Chapter 3

Nut Compiler

In this chapter we are going to describe Nut compiler which is written in Nut. A compiler can be written using the target language (of the compiler). Writing a compiler with its own target language demonstrates two points:

1. The language is not trivial. At least it can be used to write some complex program such as a compiler. This shows a kind of *completeness* of the language.
2. The understanding of meaning the language is complete enough to use it to write such a non-trivial program.

And our favourite third point:

3. It is beautiful, in a sense that the language is self-describing.

Writing a compiler with the target language is not new (for example you can write a C compiler in C and compile your C compiler into the executable code using any existing C compiler)[1]. It has been practiced especially in the early days of computer software development. Some conceptual difficulty must be overcome concerning the confusion of the "compiler" and the "compiled" program (since we use the compiler to compile itself!). The run-time facilities are always posing difficulty as the memory is shared between the compiler and the compiled program. However, these points are not a priority in our study.

A compiler translates a source code to a target code. In our study, the Nut compiler translates a Nut program to an N-code object. A code generator translates an object code to a machine specific code. Our code generator translates an N-code object to machine codes.

[1] The question arises as how the first compiler was written? This question is explored in [CHO05].

```
compiler
 input source program (in nut)
 output n-code

code generator
 input n-code
 output machine code of a specific processor
```

To develop “Nut-in-Nut” compiler, we will use a special version of Nut compiler (written in C), “nutc”. This version of Nut-compiler has many additional operators that support compilation. To run the compiler, a version of the *N-code virtual machine* (or the interpreter for N-code), called “nvm” is used. Nvm contains many supporting functions to facilitate compilation (which are cumbersome to write in Nut or which we are not interested in discussing). Remember that our goal is to develop the Nut compiler itself including its own virtual machine. Nutc and nvm are the tools to bootstrap these programs. Once our version of compiler and virtual machine are working, the initial tools will become unnecessary.

Next, we will explain the output of the compiler. It is important to understand the N-code; it is what the compiler must produce.

3.1 N-code

An example, the source program to be compiled:

```
(def add1 x ()
 (+ x 1))

(def main () () (sys 1 (add1 2))
```

Where (sys 1 x) is a system call (analogous to OS call for I/O). It will print an integer x to the screen.

The N-code of the above program in human-readable form is:

```
add1
(fun.1.1 (add get.1 lit.1 ))

main
(fun.0.0 (sys.1 (call.17 lit.2 )))
```

and in an *absolute* object form (as in the output file a.obj):

```
22 22
2 1 16 1 0
4 1 14 1 2
6 1 6 0 4
8 0 0 6 0
10 1 19 257 8
12 1 16 2 0
14 1 13 10 12
16 0 0 14 0
18 1 20 1 16
20 0 0 18 0
22 1 19 0 20
0

17 add1 3 10 1 1
19 main 3 22 0 0
```

There are three parts in the object file:

1 the code
2 the data (the line contains a zero)
3 the symbol table

The code is a contiguous block of memory. The code "main" started at 22. The format of the object code is:

```
{address tag op arg next}
```

`tag` 0 is dot-pair
`tag` 1 is atom
`op arg` is the operator
`next` is the address of the next cell

Now, we will read the object code as follows.

```
22 1 19 0 20
```

It is an atom "fun.0" next is 20

```
20 0 0 18 0
```

It is a list (dot-pair), the head pointed to 18 and the next is NIL.

```
18 1 20 1 16
```

It is an atom “sys.1”, next is 16 (the argument of sys.1)

```
16 0 0 14 0
```

It is a list, the head is 14, next is NIL.

```
14 1 13 10 12
```

It is an atom “call.10”, next is 12 (the argument of the function)

```
12 1 16 2 0
```

It is an atom “lit.2”, next is NIL.

At 10, is the “fun.x” (the function “add1”) etc.

3.2 Compiler

The whole compiler is about 500 lines. We will describe the compiler using a pseudo code and sometimes in Nut language to illustrate some concrete implementation. Full listing of the compiler in Nut is available in the appendix B. We will refer to the source using the notation [name lineno].

The compiler has four main functions.

```
main  [nut 432]
 readinfile
 parse
 resolve
 outobj
```

“readinfile” reads the source program from a standard input stream (`stdin` in Unix). The compiler reads the whole input at once and keeps it in a big array of characters (maximum input size is 50 Kbytes). “parse” is the main parser that scans the input stream and generates N-code. “resolve” performs renaming and binding of the actual code to generate the executable code. “outobj” prints out the object file to a standard output (`stdout`).

