What's a Language?

Wake up, Neo.

Wake up.. and smell the ashes.

Low Level Languages

In “Thinking Forth”, Leo Brodie writes:

> ...traditional high-level languages broke away from assembly-language by eliminating not only the one-for-one correspondence between commands and machine operations, but also the linear correspondence.

Assembly-language is a low level language. A computer runs instructions on its CPU. In assembly, the instructions look like this:

```
LOAD 5 into A                    ; A is a CPU register
LOAD 6 into X                    ; X is a CPU register
ADD A and X                      ; This stores the result in A
STORE A into Location 49152      ; This stores the value in A to memory
```

Each one of the above are state machine instructions. The CPU is, essentially, an atomic state machine. Commands execute one by one, and they alter the state of the CPU. Looking at the above example you may wonder why the third instruction doesn't just store the result directly to memory. Well, it could – if it was designed that way. but ultimately, it would be like executing two instructions at once. And sometimes you don't want to do that. Keeping ADD and STORE separate is worth more than adding an extra instruction that is essentially superfluous. Now, if you had a set of 10, 20 or 30 instructions (or more!) that were needed for a common task, this is a prime canidate for a new opcode. Here's an example from the 6502:

```
; Multiply A × X → 16-bit result in factor2 (high), factor1 (low)
; By Leif Stensson, ~130 cycles average

    STA factor1     ; store first factor
    LDA #0          ; clear high byte of result
    LDX #8          ; 8 bits to process
    LSR factor1     ; shift low bit into carry
loop:
    BCC no_add      ; if bit was 0, skip add
    CLC
    ADC factor2     ; add second factor
no_add:
    ROR             ; rotate result right (high byte)
```

```
        ROR factor1     ; rotate result right (low byte)
        DEX
        BNE loop
        STA factor2     ; store high byte
        RTS
```

That's 13 instructions, 130 cycles, two zero-page temporaries, for an 8×8 = 16-bit multiply. The 6502 had to bit-shift and conditionally add 8 times to produce the answer. Divide is even worse; the 6502 needs a similar loop for division. Earlier, the 8086 did have a MUL, but it was in microcode and took 70 to 80 cycles. The 6809 had a MUL that took 11 cycles, groundbreaking at the time. Later the SNES added an external math chip that could do 8 bit MUL in 8 cycles, but it was memory mapped so it was constrained by the need to read and write to memory. Even so, all of these methods indicated the need for a true MUL (and DIV). By the 68000 era (the 80s and early 90s), 32 bit MUL was available in just 28 cycles (on the 68020). The older 8 bit chips started to die out as a result of this and other improvements. The 80286 had a 21 cycle 16×16 MUL that fed into a 32 bit register. ARM2 and the 386 iterated on this, doing 32×32 into a 64 bit result in just ~16 cycles.

This is important because MUL, itself, is an abstraction. It's a new function; a new word. Assembly language, the language of the CPU, ***is*** *a programming language.* It's just a very low-level one.

What's a (high level) language?

A “high level” programming language attempts to abstract assembly-language by leveraging certain mathematical truths *about assembly language* into data structures (or vice versa).

The purpose is simple. We don't want to think like a CPU. We want to think like a human being. And therein lies the fruit at the center of the Garden of Eden. If abstraction is turning the computer's thoughts into our words, why do we write languages that turn our words into the computer's thoughts? Computers don't think like humans. Making a computer think like a human, will always dull the ability of the computer to carry out human instructions. The best programmers are **always** the ones that learn to think like a computer. Like good 'ol Mel.

“But”, I hear you say.

Not everyone has the time, the drive, the energy, to become *that* involved. Truth be told, even among those who call ourselves “professional programmers”, as if programming were anything other than a game, a trifle – few and far between of us really care very much about computers. We're too busy programming in Java, Swift, Python or something even crazier (Perl?). The forth wall? Let's look at BASIC. Basic in its crudest form has 26 one-letter integer variables. These act like registers, but they're not. What you could do instead is PEEK and POKE to store variables, and use the letters as temporary storage. In this sense, the letter variables act like registers on a CPU.

