CSC324
Functional Programming
Efficiency Issues, Parameter Lists

Afsaneh Fazly

January 28, 2013

---

1 Thanks to A. Taflovich, P. Ragde, S. McIlraith, E. Joanis, S. Stevenson, G. Penn, D. Horton

---

Example:

;; Pre: 11 and 12 are lists
;; Return -1 if both lists are empty,
;; otherwise return length of the longer list
(define (longest-nonzero 11 12)
  (cond ((and (null? 11) (null? 12)) -1)
        ((> (length 11) (length 12)) (length 11))
        (else (length 12)))))

Problem?

• evaluating the same expressions more than once.

Any ideas how to solve this?

---

Example:

;; Pre: 11 and 12 are lists
;; Return -1 if both lists are empty,
;; otherwise return length of the longer list
(define (longest-nonzero 11 12)
  (cond ((and (null? 11) (null? 12)) -1)
        ((> (length 11) (length 12)) (length 11))
        (else (length 12)))))

Problem?

• evaluating the same expressions more than once.

Any ideas how to solve this?

---

efficiency issues: helper procedures

Solution 1: bind values to parameters using a helper procedure.

(define (maximum x y)
  (if (> x y) x y))

(define (longest-nonzero 11 12)
  (cond ((and (null? 11) (null? 12)) -1)
        ((> (length 11) (length 12)) (length 11))
        (else (maximum (length 11) (length 12)))))

Notes:

• there is a built-in max function.
• helper procedures are an important and useful tool.
efficiency issues: binding constructs

Solution 2: use a binding construct such as let to create local bindings:

\[
\text{(let ([id1 expr1] ... [idN exprN]) body)}
\]

• creates local bindings of identifiers to expression values.
• scope of these bindings is the body of let.
• evaluation: expr1, ..., exprN are evaluated in some undefined order, saved, and then bound to id1, ..., idN.
• evaluations have the appearance of being done in parallel.

let: example

Consider:

(define sq-cube (lambda (x) (let ([sqr (* x x)] [cube (* x (* x x))) (list sqr cube)))))

Result of evaluating (sq-cube 3)?

let: example

Consider:

(define sq-cube (lambda (x) (let ([sqr (* x x)] [cube (* x (* x x))) (list sqr cube)))))

Result of evaluating (sq-cube 3)?

• (9 27)

Want to reuse sqr.
let: example

Reusing sqr:

(define sq-cube
  (lambda (x)
    (let ([sqr (* x x)]
           [cube (* x sqr)]
           (list sqr cube)))))

Result of evaluating (sq-cube 3)?
• Error!

Why doesn't this work?

• sqr is undefined at the time of evaluating (* x sqr).

Any ideas how to solve this?

let: example

Solution 1: use nested let

(define sq-cube
  (lambda (x)
    (let ([sqr (* x x)]
           [cube (* x sqr)]
           (list sqr cube)))))

Why does this work?
**let: example**

Solution 1: use nested let

```lisp
(define sq-cube
  (lambda (x)
    (let ([sqr (* x x)])
      (let ([cube (* x sqr)])
        (list sqr cube)
      ))))
```

Why does this work?

- **scope** of each binding is the **body** of let

  - scope of `sqr ← (* x x)` is
    ```lisp
    (let ([cube (* x sqr)])
      (list sqr cube)
    )
    ```

**let***

Solution 2: use let*

```lisp
(let* ([id1 expr1]
        ...[
          idN exprN]
  body)
```

- **scope** of each binding is the part of the let*-expression to the right of the binding.
- **evaluation**: `expr1, ..., exprN` are evaluated sequentially, from left to right.

**let***

sq-cube with let*:

```lisp
(define sq-cube
  (lambda (x)
    (let* ([sqr (* x x)]
           [cube (* x sqr)])
      (list sqr cube)))
```

**let versus letrec**

```lisp
(let ([sumup (lambda (x)
                (cond ([zero? x] 0)
                       (+ x (sumup (- x 1)))]
           [s (sumup 10)])
      (list s))
```

What is the result?
let versus letrec

(letrec ([sumup (lambda (x)
    (cond ([zero? x] 0)
          (else (+ x (sumup (- x 1))))))]
    [s (sumup 10)])
  (list s))

letrec

(letrec ([id1 expr1]
  ...
  [idN exprN])
  body)

- **scope**: each binding has the entire letrec expression as its region (including the bindings and body).
- **evaluation**: expr1, ..., exprN are evaluated saved, and then bound to id1, ..., idN.
- **commonly used** for defining local recursive helper procedures.

