Parsing

Grammars
Parse trees
left vs right

Recap:

- HW due today
- Next HW – flex scanner (due next Thursday)
- Git repo guide

Compilation

Scanning ✓

↓

Parsing

↓

⋮ (dragons)

The limits of regular expressions: ☺

Certain languages can <u>never</u> be recognized by regular expressions.

Simple example: $0^n 1^n$

$L = \{\varepsilon, 01, 0011, 000111, \ldots\}$

Only option: *

$0^* 1^* \rightarrow$ this accepts 011

More real world: math equations

$((((x+7)*2)+3)-1)$

Why? At heart,

$(^n \ldots)^n \ldots$

So, in our next level down, we need something stronger

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: tokens (via scanner)
  ids, +, (,), -
- nonterminals (one a start symbol)
  (usually capital letter)
- production rules

$$E \rightarrow (E)$$

$$\rightarrow \underline{id} + \underline{id}$$

Example: $0^n 1^n$

terminals: 0 & 1 (& ε)

L: $S \rightarrow 0S1$ ($\leftarrow$)

$S \rightarrow \varepsilon$ ($\leftarrow$)

(nonterminals: S also start)

Show $000111 \in L$

$S \xrightarrow{(1^{st})} 0\underline{S}1$

$\rightarrow 00S11$ (1^{st})

$\rightarrow 000S111$ (1^{st})

$\rightarrow 000111$ (2^{nd})

term: id, −, +, /, ↑,

nonterms: E, A *

Ex: (bigger)

$$
\begin{array}{lcll}
E & \rightarrow & E\ A\ E & 1 \\
E & \rightarrow & (E) & 2 \\
E & \rightarrow & -E & 3 \\
E & \rightarrow & \underline{id} & 4 \\
 & & & \\
A & \rightarrow & + & 5 \\
A & \rightarrow & - & 6 \\
A & \rightarrow & * & 7 \\
A & \rightarrow & / & 8 \\
A & \rightarrow & \uparrow & 9
\end{array}
$$

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

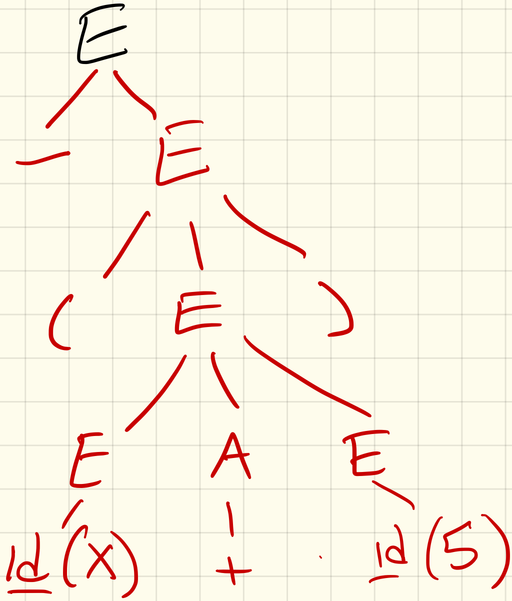
Ex: Show −(x+5) is accepted by the above grammar:

$$E \underset{(3)}{\Rightarrow} -E \underset{(2)}{\Rightarrow} -(E)$$

$$\underset{(1)}{\Rightarrow} -(E\ \underline{A}\ E) \underset{(5)}{\Rightarrow} -(E+E)$$

$$\underset{(4)}{\Rightarrow} -(\overset{x}{\underline{id}}+E) \underset{(4)}{\Rightarrow} -(\underline{id}+\overset{5}{\underline{id}})$$

Parse tree: A graphical representation of this derivation:



Each parent/child(ren) 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: $-(id+id)$

$$E \Rightarrow -E \Rightarrow -(E)$$
$$\Rightarrow -(\underline{E}AE) \Rightarrow -(i\underline{d}\ \underline{A}E)$$
$$\Rightarrow -(i\underline{d} + \underline{E}) \Rightarrow -(id+i\underline{d})$$

- 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) $E \Rightarrow E\ A\ E$

$\Rightarrow E\ A\ E\ A\ E$

$\Rightarrow id(5)\ A\ E\ A\ E$

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

$\Rightarrow id(5) + id(2) * id(6)$

(2) $E \Rightarrow E\ A\ E$

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 (CFG)

nonterm → char

Ex: $\underbrace{(a|b)^*}$ abb

Grammar:

$$S \rightarrow A\,a\,b\,b$$

$$A \rightarrow \varepsilon$$
$$\rightarrow aA$$
$$\rightarrow bA$$

(However, the reverse is not true!)

