How to Write a Plain English Compiler in Plain English

The Osmosian Order of Plain English Programmers

www.osmosian.com

GETTING STARTED

Well, it's just the three of us now – you, me, and the CAL-4700. I'm human, like you, so not a lot needs to be said about that. But the CAL is quite different from all others of his kind. In fact, the CAL was designed to make programmers ask questions that challenge almost every preconceived notion a modern practitioner might have. Questions like these:

Is it easier to program when you don't have to translate your natural-language thoughts into an alternate syntax? Can high-level languages like English be used to conveniently and efficiently write low-level programs like compilers? Are installation programs necessary? Can a workable interface be designed without icons, scroll bars, radio buttons, and a wide variety of other widgets? Can complex programs be clearly and concisely written without nested ifs and loops? Can a polymorphic drawing program be effectively programmed without objects? Can we do trigonometry using nothing but integers?

And one more:

Can natural languages be parsed in a relatively "sloppy" manner and still provide a stable enough environment for productive and reliable programming? In short, can we talk to computers like we talk to dogs, and still get a working system?

The CAL-4700 is the answer to all of these questions. You'll see that for yourself when you get to know him. This is where he lives:

www.osmosian.com/cal-4700.zip

DO THIS

Just download and unzip. Open the "documentation" folder and print the first 54 pages of the "instructions.pdf". Study them diligently, doing exactly what they say as you go along.

WHAT OUR COMPILER DOES

Our compiler converts English-language sentences (that we can easily understand) into oodles and oodles of binary digits called bits (that a computer running Windows can understand). This sentence, for example...

```
Beep.
```

...gets translated in to this string of bits:

```
11101000001011100000000000000000000000000
```

Since it's hard to read binary, we typically take just four bits, a "nibble" at a time, like this:

```
1110 1000 0010 1110 0000 0000 0000 0000 0000 0000
```

And then we write the result in hexadecimal, like so (one character per nibble):

```
E82E000000
```

(If you don't know how to convert binary to hex, look up "hexadecimal" in the Wikipedia. But don't spend all day on it - just get the gist and come back.)

The "E8" part of that hexadecimal string is called an "op code" - it tells the machine which operation to do. In this case, "E8" means "call the routine that is 2E000000 bytes (in hexadecimal) further along in memory."

Now if you type 2E000000 into a hexadecimal calculator and convert it to our human number system, you'll find it's a ridiculously large number: 771,751,936. But that's not the way Intel computers understand numbers like 2E000000, because Intel computers store numbers backways, byte by byte, in memory. Which means that 2E000000 is actually 0000002E, or a mere 46. If you start up the Windows calculator before you start up the CAL, you can flip back and forth between the two by holding the ALT key and pressing TAB.

Keep in mind that you don't have to worry about any of this: just remember that a byte is 8 bits, a nibble is 4 bits, and that Intel computers store numbers backways.

A LARGER EXAMPLE USING A VERY SMALL PROGRAM

Let's see what the CAL does with a real program.

Make a new program (a new directory) on your "C" drive called "a simple program". (The CAL's various "New" commands are under "N".) Then open that directory and make a new text file in it called "simple sentences". Finally, open that text file and type in this Plain English code, exactly as you see it here:

```
The sample number is a number equal to 4022250974.

To run:
Put the sample number into another number.
Buzz.

 \ noodle stuff below

A number has 4 bytes.

To initialize before run:

To put a number into another number:
Intel $8B8508000000. \ mov eax,[ebp+8] \ the number
Intel $8B00. \ mov eax,[eax]
Intel $8B9D0C000000. \ mov ebx,[ebp+12] \ the other number
Intel $8903. \ mov [ebx],eax

To buzz:
Call "kernel32.dll" "Beep" with 220 [hertz] and 200 [ms].

To finalize after run:
```

Note that we've borrowed some things from the CAL's noodle here, including a bit of Intel machine code (in hexadecimal) and the "buzz" routine that calls a routine that is in a DLL (dynamic link library) that comes with Windows. Normally we'd include the CAL's whole 15,000-line noodle in our program, but then this wouldn't be a simple program.