Let's look at FORTH. Forth uses a stack. The CPU has a stack too. Forth has two stacks. Well, the thing is, the CPU *has* a stack, but it doesn't have to use it. Forth abstracts itself around the stack as a means to avoid having to deal with the state machine of the CPU. It's a very powerful paradigm, but it is utterly ridiculous that holy wars exist in the Forth community over things like local variables. A CPU lives and breathes on being able to LOAD and STORE anywhere in memory. Why take that power away from the programmer? In fact, you can't. You can define your own local variables in Forth if you

really wanted to. To those who complain about this, aren't you really just complaining that Forth is too versatile? Who was the marketing genius who once said Forth is so powerful you can even redefine the value of integers? Oh right, the same guy that invented Forth!

Let's look at Python. Python does not have a GOTO. Frankly, JUMP and BRANCH in it's various forms are staples of any CPU instruction set. You cannot write practical code without branches and jumps. What Python and other languages that remove goto set out to do is restrict what kind of loops you can write. They want to control the flow of the program along some ordered thread. Forth is like this too with it's stack; it ties the stack and flow control together.

Look at Haskell. In Haskell, expressions are not evaluated when they are executed, but only when their value is actually needed to produce the final result.

Why?

At some point, you have to step back and ask yourself, why? Why is everyone trying to hide the truth? GOTO exists. Someone, somewhere, somehow, has to understand what a GOTO is. A CPU cannot plausibly exist without it. A CPU is a state machine.

Look at C++, Java and some others.

```
var a = b + c.d
```

c.d might call a function. That's hidden control flow. Why are people trying to make you think in strange ways that a) aren't natural anyways and b) aren't in-line with the CPU?

Ultimately, learning about programming and learning about the CPU is the responsibility of every programmer. Look at the legendary programmers. Mel, the greatest coder who ever lived, earned that title because he understood the hardware.

There's just one problem. Assembly kind of sucks. I mean, it takes a long time to do anything. And here's the secret; you have to come up with your own convention to do anything. It doesn't have to be the same convention all the time – that's the mistake most “modern” programming languages make. They try to follow the *convention of the week.* But the thing is, conventions are tools. They are problem-bound (problem-specific). The power of raw assembly is that you aren't bound to a convention.

Rising above the convention of the CPU state machine

TO rise above, if one must, means that one must not /remain *bound by the CPU state machine. One may still optimize by serving the CPU. But it must no longer be the primary paradigm. In this sense, all of the bounds of the CPU must be immediately abstracted.*

1. Bytes are integers. Words are longer integers. Long integers are 32 bit. And so forth. u64, u128, don't mind if I do. But these require direct memory access. That's it. On one side, you can say that direct memory access is the programmer's responsibility. On the other hand, you could abstract it with labels. The label abstraction can be done in two ways. One is to define new registers, such as a two-letter register system, or, an ID system say of 256 registers qualified by the value in a byte). Having 256 integer variables available at any time is good enough for most applications–we can worry about advanced data structures in a moment.

2. Strings are pointers. The difficulty with strings versus integers is that they ultimately require different kinds of interface. You don't really want to mix all variables into a table, although it is very convenient! So maybe do it. Its a simple system. A pointer points to a string; based on this they system can tell you what kind of variable it is (type) and where it is (or, its value), its width (in bits) or its length, and so on. It's just a helper library. Like how BASIC inserts and deletes line numbers.

3. Memory management. Memory does not understand what you want to put in it. CPUs do not understand STRINGS. Strings are conventions. A 'string' of bytes terminated by a zero has no intrinsic meaning to a CPU. It's your convention. You have to keep track of it, by convention. The zero at the end isn't enough to associate a string with a label (also, itself, a string). ultimately you need a table for this (or a dictionary). Some data structure by convention. At that means, certain regions of memory will be marked OPEN or CLOSED – FREE or USED – and dealt with accordingly.

Tying it all together

Known as “threading”. Forth changed how I thought about languages because of it's innovative use of a dictionary structure. BASIC and Python, in contrast, are interpreted languages. You sit there and write a parser for the *text*, to then run commands by calling a function inside the interpreter. Ultimately, you have created a state machine and are interpreting commands which modify that state machine. If it's just “commands” it might be like a script or a macro language. But most languages like BASIC or Python are really state machines. They maintain lists of variables. They have expression parsers, and so forth.

