

Starting From Scratch

Bootstrapping a Compiler for Fun and Profit

Disclaimer: Profit not guaranteed

- ◆ Neither is fun.
- ◆ A background in compilers is neither assumed, implied, nor expected.
- ◆ Questions are welcome; and will be answered, possibly correctly.

What's a Compiler?

- ◆ A program that translates from one programming language into another programming language.
- ◆ To create a compiler, we need at least three things:
 - ◆ A source language (e.g. C, Java, LLVM byte code)
 - ◆ A target language (e.g. C, Java ByteCode, LLVM byte code, or the machine code for a specific CPU) to translate it to
 - ◆ Rules to explain how to do the translation.

How do we make a great compiler?

- ◆ There are two approaches to this problem.
 - ◆ A) Write the perfect compiler on your very first try. (Good luck!)
 - ◆ B) Start out with a not-so-great compiler. Use it to write a better one. Repeat.

That's great, but...

- ◆ What if I don't have a not-so-great compiler to start out with?
- ◆ What if I don't have a compiler at all?

The Modern Answer:

- ◆ Just Google it. Someone will have written one for you by now.
- ◆ If they haven't, you can start with an existing compiler for an existing source language (such as C), and change the target language that it generates.

Starting from Scratch...

- ◆ What did we do before we had the Internet?
- ◆ What would we do if we didn't have a compiler to start from?
- ◆ We'd make one! There are many possible ways to go about doing this.
- ◆ Here's my approach...

What are we trying to Do?

- ◆ We want to write a program that can translate from the language we HAVE to the language we WANT.
- ◆ Okay! What language do we have? We don't know yet, until we know what system we're talking about!
- ◆ But... (almost) every computer has a built in CPU chip that has a "machine language" hard-wired into it.
- ◆ So, we just use this "machine language", and all our problems are solved, right?

Wrong!

- ◆ In general, every CPU chip has its OWN version of “machine language”, called its “instruction set”. Sometimes later chips are backward compatible with previous chips, sometimes not.
- ◆ The instruction set that the CPU chip supports is part of the fundamental design of the CPU; the hardware features that the CPU implements are reflected in the CPU instruction set.
- ◆ That means that we can’t just write in a single, universal “machine language” that will work on every CPU.
- ◆ So, we’ll try to find a generic approach, one that would work for a wide variety of processors and platforms.

Our Master Plan

1. First, we'll define a very simple byte-code language; one that's just barely powerful enough to write a compiler with.
2. Next, we'll design a generic compiler for our tiny language; with a set of requirements that we hope we can implement on any machine we like. That way, we'll be able to implement our little compiler using a small amount of machine language from our target machine.
3. From there on in, we can use our existing compiler to write a better compiler that can compile a nicer source language.
4. We repeat step 3 until we have the compiler we want for the language we want.

The Simple Language

Name	Syntax (bytecode)	Meaning
Exit	0	Exit Program
Define	1 n definition	Define byte n as having definition
Quote	2 byte	Outputs byte
Write	n	Output the definition of byte n

A generic compiler (pseudocode)

```
putBytes(ProgramHeader.data, ProgramHeader.length)
```

```
loop
```

```
  code ← getByte()
```

```
  if code = 0 Exit
```

```
    exitProgram()
```

```
  if code = 1 Define
```

```
    n ← getByte(); len ← getByte()
```

```
    ByteCodeTable[n].length ← len
```

```
    ByteCodeTable[n].data ← getBytes(len)
```

```
  else if code = 2 Quote
```

```
    quotedByte ← getByte(); putByte(quotedByte)
```

```
  else Write
```

```
    putBytes(ByteCodeTable[code].data, ByteCodeTable[code].length)
```

A generic compiler (Assembly Language Style)

```
putBytes(ProgramHeader.data, ProgramHeader.length)
```

```
loop: code ← getByte()
```

```
    if code ≠ 0 goto define:
```

```
    exit()
```

```
define: if code ≠ 1 goto quote:
```

```
    code ← getByte(); address ← byteCodeTable[code]
```

```
    code ← getByte(); address[0] ← code; getBytes(address+1, address[0])
```

```
    goto loop:
```

```
quote: if code ≠ 2 goto write:
```

```
    code ← getByte(); putByte(code)
```

```
    goto loop:
```

```
write: address ← byteCodeTable[code]
```

```
    putBytes(address + 1, address[0])
```

```
    goto loop:
```

