My history with Forth & stack machines

September 10th, 2010 | hardware, software

My VLSI tools take a chip from conception through testing. Perhaps 500 lines of source code. Cadence, Mentor Graphics do the same, more or less. With how much source/object code?

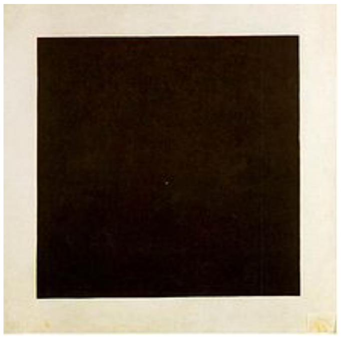
– Chuck Moore, the inventor of Forth

This is a personal account of my experience implementing and using the Forth programming language and the stack machine architecture. "Implementing and using" - in that order, pretty much; a somewhat typical order, as will become apparent.

It will also become clear why, having defined the instruction set of a processor designed to run Forth that went into production, I don't consider myself a competent Forth programmer (now is the time to warn that my understanding of Forth is just that - my own understanding; wouldn't count on it too much.)

Why the epigraph about Chuck Moore's VLSI tools? Because Forth is very radical. Black Square kind of radical. An approach to programming seemingly leaving out most if not all of programming:

> …Forth does it differently. There is no syntax, no redundancy, no typing. There are no errors that can be detected. …there are no parentheses. No indentation. No hooks, no compatibility. …No files. No operating system.



I've never been a huge fan of suprematism or modernism in general. However, a particular modernist can easily get my attention if he's a genius in a traditional sense, with superpowers. Say, he memorizes note sheets upon the first brief glance like Shostakovich did.

Now, I've seen chip design tools by the likes of Cadence and Mentor Graphics. Astronomically costly licenses. Geological run times. And nobody quite knows what they do. To me, VLSI tools in 500 lines qualify as a superpower, enough to grab my attention.

So, here goes.

***

I was intrigued with Forth ever since I read about it in Bruce Eckel's book on C++, a 198-something edition; he said there that "extensibility got a bad reputation due to languages like Forth, where a programmer could change everything and effectively create a programming language of his own". WANT!

A couple of years later, I looked for info on the net, which seemed somewhat scarce. An unusually looking language. Parameters and results passed implicitly on a stack. 2 3 + instead of 2+3. Case-insensitive. Nothing about the extensibility business though.

I thought of nothing better than to dive into the source of an implementation, pForth - and I didn't need anything better, as my mind was immediately blown away by the following passage right at the top of system.fth, the part of pForth implemented in Forth on top of the C interpreter:

```
: (   41 word drop ; immediate
( That was the definition for the comment word. )
```

```
    ( Now we can add comments to what we are doing! )
```

Now. we. can. add. comments. to. what. we. are. doing.

What this does is define a word (Forth's name for a function) called "(". "(" is executed at compile time (as directed by IMMEDIATE). It tells the compiler to read bytes from the source file (that's what the word called, um, WORD is doing), until a ")" - ASCII 41 - is found. Those bytes are then ignored (the pointer to them is removed from the stack with DROP). So effectively, everything inside "(…)" becomes a comment.

Wow. Yeah, you definitely can't do that in C++. (You can in Lisp but they don't teach you those parts at school. They teach the pure functional parts, where you *can't* do things that you *can* in C++. Bastards.)

Read some more and…

```
     \ conditional primitives
     : IF     ( – f orig )  ?comp compile 0branch  conditional_key >mark     ; immediate
     : THEN   ( f orig – )  swap ?condition  >resolve   ; immediate
     : BEGIN  ( – f dest )  ?comp conditional_key <mark   ; immediate
     : AGAIN  ( f dest – )  compile branch  swap ?condition  <resolve  ; immediate
     : UNTIL  ( f dest – )  compile 0branch swap ?condition  <resolve  ; immediate
     : AHEAD  ( – f orig )  compile branch   conditional_key >mark     ; immediate
```

Conditional *primitives*?! Looks like conditional primitives aren't - they *define* them here. This COMPILE BRANCH business modifies the code of a function that uses IF or THEN, at compile time. THEN - one part of the conditional - writes (RESOLVEs) a branch offset to a point in code saved (MARKed) by IF, the other part of the conditional.

It's as if a conventional program modified the assembly instructions generated from it at compile time. What? How? Who? How do I wrap my mind around this?

Shocked, I read the source of pForth.

