โปรแกรม ฟ. ฟัน

Program For Fun

Prabhas Chongstitvatana

Department of Computer Engineering
Chulalongkorn University

(draft version: December 2010)

Table of contents

Preface

Programming is enjoyable

To study programming is to do programming. Program is an interesting artifact that can be very fascinating. The best way to learn is to enjoy it by doing it. This book will introduce you to a journey of creating a computer language. The story tells an evolution of a computer language, its internal working mechanism and the ideas behind it. The medium of this journey is a computer language and its accompany source code and executable code, called Som language. All the materials can be found at:

http://www.cp.eng.chula.ac.th/faculty/pjw/project/som/index.htm

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.

(Picture of a girl drew by my young daughter)

P. Chongstitvatana
Chulalongkorn University, 2010

Quick introduction

Som is a simple language that is complete enough to be self-hosting, that is, its compiler and its virtual machine are written in Som. It is small. The whole package is around 2,000 lines of code written in Som so it can be read and understood by one person in a short time. Som language has been used in teaching computer architecture, compiler and machine language in my classes for the past 10 years. Its compiler front end is driven by an automatically generated parser. The parser generator in Som is provided. The back end is easily retargeted to different machine languages.

Here is a sample of the language: Matrix multiplication with macro

```
: index i j = (i * N) + j    // this is a macro

to matmul | i j s k =
    for i 0 N-1
        for j 0 N-1
            s = 0
            for k 0 N-1
                s = s + (a[index i k] * b[index k j])
            c[index i j] = s
```

Som's virtual machine has an interesting development. It started the life as a zero-address (stack-based) instruction set and evolved through one-address and then three-address formats. In fact, the history of Som's virtual machine read like a study in instruction set evolution. Som's environment includes a batch-mode compiler and an interactive one. An expression can be typed in and is evaluated immediately. Here is an example of an interactive session:

```
> print 2 + 3 nl
5
> to sq x = x * x
> print sq 4 nl
16
>
```

If you are curious about language design, compiler construction or virtual machine instructions, then you are welcome to try Som. The package is extremely small. All source files and the executable virtual machine plus the compiler object is under 200K bytes. The release includes all source of compiler system (in Som) and the virtual machine (in C). The executable vm is compiled on Windows XP platform, Vista and Windows 7. Because of its smallness, it can be used as an embedded language, or it can be modified for a domain-specific language (through the change of grammar) easily. Hope you enjoy playing with Som.

PS. “Som” is usually a nickname of a girl in Thai language. “Som” is translated to English as a colour “orange” or a fruit “orange” or “tangerine”.

Chapter 1 The language

Motivation

The aim of Som 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 projects easily. Som has a familiar syntax, infix operators and it is designed to be minimal. The basic element in Som is an expression. An expression returns a value. A variable is evaluated to its value. Som has a minimal set of operators. It has small set of reserved words:

to, if, else, while, for, case, break, enum.

Operators

arithmetic:	*+ - * / %*
logic:	*& | ! ^ << >>*
relation:	*== != < <= >= >*
assignment:	*=*
other:	*array* (memory allocation)

Som has three types of variable: global, local, and array. A global variable must be declared outside a function definition before it is used. A local variable's scope is in its defined function. An array variable has its space allocated by calling the "*array*" operators and assigns the return value to the array variable. If an array variable is defined outside a function definition, i.e. global, it is static. A static memory is in data segment. The compiler knows the base address at compile time and can perform some optimisation to achieve faster execution. An array that is defined inside a function definition, i.e. local, is dynamic. Its space is allocated in the heap. The life time of a dynamic memory depends on the use. When there is no reference to it, it is said to be garbage. The run-time system may support garbage collection. Som's programming environment allows using indentation for grouping expressions.

Example: a program to solve tower of Hanoi problem

```
num = array 4      // a global array variable

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

to mov n from t | other =
  if n == 1
    num[from] = num[from] - 1
    num[t] = num[t] + 1
  else
    other = 6 - from - t
    mov n-1 from other
    mov 1 from t
    mov n-1 other t
```

interactive mode

```
disk = 3
num[0] = 0
num[1] = disk
num[2] = 0
num[3] = 0
mov disk 1 3
```

Grammar of Som language

notation:
* zero or more times
+ one or more times
[..] optional
' constant symbol
Indentation is used for grouping, optionally braces can be used '{ '}

```
toplevel -> 'to fundef | ex
fundef -> id args '= ex
args ->    id*  ['| id+ ]
ex -> '{ ex* '} | ex1

ex1 ->
  'if ex0 [ 'else ex ] |
  'while ex0 ex |
  'for lvar ex0 ex0 ex |
  'break |
  'case ex0 caselist |
  'enum '{ [ number ] id+ '}
  id '= ex0 |
  ex0

caselist -> caseitem | '{ caseitem+ '}
caseitem -> number ': ex | 'else ': ex

ex0 -> term term*
term ->
  number | id | vec |
  fun ex0* |
  '! ex0 |
  'array ex0 |
  '( ex0 ')

vec -> id '[ ex0 ']
bop -> '+ | '- | '* | '/ | '& | '| | '^ |
  '== | '!= | '< | '<= | '>= | '> | '% | '<< | '>>
```

There are several interesting points about Som grammar. First, it is expression-based. An expression returns a value. Second, the syntax allows very compact writing with minimum number of separator and parentheses. For example, a semicolon at the end of statement is not necessary as all operators have known arity. Third, the language has very small vocabulary. This makes it very easy to learn. Recursion is quite natural in Som. Look at the following example. It is a definition of Fibonacci number.

```
to fib n =
  if n < 3
    1
  else
    (fib n - 1) + (fib n - 2)
```

Here are some elegant examples. Define some logical functions using only: `if, ==, <`.

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

Control flow operators: for, break, case

For iteration (loop), there are operators: while, for, break. For branching, there are operators: if (else), case. In "for" loop, the index variable must be a local variable. "for i start end body" means:

```
i = start
while i <= end
  body
  i = i + 1
```

See the following example of the use of "for":

```
// fill in an array and print it
max = 10
N = array max

// array is passed by reference
to fill ar n | i =
  for i 0 n-1
    ar[i] = i

fill N max
```

The "`break`" has three meanings:
1) break for loop
2) break while loop
3) force return from a function call.

This is a rather nice semantics as it is very consistent.

The case construction is an efficient way for a multiway branch. The label in each case is an integer. To help readability, "enum" is used to give labels their symbolic names. The symbol ":" makes the syntax look more familiar. A label is stored in the symbol table with a unique reference. The following example shows "case" being used to make an efficient inner interpreter loop in decode and dispatch each instruction. “case” uses a constant time (indexing) to go to the matched label.

```
enum
  1 tAdd tSub

while running
  case opcode
    tAdd :  add
    tSub :  sub
    ...
    else : error "undef opcode"
```

System calls

To enable input/output and other system functions, Som uses a primitive "syscall". Syscall has a variable number of arguments. The first one is a constant, the number that identifies the system function. Syscall is used to implement library functions such as print, printc, loadfile etc. The implementation of syscall is dependent on the platform. It is implemented with C in this version. Here are examples how syscall is used in the library.

```
to print x = syscall {1 x}
to printc c = syscall {2 c}
to getchar = syscall {3}
to loadfile fn = syscall {19 fn}  // fn is som-string
to nl = syscall {2 10}            // 10 is a newline char
```

Why syscall has variable number of argument?

A syscall instruction is an escape hatch. It allows new commands to be added to the language without changing the compiler. Only the virtual machine needed to be updated. Therefore the form (number of argument, whether it outputs any value) of a particular syscall is not known when writing the compiler. An open stack coding can be used to cope with a variable number of arguments. This is easy and does not require any special treatment in parsing. However, it makes a program unreadable. See how confusing it can be in this code (from eval-s.txt):

```
 xArray:            // be careful open stack coding
    pop              // get n from user stack
    a = syscall 8    // alloc M
    push a
```

syscall 8 needs one argument and returns one value. The complexity arises because of open stack coding which does not allow putting argument to syscall like this:

```
  a = syscall 8 pop
```

This is syntax error because function call must know the number of argument.

Tuple

An alternative to open stack coding is to use "tuple". A tuple is a special syntax form that encloses a list of arguments (similar to "block" enclosing a list of expressions). Using tuple appears frequently in the library. The token "{" "}" are used and they are similar to "block".

```
 tuple ->  { ex0 ex0 ... }
```

String

A string data structure in Som is implemented as an array of integer. An integer is 32-bit and contains at most 4 characters. It is right padded with 0 and terminates by an integer 0 (an extra one). This is called a packed string. It is a compromise between the ease of manipulating a string (insert and delete a character is slow) and the space efficiency (storing each character in a 32-bit integer is expensive). See the following example how to manipulate a string.

```
// copy s2 to s1
to strcpy s1 s2 | i =
  i = 0
  while s2[i] != 0
    s1[i] = s2[i]
    i = i + 1
  s1[i] = 0

s1 = array 20
strcpy s1 "test string"
```

The compiler translated a constant string in the program into a constant which pointed to data segment storing the string. Some thought must be exercised to handle string in Som. String constant is useful in source language, for example, to present an error message. 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 * in C language) is just an integer.

Macro definition

To reduce the overhead of a function call, a function can be defined as a "macro". A macro definition is just like a function definition. The difference is that the body of a macro definition is substituted into the call. It is done by textual substitution (similar to a macro language). Hence, the size of a program with a macro is larger than with a function (which is reused). The advantage is that the program will be executed faster. The syntax of a macro definition is similar to a function definition, only the keyword is ":" instead of "*to*". For example,

```
: print x = syscall {1 x}
```

Whenever the macro appears in the program, the macro body is substituted.

```
to report a b =
  ...
  print a
  ...
```

will become

```
to report a b =
  ...
  syscall {1 a}
  ...
```

Macro is suitable for defining access functions such as,

```
to setcar a value = cell[a] = value
to setcdr a value = cell[a+1] = value
to car a = if a == NIL a else cell[a]
to cdr a = if a == NIL a else cell[a+1]
```

These functions will be executed much faster because there is no overhead associated with "call" and "return" such as create/destroy the stack frame. Sometimes the use of macro can create a new control flow, for example in the library, the "short-circuit" and/or are defined:

```
: or a b = if a 1 else b
: and a b = if a b else 0
```

These and/or evaluated their arguments just enough to decide the outcome. This is different from the use of "&" and "|" which always evaluate both of their arguments. A warning is qualified here: an expression written in a macro is different from a normal expression and can cause a subtle bug if it is used carelessly.

More examples and language tutorial

Example 1 Hello world

```
prints "hello world"  // prints is in the string lib
```

Example 2 modulo (it is a built-in operator)

```
to mod m n = m - (n * (m / n))
```

Example 3 quick sort

```
enum
  20 N
```

```
a = array N

to swap i j | t =
  t = a[i]
  a[i] = a[j]
  a[j] = t

to partition p r | x i j flag =
  x = a[p]
  i = p - 1
  j = r + 1
  flag = 1
  while flag
    j = j - 1
    while a[j] > x
      j = j - 1
    i = i + 1
    while a[i] < x
      i = i + 1
    if (i < j) swap i j else flag = 0
  j

to quicksort p r | q =
  if p < r
    q = partition p r
    quicksort p q
    quicksort q+1 r

quicksort 0 (N - 1)
```

Example 4 string manipulation (part of the string lib)

```
// copy s1 = s2
to strcpy s1 s2 | i =
  i = 0
  while s2[i] != 0
    s1[i] = s2[i]
    i = i + 1
  s1[i] = 0

// print string s1, s1 is som-string
to prints s1 | a i c1 c2 c3 c4 =
  i = 0
  a = s1[i]
  while a != 0
    c4 = a & 255
    c3 = (a >> 8) & 255
    c2 = (a >> 16) & 255
    c1 = (a >> 24) & 255
    if c1 != 0 printc c1
    if c2 != 0 printc c2
    if c3 != 0 printc c3
    if c4 != 0 printc c4
    i = i + 1
    a = s1[i]

// b is a constant string, c is a dynamic array
to teststring | b c =
  b = "123456"
  c = array 10
  strcpy c b
  prints c
```

Syntax discussion

The language LISP is an example of a beautiful language. Its syntax is trivial and very flexible. However, it is not easy to get the matching parentheses right without the help from the editor. Functional programming languages such as Haskell also have very elegant syntax, for example pattern matching. To make a language easy to write, the number of parentheses should be minimised. The rules to reduce the amount of parenthesis are:
1) do not use parenthesis with arguments of function call. The arity of any function is known.
2) use infix, left association, and override with parenthesis

With these two simple rules, most parentheses are eliminated. Only 'block' must be decided. The symbols such as { } can be used. Parentheses should not be used to group statements as it is ambiguous between using them to indicate precedency and grouping. The lexical analyser is made so that indentation is recognised and is converted into the proper { }, so in the program text { } are never necessary. Some syntax needs more carefully consideration. See the following grammar.

```
e -> [ e1* ] | e1
e1 -> 'if e e e | 'while e e ... | e Bop t
t -> number | ... | '( e ')
```

Consider "if cond e e", should the "condition" be restricted to non-block? The same reasoning must be applied for the "while e e". Another consideration is the use of ( ). Here I use "(e)" which allows any expression to be ( ). Again, should ( ) be restricted to non-block i.e. ( ) should not be used over { }? I decide to allow full flexibility. However, semantic is not clear. For example for assignment statement "v = e", e must return value. Should a block return a value (perhaps the last e)? What to do with some e that does not return value? The obvious case is "v = e". The value returned by "if e1 e2 e3" depends on what e2, e3 return or they may not return any value. User defined functions may or may not return any value. If "v = e" should return a value (so that cascading them is possible, a = b = 3) then during execution in a loop the evaluation stack will grow. The situation is not too bad because the stack will be clear upon returning from a function call.

