Parameter passing

- Looked at lisp's conventional parameter passing mechanism: where parameters passed in the order listed, by value, and all parameters are mandatory
- Several alternatives available:
- Optional parameters with default values
- Parameter lists which accept any number of arguments
- Parameters passed by keyword instead of position

Optional parameters, &optional

- After the mandatory parameters (if any), we can include the keyword &optional followed by pairs of parameters/default values
- e.g. to make params w, x mandatory but y, z optional:

```
(defun f (w x &optional (y 10) (z “foo”))
```

- If 2 params passed, defaults are used (10 for y, “foo for z)
- If 3 params passed, third goes to y and default used for z
- If 4 params passed, third goes to y, the fourth to z

Example: read with optional prompt

- Recursive function to get/check an integer, with optional min/max for range

```
(defun getint (&optional (min most-negative-fixnum)
                         (max most-positive-fixnum))
      .... use min/max normally within func .... )
```

- (getint) ; get any int
- (getint 10) ; get int >= 10
- (getint 5 100) ; get int in range 5..100

Error checking our parameters

- Of course, still need to error check passed values

```
(defun getint (&optional (min most-negative-fixnum)
                         (max most-positive-fixnum))
; errcheck min/max, use most-neg, most-pos if bad
(let* ; establish a valid min first, valid max second
  ((realmin (if (integerp min) min most-negative-fixnum))
   (realmax (if (and (integerp max) (<= min max))
                max most-negative-fixnum))
   (userval nil))
```

Customize prompt based on range

```
(cond
   ((and (= realmin most-negative-fixnum)
         (= realmax most-positive-fixnum))
         (format t “Enter an integer: “))
   ((= realmin most-negative-fixnum)
         (format t “Enter an integer less than ~A: “ realmax))
   ((= realmax most-positive-fixnum)
         (format t “Enter an integer greater than ~A: “ realmin))
   (t (format t “Enter an integer in the range ~A..~A: “
         realmin realmax)))
```

Get/check value, recurse if needed

```
(setf userval (read))
 (cond
   ; if bad value then recurse, otherwise return it
   ((not (integerp userval)) (getint realmin realmax))
   ((< userval realmin) (getint realmin realmax))
   ((> userval realmax) (getint realmin realmax))
   (t userval))))
; note: should probably print error messages before recursing, e.g.
(block TooBig
    (format t "~A too big, try again~%" userval)
    (getint realmin realmax))
```

&optional and accumulators

- For tail recursive functions, typically use optional params for the accumlators we added, e.g. rewrite of ‘smallest’:

```
(defun smallest (L &optional (sofar most-postive-fixnum)
    (cond
        ((or (not (listp L)) (null L)) sofar)
        ((not (integerp (car L)) (smallest (cdr L) sofar))
        ((< (car L) sofar) (smallest (cdr L) (car L)))
        (t (smallest (cdr L) sofar))))
```

Variable num of params, &rest

- We can list mandatory parameters (if any), then optional parameters (if any), then specify that any number (0+) of additional parameters may follow: `&rest paramname`

```
; mandatory x, optional y, all the rest go in L
(defun f (x &optional (y 1.5) &rest L)
    ; L is actually a list of the “extras”
     ...)
(f 1)  ; y uses default, L is nil
(f 1 2) ; x=1, y=2, L is nil
(f 1 2 3) ; x=1, y=2, L is ( 3 )
```

Recursion and &rest

- Suppose we have `(defun f (x &rest L) ...)`
- Initial call might look like `(f 1 2 3 4 5)`

  so X=1, L=(2 3 4 5)
- Suppose inside function we want to chop off head of L, make recursive call with x and tail of L, desired effect is `(f 1 3 4 5)`
- Can't just say `(f x (cdr L))`, since that would really turn out like `(f 1 (3 4 5))`
- Sneak peak at higher order functions, pass f as a parameter:

Apply and lists of parameters

- Higher order functions are just functions that accept other functions as parameters, we'll revisit them lots later
- `(apply 'f L)` runs function f with parameters from list L

  `(apply '+ '(1 2 3))` acts like `(f 1 2 3)`
- So when L was `(2 3 4 5)` our call

  ```
  (apply 'f (cons x (cdr L)))
  ```
- Acts just like `(f x 3 4 5)`, which is what we wanted

smallest using &rest

- smallest again, accept any num of params, returning most-positive-int by default (skip any elements that aren't ints)

```
(defun smallest (&rest L)
(let
   ; locals for value of front element, and smallest of
rest
   ; (both default to most-positive-fixnum)
   ((smInRest (if (> (length L) 1)
       (apply 'smallest (cdr L)) most-positive-fixnum)
    (front (if (and (not (null L)) (integerp (car L)))
       (car L) most-positive-fixnum)))
```

Smallest using &rest

- Now simply compare the front to the rest, return smaller

```
(if (< front smInRest) front smInrest)))
```

- Note that this time we did all the heavy lifting when computing the local vars
- Unfortunately, not a tail recursive solution

Keyword parameters, not positional

- So far, all our parameter passing has been positional: the first value passed goes to the first function parameter, the second value passed goes to the second, etc
- This is the way most languages handle parameter passing
- Lisp provides an alternative: keyword passing, using &key in the function call we actually specify which value we want assigned to which parameter
- Means we need to know what the function parameter names are instead of which order they're in

&key example

- Toy example function

```
(defun display (&key e L)
    (format “~A, ~A~%” e L))
```

- Call passing e first, L second

```
(display  :e 3  :L ‘(10 20 30))
```

- Call passing L first, e second

```
(display  :L ‘(10 20 30)  :e 3)
```

- We’ll see the : notation again with structures