Nut language

Prabhas Chongstitvatana
Department of computer engineering
Chulalongkorn University

November 2005

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 intension in this short lecture to initiate you towards the study of programs. Hopefully, to give you some insight into programming but my higher hope is to make you appreciate programs as beautiful man-made objects.

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. (* Prof. Samuel Kamin of the department of computer science, University of Illinois at Urbana Champagne, Programming Languages: An interpreter-based approach, Addison-Wesley, 1990.) 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 syntax 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, I will illustrate the beauty of its completeness by showing that the whole executable system including a parser and an evaluator can be completely written in itself.

The aim of Nut language is for teaching. It has been used in computer architecture class 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 architectural studies.

Nut has a very simple syntax, it is prefix and has only one form, (op arg*). It is designed to be minimal to make it easy to understand. The internal code is a non-linear code (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 allocation for flexibility.

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

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

Operators are: + - = < >

variables

Nut has 3 types of variable: global, local, array. A global variable must be declared outside a function definition before it is used using (let v). A local variable's scope is in its defined

function. An array variable has its space allocated by calling (new n) and assigns the return value to the array variable. An array is dynamic. Its space is allocated in the heap.

Example: a 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 3)
  (setv num 0 0)
  (setv num 1 disk)
  (setv num 2 0)
  (setv num 3 0)
  (mov disk  1 3)))
```

Recursion is quite natural in Nut.

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

Some elegant examples: define some primitives using only: if , =, <.

```
(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 )))
```

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

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

xAdd is 10, xSub 11, xLit 12.

Nut syntax

Every sentences in Nut are expression which return values. An expression has the form

(op e)

where e denotes an expression, op can be any reserved word or user-defined words. 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 user-defined words can be any string of characters except the reserved words. The syntax for defining a user function (non-builtin) is

(def name (formals) (locals) e)

where formals is the list of formal parameters, locals is the list of local variables.

The grammar for Nut is as follows. (* denotes zero or more repetition)

```
toplevel -> e | define-op
e -> name | control-op | value-op
control-op -> "(" "set" name e ")" |
            "(" "if" e1 e2 e3 ")" |
            "(" "while" e1 e2 ")" |
            "(" "do" e1 e2 ... en ")"
value-op -> "(" op args ")"
define-op -> "(" "def" name "(" formals ")" “(“ locals “)” e ")" |
             “(“ “let” name “)” |
             “(“ “enum” number name … “)”
op -> "+" | "-" | "=" | "<" | ">" | name
args -> name* | number*
formals -> name*
locals -> name*
number -> integer
```

Nut semantic

The value-op evaluates all of its arguments before applying the operator. The control-op treats its arguments in a different way.

(set name e)

"set" assigns a value of an expression e to the variable "name". A variable can be local or global. A variable is local when its name is listed in the formal parameters of the current function otherwise it is global. The returned value is the value of e.

(if e1 e2 e3)

"if" evaluates e1 and if its 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 its 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 terminate.

(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.

System calls

To enable input/output and other system functions, Nut uses a primitive "sys". Sys has a variable number of arguments, the first one is a constant, the number that identifies the system function. Sys is used to implement library functions such as print, printchar, etc. The implementation of "sys" is dependent on the platform. For a PC, sys is implemented with the implementation language (C in this version). The following is the list of available system functions:

1 print integer
2 print character
3 get character

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 language. Three major types of syntax (based on the concept of operator): prefix, infix, 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 the postfix syntax; a 2 b + *. The meaning of three forms are the same. However, the difficulty of parsing them are different. Infix syntax requires precedency of operators for correct association and needs parentheses to override that precedency. Grammar can be written to deal with precedency for infix. On the other hand, parsing of prefix and postfix expression is trivial. Parsing the infix and prefix naturally results in a structure of parse tree while postfix results in a linear sequence. However, a prefix language although it is trivial to parse, it tends to have 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 affect its "form". The current language distinguishes between "statement" and "expression". A statement becomes a special property where as 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) = if(x) then return y; else return 0;

We are more familiar with the right hand side (C syntax) 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 example of real languages are; prefix language: LISP, postfix language: FORTH, Postscript.

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

