The Mini Lisp Interpreter

The interpreter is interactive. The user enters two kinds of inputs.

Function definitions such as

```
(define double (x) (+ x x ))
```

and expressions, such as:

```
(double 10)
```

Function definitions are simply remembered by the interpreter, and expressions are evaluated. Evaluating an expression is the same as running program in most other languages.

Syntax

input--> expression | fundef

fundef --> (define function arglist expression)

arglist--> (variable*)

equation --> value | variable
  | (if expression1 expression2 expression3 )
  | (while expression1 expression2)
  | (set variable expression)
  | (begin expression+)
  | (optr expression*)

optr --> function | value-op

dam value --> integer
value-op --> + | - | * | / | = | < | > | print

function --> name

variable --> name

integer---> sequence of digits, possibly preceded by a minus sign

name --> any sequence of characters not an integer, and not containing (, ), ;, or space

A function cannot be one of the keywords define, if, while, begin or set or any of the value-ops.

Comments are introduced by the character ';' and continue to the end of the line.

A session is terminated by entering quit.

Expressions are fully parenthesized so parsing can be simplified. For example an expression in C

    i = 2*j + i - k/3

becomes

    (set i (- (+ (* 2 j) i ) (/ k 3)))
Semantics

The meanings of expressions are presented here informally here. Note integers are the only values, so for conditional; 0 represents false and any other value represents true.

1) \( \text{if } e_1 \ e_2 \ e_3 \)  

\( e_1 \) evaluates to true then evaluate \( e_2 \) else evaluate \( e_3 \).

2) \( \text{while } e_1 \ e_2 \)  

Evaluate \( e_1 \); if it evaluates to 0, \text{return 0}. Otherwise evaluate \( e_2 \) and then reevaluate \( e_1 \) until \( e_1 \) evaluates to 0.

3) \( \text{set } x \ e \)  

Evaluate \( e \) and get the value for \( e \), (say \( v \)). Assign \( v \) to \( x \) and return \( v \).

4) \( \text{begin } e_1 \ e_2 \ldots e_n \)  

Evaluates each of \( e_1, e_2, \ldots e_n \), in that order once, and return the value of \( e_n \).

5) \( \text{f } e_1 \ e_2 \ldots e_n \)  

Evaluate each of \( e_1, e_2, \ldots e_n \) and apply that function \( f \) to those values. \( f \) may be a value-op or user defined function; if the latter its definition is found and expression defining the body is evaluated with the variables of its arglist associated with the values of \( e_1, e_2, \ldots e_n \)

\( \text{if}, \text{while}, \text{set} \) and \( \text{begin} \) are called control operators.

All value-ops take two argument except print which takes one. The arithmetic operators and the comparison operators do the obvious. \text{print} evaluates the argument prints it and returns the value.
Example: Greatest Common Divisor in C:

```c
int gcd(int m, int n)
{
    int r = m % n;

    while ( r != 0 )
    {
        m = n;
        n = r;
        r = m % n;
    }

    return n;
}
```

To write this in mini lisp we have to define our own Operators !=, % first.

(Note we don't have ! (not) and % (mod) in our alphabet, so we have to use other characters)

```lisp
(define not( x ) ( if x 0 1) ) ; not operator is Boolean

(define ne (x y) (not (= x y) ) )

(define mod (m n) (- m (* n (/ m n))) )
```
(define gcd (m n)
  (begin
    (set r (mod m n))
    (while (ne r 0 )
      (begin
        (set m n)
        (set n r)
        (set r (mod m n))
      )
    )
  )
  n)
)

Another recursive version:

(define gcd (m n)
  (if (= n 0) (gcd n (mod m n)))
)