Chapter 2 Internals

<more on code generation>

Compiler

The compiler takes a source file and compiles it in two steps. The first step is to read every token (so called lexical analysis) and generates a parse tree representing a syntax tree of the source file. In this step a symbol table containing the names and attributes of all identifiers in the source is also created. The second step takes the parse tree and generates the target language. The parser is a recursive descent parser generated automatically from Som grammar. The parser generator is explained in a separate section. The target language is Som virtual machine instructions. This virtual machine allows Som to be independent of platforms which the code will be executed. It is portable across platforms, only the virtual machine needs to be implemented on the target platform to run all Som programs.

Som virtual machine has a long line of evolution. The first version, s-code, is a simple stack-based instruction set. The current version, t2-code, is a high performance three-address instruction set. To explain how the compiler works, s-code will be used as it is very easy to understand. S-code is briefly explained here. A full detail of s-code can be read in the chapter of virtual machines.

S-code

S-code is a linear sequence of instructions that resemble machine codes. S-code can be executed on a stack-based virtual machine. The goal of S-code is to be a simple language. Essentially it has only around 40 instructions. Two types of instructions are zero-argument and one-argument. The zero-argument instructions do not have any argument embedded in the instructions. They take arguments from the evaluation stack, operate and push the result back to the stack. The most frequently used zero-argument instructions are the binary operators such as add, sub. The one-argument instruction has one argument embedded in the instruction. Mostly this argument is a reference to a local variable. All control instructions (jmp, call etc) need a displacement as their arguments.

zero argument instructions

s-code	description
`add,sub,mul, div, mod`	integer arithmetic, take two operands from the stack and push the result.
`shl, shr`	take two operands: number, no-of-bit and shift the number and push the result back. shr is an arithmetic shift, preserved sign.
`band, bor, bxor`	bit-wise, take two operands from the stack and push the result.
`not`	logical, take one operand and push the result.
`eq, ne, lt, le, ge, gt`	logical, take two operands from the stack and push (T/1, F/0).
`ldx`	take an address, an index and return data at [ads+idx].
`stx`	take an address, an index, a value x, and store x to data at [ads+idx].

case	take a value (key), compare it to the range of label, goto the matched label, or goto else/exit if the key is out of range.
array	allocate x words in data segment, return ref v to the allocated data.

one argument instructions

s-code	description
lit n	push n
inc v	increment local variable
dec v	decrement local variable
ld v	load a global variable, push data[v].
st v	store a global variable, take a value x and store to data[v].
get v	get local variable v.
put v	store a value x to local variable v
call f	call a function, create new activation record, goto f
callt f	tail call, use old activation record.
ret n	return from a function call, n is the size of activation record. remove the current activation record, return a value if function returns a value.
fun n	function header, n is the number of local variables.
jmp n	goto pc+n
jt n	goto pc+n if top of stack != 0, pop
jf n	goto pc+n if top of stack = 0, pop
sys n	call a system function n, interface to external functions, the arguments are in the stack, the number of arguments can vary.

Scheme of compiling is as follows:

notation: op.arg /label

```
if e1 e2 e3      =>  e1 jf.a e2 jmp.b /a e3 /b
while e1 e2      =>  jmp.a /b e2 /a e1 jt.b
for e1 e2 e3     =>  for loop
case e0 ...      =>  case lit.low lit.hi jmp.else jump-
table
v = e            =>  e st.v  (global var)
                 =>  e put.v (local var)
v                =>  ld.v    (global var)
                 =>  get.v   (local var)
f arg*           =>  arg* call.f
n                =>  lit.n
e1 bop e2        =>  e1 e2 bop
unaryop e        =>  e unaryop
 v[i] = e         =>  ld/get.v i e stx
... = v[i]       =>  ld/get.v i ldx ...
```

For loop

"for i start end body" means

```
i = start
while i <= end
  body
  i = i + 1
```

To generate S-code for “for loop” a new local variable is created to be used to store "end". In terms of speed, it is faster because the "end" is evaluated and stored in a local variable just once in the initialization, so testing "`if i <= end`" is faster than evaluate the expression every time around the loop. The code generation is simple. A new local is created to store "stop" value, let's call it "end". The simple form is: (assuming i is local)

```
 lit start
  put i
  lit stop
  put end
  /loop
  get i
  get end
  le
  jf exit
  body
  inc i
  jmp loop
  /exit
```

This is inefficient as it executes two jumps (jf, jmp) per loop. The following form requires only one jump in the loop.

```
  lit start
  put i
  lit stop
  put end
  jmp test
  /loop
  body
  inc i
  /test
  get i
  get end
  le
  jt loop
```

This is also applicable to while loop.

case

There are two alternatives to compile case. The first alternative is to generate table-lookup instruction of the form:

```
<case, else, n, value1, goto1, ... valuen, goton>
```

follows by the code of each case item. The code for each case item is ended with jump to exit. (perhaps using "break" which is transformed into jump) where n is the number of case list, else is the default goto, value is the case label. The jump list is sorted according to value to enable the matching (searching) in log n time using for example binary search. This is very similar to lookup instruction in JVM.

The above compare-and-jump table is the most general form. It can handle all cases of "case". However it is not the most efficient method. A jump table can be constructed that take constant time in searching. When the case label is densed, which is usually the case, the index can be used to access the jump table directly without search.

```
<case2, low, hi, else, goto1...goton>
```

where low, hi is the range of label. If the index is out of range then use "else". hi-low+1 is the size of jumptable. index-lo is used to access the gotos. This is the form we used in Som compiler. Each case action is compiled into a normal sequence of code ended with jmp.end. The entry into the jump table is the code "jmp" to the corresponding ex.

```
case e0 ecase   =>
  e0
  case
  lit.low
  lit.hi
  jmp.else
  jmp.L1
  ...
  jmp.Ln         // jump table
  /L1
  e1
  jmp.end       // each case action
  /L2
  ...
  /Ln
  en
  jmp.end
  /else
  e-else        // default action
  /end
```

An example

Here is some source snippet (from bubble sort), data[ ] is a global array, maxdata = 20.

```
to swap a b | t =
  t = data[a]
  data[a] = data[b]
  data[b] = t

to sort | i j =
  for i 0 maxdata-1
    for j 0 maxdata-2
      if data[j+1] < data[j]
        swap j j+1
```

The code below shows the output listing of S-code (from Som v 2.4). The left column is the function swap. The right column is the inner for-loop of sort. The local variable is numbered in reversed order (to fit the offset from fp). For example, in swap: a is 3, b is 2, t is 1.

```
  45 Fun swap   69 Lit 0
  46 Ld data    70 Put 3
  47 Get 3      71 Lit 20
  48 Ldx        72 Lit 2
  49 Put 1      73 Sub
  50 Ld data    74 Put 1
  51 Get 3      75 Jmp 92
  52 Ld data    76 Ld data
  53 Get 2      77 Get 3
  54 Ldx        78 Lit 1
  55 Stx        79 Add
  56 Ld data    80 Ldx
  57 Get 2      81 Ld data
  58 Get 1      82 Get 3
  59 Stx        83 Ldx
  60 Ret 4      84 Lt
                85 Jf 91
                86 Get 3
                87 Get 3
                88 Lit 1
                89 Add
                90 Call swap
                91 Inc 3
                92 Get 3
                93 Get 1
                94 Le
                95 Jt 76
                96 Inc 4
                97 Get 4
                98 Get 2
                99 Le
                100 Jt 69
```

Interactive mode

Som programming system is interactive. The input source program is typed in and the compiler translates the source into S-codes which then are executed immediately. All expressions are stored in the code segment. We need some markers in the code segment to mark out the defined function so that when a S-code file is loaded it is possible to know which section is to be executed immediately and which section is the function definition. The symbol table is not required when executing the S-code. However the presence of a symbol table facilitates the debugging.

A defined function is marked in CS by "fun m" and ended with "ret n". The layout in CS is like this:

```
to name arg* e  =>   fun m, ... body ..., ret n.
```

At the toplevel, evaluating an expression returns a value which may or may not be used. The unused values are accumulated and they are purged when "ret" is performed.

```
e   =>  e end
```

The action at the toplevel is as follows. Once S-code is loaded (or having been translated into CS), the execution begins. The state of computation is created causing changes in DS and SS. The "end" will return the control back to the console.

Parser Generator

A parser generator takes a grammar (in some form) of a language and generates a parser for that language. The parser generator generates Som parser. The grammar of the parser to be the input of the generator is described as follows:

```
grammar -> 'string | rule | 'eof
rule -> 'id rule2
rule2 ->  '= es | '[ var
var -> 'id var | '] '= es
es -> e1 es | '| es | '%
e1 -> 'id | 'string
```

'string is passed through. It is the "action" part of the generator. If 'id is a terminal symbol, it is a token name (tkEQ ...). If 'id is a nonterminal symbol, it is the rule name appeared on the left hand side of other rules. '| indicates alternatives. '% terminates a rule. The terminal 'nil indicates empty match (always match).

Example This grammar:

```
 args =
    tkIDEN "enterLocal tokvalue" args |
    tkBAR "ypush Nlv" local |
    "ypush Nlv" nil %
  ex =
    tkBB "ypush MARK" exs tkBE "doblock" |
    ex1 %
```

turned into Som parser:

```
 to args =
    while tok == tkIDEN
      enterLocal tokvalue
      lex
    if tok == tkBAR
      ypush Nlv
      lex
      commit local
      1 break
    ypush Nlv
    1

  to ex =
    if tok == tkBB
      ypush MARK
      lex
      commit exs
      expect tkBE
      doblock
      lex
      1 break
    if ex1
      1 break
    0
```

A rule always returns 0 (fail) or 1 (success). As this is really a one-look-ahead parser, returning a 0 means there is an error. An alternative consists of several "match" tokens. The first match is handled differently from the rest. If a token starts with "tk" (tkXX), it is a terminal symbol, otherwise it is a nonterminal symbol.

match first token

```
   tkXX      ->  if tok == tkXX
   nonterm   ->  if nonterm
```

match other token

```
   tkXX      ->  expect tkXX
   nonterm   ->  commit nonterm
```

A match token is followed by "lex" to move over this input token but if there is a string (pass through), "lex" is will be placed after the string.

Example

```
  ex1 =
    tkIF ex0 ex ... |
    tkBREAK "ypush newatom"
```

Becomes

```
  to ex1 =
    if tok == tkIF
      lex
      commit ex0
      commit ex
      ...
    if tok == tkBREAK
      ypush newatom
      lex
      ...
```

An alternative is ended with "1 break" and a rule is ended with "0" except when the last one is "nil", then it is 1 (always match).

Example
```
  ex = ... | ex1 %
```

becomes

```
  to ex =
    ...
    if ex1
      1 break
    0
```

When an alternative is recursive, "while" loop is used.

Example
```
  local =
    tkIDEN "enterLocal tokvalue" local |
    nil %
```

becomes

```
  to local =
    while tok == tkIDEN
      enterLocal tokvalue
      lex
    1
```

When there is multi-choice and the first set are all terminals then a more efficient branch, "case", is used.