```
infix:  a = a + 1
prefix: (= a (+ a 1))
postfix: &a a 1 + =
```

For infix and prefix, the operator "=" (assign) treats its first argument "a" as special, it is an address. For postfix 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. Evaluate a variable push its value into the stack. An operator takes its argument from stack and pushes its result back to stack. "Form" also affects the way an operator works. This is an infix language (C):

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

The "=" here is not 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.

String

Strings in Nut are implemented with a word-aligned addressing in mind. A string is an array of integer. An integer is 32-bit. It contains at most 4 characters, right pads with 0, terminates by an integer 0. See the following program for string manipulation, 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 in the program text into a constant pointed to data segment storing the string.

Some thought must be exercised to handle string in Nut. String constant is useful in source language for example to present an error message. It is handle at compile time. The most convenient way to implement this is to represent string as an array of integer and the pointer to this string is an address of the memory and let the operator be type-specific, i.e. the operator knows what type its argument is. The pointer to string (similar to char *) is just an integer.

Data structures

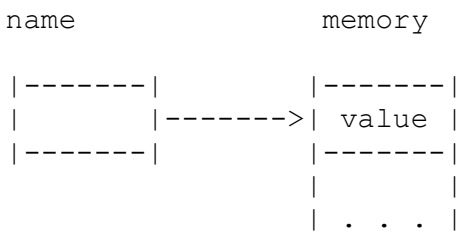
How all the data structure can be implemented in our system, which provide only scalar values in integer domain? For example, how to implement a pointer (so that we can have linked-list and others)? The trick is simple. If we know the "base" of the data, then the reference to the data is just the offset from the base. This offset is not the "address", it is an ordinary integer. In order to provide aggregate of data, we need an array, which is a data structure accessible by an index. The index is the offset that we use as a reference to data. To have an array we need 3 operators: new, setv, vec in this syntax:

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

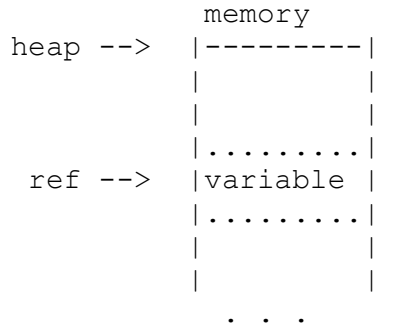
"new" allocates memory of "size", where "size" is an expression, for example (* 4 10) or 40. What really is "name"? It is a global variable, just like the variable that is defined and set its value by "set", for example

```
(set name 1)
```

and the name stores the reference to that memory block.



The memory in our system is a heap, a large block of memory with the base address at "heap". To address any where in the heap we use "ref" which is an index into this array of integer.



"vec" evaluates its argument ("name") gets its value, which is the "ref" to the data segment and using this reference (plus index) to get the value. This "indirection" is called dereferencing. "setv" similary performs storing a value into a name indirectly.

With "vec" and "setv" you can define access functions to your user-defined data structure. A calculation on pointer to a variable becomes an ordinary arithmetic on integer because the reference is just an integer.

The following is a program to copy one array to another (in the example copy 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)))
```

<more example on programming in Nut>

Iteration vs Recursion

The following examples contrast two styles.

```
(def findName2 (name i found)
  (do
    (while (and (<= i numNames) (not found))
      (if (str= (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 (str= (def-name-at i) name) i
  (findName3 name (+ 1 i)))))
```

"findName" searches a name in the symbol table (def-name). "findName2" is iterative and uses "found" to break 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
    (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 growing can be eliminated.

```
stack   |-----|  activation record for call n
  |     |     |
  |     |-----|
  v     |     |  activation record for call n+1
growing |-----|
```

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 need not to be saved (as they are not used anymore). The actual parameters of the next recursive call can substitute the current 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.

