Self-Generating Systems: How a $10,000,000_2$-line Compiler Assembles Itself

Prabhas Chongstitvatana
Department of Computer Engineering, Chulalongkorn University
Bangkok 10330, Thailand
Email: prabhas@chula.ac.th

Abstract

This work described a self-generating system. A system is self-generated when it can bootstrap itself on a system without an external help. A compiler is used as a concrete example. A small $10,000,000_2$-line[1] compiler is illustrated to generate an executable version of itself from an input description. The automatic bootstrapping process consists of several stages. Each stage reads the input description and extends the system itself into a "higher" stage. The input description is "evolved" through these stages from a low-level language into a high-level language. The result is an executable system that built itself step by step into a full compiler.

Key Words: self-generating systems, compiler, bootstrap

1. Introduction

How a complex system is created? One possible scenario is that it starts from a simple working system (how this simple system comes into being is another topic), then it extends its capability, very likely that it becomes a tool to develop a next generation system. This process is repeated and the system "advances" itself into a complex system.

Let us consider how complex software is created. How contemporary application software is implemented? It needs many powerful tools such as a compiler and some ready-built software components. One can ask further, how a compiler is created? Most of the present day compilers have been developed through many generation of compiler development (for example, Pascal, Oberon etc.). Each succeeding version was built from the compiler of the earlier version.

To investigate further it is natural to ask "How the first compiler is created?" It was probably developed from a simpler tool such as an assembler. The tools constrain a programmer to write a compiler in an assembly language. If we follow this questioning further, we will reach the question "How an assembler is created?", and so on.

The purpose of this article is to explore the space from the very beginning of perhaps nothing to the first compiler. We will follow the iterative process of incrementally extending a system to be more capable so that it becomes a tool for the next generation development. During its evolution, the description (representation) of the system is also evolved.

What is the point of this exercise? Besides creating a beautiful piece of abstraction (software) with interesting qualities, this exercise demonstrates *an evolvable system of description that describes the process of its own evolution.* It is very interesting that the process of evolution of a complex system can be described within itself.

2. General idea

To build evolvable software, we need to dynamically compile and execute a program on the fly. In general, we can think of each successive generation of software as a code block. This code block must be created in the system memory and then it is activated to work. Starting from the first code block, let us call it Block0, we assume it is already resided in the memory. Once it starts working, it creates the next version of software, the code Block1. At the termination of execution Block0, it activates Block1 then Block1 creates Block2 and actives it and so on to BlockN.

How can a code block create a new code block? It reads a description of the new code block from an input stream. This description is an extension of the

[1] I attribute this parody to D.E.Knuth [1]

current program to be the next version of that program. This description is the key to our bootstrapping process.

3. Implementing a compiler

How can we write the first compiler from nothing? Not really, we must assume that at the bottom line there is a machine language that our program will finally be transformed into. This machine language is the one that can be executed on a computer (either virtual or real). Let us call it the language *M*. We are not going to write the whole thing in machine codes. The semantic content of *M* is too low for a human programmer to develop a complex software. Preferably, we want to write a program for a particular task using the highest language semantic available. For example, for a compiler, we would like to have a language that have this statement "compile *P*" where *P* is our target language, let us call the target language, the language *L*. We will need to evolve *M* to *L* through bootstrapping process. We want to stress that this bootstrapping process is entirely automatic.

Suppose there are successive system evolution *s0*, *s1*, *...sn*. The bootstrapping process works like this. Starting from *s0* resided in the memory, *s0* is activated and it reads a description of a language *l1* and builds *s1* in the memory then activates it. *s1* goes on to read *l2* (perhaps from the same input stream) and builds *s2* and so on until the process reaches the target system with the language *L*. Each succeeding *l1*, *l2*, *...L* will have higher semantic content than the preceding one. In other words, it will be a higher level language than its predecessor.

