Chapter 2

High Level Language Nut

Programming is still an art. It requires skill which is acquired through a lot of practice. Its foundation lays in mathematics. The study of programs as an object in itself is interesting and useful. By such study we can understand more thoroughly the relationship between a program and the result we want it to accomplish. It is my intention in this chapter to introduce the study of programs. It will give some insight into programming and gives an appreciation of programs as beautiful man-made objects. A particular high level language called Nut is defined. Nut is the language used to describe all aspects of the system studied in this text. Its syntax and semantic including the internal form will be studied in this chapter.

2.1 Motivation

I will describe a language, Nut language. Nut is inspired by a language defined by S. Kamin in the chapter 1 of his textbook [KAM90]. The beauty of this language stems from its smallness and its elegance. There are 11 words which are already defined (called reserved words). Only one form of syntactic rule is required, using only two characters as syntactic features (the left and right parenthesis). The grammar for this language can be written down in just a few lines. Despite of its look of a toy-language, the beauty of its completeness can be illustrated by showing that the whole executable system including a parser and an evaluator can be completely written in this language.

2.2 Nut Language

Nut employs the same syntax as Kamin's language. It is actually originated in LISP [MCA65]. Nut has a few simple data types such as array and string. The aim of Nut language is for teaching. It has been used in several computer architecture classes to teach how high level programming languages and machine codes are related. The whole language translation process is simple enough that students can modify it to generate code for their studies.

Nut has a very simple syntax. It has only one form, (op arg*). It is designed to be minimal to make it easy to understand. The intermediate code (or internal form) is called N-code. N-code is the data structure representing a program in Nut language. It has a simple static memory model for efficiency, and it also has dynamic memory allocation for flexibility.

The basic element in Nut is an expression. An expression returns a value, except for an assignment which does not return any value. A variable is evaluated to its value. Nut has a very small set of operators as it is intended to be used as a teaching tool. It has a small set of reserved words:

def, let, enum, if, while, do, set, setv, vec, new, and sys.

The operators are: +, −, =, <, and >.

Variables

Nut has three types of variable: global, local, and array. A global variable must be declared outside a function definition before it is used, for example (let v) declares a global variable v. A local variable is defined within a function definition. A local variable's scope is in its defined function. An array variable is a variable that can hold a one-dimensional vector which can be accessed by an index. An array variable has its space allocated by calling (new n) where n is the size of the array. (new n) returns an address which is stored in the array variable. An array is dynamically created. Its space is allocated from the heap.

Simple illustrative examples of Nut programs

The easiest way to introduce a new language is to illustrate many examples of the use of elements of language. The following examples are expressions written in Nut language. The expression is printed in *Italics*. The Nut language is printed in Arial font.

1. A simple expression

b + c + d => (+ b (+ c d))

2. An assignment

   `a = b - d =>` (set a (- b d))

3. A while loop

```
while i < 20
 i = i + 1
```

   (while (< i 20) (set i (+ i 1)))

4. A conditional expression

```
if a > 2 then b = 3 else b = 4
```

   (if (> a 2) (set b 3) (set b 4))

5. A sequence of expression

```
s = 0
a = a + 2
b = 3
```

   (do (set s 0) (set a (+ a 2)) (set b 3))

6. Declaring and allocating an array. A global variable must be declared before its use.

```
ax[20] =>
```

```
(let ax)
(set ax (new 20))
```

7. Getting a value of an element of an array

   `ax[i] =>` (vec ax i)

8. Setting a value of an element of an array

   `ax[k] = 4 =>` (setv ax k 4)

9. Defining a function

```
sq(x) is x * x
```

```
(def sq (x) () (* x x))
```

A function with local variables, swap interchanges ax[a] and ax[b] using a local variable t.

```
(def swap (ax a b) (t)
 (do
 (set t (vec ax a))
 (setv ax a (vec ax b))
 (setv ax b t)))
```

10. To help readability, the enum is used to create symbolic names.

```
(enum 10 xAdd xSub xLit)
```

The symbolic name xAdd is 10, xSub is 11, xLit is 12.

Some elegant examples: defining new logical operators using only if , =, and <.

```
(def and (x y)() (if x y 0))
(def or (x y)() (if x 1 y))
(def not (x)() (if x 0 1))
(def eq (x y)() (= x y))
(def neq (x y)() (not (= x y )))
(def lt (x y)() (< x y))
(def le (x y)() (or (< x y ) (= x y )))
(def gt (x y)() (not ( le x y )))
(def ge (x y)() (not (< x y )))
```

2.3 Nut syntax

Every sentence in Nut is an expression. An expression has the form

```
(op e),
```

where e denotes a list of expressions, op can be any reserved word or a user-defined word. The control-op has the following syntax.

```
(set name e)
(if e1 e2 e3)
(while e1 e2)
(do e1 e2 ... en)
```

The name of a variable and a user-defined word can be any string of characters except the reserved words. The syntax for defining a user-defined function (not built-in) is

```
(def name (formals) (locals) e )
```

where formals is the list of formal parameters, locals is the list of local variables, and e is the body of the function.

