -(id(x) + id(5))$

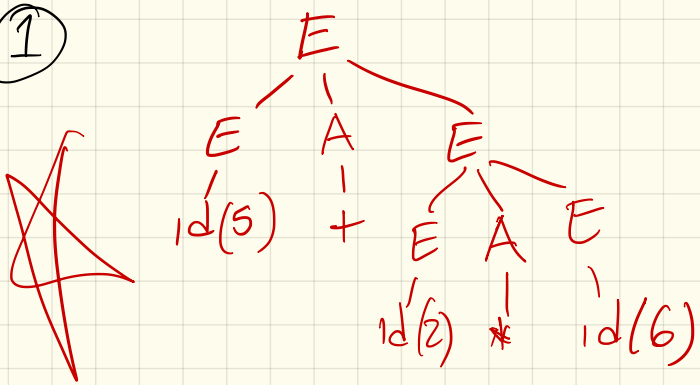
- Rightmost: always the one on the right

(Often we'll fix one way, since each tree has a unique left & right most derivation.)

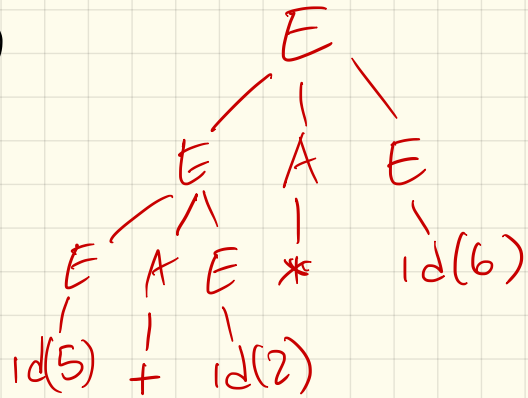
Note: Not necessarily unique!

Ex: 5 + 2 * 6

(1)



(2)



Ambiguity:

- Any grammar that produces more than one parse tree for some sentence is ambiguous.

This can be very undesirable!

We'll spend time trying to rule this possibility in our grammars.

Note: Any regular expression can also be written as a grammar

Ex: $(a|b)^* abb$

Grammar: $S \to aS \mid bS \mid A$

$A \to abb$

Book: $A_0 \to aA_0 \mid bA_0 \mid aA_1$

another

$A_1 \to bA_2$

$A_2 \to bA_3$

$A_3 \to \varepsilon$

(However, the reverse is not true!)

$0^n 1^n$ is not reg

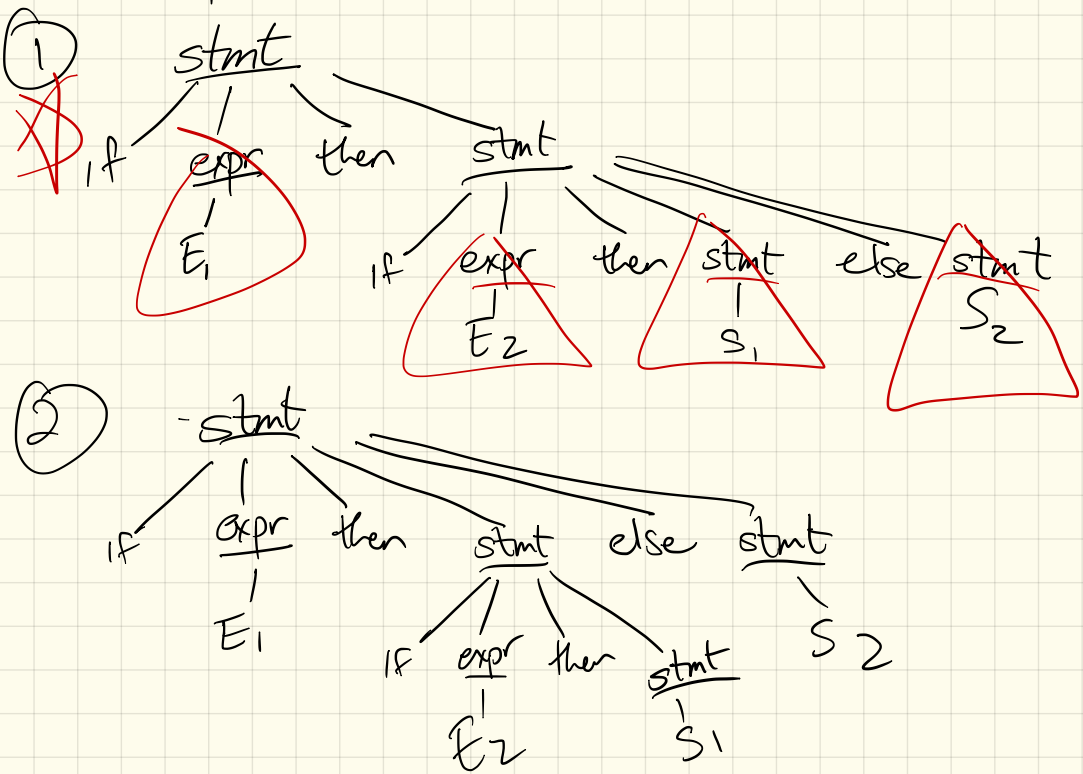
Still do regular expressions: simpler + faster