Let us do a concrete exercise on a compiler. We are going to implement a compiler. To keep the size of this exercise small, the target language must be simple to compile. So, we design a language *L* to be a postfix language. It has global variables, function definition, local variables, control flow such as if, else, while, and usual operators. It has only integer data type and dynamically allocated array data structure. The following programs show the language *L*.

Notation: Algorithms and programs are shown in **`courier`** font, the input stream is in **Arial** font.

```
to print i { }  i sys 3 end

; factorial
to fac n { }
     if n 0 == { 1 else n n 1 - fac * }
end

; a global variable
var ax

; sum all elements of an array
to sum ar size { i s }
  1 -> i  0 -> s
  while i size <= {
    s ar i ldx + -> s   ;s = s + ar[i]
    inc i
  }
  s                     ; return s
end

to main { }
  6 fac print
  20 array -> ax
  ax 0 1 stx            ; ax[0] = 1
  ax 1 2 stx            ; ax[1] = 2
  ax 20 sum print
end
```

where `ax` is a dynamically allocated array, **`->`** is an assignment operator, `ldx/stx` are array access operators.

The compiler for *L* can be written in itself in 128 lines (the full program is shown in the appendix). However, when we describe algorithms in this work, we will use an infix version of *L* for the sake of familiarity.

4. Machine code

We also define a machine language, *M*, that will run on a virtual machine. This language is stack-based. It uses a stack for evaluation. It has a high level "stack frame" that stores state of computation and local variables. This is used for function call and return. The instruction has two formats: operator and operator with one argument. Each instruction is a fixed length 32 bits, the left most 24-bit is an argument, the right most 8-bit is an operation code. The instruction without an argument will have its argument field filled with zero.

no-arg operators:
`add, sub, mul, div, eq, ne, lt ....`
one-arg operators:
`jmp,jt,jf,call,ret,ld,st,get,put...`

The complete set of instruction can be found in [2]. For example, the factorial function above can be compiled into *M* as follows:

```
; factorial
to fac n { }
  if n == 0 { 1 else n * (fac n - 1) }
end
```

```
 3 Fun 1
 4 Get 1
 5 Lit 0
 6 Eq
 7 Jf 3
 8 Lit 1
 9 Ret 2
10 Get 1
11 Get 1
12 Lit 1
13 Sub
14 Call 3
15 Mul
16 Ret 2
```

A virtual machine for *M* is assumed to run on the target platform where we want to bootstrap the self-assembling system.

5. Bootstrapping

The only input/output functions assumed in *L* are reading a character from an input stream and writing an integer to an output stream. The evolution process from *M* to *L* starts with a simple loader that can read only integers. The first code block is loaded by the virtual machine. The next step is a lexical analyser that understands simple machine code format and comment lines. Then the system can read symbols, which is used in a simple assembler. The next step is a better assembler with symbolic names. Then the system is evolved into a simple compiler which understand control flow; if, else, while. The final evolution is a full compiler for *L* that has function definition, global and local names. Many routines from an earlier evolutionary step are reused in the subsequent evolutionary steps. These steps are illustrated below (see Fig. 1 at the end of this article for the flow of the whole process).

Block0
program
  read integer
  loader1
input
  machine code of Block1

Block1
program
  symbol table (sym)
  lexical analyser (lex)
  a simple assembler (asm1)
input
  initial symbols
  asm2 written in the language of asm1

Block2
program
  better assembler (asm2)
  with symbolic names
input
  com1 written in asm2

Block3
program
  a simple compiler (com1)
  with control flow
input
  com2 written in com1

Block4
program
  the final compiler for L (com2)
input
  application programs written in L