We will concentrate on “parse”. To understand “resolve” you need to know the run-time system which is the topic of the next section. Nut has a trivial syntax by design hence the parser for Nut is very simple. Our parser is a *recursive descent* parser. It calls parsing routines recursively with one look ahead symbol and never backtrack (this is called LL(1) parser [AHO86]). This is a simple and straightforward kind of parser. You can read more about this kind of parser in any standard textbook in compiler.

Before we look into the parser, we need to be able to scan the input which is the stream of characters and forms “token”. This is called *lexical analyser*. The tokens are separated with special characters called *separator*. There are only three separators in Nut: space, “(“ and “)”. “tokenise” is one of the system call that is implemented in the “nvm”, (sys 3). It parses the token and returns a string of characters (string of Nut language). Here is the sample use of “tokenise”:

```
; token is a global variable pointed to the string

(let tok)

(def tokenise () ()   [nut 169]
  (set tok (sys 3))

; tokenise input stream until end of file

(def testtok () ()
  (do
  (tokenise)
  (while (!= (vec tok 0) EOF)
    (do
    (prstr tok)
    (space)
    (tokenise)))))
```

Where “prstr” is a function to print a Nut string. The end of file is signified by the first character of the returned string as EOF (127). Run testtok with the example stream and here is the output:

```
( def add1 x ( ) ( + x 1 ) ) ( def main ( ) ( )
( sys 1 ( add1 2 ) )
```

Now we take a look at the parser.

```
parse   [nut 309]
 tokenise
 while not EOF
  expect "("
  tokenise
  if token == "def"
   parseDef
  if token == "let"
   parseLet
  if token == "enum"
   parseEnum
  tokenise
```

The parser gets a token and calls “parseDef” or “parseLet” or “parseEnum” according to the token then loops until it reaches the end of file.

```
parseDef  [nut 269]
 tokenise              get fun name
 parseNL               get formal arg
 parseNL               get local
 tokenise
 e = parseExp          get body
 tokenise              skip “)”
 update symtab
 out (fun.k e)
```

“parseDef” parses the “header” of the function definition then the main part is “parseExp” to parse the body of the function definition. The declaration part of a function definition composed of:

(def add1 x () ...)

“add1” is the function name. “x” is the list of formal parameters. “( )” is the list of local variables. The list of formal and locals is parsed by “parseNL” (parse name list) which will store the formal and local names in the symbol table and gives them the references as a running number *1..n* in order of their appearance.

For sake of clarity, let’s assume that the name list will not be an atom.

```
parseNL  [nut 185]
 tokenise
 while token != ")"
  installLocal token
  tokenise
```

" parseNL" merely gets a name and stores it in the symbol table using "installLocal" until exhausting the list (found the token ")").

Before getting into "parseExp", let's study the symbol table.

Symbol table

The symbol table is a one-dimension array of the entry. Each entry has five fields: *name, type, value, arity, lv*. "*name*" is a pointer to a string, the symbol. "*type*" is the type of symbol. The type value is shown below. "*value*" stores the value of the symbol, which is a reference for function, local/global variable, the opcode of an operator, the syscall number of "sys.x" or the value of an enum symbol. "*arity*" and "*lv*" are for the function symbol, storing its arity and total number of local variables.

Table 3.1 Type of symbols

2	VAR is a local variable
3	FUN is a function
4	OP is an operator
5	OPX is an op that has one special argument
6	SYS is "sys.x"
7	UD is undefined
8	GVAR is a global variable
10	ENUM is an enum symbol

The function "install nm" does a scan for "nm" (value of a pointer to a string, the string of symbol) in the symbol table. Searching a symbol table is efficiently implemented using a hash table. In our implementation, for simplicity, a sequential search is used. If "nm" is already present, "install" returns the index to that entry. If it is a new symbol, it is inserted into the symbol table and the function returns its index. Initially, all keywords are primed into the symbol table. They are treated the same as any other symbol.