Example
```
  bop =
    tkPLUS "ypush tok" |
    tkMINUS "ypush tok" |
    tkSTAR "ypush tok" %
```

becomes

```
  to bop =
    case tok
      tkPLUS:
        ypush tok  lex 1 break
      tkMINUS:
        ypush tok  lex 1 break
      tkSTAR:
        ypush tok  lex 1 break
     0
```

Implementation

The following paragraphs are a short description of how the generator works. It is best read side-by-side with the source code (this is only the main part, for the complete source please see the package somv42a.zip). Grammar is read by a lex syscall in vm (of som 4.2 vm). This lexical analyser knows Som tokens. (If the built-in lex is not used then one can use lex in som from token-s.txt from som 4.1 and earlier). The input grammar has the following form:

```
     grammar -> 'string | rule | 'eof
     rule -> 'id rule2
     rule2 ->  '= es | '[ var
     var -> 'id var | '] '= es
     es -> e1 es | '| es | '%
     e1 -> 'id | 'string
```

Two lists are built from the input grammar. One list is for the "header" and the other list is all rules from the grammar. The parser that parses the input grammar is a recursive descent parser. The output of the parser are all rules that have this form:

```
   ((lhs1  (var)(alt1)(alt2)...)
  (lhs2  (var)(alt1)(alt2)...)...)
```

Each symbol is tagged (type.value) where

```
  type = TERM NONTERM STRING NIL
  value = pointer to its print-string
```

Each token is tagged. A rule in the input grammar is transformed to a list of alternatives and rules. After the input grammar is parsed, the generation of an output parser from the list of rules is done for each rule. The generator extracts the "lhs" (head) of the rule and generates alternatives. Each alternative is checked for recursion. Each match in the grammar is transformed into the appropriate output. The first match is different from the rest. The last match is ignored if the rule is recursive. Finally, the end rule is generated.

The main goal of this parser generator is to generate "multiway branch" (switch, case). It will be compiled into a much faster code. Previously, the parser has been hand-coded in this part. To decide if a rule needs a multiway branch, there are 3 factors:

1) it must not contain recursion
2) it has more than two choices (otherwise if..then else is sufficient).
3) all choices except the last one must has tkXX (non terminal symbol) as the first set.

Example

```
  ex1 =
    ...
    tkENUM tkBB elist tkBE "ypush NIL" |
    tkBREAK "ypush newatom OPER tkBREAK" |
    exas %
```

becomes

```
  to ex1 =
    case tok
      ...
      tkENUM: ...
      tkBREAK: ...
      else:
        if exas
           1 break
     0
```

It should be fine to do "commit exas" but it causes parsing error. This dues to the way "exas" works (and this is the most difficult bug to track down). In the end, this is a good parser generator in ~700 lines of code.

Som parse tree

Here is the data structure of Som parse tree.

list

A list is linked dot-pairs. A dot-pair has two fields: head.tail. The head can be either a dot-pair or an atom. The tail is a pointer points to list or nil (end of list). An atom has two fields: type.value. An atom is distinguishable from a dot-pair because its type field has a small value (0..9), this value is less than any pointer to dot-pair.

Example of a list of A, B, C is *(A B C)*

atom

```
oper.op    gname.idx   lname.idx   string.str   num.value
```

op

```
 add sub mul div eq ne lt le gt ge not band bor bxor
 mod shl shr set vec mac fun call mx if ifelse while for
 break array case sys
```

program

```
fun/mac definition    (oper.fun/mac gname.idx e...)
for                   (oper.for lv ex0 ex0 ex)
while                 (oper.while ex0 ex)
if                    (oper.if ex0 ex)
ifelse                (oper.ifelse ex0 ex ex)
case                  (oper.case ex0 block)
assign                (oper.set var ex)
call/mx               (oper.call/mx gname.idx e...)
block                 (oper.block e...)
syscall               (oper.sys num.value e...)
```

var

```
   gname.idx  lname.idx (oper.vec var ex)
```

Example

```
to sort | i j =
  for i 0 maxdata-1
    for j 0 maxdata-2
      if data[j+1] < data[j]
        swap j j+1

(fun sort
  (for #1 0 (- 20 1 )
    (for #2 0 (- 20 2 )
      (if (< (vec data (+ #2 1 ))(vec data #2 ))
        (call swap #2 (+ #2 1 )))))))

: print x = syscall {1 x}

(mac print (sys 1 #1 ))
```

encoding

encoding of type (0..9)

```
   sp 0  oper 1  num 2  gname 3  lname 4  * 5  string 6
```

encoding of op

```
mul 50    div 51    sub 52     add 53    set 54
 eq 55     band 56   bor 57     bxor 58   mod 59
 not 60    ne 61     lt 62      le 63     shl 64
 gt 65     ge 66     shr 67     mac 68    * 69
 * 70      vec 71    mx 72      block 73  call 74
 fun 75     if 76      ifelse 77  while 78  for 79
 break 80   array 81   case 82    * 83      sys 84
```

S-code optimisation

S-code is intentionally kept simple and minimal to make it easy to be changed. Most of the optimisation described here has been done for a variety of reason. However, the basic Som system employs just a few of these optimisation. Specifically the following: inc v, dec v, short cut jmp to ret, jmp to jmp. In performing the code optimisation, the goal is to reduce the number of instruction executed. This is related to the speed of virtual machine execution. For example, by combining two instructions to make a new instruction, this new instruction is more complex hence taking more time to execute them. However, as the semantic of instruction does not change, the new instruction performs the same amount of work as several old instructions. The execution speed of the virtual machine stems from the reduction in its overhead. The virtual machine is implemented as a big while switch loop:

```
fetch op code at ip
while no exception
  decode and execute by
  switch op
     case add : do ...
     ....
  ip = ip + 1
```

Each time through the loop takes some overhead. Reducing the number instruction also reduce the number of time through this loop. Hence the new instruction makes it faster.

Inc, dec

```
get lv, lit 1, add, put lv  ->  inc lv
```

and vice versa for dec. Increment uses quite often and it replaces 5 instructions which is significant saving. It is observed that the sequence:

```
ld ip, lit 1, add, st ip
```

occurs frequently and it can not be optimised to inc x because the argument is not a local variable. The sequence "lit n, add" or "lit n, sub" can be substituted by "addi n", where n can be positive and negative. The optimised codes are for performance reason, they are not absolute necessity. The markers are not executable. They are for compiler internal use to help generate the correct executable. This set is more or less a typical stack-virtual-machine instruction set. It is not too much different from JVM. JVM set is much larger with more data types. Translating S-code to JVM should be straight forward.

Sequence of transformation

1. change break to jump
scan for the matched innermost loop of break: efor, ewhile, ret, end. Patch jmp to that location. case is not a loop, break in case must break loop, or return from function call. Strictly speaking this is not an optimization. It is a usual code generation.

2. short cut the jump to jump, jmp to ret

3. ewhile to nop

4. `a = a + 1  => inc a` (if a is local) and similarly `-1` for `dec`

5. combine conditional jump
conditional: `eq, lt, le, ge, gt`
jump: `jt, jf`
to: `jeq, jlt, jle, jge, jgt`

Instructions for "for" loop

Two special instructions for implementing for loop are: ifor, efor. In the ideal case ifor and efor should have three arguments, the two local variables to do "`i <= end`" and the jump offset. However, to conform to one argument format, the trick is the use a "pair" of consecutive local variable. Therefore only one argument needed to be specified. To remove the "offset" argument from the instruction, ifor/efor v1 v2 offset is splited into two instructions. One is to ifor/efor and follows by another instruction, "jump". When ifor/efor is evaluated to be false, it skips the next instruction. This can be paraphrased as "test and jump next instruction if false". ifor does the initialisation of the index variable and calculates the end value and stores it in the "end" variable. This allows the end value to be evaluated only once. The assumption that the end value is never changed in the body of loop must hold.

```
ifor v1  (start end -- )
follows by jmp end_for
```

"ifor" uses a pair of locals, v1 v2, to store index and end values.

```
v1 = start (index)
v2 = end
```

```
efor v1 ( -- )
follows by jmp begin_for
```

"efor" uses a pair of locals, v1 v2, that store index and end values.

The instruction "ifor" takes two items from stack "start, end" and stored their values to v1 and v2. If v1 > v2 then jump out of the loop by executing the next instruction. Otherwise it skips the next instruction. The instruction "efor" is at the end of body. It increments v1, and tests if v1 <= v2 then executes the next instruction which is jmp begin_for. Otherwise, the next instruction is skipped, hence exits the for-loop.

```
  ...
  get i
  get end
  le
  jf exit
  /loop
  body
  get end
  inc-skip-lt i  // i++, if tos < i skip next
  jmp loop
  /exit
```

“efor” reduces five instructions to three instructions. If we can make "end" to be adjacent to “i” (allocating "end" will be more complex) then it will reduce to two instructions:

```
  ...
  /loop
  body
  efor i         // i++, if end < i skip next
  jmp loop
  /exit
```

These new primitives reduce the static size by another 2%, and dynamic size 8%. In terms of speed, it is also faster because the "end" is evaluated and stored in a local variable just once in the initialization by ifor, so testing "`if i <= end`" is faster than evaluate the expression every time around the loop.

Optimisation macro and/or in (Som v4.0)

Beside simple peep-hole optimisations such as:

```
not jf => jt
not jt => jf
lit.0 eqv.x jf => get.x jt   and its family
jmp.x to jmp.y => jmp.y ...
jmp.x to ret   => ret
lit.1 jt => jmp  (while 1)
```

There are a complex cascade jumps created by macro expansion of and/or. Doing a good code optimisation here improve performance significantly, for example, the 8-queen benchmark. We start the explanation with a simple case first.

```
: and a b = if a b else 0
: or a b = if a 1 else b
```

- and a b

```
a jf.1 b jmp.2 <1> lit.0 <2>
```

- and (and a b) c

```
<--- and a b ------------->
a jf.1 b jmp.2 <1> lit.0 <2> jf.3 c jmp.4 <3> lit.0 <4>
```

We recognise the pattern: `jf.1 to lit.0 jf.3 => jf.3 ...` because `lit.0 jf` always jump. We cannot do anything to `jmp.2 to jf.3`. If we move `jf.3` left then it will be incorrect when it does not jump. The ideal code is

```
a jf.1 b jf.1 c jmp.2 <1> lit.0 <2>
```

But that require the code generator to be clever. Now the more difficult case of or/or.

- or a b

```
a jf.1 lit.1 jmp.2 <1> b <2>
```

- or (or a b) c

```
<----- or a b ------------>
a jf.1 lit.1 jmp.2 <1> b <2> jf.3 lit.1 jmp.4 <3> c <4>
```

Recognising that:

```
lit.1 jmp.2 to jf.3 lit.1 jmp.4 => lit.1 jmp.4
```

This requires one look back and three look forwards. Even with different association the code sequence remains the same.

- or a (or b c)

```
          <-----  or b c --------------->
a jf.1 lit.1 jmp.2 <1> b <2> jf.3 lit.1 jmp.4 <3> c <4>
```

If the cascade is mixed of and/or.

- and (or a b) c

```
<------ or a b ----------->
a jf.1 lit.1 jmp.2 <1> b <2> jf.3 c jmp.4 <3> lit.0 <4>
```

The optimisable sequence is a difficult one.

```
lit.1 jmp.2 to jf.3 =>  jmp.3
```

Other situation of mixing does not have any new pattern. In summary, there are 3 cases:

1. cascade and: `jx to lit.0 jf.y => jx.y`
2. cascade or: `lit.1 jmp to jf lit.1 jmp.y  => lit.1 jmp.y`
3. or with other: `lit.1 jmp to jf <z>  => jmp.z`

Indentation

Som uses indentation for grouping block statements. Grouping is done in the scanner because the scanner must know about when to start and end the group. Assume the source contains no tab (converts tab to space) to simplify the implementation. The rule to recognise block-begin block-end is:

```
check the column of the first token on the new line
if col == previous  proceed as normal
   col > previous   it is block-begin, push col
   col < previous   it is block-end, pop col
                    and repeat the check to match
                    block-begin
```

The complication is in the state of lex. That lex must sometimes return with block-begin, block-end, especially when there are many block-ends. Care must be taken to synchronise the state of lex. Doing lookahead in parsing an assignment expression proves to be the source of difficulty as it backtracks lex and causes confusion on lexstate. An easy fix is do not use lex in lookahead.

lexstate

To implement using indentation as block, the lexical analyser (lex) has a Finite State Machine to control its state. A transition occurs at a call to lex. The starting state is Neutral and lex returns a token. At event newline the column position is compared to the previous start column. There are three possibilities:
1) equal, returns token,
2) more than, returns block-begin and marked this (pushing it to colstack) next state is Forward,
3) less than, returns block-end and pop the previous mark; next state is Back. block-begin and block-end are inserted by lex. The token that is scanned from the source is kept and will outputs it at the Neutral state.