The grammar for Nut is as follows. (* denotes zero or more repetition, terminal symbols are in **bold**)

```
toplevel →    e | define-op
e →           name | control-op | value-op | data-op
control-op → ( if e e e ) |
      ( while e e ) |
      ( do e* )
value-op →    ( op args )
data-op →     ( set name e ) |
      ( vec name e ) |
      ( setv name e e )
define-op →   ( def name ( formals ) ( locals ) e ) |
      ( let name ) |
      ( enum number name name* )
op →          + | - | = | < | > | name
args →        name* | number*
formals →     name*
locals →      name*
number →      integer
```

A *name* is the identifier name. There are three types of names: global, local and enumerate. A *global* variable must be declared (using “let”) before its use. A `local` variable is declared inside a scope of the function definition. The `enumerate` is used as a symbolic name referring to some constant value. The `define-op` is the defining operator. There are three define-ops: def, let, and enum. The `value-op` is the value producing operator. The operators are +, -, =, <, and >. The `control-op` is the flow control operator: if, while, and do. The `data-op` is the data access operator: set, setv, and vec.

2.4 Nut semantic

To understand the meaning of a program, the meaning of each of its element must be understood. The arithmetic operators (`value-op`) have their usual meaning on the domain of integer. They evaluate all their arguments which must return integers then apply the operator to these arguments and return the value of integers. The `value-op` including a function call evaluates all of its arguments before applying the operator. This is called *call-by-value* semantic. The other possible meaning is the *call-by-reference*, it is not used in this language. The `control-op` treats its arguments in a different way.

```
(set name e)
```

"set" is an assignment operator. It evaluates an expression e and assigns the value to the variable "name". A variable can be local or global. A variable is local when its name is listed in the formal or local parameters of the current function otherwise it is global. A global variable must be declared by "let".

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

"if" is a conditional operator. It evaluates e1 and if the value is non-zero (true) it evaluates e2 otherwise evaluates e3. The returned value is the value of the last expression it evaluates.

```
(while e1 e2)
```

"while" is an iterative operator. It evaluates e1, if the value is non-zero it evaluates e2. This process is repeated until e1 returns zero. The returned value is the value of e2 before the loop terminates.

```
(do e1 e2 ... en)
```

"do" is a sequencing operator. It evaluates e1 e2 ... en sequentially and returns the value of en.

```
(def name (formals) (locals) e )
```

The *define-op* is used to define a user-defined function. Recursion is quite natural in Nut. For example, the following expression defines a Fibonacci function.

```
fib n is
 if n < 3 then
  return 1
else
 return fib(n-1) + fib(n-2)
```

```
(def fib (n) ()
  (if (< n 3)
    1
    (+ (fib (- n 1)) (fib (- n 2)))))
```

The following example shows a complete program to solve tower of Hanoi problem.

```
(let num)                  ; a global array

; define function "mov" with 3 arguments: n, from, t
; and one local variable: other

(def mov (n from t) (other)
  (if (= n 1)
    (do
    (setv num from (vec num (- from 1)))
    (setv num t (+ (vec num t) 1))
     ; else
    (do
    (set other (- 6 (- from t)))
    (mov (- n 1) from other)
    (mov 1 from t)
    (mov (- n 1) other t))))

(def main () (disk)
  (do
  (set num (new 4))
```

```
    (set disk 6)
    (setv num 0 0)
    (setv num 1 disk)
    (setv num 2 0)
    (setv num 3 0)
    (mov disk 1 3)))
```

The function "main" is the first function to be executed when run this program.

System calls

To enable input/output and other system functions, Nut uses a primitive "sys". It has a variable number of arguments; the first one is a constant, the number that identifies the system function, the rest is the actual parameters passed to the function. "sys" is used to implement library functions such as print an integer, print a charactor, etc. Its implementation is dependent on the platform. For a PC platform, our implementation used C language. The following is the list of available system functions:

(sys 1 a)	evaluate a and print the returned value as an integer
(sys 2 c)	evaluate c and print the returned value as a character
(sys 3)	get a character from a standard input stream

Many system calls are introduced in the later chapters to facilitate low-level system dependent functions.

2.5 Data structures

How can all data structures be implemented in a system which provides only scalar values and arrays in integer domain? For example, how to implement a pointer (so that we can have linked-list and other data structures)? To provide an aggregate of data, an array is used as a general mechanism to provide an indirect access to memory. Accessing an array using the "base" address and the "index" can be regarded as an indirect access to memory. If we know the "base" of the data, then the reference to the data is just an offset (the index) from the base. The index is an ordinary integer. The index is started with zero for the first element of an array. To access an array we need 3 operators: new, setv, vec in this syntax:

```
(new size)
(vec name index)
(setv name index value)
```

"new" allocates memory of "size", where size is an expression, for example (* 4 10) or 40. "new" returns the base address of the allocated memory. The following examples show how these operators are used.

```
(set ax (new 10))          ; set ax to be an array of size 10
(setv ax 1 20)             ; set ax[1] = 20
(set a (vec ax 1))         ; a = ax[1]
```

What really is "name" (such as ax)? It is a variable name. It can be global, or local. A variable name is associated with a reference to the memory that stored its value.

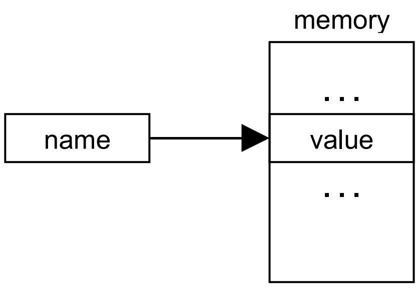


Figure 2.1 A name stored a reference to a memory

A scalar variable refers to only one location in the memory as opposed to an array variable which is associated with a contiguous block of memory. This block of memory is allocated from a part of memory called "heap". To address anywhere in the heap we use "ref" which is an address of this memory block.

