

Tzewen Wang
ECE578
Winter 2005

Additional Homework

Source code with description

```
/**
 * Additional Homework
 *
 * Tzewen Wang
 * ECE578
 * Winter 2005
 *
 * These functions will convert an infix algebra expression in a list format
 * containing only '+', '-', '*', and '/'. The '-' is concerned only to be a
 * minus operator. A negative real number should have the '-' sign resided
 * with the number. For example, (-1 - -2 + 3) will be turned into
 * (- -1 (+ -2 3)).
 *
 * The basic idea is to take advantage of LISP's list structure and then
 * process every single list including nested list in the following operator
 * precedence order:
 *
 *                               ( ) * / + -
 *
 * All functions are recursive functions. Thus, the inner-most list will be
 * first visited and processed. The list will be processed begin with '*'
 * operator. All infix format with '*' operator will be turned into prefix
 * format and put into a list. For example, an infix expression
 * (1 + 2 * 3 + 4 * 5) will be turned into (1 + (* 2 3) + (* 4 5)). The
 * operands that are in prefix format within a list will not be visited
 * after this. Thus, the next step is to apply the same strategy on a
 * different operator, '+', for instance. The expression will be turned into
 * (+ 1 (* 2 3) (* 4 5)). As a consequence, we may simply apply the same
 * strategy several times with different operators in order of the math
 * precedence.
 */

/**
 * convert (EXP)
 *
 * Converts an infix expression into prefix expression recursively.
 * This is the top level as well as called by another function.
 *
 * Parameter:
 *
 * EXP - an arbitrary expression in infix format.
 *
 * Returns the same expression in prefix format.
 */
(defun convert (EXP)
  (if (atom EXP)
      EXP
      (let ((PEXP (convert-to-prefix '- (convert-to-prefix '+
        (convert-to-prefix '/ (convert-to-prefix '*
          (convert-parent EXP)))))))
        ; Fix "list of list" problem.
```

```

        (if (and (listp PEXP) (equal (length PEXP) 1))
            (first PEXP)
            PEXP
        )
    )
)

;/**
; * convert-parent (EXP)
; *
; * Walks through a list recursively. The recursion does not happen within
; * this function. Another recursion may occur with convert() function in
; * a "cross-talk" fashion.
; *
; * Parameter:
; *
; * EXP - an arbitrary expression in infix format.
; *
; * Returns the same expression in prefix format.
;*/
(defun convert-parent (EXP)
  (if (null EXP)
      NIL
      (cons (convert (first EXP)) (convert-parent (rest EXP))))
  )
)

;/**
; * convert-to-prefix (OP EXP)
; *
; * Processes a list with an operator.
; * This function converts infix expression into prefix expression on a
; * particular operator.
; *
; * Parameter:
; *
; * OP - operands around OP will be processed.
; * EXP - an arbitrary expression in infix format.
; *
; * Returns the same expression in prefix format except the subexpressions
; * in prefix format within lists.
;*/
(defun convert-to-prefix (OP EXP)
  (cond ((equal (length EXP) 1) EXP)
        ((and (>= (length EXP) 2) (equal (second EXP) OP))
         ; when operator is matched, recurse the EXP in different ways...
         (if (null (fetch-after OP EXP))
             (list (cons OP (fetch-before OP EXP)))
             (cons (cons OP (fetch-before OP EXP))
                   (cons (first (fetch-after OP EXP))
                         (convert-to-prefix OP (rest (fetch-after OP EXP))))))
        )
        )
  ; otherwise, keep going,
  (T (append (list (first EXP))
             (list (second EXP))
             (convert-to-prefix OP (rest (rest EXP))))))
)

```

```

)
)

;/**
; * fetch-before (OP EXP)
; *
; * Returns a list of operands that have the given operator consecutively.
; * For example, (1 * 2 * 3 * 4 + 5) would return (1 2 3 4) for '*'
; * operator.
; *
; * Parameter:
; *
; * OP - operands around OP will be processed.
; * EXP - an arbitrary expression in infix format.
; *
; * Returns a list of operands around the operator.
;*/
(defun fetch-before (OP EXP)
  (cond ((equal (length EXP) 1) EXP)
        ((and (>= (length EXP) 2) (equal (second EXP) OP))
         (cons (first EXP) (fetch-before OP (rest (rest EXP)))))
        )
  (T (list (first EXP)))
)

;/**
; * fetch-after (OP EXP)
; *
; * Returns a list of operands and operators beyond the given operator.
; * For example, (1 * 2 * 3 * 4 + 5) would return (+ 5) for '*' operator.
; *
; * Parameter:
; *
; * OP - operands around OP will be processed.
; * EXP - an arbitrary expression in infix format.
; *
; * Returns the rest of expression beyond the operator.
;*/
(defun fetch-after (OP EXP)
  (cond ((equal (length EXP) 1) EXP)
        ((and (>= (length EXP) 2) (equal (second EXP) OP))
         (if (= (length EXP) 3)
             NIL
             (fetch-after OP (rest (rest EXP))))
        )
  (T (rest EXP))
)

;/**
; * Demonstrates the program.
;*/
(defun main ()

```

```
(setf EXP `(1 + 2 * 3 + 4 - 5 * (6 + 7) / 8 - 9))
(print "Infix:")
(print EXP)
(print "Prefix:")
(convert EXP)
)
```

```
[91]> (main)
```

```
"Infix:"
(1 + 2 * 3 + 4 - 5 * (6 + 7) / 8 - 9)
"Prefix:"
(- (+ 1 (* 2 3) 4) (/ (* 5 (+ 6 7)) 8) 9)
```

Program output

```
[91]> (main)
```

```
"Infix:"
(1 + 2 * 3 + 4 - 5 * (6 + 7) / 8 - 9)
"Prefix:"
(- (+ 1 (* 2 3) 4) (/ (* 5 (+ 6 7)) 8) 9)
```