Direct-coded scanners

- Meant to reduce cost associated with large lookup tables (which can cause issues with cache/paging performance)
- Increases code size to eliminate tables, and generates code that is language-specific
- Instead of a generic while loop to “read forward”, each state has its own code segment to read/process characters and decide what next state should be, then performs direct branch to that state's code segment

Our [A-Z][a-z]+ example

- Initialize token to “”, stack to <BOTTOM>, and S to s0 (just like with the table-driven), each state gets labelled code:
- s0:
  - read next char, append to token
  - push s0
  - if char is upperalpha goto s1
  - else goto Final

Example continued

- s1:
  - read next char, append to token
  - push s1
  - if char is loweralpha goto S2
  - else goto Final

Example continued

- s2:
  - read next char, append to token
  - clear stack (since s2 is an accept state)
  - push s2
  - if char is loweralpha goto s2
  - else goto final

Example continued

- Final:
- while S is not <BOTTOM> or s2 (the accept state)
  - pop top state into S
  - chop last char off token
  - roll back input stream one char
- return token type based on S

Note basic structure of segments

- state label:
  - read next char and append to token
  - if it's an accept state then clear stack
  - push state
  - for each available transition function add
    - If char is RIGHTTYPE goto NEXTSTATE
  - add default else case to to to Final

Basic structure of Final

- while S not an accept state and not <BOTTOM>
  - pop top state into S
  - chop last char off token
  - roll back input stream one char
- if S is an accept state then return matching token type
- else reject

Notes

- Sometimes character classification can be complex, and can actually be more expensive than table lookup
- Need to consider tradeoff: size/speed of table lookup vs size/speed of code, heavily dependent on the actual language being scanned