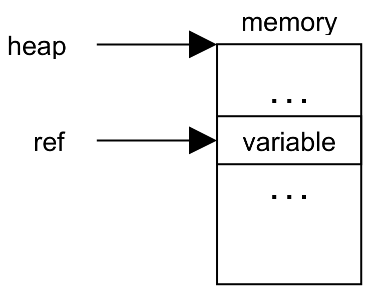


Figure 2.2 A heap in a memory

"vec" evaluates its argument ("name"), gets its value, which is the "ref" to the data segment and uses this reference (plus index) to get the value of the array variable. This indirection is called *dereferencing*. "setv" similarly performs storing a value into an array variable indirectly.

With "vec" and "setv" you can define access functions to your user-defined data structure by implementing the data structure as an array. A calculation on address a variable becomes an ordinary arithmetic on integer.

The following is a program illustrates how to manipulate array variables. The function "array-copy" copies one array to another (in the example, it copies y to x).

```
(enum 10 N)
(let a1)
(let a2)

(def array-copy (x y n) ()
  (if (= 0 n) 0
     (do
     (setv y n (vec x n))
     (array-copy x y (- n 1)))))

(def main () ()
  (do
  (set a1 (new N))
  (set a2 (new N))
  (array-copy a1 a2 N)))
```

2.6 Strings

An array is used to store a string in Nut. A constant string is useful in a source program, for example to present an error message. It is converted into a constant array at compile time. Strings in Nut are implemented with a word-aligned addressing in mind. A string is an array of integer. The string is terminated by an integer 0. See the following program for string manipulation, a string copy.

```
; copy s1 = s2
(def strcpy (s1 s2) (i)
  (do
  (set i 0)
  (while (neq (vec s2 i) 0)
    (do
    (setv s1 i (vec s2 i)
    (set i (+ i 1))))
  (setv s1 i 0)))

(def main () (s1)
  (do
  (set s1 (new 20))
  (strcpy s1 “test string”)))
```

The compiler translated a constant string “test string” in the program text into a constant pointed to data segment storing the string.

2.7 Readability

How easy it is to read a program? This is very much dependent on prior experience. It is a matter of syntax or *form* of the language. Three major types of syntax (based on the concept of operator) are: prefix, infix, and postfix. Most of us grow up to be familiar with infix syntax; (a * 2) + b. For us, this is easier to read than prefix syntax; (* a (+ 2 b)), or postfix syntax; a 2 b + *. The meaning of three forms is the same. However, the difficulty of parsing them is different. The infix syntax requires specifying precedence of operators for the correct association and needs parentheses in places where that precedence must be overridden. A grammar can be written to deal with the precedence. On the other hand, parsing of a prefix and a postfix expression is trivial. Parsing the infix and prefix expression naturally results in a structure of tree while the postfix expression can be transformed into a linear structure easily. However, although a prefix language is trivial to parse, it tends to need a lot of parentheses especially on the far right-hand of the expression which is hard to get it right without the help from an editor that can match parentheses automatically.

The *model* of language also affects its form. The current language distinguishes between *statement* and *expression*. An expression has well-defined mathematical meaning, evaluating an expression returns a value. A language can have expression as the only basic unit. This will make it more compact. Consider the following example:

```
(if x y 0)    is the same as    if( x ) then return y; else return 0;
```

We are more familiar with the right-hand side than the left-hand side (LISP). However you can notice that the left-hand side is much more compact than the right-hand side. The meaning is "evaluate x, if true then evaluate y else evaluate 0". The value returned is the value of the last evaluated expression. There is no need to explicitly "return". The examples of real languages with different syntax are; prefix language, LISP [MCA65], postfix language, FORTH [MOO70] and Postscript.

Let us consider an example of adding one to a variable.

infix syntax	a = a + 1
prefix syntax	(= a (+ a 1))
postfix syntax	&a a 1 + =

For the infix and prefix syntax, the operator "=" (assign) treats its first argument "a" as special, it is an address. For postfix syntax this must be done explicitly using another operator "&". You can not write it the other way. The postfix expression must be understood using the *model* of stack. The central concept is the evaluation stack. Evaluating a variable pushes its value into the stack. An operator takes its argument from the stack and pushes its result back. *Form* also affects the way an operator works. This is an infix language (C):

```
a[1] = a[2] + 1;
```

It actually means `*(&a + 1) = *(&a + 2) + 1;`

The "=" here does not have the same meaning as in `a = a + 1` because it takes the left-hand argument as an expression which must be evaluated to give a value as address where as the "=" in a = a + 1 takes a simple value directly. The parser must know this difference.