In Forward, the only thing to do is to output the saved token and go to Neutral. In Back, the block-end is outputted until the matched position for block-begin is found by poping the colstack each time lex is called, then go to Neutal. At the end of file, care must be taken to output block-end to match the rest of block-begin by poping out the colstack until col == 1 each time lex is called.

When parsing an assignment statement, a lookahead for '=' is done inside the action routine instead of using lex to avoid the complex interaction with the new lex FSM when doing backtracking (saving and restoring the lex state).

Chapter 3 Virtual Machines

There are several virtual machine implementations for Som. The evolution of the design started from simplicity to performance oriented. S-code is the simplest one. Several extensions have been made to improve the performance by adding a one-address format. U-code is the design that departs from stack-based. It uses an accumulator. This allows the execution cycle of the virtual machine to be faster because of not accessing an evaluation stack. U-code also has one extension to include two-address format. The current virtual machine is a three-address format. It is a register-based instruction set, similar to modern processors.

S-code

S-code is a linear sequence of instructions that resemble machine codes. Hence it is easy to translate the S-code to an assembly language of any processor. The S-code can be executed on a stack-based virtual machine. The goal of the design of S-code is to emphasis a small number of instructions, and ease of modification. It should be reasonably fast when interpreting. A "clean" implementation is the goal, so that it is easy to modify or to make a new code generator. Essentially, S-code has only around 40 instructions. Many extensions can be experimented with easily. A fixed 32-bit instruction format is suitable. It is not the most compact form but it is easy to generate code and reasonably fast when interpreting. This format simplifies code address calculation and allows code and data segment to be the same type (32-bit integer).

Two types of instructions are: zero-argument and one-argument. The zero-argument instructions do not have any argument embedded in the instructions. They take arguments from the evaluation stack, operate and push the result back to the stack. The most frequently used zero-argument instructions are the binary operators such as add, sub. The one-argument instruction has one argument embedded in the instruction. Mostly this argument is a reference to a local variable. All control instructions (jmp, call etc) need a displacement as their arguments. The evaluation stack is implicit and automatic, that means, it can not be explicitly accessed by programmers (the stack pointer is not settable). The top-of-stack is usually cached into a register in the virtual machine to speed up the operation.

notation:
n is a 24-bit constant (2-complement)
x is a 32-bit value
v variable reference, for a global variable, it is an index to Data segement, for a local variable, it is an offset to a current activation record in Stack segment.
f is a reference to Code segment.
DS[ ] data segment, SS[ ] stack segment, CS[ ] code segment.
pc is program counter, pointed to the current instruction.
stack notation: (arg tos -- result)

zero argument instructions (arg field is 0)

s-code	description	stack effect
`add,sub,mul, div, mod`	integer arithmetic, take two operands from the stack and push the result.	(a b -- a op b)
`shl, shr`	take two operands: number, no-of-bit and shift the number and push the result back. shr is an arithmetic shift, preserved sign.	(a n -- a shift n)
`band, bor, bxor`	bit-wise, take two operands from the stack and push the result.	(a b -- a bitop b)

not	logical, take one operand and push the result.	(a -- 0/1)
eq, ne, lt, le, ge, gt	logical, take two operands from the stack and push (T/1, F/0).	(a b -- 0/1)
ldx	take an address, an index and return DS[ads+idx].	(ads idx -- DS[ads+idx])
stx	take an address, an index, a value x, and store x to DS[ads+idx].	(ads idx x --)
case	take a value (key), compare it to the range of label, goto the matched label, or goto else/exit if the key is out of range.	(key --)
array	allocate x words in Data segment, return ref v to the allocated data.	(x -- v)

one argument instructions

s-code	description	stack effect
lit n	push n	(-- n)
inc v	increment local variable, SS[fp-v]++.	(--)
dec v	decrement local variable, SS[fp-v]--.	(--)
ld v	load a global variable, push DS[v].	(-- DS[v])
st v	store a global variable, take a value x and store to DS[v] = x.	(x --)
get v	get local variable v.	(-- SS[fp-v])
put v	store a value x to local variable v, SS[fp-v] = x.	(x --)
call f	call a function, create new activation record, goto f in CS.	(args --)
callt f	tail call, use old activation record.	(args --)
ret n	return from a function call, n is the size of activation record. remove the current activation record, return a value if function returns a value.	
fun n	function header, n is the number of local variables.	
jmp n	goto pc+n in CS	
jt n	goto pc+n if top of stack != 0, pop	(0/1 --)
jf n	goto pc+n if top of stack = 0, pop	(0/1 --)
sys n	call a system function n, interface to external functions, the arguments are in the stack, the number of arguments can vary.	(args --)

Format

Each instruction is 32-bit. Right most 8-bit is the operational code. Left most 24-bit is an optional argument. This format allows simple opcode extraction by bitwise-and with a mask without shifting, but needs 8-bit right shift to extract an argument. Because zero argument instruction is more frequent, this format is fast for decoding an instruction.

Encoding

```
1  add     2  sub     3  mul     4  div     5  band
6  bor     7  bxor    8  not     9  eq      10 ne
11 lt      12 le      13 ge      14 gt      15 shl
16 shr     17 mod     18 ldx     19 stx     20 ret
21 -       22 array   23 -       24 get     25 put
26 ld      27 st      28 jmp     29 jt      30 jf
31 lit     32 call    33 callt   34 inc     35 dec
36 sys     37 case    38 fun

[- ]  reserved
```

Activation record (run-time data structure)

An activation record stored a computation state. It resides in the stack segment. The computation state consists of: pc (return address), fp (frame pointer), all locals (local var and parameters). sp (stack pointer) needs not be stored as it will be recovered when return from a function call. The “ret” instruction knows the size of activation record. The following diagram shows the layout of an activation record in the stack segment:

```
hi address

retads'     <- sp
fp'         <- fp
lv          <- lv 1
...
pv          // no. of pv, arity of func
...         <- lv n
            <- sp'', sp after return

lo address
```

A function call creates a new activation record. The new fp is sp + lv + 1. The value lv + 1 is the argument of "fun m", m = lv + 1. A local variable is indexed by an offset from the current fp. When returning, "ret n", n is the size of activation record + 1. Restoring sp by (not considering the return value yet)

```
sp'' = fp - n
```

The arity of the function can be calculated from

```
arity = n - m
```

A function call does the following. Let a be an offset to create a new frame, IP be the instruction pointer, ads be the address of the function.

```
Call:
1  SS[SP+a] = FP
2  FP = SP + a
```

```
3   SP = FP + 1
4   SS[SP] = IP + 1
5   IP = ads + 1
```

Line 1 saves the old FP at the new FP. Line 2 moves FP to the new place. Line 3 sets the new SP on top of the new frame. Line 4 saves the return address. The current instruction pointer is at the caller, therefore the return address is IP + 1. Line 5 jumps to the body of function. A tail-call (callt) does not create a new activation record. It reuses the old one. The function parameters are copied to the old activation record.

The return instruction is divided into two cases: return with a value, return without any value. To return with a value, the current top-of-stack value must be pushed to the previous evaluation stack. These two cases can be distinguished by checking whether at the time of return, SP comes back to its initial position or not (at the beginning of a new frame SP = FP + 1). Let data be the offset in the return instruction.

```
Return-with-value:
1   IP = SS[FP+1]
2   a = TOS
3   SP = FP - data + 1
4   FP = SS[FP]
5   SS[SP] = a
```

Line 1 jumps to the return address. Line 2 saves the return value. Line 3 restores SP. Line 4 restores FP to the previous value hence delete the current stack frame and moves back to the previous one. Line 5 pushes the return value to the current evaluation stack.

```
Return:
1   IP = SS[FP+1]
2   SP = FP - data
3   FP = SS[FP]
```

Return without a value is simply restoring the return address, the previous SP and FP.

case instruction

The layout of code in "case" is as follows:

```
case
lit low
lit hi
jmp else
jump table
...

code of each case
```

case does:

```
extract range of label: low, hi
if key < low or key > hi
  pc = pc + 3        // goto else-case
else
  pc = pc+key-low+4  // goto matched label
```

In this implementation, the jump-table is fully-filled with the labels in the range. Finding the matched label is simply an index calculation, a constant time operation. This enables "case" to be fast but it consumes the memory in the code segment as large as the range of label. This is wasteful if the label is not densed. If the label is sparse, a binary search can be used. The jump-table is the sorted label of the pair (label, goto code). This is not used in this virtual machine.

S-code virtual machine

The main loop is simply a decode-execute cycle using a "switch( )". Let IP be the current instruction pointer.

```
Eval:
  while(runflag )
    opcode = CS[IP] & 255
    data = CS[IP] >> 8
    IP++
    switch(opcode)
      case Add: ...
```

The run-time data structure includes: a memory M[.], a stack-segment SS[.], a code segment (it is relocatable), a data segment (it is absolute). The code and data segment are in M[.]. The memory map is as follows (from lo mem to hi mem):
system area
data segment
code segment

The system area contains some values used in communicating between system functions in the compiler and the virtual machine. The stack segment is a separate data structure from the memory. The stack segment contains a run-time data structure called "stack frame" used in a function call and the evaluation stack. A pointer, FP, points to the current stack frame. A pointer, SP, points to the evaluation stack. A stack grows from lo to hi address. The structure of a stack frame is as follows:

```
hi


        <-  SP
retads
FP'     <-  FP
lv_1
...
lv_n

lo
```

The evaluation stack is "on-top" (higher address) of the stack frame. To pass parameters from the current context to a function, the new stack frame is "overlapped" with the evaluation stack. The new evaluation stack is then started at an address "after" the return address in the new stack frame. The current top-of-stack is at:

```
SS[SP]
```

To push a value to the evaluation stack requires:

```
SP++
SS[SP] = x
```

To pop a value from the evaluation stack is the reverse of push:

```
x = SS[SP]
SP--
```

To access a local variable in the current stack frame, a negative offset (the reference of a local variable) is used relative to FP. For example the first local variable is at SS[FP-1], the second SS[FP-2] etc. It is not necessary to save SP as the "ret" instruction contains a proper offset to restore SP back to a previous context. Similarly, the "fun" instruction contains a proper offset to build a new stack frame.

<to explain how to eval from Som language>

Sx-code

Sx-code is an extension of S-code to have additional one address. The aim is to improve the execution speed of the interpreter. As one-address will reduce the number of instruction by 30-40%, it should be faster than S-code. The decoding of zero+one address is exactly the same as zero-address s-code as the instruction has two fields: op, arg. In a sense, we get the one-address for "free". If the VM is as fast as som-v16 then by reducing the number of executed instruction by 30-40%, the new VM will be faster (even faster than Som v 1.7 which uses T-code). Chronologically Sx-code is designed after T-code. Sx-code is used in Som series 3. It is one of the largest instruction set in terms of the number of instructions.

All binary operators are extended to have one-address to access local frame, therefore in many cases the sequence "get.x get.y bop" becomes "get.x bop.y". The immediate mode stored a literal in the argument of the instruction. The sequence "get.x lit.y bop" becomes "get.x bopi.y". To blend one-address into zero-address, arg = 0 is used to indicate the top of stack addressing.

The load/store index, are extended to store the base-address in the argument. The sequence "get.base get.index ldx" becomes "get.index ldx.base". When base is global, a new instruction "ldy.base" is used. The order of argument for store index is different from S-code. The sequence "get.base get.index get.val stx" becomes "get.index get.val stx.base". When the base is global, "sty.base" is used. There is no use for the old "ldx/stx" (zero-address) as there is always the base address in either local or global. To optimise the for-loop, "efor" instruction is introduced. "efor.x" does the following:

```
 x++, push(x <= adj(x))
```

where x is a local, adj(x) stored the terminal value of x. The sequence at the end of for-loop is usually "inc.x get.x get.end le jt.loop" becomes "efor.x jt.loop" where adj(x) is end. The compiler must allocate adj(x) accordingly.

