The forms of a function

- A function could be a compiled function, an actual lambda expression, or a function built by defun, label, or a call to lambda
- Some built-in constructs (let, cond, etc) provided through macros will also need to be addressed
- `(functionp f)` returns t iff f is a function
- `(fboundp f)` returns t iff f was created as a defvar
- `(function f)` or, equivalently, `#’f` , returns the implementation of a function

The forms returned by (function f)

- We can test if f is a compiled function using

  `(eq (type-of (function f)) ‘COMPILED_FUNCTION)`

- Otherwise, functions will have one of four forms:
  - `If defined by defun:`

    `(‘lambda-block name paramlist ...body...)`
  - `If an actual lambda expression: \`

    `(‘lambda paramlist ...body...)`
  - `If defined by lambda:`

    `(‘lambda-closure env1 env2 env3 paramlist ...body...)`
  - `If defined by labels:`

    `(‘lambda-block-closure env1 env2 env3 name paramlist body)`

Complication: function operation

- function is an operator, not a typical function
- For defun and labels, (function f) only works if f is the actual name of the function (not a variable holding the name) and f is actually in scope at the point of call
- Suppose we want to create a function called getparams, and have it look up the parameter list for a function, e.g. (getparams foo)
- getparams can’t simply call function on its parameter (won’t work for defun or labels)

Solution: getparams macro

- We could force the user to make calls like

  `(getparams (function foo))` but that’s clunky, error prone

- use a getparams macro: it calls function and passes result to a secondary function to extract the actual parameter list

  `(extractparams (function func))))`

- `extractparams` works with the actual func implementation, user still makes their call as `(getparams foo)`

extractparams

Source: `(getparams f)`, macro call: `(extractparams (function f))`

```
(defun extractparams (f)
(if (functionp f) ; makes sure param was ok
   (let ((ftype nil)) ; will store function type as a string
   ; lambda, lambda-block, lambda-closure, lambda-block-closure
   ;    or, for compiled functions, set to compiled
   (if (listp f) (setf ftype (symbol-name (car f)))
       (if (eq (type-of f) ‘COMPILED_FUNCTION) “COMPILED”))
```

Now figure out param list

- The ftype allows us to figure out where to find the param list (‘error for bad function form or ‘compiled for compiled func)

```
(cond
   ((string= ftype “LAMBDA”) (nth f 1))
   ((string= ftype “LAMBDA-BLOCK”) (nth f 2))
   ((string= ftype “LAMBDA-BLOCK-CLOSURE”) (nth f 5))
   ((string= ftype “LAMBDA-CLOSURE”) (nth f 4))
   ((string= ftype “COMPILED”) ‘compiled)
   (t ‘error))))
```

Handling built-ins

- This still doesn't help if user tries to call on something like let or cond, anything built-in or provided by a macro
- Perhaps one could build up a hash-table of such values to be accepted, e.g.
    - for `‘let` it could return the parameter list as

      `(varpairs &rest statements)`
    - For `‘defun` it could return the parameter list as

      `(fname paramlist &rest statements)`
    - etc

Arity of functions

- “arity” of a function is the number of parameters it expects (a little complicated in lisp, where there can be required parameters, optional parameters, and &rest)
- we want a lookupArity, to find the arity of a function
- use a lookupArity macro with an extract Arity function
- Programmer writes `(lookupArity f)`, macro transforms that to a call like `(extractArity (function f))`
- `extractArity` uses `extractparams` to get f’s param list

extractArity

- Suppose we now have f’s param list
- It may/may not contain some mandatory parameters, some optional, and the `&rest`
- Perhaps we have `extractArity` return 3 values: the number of mandatory parameters, the number of optional parameters, and either `t/nil` to indicate `&rest` supported/not
- Remember `(multiple-value-bind x y z)` allows us to return multiple values, and `(nth-value 1)` or `(nth-value 2)` allows the caller to capture y or z

Counting expected params

- Our param list can look something like

```
(a1 a2 a3 &optional (x1 v1) (x2 v2) &rest r)
```

- Perhaps we divide into three smaller lists and count elements in each:
  - List of items after &rest (if any)
  - List of items after &optional once &rest content removed
  - List of items after the &optional/&rest content removed
- Then we simply count/return the sizes of the three lists

The three sublists

- Use `(position e L)` and `(subseq S i j)` to get the sublists

```
(let ((mand params) (opt params) (rest nil))

; check for &rest first
;     if found chop it out of param list
(if (member ‘&rest params) (block ‘foundRest
    (setf rest t)
    (setf opt (subseq params 0 (position ‘&rest params))))
```

Sublists continued

```
   ; check for &optional next
(if (member ‘&optional opt) (block ‘foundOptional
    ; count the elements after &optional
    (setf opt (subseq opt (position ‘&optional opt)
                          (length opt)))
    (setf opt (- (length opt) 1))
    ; remove everything from &optional to end of list
    (setf mand
            (subseq params 0 (position ‘&optional params)))))
; count the mandatory args (whatever’s left)
(setf mand (length mand))
```

Parsing lisp in lisp - revisited

- Look at how a lisp program can interactively examine its own content/structure
- 1st, creating a set of macros/functions to look up the parameter list expected by a function (regardless of whether the function was a lambda, a label, a defun, etc)
- 2nd, creating macros/functions to count the number of mandatory, optional, &rest arguments a function supports
- We need to understand the structure of each different form of function and how lisp can find/access it