INTO THE BELLY OF THE BEAST

If you've typed the program correctly, it should compile and run and you should hear a beep when you select the Run command off the "R" menu (or just press CTRL-R or ALT-R).

Okay so far? Good. What isn't obvious is that the CAL created a new file in your program directory called "a simple program.exe". This file contains the oodles and oodles of bits that he generated, formatted in the special way that Windows likes. It's called the Portable Executable format, and it's horrendously (and unnecessarily) complicated. Look it up on Wikipedia if you want to see for yourself.

Fortunately, we Osmosians thrive on making things simple. As Antoine de Saint Exupery (the author of "The Little Prince") once said: "In anything at all, perfection is finally attained not when there is no longer anything to add, but when there is no longer anything to take away." I think it's ironic how wordy that statement is. Einstein said it more succinctly: "Make everything as simple as possible, but no simpler."

Turns out only five parts of a Portable Executable are actually necessary:

1. A tiny bit of the DOS Header (which is left over from days of old)
2. A few pieces of the PE Header
3. The IMPORT section (that lists the Windows' DLL routines we call)
4. The DATA section (that holds our global variables and literals)
5. The CODE section (that holds our code).

Let's take a look. I'm assuming you've got your simple sentences on the leftmost tab of the CAL's desktop. Click on the second tab, find your program directory, and open it up. You should see both "a simple program.exe" and your "simple sentences" in there. Now click on that ".exe" file and use the "Open as Dump" command on it:

```
00000000  4D 5A 00 00 00 00 00 00 00 00 00 00 00 00 00 00  MZ..............
00000010  00 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00  ........@.......
00000020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

The numbers on the left are addresses in the file (in hex); the numbers in the middle are the actual bytes in the file, sixteen at a time (also in hex). And the characters on the right are the sixteen printable equivalents of each of those bytes, if one exists.

GOING DEEPER

Let's go a bit deeper. What we're looking at is the DOS header, followed by the PE Header:

```
00000000  4D 5A 00 00 00 00 00 00 00 00 00 00 00 00 00 00  MZ..............
00000010  00 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00  ........@.......
00000020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000030  00 00 00 00 00 00 00 00 00 00 00 00 00 01 00 00  ................
00000040  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000050  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000060  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000070  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000080  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000090  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000000A0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000000B0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000000C0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000000D0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000000E0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000000F0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

```
00000100  50 45 00 00 4C 01 03 00 00 00 00 00 00 00 00 00  PE..L...........
00000110  00 00 00 00 E0 00 8E 81 0B 01 00 00 BC 00 00 00  ....à.Ž.....¼...
00000120  14 00 00 00 00 00 00 00 80 30 00 00 00 30 00 00  ........€0...0..
blah, blah, blah
000001F0  00 00 00 00 00 00 00 00 69 64 61 74 61 20 00 00  ........idata...
00000200  58 00 00 00 00 10 00 00 58 00 00 00 00 10 00 00  X.......X.......
00000210  00 00 00 00 00 00 00 00 00 00 00 00 40 00 00 C0  ............@..À
00000220  64 61 74 61 20 20 00 00 14 00 00 00 00 20 00 00  data........... 
00000230  14 00 00 00 00 20 00 00 00 00 00 00 00 00 00 00  ..... ..........
00000240  00 00 00 00 40 00 00 C0 63 6F 64 65 20 20 00 00  ....@..Àcode....
00000250  BC 00 00 00 00 30 00 00 BC 00 00 00 00 30 00 00  ¼....0..¼....0..
00000260  00 00 00 00 00 00 00 00 00 00 00 00 20 00 00 E0  ............ ..à
00000270  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000280  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

Note that most of the DOS header is zeros because most of it is unnecessary. The important parts are the "MZ" (which are the initials of Mark Zbikowski, the guy who came up with the DOS header), and the address of the PE header, backways, so Windows can find it.

The important parts of the PE header are the backways addresses of our IMPORT, DATA, and CODE Sections. You can jump to those places in the dump with the usual Find procedure: hit CTRL-HOME to get to the top, CTRL-F to start finding, then type the address you're looking for (frontways, like 1000 or 2000 or 3000).

At each of those locations you should see some non-zero bytes: DLL names near the first, DEADBEEF near the second, and some machine code at the third.

WHAT WAS THE CAL THINKING?

To get from the simple sentences of our little program to that Portable Executable, the CAL had to do a lot of thinking. We can find out what he was thinking by asking him to list the thoughts he thought as he went along translating the one to the other. So let's do that.

Click on your leftmost tab (the one with the simple sentences on it) and execute the List command from the "L" menu. Now click on the third tab and open up your program directory again. You should have three files in there now:

```
c:\a simple program\a simple program.exe
c:\a simple program\a simple program.lst
c:\a simple program\simple sentences
```

Double-click that ".lst" file to open it up (it's just text) and you'll see a lot of gobbledygook that starts like this:

```
TYPES:

/type/byte/bytes/00000001/byte/byte///0/

/type/record/records/00000000/record/record///0/

/type/number/numbers/00000004/record/record///0/
/variable/field/no///byte/byte/00000000/no/4/no////

GLOBALS:

/variable/global/yes/sample number//number/number/00402000/no/1/no//~L1//
```

We taught the CAL, when we were developing him, to list his thoughts this way so we could make sure he was working properly and efficiently. There are eight sections in a listing: TYPES, GLOBALS, LITERALS, ROUTINES, INDICES, IMPORTS, SOURCE FILES, and TIMERS. You can find them all simply by looking for colons in the listing file (CTRL-HOME, CTRL-F, type a colon, then use CTRL-N to find the next, and the next, etc). Note that the CAL has lots of indices (to make him fast), and lots of timers (to make sure he is).

THREE THINGS

All programming languages need three, and only three, kinds of things: TYPES, VARIABLES, and ROUTINES. The rest is decoration.

TYPE definitions describe the different KINDS of data the programmer wants work with: what they're called, how big they are, and what, if any, parts they have. Here's a sample from our simple sentences:

```
A number has 4 bytes.
```

VARIABLE definitions describe ACTUAL PIECES of data the the programmer wants to work with: what they're called, what type they are, and sometimes an initial value, like this:

```
The sample number is a number equal to 4022250974.
```

ROUTINE definitions describe what the programmer wants to do with his variables. Every program needs at least one routine (so the computer will know where to start). Here's ours:

```
To run:
Put the sample number into another number.
Buzz.
```

Routines can call upon one another when they need help. In such a case, the routine that is called becomes a SUB-ROUTINE of the routine that is doing the calling. The first sentence in the "run" routine above, for example, calls on the "put a number into another number" noodle routine for help, passing two variables for him to work on: "the sample number" and "another number". When we pass a variable to another routine, we call it a PARAMETER.

The second sentence in the "run" routine above also asks for help, this time from the noodle routine called "buzz", but doesn't pass any parameters.

Note that the CAL always passes the address of a variable as a parameter, not the variable's contents (or value). The called routine therefore gets to work on the original variable, and not just a copy. Some languages do it the other way (passing a value rather than an address), and some are goofy, passing an address in one case, a value in another.

THREE KINDS OF CODE

Let's take a closer look at the routines in our simple sentences file:

```
To run:
Put the sample number into another number.
Buzz.
```

The "run" routine, above, is written entirely in English. This routine, on the other hand...

