Why functional programming/lisp?

- Assuming you've all had decent exposure to mix of procedural and OO programming (e.g. C++ or Java)
- Assuming you've not had a lot (or any?) exposure to functional programming
- A very different paradigm, think about problems in different ways, useful in different situations
- Common lisp is widely used, hybrid of functional and procedural (and can see lisp's influence on OO)
- Also a homoiconic language, so can treat it's own code as lisp data, allows lots of potential for metaprogramming

(pure) Functional programming

- Put simply, everything is a function, e.g. “if” is a function, loops are actually recursive functions, even defining a function is done by a function call
- All computation done by composing function calls
- No stored state (no variables, no explicit memory allocation/use)
- Functions have no side effects (no global variables, no access to shared memory, no reference parameters)

Programming by function composition

- Instead of providing a sequence of actions and storing intermediate values in memory, we directly pass the result of function calls as parameters to other function calls
- e.g. suppose we want to read a number from the user, compute its square root, add 10, and print it out:
- print(add(10, sqrt(read())))
- When it runs, it evaluates from inside out (i.e. read returns value to sqrt, which returns value to add, which returns value to print, which displays it)

Advantage: simplicity of expression

- Often (certainly not always) a functional solution can be expressed in a very concise form, eliminating much of the logistical clutter that comes with procedural or OO languages
- Often closely related to the ease/simplicity of expressing recursive solutions to problems compared to expressing procedural solutions

Advantage: testing simplicity

- No function side effects, so we can unit test each function in isolation – if each function works individually then (as long as our final call logic is correct) the program as a whole will work
- No variable states – never need to worry if values are set in the correct order, if variables have been initialized or updated in the right order, never need to worry about race conditions

Advantage: easy parallelization

- Suppose we have following composition:
- f (g(7), h(12), i(60))
- g, h, i can’t affect one another (no state, no side effects) so they can be run in any order
- If we have free processors available, we can simply throw one function call at each to have them run safely in parallel
- Automatic parallelization of programs in procedural languages much more challenging

Advantage: proofs of correctness

- As with testing, if we can ‘mathematically’ prove each of our functions is correct individually, and prove each composition is correct individually, then the entire program is provably correct
- This is much more challenging in procedural languages because of the complications introduced by side effects and variables/shared memory

Problem: inefficient recursion

- Recursive solutions tend to rely on stack based approach, with each recursive call generating a new stack frame, cleaned up when that call eventually returns
- Deep recursion uses lots of stack space this way, along with the overhead of a function call (setting up stack pointers, adjusting program counters, etc)
- Solution: tail recursive solutions use time and space proportional to the comparable iterative solutions, but are generally not as elegant as the “simple” recursive solutions

Problem: repetitive computation

- Suppose we compute f(x) where f is very computation intense, then we need to pass the result to two seperate functions, e.g. foo(g(f(x)), h(f(x)))
- We wind up computing f(x) twice – slow/expensive
- Solution: introduce new intermediate function, e.g. have function foo(a) actually perform foo(g(a),h(a)) then have top level function pass f(x) to foo
- Can use a similar solution if we want to read a value from user and pass it to multiple other function calls

Problem: what if we **want** a sequence

- Maybe sometimes we really really want to force things to happen in a particular sequence (e.g. prompt the user then read a value from them)
- Need to ensure the two actions are not in parallel streams of parameters, e.g. f(g(x), h(x)): we can’t know (in pure form) which of g or h will happen first
- If we want g to happen first and f to happen last, then our structure needs to reflect f(h(g(x))) and function parameters and return values need to be designed accordingly

Hybrid functional langages

- As with “OO” languages, most functional languages are actually hybrid of functional and procedural (and often OO)
- Allows developer to take advantage of functional style and functional solutions where appropriate, but also to use procedural style where that is more effective
- We’ll use common lisp, which is very much a hybrid