Iterative Refinement

5 November 2020
Well, it's Election Day, again,
Assignment 1
Assignment 2
Exam 1
Assignment 3
Assignment 4
Exam 2
Assignment 5
Exam 3
Let’s think on happier things.
In *The Mythical Man-Month* (1975), Turing Award recipient Fred Brooks writes:
“The programmer, like the poet, works only slightly removed from pure thought-stuff. He builds castles in the air, from air, creating by exertion of the imagination…
“Few media of creation are so flexible, so easy to polish and rework, so readily capable of realizing grand conceptual structures. **Yet the program construct, unlike the poet’s words, is real in the sense that it moves and works**, producing visible outputs separate from the construct itself…”
“One types the correct incantation on a keyboard, and a display screen *comes to life*, showing things that never were nor could be… It prints results, draws pictures, produces sounds, moves arms. The magic of myth and legend has come true in our time…
“The computer resembles the magic of legend in this respect, too. If one character, one pause, of the incantation is not strictly in proper form, the magic doesn’t work. Human beings are not accustomed to being perfect, and few areas of human activity demand it. Adjusting to the requirement for perfection is, I think, the most difficult part of learning to program.”
Example: Files and directories
A Pittsburgh road is ...

4 8 15 16 23 42
A file system is a tree structure where each directory can have an arbitrary number of children.

This is a lot like the “more realistic rumor mill” we saw last week! These are both kinds of \textit{n-ary trees}. 
Iterative refinement
Iterative refinement
Iterative refinement

Start simple.
Iterative refinement

Start simple.

When the problem needs more:

- Expand your data definition
- Adjust your functions
Iterative refinement

;; find : Directory, String → Boolean
Iterative refinement

;; find : Directory, String → Boolean

A Directory has:

  a name
  files and other directories

A File has:

  a name
Initial data definitions

(define-struct dir [name content])
;; A Directory is
;; (make-dir String LOFD)

;; A LOFD is either:
;; - '()
;; - (cons FileOrDirectory LOFD)

;; A FileOrDirectory is either:
;; - File
;; - Directory

;; A File is a String
Initial data definitions

(define-struct dir [name content])

;; A Directory is
;;   (make-dir String LOFD)

;; A LOFD is either:
;;   - '()
;;   - (cons FileOrDirectory LOFD)

;; A FileOrDirectory is either:
;;   - File
;;   - Directory

;; A File is a String
Initial implementation

;; find : Directory, String -> Boolean
(define (find dir n)
  (... (dir-name dir) ...
       (find-for-lofd (dir-content dir) n) ...))

;; find-for-lofd : LOFD, String -> Boolean
(define (find-for-lofd lofd n)
  (cond [(empty? lofd) ...
         [(cons? lofd)
          (... (find-for-fod (first lofd) n) ...
               (find-for-lofd (rest lofd) n) ...)])

;; find-for-fod : FileOrDirectory, String -> Boolean
(define (find-for-fod fod n)
  (cond [(string? fod) (... (find-for-file fod n) ...)]
        [(dir? fod) (... (find fod n) ...)])

;; find-for-file : File, String -> Boolean
(define (find-for-file file n)
  (... file ...))
New problem

Compute how much space is used on the disk:

;;; du : Directory -> Number
Revised data definitions

```
(define-struct dir [name content])
;;  A Directory is
;;  (make-dir String LOFD)

;;  A LOFD is either:
;;  - '()
;;  - (cons FileOrDirectory LOFD)

;;  A FileOrDirectory is either:
;;  - File
;;  - Directory

(define-struct file [name size])
;;  A File is
;;  (make-file String Number)
```
Revised implementation

;; find : Directory, String -> Boolean
(define (find dir n)
 (... (dir-name dir) ...
    (find-for-lofd (dir-content dir) n) ...)))