Internal forms

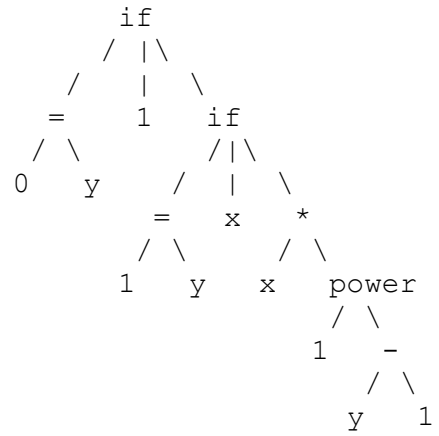
When an expression (in a source language) is processed, it is transformed into an internal form before it is evaluated (or is used to generate executable codes). This internal form has the structure in the form of a tree (inverted, the root is at the top). This internal form is distinct from the surface language, it is called N-code. 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 unlike a machine language. It is not directly

executable in 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:



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. 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.

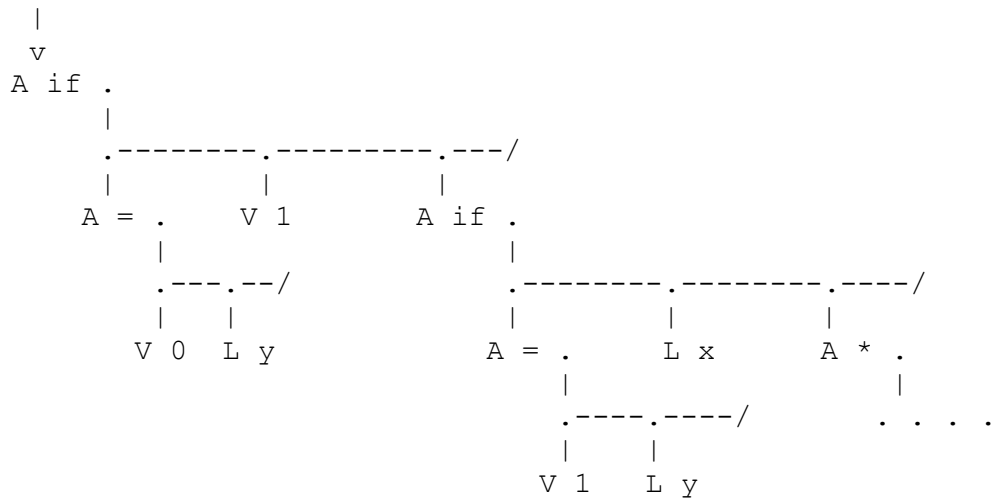
```
(A B C)

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

(A (B C))

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

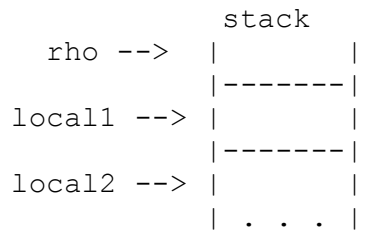
The internal form composed of the linked-list node with two fields: head and tail. Now we will draw the previous program (power) in this concrete form. The type of node is denoted by V (value), L (local), G (global) and A (application).



How eval works?

The evaluator (function eval) is the machine that executes the internal forms (N-code). The global data is allocated from the data segment when the variable is defined. The local data is dynamic and is allocated from the stack segment. The local data is created when passing the actual parameters to a function and is destroyed when exit from the function. Because the function call has the behaviour of last-in-first-out (LIFO) as the earliest call will exit the last, a stack structure is suitable for allocating the local data for function calls.

Using a stack gains a huge benefit of automatic reclamation of the memory when the local data is not longer in used. (You may think this is obvious but this is the beauty of it. Think about other alternative way of storing local data such as linked-list. The local data once ceased to exist will have to be reclaim by some method).



The global and local data can be handled in the same way except that the global data is in the data segment and the local data is in the stack segment.

2 November 2005
P. Chongstitvatana