# FL Example Programs

An FL program that defines a function that counts the number of elements in a given list:

(  
 (count L = (if (null L)  
 0  
 (+ 1 (count (rest L)))))  
 )

An FL program that defines reverse which uses append:

**(extra brackets removed Feb 28, see forum discussion)**

(  
 (reverse X = (if (null X)   
 nil  
 (append (reverse (rest X))   
 (cons (first X) nil))))   
  
 (append X Y = (if (null X)   
 Y  
 (cons (first X) (append (rest X) Y)))  
 )  
 )

Last modified: Sunday, 28 February 2016, 3:40 PM

# fl-interp

Example calls to fl-interp

; a program may be empty.   
  
 (fl-interp '(+ 1 2) nil)  
=> 3  
  
; a function call may be nested  
  
 (fl-interp '(f (f 2))   
 '( (f X = (\* X X)) )  
 )  
=> 16  
  
  
  
; function symbols may be quite arbitrary  
  
 (fl-interp '(a (+ 1 2))   
 '( (a X = (+ X 1)) )  
 )  
=> 4  
  
 (fl-interp '(b (+ 1 2))   
 '( (b X = (+ X 1)) )  
 )  
=> 4  
  
  
 (fl-interp '(h (g 5))  
 '( (g X = (g (g X)))  
 (h X = a ) )  
 )  
  
=> a ; for normal order reduction, and   
 ; non-terminating for applicative order reduction  
  
  
; in general, to avoid typing long expressions repeatedly, you may do, e.g.,   
  
  
(setq E '(reverse (1 2 3)))  
(setq P '(  
 (reverse X = (if (null X)   
 nil  
 (append (reverse (rest X))   
 (cons (first X) nil)))  
 )  
  
 (append X Y = (if (null X)   
 Y  
 (cons (first X) (append (rest X) Y)))  
 )  
 )  
)  
  
; then invoke  
  
(fl-interp E P)  
=> (3 2 1)  
  
  
; But don't use setq in your program, it's not allowed!

Last modified: Thursday, 4 October 2012, 1:57 PM

# Orders of Reduction

Essentially, applicative order reduction requires that the arguments be evaluated before a function is applied. It applies the (leftmost) innermost function first.

Suppose P is

(  
 (f X Y = (+ X Y))  
 (g X = (+ 1 X))  
 )

Consider applicative order reduction from (f (g 2) (g 1)).

(f (g 2) (g 1))  
=> (f (+ 1 2) (g 1))  
=> (f 3 (g 1))  
=> (f 3 (+ 1 1))  
=> (f 3 2)  
=> (+ 2 3)  
=> 5

In contrast, normal order reduction applies the outermost leftmost applicable function first.

(f (g 2) (g 1))  
=> (+ (g 2) (g 1)) ; f is applied even if its arguments have not been evaluated  
=> (+ (+ 1 2) (g 1))  
=> (+ 3 (g 1))  
=> (+ 3 (+ 1 1))  
=> (+ 3 2)  
=> 5

Normal order reduction has better termination property.

Example.

Let program be

(  
 (g X = (+ X (g (+ X 1)))  
 (f X Y = (if (eq X 0) 0 Y))   
 )

Normal order:

(f 0 (g 1))  
=> (if (eq 0 0) 0 (g 1))  
=> (if T 0 (g 1)) // fixed Oct 17: replaced "true" by T  
=> 0

Applicative order:

(f 0 (g 1))  
=> (f 0 (+ 1 (g (+ 1 1))))  
=> (f 0 (+ 1 (g 2)))  
=> (f 0 (+ 1 (+ 1 (g (+ 2 1)))))  
=> ......

Last modified: Wednesday, 17 October 2012, 1:37 PM

# Example of Reduction

Consider the program

(  
 (count L = (if (null L) 0 (+ 1 (count (rest L)))) )  
 )

The following sequence of reduction steps should give you a good idea how your interpreter should work.

(count (1 2 3))  
  
=> (if (null (1 2 3))  
 0  
 (+ 1 (count (rest (1 2 3)))))  
  
=> (if nil  
 0  
 (+ 1 (count (rest (1 2 3)))))  
  
=> (+ 1 (count (rest (1 2 3))))  
  
=> (+ 1 (count (2 3)))  
  
=> (+ 1 (if (null (2 3))  
 0  
 (+ 1 (count (rest (2 3))))))  
  
=> (+ 1 (if nil  
 0  
 (+ 1 (count (rest (2 3))))))  
  
=> (+ 1 (+ 1 (count (rest (2 3)))))  
  
=> (+ 1 (+ 1 (count (3))))  
  
=> (+ 1 (+ 1 (if (null (3))  
 0  
 (+ 1 (count (rest (3)))))))  
  
=> (+ 1 (+ 1 (if nil  
 0  
 (+ 1 (count (rest (3)))))))  
  
=> (+ 1 (+ 1 (+ 1 (count (rest (3))))))  
  
=> (+ 1 (+ 1 (+ 1 (count NIL))))  
  
=> (+ 1 (+ 1 (+ 1 (if (null NIL)  
 0  
 (+ 1 (count (rest NIL)))))))  
  
=> (+ 1 (+ 1 (+ 1 (if T  
 0  
 (+ 1 (count (rest NIL)))))))  
  
=> (+ 1 (+ 1 (+ 1 0)))  
  
=> (+ 1 (+ 1 1))  
=> (+ 1 2)  
=> 3

Last modified: Sunday, 31 January 2016, 3:28 PM

# Examples of Primitive Functions

(fl-interp '(rest (1 2 (3))) nil)  
==> (2 (3))  
  
(fl-interp '(rest (p 1 2 (3))) nil)  
==> (1 2 (3))  
  
; application can be nested  
  
 (fl-interp '(first (rest (1 (2 3)))) nil)  
==> (2 3)  
  
  
 (fl-interp '(eq (< 3 4) (eq (+ 3 4) (- 2 3))) nil)  
==> nil // changed Oct 17: replaced false, true by nil, T   
   
  
 (fl-interp '(if (> 1 0) (+ 1 2) (+ 2 3)) nil)  
==> 3  
  
 (fl-interp '(if (> 1 0) (if (eq 1 2) 3 4) 5) nil)   
==> 4  
  
 (fl-interp '(cons (first (1 2 3)) (cons a nil)) nil)  
==> (1 a)  
  
 (fl-interp '(and (or T nil) (> 3 4)) nil)  
==> NIL  
  
 (fl-interp '(eq (1 2 3) (1 2 3)) nil)  
==> NIL  
  
 (fl-interp '(equal (1 2 3) (1 2 3)) nil)  
==> T

Last modified: Wednesday, 17 October 2012, 1:41 PM

# Skeleton Program

(defun fl-interp (E P)  
 (cond   
 ((atom E) E) %this includes the case where expr is nil  
 (t  
 (let ( (f (car E)) (arg (cdr E)) )  
 (cond   
 ; handle built-in functions  
 ((eq f 'first) (car (fl-interp (car arg) P)))  
 .....   
  
 ; if f is a user-defined function,  
 ; then evaluate the arguments   
 ; and apply f to the evaluated arguments   
 ; (applicative order reduction)   
 .....  
  
 ; otherwise f is undefined; in this case,  
 ; E is returned as if it is quoted in lisp

Last modified: Sunday, 31 January 2016, 3:30 PM