;; find-for-lofd : LOFD, String -> Boolean
(define (find-for-lofd lofd n)
 (cond [(empty? lofd) ...]
       [(cons? lofd)
        (... (find-for-fod (first lofd) n) ...
            (find-for-lofd (rest lofd) n) ...)])

;; find-for-fod : FileOrDirectory, String -> Boolean
(define (find-for-fod fod n)
 (cond [(string? fod) (... (find-for-file fod n) ...)]
       [(dir? fod) (... (find fod n) ...)]))

;; find-for-file : File, String -> Boolean
(define (find-for-file file n)
 (... (file-name file) ...
    (file-size file) ...))
Iterative refinement: *Encapsulation*
Iterative refinement

Start simple.

When the problem needs more:

- Expand your data definition
- Adjust your functions

When you have a working solution, consider how it could be simplified.
When we’ve had mutually recursive functions, we’ve defined them all at the top-level.

Given these data definitions:

```scheme
;; A Directory is
;;  (make-dir String LOFD)
(define-struct dir [name content])

;; A File is a String

;; A FileOrDirectory is either:
;;  - File
;;  - Directory

;; A LOFD is either:
;;  - '()
;;  - (cons FileOrDirectory LOFD)
```
When we’ve had mutually recursive functions, we’ve defined them all at the top-level. We wrote:

```hs
;; find : Directory, String -> Boolean
;; find-for-file : File, String -> Boolean
;; find-for-file-or-directory : FileOrDirectory, String -> Boolean
;; find-for-lofd : LOFD, String -> Boolean
```

But the only one another programmer or a user wants to see is `find`; the other functions are just for making `find` work.
We can encapsulate those functions inside `find`:

```scheme
;; find : Directory, String -> Boolean
;; Return #true if a File or a Directory with name `n`
;; exists somewhere under Directory `d`
(define (find d name)
  (local [((define (find-for-file ... ) ...)
    (define (find-for-file-or-directory ... ) ...)
    (define (find-for-lofd ... ) ...)]
    (or (string=? (dir-name d) name)
        (find-for-lofd (dir-content d) name)))))
```

The functions can still call each other, but they’re not exposed at the top-level.

Because of this, `find` is the only one of the functions we need to publish a signature for, and the only one of them we need to write tests for.
A good candidate for encapsulation is a function that has one or more helpers closely linked to it, where the outside program really only wants to call the main function, not the helpers.
Try doing the same for du.
Since we end up encapsulating our functions, maybe we should have encapsulation in our template!

If we do this, we can do less renaming. It’s fine for every one of our functions that operates on a directory to have local functions that have the names `fun-for-file, fun-for-dir`, etc.
;; Directory -> ...
;; Template for a function that consumes a Directory
(define (fun-for-dir d)
  (local [(define (fun-for-file f)
               (... (file-name f) ...
                   (file-size f)))
           (define (fun-for-file-or-directory fod)
             (cond [(file? fod)
                    (... (fun-for-file fod))
                   [(dir? fod)
                    (... (fun-for-dir fod))]))
           (define (fun-for-lofd lofd)
             (cond [(empty? lofd) ...]
                   [(cons? lofd)
                    (... (fun-for-file-or-directory
                          (first lofd)) ...
                         (fun-for-lofd (rest lofd)))]))
     (... (dir-name d) ...
         (fun-for-lofd (dir-content d)) ...))


Directory, String -> Boolean
Return true if a File or a Directory with name `n`
exists somewhere under Directory `d`

(define (find d name)
  (local [(define (fun-for-file f)
           (... (file-name f) ...
                (file-size f)))
         (define (fun-for-file-or-directory fod)
           (cond [(file? fod)
                   (... (fun-for-file fod))]
                 [(dir? fod)
                  (... (find fod))])
         (define (fun-for-lofd lofd)
           (cond [(empty? lofd) ...]
                 [(cons? lofd)
                  (... (fun-for-file-or-directory (first lofd)) ...
                       (fun-for-lofd (rest lofd))))]
                 (... (dir-name d) ...
                      (fun-for-lofd (dir-content d)) ...)))
Iterative refinement: Efficiency
“The real problem is that programmers have spent far too much time worrying about efficiency in the wrong places and at the wrong times; premature optimization is the root of all evil (or at least most of it) in programming.”