The main part of "install" is shown here. The "getName", "setName", "setType" are the access functions for the fields in the symbol table. The variable "numNames" is the number of symbols in the table. "esize" is a constant value of 5, the size of an entry in the table. The "str=" is a string comparison function. The "newName nm" returns a copy of the string "nm".

```
; search symtab for nm, if found, return its index, else insert it

(def install nm (i flag end)   [nut 77]
  (do
  (set i 0)
  (set flag 1)
  (set end (* esize numNames))
  (while (and flag (< i end))        ; sequential search
    (if (str= (getName i) nm)
      (set flag 0)
      ; else
      (set i (+ i esize))))
  (if flag                           ; not found
    (do
    (if (> i MAXNAMES)
      (error "symtab overflow"))
    (setName i (newName nm))
    (setType i tyUD)
    (set numNames (+ numNames 1))))
  i))
```

We conclude the discussion of the symbol table here and go back to discuss the parser. The next function is "parseExp".

```
; An expression is a list, a number, a string or a name
parseExp   [nut 255]
 if token == "("
  tokenise
  nm = parseName
  e = parseEL
  out (nm e)
 if isNumber token
  n = atoi token
  out lit.n
 if isString token
  e = makestring token+1
  out str.e
 parseName                   it is OP, OPX, VAR, FUN
```

The function “parseExp” parses an expression in Nut language. An expression can be: a name (such as a variable), a number, a constant string, a list (function application). For a list, “parseExp” uses two auxiliary functions to parse: parseName, parseEL (expression list).

```
parseName  [nut 214]
 type = symbol.type
 v = symbol.val
 switch type
  OP: out v
  VAR: out get.v
  GVAR: out ld.v
  FUN: out call.idx
  OPX:
   tokenise            get var name
   ty2 = symbol.type
   v2 = symbol.val
   if ty2 == VAR
    switch v
     SET: out put.v2
     SETV: out stx.v2
     VEC:  out ldx.v2
   else if ty2 == GVAR
    switch v
     SET: out st.v2
     SETV: out sty.v2
     VEC:  out ldy.v2
  SYS:
   tokenise
   k = atoi token
   out sys.k
  ENUM:
   out lit.v
```

“parseName” takes one token and depends on its type, it outputs an appropriate N-code. The table 4.2 shows the type and the N-code associated with it.

The operator of type OPX takes the next token as an “unevaluated name”, i.e. a reference of that name, not its value. Three operators are OPX: “set”, “setv” and “vec”. Two possibilities for the next token, either it is a local variable or a global variable.

Local, out put.ref, stx.ref, ldx.ref
Global, out st.ref, sty.ref, ldy.ref

Table 3.2 Type and the associated N-code

OP	an operator, out *opcode*
VAR	a local var, out *get.ref*
GVAR	a global var, out *ld.ref*
FUN	a fun, out *call.ref*
SYS	a system call, gets sys num, out *sys.k*
ENUM	a enum symbol, out *lit.ref*
OPX	a special op

The last piece of the parser, "parseEL" recursively parses the rest of the list.

```
parseEL  [nut 243]
 tokenise
 if token == ")" return NIL
 e = parseExp
 e2 = parseEL
 out (e e2)
```

Now that the major part of compiler is completed, we turn our attention to the housekeeping task. The remaining parts are the "resolve" and the low level "tokenise". We will discuss only their pseudo code.

```
resolve  [nut 368]
 for all func in symtab
  reName call and local var

reName  [nut 356]
 if op == get, put, ldx, stx
  rename local var        (lv-arg+1) 1..n to n..1
 if op == call
  update reference
```

The rename function changes the number of local variables by reversing their order. The reason ties to the way an activation record is created. This run-time behaviour is discussed in the next section. Once the compiler reaches the end of input source, all references to functions should be known. Initially the "call" to a function has the argument as the index to that function in the symbol table. "resolve" also instantiates the actual reference to all "call" instructions. Now the last bit, the tokeniser.

```
tokenise
 skip blank
 get a char
 if isSpecial  return
 if isQuote                 it is a string
  get to other quote
 else
  get to delimiter                 LP RP blank
```

"tokenise" leaves a token string in token[.]. The tokeniser is implemented in the nvm and is available to be used as a system call, (sys 3). See the earlier discussion of the function "tokenise".