A more complex example:

stmt → if expr then stmt
| if expr then stmt else stmt
| other

Then: if E_1 then if E_2 then S_1 else S_2

2 parse trees:



in C++:

$$\underline{stmt} \rightarrow if\ (expr)\ \underline{stmt}$$
$$\mid \{\ stmt\ \}$$

$$\mid if\ (expr)\ \underline{stmt}\ else\ \underline{stmt}$$

General rule:

Match each else w/ closest unmatched then

How?

- Rewrite so any statement between an "else" + a "then" must be matched (so no if-then w/no else)

Grammar:

stmt → matched_stmt
| unmatched_stmt

matched_stmt → if expr then matched_stmt else matched_stmt
| other

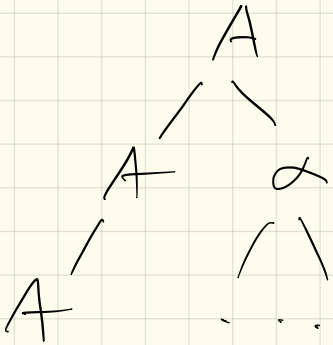
unmatched_stmt → if expr then stmt
| if expr then matched_stmt else unmatched_stmt

Next time — parsing

Dfn: A grammar is left-recursive if it has a non-terminal A with some rule

$$A \rightarrow A\alpha$$

These are bad for persers:



When scanning tokens + trying to build a tree, not sure when to stop!

Ex: $E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

Parse: $x + y * 10$

However, we do have left recursion!

To eliminate:

$$A \rightarrow A\alpha \mid \beta$$

$$A \rightarrow \beta A'$$

$$A' \rightarrow \alpha A' \mid \varepsilon$$

On

$$E \rightarrow E + T \mid T$$

$$T \rightarrow T * F \mid F$$

$$F \rightarrow (E) \mid id$$

Back to the practical:

- Any CFG can be parsed
  - ↳ Chomskey Normal Form
  - CYK algorithm
  - Run time:

This is too slow!

Most modern parsers look for certain restricted families of CFGs.

Result:

Top down parsing

Called predictive parsing.

Works well on LL(1) grammars.

Ex: $S \rightarrow cAd$

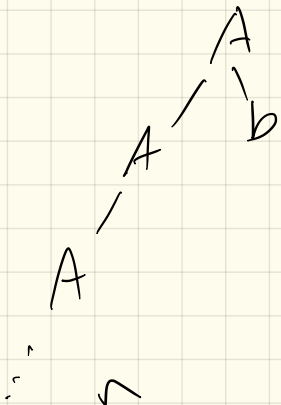
$A \rightarrow ab \mid a$

Parse cad:

Rule: String w/ S, apply rules until one matches the next input (back track if there is a mistake)

Note: Left recursion is very bad on these!

$$A \rightarrow A\,b$$



never matches an input or hits a conflict

So never forced to backtrack.

How predictive parsing works:

- the input string w is in an input buffer.
- Construct a predictive parsing table for G.
- if you can match a terminal, do it (& move to next character)
- otherwise, look in table for rule to get transition that will eventually match

Hard part (next time):
the table