2.8 Iteration versus Recursion

Programs can be written in iterative or recursive style. The following examples contrast two styles.

```
(def findName2 (name i) (found)
  (do
  (set found 0)
  (while (and (<= i numNames) (not found))
    (if (streq (def-name-at i) name)
      (set found 1)
      (set i (+ 1 i))))
  (if found i 0)))

(def findName3 (name i)
  (if (> i numNames) 0
  (if (streq (def-name-at i) name) i
  (findName3 name (+ 1 i)))))
```

"findName2" performs a linear search for a name in the symbol table (def-name). "findName2" is iterative and uses "found" to break the while loop. "i" is set to "i + 1" for the next iteration. "findName3" is recursive, "i + 1" is passed as a parameter to the next recursion. Please note the absence of "set" in the recursive version.
The next example is the function "atoi" which converts a string such as "-1234" into its value -1234.

```
(def atoi4 (s1 i) (m)
  (do
  (set m 0)
  (if (= 45 (vec s1 0)) (set i 1) 0)
  (while (!= 0 (vec s1 i))
    (do
    (set m (+ (* 10 m) (- (vec s1 i) 48)))
    (set i (+ 1 i))))
  (if (= 45 (vec s1 0))
    (- 0 m)
    m)))
```

```
(def atoi (s1) ()
  (if (= 45 (vec s1 0))
    (- 0 (atoi2 (+ 1 s1) 0))
    (atoi2 s1 0)))

(def atoi2 (s1 m) ()
  (if (= 0 (vec s1 0))
    m
    (atoi2 (+ 1 s1) (+ (* 10 m) (- (vec s1 0) 48)))))
```

"atoi4" uses iteration with "i" as an index of character and "m" as a local variable storing the converted value. "atoi" and "atoi2" are the recursive version. "atoi" handles the negative sign and calls "atoi2" to convert the string. You can see the simplicity of the structure in the recursive version and the lack of "set".

You may think that recursion consumes more memory and runs slower than iteration. Let us expose more details of this argument. First, the memory concern, most procedural languages use stack to store all local variables and actual parameters. Recursive call will consumes this stack where as iteration does not. However, for the case that the recursive call is the last function executed in a user-defined function, so called *tail-recursion*, this stack growth can be eliminated.

We can eliminate the activation record of the next call (n+1) by realising that the call is the last function executed hence all local variables and parameters of the current activation record need not to be saved (as they are not used anymore). The actual parameters of the next recursive call can substitute the current

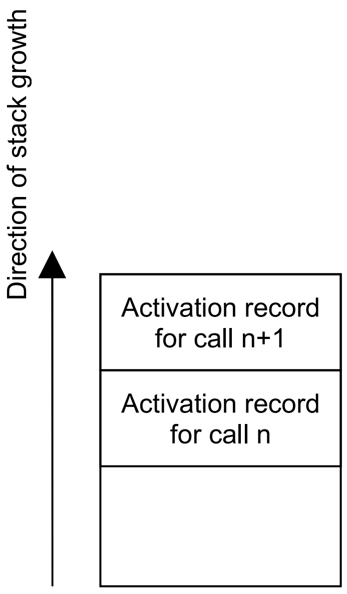

Direction of stack growth
Activation record for call n+1
Activation record for call n

Figure 2.3 A nested call (including recursion) causes growing of activation records

activation record in-place. A parser or a compiler can recognise tail-recursion and performs this optimisation.

Second, the speed concern, the speed of recursive call can be slow due to the overhead of a function call. A function call requires calculating a number of pointers to adjust the stack. Where as for the iteration the loop can be achieved by “jumping” which is a cheaper operation than a call. It depends on the implementation how much this difference will be.

2.9 Internal forms

When an expression (in a source language) is processed, it is transformed into an internal form before it is evaluated (the internal form is also used to generate executable codes). This internal form has the structure in the form of a tree (inverted, the root is at the top). This internal form is distinct from the surface language. One surface language may have different internal forms and different surface languages may have the same internal form. You can think of an internal form as a machine language and a surface language as a high level language. However, an internal form is not a machine language. It is not directly executable by any processor (except you want to design a special processor for it). There is a program that takes an internal form and runs it. This program is called in many names: an interpreter, a virtual machine or an evaluator.

Suppose we have a function power(x, y) which raises x to the power y.

```
(def power (x y)
  (if (= 0 y) 1
  (if (= 1 y) x
  (* x (power x (- y 1))))))
```

The expression defining the body of power can be drawn as Fig. 2.4. A general purpose linked structure is used to represent this tree structure, called *list*. List composed from two kinds of nodes: dot-pair and atom (Fig. 2.5). A dot-pair stores two components; first component is a pointer to an element of the list and second component is a link to other dot-pair. An atom stores information (or element of list). See the following example: (atom is shown in CAPITAL letter and list is `(.)`. `/` is NULL pointer signifying the end of a list.)

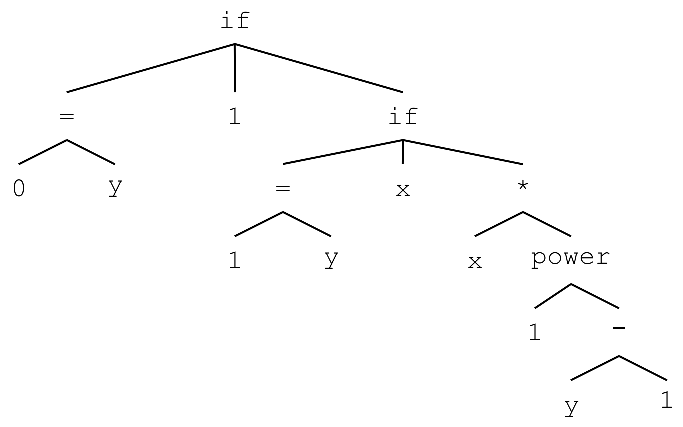


Figure 2.4 The tree representing an expression

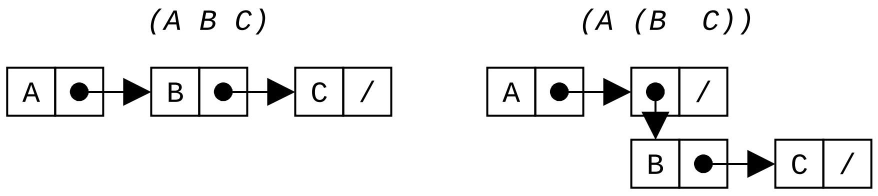


Figure 2.5 Lists can be represented by linked dot-pairs and atoms

The internal form composed of the linked-list nodes with two fields: head and tail. Now we will draw the previous program (power) in this concrete form. The type of node is denoted by `V` (value), `L` (local), `G` (global) and `A` (application).