Encoding

Arrange the instruction so that grouping is easy.

```
bop-zero-arg:  add..shr  (1..16) (17..20 reserved)
bop-one-arg-v: add+20   (21..36) (37..40 reserved)
bop-one-arg-i: add+40   (41..56) (57..60 reserved)
other:         get..calli  (61..82)
zero-arg:      not case end (83..85)

bop is   add sub mul div band bor bxor mod
    eq ne lt le ge gt shl shr
other is get put ld st ldx stx ldy sty
    jmp jt jf call ret - efor
    inc dec lit ads sys fun calli
    not case end
```

The instructions "fun" and "calli" are not executable. They are markers in code segment.

U-code

The aim of this design is to make a compact instruction set that is fast and has a clean semantic. Toward these goals, the instruction set has less than 50 instructions. It is a one-address format with a few two-address, based on using an accumulator. The encoding is fully decoded to allow fast virtual machine execution without decoding an instruction. The following description is for the version u2-code.

U-code instruction set

```
bop     :  add sub mul div band bor bxor mod
           eq ne lt le gt ge shl shr
bim     :  addi subi bandi bori eqi lti lei shli shri
data    :  ld st get put lit
vector :  ldx/ stx/ ldy/ sty/ ldxa ldya
control:  jmp jt jf jle/ case call callt ret
extra   :  fun/ sys inc dec not push
```

Total 50 instructions. The suffix / indicates the instruction with two arguments.

The `ldxa ldya` can shorten "putting" the result into a temporary register (called cascading). Not all the immediate mode is included. That will make the instruction set too fat. The logical one: `nei, gti, gei` can be emulate by the inverse. The others are rarely used: `muli divi xori modi. shl` and `shr` are added to make the instruction set complete.

Format

```
 one-address  op:32  arg:32
 two-address  op:32  a:24,b:8
```

Semantic

operators

```
v is M[fp+v]

bop v       ::  AC = AC op v
bim a       ::  AC = AC op a
```

data

```
lit a       ::  AC = a
ld a        ::  AC = M[a]
st a        ::  M[a] = AC
get v       ::  AC = v
put v       ::  v = AC
```

vector

```
ldy a,v     ::  AC = M[M[a]+v]
sty a,v     ::  M[M[a]+v] = AC
ldx v1,v2   ::  AC = M[v1+v2]
stx v1,v2   ::  M[v1+v2] = AC
ldxa v      ::  AC = M[v+AC]
ldya a      ::  AC = M[M[a]+AC]
```

control

```
jt a        ::  if AC != 0 pc = a
jf a        ::  if AC == 0 pc = a
jle a,v     ::  if AC <= v pc = a
call a      ::  if arity(a) > 0 pusha passing the last
parameter if any
                create a new activation record
ret size    ::  delete current activation record
case lo; lit hi ::  if lo <= AC <= hi skip 4+2*(AC-lo)
```

extra

```
inc v       ::  v++, AC = v
dec v       ::  v--, AC = v
push v      ::  sp++, if(v==0) M[sp]=AC else M[sp]=v
fun arity,size    a place holder for arity,size
```

Usage of for and case

for loop
```
...
inc i
jle v loop
```

case
```
<index in AC>
case lo
lit hi
jmp else
jmp case1
...
jmp casen
```

Instruction encoding

```
1   add sub mul div band bor bxor mod eq
10  ne lt le gt ge shl shr addi subi bandi
20  bori eqi lti lei shli shri inc dec lit sys
30  not push get put ld st ldxa ldya call ret
40  callt jmp jt jf case ldx/ stx/ ldy/ sty/ jle/
50  fun/
```

Two-addressing allows the base address to be specified in vector instructions. Two fields are also used in "`jle`" which is the variation of "`efor`". It is simpler but need a modified "`inc v`" to work. It does not required to "decrement" the initial index and there is no "hidden" adjacent variable. "`case`" is a long process of refinement. I think I got a good compromise here. Using AC for "hi" value and keep "`jmp`" instruction in the jump table. This design trades off the size of the code for simplicity in semantic. The previous form is of "`case`" (as shown below) is correct in sense of being one instruction without any continuation so its semantic is clear. VM does not behave differently from executing other instruction. VM does not fetch any argument from the next instruction.

```
lit hi
case lo,v
jmp else
jmp case1
...
jmp casen
```

The current instruction is faster but its semantic is not consistent with the rest of the instruction set. It fetches the next instruction to be used in deciding the transfer of control. So, the question is what do we prefer, clean semantic or performance?

Activation record

U-code stack frame has "positive" order of local variables (v is `M[fp+v]`) unlike S-code. The "usual" (s-code style) has (v is `M[fp-v]`) backward order and required renaming of local variables. The rename process scans the code after the code body is completely generated as there may be some additional local variable allocated during the code generation. If the order is forward then renaming is not necessary.

```
  hi

...      <- sp
retads
fp'
vn
...
v1
         <- fp
   lo
```

To know where fp' and retads are, the size of the activation record must be known. It is record as the argument of the "*ret*" instruction. To create a new activation record, two arguments are used: arity, and the number of local variables. They are recorded as two arguments of the "*fun*" instruction.

Parameter passing

Passing parameters to a function requires a special treatment. The space in stack segment, SS[.] (where it stores the activation records), is used as a "virtual stack" to pass parameters to a new frame. A new instruction is created for this task: push. It pushes parameters to a virtual stack. A tail-call instruction (`callt`) is revived as it is appropriate. It is far simpler than trying to generate codes to pass parameters back to the old frame and do a jump. Callt is faster too (it is another "big" instruction according to our philosophy of trying to create big instruction. A "big" instruction does more in one instruction). The last parameter is passed through AC, occasionally saving one push instruction.

T-code

T-code aims for performance. S-code is a stack-based instruction set, aimed for simplicity and tends toward minimalist. In contrast, T-code is a register-based instruction set and has a richer set of operations. From the experience of designing various instruction set for real chips using mostly stack-based instruction set, many designs have fewer number of instruction executed than S-code. For example, the one-address, or the stack mixed with register (aka register window), reduces the number of instruction executed by almost half compared to pure stack-based. This observation supports the argument that using register-based instruction set improves performance. In implementing a virtual machine, the main instruction dispatch has a high cost. Therefore reducing the number of instruction executed helps reducing this cost. Another benefit is that executing each instruction for a register-based instruction may be faster due to the ability to access many arguments in one instruction. However, the cost of decoding multiple fields in an instruction may be higher than stack-based.

Format

Almost all instructions have three-address. An exception is "mov" instruction that has two-address. 64-bit is quite a natural size for modern processors. There is no distinction between global/local/immediate in the instruction format. The format is very uniform. (This report is for t2-64-code used in Som v 5.1. Another version of T2-code uses 96-bit format in Som v 5.0.)

3-arg *a:16 c:10 op:6, b:32*
2-arg *d:26 op:6, b:32*

Instruction Set

```
bop:      add sub mul div mod and or xor shl shr eq ne lt
le gt ge
control:  jmp jt jf jeq jne jlt jle jgt jge fun call
callt ret efor case
data:     ldx stx mov push
etc:      not sys
```

There are 38 instructions, so op code field is 6 bits. That leaves 26 bits to be divided between two arguments. One argument should be a bit large because it is used as displacement in conditional jump instructions, so it is divided into 16-bit and 10-bit. The third argument, “b”, is 32 bits so it does not need decoding.

```
bop  dest (c), src1(a), src2 (b)      ;;  c = a op b
ldx  dest (c), idx (a), base (b)
stx  src  (c), idx (a), base (b)
not  dest (c), src (a)

jmp  disp (a)
jt   disp (a), src (c)
jop  disp (a), src1 (c), src2 (b)
efor disp (a), src+ (c), src2 (b)
```

disp of jump is 16 bits so use (a).

```
fun  arty (a), fs (c)
ret  src  (a), fs (c)
push src  (a)
```

To reduce the chance of argument not fit into the field, we use the larger field first (as a priority). Therefore the instruction "call" uses the field "a" then "c". The instruction "sys" uses the field "b" then "a".

```
call arg2 (c), arg1 (a), ads  (b)
sys  num  (c), src2 (a), src1 (b)
case src  (c), lo   (a), hi   (b)
```

The instruction “mov” is an exception. It is two-address and hence has the largest dest/src field. This allows it to access a large address space to move values around.

2-arg *d:26 op:6, b:32*

```
mov  dest (d), src (b)
```

Semantic

```
bop d a b      ==   M[d] = M[a] bop M[b]
jmp d          ==   goto d
jt d a         ==   if a != FALSE goto d
jf d a         ==   if a == FALSE goto d
jxx d a b      ==   if (a op b) != FALSE goto d
ldx d a b      ==   M[d] = M[ M[a] + M[b] ]
stx d a b      ==   M[ M[a] + M[b] ] = M[d]
call d a ads   ==
                   get arity and numlocal (framesize)
                   1. move parameters to temp (param: d,a
and from stack)
                   2. save locals (numlocal), pc,fp  (at
stack)
                   3. update fp, sp
                   4. move temp to locals (arity)
ret d fs       ==
                   1. move return value d to M[retval]
                   2. restore locals , pc,sp,fp
callt d a ads ==
                   1. move parameters to temp
                   2. move temp to locals
efor d a b     ==  M[a]++; if M[a] <= M[b] goto d
case d lo hi   ==
                   jump table is set of displacements
                   size of table is hi-lo+1
                   if lo <= d <= hi goto entry[lo-d+1]
                      else          goto end of table
                   where current pc is at the "case"
not d a        ==  M[d] = not M[a]
sys d a b      ==  syscall d, optional param: a,b
push d         ==  sp++; M[sp] = M[d]
mov d a        ==  M[d] = M[a]  equiv. to  add d a #0
fun d a        ==  it stores arity (d), and numlocal (a)
                   can be encoded into 24 bits
```

case instruction

```
case src lo hi
<jmp table>
$end
```

The jump table is a table of displacement relative to the address of "case" instruction. Each entry is a 32-bit value. The size of the table will be padded at the end so that the address $end aligns at an even address. Its organisation is as follows:

```
disp. to the end of table, the "else" case
disp. to case lo   (case_1)
disp. to case lo+1 (case_2)
....
disp. to case hi   (case_n)
<pad>
$end:

jmp else_case
case_1:  ... , jmp exit
case_2:  ... , jmp exit
...
case_n:  ... , jmp exit
else_case: ...
$exit
```

The first entry in the table is the displacement to $end. It is used to jump to else_case. This arrangement makes it easy to locate the end of the table. The jump table is fully mapped to values in the range lo..hi. It is a direct map. Any missing label will be filled with a displacement to $end (so it goes to else_case). The size of table is even(hi-lo+2). The code for body of each case is located after the end of table. The first instruction at the end of table is "jmp else_case". Each case is ended with "jmp exit". The jump table makes "case" instruction a strange object of variable length in the code segment. If instead of a displacement, an instruction "jmp" is used in each entry in the table, the size of the table will be quite large (2 words per entry).

Encoding

```
0..9   nop add sub mul div mod and or xor eq
10..19 ne lt le gt ge shl shr not mov ldx
20..29 stx ud push call callt fun ret efor case jmp
30..38 jt jf jeq jne jlt jle jgt jge sys
```

Activation record

```
hi

        <- sp
 .. param
--------
 fp'    <- fp
 retads
  lv    v_n
  ..
  pv    v_1
--------
        <- sp'
lo
```

The current frame stores: fp', retads. There is no need to store sp as it tracks fp when return. The number of slots, pv+lv, are the saved registers. Now that a local is in an absolute place (M[0]...M[256]), no renaming of registers during compilation is necessary.

Parameter passing

Two parameters can be passed via "call" instruction. If there are more than two parameters, the rest are pushed to stack (via SS[sp]). Inside v5-vm, an array param[.] is used to collect actual parameters to be instantiated to registers.

Constants as globals

To reduce the number of distinct instruction, the "immediate" mode can be eliminated by using constants as globals. It means all constants will be stored as global variables. For small constants, -10..300, they can be stored permanently (immutable) in M[a]..M[b]. There is a direct relation between the value and its address. For larger constants, they will be allocated and stored as globals using the symbol table. This way, the access to symbol table will not be overwhelm as I expect 90% of constants will be small and not require symbol table to retrieve them during the compilation.

What should be m and n? -1 is used often, 256 seems to be the largest small number (as least in our benchmark). So the range -10..300 should cover most small numbers without being too many. The address 300 is used to signify a constant so a > 300. Let a = 390, b will be 700.   -10..300 is M[390]..M[700] or c is represented by M[400+c].   For large constants, they are stored in the data segment starts at the address 1000 (the same as all other globals allocated by the compiler).

