MASSACHUSETTS INSTITUTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Fall Semester, 1996

Lecture Notes — 10 December

Computability

Q: Can we eliminate the need for `define` and rely solely on `lambda`?

A:

We used `define` to name `loop` to generate a procedure that never terminates. An infinite loop without `define` is

```
    ((lambda (h) (h h))
     (lambda (h) (h h)))
  -->
      ((lambda (h) (h h))
       (lambda (h) (h h)))
```

which evaluates to itself.

What about recursive procedures, such as

```
(define fact
  (lambda (n)
    (if (= n 0)
        1
        (* n (fact (- n 1))))))
```

Here, we use `define` to name the procedure so that we can call it recursively. We can eliminate the `define` by *lambda-abstracting* the procedure name:

```
(lambda (fact)
  (lambda (n)
    (if (= n 0)
        1
        (* n (fact (- n 1))))))
```

Call this F. But what do we apply F to to get factorial? For example, we could apply F to `1+` and call this on `6`; since `6` is not `0`, this yields `(* 6 (1+ (- 6 1)))`, or `(* 6 6)`.

We really want to apply F to factorial, i.e. to F itself. If we apply F to F and call this on `6`, we get `(* 6 (F (- 6 1)))`; but the recursive call to `(F 5)` fails because `fact` isn't bound to F in this call. So what we need is a method of applying F to itself "just enough."

Suppose we had an operator, Y, with the property that for any `f`, `((Y f) n) = ((f (Y f)) n)`.

```
((Y F) 3) = ((F (Y F)) 3)
          = (((lambda (fact)                    ;  substitute for first  F
                (lambda (n)
                  (if (= n 0)
                      1
                      (* n (fact (- n 1))))))
              (Y F))
             3)                                 ;  apply (lambda (fact) ... )
          = ((lambda (n)                        ;  to (Y F)
               (if (= n 0)
                   1
                   (* n ((Y F) (- n 1)))))
             3)                                 ;  apply (lambda (n) ... )
          = (if (= 3 0)                         ;  to 3
                1
                (* 3 ((Y F) (- 3 1))))           ;  evaluate (if ... )
          = (* 3 ((Y F) 2))
          = (* 3 (* 2 ((Y F) 1)))
          = (* 3 (* 2 (* 1 ((Y F) 0))))
          = (* 3 (* 2 (* 1 1)))
          = 3!
```

Now how do we construct Y using only `lambda`?

We can get our infinite loop to do some work by adding an `f`:

```
    ((lambda (h) (f (h h)))
     (lambda (h) (f (h h))))
  -->
      (f ((lambda (h) (f (h h)))
          (lambda (h) (f (h h)))))
```

Unfortunately, if we actually type this into SCHEME, it will go into an infinite loop (adding an infinite number of applications of `f`). We can prevent this by adding a dummy `lambda` and application, delaying evaluation until we need it:

```
  Y = (lambda (f)
          ((lambda (h) (lambda (x) ((f (h h)) x)))
           (lambda (h) (lambda (x) ((f (h h)) x))))
```

We'll abbreviate `(lambda (h) (lambda (x) ((f (h h)) x)))` by D; `(Y f) = (D D)`.

```
Now   ((Y f) n) = ((D D) n)                       ;  expand (Y f)
               = (((lambda (h)                    ;  expand first  D
                     (lambda (x)
                       ((f (h h) x))))
                   D)
                  n)                              ;  apply (lambda (h) ... )
               = ((lambda (x)                     ;  to  D
                    ((f (D D) x)))
                  n)                              ;  apply (lambda (x) ... )
               = ((f (D D)) n)                    ;  to n
               = ((f (Y f)) n)                    ;  (Y f) = (D D)
```

In fact, we can test this in SCHEME by evaluating

```
==> (((lambda (f)
        ((lambda (h) (lambda (x) ((f (h h)) x)))
         (lambda (h) (lambda (x) ((f (h h)) x)))))
      (lambda (fact)
        (lambda (n)
          (if (= n 0)
              1
              (* n (fact (- n 1)))))))
      10)
 3628800
```

Q: Are there well-defined things that cannot be computed?

A:

Assume that we have a procedure `(safe? p a)` that returns true if evaluating `(p a)` produces an answer and returns false otherwise. An example of a non-terminating procedure (with no arguments) is:

```
(define (loop) (loop))
```

Now what happens when we do

```
(define (diag? x)
  (if (safe? x  x)
      (not (apply x (list x)))
      nil))

(diag? diag?)
```