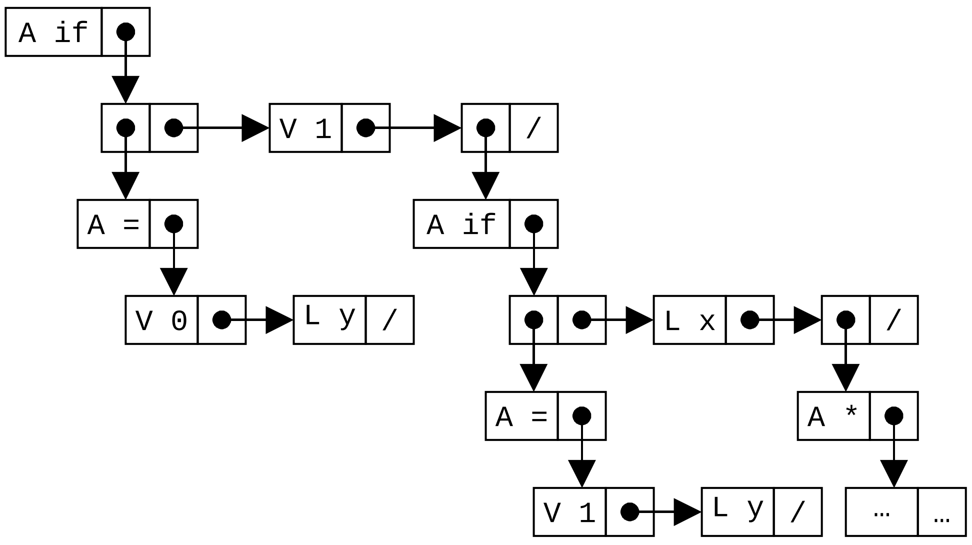
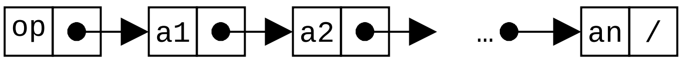


Figure 2.6 The internal form showing the function (power x y)

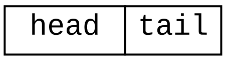
How this internal form is implemented depends on the choice of data structure. In the next section we will discuss this implementation issue in more details.

2.10 N-code

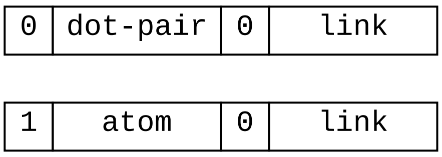
N-code is the internal form of Nut language. The structure of program is a list, composed of dot-pairs. An instruction has the form (op a_1... a_n) represented by



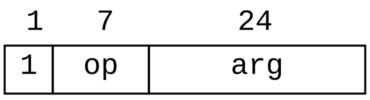
"op" is an atom. Arguments can be either atom or list. A dot-pair composed of a pair of head and tail cells:



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 dot-pair (a pointer to other cell), and the first bit is "0". The tail stores a pointer to other cell, called "link". The basic data structure is a pair of consecutive cells which each cell is large enough to store an atom or a link. There are two kinds of pairs: dot-pair/link and atom/link.



An atom encodes an instruction and one argument.



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.

2.11 N-code instruction set

N-code instruction set is a definition for the internal representation of Nut language. The instruction follows from the Nut language pluses some extra instructions to implement precise operational semantic of Nut language. The instruction set is divided into four groups: control, value, arithmetic and system. Each instruction has the form of an atom with 7-bit opcode and 24-bit argument.

Control	if while do call fun
Value	get put ld st ldx stx ldy sty lit str
Arithmetic	+ − = < >
System	new sys

Encoding

Table 2.1 N-code and its 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			

Totally there are 22 instructions in N-code instruction set. Only value-instructions have arguments, denoted by "op.arg". "fun" has special arguments (to be explained later). "call" has a pointer to its body of a function (the N-code) as its argument.

To understand its operational semantic, we need to know its run-time environment. The run-time environment consists of an evaluation stack and the data segment which provides the place to hold all global, strings and array data.

The evaluation (execution) of a program employs a stack data structure. This evaluation stack has two purposes, one is to store a dynamic local context, called *activation record*, and the second purpose is to be a temporary stack to store the intermediate results. All local variables are accessed through the activation record. When a function is evaluated, it 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). FP points to the activation record. SP points to the temporary stack area. SP is on top of FP.