3.3 How to compile and run Nut-compiler

The first Nut-compiler is written in C, "nutc". It is used to compile the Nut-compiler describing in this chapter. Nutc outputs the object code to a file, named "a.obj". This object is executable by a Nut virtual machine "nvm". Here is a sample session of compiling the Nut-compiler in the file "nut.txt":

```
c:>nutc < nut.txt

!=
(fun.2.2 (if (eq get.1 get.2 )lit.0 lit.1 ))
and
(fun.2.2 (if get.1 get.2 lit.0 ))
...
main
(fun.0.0 (do (call.79 )(call.124 )(call.157 )
(call.29 )(st.3 (sys.9 ))(call.145 )(call.155 )
(call.156 )(call.23 )))
```

A lot of human-readable N-code is displayed on the screen. It can be visually checked whether it is correct (no error message). The object code can be executed under the nvm simulator. Nvm loads "a.obj" (to save `stdin` for Nut-compiler to use to read its source) and then starts the execution. The result is the execution of Nut-in-Nut compiler, now is in an executable form in "a.obj". The compiler reads the source from `stdin`. Suppose we compile the simple example shows at the beginning of this chapter. Suppose it is in the file "t2.txt".

```
c:>nvm a.obj < t2.txt

add1
(fun.1.1 (add get.1 lit.1 ))
main
(fun.0.0 (sys.1 (call.75 lit.2 )))

9392 9392
9372 1 16 1 0
9374 1 14 1 9372
9376 1 6 0 9374
9378 0 0 9376 0
9380 1 19 257 9378
9382 1 16 2 0
9384 1 13 9380 9382
9386 0 0 9384 0
9388 1 20 1 9386
9390 0 0 9388 0
9392 1 19 0 9390
0
3
add1 3 9380 1 1
x 2 1 0 0
main 3 9392 0 0
```

The object code is outputted to `stdout`. The whole output can be redirect to a file, then select only the code segment to be the object file. Let the name of the object file be "t2.obj"[2]. This object file can be executed under nvm.

```
c:>nvm a.obj < t2.txt > t2.obj
```

Edit "t2.obj" to eliminate surplus listing at the beginning then run it.

[2] The reason why the object of the sample program started at somewhat far address is because there is the Nut-compiler itself (in N-code) already resided in the memory. The compiler (in N-code) takes around 3600 words; the associated data including the symbol table occupies another 5700 words.)

Rename the previous "a.obj" which is the N-code of Nut-compiler and save it in other name, "nut.obj". Now, run "t2.obj".

```
c:>nvm t2.obj
3
c:>
```

This concludes the compilation part. The compiler we discussed so far has no error recovery capability. The error recovery is very important in a practical compiler. It helps programmers to find errors in the program being developed. However, including error recovery will make the compiler itself much more complex. Hence it has been omitted in this presentation.

3.4 Run-time system and the evaluator

The evaluator (function eval) is the program that executes the internal forms (N-code). The listing of the evaluator in Nut (N-code evaluator) is in the appendix D. 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 a last-in-first-out queue (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 an automatic reclamation of the memory when the local data is no 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 reclaimed 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.

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, 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).

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 SS[FP-3] where SS[.] is the memory. Usually this part of memory is called *stack segment*.

Most instructions take their arguments from the evaluation stack. The result (if any) is pushed back to the stack. In this sense, N-code is said to be stack-based instructions. The evaluation stack is local to the current activation record (from FP upward, pointed to by SP).

The instruction "fun.a.v" creates a new activation record; passing arguments from the evaluation stack to this environment, (eval e) where e is the body of function, and deletes the activation record. Two parameters are required to handle creation and deletion of an activation record: *arity* and the *size* of frame. The encoding is "fun.a.v" where *a* is the arity, *v* is the size of frame. They are used in the deletion of activation record. The value *k* is *v− arity+1*, it is used in the creation of activation record. The operational semantic of "fun.a.v" is discussed in detailed in Chapter 2.

3.5 Run-time supports

To actually run N-code, the simulator provides run-time supports. The memory model is an important factor. The actual memory is provided through the implementation language (C in our case). In general, three parts of memory exist:

- code segment − storing N-code
- data segment − storing static/dynamic data
- stack segment − the run-time stack storing activation record and evaluation stack

The pictorial view of the memory is given below. The memory is M[.] with size MEMEND. The code segment and data segment occupied the memory to the maximum limit MEMMAX. The rest is the stack segment. This is how the nvm (the base simulator written in C) arranges its memory.

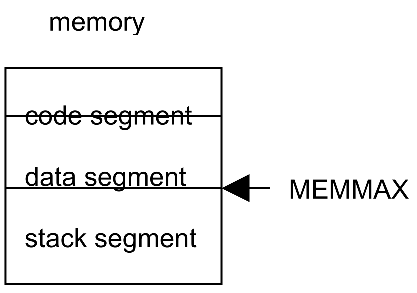

Figure 3.1 The memory layout of the base simulator

How the evaluator arranges its memory?