The first code block, Block0, starts the whole bootstrapping process (Fig. 2). It is a loader that loads machine code of Block1. Block0 itself is loaded by the virtual machine. It is written in machine code (*M*) in the format that is suitable for the virtual machine to read. Block0 contains a program, `loader1`, which reads s-code in the format: (operator argument)* terminated by 0 (Fig. 3). For the reason of simplicity of the routine to read an integer in `loader1`, a negative integer is represented by a positive integer prefixed by a zero "0". You can see the difference between the code in Block0 and Block1 by noticing this negative integer.

```
32 139 23 0 38 1 24 1 31 48 13 0 24 1 31 57 12 0 5 0
40 2 38 1 24 1 31 32 9 0 24 1 31 10 9 0 6 0 24 1
31 40 9 0 6 0 24 1 31 41 9 0 6 0 24 1 31 10 11 0
6 0 40 2 38 2 36 3 25 1 24 1 31 59 9 0 30 8 28 3
36 3 25 1 24 1 31 10 10 0 29 -5 24 1 27 1001 24 1 36
2
…
0
```

Figure 2. the machine code of Block0

```
to outc op arg { }   ; put code at CP
  CS[CP] = (arg << 8) | op
  CP = CP + 1

; format (op arg)* .. 0
to loader1 { op arg }
  op = readint
  while op != 0
    arg = readint
    outc op arg
    op = readint
```

Figure 3. `loader1` in Block0, `readint` reads an integer from an input stream, CS is code segment, CP is a pointer to CS

```
32 227 32 252 32 276 32 403 32 443 32 788 23 0 1 0
38 2 26 1004 25 1 26 1004 31 4 1 0 27 1004 26 1005
24 1 24 2 19 0 24 1 40 3 38 1 26 1005 24 2 18 0 31 0
9 0 30 6 26 1005 24 2 24 1 32 150 19 0 26 1005 24 2
18 0 40 3 38 2 28 6 26 1001 31 0 11 0 30 2 28 4 32
33 32 12 29 07  26 1001 32 3
...
0
```

Figure 4. The input of the code Block1, a negative integer is underlined

Block1's code contains a routine to handle a symbol table and a routine to do lexical analyser. The symbol table is implemented as a Trie using the linked list of characters. All lexicons in the input stream are inserted into this table. The lexical analyser understands the comment line and two formats for operators: op without argument and op with argument. The machine code format becomes: operator [argument] [comment line]* where [] denotes option.

```
; a tiny assembler
to asm1 { i j op }
  i = lex
  while i > 1 {
    op = sym[i+3]   ; get attribute
    tkval = 0
    if op > 23 { j = lex }
    outc op tkval
    i = lex
  }
```

Figure 5. A tiny assembler in Block1

With the symbol table, we can associate symbols with their internal representation (integer). Block1 is a simple assembler. Fig. 5 shows its algorithm. The system now can understands the input stream of the format: symbolic-operator [numeric argument]. The initial symbols for the symbol table are read from the input stream with the format (symbolic-operator operation-code)* terminated by 0 (Fig. 6).

Block2 is a better assembler written in the language of the simple assembler of the previous generation of the system (Fig. 7). Block2's assembler understands labels. A label is used to associate the current code address with a symbol. This is used for labeling a function definition. The assembly language statement "`def symbol value`" is used to associate the symbol to the value. It is used to define symbolic names, for example, global variables or previously defined machine code routines.

```
; init symtab
+ 1  - 2  * 3  / 4  & 5  | 6  ! 8  == 9  != 10  < 11
<= 12  > 14  >= 13  ^ 7  % 17  << 15  >> 16
ldx 18   stx 19   array 22  halt 23
get 24   put 25   ld 26     st 27    inc 34
fun 38   ret 40   jmp 28    jt 29    jf 30   call 32
lit 31   sys 36
stop 50  { 51    } 52     else 53  def 54
if 55    while 56  -> 57    to 58    end 59  var 60
0
; end symtab
```