Sort of understood how Forth code was represented and interpreted. Code is this array of "execution tokens" - function pointers, numbers and a few built-ins like branches, basically. A Forth interpreter keeps an instruction pointer into this array (ip), a data stack (ds), and a return stack (rs), and does this:

```
while(true) {
 switch(*ip) {
  //arithmetics (+,-,*…):
  case PLUS: ds.push(ds.pop() + ds.pop()); ++ip;
  //stack manipulation (drop,swap,rot…):
  case DROP: ds.pop(); ++ip;
  //literal numbers (1,2,3…):
  case LITERAL: ds.push(ip[1]); ip+=2;
  //control flow:
  case COND_BRANCH: if(!ds.pop()) ip+=ip[1]; else ip+=2;
  case RETURN: ip = rs.pop();
  //user-defined words: save return address & jump
  default: rs.push(ip+1); ip = *ip;
 }
}
```

That's it, pretty much. Similar, say, to the virtual stack machine used to implement Java. One difference is that compiling a Forth program is basically writing to the code array in a WYSIWYG fashion. COMPILE SOMETHING simply appends the address of the word SOMETHING to the end of the code array. So does plain SOMETHING when Forth is compiling rather than interpreting, as it is between a colon and a semicolon, that is, when a word is defined.

So

```
: DRAW-RECTANGLE 2DUP UP RIGHT DOWN LEFT ;
```

simply appends {&2dup,&up,&right,&down,&left,RETURN} to the code array. Very straightforward. There are no parameters or declaration/expression syntax as in…

```
void drawRectangle(int width, int height) {
  up(height);
  right(width);
  down(height);
  left(width);
}
```

…to make it less than absolutely clear how the source code maps to executable code. "C maps straightforwardly to assembly"? Ha! *Forth* maps straightforwardly to assembly. Well, to the assembly language of a virtual stack machine, but still. So one can understand how self-modifying code like IF and THEN works.

On the other hand, compared to drawRectangle, it is somewhat unclear what DRAW-RECTANGLE *does*. What are those 2 values on the top of the stack that 2DUP duplicates before meaningful English names appear in DRAW-RECTANGLE's definition? This is supposed to be ameliorated by stack comments:

```
: DRAW-RECTANGLE ( width height — ) … ;
```

…tells us that DRAW-RECTANGLE expects to find height at the top of the stack, and width right below it.

I went on to sort of understand CREATE/DOES> - a further extension of this compile-time self-modifying code business that you use to "define defining words" (say, CONSTANT, VARIABLE, or CLASS). The CREATE part says what should be done when words (say, class names) are defined by your new defining word. The DOES> part says what should be done when those words are used. For example:

```
: CONSTANT
   CREATE ,
   DOES> @
;
\ usage example:
7 CONSTANT DAYS-IN-WEEK
DAYS-IN-WEEK 2 + . \ should print 9
```

CREATE means that every time CONSTANT is called, a name is read from the source file (similarly to what WORD would have done). Then a new word is created with that name (as a colon would have done). This word records the value of HERE - something like sbrk(0), a pointer past the last allocated data item. When the word is executed, it pushes the saved address onto the data stack, then calls the code after DOES>. The code after CREATE can put some data after HERE, making it available later to the DOES> part.

With CONSTANT, the CREATE part just saves its input (in our example, 7) - the comma word does this: *HERE++ = ds.pop(); The DOES> part then fetches the saved number - the @ sign is the fetch word: ds.push(*ds.pop());

CONSTANT works somewhat similarly to a class, CREATE defining its constructor and DOES> its single method:

```
class Constant
  def initialize(x) @x=x end
  def does() @x end
end
daysInWeek = Constant.new(7)
print daysInWeek.does() + 2
```

…But it's much more compact on all levels.

Another example is defining C-like structs. Stripped down to their bare essentials (and in Forth things tend to be stripped down to their bare essentials), you can say that:

```
struct Rectangle {
  int width;
  int height;
};
```

…simply gives 8 (the structure size) a new name Rectangle, and gives 0 and 4 (the members' offsets) new names, width and height. Here's one way to implement structs in Forth:

```
struct
  cell field width
  cell field height
constant rectangle

\ usage example:
\ here CREATE is used just for allocation
create r1 rectangle allot \ r1=HERE; HERE+=8
2 r1 width !
3 r1 height !
: area dup width @ swap height @ * ;
r1 area . \ should print 6
```