The loader loads the object into the memory. The N-code starts at the address 2. The data starts at the address 0 and must be relocated to be next to the code segment. In relocating the data, the instructions that involve global variables: ld, st, ldy, sty, str, must offset their arguments (this part is done in the function "resolve").

The base simulator (nvm) loads the object of the evaluator (a.obj) first, then starts executing it. This causes the evaluator to read the object code from `stdin` and evaluating it. The evaluator must relocate its N-code to begin behind a.obj and also its data behind its N-code. To relocate the code, the argument to a call instruction must be changed. To relocate the data, the argument to the instruction accessing globals must be changed. The N-code, data, stack of the evaluator is actually resided in the data segment of the base simulator (nvm). The picture of the memory is shown in Fig. 3.2.

Once the object code is loaded by the evaluator, it starts its execution by allocating its stack segment and sets SP, FP appropriately.

All registers: FP, SP, have been declared as global variables. The stack segment is allocated. The evaluator initialises them before use.

```
(let tok DP CS M)       ; token, data pointer, code segment, memory
(let SS SP FP)          ; stack, stack pointer, frame pointer
```

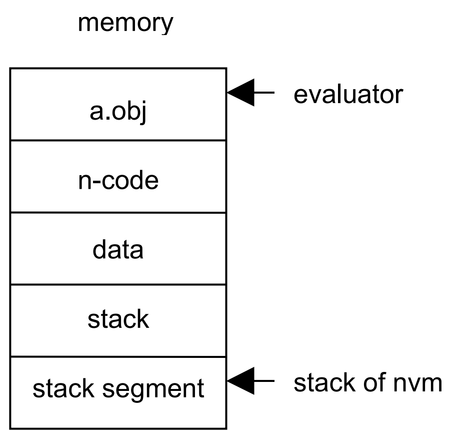

Figure 3.2 The memory layout after loading the evaluator

```
(def init () ()  [eval 23]
  (do
  (set M 0)                    ; base ads, absolute
  (set SS (new STKMAX))        ; allocate stack
  (set FP SS)
   (set SP SS)))
```

3.6 Evaluator

The evaluator is implemented as a separate program. It takes N-code produced from the Nut compiler and runs it. The simulator started by reading the whole input stream (N-code object) into a buffer. Then execute it to instantiate the code segment properly and initialises the simulator variables then begins the evaluation.

```
main   [eval 214]
 readinfile
 loadobj
 initialise
 eval start
```

The main part is the “eval”. We will concentrate on this function. The function “loadobj” performs the housekeeping for the relocating the code and data segment to the appropriate address in the memory. It will be discussed later.

The example below is a fragment of program consists of three functions: prints, add1 and main. (sys 1) prints an integer. (sys 2) prints a character.

```
(let tv)

(def prints s ()    ; print string
  (if (vec s 0)
    (do
    (sys 2 (vec s 0))
    (prints (+ s 1)))))

(def add1 x () (+ x 1))

(def main () ()
  (do
  (set tv 5)
  (prints “string”)
  (sys 1 (add1 11))))
```

This is its N-code in a readable form. A constant string “string” is kept in the data segment at the location 2, it is presented in the object code as “str.2”.

```
prints
(fun.1.1 (if (ldx.1 lit.0 )(do (sys.2 (ldx.1 lit.0 ))(call.15 (add get.1 lit.1 ))))
add1
(fun.1.1 (add get.1 lit.1 ))
main
(fun.0.0 (do (st.1 lit.5 )(call.15 str.2)(sys.1 (call.17 lit.11 ))))
```

The object code of the above fragment is shown below. It is the input stream of the evaluator.

```
2 1 16 0 0
4 1 17 1 2
6 1 16 0 0
8 1 17 1 6
10 0 0 8 0
12 1 20 2 10
. . .
66 0 0 64 0
```

```
68 0 0 56 66
70 0 0 52 68
72 1 3 0 70
74 0 0 72 0
76 1 19 0 74
```

The "eval" function takes a pointer to an expression and evaluates it. "eval" traverses the expression list, when it finds an atom, it evaluates that atom immediately. When if finds a list, the list is in the form (op arg*), it decomposes the list into its operator (op) and the argument list (e1). The action of evaluation is taken according to the operators. The main part of "eval" is this multi-way branch to do each operation.

```
eval e   [eval 160]
 if e is nil return nil
 get the operator and its arg-list, e1
 decode op arg
 switch op
  ADD
   v = (eval arg1 e1) + (eval arg2 e1)
  IF
   if (eval arg1 e1) != 0
    v = eval arg2 e1
   else
    v = eval arg3 e1
  CALL
   eval all arg and push them to eval stack
   v = eval the function
  LIT
   v = arg
  GET
   v = SS[fp-arg]
  ...
 else
  error "unknown op"
return v
```