Requirements to Implement a Generic Compiler (page 1/2)

Code	Requirement
Header (with length)	A string of bytes that identifies program to the operating system
<code>code</code> $\leftarrow$ <code>getBytes()</code>	Can get a byte from our input.
<code>putByte(code)</code>	Can send a byte to our output
if <code>code</code> $\neq$ <code>number</code> goto <code>address</code>	Can compare a byte to a constant, branch if not the same.
goto <code>loop:</code>	Can branch to a fixed address.

Requirements to Implement a Generic Compiler (page 2/2)

<code>exit()</code>	Can exit the program.
<code>address + 1</code>	Can add / Increment
<code>address ← byteCodeTable[n]</code>	Can get the address of a pre-allocated array element.
<code>address[0] ← code</code>	Can set a byte at an address
<code>address[0]</code>	Can get a byte from an address
<code>byteCodeTable</code>	Can allocate/reserve an array of memory.
<code>putBytes(address, length)</code>	Equivalent to calling <code>putByte()</code> in a loop.
<code>getBytes(address, length)</code>	Equivalent to calling <code>getByte()</code> in a loop.

Things We Need To Know (based on our hardware & operating system)

1. How to write a valid program in the CPU instruction set

- Programs must start at memory address divisible by four?
- Must/may not execute certain CPU instructions in certain modes?

2. How to load & run a program

- File with a specific header & permissions: (eg. ELF (Linux), Mach-O (Mac), COM (MS-DOS))?
- Specific boot loader/block device addresses (eg. GRUB (x86 chips), uBoot (ARM chips))?
- Serial device interface with a specific voltage applied to certain pins on the hardware (eg. Microchip's PIC-10 chips)?

3. How to input & output data to & from our program

- System calls, driver functions, or memory mapped hardware?
- Special hardware instructions in the CPU Instruction set?
- Special memory addresses defined by the operating system / hardware?

Initial Compiler for an Android Cellphone (ARM-7 Thumb Instruction Set) (in hexadecimal)

7f 45 4c 46 01 01 01 00 00 00 00 00 02 00 00 28

01 00 00 00 54 00 00 00 34 00 00 00 00 00 00 00

00 00 00 00 34 00 20 00 01 00 00 00 00 00 00 00

01 00 00 00 52 00 00 00 52 00 08 00 00 00 00 00

ff 00 00 00 ff 00 10 00 07 00 00 00 52 00 08 00

04 27 01 20 39 46 58 39 54 22 00 df 0d 18 ff 35

f7 46 22 e0 00 02 29 18 00 28 01 d1 01 27 00 df

01 28 0a d1 02 38 0c 1c f7 46 0f e0 21 1c 08 68

01 31 10 1c 03 27 00 20 00 df e4 ef 02 38 04 27

00 02 29 18 01 20 0a 60 01 31 00 df de ef 03 27

00 20 39 46 04 31 01 22 00 df 08 60 fe 46

ELF "File Header"

ELF "Program
Header"

Write Header

Loop Exit

Define Quote

Write getByte

What now?

- ◆ So, we have a minimal compiler for our tiny language, and we'll even pretend that it works!
- ◆ What's the next step?
 - ◆ Answer: We'll invent a better language, use our existing compiler to implement it!

How do we do that?

- ◆ We'll define a new language based upon symbols that are one byte long.
- ◆ Next, we'll use our simple language to write a compiler for our new language.
- ◆ Finally, we'll write a compiler for our symbolic language **in** our symbolic language, and compile that.

Outline for our new compiler (Minimal language version)

- ◆ For each symbol in our language, we define that symbol as the machine language instructions to run for that symbol.
 - ◆ ^A<symbol><length><machine language>
- ◆ For the rest of our program, whenever we write a symbol, our program will write the machine language to run that symbol.
 - ◆ We're 90% of the way to a compiler for our symbolic language already!
- ◆ If we want to output a specific byte as an argument to a machine code, we can use ^B (Quote) to quote that byte.
- ◆ Lastly, we end our program with a ^@ (Exit) symbol.