Still do regular expressions: simpler + faster

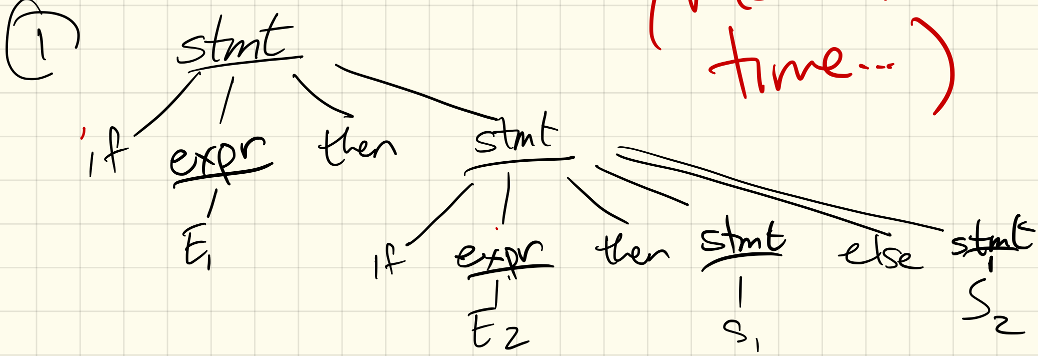
A more complex example: If statements

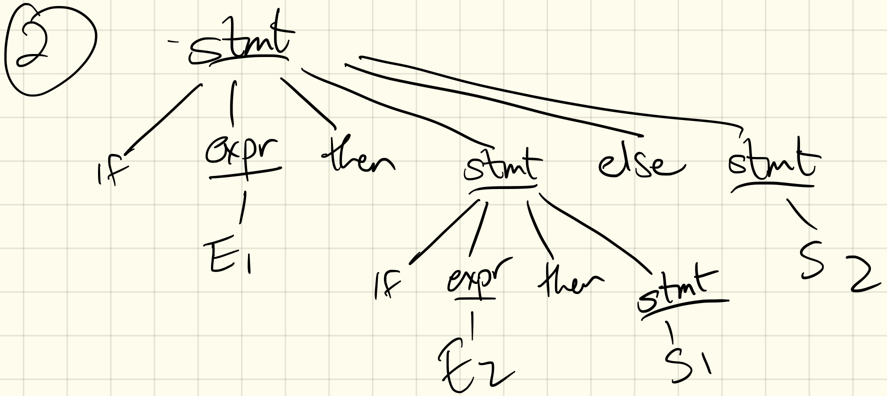
$$
\begin{aligned}
\underline{stmt} \rightarrow\ & \text{if } \underline{expr} \text{ then } \underline{stmt} \\
|\ & \text{if } \underline{expr} \text{ then } \underline{stmt} \text{ else } \underline{stmt} \\
|\ & \underline{other}
\end{aligned}
$$

Then: if E_1 then if E_2 then S_1 else S_2

2 parse trees:

(More next time...)





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

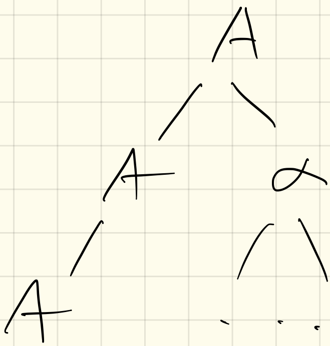
matched_stmt → if expr then matched_stmt else matched_stmt
| other

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

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

$$A \to A\alpha$$

These are bad for parsers:



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

Ex:

$$E \to E + T \mid T$$
$$T \to T * F \mid F$$
$$F \to (E) \mid id$$

Parse: $x + y * 10$

However, we do have left recursion!

To eliminate:

$$A \to A\alpha \mid \beta$$

$$A \to \beta A'$$

$$A' \to \alpha A' \mid \varepsilon$$

On

$$\boxed{\begin{aligned} E &\to E + T \mid T \\ T &\to T * F \mid F \\ F &\to (E) \mid id \end{aligned}}$$

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: $S \to cAd$

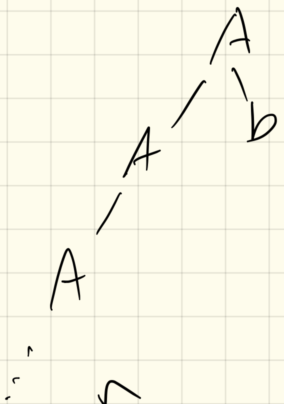
$A \to ab \mid a$

Parse cad:

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

<u>Note</u>: Left recursion is <u>very</u> bad on these!

$$A \to A\,b$$



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

<u>Hard part</u> (next time):
the table