```
To put a number into another number:
Intel $8B8508000000. \ mov eax,[ebp+8] \ the number
Intel $8B00. \ mov eax,[eax]
Intel $8B9D0C000000. \ mov ebx,[ebp+12] \ the other number
Intel $8903. \ mov [ebx],eax
```

...is a hybrid with an English-language header and an Intel machine-code body. This is a bottom-level routine, describing, in hexadecimal, the exact instructions we want in the executable file. There are a handful of such routines in the CAL's Noodle, but a typical Plain English programmer will never need to write such gobbledygook.

Nor will a typical Plain English programmer ever need to write the goofy kind of stuff we see in the "buzz" routine:

```
To buzz:
Call "kernel32.dll" "Beep" with 220 [hertz] and 200 [ms].
```

This routine is a hybrid of English and Windows-Speak. It calls a routine called "Beep" that lives in the "kernel32.dll", passing two literal parameters (variables): 220 and 200. The CAL sticks those literals in our Portable Executable's Data Section, as variables, but since Windows is goofy and wants some parameters passed as values (and others passed as addresses), the CAL makes a necessary exception in this case and passes the contents of those variables, rather than their addresses, to the Windows' "Beep" routine.

Note that Windows is also goofy in that it considers "Beep" (with a capital "B") and "beep" (with a lowercase "b") to be two different words. It's like they WANT things to be difficult.

REGISTERS AND THE STACK

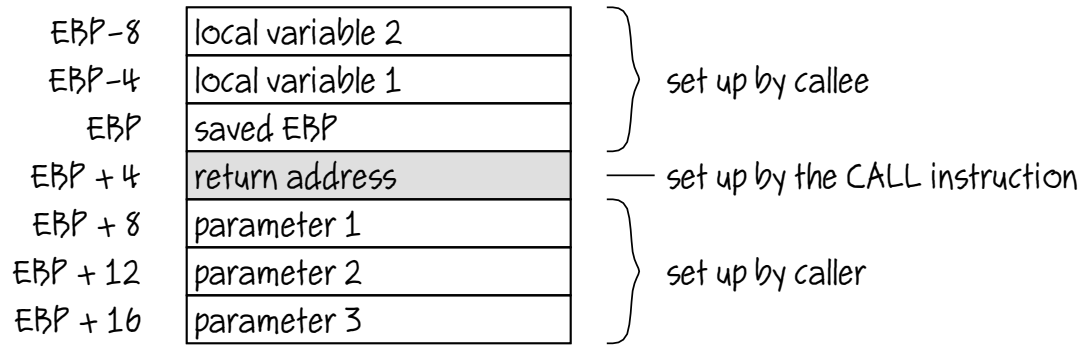
Intel processors have a handful of registers, each 4-bytes long. Here are the ones we use:

EAX	EBX	ECX	EDX	EBP	ESP

The EAX, EBX, ECX, and EDX registers are general-purpose, more or less, and most of the Intel's instructions operate on (or with) one or more of them.

The EBP and ESP registers, on the other hand, are used to manipulate the STACK which is an area of memory that you can think of like a stack of dinner plates, with each plate representing a saved register value, or an address somewhere in the program's code, or the address of a variable (that lives somewhere else in memory), or the contents of a variable. As is typical with Windows, the stack grows backways: that is, the address of the "top" of the stack is always less than the address of the "bottom".

The ESP register always holds the address of the "top" of the stack. The EBP register holds the address of the "bottom" of the stack for the current routine. By convention, this is how the stack is used:



You can see that, once things are set up, our first parameter will be at the address in the EBP register plus 8, the second parameter at the address in EBP plus 12, etc. And that our local variables will be a the address in the EBP register minus 4, minus 8, etc.

The Intel knows how to PUSH 4-byte values and addresses onto the stack, and how to POP them back off. To get larger things on and off the stack we need to play with the ESP register ourselves. Subtracting 16 from the address in ESP, for example, makes room for 16 bytes of local data at the "top" of the stack; adding 16 takes those bytes off the stack.

A CLOSER LOOK AT PARAMETER PASSING

Find the "run" routine in our listing by looking for "/run/" (without the quotes). Then focus on these three lines:

```
/fragment/push address/other number////00000000/0040300D/8D95FCFFFFFF52/
/fragment/push address/sample number////00000000/00403014/C7C20020400052/
/fragment/call internal///put [number] in/into/to [number]//00000000/0040301B/E818000000/
```

Each of these is a fragment of the code that the CAL will be putting in the executable file. The first fragment pushes the address of "the other number" onto the stack. The second fragment pushes the address of "the sample number" on top of that. And the third fragment calls our internal "put [number] in/into/to [number]" routine. The Intel machine code that does each of these things is shown at the end of each line.

When the "put a number into another number" routine gets control, the top of the stack will look like this:

return address
address of the sample number
address of the other number

Now find the "buzz" routine and focus on these three lines:

```
/fragment/push value/~L3////00000000/00403057/C7C20C204000FF32/
/fragment/push value/~L2////00000000/0040305F/C7C208204000FF32/
/fragment/call external////Beep/00000000/00403067/FF1528104000/
```

This time, the CAL is pushing values (instead of addresses) and calling an external, rather than an internal, routine. So when the "Beep" routine in the Windows' "kernel32.dll" gets control, the top of the stack will look like this:

return address
220
200

Note that in both cases, the parameters are pushed backways so the leftmost parameter in the source sentence ends up on top.

LOCAL VARIABLES

Let's take a closer look at this sentence:

```
Put the sample number into another number.
```

We know where "the sample number" is, because we defined it earlier as a GLOBAL variable:

```
The sample number is a number equal to 4022250974.
```

You can see it in the Data Section of our Portable Executable file at address 2000 (just click on your second tab, find "2000"). The initial value literal occupies the next four bytes:

```
00001FF0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00002000  00 00 00 00 DE AD BE EF DC 00 00 00 C8 00 00 00  ....Þ¾ïÜ...È...
00002010  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

It looks like "DEADBEEF" because the number 4022250974, in backways hex, is just that.

Back to our original sentence. We know where "the sample number" is, but where is "the other number"? On the stack, just above the saved EBP, because it's a LOCAL variable.

the other number
saved EBP
return address

The CAL recognized this variable as a LOCAL variable (because of the indefinite article "another" preceding it in the source sentence, and remembered this, and made room for it:

```
/routine/run/yes/no/no/no//0/4/00403000/
/variable/local/yes/other number/other/number/number/FFFFFFFC/no/1/no////
/fragment/prolog/////00000000/00403000/558BEC8901000000 6A004976FB/
---
/fragment/epilog/////00000000/00403025/8BE55DC200000000/
```

The PROLOG of every routine begins by saving the caller's EBP on the stack, just above the return address. In this case, the PROLOG also pushes four zero bytes on top of that to make room for "the other number". The EPILOG of every routine removes any local variables, restores the EBP, and returns to the caller, popping any parameters as he does.

STEP BY STEP

Enough of the gobbledygook.

Let's start at the top and see exactly how the CAL translates Plain English sentences into executable code. It all begins (and ends) with the routine you see below.

Whenever you select the Compile, List, or Run command from the menu, the Desktop calls this routine, passing the full path of the directory containing the source file that is open on the current tab.

```
To compile a directory:
Compile the directory (start).
Compile the directory (load the source files).
Compile the directory (scan the source files).
Compile the directory (resolve the types).
Compile the directory (resolve the globals).
Compile the directory (compile the headers of the routines).
Compile the directory (calculate lengths and offsets of types).
Compile the directory (add the built-in memory routines).
Compile the directory (index the routines for utility use).
Compile the directory (compile the bodies of the routines).
Compile the directory (add and compile the built-in startup routine).
Compile the directory (offset parameters and variables).
Compile the directory (address).
Compile the directory (transmogrify).
Compile the directory (link).
Compile the directory (write the exe).
Compile the directory (stop).
```

As you can see, there are 17 steps in the process.

Some are trivial, like (start) and (stop). Others are less trivial, like (compile the bodies of the routines), and some are downright mysterious, like (transmogrify). But none of them are difficult, if we take things a step at a time – which is what we're going to do on the following pages.