An activation record has the following structure. At FP, the previous FP (FP') is stored so that the old context can be restored after the current context is complete at the end of a function call. The return address is stored next on the top of FP. This return address is used to restore the instruction pointer (or so called program

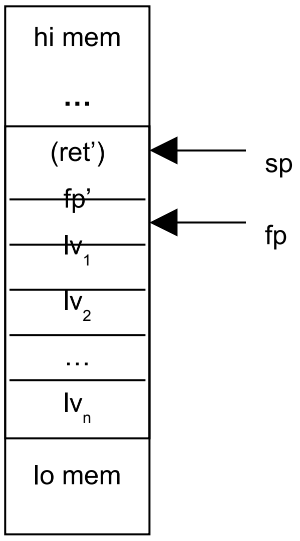


Figure 2.8 An activation record

counter) to enable a program to return to its caller. Storing a return address in the context is necessary in a real processor which implements a data path based on traditional architecture. It is not necessary if we implement an evaluator of N-code as software because the evaluator can be implemented as recursive calls to evaluate each instruction and follows the "`link`" field without using any instruction pointer (the next chapter discusses this Nut-evaluator)[1].

[1] Another possibility is to implement a special processor to execute N-code directly using the recursive evaluation style (with recursive microprogramming). This alternative also does not need to store a return address in the context.

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

The program written in N-code is presented as a list of N-code. It looks similar to the source language Nut but the node values are the operational codes not the tokens of the high level language.
Here is a simple example. A program in Nut to compute a Fibonacci value is shown below.

```
(def fib (n) ()
  (if (< n 3)
    1
    (+ (fib (- n 1)) (fib (- n 2)))))
```

This program when translated into N-code will look like this (N-code is printed in *Italics*).

```
(fun.1
  (if (lt get.1 lit.3)
    lit.1
    (add (call.fib (sub get.1 lit.1)) (call.fib (sub get.1 lit.2)))))
```

The object code represented in the code segment is shown below. The format of object code is as follows. Each line of object code represent one pair of cells written as a tuple of `{address tag op arg link}` where `address` denotes the address of this cell in the code segment, `tag` denotes the first bit – 0 for dot-pair, 1 for atom, `op` denotes the operation code, `arg` denotes its argument, `link` denotes the address of the next cell. The code is generated using the preorder traversal of the source expression; hence the code is generated with the left-most, depth-first order. The last line of the object is the entry point of the expression.

```
2 1 16 3 0
4 1 14 1 2
6 1 11 0 4
8 1 16 1 0
10 1 14 1 8
12 1 7 0 10
14 0 0 12 0
16 1 13 44 14
18 1 16 2 0
```

```
20 1 14 1 18
22 1 7 0 20
24 0 0 22 0
26 1 13 44 24
28 0 0 26 0
30 0 0 16 28
32 1 6 0 30
34 0 0 32 0
36 1 16 1 34
38 0 0 6 36
40 1 1 0 38
42 0 0 40 0
44 1 19 257 42
```

The code above can be read as follows.

```
44 1 19 257 42
```

It is an atom “fun”, the next link pointed to Mem[42].

```
42 0 0 40 0
```

This is a dot-pair with the Mem[40] as the first element and the only element of this list as the next link contains 0 signified the end of list. This 0 is usually called a NIL atom.

```
40 1 1 0 38
```

This is an atom “if”, the next link pointed to Mem[38].

```
38 0 0 6 36
```

This is a dot-pair, the first element is Mem[6], the next link pointed to Mem[36].

```
6 1 11 0 4
```

This is an atom “lt”, with its argument at Mem[4].

```
4 1 14 1 2
```

This is an atom “get.1” and the next argument is Mem[2].

```
2 1 16 3 0
```

This is an atom “get.3” and the list of argument ends here. These three lines can be written out as:

```
(lt get.1 get.3)
```

Other lines of object code can be read similarly. We will not pursue reading the object code of the argument of “if” at Mem[36] any further.

2.12 Meaning of instructions

Now we discuss the meaning of each instruction with respect to its run-time environment. Mem[.] denotes the memory. SS[.] denotes part of the memory that is designated for the stack segment. The code and data are stored in Mem[.] and are called the code segment and the data segment respectively.

The notion of meaning is best explained as the effect of each instruction on its environment. This style of describing the meaning to a program is called *operational semantic* (other ways to describe semantic are axiomatic, denotational and functional). This can be presented as a function “eval()” which takes a valid expression of N-code and produces its result. This function eval() is the evaluator of N-code. It can be implemented both in software (as a virtual machine) or hardware (a special processor that executes N-code directly).

Control-instruction

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

“if” does a conditional execution. If eval(e1) is true then eval(e2) else eval(e3)

```
(while e1 e2)
```

“while” performs a repeat loop. While eval(e1) is true eval(e2) repeatedly, it returns the last eval(e2)

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

“do” is a sequencing operator. eval(e1) then eval(e2) ... eval(en) return eval(en)

(call.x e1 e2..en)

The above expression calls a function with the argument list (e1..en). The element of this list is evaluated one-by-one, eval(e1)... eval(en), the results are pushes to the evaluation stack and then goto eval the body of function at x.

(fun.a.v e)

"fun" is an operational code at the beginning of a function definition. It creates a new activation record. The arguments of the function call are passed from the evaluation stack to this environment, and the body of function is evaluated. Once the evaluation of the body is finished, the activation record is deleted. Two parameters are required to handle creation and deletion of the activation record: arity and the size of frame. The encoding is "fun.a.v" where a is arity, v is the size of frame. The size, v, is used in the deletion of activation record, k is v-arity+1, used in the creation of activation record

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)].

(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 global, Mem[Mem[a] + eval(e1)] = eval(e2), return eval(e2).
lit.a	return a.
str.a	a string constant, a is a pointer to a string, return a.