System Calls

To decouple the language from specific system functions (I/O and files), these system functions are made into one instruction "sys" with numeric parameter to specify a number of functions. Not all versions of Som support all of these calls. This is the most current version.

```
1 print x
2 printc c
3 getchar
4 gets
5 fopen name mode
6 fclose fp
7 fprint fp x
8 fprintc fp c
9 fgetc fp
10 fgets fp buf
11 --
12 eval x
13 stop
14 alloc x
15 load name ; exec from user
16 lex fc    ; return tok from the current file
```

More macros are defined in lib2.som:

```
: print x = syscall {1 x}
: printc x = syscall {2 x}
: getc = syscall {3}
: gets buf = syscall {4 buf}

// mode 0-read, 1-write, 2-readwrite
: fopen fn mode = syscall {5 fn mode}
: fclose f = syscall {6 f}
: fprint f x = syscall {7 f x}
: fprintc f  x = syscall {8 f x}
: fgetc f = syscall {9 f}
: fgets f buf = syscall {10 f buf}

// for som-compiler
: eval a = syscall {12 a}
: exit = syscall {13}
: load fn = syscall {15 fn}
: syslex f = syscall {16 f}   // in som 4.2
```

Object File Format

Som v 1.7 Format

```
start end
code* (in hex)
start end
data* (in decimal)
num  (number of entries of symbol table)
symbol table*
```

Each symbol table entry is : `name type ref arg arg2`

What is stored in the object file?

The object file stored code, data and exported symbol table. The code (v 1.7 uses T-code instructions) is in hex (for fast load as it is 32-bit). The data is in decimal. The data is the snapshot of memory when finished compiling and executing the immediate lines. The amount of data is dictated by the amount that has been dynamically allocated when compile the program. The symbol table is important for initialising the global variables. This is a change from previous versions which relied on the code generator to generate a "replay" of the immediate line so it is not necessary to store the data in the object file as it can be recreated. However, having snapshot is useful in many situations, such as the static array which is introduced in this release.

Consider the symbol table. What information must be recorded to allow CS, DS to be relocatable? For CS, start address, there is only one kind of object, code. For DS, there are several kind of objects: global variables, constant static array etc. As the snapshot is a contiguous block, it can be relocated as a whole. However, if a variable contains a pointer to DS, it must be noted so that its value can be relocated. These pointers are:

1) base-address of static array. A static array is an array that is allocated at compile-time, hence its base-address is known at compile-time. This happens when define an array by an immediate line (outside function definition), such as

```
 a = array 10
  b = array {11 22 33}
```

2) a pointer to static string. A static string is a string created at compile-time such as a string embedded in the source code or a string created by an immediate line that its value is assigned to a global variable, such as

```
 to warn = { prints "warning message" }
  s = "this is a string"
```

Aliasing of variables will not be analysed hence they will not be relocated. Care must be taken in using such variables because interactive-mode and run-only-mode may behave differently when DS is relocated. To denote the kind of global variable: scalar, static array, string pointer; the field "Arg" in the symbol table is used:

```
0  scalar
1  static array
2  string pointer
```

This information will be used in the loader to relocate DS.

Som v 2.4

The object file is the same format as som-v2 except:
1) magic cookie is 5678916 (som16 object)
2) it includes symbol table with the form: (name type ref arity lv)

Format

```
magic      (5678916  for som v24)
start end  (code segment)
code*
start end   (data segment)
data*
num  (number of entries of symbol table)
symbol table*
```

Each symbol table entry is: `name type ref arg arg2`

Som v 3.1

```
magic                             5678931
start end (op arg)*               code segment
start end data*                    data segment
size  (name type ref arity lv)*   symbol table
```

Som v 4.0, 4.1, 4.2

Similar to v3.1 except magic are 5678940, 5678941, 5678941 (yes 4.2 uses 5678941)

Som v 5.0, 5.1

A little change has been made in the instruction format in the code segment (to make it easy to locate individual instruction). One additional object in the file is Link vectors. They make loading object file faster because of not scanning the symbol table. Link vectors are special locations, in v5.0 there are: addresses "loadfile" and "CS". If there is no link vectors (size = 0) the loader will scan the symbol table for the required information. This link vectors are not produced by the compiler. They are manually edited into the object file because the compiler is not specific to any source, hence it does not know about link vectors of a particular program. Som v 5.1 has magic cookie 5678951.

```
magic                                 5678950
start end (ads op arg1 arg2 arg3)*   code segment
start end data*                       data segment
size (vector)*                        link vectors
size (name type ref arity lv)*        symbol table
```

Chapter 4 Happy Birthday Som

August 8th is marked as Som's birthday. It has been good many years (almost five years) that Som language and system was continuingly developed. It is a good time to look back and contemplate what has been accomplished. This chapter describes a brief history of the development of the language and presents performance measurement of all versions (up to v 4.1) of the implementations.

History

4 Dec 2004	som-v1	first public release
31 Dec 2004	som-v2	second public release, with som-in-som
26 June 2005	som-v1.5	with macro and tail-call
5 Jan 2006	som-v1.7	with new VM, T-code
23 Dec 2006	som-v1.8	bug fixed v 1.7, T-code
12 Jan 2007	som-v2.4	(som-in-som for 2007) Children-day release
9 Mar 2007	som-v3	som-in-som with sx-code vm
19 Aug 2007	som-v3.1	fast sx-code vm
2 July 2008	som-v4.0	fast u-code vm
9 Aug 2008	som-v4.1	(improve u-code and compiler) Birthday release

Som project started her life in late 2003. She is based on my earlier work on many language interpreters. The basis is the stack-oriented instruction set, the very simple S-code. The whole 2004 is the developmental year to get the code base to crystallise.

The first public release is on 4 December 2004. Som v.1.0 is all written in C. It took Som v2.0 so that everything is written in Som herself. The next year, 2005, we saw the development of macro (v.1.5), code optimisation, constant array and file i/o. These improvements are included in the next release (v.1.7) with the new instruction set, T-code on the New Year day of 2006. With the complex t-code, Som v.2.3 which attempts to use T-code as her instruction set, is never complete. The bug fixed for v.1.7 is released by the end of the year (v.1.8). The year 2007 is the update of Som v.2.4 that brings all the updates into Som written in Som. This year also is the year of improving the execution speed of the virtual machine. Many instruction set formats have been experimented with. Som v.3.0 with the extended S-code (Sx-code) is released in March. The fully decoded instruction format is released as v.3.1 in August. This version employs a lot of improvement to make the virtual machine as fast as possible.

Due to my health reason, the development was stopped for six months. In mid 2008, a new refinement is released, Som v.4.0. It uses a stack-less instruction set, U-code. It is a simplification of Sx-code. In 2008, Som v.4.1 is released. It is a gentle refinement of U-code and a lot of improvement in the compiler. To celebrate the birthday, I did benchmarking all versions to record the development is a quantitative term.

Benchmark

The benchmark programs are chosen so that all versions can compile and run them. They are:

1. **bubble** sorts 20 items from 20..1 to 1..20.
2. **matmul** performs 8x8 matrix multiply.
3. **queen** solves all solutions of 8-queen problem.
4. **queen2** incorporate macro (Som v.1 and Som v.2 do not have macro).
5. **quick** sorts 100 items 100..1 to 1..100.

These benchmarks indicate the performance of the instruction set (measuring the number of instruction executed) of various format (S-code, T-code, Sx-code and U-code) plus the quality of the code generators. The running time measured the performance of the virtual machines. Another measurement is performed on the compiler. The source of the compiler of Som v.2.0 is used (around 2000 lines of code). All modules are concatenated into one file to be the input of the compiler. This benchmark indicates the performance of the compilers. How fast it is to compile one program. For the compiler benchmark, the performance is relative to Som v.2.0.

The number of instruction executed (noi) is a reliable metric because it is not dependent on the machine that runs the benchmark. But noi alone can not compare the quality of the implementation of the virtual machines. The running time is tricky to collect and highly variable. The results are presented as a relative measure, or the speedup, calculated by (1 - t2/t1) * 100 where t1 is the running time of Som v.1.0 and t2 is the version to compare with it (v1.0 is supposed to be the slowest one). The running times are measured using the time function in C , time.h and clock( ), instrumented into the virtual machine. The program is run 3 times and the data are averaged. The machine used to run all benchmarks is Dell D500, a laptop with Pentium M 1.3GHz and 1Gbytes of memory running Windows XP (SP2).

Results

General benchmarks

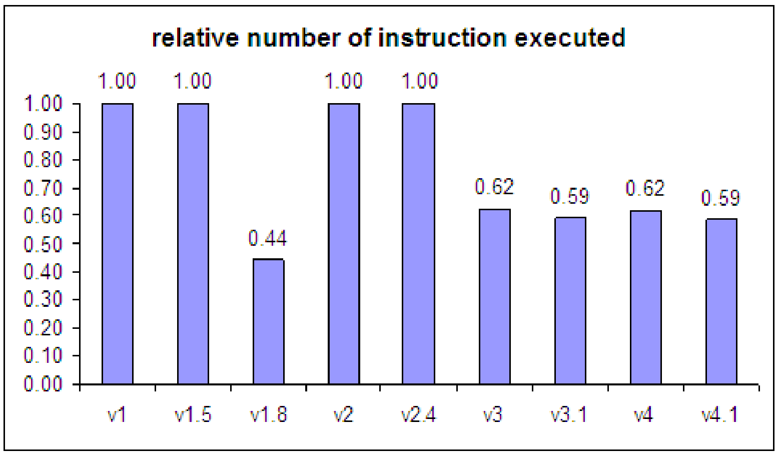

Figure 1 General benchmark: the relative number of instruction executed compared to som v.1.0 *(vx/v1)*

The number of instruction executed (noi) of v.1.0, v.1.5, v.2.0 and v.2.4 are the same. They are S-code. The T-code of v.1.8 is less than half, so T-code is very effective. The Sx-code is also fast, its noi from v.3.0 is only 62% and v.3.1 is even better at 59%. The serie 4, U-code, is similarly effective compared to Sx-code.

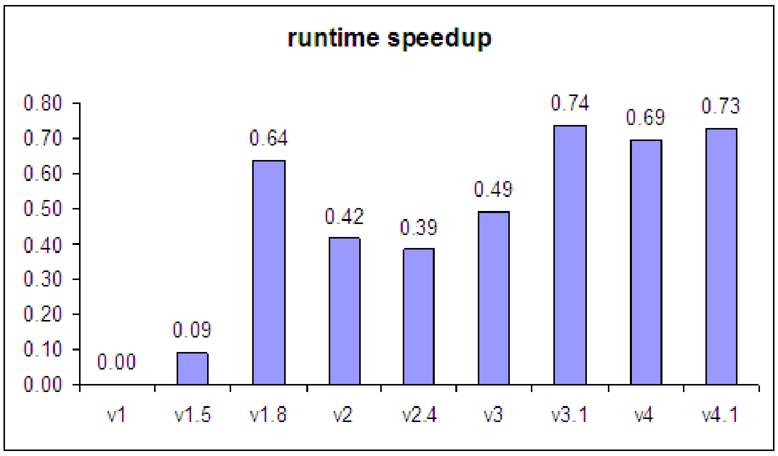

Figure 2 General benchmark: the speeup of running time relative to Som v.1.0 *(1-vx/v1)*(0.73 is around 4x)

When look at the running time, T-code is 64% faster than v.1.0 (around 2.7x). The virtual machine of series 2, v.2.0 and v.2.4 (two vm are the same), are fast. They are around 40% faster than v.1.0, or 1.6x. The series 3 Sx-code have impressive results. The virtual machine of v.3.0 is 49% faster, or almost 2x. The new virtual machine for v.3.1 (with separate op and arg and top-of-stack register) is the fastest. It is 74% faster, or 3.8x. The series 4, the virtual machine of U-code v.4.0, is 69% faster and the refined U-code of v.4.1 is as fast as the best v.3.1 at 73%, or 3.8x.

Compiler benchmark

First, the size (measuring in line of codes) of the compilers and the virtual machines are compared. These figures indicate the complexity of the programs. The v.1.8 compiler is the largest at 3700 lines whereas the recent ones (v.4 and v4.1) are at around 2500 lines. Perhaps this reflects the complexity of T-code versus Sx-code. The sizes of virtual machines also have this trend but they are less different.