To limit the scope of discussion, we will discuss in details only the subset of the instruction which is suitable to run the example only (about 11 instructions). These instructions are {fun, if, ldx, lit, do, sys, call, add, get, st}. The value-op are {add, get, lit, str, ldx}. The control-op are {do, if, fun, call}. The other instructions are {sys, st}. Their actions are very like the description of their operational semantic discussed in Chapter 2.

We start with the straightforward value-op.

```
(if (= op xADD)   [eval 181]
  (set v (+ (eval (arg1 e1)) (eval (arg2 e1))))
```

arg1, arg2, arg3 are the access functions to get the first, second and third argument from the argument list. The variable v is the value returned by the function "eval". The operator "add" evaluates two arguments and adds them.

The operator "get" gets a value of a local variable from the activation record.

```
(if (= op xGET)  [eval 194]
  (set v (vec M (- FP arg)))
```

"lit" and "str" have the same effect. The difference in the operation codes is used to distinguish two operators. The loader must relocate everything stored in the data segment. "str" has its argument as a pointer to a constant string stored in the data segment, therefore its argument must be identified and relocate at the load time.

```
(if (= op xLIT)  [eval 190]
  (set v arg)

(if (= op xSTR)  [eval 192]
  (set v arg)
```

"ldx" takes the base address from the argument list, and takes the index from its argument. The effective address is calculated as base + M[FP-arg]. The base is the base address of M[.]. The value is taken from the data segment. The data segment is just a location in the memory.

```
(if (= op xLDX)  [eval 202]
  (do
  (set idx (eval (arg1 e1)))
  (set v (vec M (+ (vec M (- FP arg)) idx))))
```

The control-op alters the sequence of evaluation.

```
(if (= op xDO)  [eval 176]
  (while e1
    (do
    (set v (eval (head e1)))
```

```
          (set e1 (tail e1))))
```

Where (head e) is the first argument of e, (tail e) is the rest of e without (head e). "do" evaluates all the elements in the list. "if" evaluates the condition and selects one of the alternate actions.

```
(if (= op xIF)  [eval 171]
   (if (eval (arg1 e1))
    (set v (eval (arg2 e1)))
    ; else
    (set v (eval (arg3 e1))))
```

"call" evaluates all of its arguments and their value in the stack before evaluating the function.

```
(if (= op xCALL)  [eval 183]
   (do
   (while e1                          ; eval all arg
    (do                        ; and push it to stack
    (push (eval (head e1)))
    (set e1 (tail e1))))
   (set v (eval arg)))                ; eval function
```

And here is how to push a value to the evaluation stack. The evaluation stack is an array of memory pointed to by SP.

```
; push a value to the evaluation stack
(def push e ()  [eval 124]
  (do
  (set SP (+ SP 1))
  (if (> SP (+ SS STKMAX))
    (error "stack overflow"))
  (setv M SP e)))
```

When evaluating a function, the "fun" performs a complicate task of creating a new activation record, setting a new SP, evaluating the body of function, then restore old activation record and SP. The passing of parameters is through the evaluation stack where the new activation record just happens to overlap these variables. There is no copying of passing parameters; they are arranged such that the overlap occurred properly. The numbering of the variables must be ordered in the reversed order of their appearance. This is done during the final compilation phase.

```
(if (= op xFUN)  [eval 145]
  (do
  (set v (& arg 255))              ; decode a, v
  (set a (>> arg 8))
  (set k (+ (- v a) 1))
  (setv M (+ SP k) FP)   ; save old FP
  (set FP (+ SP k))                ; new frame
  (set SP FP)
  (set v (eval (arg1 e)))  ; eval body
  (set SP (- (- FP v) 1))  ; delete frame
  (set FP (vec M FP)))   ; restore FP
```

"st" gets the value of local variable and stores it to the memory. The address is the first argument of argument list.

```
(if (= op xST)  [eval 198]
  (do
  (set v (eval (arg1 e1)))
  (setv M arg v))
```

The last instruction is special. It performs the input/output which are dependent on the underlying physical system. It looks strange that xSYS calls to (sys 1) and (sys 2), but this is absolutely correct because these two functions implement the correct actions; to print integer and character to a display.

```
(if (= op xSYS)   [eval 208]
  (do
  (set v NIL)
  (set a (eval (arg1 e1)))
  (if (= arg 1) (sys 1 a)
  (if (= arg 2) (sys 2 a)
  ; else
  (error "undef sys"))))
```

The rest of the instructions are evaluated similarly. This will conclude our presentation of the function eval.