Figure 6. The symbol table, this symbol table included keywords (value > 49) for subsequent development such as `{ } if else while stop to end var`

```
; sym 1005
; lex  179
; isStop 394
  ...
; 403 asm2 { i a }

fun 3
call 179  put 2                        ; i = lex
ld 1005  get 2  lit 3  + ldx  put 1    ; a = sym[i+3]
get 1  call 394  jf 2  ret 3           ; if isStop a break
get 1  get 2  call 336                 ; dolabel a i
get 1  call 348                        ; dodef a
get 1  call 362                        ; doop a
jmp 019                                ; asm2  jmp -19
ret 3

0
```

Figure 7. A better assembler of Block2 written in `asm1`

Now the system is ready to evolve into a high-level language compiler with control flow and function definition. The first step is to do a compiler with control flow. Block3 extends the better assembler (`asm2`) to understand "`if`" "`else`" and "`while`". This is done using a recursive parser that generates all the required jump, conditional jump instructions (Fig. 8).

```
def tkval 1000
def CP 1002
def CS 1003
def sym 1005
def TK 1007

def outc 81
def lex 179
def dolabel 336
def dodef 348
def doop 362
def isStop 394
                          ; 443 com1 { i a b }
```

```
com1 fun 4
call lex  put 3                  ; i = lex
ld sym  get 3  lit 3  +  ldx     ; TK = sym[i+3]
st TK
ld TK  call isStop  jf 2         ; if isStop TK break
  ret 4
get 3  lit 1  ==  jf 5           ; if i == 1
  lit 31  ld tkval call outc     ;   out lit tkval
  jmp 59
ld TK  get 3  call dolabel       ; dolabel TK i
ld TK  call dodef                ; dodef TK
ld TK  call doop                 ; doop TK
ld TK  lit 56  == jf 19          ; if TK == while
  ld CP  put 2                   ;   a = CP
  call com                       ;   com1
  ld CP  put 1                   ;   b = CP
  lit 30  lit 0  call outc       ;   out jF 0
  call com1                      ;   com1
  lit 28  get 2  ld CP  -
  call outc                      ;   out 28 a-CP
  get 1  ld CP  call patch       ;   patch b CP
  jmp 30
ld TK  lit 55  == jf 26          ; if TK == if
. . .
stop
```

Figure 8. The compiler, `com1`, of Block3 written in `asm2`

With this language, the final compiler (`com2`) can be written that will be able to compile L. The final code, Block4 is the compiler, `com2`. It extends `com1` to handle the function definition "`to`" "`end`", local variables in the body of a function and global variables using "`var name`" (Fig. 9).

```
com2 fun 5                           ; com2 { i j a b }
 call lex  put 4                     ; i = lex
 ld sym  get 4  3 +  ldx st TK       ; TK = sym[i+3]
 if ld TK call isStop { ret 5 }      ; if isStop TK brk
 if get 4  1 == {                    ; if I=1 out lit tkval
   31  ld tkval call outc
 else                                ; else
   ld TK  call dodef                 ;   dodef TK
   ld TK  call doop                  ;   doop TK
   if ld TK 56 == {                  ;   if TK == while
     ld CP  put 2                    ;     a = CP
     call com2                       ;     com2
     ld CP  put 1                    ;     b = CP
     30 0 call outc                  ;     out jf 0
     call com2                       ;     com2
     28  get 2 ld CP - call outc      ;     out jmp a-CP
     get 1 ld CP call patch          ;     patch b CP
 . . .
 else if ld TK 100 > {               ;  else if TK > 100
   get 4 call typeof put 3           ;     j = typeof i
   get 4 call getref put 2           ;    a = getref i
   if get 3 1 == {                   ;    if j == 1
     32 get 2 call outc              ;      out call a
   else if get 3 2 == {              ;     else if j == 2
     26 get 2 call outc              ;      out ld a
  else                               ;    else
    24 get 2 call outc } }           ;      out get a
 } } } } } } }
 jmp 0206                            ; com2
ret 5

stop
```

Figure 9. The final compiler, `com2`, written in `com1` with control flow in Block4