CELL is the size of a word; we could say "4 field width" instead of "cell field width" on 32b machines. Here's the definition of FIELD:

```
 : field ( struct-size field-size — new-struct-size )
    create over , +
    does> @ +
 ;
```

Again, pretty compact. The CREATE part stores the offset, a.k.a current struct size (OVER does ds.push(ds[1]), comma does *HERE++=ds.pop()), then adds the field size to the struct size, updating it for the next call to FIELD. The DOES> part fetches the offset, and adds it to the top of the stack, supposedly containing the object base pointer, so that "rect width" or "rect height" compute &rect.width or &rect.height, respectively. Then you can access this address with @ or ! (fetch/store). STRUCT simply pushes 0 to the top of the data stack (initial size value), and at the end, CONSTANT consumes the struct size:

```
struct \ data stack: 0
  cell ( ds: 0 4 ) field width  ( ds: 4 )
  cell ( ds: 4 4 ) field height ( ds: 8 )
constant rectangle ( ds: as before STRUCT )
```

You can further extend this to support polymorphic methods - METHOD would work similarly to FIELD, fetching a function pointer ("execution token") through a vtable pointer and an offset kept in the CREATEd part. A basic object system in Forth can thus be implemented in one screen (a Forth code size unit - 16 lines x 64 characters).

To this day, I find it shocking that you can *define defining words* like CONSTANT, FIELD, CLASS, METHOD - something reserved to built-in keywords and syntactic conventions in most languages - and you can do it so compactly using such crude facilities so trivial to implement. Back when I first saw this, I didn't know about DEFMACRO and how it could be used to implement the defining words of CLOS such as DEFCLASS and DEFMETHOD (another thing about Lisp they don't teach in schools). So Forth was completely mind-blowing.

And then I put Forth aside.

It seemed more suited for number crunching/"systems programming" than text processing/"scripting", whereas it is scripting that is the best trojan horse for pushing a language into an organization. Scripting is usually mission-critical without being acknowledged as such, and many scripts are small and standalone. Look how many popular "scripting languages" there are as opposed to "systems programming languages". Then normalize it by the amount of corporate backing a language got on its way to popularity. Clearly scripting is the best trojan horse.

In short, there were few opportunities to play with Forth at work, so I didn't. I fiddled with the interpreter and with the metaprogramming and then left it at that without doing any real programming.

Here's what Jeff Fox, a prominent member of the Forth community who've worked with Chuck Moore for years, has to say about people like me:

> Forth seems to mean programming applications to some and porting Forth or dissecting Forth to others. And these groups don't seem to have much in common.
>
> …One learns one set of things about frogs from studying them in their natural environment or by getting a doctorate in zoology and specializing in frogs. And people who spend an hour dissecting a dead frog in a pan of formaldehyde in a biology class learn something else about frogs.
>
> …One of my favorite examples was that one notable colorforth [a Forth dialect] enthusiast who had spent years studying it, disassembling it, reassembling it and modifying it, and made a lot of public comments about it, but had never bothered running it and in two years of 'study' had not been able to figure out how to do something in colorforth as simple as:
>
> **`1 dup +`**
>
> …[such Forth users] seem to have little interest in what it does, how it is used, or what people using it do with it. But some spend years doing an autopsy on dead code that they don't even run.
>
> Live frogs are just very different than dead frogs.

Ouch. Quite an assault not just on a fraction of a particular community, but on language geeks in general.

> I guess I feel that I could say that if it isn't solving a significant real problem in the real world it isn't really Forth.

True, I guess, and equally true from the viewpoint of someone extensively using any non-mainstream language and claiming enormous productivity gains for experts. Especially true for the core (hard core?) of the Forth community, Forth being their only weapon. They actually live in Forth; it's DIY taken to the extreme, something probably unparalleled in the history of computing, except, perhaps, the case of Lisp environments and Lisp machines (again).

Code running on Forth chips. Chips designed with Forth CAD tools. Tools developed in a Forth environment running on the bare metal of the desktop machine. No standard OS, file system or editor. All in recent years when absolutely nobody else would attempt anything like it. They claim to be 10x to 100x more productive than C programmers (a generic pejorative term for non-Forth programmers; Jeff Fox is careful to put "C" in quotes, presumably either to make the term more generic or more pejorative).

> …people water down the Forth they do by not exercising most of the freedom it offers… by using Forth only as debugger or a yet another inefficient scripting language to be used 1% of the time.
>
> Forth is about the freedom to change the language, the compiler, the OS or even the hardware design and is very different than programming languages that are about fitting things to a fixed language syntax in a narrow work context.