Now we turn our attention to the housekeeping task, the relocation of the code segment. To relocate the code, the argument to call instruction must be offset. To relocate the data, the argument to the instruction accessing globals must be offset. Assuming the object codes have been read and the following variables have been instantiated: ads, type, op, arg, next. The following code fragment creates a new node for this code and does the relocation.

```
(set CS (sys 9))  [eval 80]                ; find start of code segment
(set DP (+ (+ CS end) 2))                  ;  start of data segment
(if (= type 1)
  (set a (reName op arg))
  ; else dot-pair
  (set a (shift (+ (<< op 24) arg) CS)))
(set a2 (new 2))                           ; create a new node
(sethead a2 a)
(settail a2 (shift next CS))               ; reloc the next
```

Where CS is the start of our code segment (not the N-code of eval itself!), DS is the start of data segment, "sethead" and "settail" update the head and next cells, "shift" performs the offset calculation. The renaming of an atom is done in "reName". "mkAtom" creates an atom from op and arg.

```
; relocate arg of an op
(def reName (op arg) ()   [eval 62]
  (do
  (if (= op xCALL)
    (set arg (shift arg CS))
  (if (or (= op xLD) (= op xST))
    (set arg (shift arg DP))
  (if (or (= op xLDY) (= op STY))
    (set arg (shift arg DP))
  (if (= op xSTR)
    (set arg (shift arg DP))))))
  (mkATOM op arg)))

; offset a by disp, code segment started at 2
(def shift (a disp) ()  [eval 56]
  (if (> a 0)
    (- (+ a disp) 2)
    0))
```

Please bear in mind that the evaluator runs on top of the base simulator "nvm". The N-code of the evaluator itself is loaded as "a.obj" by the "nvm" then the evaluator starts by reading the `stdin` stream which must be the object code to be evaluated.

Understanding "eval" let us confirm the meaning of each N-code instruction. In fact, the "eval" itself can be regarded as the specification of the operational semantic of N-code.

3.7 Lab session

Use the Nut compiler to compile a program, "quick.txt" (a quicksort program). First, compile the Nut-compiler into "a.obj". Then run the compiler with nvm to compile "quick.txt" (the source program). The output is put into a file and edits it to be "quick.obj" (an N-code object). Run "quick.obj" using nvm to see the result. First, compile the compiler.

```
C:\test>nutc < nut.txt
```

Then, use the compiler to compile "quick.txt".

```
C:\test>nvm a.obj < quick.txt > q.obj
```

Edit "q.obj" and put it to the file "quick.obj". The file "quick.obj" looks like this.

```
516 516
2 1 14 1 0
4 1 20 1 2
6 0 0 4 0
8 1 19 257 6
10 1 14 1 0
12 1 20 2 10
...
510 0 0 474 508
512 1 3 0 510
514 0 0 512 0
516 1 19 1 514
0

27
print 3 12952 1 1
...
swap 3 13072 3 4
partition 3 13238 3 7
flag 2 7 0 0
quicksort 3 13298 3 4
```

```
q 2 4 0 0
inita 3 13352 2 3
s 2 2 0 0
show 3 13410 2 3
main 3 13460 0 1
```

Run the N-code object of the quicksort program using nvm.

```
C:\test>nvm quick.obj
20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

C:\test>
```

3.8 Further reading

The intermediate code is a widely known technique used almost in all compilers. It separates the task of compilation into two major phases; the first phase is to parse the source language into this intermediate code and the second phase is to generate the target language (usually machine codes of the target machine) from this intermediate code. This helps to simplify the compiler and also separating the target dependent part from the compiler. This separation is useful when there are many target machines. The first phase can remain the same, only the code generator needs to be done for each new machine.

