A (Lisp-Like) TOY Language [Version2]

The language **TOY** is composed of terms (as opposed to instructions/statements) built up from constants, variables and lesser terms by applying functions. The object of computation (ie. the data structures) are integers … -2 -1 0 1 2 3 ... . **Z** denotes the set of integers. **FUN** is a set non-numeric words naming functions. **MINUS** and **IF** (the constants of **FUN**) name primitive functions from ZxZ to Z. Other names in **FUN** refer to functions defined by terms of the language **L**.

L is the set of all **terms** as defined by the following.

(t1) The variables v1 v2 v3 v4 . . . are terms.

(t2) The constants . . -3 -2 -1 0 1 2 3 . . are terms.

(t3) IF t1 and t2 are terms, then **(MINUS t1 t2)** and **(IF t1 t2)** are terms.

(t4) If fn names a function defined by a term t1 of k variables and s1 . . sk are k terms, then **(fn s1 . . sk)** is a term.

DEFUN

DEFUN is used to create functions in the TOY language. It has the following syntax:

(DEFUN <function name> (<parameter 1> < parameter 2> . . . < parameter n>)
<process description>)

Example: (DEFUN ADD (x y) (MINUS x (MINUS 0 y)))

DEFUN does not evaluate its arguments. It just looks at them and establishes a function definition, which can later be referred to by having the function name appear as the first element of a list to be evaluated. The function name must be a symbolic atom. When DEFUN if used, like any function, it gives back a value.
The value DEFUN gives back is the function name, but this is of little consequence since the main purpose of DEFN is to establish a definition, not to return some useful value.

INTERPRETER. A higher-level expression of interpreting term is built on top of the terms of TOY. These interpreting terms use the functions VALUE<>, SUBST < >, and APPLY < >. The interpreter itself consists of a set of equations (rewrite rules) in these terms used to direct computations in TOY.

SUBST is a function whose domain includes certain t4-terms. Given the t4-term (fn n1 . . nk), let t1 be the term defining the function fn and u1 .. uk be the variables of t1 listed in order of

increasing index. Define **SUBST < fn (n1 . . nk) >**
to be the term formed from t1 by replacing each uj by the corresponding nj.

The functions **VALUE**: TOY --> Z and **APPLY**: FUN x Z* --> TOY are defined as follows.

(v1) VALUE< u > = undefined if u is a variable,

(v2) VALUE< n > = n if n is an integer,

(v3) VALUE< (MINUS t1 t2) >

= < t1 – t2 > if t1 and t2 are integers,

= VALUE< (MINUS VALUE<t1> VALUE<t2>) > otherwise.

VALUE< (IF t1 t2) >

= VALUE<t2> if t1 is positive integer and t2 has a value.

= 0 if t1 is 0 or a negative integer,

= VALUE< (IF VALUE<t1> t2) > otherwise

(v4) VALUE< (fn (s1 .. sk) >

= APPKY<fn (VALUE<s1> .. VALUE<sk>) > if fn is neither IF nor MUNUS

(a) APPLY< fn (n1 .. nk) >.

= VALUE< SUBST<fn (n1 ,. nk) > >.

Where the n1 .. nk are integers corresponding to the values of
VALUE<s1> .. VALUE<sk>.

The evaluation of **APPLY< fn (VALUE<s1> . . . VALUE<sk>) >**
must begin with the . . . VALUE<si> . . . expressions. Viewing interpreter terms in tree-form, in which successive levels correspond to successive nestings of interpreter functions we have

APPLY< . . . >

VALUE<s1> . . . . VALUE<sk>

Such a tree may achieve arbitrary finite depth. But regardless of depth evaluation can occur only at the terminal nodes.

****

A **PROGRAM** in TOY is an expression of the type

(fn n1 . . nk)

Where the term named by fn has k variables, and n1 . . nk are integers. Such a program's computation is the evaluation of

APPLY< fn (n1 . . nk) >

by the equations of the previous paragraph, and the output of the computation (if it terminates) is the value of APPLY<fn (n1 . . nk) >.

Examples.

(el) Let “ADD” name the function defined as follows:

(DEFUN ADD (x y) (MINUS x (MINUS 0 y)))

Then (ADD 4 76) is evaluated by substituting arguments in to the term, using the equations v1, . . v4, a. The program is computed as follows.

APPLY< ADD (4 76) >
VALUE< SUBST<ADD (4 76) >>
VALUE< (MINUS 4 (MINUS 0 76)) >
VALUE< (MINUS VALUE<4> VALUE<(MINUS 0 76)>) >
VALUE< (MINUS 4 <0 - 76>) >
VALUE< (MINUS 4 -76) >
<4 - -76>
80.

(e2) Let “EQUAL” name the function determined by the term

(MINUS (MINUS 1 (IF (MINUS x y) 1)) (IF (MINUS y x) 1))

for integers m and n the program

(EQUAL m n)

returns 1 if m = n, otherwise 0 is returned.

(e3) Let POS, ZERO, and NEG be names for

(IF x 1)
(EQUAL x 0)
(IF (MINUS 0 x) 1)

respectively. These functions are the characteristic functions for the indicated sets of integers.

(e4) Let IF/THEN/ELSE denote

(ADD (IF x y) (IF (MINUS 1 x) z))

The value of the term

(IF/THEN/ELSE x y z)

is VALUE<y> if VALUE<x> is positive,
is VALUE<z> if VALUE<x> is zero or negative,
is undefined otherwise.

Recursive terms are either (1) named terms which contain their own names, or (2) terms containing recursive subterms. Recursive terms can be used to program recursive computations. For an example, multiplication in this TOY language corresponds to a recursive term.

(e5) Let TIMES name the term

(IF/THEN/ELSE x (ADD y (TIMES (MINUS x 1) y)) 0)

We will now prove, for non-negative n and all k, that

(TIMES n k)

yields the product of n with k.

When n = 0

1 TIMES (0 k)

2 APPLY< TIMES (0 k) >

3 VALUE< SUBST< TIMES (0 k) >>

4 VALUE< (IF/THEN/ELSE 0 (ADD k (TIMES (MINUS 0 1) k)) 0) >

…….

5 VALUE<0>

6 0

Assume that TIMES (m k) returns m*k and that n=m+1>0.

1′ TIMES (n k)

2′ APPLY< TIMES (n k) >

3′ VALUE< SUBST< TIMES (n k) >>

4′ VALUE< (IF/THEN/ELSE n (ADD k (TIMES (MINUS n 1) k)) 0) >

5′ APPLY< IF/THEN/ELSE (n VALUE<(ADD k (TIMES (MINUS n 1) k)) > 0)>

……

6′ APPLY< IF/THEN/ELSE (n VALUE<k>+ VALUE<(TIMES m k)> 0) >

……

7′ APPLY< IF/THEN/ELSE (n k+m*k 0) >

8′ VALUE< (IF/THEN/ELSE n n*k 0) >

9′ n*k

Notice that steps have been left out after lines 4, 5′ and 6′. However, by induction on n, we have verified that for non-negative n and all m the program (TIMES n k) return the product n*k.