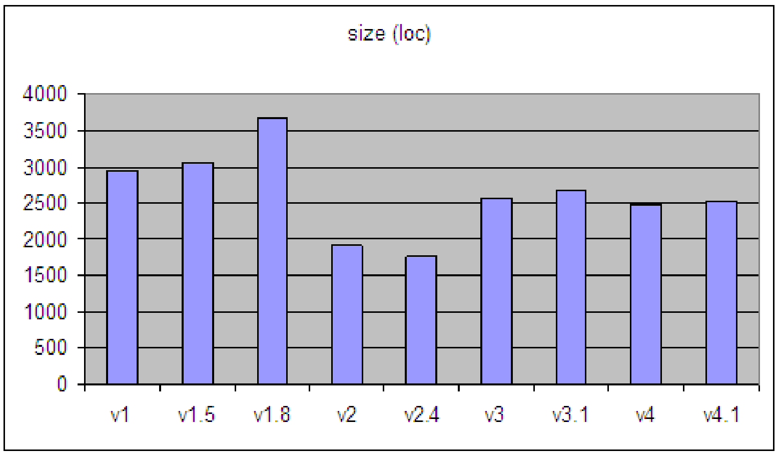

Figure 3 The size (lines of code) of the compilers

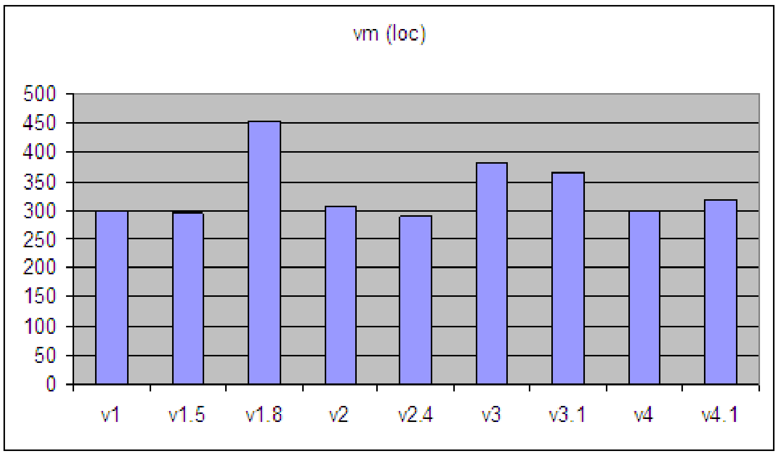

Figure 4 The size (line of code) of the virtual machines

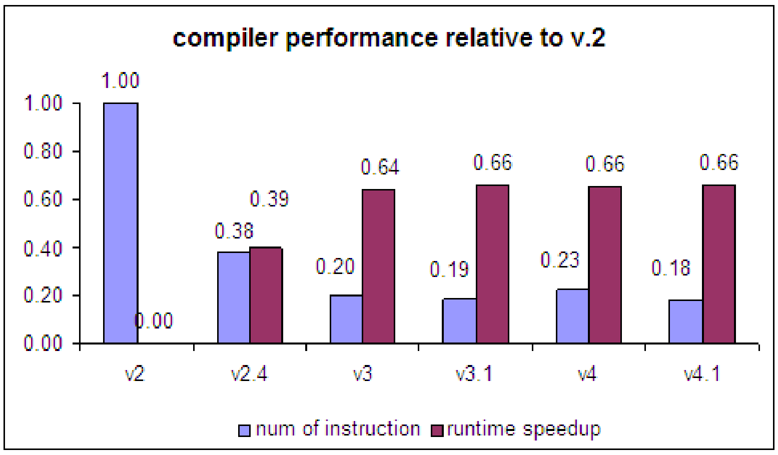

Figure 5 Compiler benchmark: the compiler performance relative to Som v.2.0, the number of instruction executed and the speedup. noi is vx/v2. speedup is 1-vx/v2 (0.66 is around 3x)

In terms of noi (the number of instruction executed to compile the program), the newer compilers are better with v.4.1 is only 18% of v.2.0 (it is very impressive, just 1/5). A lot of code improvement has been done on these compilers. The runtime speedup is also reflected these improvements, with v.4.1 at 66% faster, or almost 3x. The compiler does a lot of i/o so in term of the performance of a virtual machine, it may be better than this figure.

To give some absolute number on the performance of the virtual machine, the runtime for general benchmark is reported as Million instructions per second (noi/running_time). The machine that runs the benchmark is Dell D500 laptop with 1.3GHz Pentium M (single cpu) with 1Gbyte memory running Windows XP (SP2). The compiler is lcc-win32 (version Oct 2007) with no code optimisation. These figures reflected the effect of the instruction set and the implementation of its virtual machine.

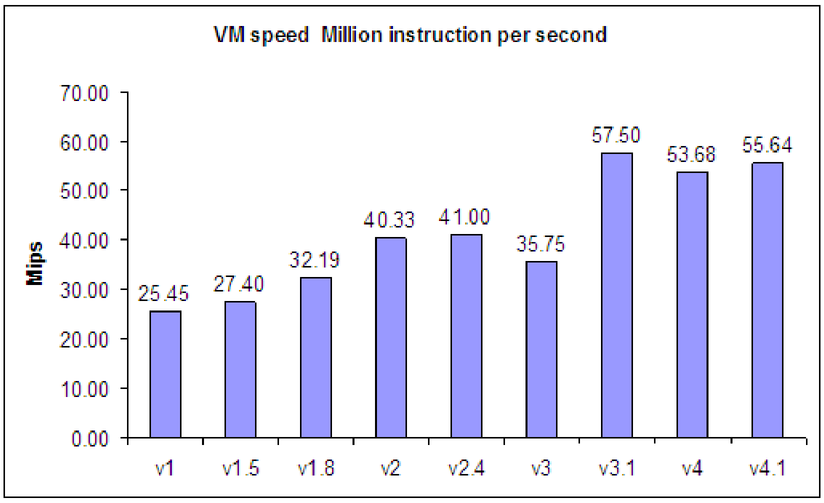

Figure 6 The performance of virtual machines (Million instruction per second)

The jump from v.3 to v.3.1 (35.7 Mips to 57.5) is the result of the engineering of the virtual machine. It is interesting to note that the series 4 (v4 and v4.1), which employs U-code, the figure is not as good as v.3.1 (Sx-code). However, in terms of running time, they are similar. It may be just the variance in the time measurement.

Chapter 5 History

This chapter is a record of the development history of all versions of Som. The description is ordered chronologically.

Release history

4 Dec 2004	som-v1	first public release
31 Dec 2004	som-v2	second public release (with som-in-som)
26 June 2005	som-v1.5	with macro and tail-call
5 Jan 2006	som-v1.7	with new VM, T-code
23 Dec 2006	som-v1.8	bug fixed v 1.7, T-code
12 Jan 2007	som-v2.4	Children-day release (som-in-som for 2007)
9 Mar 2007	som-v3	som-in-som with sx-code vm
19 Aug 2007	som-v3.1	fast sx-code vm
2 July 2008	som-v4.0	fast u-code vm
9 Aug 2008	som-v4.1	Birthday release (improve u-code and compiler)
9 Sept 2009	som-v4.2	Triple 9 release (lex in vm)
22 Sept 2009	som-v4.2a	bug fix interactive mode (new parser)
5 Dec 2009	som-v5	T2-code vm (Long Live the King)
25 Dec 2010	som-v5.1	T2-code 64-bit format (Christmas release)

som0 (first integration of som)

13 September 2003
This implementation aims to separate compiling into an executable sequential code for a virtual machine and generating machine code for a machine specific processor. For an executable sequential code, the emphasis is on a small number of instructions and ease of modifying this set. The code for Som which will be called s-code, should be reasonably fast when interpreting, and easy to generate machine dependent code. Therefore, the optimisation should not be emphasised. Instead, a "clean" implementation is the goal, so that, it is easy to modify or to make a new code generator. A fixed 32-bit instruction is suitable (not compact but easy to generate code and reasonably fast when interpreting). The instruction set is of A1a (an earlier language) but eliminates some extended code.

som-som (start of som-in-som)

26 Decenber 2003
The goal is to start a boostrapable system that eventually will be written in Som itself. The starting system is written in C and gradually it is replaced by Som. The system is divided into three parts: a parser, a code generator, a virtual machine. The

parser transforms a source (Som program) into an abstract program (parse tree). The code generator generates S-code from this parse tree. The virtual machine executes S-code. This division makes the system easy to understand. The parser separates the handle of two domains, one is the character domain represented the source program, and the other is the token (integer) domain represented the abstract program. The token is suitable for Som-language. It makes the code generator and virtual machine easy to be written in Som. The virtual machine has been written in A1a (the precursor of Som). The parser and code generator are integrated in the current system (Som, Som1). Separating them into two parts enables the code generator to be written in Som easily. It may not be interesting to write the parser in Som, as there are tools for parser generators, which accept the input as a grammar and output a parser in some high level language program. It is possible to translate the output from these tools into Som language.

29 Jan 2004 Implement parser generator to generate parse.som
12 Feb 2004 Add tuple to som-language to handle variable arguments to a function. Use it for syscall. (this design eliminate the open stack coding)

som-v1

4 Dec 2004
This is a reference system for Som-language. It is som3 (the development version) released to public. The major change in som3 is that it can do recursive load and has a good interactive mode. Effort will be made to document this release so it is understandable for the public (mainly students and my research assistances).

som-v2

31 Dec 2004
This version is an evolution of Som system to be self-replicating. That is the system can generate itself with minimum support from a host language. This is achieved by writing most of Som system (lexical analyser, compiler, code generator) in Som. Only the eval function must be written in the host language (C). This approach has been used to port a compiler to a new platform since the early days of computer science, for example, Pascal compiler. However, the interesting point is not "porting" a system to another platform. It is the ability to "self-replicate" that I try to achieve.

This Som system comprises of the whole Som in object format (som-code, "som.obj") that is loaded into the memory and is executed by eval() (written in C). This image plus eval-in-C works as Som compiler. The first image "som.obj" is generated by Som itself. However, this is not done in this release. The present "som.obj" is generated from a modified Som v1. There are minor differences that are needed to be resolved before it is truely self-replicate.

som-v1.5

26 June 2005
The aim is to improve performance when it goes to a real chip. This version contains macro and proper tail-call. Modify Som v1 to include macro, a bit of jump

improvement (jle for "for") and perhaps hash from symtab5 (from som-in-som, som-v2). (23 Feb 2005)
1 macro
2 jle
3 eliminate callt
4 retv
5 hash symtab

som v 1.7 T-code

23 December 2005
It is going to be new year soon!   I have a great new idea on er... a new VM for som, called T-code. T-code will be a register based VM as opposed to S-code. As T-code is 3-address format, it will have less number of instruction executed (dynamic instruction count) than s-code. The data from various chip designs pointed to 40% noi. of s-code [aisd eecon 2003, sr, compact code jcsse 2005, xs]. This means if all else is equal, T-code vm will be 2.5 times faster than s-code. Therefore, I will try to have a new year release of Som-v17 with T-code (a major release every year, eh?). Som-v17 will be a som-v16 (which never made public): macro, static array, new object, no immediate line, hex, file.

som v 1.8

23 December 2006
This is a bug-fixed version of som-v17. Most bugs are in the code generator. "gencode.c" has been heavily rewritten. There is still some code sequence that is not optimised but it is correct. It has been tested and passed all benchmarks in "test" directory.   The macro has been fully debugged as well, especially the "full" macro. This version is used to develop som-v23 (som-in-som in progress). som-v23 can compile and generate code correctly including all macros. The "eval" is being developed. So, som-v18 has been subject to extensive test to run som-v23 (around 2500 lines of som-code) except the "eval" part. som18 can do "load".

som v 2.4   Children-day release

12 January 2007
It is a som-in-som interpreter/compiler system. It used som v 1.6 as a development platform. som16 is an updated of som v 1.5 toward som17 to have "loadfile". The idea here is to release som v 2.4 as an up-to-date, stable version of Som for year 2007. The goals for this version are:
1) use simple s-code.
2) improve interp.c for faster speed.
3) has the following (of som16) : macro, static array, object with no immediate line, hex, file, and from som17, loadfile.
4) make som-in-som as clean as possible.

Why not t-code? From som17, I found that a lot of development time is spending on subtle bugs in code generator. However, the speed improvement (in term of execution time) comes mostly from engineering the interpreter not the t-code itself. T-code which has 40% number of instruction does not translate into 2.5 times faster in term of execution time (it is only 25% faster). So, the complexity of t-code does not worth it. som v 1.8 is sort of closing down the experiment on t-code.