letrec example: mutually recursive procedures

(letrec ([my-even? (lambda (x)
    (if (= x 0)
      #t
      (my-odd? (- x 1)))]
  [my-odd? (lambda (x)
    (if (= x 0)
      #f
      (my-even? (- x 1)))]
  (if (and (my-even? 4) (not (my-odd? 4)) (not (my-even? 5)))
    42
    0))

Result?

example: simple Fibonacci

;; simple (inefficient) Fibonacci
(define fib (lambda (n)
    (cond ((= n 0) 0)
          ((= n 1) 1)
          (else (+ (fib (- n 1)) (fib (- n 2)))))))
faster Fibonacci: using accumulators and helper procedure

;; faster Fibonacci: using accumulators to improve efficiency
(define fastfib
  (lambda (p1 p2 i n)
    (if (= i n)
      p1
      (fastfib p2 (+ p1 p2) (+ i 1) n))))

(define fib
  (lambda (n)
    (fastfib 0 1 0 n)))

faster Fibonacci: local helper procedure using letrec

;; faster Fibonacci: using letrec
(define (fib n)
  (letrec ([fastfib (lambda (p1 p2 i n)
                           (if (= i n)
                             p1
                             (fastfib p2 (+ p1 p2) (+ i 1) n)))]
           (fastfib 0 1 0 n)))

faster Fibonacci: let vs. letrec

;; faster Fibonacci: using let?
(define (fib n)
  (let ([fastfib (lambda (p1 p2 i n)
                        (if (= i n)
                          p1
                          (fastfib p2 (+ p1 p2) (+ i 1) n)))]
        (fastfib 0 1 0 n)))

Does this work?

• No. Scope of fastfib is the body of let.

faster Fibonacci: let vs. letrec

;; faster Fibonacci: using let?
(define (fib n)
  (let ([fastfib (lambda (p1 p2 i n)
                         (if (= i n)
                           p1
                           (fastfib p2 (+ p1 p2) (+ i 1) n)))]
        (fastfib 0 1 0 n)))

Does this work?
versions of let: examples

(let ([x 2]) (* x x))
⇒  
(let ([x 4] [y (+ x 2)]) (* x y))
⇒  
(let ([x 4]) (let ([y (+ x 2)]) (* x y)))
⇒  
(let* ([x 4] [y (+ x 2)]) (* x y))
⇒  

bindings in let and let*

(let ((id1 e1)...(idN eN)) body-expr)
⇌
((lambda (id1...idN) body-expr) e1...eN)

AND

(let* ((id1 e1) (id2 e2)) body-expr)
⇌
((lambda (id1) ((lambda (id2) body-expr) e2)) e1)

• The two constructs let and lambda are equivalent in what they evaluate to.
• Binding of values to identifiers is by parameter passing OR lambda reduction ⇒ no assignment

let and lambda reduction: examples

(let ([a 5]
      [b 10])
  (+ a b))
⇒ a bound to 5, b bound to 10
⇒ evaluate (+ a b) by substitution
⇒ evaluate (+ 5 b)
⇒ evaluate (+ 5 10)
⇒ 15

((lambda (a b) (+ a b)) 5 10)
⇒ a bound to 5, b bound to 10
⇒ evaluate (+ a b) by substitution
⇒ evaluate (+ 5 b)
⇒ evaluate (+ 5 10)
⇒ 15
Recall lambda-expression:

\[
\text{lambda (formal-params) <body>)}
\]

where \(\text{(formal-params)}\) specifies a list of \(N\) parameters.

What if we do not know the exact number of actual arguments?

- use parameter lists, as in:

\[
\text{lambda formal-params <body>)}
\]

where \(\text{formal-params}\) binds a formal parameter to a (possibly empty) list of actual parameters.

Compare proc1 and proc2:

\[
\text{define proc1}
\]

\[
(\text{lambda} (x)
\text{<body>))
\]

\[
(\text{proc1}) \Rightarrow \text{Error! requires exactly one argument}
\]

\[
(\text{proc1} 'a) \Rightarrow x \text{ is bound to 'a}
\]

\[
(\text{proc1} 'a 'b 'c) \Rightarrow \text{Error! requires exactly one argument}
\]

\[
\text{define proc2}
\]

\[
(\text{lambda} x
\text{<body>))
\]

\[
(\text{proc2}) \Rightarrow x \text{ is bound to '}
\]

\[
(\text{proc2} 'a) \Rightarrow x \text{ is bound to '('}
\]

\[
(\text{proc2} 'a 'b 'c) \Rightarrow x \text{ is bound to '('}\)
\]

\[
\text{Define list-args}
\]

\[
(\text{lambda} \text{varparam}
\text{<varparam>})
\]

\[
(\text{list-args}) \Rightarrow ()
\]

\[
(\text{list-args} 'a) \Rightarrow (a)
\]

\[
(\text{list-args} 'a 'b 'c 'd) \Rightarrow (a b c d)
\]
parameter lists: example