The example below shows a program in *L* and the *M* output code (Fig. 10).

```
to print i { } i sys 1 end

; a global variable
var ax

; sum all elements of an array
to sum ar size { i s }
  1 -> i  0 -> s
  while i size <= {
    s  ar i ldx + -> s   ;s = s + ar[i]
    inc i
  }
  s               ; return s
end

to main { }
  20 array -> ax
  ax 0 1 stx           ; ax[0] = 1
  ax 1 2 stx           ; ax[1] = 2
  ax 20 sum print
end

32 25 23 0 38 1 24 1 36 1 40 2 38 3 31
1 25 2 31 0 25 1 24 2 24 3 12 0 30 8 24
1 24 4 24 2 1 0 25 1 34 2 28 -10 24 1
40 6 38 1 31 20 22 0 27 1001 26 1001
31 0 31 1 26 1001 31 1 31 2 24 1
26 1001 31 20 32 7 32 3 40 2
```

Figure 10. An example of an application program translated into a correct *M*-code from the first example of this paper in the format of the raw code similar to Block0

6. Discussion and related work

Now we can discuss the question "How the first compiler is built?". Using a contemporary software tool such as a C compiler, the final 128-line compiler, which is itself written in *L*, can be written in C and then compile to generate an executable code. The self-assembling compiler described in this work is different. It does not require any outside tool such as a compiler to generate an executable code. It does bootstrap itself from a small beginning then extends itself into a more complex system by reading many sequences of description from an input stream. The input description that the system used to build itself has an interesting character. As the system evolves itself, the description becomes richer in its semantic. This is intentional because it was designed so that a human programmer find himself/herself a higher and higher level of tool that facilitates the task of writing programs. Reading through the whole sequence of input description to this self-generating system, you can notice the "evolution" of the description easily, from machine codes, to an assembly language and finally to a target high level language (see the full input stream at the reference website [2]).

This description is planned and is written by a human. In an ordinary task of writing a compiler, it also requires a human designer. The main distinction of this self-generating system is that the description itself is also "evolved" through bootstrapping process. The stages of bootstrapping process must be carefully planned. However, it is possible to imagine a system that can search for this description. For example, evolutionary techniques such as Genetic Algorithms [3] or Genetic Programming [4] can be used to search for such description. Because the evolution step of this description is small, it is more like a continuum rather than a big jump. The search for a description that will extend a system to grow in the right direction is more likely to be possible.

The bootstrapping process has been used to develop software systems since the early days of computer evolution. Human is the main agent to do the bootstrapping process. By creating a more and more powerful system based on a previous one, a very powerful system has been successfully developed. An evolution of any programming language is the case, for example FORTRAN and many of its variants, B to BCPL to C, C to C++ and then to JAVA, ALGOL to many languages as its descendant such as Pascal, Oberon etc. Most of the history of programming language was recorded in Annal of history of computing [5] and the book [6].