Forth on the other hand, has a “dictionary” of words that are “chained” together. This means adding a new world is trivial, it just chains it to the end of the dictionary like a linked list. And how does it “compile” these “words”? Simple. It writes CALL functions to the addresses of native words. A set of such CALLs is a function definition (written out of other Forth words). This is called “subroutine threaded compilation”. You can also inline some functions by directly copying the function into the binary versus writing a CALL to the runtime. In that sense, Forth is a kind of bytecode– but one in which you can add your own bytes to the code. Forth itself is a state machine as well, since it maintains its stacks, the depth of them, and maintains a dictionary. The native words in the dictionary are part of it's initial state. The construction of the apparatus which holds them is also known as a 'state', in terms of a 'state machine'.

So we see, that in the design and construction of a programming language, what is very important is the 'state machine' you have chosen to adopt, or rather, the state machine you have chosen to replace the CPU. So you don't have to actually understand the CPU.

More RANTING

On Page 53-54 of “Thinking in Forth”, Leo Brodie writes:

```
  Structured English is a sort of structured pseudocode in which our rate
state-
  ment would read something like this:
  IF full rate
     ( code block )
  ELSE ( not-full-rate)
```

```
    ( code block )
  THEN
    ( code block )
```

His conclusion on the matter is straightforward:

```
  This is just plain awkward. It’s hard to read, harder to maintain, and
hardest
  to write. And for all that, it’s worthless at implementation time. I don’t
even
  want to talk about it anymore.
```

Okay, but this is *exactly* the analogy Brodie uses to explain what programming is on page 8-11, and in page 19 he dares to write this:

```
  Here’s an example of Forth code;
  : BREAKFAST
      HURRIED? IF CEREAL ELSE EGGS THEN CLEAN ;
  This is structurally identical to the procedure MAKE-BREAKFAST on page 8.
```

Lest we accuse Brodie of not being self-aware as a sort of cute excuse (“We still love you Brodie!”) on page 230 to 232 he explains how to factor out nested conditionals. What he doesn't tell you is what the tradeoff is.

The tradeoff is … *nothing.* You still have to nest conditionals as the best-case solution sometimes (see pg. 235). Nothing will stop you from nesting conditionals. Ever.

In case you missed it, here's what Brodie says is the solution:

```
  : PROCESS-KEY
      KEY DUP LEFT-ARROW = IF CURSOR-LEFT ELSE
          DUP RIGHT-ARROW = IF CURSOR-RIGHT ELSE
          DUP UP-ARROW = IF CURSOR-UP ELSE
                           CURSOR-DOWN
          THEN THEN THEN DROP ;
```

Here, let me refresh your memory:

```
  IF full rate              \ This is just plain awkward.
     ( code block )         \ It’s hard to read, harder to maintain,
  ELSE ( not-full-rate)     \ and hardest    to write.
     ( code block )         \  And for all that,
  THEN                      \ it’s worthless at implementation time.
     ( code block )         \  I don’t even want to talk about it anymore.
```

On achieving simplicity, Brodie writes “Given two solutions to a problem, the correct one is the simpler.” (ibid, pg. 60). I ask you, which one is easier to understand? The nested ifs, or the other nested ifs?

What is going on here is simple. You can factor out nested ifs but all you are doing is naming the loops; essentially you are tying the execution of a loop to the comment that describes what it does. if

every word in FORTH was three random letters, the program would still work. You just wouldn't be able to understand it. But that's the same in any language. Comments matter; well-named functions and variables matter. Requiring you to name your inner loops does not mean you do not have inner loops. But introducing that into the execution cycle will always slow down your program.

No, the way it should be is both ways should be available. You should be free to use one way or another, and it should compile to the same code. Then again, that's exactly what comments are. 100% pure syntax sugar. Thats for the humans. The abstractions are for us. For the humans. The code should align with the state machine. Brodie is right, but Brodie is also wrong. The fact that Brodie “doesn't want to talk about it anymore” is fine. No one is forcing him to eat food he doesn't like. It's a personal choice. But if Brodie wants to say that his favorite food is everyone's favorite food he is wrong.

Your favorite food is all well and good, but it is food you must eat and in that there is no choice, only the humbling acceptation of reality that we are all human. Yes, we are human. We must interact with the world. Asking the world to interact with us, as if it owes us a living, is a strange concept. The new language must operate in accord with the CPU. This is one thing Forth got very very correct. STC and direct threading (i.e. inlining) is the way to go. The issue is how much freedom you have in the chunks (words). You can define your own words – thats amazing. But you're locked into the little Cage Forth puts you in – you HAVE TO use the stack machine.