Donald Knuth, “Computer Programming as an Art”, Turing Award Lecture, 1974
Nonetheless, there are some big problems of efficiency that we want to be sure we take care of.

We can use local to deal with redundant computation that can lead to this kind of poor performance.
Directory depth: 0
Directory depth: 1
Directory depth: 2
Directory depth: 3
;; make-skinny : Natural -> Directory
;; Produce a skinny directory tree n+1 levels deep,
;; where the leaf is a file named "Y" of size 1.
(define (make-skinny n)
  (cond [(zero? n)
         (make-file "Y" 1)]
        [else
         (make-dir "X" (list (make-skinny (sub1 n))))]))

(check-expect (make-skinny 0) (make-file "Y" 1))
(check-expect (make-skinny 2) (make-dir "X" (list (make-dir "X" (list (make-file "Y" 0))))))
(make-skinny 10) will produce 10 directories named "X" and then a file "Y".

(make-skinny 100) will produce 100 directories named "X" and then a file "Y".
If we want to search a directory tree to find the size of a file with a given name, we could use our template with encapsulated helper functions to quickly write it.
;; find-size : Directory, String -> [Maybe Integer]
;; Search the given directory tree for a file with the given name.
;; Produce its size if found; #false otherwise.
(define (find-size d name)
  (local [(define (fun-for-file f)
            (if (string=? (file-name f) name)
                (file-size f)
                #false))
         (define (fun-for-file-or-directory fod)
           (cond [[(file? fod) (fun-for-file fod)]
                   [(dir? fod) (find-size fod name)]))
         (define (fun-for-lofd lofd)
           (cond [[(empty? lofd) #false]
                  [(cons? lofd)
                   (if (not (false? (fun-for-file-or-directory (first lofd))))
                       (fun-for-file-or-directory (first lofd))
                       (fun-for-lofd (rest lofd)))])))]
  (fun-for-lofd (dir-content d))))
Let’s look at how long it takes to run our `find-size` function on directory trees of different depths.

To do this, we can use a new Racket primitive named `time`, which return the time it takes to evaluate its operands:

```
> (time (+ 1 2))
cpu time: 1 real time: 0 gc time: 0
3
```
> (time (find-size (make-skinny 10) "Y"))
cpu time: 1 real time: 2 gc time: 0
1
> (time (find-size (make-skinny 11) "Y"))
cpu time: 3 real time: 3 gc time: 0
1
> (time (find-size (make-skinny 12) "Y"))
cpu time: 5 real time: 6 gc time: 0
1
> (time (find-size (make-skinny 13) "Y"))
cpu time: 10 real time: 10 gc time: 0
1
> (time (find-size (make-skinny 14) "Y"))
cpu time: 20 real time: 21 gc time: 0
1
> (time (find-size (make-skinny 15) "Y"))
cpu time: 47 real time: 47 gc time: 0
1
> (time (find-size (make-skinny 16) "Y"))
cpu time: 95 real time: 94 gc time: 0
1
> (time (find-size (make-skinny 17) "Y"))
cpu time: 185 real time: 184 gc time: 0
1
;;; find-size : Directory, String -> [Maybe Integer]
;;; Search the given directory tree for a file with the given name.
;;; Produce its size if found; #false otherwise.
(define (find-size d name)
  (local [(define (fun-for-file f)
    (if (string=? (file-name f) name)
        (file-size f)
        #false))
    (define (fun-for-file-or-directory fod)
      (cond [[(file? fod) (fun-for-file fod)]
              [(dir? fod) (find-size fod name)])])
    (define (fun-for-lofd lofd)
      (cond [[(empty? lofd) #false]
              [(cons? lofd)
                (if (not (false? (fun-for-file-or-directory (first lofd))))
                    (fun-for-file-or-directory (first lofd))
                    (fun-for-lofd (rest lofd)))])))
  (fun-for-lofd (dir-content d))))
search tree with 10 Xs
search tree with 10 Xs