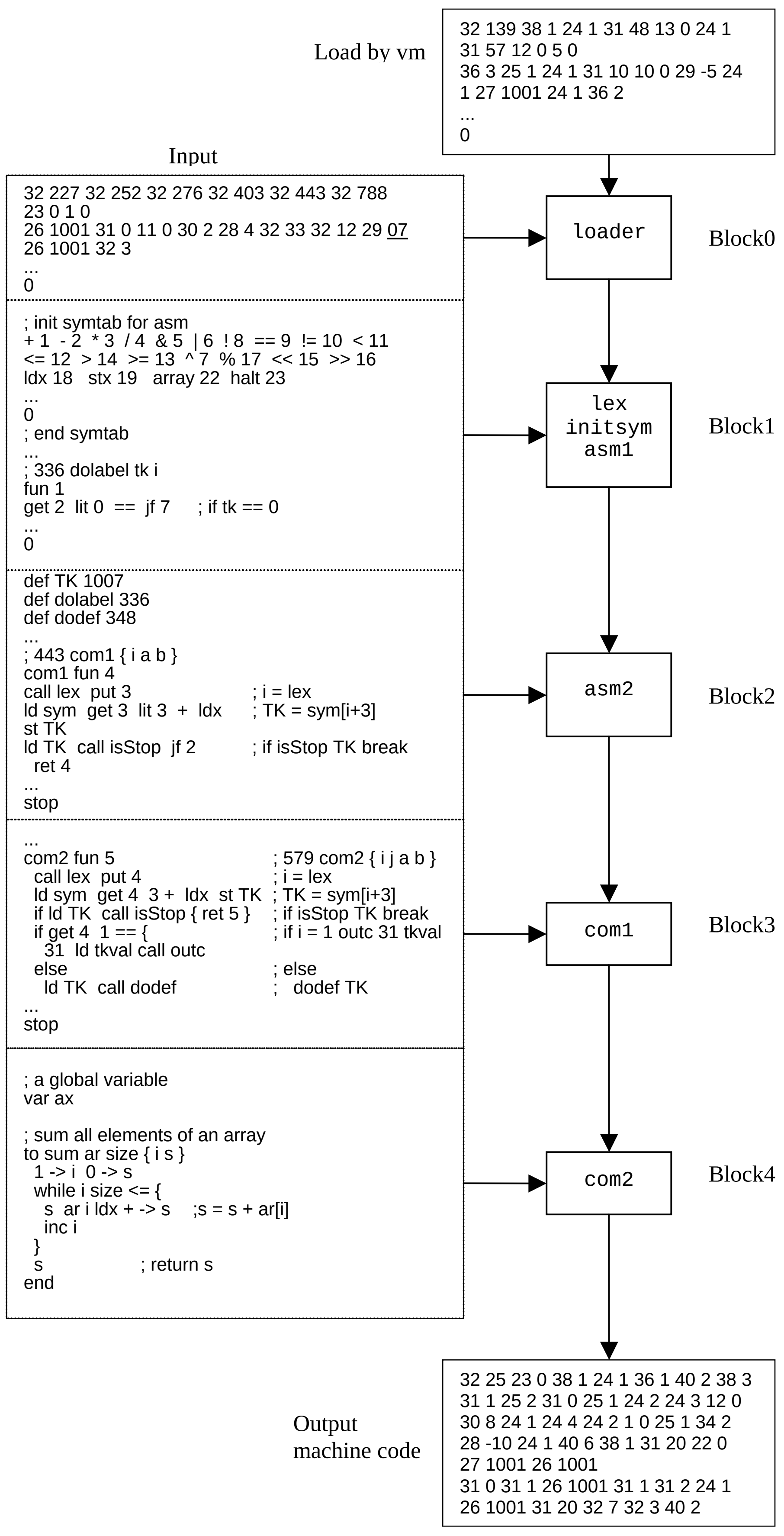
The minimalist approach to programming was championed by FORTH [7]. The bootstrapping process was described as a meta-compiler, one of the most significant contribution in this area is cmFORTH [8] which the whole meta-compilation process (FORTH code plus machine code) to build a new system on another platform can be described in about 10 pages. Comparatively, the input description of the system described in this work is 6 pages long. It is interesting to note that, an interpreter is more compact than a compiler. A self describing interpreter, so called meta-interpreter is very compact. An extreme example can be cited from PROLOG where its meta-interpreter is almost trivial to write. A full PROLOG interpreter that can handle "cut" can be written in less than one page of PROLOG [9]. You can see the whole code including the next evolution step of this compiler that has tail-call optimisation in my webpage [2]. It also contains the source and executable code for the virtual machine used in this work.

Acknowledgements

I would like to thank my colleagues, Yanyong Tengamnuay and Somchai Prasitjutrakul for a lively debate of my early idea of this work which helped me to polish it to this stage. I like to thank Jaruloj Chongstitvatana who did all illustrations in this work.