So, what's the new language like?

- ◆ 3 variables (a, b, & c)
- ◆ Less control codes (more printable ASCII characters)!
- ◆ Allow comments, nested loops, and conditionals
- ◆ Allow us to add / modify symbol definitions!

A new Language (Literals)

Name	Syntax	Meaning
Comment	<code>; <comment> ^J</code>	Comment until end of line.
Quote	<code>'n</code>	Output byte <code>n</code> unchanged
Hex	<code>x hh</code>	Output byte with hexadecimal value <code>hh</code>

A new Language (Loop & Calls)

Name	Syntax	Meaning
Loop	{	Push <here> on stack
Return	}	goto(pop stack)
Break	^	pop()
Skip	nS	Skip next n bytes
Quit	Q	Exit(b)

A new Language (Conditionals)

Name	Syntax	Meaning
If	$n?$	If false, skip n bytes.
Less	nL	If $\leq$, skip n bytes.
Equal	$n=$	$b \stackrel{?}{=} n$
Same	$n\sim$	$b \stackrel{?}{=} c$
Null	0	$c \stackrel{?}{=} 0x00$

A new Language (I/O features)

Name	Syntax	Meaning
Header	H	Write out program header
Get	<	$b \leftarrow \text{getByte}()$
Put	>	$\text{putByte}(b)$
String	\$	Write

A new Language (Variables)

Name	Syntax	Meaning
Here	$n.$	$a \leftarrow \langle \text{here} \rangle + n$
At	@	$c \leftarrow [a++] \text{ (Byte)}$
Advance	+	$a + \leftarrow c$
Minus	$n-$	$b - \leftarrow n$
Copy	C	$c \leftarrow b$
Times	*	$b * \leftarrow 16$
Plus	P	$b \leftarrow b + c$

Our new compiler

H ; print our header

{ ; loop

x02. ; a $\leftarrow$ &table (HERE+2)

nS ; skip over our table (2)

; The table below defines the symbols for our compiler

; They're of the form <symbol>,<n> (byte),<definition> (n bytes)

'H x54 H 'Q x04 Q '{x06 }'} x02 ' < x10 < ' > x10 > '= x02 = '? x05 ?

'S x02 S '. x05 . '@ x04 @ '^ x02 ^ '+ x02 + '~ x02 ~ '\$ x08 \$ '# x01 # 'O x02 O

x00 x00

Our new compiler (quote & exit)

< ; b ← read

x04 = x04? ; if not ^D, skip to quote check

Q ; exit (4)

" = x22? ; if not quote, skip to hex check

<> ; output(getByte()) (32)

} ; back to main loop (2)

Our new compiler (hex command)

'x = x4c? ; if not hex, skip to comment check

< '0 ~ ; b ← read(), (18) b ~ ← '0

x09 = x02L ; if b ≤ 9, skip 2 (6)

x07 ~ ; b ~ ← 7 (2)

* C ; b *= 16; c ← b (4)

< '0 ~ ; read second byte; (18)

x09 = x02L ; convert to hex digit (6)

x07 ~ ; like before (2)

P ; b ← b+c (2)

> ; write(b) (16)

} ; back to main loop (2)

Our new compiler (comment command)

' ; = x26? ; if not comment, skip to table lookup

{ ; loop (6)

< ; b ← read (16)

x0a = x04? ; if b = "\n"

^} ; go back to main loop (4)

} ; end loop (2)

Our new compiler (table lookup)

; table lookup

{ ; loop

@ ; $c \leftarrow [a++]$

0x04? ; if end of table,

^} ; go back to main loop

~x10? ; if $c \neq b$

@\$; $c \leftarrow [a++]$; write(address=a, length=c)

^} ; back to main loop

@ ; $c \leftarrow \text{length}$

+ ; next entry in table

} ; end loop

What Next?