What can be said of this? If, in order to "really" enter a programming culture, I need to both "be solving a significant real problem in the real world" *and* exercising "the freedom to change the language, the compiler, the OS or even the hardware design", then there are very few options for entering this culture indeed. The requirement for "real world work" is almost by definition incompatible with "the freedom to change the language, the compiler, the OS and the hardware design".

And then it so happened that I started working on a real-world project about as close to Forth-level DIY as possible. It was our own hardware, with our own OS, our own compilers, designed to be running our own application. We did use standard CAD tools, desktop operating systems and editors, and had standard RISC cores in the chip and standard C++ cross compilers for them. Well, everyone has weaknesses. Still, the system was custom-tailored, embedded, optimized, standalone, with lots of freedom to exercise - pretty close to the Forth way, in one way.

One part of the system was an image processing co-processor, a variation on the VLIW theme. Its memory access and control flow was weird and limited, to the effect that you could neither access nor jump to an arbitrary memory address. It worked fine for the processing-intensive parts of our image processing programs.

We actually intended to glue those parts together with a few "control instructions" setting up the plentiful control registers of this machine. When I tried, it quickly turned out, as was to be expected, that those "control instructions" must be able to do, well, everything - arithmetic, conditions, loops. In short, we needed a CPU.

We thought about buying a CPU, but it was unclear how we could use an off-the-shelf product. We needed to dispatch VLIW instructions from the same instruction stream. We also needed a weird mixture of features. No caches, no interrupts, no need for more than 16 address bits, but for accessing 64 data bits, and 32-bit arithmetic.

We thought about making our own CPU. The person with the overall responsibility for the hardware design gently told me that I was out of my mind. CPUs have register files and pipeline and pipeline stalls and dependency detection to avoid those stalls and it's too complicated.

And then I asked, how about a stack machine? No register file. Just a 3-stage pipeline - fetch, decode, execute. No problem with register dependencies, always pop inputs from the top of the stack, push the result.

He said it sounded easy enough alright, we could do that. "It's just like my RPN calculator. How would you program it?" "In Forth!"

I defined the instruction set in a couple of hours. It mapped to Forth words as straightforwardly as possible, plus it had a few things Forth doesn't have that C might need, as a kind of insurance (say, access to 16-bit values in memory).

This got approved and implemented; not that it became the schedule bottleneck, but it was harder than we thought. Presumably that was partly the result of *not* reading "Stack Computers: the new wave", and *not* studying the chip designs of Forth's creator Chuck Moore, either. I have a feeling that knowledgable people would have sneered at this machine: it was trivial to compile Forth to it, but at the cost of complicating the hardware.

But I was satisfied - I got a general-purpose CPU for setting up my config regs at various times through my programs, and as a side effect, I got a Forth target. And even if it wasn't the most cost-effective Forth target imaginable, it was definitely a time to start using Forth at work.

(Another area of prior art on stack machines that I failed to study in depth was 4stack - an actual VLIW stack machine, with 4 data stacks as suggested by its name. I was very interested in it, especially during the time when we feared implementation problems with the multi-port register file feeding our multiple execution units. I didn't quite figure out how programs would map to 4stack and what the efficiency drop would be when one had to spill stuff from the data stacks to other memory because of data flow complications. So we just went for a standard register file and it worked out.)

The first thing I did was write a Forth cross-compiler for the machine - a 700-line C++ file (and for reasons unknown, the slowest-compiling C++ code that I have ever seen).

I left out all of the metaprogramming stuff. For instance, none of the Forth examples above, the ones that drove me to Forth, could be made to work in my own Forth. No WORD, no COMPILE, no IMMEDIATE, no CREATE/DOES>, no nothing. Just colon definitions, RPN syntax, flow control words built into the compiler. "Optimizations" - trivial constant folding so that 1 2 + becomes 3, and inlining - :INLINE 1 + ; works just like : 1 + ; but is inlined into the code of the caller. (I was working on the bottlenecks so saving a CALL and a RETURN was a big deal.) So I had that, plus inline assembly for the VLIW instructions. Pretty basic.

I figured I didn't need the more interesting metaprogramming stuff for my first prototype programs, and I could add it later if it turned out that I was wrong. It was wierd to throw away everything I originally liked the most, but I was all set to start writing real programs. Solving real problems in the real world.

It was among the most painful programming experiences in my life.