Lorthio, Forth's Evil Twin

I will now introduce you to Lorthio. Lorthio wears a hat with an upside-down F on it. He is Forth's evil twin. His name is not actually Lorthio. You can't pronounce his name in the normal language. We had to invent a new letter. And that is exactly what we will do today. Today, we are going to set Forth on it's head and do things our way. Because excuse me, Brodie, Sir, this is a Wendy's.

BASIC	POKE 49152, 10
Forth	10 49152 POKE
Lorth	POKE 49152, 10

This is a pretty accurate statement. IN forth, you can write

```
10 49152 C!
```

Therefore,

```
: POKE C! ;
```

This POKE adds nothing. It's an alias. 10 49152 C! already means “store 10 at 49152.” The stack order matches; value then address, which is what C! expects. Getting fancy, you could write it as:

```
49152 10 POKE
```

This would be:

```
: POKE SWAP C! ;
```

But now let's bring Lorth in on this. Let's do this:

```
: POKE WORD NUMBER WORD NUMBER C! ;
```

Now suddenly we can do:

```
POKE 10 49152
```

But we want it in BASIC order, so we add a swap:

```
: POKE WORD NUMBER WORD NUMBER SWAP C! ;
```

Now we can do:

```
POKE 49152 10         \ This is valid Forth
```

But Lorthio is not done yet. After all, Lorthio is an evil genius, bent on taking over the earth. Let's try this:

```
: STRIP, DUP
  BEGIN
    DUP C@ 0= IF DROP EXIT THEN
    DUP C@ 44 = IF 0 OVER C! THEN
    1 +
  AGAIN ;
: ARG WORD STRIP, NUMBER ;
: POKE ARG ARG SWAP C! ;
POKE 49152, 10       \ Now this is valid Forth.
```

Lorthio lives!

```
: X1 ARG VAR_X1 ! ;
: Y1 ARG VAR_X2 ! ;
: X2 ARG VAR_X3 ! ;
: Y2 ARG VAR_X4 ! ;
: COLOR ARG VAR_A5 ! ;
```

```
: DRAWLINE X1 Y1 X2 Y2 COLOR
  A1 @ A2 @ A3 @ A4 @ A5 @
  ( now stack has x1 y1 x2 y2 color )
  ( ... draw the line ... )
;
```

```
DRAWLINE 50, 10, 100, 20, 1
```

This is valid Forth. But Lorthio is not done yet!

```
: SPLITARG
  WORD DUP
  BEGIN
```

```
    DUP C@ 0= IF DROP NUMBER EXIT THEN
    DUP C@ 44 = IF
      0 OVER C!           \ null-terminate first part
      1 +                  \ pointer to second part
      SWAP NUMBER          \ convert first part
      SWAP NUMBER          \ convert second part
      EXIT
    THEN
    1 +
  AGAIN ;
: POKE SPLITARG SWAP C! ;
POKE 49152,10        \ Now this is valid Forth too.
```

Lorthio will never stop laughing. And one day, if he works hard enough, he can make FORTH run BASIC.

```
VARIABLE LABEL
: GOTO! LABEL ! ;
: @LABEL LABEL @ ;
: GAME
  1 GOTO!
  BEGIN
    @LABEL 1 = IF ." Room 1" CR 2 GOTO! THEN
    @LABEL 2 = IF ." Room 2" CR 3 GOTO! THEN
    @LABEL 3 = IF ." Done" CR EXIT THEN
  AGAIN ;
GAME
```

but, Lorthio does, ultimately, have his limits. Forth is, after all, Forth. You simply cannot, ultimately, GOTO. The reason why is because a Forth program has two parts; the ": ;" blocks which are compiled words, and the program itself which is interpreted. Even if you could somehow define a back-jumping label system its just a kind of loop. Nothing special. No, a real forth hacker has to break out of the system. A real forth hacker has to stand up and say, "I don't like the idea that someone else is in control of my life." What is the Matrix? A real Forth hacker has to implement Forth. "To understand forth, you must implement Forth." –Charles H. Moore.