The evaluator or the virtual machine is one of the most important ideas in computer science. The *emulation* of other machine is a powerful concept. It has been the major cause of the success of computer industry both in terms of producing different hardware that can use the same executable software and in terms of software that can be run on different platform virtually unchanged. The hardware example is the IBM S360 family [PAD81] that can emulate many early IBM computers to such a degree that the customers bought the new machines to run their existing software unchanged. The software example is the Java Virtual Machine (JVM) [VEN98] which is the virtual machine that is available on almost any machines. The use of intermediate code for the purpose of porting a compiler to a new machine is popularised by P-code [BUR78] [WIR91]. Early

Pascal language was compiled into P-code. This made Pascal compiler to be rapidly available throughout the microcomputer communities because the task of creating an executable Pascal compiler was reduced to porting the P-code evaluator. The latest software emulation can be seen from the Apple computer company where their latest computers use a different processor than any of their previous product but the company can made a large number of their existing software available under this new processor in a short time using the emulation.

The technique of writing an evaluator in its own language is called *meta interpreter* [STR88] or more specifically *meta circular interpreter*. It was in practice in early days of computing, the example can be drawn from LISP. The interpreter of LISP was usually written in LISP [MCA65]. The meta interpreter is also useful to reason about the meaning of program and its correctness.

References

[AHO86] Aho, A., Sethi, R., and Ullman, J., Compilers: Principles, techniques, and tools. Addison-Wesley, 1986.

[BER78] Berry, R., "Experience with the Pascal-P compiler", Software – Practice and Experience, 8:617-627, 1978.

[CHO05] Chongstitvatana, P., "Self-Generating Systems: How a $10,000,000_2$-line Compiler Assembles Itself," Proc. of National Computer Science and Engineering Conference, Bangkok, 2005.

[PAD81] Padegs, A., "System/360 and beyond", IBM Journal of research and development, September 1981.

[STR88] Sterling, L., "Constructing Meta-interpreters for Logic Programs", in Advanced School on Foundation of Logic Programming, Italy, 1988 .

[VEN98] Venners, B. Inside the Java Virtual Machine, McGraw Hill, 1998.

[WIR81] Wirth, N. "Pascal-S: a subset and its implementation", in Pascal – The language and its implementation, Barron, D. (ed.), pp. 199-260, Wiley, 1981.

Excercises

3.1 The Nut-compiler has not been finished. There are no "let", "enum" and string. Extend Nut-compiler (nut.txt) to include them. You can have a look at "nut3-compiler.txt" for a guide, or you can look at the C source in "nut31/compile" directory, nut.c.

3.2 Complete the Nut-in-Nut compiler. Use Nut completion kit. Write additional functions so that Nut compiler is able to handle the full Nut language.

3.3 Extend the Nut-in-Nut compiler to include "let" and "enum".

3.4 Nut-compiler does not have the operator: mul, div. Add it to the compiler and simulator (simulator is optional) (Hint: at compiler, you should look at the following functions:

```
#define xMUL 8  in nut.h
add reserved word to keynames[]  in nut.c
prAtom()  in data.c
```

3.5 Strings in Nut can be more efficient by packing 4 characters into one word. Do it.

3.6 The symbol table uses a sequential search [nut 82]. It is not efficient. Implement a more efficient method for searching the symbol table. (Hint: a hash table is a standard way to handle a symbol table. It has a constant running time for searching.)

3.7 How many symbols are there in the symbol table when we compile the Nut-compiler?

3.8 The N-code object can be made *relocatable*, i.e. not dependent on the absolute location in the memory. This can be achieved by *linearising* the N-code tree. The simplest form is the *prefix* form. See the following example:

Source (+ 2 3) becomes readable N-code: (+ lit.2 lit.3) which is stored in the memory (say starts at 6).

```
2 1 16 3 0
4 1 16 2 2
6 1 6 0 4
```

This object is not relocatable, it embeds the absolute location in the "*next*" link. A list can be represented by prefixing it with its length, no "*next*" link is necessary.

```
(+ lit.2 lit.3)
```

becomes

```
3 + lit.2 lit.3
```

another example:

```
(+ lit.2 (+ lit.3 lit.4))
```

becomes

```
3 + lit.2 3 + lit.3 lit.4
```

This representation can be converted into an N-code tree (at any location).

Write a program to output N-code object in linear form to a file and read it back into the memory properly at a different location. You can use "prList" to print the readable N-code out to check it.

3.9 Study eval-in-nut (eval.txt) and try to add some missing operators to it.

3.10 In the main loop of Nut-evaluator [eval 160], the evaluator spent most of its time is checking the opcode and performs the operation accordingly. It uses the form of (if (= op xxx) …). This is a sequential test. Suggest a way to improve the efficiency of the main evaluator loop.