- ◆ We can add new codes in terms of machine code.
- ◆ We can change existing code definitions the same way.
- ◆ We're not locked in to a single design, or a single decision; so we can always build whatever will make building the next step easier.
- ◆ The features we build first depend upon the features decide we will ultimately, and how much work they are to implement from scratch.
- ◆ Possible areas of improvement: Math features (division), memory access / management (variables; string handling), control structures (function calls; iterators; generators), I/O (system calls), Annotations(types, comments, whitespace)

- ◆ Appendix 1: ARM Thumb Code Example
- ◆ Disclaimer: I have only written this code; but I haven't tried to run it yet. I've hand assembled ARM Thumb code for a similar learning project before, though. I'm silly that way.

A Real World Example

- ◆ My cell phone runs Android (a version of Linux). Linux can run machine language programs stored as executable files with a valid ELF program header (according to the ELF standard at http://www.skyfree.org/linux/references/ELF_Format.pdf)
- ◆ My cell phone's CPU is an ARM CPU that's backward compatible with the ARM-7 "Thumb" Instruction set. (See <https://ece.uwaterloo.ca/~ece222/ARM/ARM7-TDMI-manual-pt3.pdf>)
- ◆ The Linux "syscall" manual page specifies that for ARM CPUs running the modern ARM Linux API ("arm/EABI"), system calls are done using the "software interrupt" instruction, with the system call number in register 7, and the result in register 0. (See <http://man7.org/linux/man-pages/man2/syscall.2.html>)
- ◆ The Linux system call table defines system calls `exit=1`, `read=3`, and `write=4` (See https://w3challs.com/syscalls/?arch=arm_thumb)

Instruction Set Summary for an ARM-7 compatible CPU

THUMB Instruction Set

Format Summary

The THUMB instruction set formats are shown in the following figure.

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
1	0	0	0	Op		Offset5					Rs		Rd				Move shifted register	
2	0	0	0	1	1	I	Op	Rn/offset3			Rs		Rd				Add/subtract	
3	0	0	1	Op		Rd			Offset8								Move/compare/add /subtract immediate	
4	0	1	0	0	0	0	Op			Rs		Rd				ALU operations		
5	0	1	0	0	0	1	Op	H1	H2	Rs/Hs		Rd/Hd				Hi register operations /branch exchange		
6	0	1	0	0	1	Rd			Word8								PC-relative load	
7	0	1	0	1	L	B	0	Ro			Rb		Rd				Load/store with register offset	
8	0	1	0	1	H	S	1	Ro			Rb		Rd				Load/store sign-extended byte/halfword	
9	0	1	1	B	L	Offset5					Rb		Rd				Load/store with immediate offset	
10	1	0	0	0	L	Offset5					Rb		Rd				Load/store halfword	
11	1	0	0	1	L	Rd			Word8								SP-relative load/store	
12	1	0	1	0	SP	Rd			Word8								Load address	
13	1	0	1	1	0	0	0	0	S	SWord7								Add offset to stack pointer
14	1	0	1	1	L	1	0	R	Rlist								Push/pop registers	
15	1	1	0	0	L	Rb			Rlist								Multiple load/store	
16	1	1	0	1	Cond					Soffset8								Conditional branch
17	1	1	0	1	1	1	1	1	Value8								Software Interrupt	
18	1	1	1	0	0	Offset11											Unconditional branch	
19	1	1	1	1	H	Offset											Long branch with link	
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		

Requirements Checklist (ARM Compiler) (page 1/2)

Requirement	Defined By
Header (with length)	Elf Header Specification
$\text{code} \leftarrow \text{getByte}()$	System call: read(code ,1)
putByte(code)	System call: write(code ,1)
if $\text{code} \neq \text{number}$ goto address	Compare Immediate Instruction; Conditional Branch Instruction
goto loop:	Unconditional Branch

Requirements CheckList: Arm Compiler (page 2/2)