7. References

[1] D. E. Knuth homepage, his birthday event in 2002.
[2] http://www.cp.eng.chula.ac.th/faculty/pjw/project/sgs/
[3] D. Goldberg, Genetic Algorithm in search optimization and machine learning. Addison Wesley, 1989.
[4] J. Koza. Genetic Programming II: Automatic discovery of reusable programs, MIT Press, 1994.
[5] http://www.computer.org/pubs/annals/annals.htm
[6] T. Bergin and R. Gibson, History of programming languages, Addison Wesley, 1996.
[7] Charles Moore, the inventor of FORTH language, http://www.colorforth.com/bio.html
[8] J. Melvin, Demise of the metacompiler in cmForth, ACM SIGFORTH Newsletter, Volume 1 , Issue 2, 1989, pp.7-8.
[9] Y. Shoham, Artificial Intelligence Techniques in Prolog, Morgan Kaufmann Publishers, 1993.



```
32 139 38 1 24 1 31 48 13 0 24 1
31 57 12 0 5 0
36 3 25 1 24 1 31 10 10 0 29 -5 24
1 27 1001 24 1 36 2
...
0
```

```
32 227 32 252 32 276 32 403 32 443 32 788
23 0 1 0
26 1001 31 0 11 0 30 2 28 4 32 33 32 12 29 07
26 1001 32 3
...
0
```

```
; init symtab for asm
+ 1  - 2  * 3  / 4  & 5  | 6  ! 8  == 9  != 10  < 11
<= 12  > 14  >= 13  ^ 7  % 17  << 15  >> 16
ldx 18   stx 19   array 22  halt 23
...
0
; end symtab
...
; 336 dolabel tk i
fun 1
get 2  lit 0  ==  jf 7      ; if tk == 0
...
0
```

```
def TK 1007
def dolabel 336
def dodef 348
...
; 443 com1 { i a b }
com1 fun 4
call lex  put 3                 ; i = lex
ld sym  get 3  lit 3  +  ldx    ; TK = sym[i+3]
st TK
ld TK  call isStop  jf 2        ; if isStop TK break
  ret 4
...
stop
```

```
...
com2 fun 5                          ; 579 com2 { i j a b }
  call lex  put 4                   ; i = lex
  ld sym  get 4  3 +  ldx  st TK    ; TK = sym[i+3]
  if ld TK  call isStop { ret 5 }   ; if isStop TK break
  if get 4  1 == {                  ; if i = 1 outc 31 tkval
    31  ld tkval call outc
  else                              ; else
    ld TK  call dodef               ;   dodef TK
...
stop
```

```
; a global variable
var ax

; sum all elements of an array
to sum ar size { i s }
  1 -> i  0 -> s
  while i size <= {
    s  ar i ldx + -> s    ;s = s + ar[i]
    inc i
  }
  s                   ; return s
end
```

```
32 25 23 0 38 1 24 1 36 1 40 2 38 3
31 1 25 2 31 0 25 1 24 2 24 3 12 0
30 8 24 1 24 4 24 2 1 0 25 1 34 2
28 -10 24 1 40 6 38 1 31 20 22 0
27 1001 26 1001
31 0 31 1 26 1001 31 1 31 2 24 1
26 1001 31 20 32 7 32 3 40 2
```