(define rev-args
  (lambda varparam
    (reverse varparam)))

(rev-args) ==> ()
(rev-args 'a 'b 'c) ==> (c b a)
(rev-args 5 4 3 2 1) ==> (1 2 3 4 5)

parameter lists: specifying optional parameters

(define proc3
  (lambda (x . y)
    <body>))

(proc3 1) ==> x is bound to 1; y is bound to ()
(proc3 1 2) ==> x is bound to 1; y is bound to (2)
(proc3 1 2 3) ==> x is bound to 1; y is bound to (2 3)
(proc3) ==> Error: requires at least 1 argument

parameter lists for optional parameters: example

(define sum-non1-args
  (lambda (first-par . varparam)
    (apply + varparam)))

(sum-non1-args 1) ==> 0
(sum-non1-args 1 2) ==> 2
(sum-non1-args 1 2 3) ==> 5
(sum-non1-args) ==> Error: requires at least 1 argument

parameter lists for optional parameters: example

(define sum-non12-args
  (lambda (first-par sec-par . varparam)
    (apply + varparam)))

(sum-non12-args 1 2) ==> 0
(sum-non12-args 1 2 3 4) ==> 7
(sum-non12-args 1) ==> Error: requires at least 2 arguments
Programs are viewed as collections of functions.

Execution of programs is viewed as evaluation.

- Referential transparency:
  - The value of a function application is independent of the context in which it occurs.
    - value of \( f(a, b, c) \) depends only on values of \( f \), \( a \), \( b \) and \( c \).
    - it does not depend on the global state of computation.

A language is referentially transparent if we may replace one expression with another of equal value anywhere in a program without changing the meaning of the program.

- No assignment statement:
  - Variables are bound to values only through the association of actual parameters to formal parameters in function calls.

A variable may not have a value, or its value may be a function that has not been applied to its arguments yet.
pure functional languages

• No assignment statement:

Variables in pure FP are said to be logical in that, having acquired a value in the course of an evaluation, they retain that value until the end of evaluation.

Similar to how variables are used in mathematics. Remember when you had to unlearn your mathematics training to make sense of \( x := x + 1 \)? In FP, this is anathema!

pure functional languages

• No side effects:

Function calls have no side effects.
  • No need to consider global state.
  • Programs are easier to reason about.
  • Programs are easier to prove correct.

pure functional languages

• Functions are first-class values:

  • Can be returned as the value of an expression.
  • Can be passed as an argument.
  • Can be put in a data structure as a value.
  • Unnamed functions exist as values.

pure functional languages

• Control Flow: conditionals and recursion

  • recursion vs. iteration:

In pure FP, recursion is generally preferred as a matter of style to iterative constructs like while-loops or for-loops. This makes programs
  • easier to prove correct,
  • easier to conceptualize as functions:

Recursions, like functions, identify the structure of one or more arguments as the only values upon which the computation (and termination) depends.
pure functional languages

- **Support higher-level programming:**
  - All storage management is implicit:
    we don’t have to think about how program state maps to a computer’s memory (or at least, not until we want to write super-efficient code). That means no assignment, no new/free calls to manage our own memory, and no pointers.
  - The state of a computation is much easier to think about.

structural recursion

- Processing recursively-structured data (e.g., a list), by decomposing it into its structural components (e.g., first & rest of the list), and then processing those components.

```
(define my-func
  (lambda (lst)
    (cond ((empty? lst) ... )
      (else ... (first lst) ...
        (my-func (rest lst)) ... ))))
```

tail-recursion

- The recursive call is in the last function application in function body, i.e., there is nothing to do after the function returns except return its value.

Tail-recursion is implemented very efficiently and should be used whenever possible.

recursion

- **Linear recursion:** there is at most one recursive call made in any execution of function body.

- **Flat recursion:** recursion applied over 'top' items of a (nested) list.

- **Deep recursion:** (aka tree recursion) recursion applied over all items.

- **Mutual recursion:** functions call each other, rather than themselves.