Arithmetic

(bop e1 e2)	bop are + – = < >. The operators have their usual meaning, return eval(e1) bop eval(e2).

System

System instructions perform the task of input/output and other services related to operating system. On a real processor, the system instructions are implemented differently due to their dependency on a target machine. However, we define these instructions for their use in the simulation. The result of input/output can be simulated on the simulator.

(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 written in N-code

An expression

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

                    (put.a (+ get.b lit.1))
```

Function definitions

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

(fun.1.1 (add get.1 get.1))

```
(def sum (a b s) ()
  (if (> a b)
    s
    (sum (+ a 1) b (+ s a))))
```

```
(fun.3.3
  (if (gt get.a get.b)
    get.s
    (call.sum (add get.a lit.1) get.b (add get.s get.a))))
```

A quicksort program

```
(def partition (a p r) (x i j flag)
  (do
  (set x (vec a p))
  (set i (- p 1))
  (set j (+ r 1))
  (set flag 1)
  (while flag
    (do
    (set j (- j 1))
    (while (> (vec a j) x)
      (set j (- j 1)))
    (set i (+ i 1))
    (while (< (vec a i) x)
      (set i (+ i 1)))
    (if (< i j) (swap a i j) (set flag 0))))
  j ))
```

```
(fun.3.7
  (do
  (put.4 (ldx.1 get.2))
  (put.5 (- get.2 lit.1))
  (put.6 (+ get.3 lit.1))
  (put.7 lit.1 )
  (while get.7
    (do
    (put.6 (- get.6 lit.1))
    (while (> (ldx.1 get.6) get.4)
      (put.6 (- get.6 lit.1)))
    (put.5 (+ get.5 lit.1))
    (while (< (ldx.1 get.5) get.4)
      (put.5 (+ get.5 lit.1)))
    (if (< get.5 get.6)
      (call.swap get.1 get.5 get.6 )(put.7 lit.0)))
  get.6))
```

```
(def quicksort (a p r) (q)
   (if (< p r)
      (do
         (set q (partition a p r))
         (quicksort a p q)
         (quicksort a (+ q 1) r))
      0))
```

```
                  (fun.3.4
                     (if (< get.2 get.3)
                        (do
                           (put.4 (call.90 get.1 get.2 get.3))
                           (call.135 get.1 get.2 get.4)
                           (call.135 get.1 (+ get.4 lit.1 )get.3))
                        lit.0))
```

2.13 Run-time data structure

There are two blocks of memory for run-time data structure supporting N-code: heap and stack segment. Heap contains the code segment and the data segment. Code segment is initialised by the loader. The loader reads an object file and puts the N-code into the code segment. Data segment is used to stored variables, the global data. The global data is allocated by the compiler at the compile-time (when declaring a global variable) and at the run-time by the "new" instruction.

Stack segment contains activation records which stored the data occurred at run-time when a function is called. The activation record contains all local variables and state of computation such as FP (frame pointer) and SP (stack pointer).

When a variable is accessed, for example

```
        (vec a (+ i 10))

                  (ldx.a (+ get.i lit.10))
```

the "get.i" instruction accesses the local variable "i" through the frame pointer from the stack segment by, SS[FP-i] where SS is the stack segment. The "ldx.a" accesses global data in data segment using the base address "a" (from SS) and the index (+ i 10) that is available from the stack (pointed to by SP). The effective address for accessing data segment is ea = SS[FP-a] + SS[SP], then the value of (vec a (+ i 10)) is heap[ea].

A part of memory SS[sp] is the evaluation stack. It stores the intermediate values during computation of an expression. Our scheme combines the activation record with the evaluation stack within a single stack segment.

To make it clear, there are four blocks of memory: code segment, data segment, stack segment, and evaluation stack. Code segment stores N-code, executable machine codes. The data in the code segment is dot-pair and atoms. Data segment stores global data, allocated by "new" instruction. Stack segment stores activation records pointed to by FP. Evaluation stack stores intermediate computation results, pointed to by SP. Our heap combines code segment and data segment. The SS combines stack segment and evaluation stack.

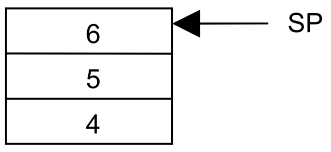
Parameter passing

Actual parameters are evaluated and reside in the evaluation stack. They are passed to a function when a function is called. When a function is called, it creates a new activation record by overlapping its stack frame with the evaluation stack. No parameters need to be copy to the new activation record. For example,

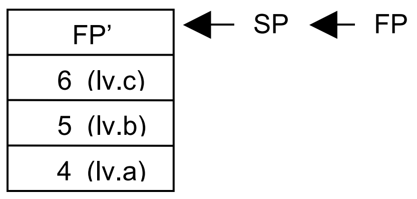
```
(def sum3 (a b c) () (+ a (+ b c)))

(def main () () (sum3 4 5 6))
```

When "main" starts the arguments of "sum3" is evaluated one-by-one and the results are resided on the evaluation stack (pointed to by SP). Here is the picture of the evaluation stack.

W



hen "sum3" is called. "sum3" creates its own activation record on top of the evaluation stack. The new evaluation stack starts at the top of the current activation record (SP = FP).



The local variable "a" can be accessed by an offset from FP, SS[FP-a]. This overlapping also reordered the position of local variables looking from the reference point of FP. The number of a local variable is renamed from *1..n* to *n..1*. This is done by the compiler. The run-time data structure is explained in more details in the next chapter.

2.14 Lab session

Try to run some Nut programs. "nutc" is the Nut compiler. We can use nutc to compile a Nut program into N-code. "nvm" is the "interpreter" of N-code. "nvm" is a virtual machine for N-code. It can execute N-code directly. (In the later chapter, a code generator will be developed which allows N-code to be transformed into a target machine code). The following session shows how to compile and run some Nut program. The following example program is used to illustrate the session.