Figure 1. The flow of self-generating compiler

Appendix

The listing of the full compiler (`com2`) in 128 lines in infix *L*.

```
; a 128-line compiler for L

; tkval       value of token num
; last        the end of last fun
; numlocal    no. of locals of current fun
; freecell    pointer to free sym
; CH          current input char
; TK          attribute of current token
; CP          code pointer
; CS          code segment, array 1000
; sym         symbol table, array 1000
; lvrec       record of locals, array 20

to isNum c { } (c >= 48) & (c <= 57) ; 0..9
to isSpace c { } (c < 33)           ; tab space nl

to readc { }        ; read one char
  CH = sys 3
  CH

; a cell has 4 fields: char,right,next,atr
to newcell c { k }
  k = freecell
  freecell = freecell + 4
  sym[k] = c
  k

to search i c { }    ;  if sym[i]=0 insert c at i
  if sym[i] == 0 { sym[i] = newcell c }
  sym[i]

; return index to symtab[], 1 if numeric
to lex { i }
  while readc
    if CH == 59
      while CH != 10 { readc }   ; skip comment
    if ! isSpace CH { break }    ; skip blank
  if isNum CH
    tkval = CH - 48
    while ! isSpace readc
      tkval = tkval*10 + CH - 48
      1 break                    ; it is numeric
  i = 0
  while ! isSpace CH             ; check space
    i = search i+2 CH            ; next of i
    while sym[i] != CH { i = search i+1 CH }
    CH = readc                   ; read next
  i

to outc op arg { }
  CS[CP] = (arg << 8) | op
  CP = CP + 1

to patch a1 a2 { op }            ; relative address
  op = CS[a1] & 255
  CS[a1] = ((a2-a1) << 8) | op

to rename d { } numlocal - d + 1  ;1..n to n..1
to typeof i { } (sym[i+3] >> 20) & 15
to getref i { } sym[i+3] & 1048575
to setattr i ty ref { } sym[i+3] = (ty << 20) | ref

to parse { i j a b }
  i = lex
  TK = sym[i+3]                    ; get attr
  if i == 1  outc 31 tkval         ; lit tkval
  else if TK == 0
    numlocal = numlocal + 1
    setattr i 3 numlocal           ; declare local
    lvrec[numlocal] = i
  else if TK < 24  outc TK 0       ; op no-arg
  else if TK < 50                  ; op arg
    j = lex                        ; get arg
    outc TK tkval
  else if TK < 54  break           ; stop { } else
  else if TK == 56                 ;   ->
    j = lex                        ;  name
    if (typeof j) == 2
      outc 27 getref j             ; gvar, st ref
    else
      outc 25 rename getref j      ; lvar, put ref
  else if TK == 53                 ;   to
    last = CP                      ; record fun
    j = lex                        ; fname
    setattr j 1 CP
    parse                          ; get pv until {
    a = numlocal                   ; set arity
    parse                          ; get lv until }
    outc 38 numlocal-a             ; fun lv
  else if TK == 54                 ;   end
    outc 20 numlocal               ; ret n
    i = 1
    while i <= numlocal
      setattr lvrec[i] 0 0
      i = i + 1
    numlocal = 0                   ; clear locals
  else if TK == 57                 ;  while
    a = CP
    parse                          ; cond {
    b = CP
    outc 30  0                     ; jf 0
    parse                          ; body }
    outc 28 a-CP                   ; jmp a
    patch b CP
  else if TK == 55                 ;  if
    parse                          ; cond {
    a = CP
    outc 30 0                      ; jf 0
    parse
    if TK == 52                    ;   else
       patch a CP+1
       a = CP
       outc 28 0                   ;   jmp 0
       parse
    patch a CP
  else if TK == 60                 ;   var
```

```
    j = lex                       ; global name
    setattr j 2 (array 1)         ; gvar, alloc DS
  else                            ; TK > 100, names
    j = typeof i
    a = getref i
    if j == 1  outc 32 a          ; fun, call ref
    if j == 2  outc 26 a          ; gvar, ld ref
    if j == 3  outc 24 rename a ; lvar, get ref
  parse

to main { i k }
  freecell = 4
  CP = 1
  CS = array 1000
  sym = array 1000
  lvrec = array 20
  i = lex                         ; init symtab
  while i > 1                     ; until 0
    k = lex                       ; get attr
    sym[i+3] = tkval
    i = lex                       ; get sym
  outc 32 0                       ; call  main
  outc 23 0                       ; end
  parse
  patch 1 last+1
```

End of listing