Pure functional programming

- Pure FP philosophy:
  - Everything is a function call returning a value,
  - No use of state and sequence, i.e.
    - no variables to store values
    - no side effects from function calls
    - no explicit ordering of steps
- Obviously common lisp impure in lots of ways...

Implications of pure fp

- No state/side effects to consider when analyzing
- Makes theoretical analysis much simpler:
  - Simpler proofs of correctness (proving parts individually is sufficient)
  - Simpler testing (again, testing parts individually is sufficient)
- Makes automated parallelization simpler, e.g. for function call `(f (g x) (h y))` we know g and h are completely independent, no order/side effects, so can simply throw them on separate processors and trust it will work

Impurities of common lisp

- Lots of places with explicit use of variables and hence stored state (e.g. defvar, let, let*)
- Lots of places with explict sequencing of steps, e.g. block, let, function bodies, etc
- Lots of places with side effects due to habit of passing pointers to items
- Clearly not pure

Impure syntax with pure implementation

- Perhaps we could use macros to let the programmer use syntax that appears impure, but have the macro rewrite it into a pure form?
- Might come with an efficiency cost, but be worth it if the purity really desired
- Will look at approaches to rewrite local variable use, sequencing of statements, and side effects

Rewriting the use of local vars

- Consider a function with two local vars

```
(defun f (x y)
   (let ((a (read)) (b (read))
        (+ (* a x) (/ b y))))
```

- Re-write it to call an extra “hidden” function that uses two extra parameters instead of the extra local vars

```
(defun f (x y) (hidden x y (read) (read)))
(defun hidden (x y a b) (+ (* a x) (/ b y)))
```

Forcing sequence of two steps

- Consider a function where we do a prompt then a read, and return the result of the read

```
(defun f ( ) (format t “Enter something”) (read))
```

- Replace by having the first action as the “condition” for an if, and the second action for both possible results

```
(defun f ( )
    (if (format t “Enter something”) (read) (read)))
```

- To do 3 steps we would rewrite the first two steps as one such function, then chain that with the third step (N steps ugly to do by hand, but possible via recursive macro)

Eliminating side effects

- Current issue with passing a list as a parameter since you’re passing a pointer not a copy of the list: if the function alters the content inside the passed list it is modifying the original
- Solution would be to actually pass lists by value: i.e. always make/pass a copy of the list, not a pointer
- Would heavily impact both memory use and run-time for recursive list handling on large lists, since we’re making a new copy every single call