Requirement	Defined By
<code>exit()</code>	System call: <code>exit(0)</code>
<code>address + 1</code>	Add/Subtract Instruction
<code>address ← byteCodeTable[n]</code>	Add Shifted Register Instruction
<code>address[0] ← code</code>	Load/Store Instruction
<code>address[0]</code>	Load/Store Instruction.
<code>byteCodeTable</code>	Elf Header: allocate extra 64K in-memory image for byte codes
<code>putBytes(address, length)</code>	System call: <code>write(address, length)</code>
<code>getBytes(address, length)</code>	System call: <code>read(address, length)</code>

Example Initial Compiler for an Android Cellphone (ARM-7 Thumb Instruction Set)

```
7f 45 4c 46 01 01 01 00 00 00 00 00 02 00 00 28
01 00 00 00 54 00 00 00 34 00 00 00 00 00 00 00
00 00 00 00 34 00 20 00 01 00 00 00 00 00 00 00
01 00 00 00 52 00 00 00 52 00 08 00 00 00 00 00
ff 00 00 00 ff 00 10 00 07 00 00 00 52 0
```

```
04 27 01 20 39 46 58 39 54 22 00 df 0d 18 ff 35
f7 46 22 e0 00 02 29 18 00 28 01 d1 01 27 00 df
01 28 0a d1 02 38 0c 1c f7 46 0f e0 21 1c 08 68
01 31 10 1c 03 27 00 20 00 df e4 ef 02 38 04 27
00 02 29 18 01 20 0a 60 01 31 00 df de ef 03 27
00 20 39 46 04 31 01 22 00 df 08 60 fe 46
```

“ELF Header”

“Program Header”

Write Header

Loop Exit

Define Quote

Write `getBytes`

ARM Thumb Instruction Set (ELF File Header)

- ♦ `7F 45 4c 46` # 0x7F"ELF" ELF identifier
- ♦ `01 01 01 00` # 32 bit, little endian, elf version 1, NO specific ABI
- ♦ `00 00 00 00` # 8 bytes of unused padding (for Linux)
- ♦ `02 00 00 28` # type 0x0002, machine=0x0028 (ARM)
- ♦ `01 00 00 00` # version of elf (1)
- ♦ `54 00 00 00` # entry_point_address # first 4 byte aligned entry in file
- ♦ `34 00 00 00` # program_header_table_offset: (immediately after ELF FILE header)
- ♦ `00 00 00 00` # no section header
- ♦ `00 00 00 00` # no flags
- ♦ `34 00 20 00` # program header offset(2), size of program header entry(32 bytes)
- ♦ `01 00 00 00` # number of program header entries(1), section header entry size
- ♦ `00 00 00 00` # number of section headers(0), section header with names(NONE)

Elf Program Header

- ◆ 01 00 00 00 # program type = load
- ◆ 52 00 00 00 # offset in the file image = 0x52 bytes
- ◆ 52 00 08 00 # (virtual address of segment in memory) # 0x80 0052
- ◆ 00 00 00 00 # (physical address (if relevant) (not)
- ◆ FF 00 00 00 # (size of bytes of the segment in the file image: 255 bytes)
- ◆ FF 00 10 00 # (size in bytes of the segment in memory: 64 K + 255 bytes)
- ◆ 07 00 00 00 # memory permissions (read, write, execute)
- ◆ 52 00 08 00 # ($p_vaddr = p_offset \% p_align$) (0,1 = no alignment)

Header, Loop & Exit

04 27 01 20 # r7 $\leftarrow$ 4 (write), r0 $\leftarrow$ 1 (to stdout)

39 46 58 39 # r1 $\leftarrow$ r15, r1 $\leftarrow$ header.len + 4 (&header)

54 22 00 DF # r2 $\leftarrow$ 0x54 (0x54 bytes), system call

0D 18 FF 35 # r5 $\leftarrow$ r1; r5 += 255 // r5 $\leftarrow$
&ByteCodeTable(load_address+0xFF)

loop:

F7 46 22 E0 # r14 $\leftarrow$ r15, b readByte: // b $\leftarrow$ readByte()

00 28 01 D1 # cmp r0,#0, bne on: // if b $\neq$ 0, goto define

00 27 00 DF # r7 $\leftarrow$ 1, syscall // exit(0)

