N-code

A general purpose linked structure is used to represent the tree structure program, called "list". List composed from two kinds of nodes: dot-pair and atom. dot-pair stored two components; first component is a pointer to an element of the list and second component is a link to other dot-pair. atom stored information (or element of list). See the following example: (atom is shown in CAPITAL letter) and list is (.). / is NULL pointer.

```
(A B C)

.--.--./
|  |  |
A  B  C

(A (B C))

 .--./
 |  |
 A  .--./
    |  |
    B  C
```

N-code is the representation (the data structure) of Nut language. The structure of program is a list, composed of dot-pairs. An instruction has the form (op arg1... argn) represented by

```
   .--.--.--  .--/
   |  |  |    |
  op  a1 a2 ..an
```

"op" is an atom. Arguments can be either atom or list. A dot-pair composed of a pair of head and tail cells:

```
   head tail
```

The head stores an atom or a pointer to other cell. If it is an atom, the first bit is "1", otherwise it is a pointer, the first bit is "0". The tail stores a pointer to other cell, called "link".

```
   0 dot-pair | 0 link
   1 atom     | 0 link
```

An atom encodes an instruction with one argument.

```
   1 op arg
```

For a 32-bit system, a cell is 32-bit. A pointer to cell is 31-bit (as one bit is used to encode atom/dot-pair). The "op" is 7-bit, the "arg" is 24-bit.

The instruction set

```
control          if while do call fun
value            get put ld st ldx stx ldy sty lit str
arithmetic       + - = < >
other            new sys
```

encoding

```
1 if,  2 while, 3 do, 5 new, 6 add, 7 sub, 10 eq,
11 lt, 12 gt, 13 call, 14 get 15, put, 16 lit, 17 ldx, 18 stx,
19 fun, 20 sys, 25 ld, 26 st, 27 ldy, 28 sty, 32 str
```

Only value-instructions have arguments, denoted by "op.arg". "fun" has special arguments (to be explain later). "call" has a pointer to n-code (a function) as its argument.

The evaluation (execution) of a program employs a stack data structure. All variables are accessed through the structure called "activation record". When a function is evaluated, its has its local environment (local variables and stack area). The activation record is maintained through two global pointers: fp (frame pointer), and sp (stack pointer).

```
hi mem

     ...   <-- sp

     fp'   <-- fp
     lv1
     lv2
     ...
     lvn

lo mem
```

The argument of value-instruction is the index relative to the frame pointer. For example, to get a value of a local variable 3, the instruction "get.3", the access is Mem[fp-3] where Mem [.] is the N-machine memory. Usually this part of memory is called "stack segment".

The meaning of instructions

control-instruction

```
(if e1 e2 e3)
```

if eval(e1) is true then eval(e2) else eval(e3)

```
(while e1 e2)
```

while eval(e1) is true eval(e2) return the last eval(e2)

```
(do e1 ... en)
```

eval(e1) then eval(e2) ... eval(en) return eval(en)

```
(call.x e1 e2..en)
```

eval(e1)... eval(en) keep the results in the evaluation stack and then goto eval at x.

```
(fun.a.v e)
```

create a new activation record, passing arguments from the evaluation stack to this environment, eval(e) the body of function, and delete the activation record. Two parameters

are required to handle creation and deletion of activation record: arity and the size of frame. The encoding is "fun.a.v" where a is arity, v is the size of frame, used in the deletion of AR, k is v-arity+1, used in the creation of AR.

The action of "fun.a.v" is:
SS[.] denotes stack segment

```
k = v-a+1              // offset from sp
SS[sp+k] = fp          // new frame
fp = sp+k
sp = fp
v = eval(e)            // eval body
sp = fp-v-1            // delete frame
fp = SS[fp]            // restore old fp
```

value-instruction

The argument is the index to a local variable. It is relative to the frame pointer.

`get.a` return SS[fp-a]

`(put.a e)` SS[fp-a] = eval(e), return eval(e)

`(ld.a)` load, a is global, return Mem[a]

`(st.a e)` store, a is global, Mem[a] = eval(e), return eval(e)

`(ldx.a e)`
load with index, a is local, return Mem[SS[fp-a] + eval(e)]
where Mem[.] is the memory

`(stx.a e1 e2)`
store with index, a is local, Mem[SS[fp-a] + eval(e1)] = eval(e2)
return eval(e2)

`(ldy.a e)`
load with index, a is global, return Mem[Mem[a] + eval(e)]

`(sty.a e1 e2)`
store with index, a is local, Mem[Mem[a] + eval(e1)] = eval(e2)
return eval(e2)

`lit.a` return a

`str.a` a is a pointer to a string, return a

arithmetic

`(bop e1 e2)`
where bop are + - = < >
they have their usual meaning, return eval(e1) bop eval(e2)

other

`(new e)`
return pointer to a newly allocated chunk of memory of size eval(e)

`(sys.a e)`
system call sys.a
a = 1 print integer eval(e)
a = 2 print character eval(e)
return eval(e)

Example of programs:

```
an expression (= a (+ b 1))

=>  (put.a (+ get.b lit.1))

(def double (x) () (+ x x))

=>  (fun.1.1 + get.1 get.1)

(def sum (a b s) ()
  (if (> a b)
    s
    (sum (+ a 1) b (+ s a))))
=>
(fun.3.3
  (if (> get.a get.b)
    get.s
    (call.sum (+ get.a 1) b (+ get.s get.a))))
```

2 November 2005
Prabhas Chongstitvatana