"Okay," says Lorthio with his trademark Evil Laugh™, "Challenge accepted".

```
  dict_fgoto:
      .bytes $00, $00, $00
      .bytes $85                  ; IMMEDIATE + len=5
      .bytes "FGOTO", 0
  code_fgoto:
      LDGLZ [@FORTH_HERE]
      LDKL @OP_JMP
      STKL [GLZ,+]
      ; Push placeholder address onto data stack
      STAB [-,ELY]
      MOV A, Z
      MOV BL, GL
```

```
        LDBH #0
        ; Write zero placeholder
        LDKL #0
        STKL [GLZ,+]
        STKL [GLZ,+]
        STKL [GLZ,+]
        STGLZ [@FORTH_HERE]
        RET
    dict_flabel:
        .bytes $00, $00, $00
        .bytes $86                  ; IMMEDIATE + len=6
        .bytes "FLABEL", 0
    code_flabel:
        ; Pop placeholder address
        MOV GLZ, BLA
        LDAB [ELY,+]               ; restore previous TOS
        ; Write current HERE into placeholder
        LDTLK [@FORTH_HERE]
        STTLK [GLZ]
        RET
```

And just like that,

```
: GAME
  FGOTO skip
  ." You never see this" CR
  FLABEL skip
  ." You see this" CR
;
GAME
```

… that's game. Lorthio wins. The universe, broken, disappears in a puff of smoke.

And yet we are all still here. Look , none of this is a rant against Forth. We all need some kind of structure in our lives. I am just saying, that structure has to be useful. Truth must have a value proposition. I have a feeling that Forth is one of the all-time most important computer languges. Java (i.e. bytecode), Python is essentially BASIC, so BASIC is another. C is another, as is C++. LISP. Forth. All important. All have meaning, history, and culture. I just get the feeling there's room for something else. Something more. They recently made Zig. Zig looks like a step in the right direction. Rust? No, seriously, no. But whatever the thing is, the new thing, it's coming.

“I have seen it.” – Emperor Palpatine.

Now Let's Have Fun

Strings.

Chuck Moore said,

> “What does a character string look like? Of all the ways you might choose, one is completely natural:
>
> “ABCDEF…XYZ”
>
> A character string is enclosed in quotes. It can contain any character except a quote, specifically including spaces.”
>
> *Q: Why not just have the parser scan from opening “ to closing ” and push the whole string onto the stack (or leave it in the input buffer as a literal)?*
>
> “We get in trouble immediately! How do you recognize a character string? By the leading quote, of course. But do you modify your word subroutine to recognize that quote? If you do so you may never use a leading quote for any other purpose. Much better that the quote is a word by itself, treated like any other dictionary entry, for it can then be re-defined. But words are terminated by spaces, and I still resist making quote an exception. So let's type character strings:
>
> ” ABCDEF…XYZ“”

In other words, Forth's outer interpreter is deliberately the simplest possible thing: it reads whitespace-separated tokens (via the WORD primitive or equivalent) and looks them up in the dictionary (or treats them as numbers).

Making “ a special syntactic delimiter that triggers automatic scanning to the matching ” would require hacking the core tokenizer with an exception. Moore hated exceptions because they break re-define-ability, extensibility, and the “everything is just a dictionary word” model.

Instead, “ (or more precisely in modern ANS Forth, the word S”) is itself a regular dictionary word. When executed or compiled, it does the special parsing: it changes the delimiter temporarily to “ , scans the following text, and leaves a string descriptor(c-addr u)—address and length—on the data stack. The actual characters stay in memory (usually compiled into the dictionary for literals, or in a temporary area like PAD).

```
  S" Hello, world!" TYPE
```

(with a space after S”) or, inside a definition:

```
  : GREET   S" Hello!" TYPE ;
```

The same principle applies to .“ (dot-quote) for printing.

This design choice has been echoed and analyzed ever since in Forth literature, tutorials, and discussions (e.g., Starting Forth, GForth manual, comp.lang.forth threads, and modern explanations like Dave's ratfactor.com article). People sometimes complain that Forth's string handling feels “clunky” compared to languages with built-in ”…“ syntax, but the counter-argument is always the same: any more “convenient” literal syntax would have violated the minimalism that makes Forth so small, fast, and hackable. (Some Forths later added string stacks or other conveniences precisely because (addr len) pairs are powerful but manual.)