Define

01 28 0a D1 # on: cmp r0,#1, bne quote // if b ≠ 1, goto quote

F7 46 22 E0 # b read Byte // code ← getByte()

02 38 00 02 # r0 -= 2 // code ← code - 2

29 18 f7 46 # r0 <= 8, r1 ← r5 + r0 // a ← &ByteCode[code]

22 E0 08 68 # b readByte,[r1] B ← r0 // l ← readByte; a[0] ← l

01 31 10 1C # r1 += 1, r2 ← r0 // x.length ← l

03 27 00 20 # r7 ← 3 (read), r0 ← 0(stdin) //

00 DF e4 EF # syscall, b loop

Quote & Write

02 28 06 D1 # cmp r0,#2, bne write:

f7 46 xx e0 # r14 $\leftarrow$ r15, b readByte

xx xx 01 20 # r1 $\leftarrow$ r0, r0 $\leftarrow$ 1,

01 22 04 27 # r2 $\leftarrow$ 1, r7 $\leftarrow$ 4

00 DF xx EF # syscall, b b loop

02 38 04 27 # write: r0 $\neq$ 2, r7 $\leftarrow$ 4 (write) // b = b - 2

02 00 29 18 # r0 $\ll$ 8, r1 $\leftarrow$ r5 + r0 // a $\leftarrow$ &ByteCode[b]

01 20 0A 60 # r0 $\leftarrow$ 1 (to stdout), r2 $\leftarrow$ [r1]B // l $\leftarrow$ x.Length

01 31 00 DF # r1 $\neq$ 1, syscall // a $\leftarrow$ x.value; write(stdout,a.value,l)

DE EF # b loop: // b loop

getBytes

FE B4 03 27 # Push r1-r7 onto r13, r7 $\leftarrow$ 3

00 20 69 46 # r0 $\leftarrow$ 0(stdin), r1 $\leftarrow$ r13

01 B4 01 22 # push r0 onto r13, r2 $\leftarrow$ 1

00 DF FF BC # sys call, pop r0-r7 from r13

FE 46 # r15 $\leftarrow$ r14

Table definitions for our symbol compiler (ARM-Thumb version) (p1/2)

00 27 00 DF ; Q (quit) $r7 \leftarrow 0$; system call

21 79 44 02 B4 ; { (loop) $r1 \leftarrow r15$; $r1 \neq 4$; push $r1$

00 BD ; } (return) pop $r15$

<code for getByte from first program> ; < (read)

FE B4 04 27 01 20 69 46 01 B4 01 22 00 DF FF BC ; > (write)

; push $r1 \leftarrow r7$; $r7 \leftarrow 4$ (write); $r0 \leftarrow 1$ (stdout); $r1 \leftarrow r13$; push $r0$; $r2 \leftarrow 1$; syscall; pop $r1-r7$

30 ; $n \neq$ (equals?) cmp $r0$, # n

D0 00 00 00 00 ; $n?$ (skip unless) be $n+4$; 4 pad bytes (no-op)

E0 00 00 00 00 ; nS (skip) b $n+4$; 4 pad bytes (no-op)

21 79 44 04 39 ; n . (here) $r1 \leftarrow n$; $r1 \neq r15$; $r1 \neq 4$

Table definitions for our symbol compiler(p2/2)

0A 78 ; @ (at) $r2 \leftarrow [r1]B; r1 += 1$

02 BC ; ^ (pop) pop r1

11 18 ; + (next) $r1 \leftarrow r2$

38 ; n- (minus) $r0 \leftarrow r - n$

FE B4 04 27 01 20 00 DF FE BC ; \$(string) push r1-r7; $r7 \leftarrow 4(\text{write})$; $r0 \leftarrow 1(\text{stdout})$; syscall; pop r1-r7

04 B4 01 BC ; C (copy) push r2; pop r0

DD ; nL (skip less) BLE $n + 4$

00 32 ; 0 (null?) cmp r2, #0

42 43 ; ~ (same?) cmp r0, r2

80 00 ; * (times) $r0 \ll \leftarrow 4$

01 18 ; P(plus) $r0 \leftarrow r2$

<header code from our first program> ; H (header)