search tree with 9 Xs

search tree with 9 Xs

2
search tree with 10 Xs

search tree with 9 Xs

search tree with 8 Xs

search tree with 9 Xs

search tree with 8 Xs

search tree with 8 Xs

search tree with 8 Xs

search tree with 8 Xs

search tree with 8 Xs

2

4
search tree with 10 $X$s

- search tree with 9 $X$s
  - search tree with 8 $X$s
    - search tree with 7 $X$s
  - search tree with 8 $X$s
    - search tree with 7 $X$s
- search tree with 9 $X$s
  - search tree with 8 $X$s
    - search tree with 7 $X$s
  - search tree with 8 $X$s
    - search tree with 7 $X$s
search tree with 10 Xs

search tree with 9 Xs
downward

search tree with 9 Xs
downward

search tree with 8 Xs
downward

search tree with 8 Xs
downward

search tree with 7 Xs
downward

search tree with 7 Xs
downward

search tree with 7 Xs
downward

search tree with 7 Xs
downward

search tree with 7 Xs
downward

search tree with 7 Xs
downward

search tree with 7 Xs
downward

search tree with 7 Xs
downward

search tree with 7 Xs
downward

search tree with 6 Xs
downward

search tree with 6 Xs
downward

search tree with 6 Xs
downward

search tree with 6 Xs
downward

search tree with 6 Xs
downward

search tree with 6 Xs
downward

search tree with 6 Xs
downward

search tree with 6 Xs
downward

search tree with 6 Xs
downward

and so on
;; find-size : Directory, String -> [Maybe Integer]
;; Search the given directory tree for a file with the given name.
;; Produce its size if found; #false otherwise.
(define (find-size d name)
  (local [(define (fun-for-file f)
            (if (string=? (file-name f) name)
                (file-size f)
                #false))]
    (define (fun-for-file-or-directory fod)
      (cond [[(file? fod) (fun-for-file fod)]
              [[(dir? fod) (find-size fod name)]])
    (define (fun-for-lofd lofd)
      (cond [[(empty? lofd) #false]
              [(cons? lofd)
               (if (not (false? (fun-for-file-or-directory (first lofd))))
                   (fun-for-file-or-directory (first lofd))
                   (fun-for-lofd (rest lofd)))]])
    (fun-for-lofd (dir-content d))))
;; find-size : Directory, String -> [Maybe Integer]
;; Search the given directory tree for a file with the given name.
;; Produce its size if found; #false otherwise.
(define (find-size d name)
  (local [(define (fun-for-file f)
            (if (string=? (file-name f) name)
                (file-size f)
                #false))
          (define (fun-for-file-or-directory fod)
            (cond [((file? fod) (fun-for-file fod))
                    [((dir? fod) (find-size fod name))])])
          (define (fun-for-lofd lofd)
            (cond [((empty? lofd) #false]
                   [((cons? lofd)
                      (local [(define try (fun-for-file-or-directory
c                                        (first lofd))])
                        (if (not (false? try))
                          try
                          (fun-for-lofd (rest lofd)))]))]
              (fun-for-lofd (dir-content d)))))

In this case, we got a big speed up because the repeated computation was in a recursive function.

If we had a repeated computation in a non-recursive function, the improvement could be much smaller, and using `local` might just make the program harder to read.
Acknowledgments

This lecture incorporates material from:

Matthias Felleisen
Robert Bruce Findler
Matthew Flatt
Gregor Kiczales
Shriram Krishnamurthi
Marc Smith