```
(def print (a) () (sys 1 a))
(def sq (x) () (* x x))
(def main () (a) (print (sq 20)))
```

Assume the example program is in the file "example.txt". "nutc" produces a.obj (an N-code executable file). The output on the screen shows the N-code of the functions that are successfully compiled. "nvm" executes a.obj which prints the result to the screen. Try to write some Nut programs and test them

```
C:\test>nutc < example.txt
print
(fun.1.1 (sys.1 get.1 ))
sq
(fun.1.1 (* get.1 get.1 ))
main
(fun.0.1 (call.18 (call.20 lit.20 )))

C:\test>nvm a.obj
400
C:\test>
```

2.15 Summary

We have introduced a high level language that will be used to describe all levels of the computer system in this text, Nut. The language itself is intended to be a minimal language in a sense that it is a very small language usable for our purpose and yet Nut is complete. It can be used to write its own compiler and evaluator which will be the topic of the next chapter. The smallness of Nut allows us to investigate its semantic in full details. The intermediate representation of Nut language is N-code. N-code is a *concrete* presentation of Nut language. The operational semantic of N-code can be defined over its run-time environment. Given this semantic, a code generator for a target processor can be implemented or a special processor can be designed to directly execute N-code.

2.16 Further reading

Language design is a topic of broad spectrum. Most languages in the past have been constrained by the machines that existed in their period. Overwhelming concern was the issue of machine efficiency. A large survey of computer language is described in [HOR83]. However, as the computing machines become faster and are available abundantly, the emphasis is shifted to the topic of compatibility and standardisation. The history of programming languages is interesting, see [WEX78] [BER96]. It takes time for a language to be widely used and for programmers who are skillful with a language to become available for the industry. Presently, Java [JOY00] dominates the programming in IT industries. It popularises the concept of the *intermediate* language (as JVM the virtual machine [LIN97] is available on almost any platform). The computer language is still evolving. A new language, especially the dynamic language such as the special purpose scripting language, comes into being every year. A special purpose language has its advantage that it can be designed to facilitate the specific programming task, such as Game design, or real-time control task. Special language features can be embedded as primitives in the language such that they are very easy to use. This can reduce the error from programmers. For example, a concurrent language for control tasks can have the message-passing primitives including the semaphore as primitive data types [CHO98]. The future language will take human behaviour into account more than just the machine that run it.

References

[BER96] Bergin, T., Gibson, R., Gibson, R. Jr., History of programming languages, vol.2, ACM Press, Addison Wesley, 1996.

[CHO98] Chongstitvatana, P. A multi-tasking environment for real-time control. Final report, Faculty of Engineering, Chulalongkorn university, research project number 132-MRD-2537, 1998. Also available on-line at http:// www.cp.eng.chula.ac.th/faculty/pjw/r1/

[HOR83] Horowitz, E., Programming languages: a grand tour, Computer Science Press, 1983.

[JOY00] Joy, B., (Ed), Steele, G., Gosling, J., Bracha, G. Java(TM) Language Specification (2nd Ed), Addison Wesley, 2000.

[KAM90] Kamin, S. Programming Languages: An interpreter-based approach, Addison-Wesley, 1990.

[LIN97] Lindholm, T. and Yellin, F. The Java™ Virtual Machine Specification, Addison Wesley, 1997.

[MCA65] McCarthy, J. et al. LISP 1.5 Programmer's Manual, MIT press, 1965.

[MOO70] Moore, C., and Leach, G. FORTH: A language for interactive computing, 1970.

[WEX78] Wexelblat, R., History of programming languages, ACM Press, 1978.

Exercises

2.1 Write a program in Nut to do reversing elements in an array. Try to compile and run it to see the result. Observe the object code. How is the object code (in N-code) corresponded to the source code?

2.2 Familiarise yourself with writing a program with the recursive style. Try the following.
   a) Do the question 1 using recursion.
   b) Search for an element in a linked list using recursion.
   c) Write sum *1..n* using recursion. (Hint: use an accumulating parameter)

2.3 Write a Nut program to read the object code and print it out as a readable N-code in the form of expression in parenthesis.

2.4 The object code (N-code) includes many pointers to other cells. Suggest a way to save memory by reducing these pointers.

2.5 Write N-code (in the printable form) of the following bubble-sort program. Let data be the array storing the elements to be sorted, maxdata be the number of element, swap be a function to swap two elements of the array data.

```
(enum 20 maxdata)
(let data)

(def sort () (i j)
  (do
  (set i 0)
  (while (< i maxdata)
    (do
    (set j 0)
    (while (< j (- maxdata 1))
      (do
      (if (< (vec data (+ j 1)) (vec data j))
        (swap data j (+ j 1)))
      (set j (+ j 1))))
    (set i (+ i 1))))))
```

2.6 Due to the way N-code is represented, a maximum literal representable in Nut language is 24-bit (because a code in n-code is a 32-bit cell, an opcode is 7-bit, an argument is 24-bit). How to represent a larger literal?