Predictive Top-Down Parsing

- The top-down parser uses a predictive algorithm.
- It starts with the 'Exp' symbol, and the rule:
  \[ Exp \rightarrow a \; Tail \mid b \; Tail \]
- Given two/more options, the algorithm bases its choice upon the particular expression to be parsed/derived.
- In general, for efficient top-down (LL) parsing to be possible, it must be possible to predict every rule application with only a fixed amount of look-ahead.
  - A top-down parser that can look ahead \( n \) tokens into the input string to make its decisions is designated an LL(\( n \)) parser.
  - Most commercial parsers are LL(1), and use a single token at most.

Non-Predictive Bottom-Up Parsing

- We can replace this grammar with one that is more effective for bottom-up LR parsing:
  \[ Exp \rightarrow Prefix \; ; \]
  \[ Prefix \rightarrow Prefix \; + \; a \mid Prefix \; + \; b \mid a \mid b \]
- This will allow the parser to start building a tree for the expression 'a + b + a;' without first needing to read all the way to the end of the input.

Friday, 15 Feb. 2019  Programming Languages (CS 421/521)
Bottom-Up Parsing

With this new grammar, parsing the expression 'a + b + a;' can start building more complex trees from very first symbol, since 'a' itself is the right-hand side of a rule.

Then, two symbols later, we have another complete right-hand side:

A Non-LL(1) Grammar

Exp → Prefix;
Prefix → Prefix + a | Prefix + b | a | b

- While this grammar allows efficient bottom-up LR parsing, it does not allow for LL(1) parsing.
- In fact, no LL(n) parser is possible for any fixed value of n.
- Given the expression 'a + b + a;' a predictive top-down parser will need to scan all the way to the end of each sub-expression as it goes.
- Such a result would mean that parsing goes from O(n) to O(n²) time, which means large programs become infesable.
- While tools exist to generate table-based or recursive descent LL(1) parsers automatically, they require a grammar in correct form.

Bottom-Up Parsing

Two more steps of the same scanning process, reading two more symbols ('+ b') and then one final symbol (';') will complete our bottom-up tree.

Before, the parser had to read in up to n symbols in a line of length n (linear work).
Algorithm now never has to read more than 2 at once (constant work).

A Non-LL(1) Grammar

Exp → Prefix;
Prefix → Prefix + a | Prefix + b | a | b

- As discussed in the text, this grammar shows one of the key features of a non-LL(1) language: left recursion.
- In general, a language has left-recursion if there is any nonterminal symbol that can occur in a derivation chain of the form (where α here is any expression whatsoever):

\[ A \rightarrow^* \alpha \]

- In this grammar, Prefix exhibits left-recursion directly.
- In other grammars, it may not be so obvious, since there may be long and indirect derivations leading to the issue.
- While there are techniques to eliminate the feature, and other problematic grammatical rules, it is non-trivial to generate such a language, and there are some that can’t be put in proper form.
Next Week

› **Topic:** Introduction to Scala programming

› **Homework 01:**
  › **Due:** Friday, 22 February, 5:00 PM on D2L

› **Office Hours:** Wing 210
  › Monday, 9:00 AM – 10:30 AM
  › Tuesday: 3:00 PM – 4:00 PM
  › Wednesday: 9:00 AM – 10:30 AM
  › Thursday: 2:00 PM – 3:00 PM
  › Friday: 9:00 AM – 10:30 AM