"som.obj" is self-compiled, that is the som.txt is compiled into its own object. Finally, the self-replicate property has been achieved in this release. In terms of speed, som-v24 virtual machine has been engineered to be as fast as som-v17.

som v 3.0 Sx-code

9 March 2007
This is a new release based on sx-code. Sx-code is zero+one-address instruction set. It is 30% faster in terms of number of instruction and running time than s-code. It is not complex, only larger than s-code (85 instruction vs 40). The added instructions are local-var mode and immediate mode of bop s-code pluses some "for performance" code such as efor. The virtual machine of som-v16u is used as VM for som-v3. This VM has been carefully engineered. It is the fastest VM for s-code family to date.

som v 3.1

19 August 2007
It is a continuing development of Som v3 with an improved vm. The new vm had fully-decode opcode and argument. It also employs a few techniques to speed up the execution of sx-code. The new vm is 40% faster than the old one (or 1.7x). The new vm has the following characters:
1. op and arg are fully decoded (no decoder)
2. use tos register (need a few special codes)
3. all jumps are absolute
4. faster access to local variables

The disadvantage is that the size of code segment is double (as xop[.] and xarg[.] are two arrays replacing cs[.]). The object file format is changed. The compiler itself has also been improved. The symbol table is changed to be more space efficient. It is also better tuned (now with only 1/3 probing of the previous version). The listing generation is much faster as it employs "index" to the symbol table instead of searching for a symbol by a reference. I found out that the meaning of a full macro and a normal macro are different. Therefore I decide to adopt only one meaning. A normal macro is much more useful, hence the full macro is discarded.

som v 4.0

2 July 2008
The 2007 series of Som are very exciting (Som v3.0 and Som v3.1). They are fast with new instruction sets and improved compilers. With their performance comes the complexity. The sx-code of som v3.1 has 93 instructions and it needs a complicate code conversion to make use of top-of-stack register. I want to retain performance of 2007 series but I really want to make the instruction as simple as the original s-code (at least in terms of the number of instruction). To this goal I design an accumulator-based instruction set with one-address format. Som v4.0 uses the new vm based on u-code. It achieves two objectives:

1) u-code instruction set is as simple as the original s-code. It has only 43 instructions with consistent format.
2) The compiler is much simpler than Som v3.1 and it produces a fast code. It is 32% faster than som v.3.0. It is comparable to som v.3.1. However v3.1 is 10% faster.

som v 4.1

9 Aug 2008
Som v.4.1 uses the improved instruction set. The new instruction set includes the immediate mode and a few AC-arg instructions (to use AC as the argument for the next instruction, mostly load index). The expectation is to reduce the noi by 10-20% and also the running time comparing to som v.4.0. The compiler has gone through several improvements and it is better than the previous version (it is faster and produces better code).

Upon analysing v4 vs. v31 compiler many possible improvements to u-code come to mind. The first is the immediate mode. The second is to use "cascade" AC to reduce "put" (using AC as argument in some instruction). However, adding everything will make u-code unattractively large. The aim to include more instruction into u-code is to improve the performance without undue increase in complexity to the instruction set.

In general benchmark, u2 does not improve much over u-code of som v.4.0. It is only 6% less noi. The running time speedup is only 8.7%. However, most improvement is done by analysing the compiler. To this end, the compiler benchmark is much improved. The noi of v4.1 compiler is 20% less than v4. The running time speedup is insignificant (may be due to heavily i/o bound?).

som v 4.2 Triple 9 release

(9/9/2009)
This is a small experiment on lex. Lex is implemented as a built-in function (via syscall 16). The hope is that this will accelerate the compiler. lex2.c is written based on token-s.txt (lex in Som). An experimental version is lex0.c. It has a better buffering. However, it is more complex. lex0.c uses buffering to reduce the number of fread( ) call. lex2.c reads one line at a time. It is quite interesting to see how the new lex is interfaced to the old token-s.txt in a simple way. In terms of performance, v42 noi is 78% of v41 (or 22% faster) but the runtime in similar.

som v 4.2a

22 Sept 2009
Fix interactive mode. Som 4.2a is som 4.2 (triple 9 release) with correction to lex that enables it to run in interactive mode. After two or three years of contemplation, I decide to write a new parser generator. A lot of code is borrowed from Som compiler herself. The parser generator is about 700 lines of Som. The new parser is faster than the original one. In terms of performance, it is 30% faster than v4.1 (noi) and 11% faster than v4.2 (noi).

som v 5.0 T2-code

5 Dec 2009 Long Live the King
This release has the goal to do "the fastest vm". To reach this goal, the vm uses three-address instruction format (t-code, som v 1.8) because it offers the lowest number of instruction executed (noi). Therefore in terms of performance, the noi will be smallest amongst all previous Som releases. Because running time is directly varied with noi, it will also be "the fastest som". This work is based on Som v19 series of experiments. T2-code is introduced. In terms of performance, it is 40% less noi than v4.2. The runtime is 10% faster than v4.2.

som v 5.1 Christmas release

25 Dec 2010 Christmas release
The previous release (Som v 5.0) introduced t2-code. It is the fastest Som vm to date. t2-code has quite a wide instruction, 96 bits. The aim of this version is simple: to design the instructions to fit into 64 bits and to achieve that without sacrificing the performance. 64-bit is a more natural size for today's machine (year 2010).

The design for t2-64 code is straightforward. It is similar to t2 code, only the format is changed. The new format has two arguments fit into the first 32-bit word and one argument in the second 32-bit word. To allow as many bits as possible to the two argument fields, it is divided into 16-bit, 10-bit and 6-bit (opcode). The argument that is too large to fit into 16-bit or 10-bit needed to be "mov"ed to a smaller size by an extra "mov" instruction that has large argument size: 26-bit and 32-bit.

The result: the executable size for all benchmarks are smaller by 30% than t2-code (not surprising!). In terms of execution speed, for small size benchmarks, t2-64 is slower (noi) by 1% and for medium size benchmarks, by 10%. In terms of wall clock time, t2-64 is 12% slower averaged over all benchmarks.

Chapter 6 How to build Som

Source files

Source code of Som system comprises of Som source files and the virtual machine (in C) source. The sequence of loading files (automatically from the project file som51.txt) is:

lib2.som	
string-s.txt	
compile-h-s.txt	define constants
list-s.txt	list construction functions: car, cdr, cons, list.
symtab-s.txt	symbol table functions
token-s.txt	tokeniser
parse-h-s.txt	parser function prototypes
stmt-s.txt	functions that support parsing
parse2.som	parser, generated from pgen
icode-s.txt	functions that support outputing code

gencode-s.txt	code generator
macro-s.txt	macro expansion
main-s.txt	main

The virtual machine consists of C source files:

som.c	main
lex.c	lexical analyser
interp.c	virtual machine functions

How to compile Som system

The whole system is compiled by "load" function. The "project" file contains the lines that load the whole sequence of files.

```
> som som51.txt
```

This will generate `som51.obj`, the compiler object. It also generates listing file for debugging purpose. The parser "`parse2.som`" is generated from the parser generator "pgen". pgen takes a text file that specifies grammar of Som-language, such as,

```
    top -> tkTO fundef | ex #
    ex1 ->  tkIF ex0 ex exelse $doif(); | ...
```

and produces a parser. The action routines in the grammar such as ""doif" are the functions reside in the "`stmt-s.txt`" file. pgen takes grammar and produces a parser in Som language,

```
    pgen < grammar.txt > parse2.som
```

It is not necessary to regenerate the parser except when you want to change the lexicon and the grammar to modify Som-language for your application.

Sample session

Som can be used in two modes: interactive, batch (produce listing and object files), and excute mode. When start the system loads "`lib2.som`" which contains small set of useful macro functions.

interactive

This mode is suitable to try out a program. A source can be loaded by the function "`load filename`". A user can interrogate global variables and executes all functions in the program. This is useful for debugging purpose.

```
C:>som
>print 2 + 3 nl
5
>to sq x = x * x
```

```
>print sq 5 nl
25
>
```

To define a function, the whole function must be completed in one line.

batch mode

This mode is used to run a program. It will produce a listing file and an object file.

```
C:>som bubble.txt
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
```

It will produce "bubble.lst" and "bubble.obj".

execute mode

This mode is used to execute a Som object (already compiled). It will execute the program immediately (not producing any output file).

```
C:>som -x bubble.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
```

Show a help message

```
C:>som -?
som          for interactive mode
som file     for batch mode, output .lst and .obj
som -x file  for load file and then interact
som -?       for this help
```

Publications

This is the published works related to Som language, stack-processors and virtual machines (1997-2008).

1. Thontirawong, P. and Chongstitvatana, P., "Augmenting a Stack-based Virtual Machine with One-address Instructions for Performance Enhancement", Int. Conf. on Embedded Systems and Intelligent Technology, Bangkok, Feb 27-29, 2008.
2. Chongstitvatana, P., "Threaded Language As a Form of Partial Evaluator", invited paper in National Conf. of Computer Science and Engineering, Thailand, 19-21 November 2007.
3. Satayavibul, C. and Chongstitvatana, P., "An embedded processor with instruction packing", Electrical Engineering, Electronics, Computer, Telecommunications and Information Technology (ECTI) International Conference, Chiang Rai, Thailand, 9-12 May 2007, pp.1135-1138.

4. Lertteerawattana, W., Jedsadawaranon, T. and Chongstitvatana, P., "Instruction Packing for a 32-bit Stack-Based Processor", International Joint Conference on Computer Science and Software Engineering, Thailand, 2-4 May 2007, pp.126-130.
5. Chongstitvatana, P., "Stack Frame Caching", invited paper in Proc. of National Conf. on Computer Science and Engineering, Khon Kan, Thailand, 23-25 Oct. 2006.
6. Sattayawiboon, C., Sripornprasert, J., Tansutthiwess, S., Tonteerawong, P., and Chongstitvatana, P., "A stack processor with integrated display circuit for a low cost CD-ROM reading device", ECTI International Conference, May 10-13, Thailand, 2006.
7. Chongstitvatana, P., "A compact code 16-bit processor for embedded applications", Joint conf. of computer science and software engineering, Nov 2005, Thailand.
8. Chongstitvatana, P., "Self-generating systems: how to a 10,000,000_2 line compiler assembles itself", invited paper, 8th National Computer Science and Engineering Conference, Bangkok, Thailand, October 27-28, 2005.
9. Nanthanavoot, P., Burutarchanai, A., and Chongstitvatana, P., "Instruction packing for a 32-bit resource efficient processor," National Science and Technology Development Agency (NSTDA) Annual Conference, Thailand, 27-30 March 2005 (in Thai).
10. Burutarchanai, A., Nanthanavoot, P., Aporntewan, C., and Chongstitvatana, P., "A stack-based processor for resource efficient embedded systems", Proc. of IEEE TENCON 2004, 21-24 November 2004, Thailand.
11. Burutarchanai, A., Kotrajaras, V. and Chongstitvatana, P., "A fast instruction fetch unit for an embedded stack processor", Proc. of Int. Conf. on Information and Communication Technologies (ICT 2004), 18-19 November, 2004. Thailand.
12. Burutarchanai, A., and Chongstitvatana, P., "Design of a two-phased clocked control unit for performance enhancement of a stack processor", National Computer Science and Engineering Conference, Thailand, 21-22 Sept. 2004, pp.114-119.
13. Nanthanavoot P. and Chongstitvatana, P., "Code-Size Reduction for Embedded Systems using Bytecode Translation Unit", Conf. of Electrical/Electronics, Computer, Telecommunications, and Information Technology (ECTI), Thailand, 13-14 May 2004.
14. 14. Chongstitvatana, P., "The art of instruction set design", invited paper in Conf. of Electrical Engineering, Thailand, 2003.
15. 15. Kotrajaras, N., Chongstitvatana, P., "Nibbling Java byte code of resource-critical devices", National Computer Science and Engineering Conference, Thailand, 2003.
16. Chongstitvatana, P. and Kotrajaras, V., "Instruction compression by nibble coding: war on the old front", IEEE Thailand section: Silver Jubilee Symposium, 15 Nov 2002.
17. Nanthanavoot, P. and Chongstitvatana, P., "Development of a data reading device for a CD-ROM drive with FPGA technology", Conf. of Electrical Engineering, Thailand, 2002.
18. Wongsiriprasert, C. and Chongstitvatana, P., "Performance comparison between two virtual machine interpreters : stack-based vs. register-based",

Proc. of 3rd Annual National Symposium on Computational Science and Engineering, Bangkok, 1999, pp. 401-406.
19. Chongstitvatana, P., "Post processing optimization of byte-code instructions by extension of its virtual machine", 20th Electrical Engineering Conference, Thailand, 1997.

last update 3rd January 2011