# 61A Problem Solving Strategies

### General Tips

* **three golden rules**
  + eval operator
  + eval operand
  + apply operator to operand
* **draw! things! out!**
  + visualization is very important
  + especially for things like linked lists/scheme lists, trees, etc.
  + can help with abstraction (esp for things like tree recursion)
* **always think about input + output/behavior**
  + input:
    - what is the “type” (i.e. a tree? a number?)
    - what does it represent? (i.e. n = depth of a tree you want to prune)
  + output:
    - what is the “type”
    - what does it represent/mean in the context of the problem?
    - think about the specified *behavior* of the function,   
      and how that relates to the output of the function
* for all recursive questions **abstract away** what’s happening within the recursive call;  
  focus on the **output/behavior** of the recursive call you make
* here’s a [**guide**](https://docs.google.com/document/d/1y_Hi__NVuN7UaBYAa0wwHXtvMO1DUqMl2JC0TZ9IcZk/edit?usp=sharing) i wrote on tree recursion/recursion on trees
  + note: it looks rly long, but a lot of it is somewhat colloquial language (ie. how i would say it irl during discussion) so hopefully it’s not too dense to read

### Tree Recursion

* as emphasized earlier, *drawing things out* can help you visualize the problem/solution
  + i.e. count\_steps from discussion
  + we drew a “decision tree” to illustrate what my problem is attempting to solve
  + what are my options if i took just one step? if i took two steps?
  + how can i use **abstraction/recursion** to help me figure out these options?
* remember tree recursion → **multiple recursive calls**; covers different cases/scenarios
* focus on the *semantics*/behavior of the function, and think about what it **should** give you if you passed in various different combinations/forms of parameters

### Recursion on Trees

* again, *draw and visualize* your tree
  + you can draw out a small example, or the doctest provided!
* ***generally*** you can expect to perform some recursive calls on branches
  + why? branches are subtrees
  + makes sense, because if your function takes in a tree….how do you make a recursive call on a smaller part of the tree? well the branch is a smaller part of a tree that fits the input type!
* when solving recursion on trees, don’t focus on the small details
  + generally, don’t think in minute detail about what’s happening in each branch/path/node
* similar to what i do during discussion:
  + “circle”/“section” off an entire branch as a subtree
  + if you called your function on that subtree, according to its definition, what value/output should come out of that function call?
  + if you did that for all your subtrees, what values would you get for all of them?
  + how are these values useful/relevant for the problem at hand?
  + ie. max(labels) // given max of all subtrees, how can u get max of entire tree?

### Tail Recursion ([slides](https://docs.google.com/presentation/d/1aVlyL9tTZxn9Wz-WrQQvsQ95esD6Pb72jEQsTFfFiTA/edit#slide=id.g59d74ce586_0_9))

* use helper functions
* can approach it by thinking of iterative solution first
  + variable initialization in iterative solution   
    → values you pass into helper function
  + updated/changing variables during iteration (ie. what’s changed in while loop)  
    → the parameters you need for your helper function
  + terminating case of your iteration (i.e. what makes your while loop stop)  
    → base case for your helper function
  + what happens after iteration terminates (ie. returning some value, like result)  
    → return value for the recursive helper’s base case
  + what happens *during* iteration (ie. in the actual body of the while loop)  
    → one step of iteration = one recursive call  
    → pass your newly updated/computed values into recursive call to helper
* can approach by writing a non-tail-recursive solution first
  + move the body into a helper function
  + put whatever operation that made it not tail-recursive into helper parameter
  + ie. (**\* n** (fact (- n 1)) → (fact-tail (- n 1) (**\* n** result))

### Macros (new [slides](https://docs.google.com/presentation/d/19GhUvcpZT1lVeO-D740SV5tgPYa-qgSYHKPI_yMLJL4/edit?usp=sharing))

* think backwards: what is the scheme expression/behavior that you want your macro to accomplish? (ie. execute a function twice)
* what is the scheme expression you could write to accomplish this goal?
  + scm> (begin (print ‘hi) (print ‘hi))
  + remember that expression above is just a **scheme list** with 3 elements:  
    the symbol begin, another list with 2 elements, and another list with 2 elements
* how do you construct that scheme expr
  + ie. how to do create that scheme list with the aforementioned 3 elements?
  + you could use stuff like list, cons, quote, quasiquote
  + quasiquote is usually the easiest/simplest to use, because you can pretty much just **write the expression you wanted**, and quasiquote the entire thing:  
    `(begin (print ‘hi) (print ‘hi))
* remember in a function when you’re defining this macro the (print ‘hi) will probably be a variable/parameter that will be passed in
  + ie. (twice f), where f can be something like (print ‘hi)
  + therefore `(begin ,f ,f) to make it more general
* recall unquote:
  + ,f tells Scheme that i want to evaluate what ***f*** is
  + currently what *is* f? what would f evaluate to?
  + f is bound to the **unevaluated operand** that was originally passed in
  + in this case, it’s the *scheme list* with 2 elements: (print ‘hi)
  + therefore f **is** the scheme list (print ‘hi)
  + so evaluating `(begin ,f ,f)   
    gives us (begin (print ‘hi) (print ‘hi))  
    which is exactly what we wanted!