Moore's later colorForth took an even more radical step—using color to tag strings instead of quotes

at all—but the 1970 reasoning above is the canonical answer to “why can't you just scan from ” to ”?“ It's one of the purest examples of Forth's “keep the core stupid-simple” ethos.

The Truth

“But words are terminated by spaces, and I still resist making quote an exception.” –Charles Moore

Here's the problem though, Strings *are* and exception, and strings do not always contain words. Sometimes, strings contain control characters like 10 and 13 (CR and LF). Sometimes others. Also, apparently Mr. Moore was more human than some of us would like to admit, since he obviously didn't realize you can just make a word that pulls in words after it. S” could have picked up everything until the next “. Saying that “you can't have a word that begins with…” is bad language. It's like saying the words benediction, benefit and beneficial are not related. They all mean “good”– *bene-*. The inability of Forth to realize this, or at the very least, the inability of you to define this, is a weakness of Forth. It restrains you. Forth is not complete without a “begins like” definition. If a word doesn't match an exact word, does it at least begin like something?

Sometimes you just need to add data. Not everything is a file. Sometimes, one file programs – aka binaries – are just easier to distribute. Sometimes you want to compile things. Not everyone wants to release source code. Ultimately, saying “this is the real world” sounds trite – and it is not being said to induce *peer pressure* to do something bad (like lazy resolution, or wrapped nulls) but it is being said because Forth fell straight back into the old trap of trying to make computers think like humans. Forthers found a neat way of using the stack – which is valid – but then assumed that was the only way to do things. They are wrong. It's not the only way. It's a way that actually can't be used for some applications. Forcing it to work is a kludge. Whether it works or not.

But what is Forth?

Lothorio whispered something in my ear. He told me that there is absolutely nothing wrong with a word that pulls in words until it reaches a ” at the end of a word. Then again, that's Lothorio speaking. Why should we trust that ne'er-do-well?

There's nothing wrong with being able to define data in a practical way. This is actually a mistake Moore made with Forth; he said “words are terminated by spaces”. Actually no, they aren't always terminated by spaces. It's not that Forth lacks a way for you to chunk off memory and have a series of string pointers. It's that it ignores the need to do so. It's not that you can't do it, it's that you get no help whatsoever in doing so, and the people teaching the language try to talk you out of doing this. Well, the truth is, strings are not words, strings are not numbers. Yes, strings need special processing. LDA #50 is not a word; it's a convention that speaks to three bytes of data; LOAD, A, and 50. The trouble with Forth is people don't want to write LOAD A 50 (or 50 A LOAD, which is precisely Forth). They want to write LDA #50.

The other thing is, the Dictionary is cool, but throwing everything onto a dictionary (or onto a stack) is obtusely simple. Returning to the idea that strings are not numbers, there *should* be an exception for them.

Or else people will just create a series of words; words that allow you to store and manage integers. *Characters in an array.* And a series of words to operate on those arrays. If you think this is the simple

solution, it isn't. It's what people did before they had MUL. No, saying strings are *Elvi* is like saying STL is Evli and then laughing when everyone rewrites STL in Forth to get things done using human analogies like lists and maps. What's the problem? Just let people use STL. Not STL, I mean, let people have abstractions that matter. They're called containers. Contain them in a word and make them native. MUL anyone? Is it just me? Am I taking crazy pills? And then you move this over here, go line by line, put the TOS as a return value, reverse PUSH arguments in a () using ARG, which you then use to create an INFIX word, and then autowrap everything in a LINE_NO block... and.. and.. and then suddenly you have C. Or Basic. Or whatever. But Forth? Ha. What is Forth? ... Wait, what *is* Forth!?! OMG, what is Forth?

Do you hear me God? I am screaming from the top of the mountain! WHAT IS FORTH?

> There is an idea of a programming language, out there, somewhere. Like a Godzilla. It hasn't come out of the water yet. It has come to punish us for our sins, of poorly used technology. It will herald a new computer language. One with the simple elegance of Forth, the expressiveness and freedom of C, the straightforward organization of BASIC, the classes of C++, and most of all it will be an assembler, and a compiler, and an interpreter. It will do what even Forth can't do – or doesn't want to do. Either way.
>
> *"I have seen it..."* –Emperor Palpatine.

Then again, look what happened to *him.* Maybe I'll try forth again this weekend.