Recursive descent parsing

- uses a different function for each nonterminal

```
function X ():
    Try each available rule X --> T1 T2 ... Tn until one succeeds
    (if one succeeds then return true, if none succeed return false)
      for each Ti in the current rule
          if Ti is a token and matches the next input symbol
              then advance to the next input token
          else if Ti is a non-terminal then call the function for Ti
              if it returns false then this rule for X fails
          else this rule for X fails
```

Efficiency

- As discussed with top down parsing earlier, this is most efficient if
  - the grammar is backtrack free, and
  - we can use lookahead to pick the best available rule for the current non-terminal
- LL(1) grammars fit these criteria
- **L**eft-to-right scanning with **L**eftmost derivations and **(1)** lookahead character, hence LL(1)

LL(1) rules

- LL(1) grammars cannot be ambigous, and cannot include left recursion
- A grammar is LL(1) iff for every rule of the form A --> X | Y
  - There is no token m such that both X and Y lead to a derivation beginning with m
  - At most one of X and Y can derive the empty string, and if one does (say X) then the other cannot derive any string beginning with a terminal in follow(A)
  - (definitions of first(A) and follow(A) to be discussed...)

First(A)

- For any token or nonterminal, A, First(A) is the set of all tokens that can appear as the first character in any string derived from A, including nil if it is possible to derive an empty string
- For nonterminals, First(A) is of interest because it identifies all the characters we might use in lookahead to try to decide which A derivation rule to apply

Computing First(A)

- If A is a nonterminal and A --> Y1 Y2 ... Yk then for each Yi, token t is in First(A) if t is in First(Yi) and each of the preceding Y's can derive the empty string
- i.e. if we can get Y1..Y(i-1) to derive empty strings then what can Yi start with?

Follow(A)

- For any nonterminal, A, the set Follow(A) is the set of terminals that can appear immediately to the right of A in a derivation
- e.g. If various rules can result in ...Ax..., ...Ac..., ...Af... then Follow(A) is { x, c, f }
- These are useful in that they mark possible terminators for A (could be useful in error/recovery later if there is a problem encountered in production A: we could look for a terminating x/c/f and try to pick up from there)

Computing Follow(A)

- For the grammar start nonterminal, S, add the empty string to its Follow set (anything derivable from S following the language rules can be followed by nothing)
- If W --> XAY then everything in First(Y) *except the empty string* is in Follow(A)
- If W --> XA or W --> XAY where First(Y) includes the empty string then everything in Follow(W) is in Follow(A) *[The transitive nature of this rule correctly adds the empty string to the Follow sets where appropriate.]*

Iterative predictive parsers

- track current item to be processed, start with root nonterm
- use stack to track rule productions that need to be finished
- keep going while stack not empty or input tokens to read
- repeat until done:
  - if current item is token matching the next input character then pop top of stack into current, read next input char
  - else if current item is nonterminal then use lookahead to pick correct rule to apply, push it's RHS productions onto stack in right-to-left order, then pop leftmost one into current
  - else error

Coded vs table-driven

- much as with scanning, the iterative approach to LL(1) could either
- use a table that matched each nonterminal + next token to lookup the set of tokens/nonterminals to be pushed on the stack, or
- direct code this information with a separate code block customized for each nonterminal (the latter will be examined in an example shortly)
- the pros/cons of each approach are similar to scanning (size/speed of table vs code)

Error handling

- the lookahead lets us decide what kind of structure should come next, so we can produce error messages (where needed) reflecting what kind of rule we were processing, what kind of token(s) we expected next and what kind of token we actually encountered next
- We may be able to scan ahead to search for something in the Follow set for the current rule, to see if we can recover and/or resume parsing
- next time: examine simple C code for an LL(1) parser: csci.viu.ca/~wesselsd/courses/csci435/code/parser/