All kinds of attempts at libraries and small test programs aside, my biggest program was about 700 lines long (that's 1 line of compiler code for 1 line of application code). Here's a sample function:

```
: mean_std ( sum2 sum inv_len — mean std )
  \ precise_mean = sum * inv_len;
  tuck u* \ sum2 inv_len precise_mean
  \ mean = precise_mean >> FRAC;
  dup FRAC rshift -rot3 \ mean sum2 inv_len precise_mean
  \ var = (((unsigned long long)sum2 * inv_len) >> FRAC) - (precise_mean * precise_mean >> (FRAC*2));
  dup um* nip FRAC 2 * 32 - rshift -rot \ mean precise_mean^2 sum2 inv_len
  um* 32 FRAC - lshift swap FRAC rshift or \ mean precise_mean^2 sum*inv_len
  swap - isqrt \ mean std
;
```

Tuck u*.

This computes the mean and the standard deviation of a vector given the sum of its elements, the sum of their squares, and the inverse of its length. It uses scaled integer arithmetic: inv_len is an integer keeping (1<<FRAC)/length. How it arranges the data on the stack is beyond me. It was beyond me at the

time when I wrote this function, as indicated by the plentiful comments documenting the stack state, amended by wimpy C-like comments ("C"-like comments) explaining the meaning of the postfix expressions.

This nip/tuck business in the code? Rather than a reference to the drama series on plastic surgery, these are names of Forth stack manipulation words. You can look them up in the standard. I forgot what they do, but it's, like, ds.insert(2,ds.top()), ds.remove(1), this kind of thing.

Good Forth programmers reportedly don't use much of those. Good Forth programmers arrange things so that they *flow* on the stack. Or they use local variables. My DRAW-RECTANGLE definition above, with a 2DUP, was reasonably flowing by my standards: you get width and height, duplicate both, and have all 4 data items - width,height,width,height - consumed by the next 4 words. Compact, efficient - little stack manipulation. Alternatively we could write:

```
: DRAW-RECTANGLE { width height }
  height UP
  width RIGHT
  height DOWN
  width LEFT
;
```

Less compact, but very readable - not really, if you think about it, since nobody knows how much stuff UP leaves on the stack and what share of that stuff RIGHT consumes, but readable enough if you assume the obvious. One reason not to use locals is that Chuck Moore hates them:

> I remain adamant that local variables are not only useless, they are harmful.
>
> If you are writing code that needs them you are writing non-optimal code. Don't use local variables. Don't come up with new syntax for describing them and new schemes for implementing them. You can make local variables very efficient especially if you have local registers to store them in, but don't. It's bad. It's wrong.
>
> It is necessary to have [global] variables. … I don't see any use for [local] variables which are accessed instantaneously.

Another reason not to use locals is that it takes time to store and fetch them. If you have two items on a data stack on a hardware stack machine, + will add them in one cycle. If you use a local, then it will take a cycle to store its value with { local_name }, and a cycle to fetch its value every time you mention local_name. On the first version of our machine, it was worse as fetching took 2 cycles. So when I wrote my Forth code, I had to make it "flow" for it to be fast.

The abundance of DUP, SWAP, -ROT and -ROT3 in my code shows that making it flow wasn't very easy. One problem is that every stack manipulation instruction *also* costs a cycle, so I started wondering whether I was already past the point where I had a net gain. The other problem was that I couldn't quite follow this flow.

Another feature of good Forth code, which supposedly helps achieve the first good feature ("flow" on the stack), is factoring. Many small definitions.

> Forth is highly factored code. I don't know anything else to say except that Forth is definitions. If you have a lot of small definitions you are writing Forth. In order to write a lot of small definitions you have to have a stack.

In order to have *really* small definitions, you do need a stack, I guess - or some other implicit way of passing parameters around; if you do that *explicitly*, definitions get bigger, right? That's how you can get somewhat Forth-y with Perl - passing things through the implicit variable $_: call chop without arguments, and you will have chopped $_.

Anyway, I tried many small definitions:

```
:inline num_rects params @ ;
:inline sum  3 lshift gray_sums + ;
:inline sum2 3 lshift gray_sums 4 + + ;
:inline rect_offset 4 lshift ;
:inline inv_area rect_offset rects 8 + + @ ;
:inline mean_std_stat ( lo hi — stat )
  FRAC lshift swap 32 FRAC - rshift or
;
: mean_std_loop
 \ inv_global_std = (1LL << 32) / MAX(global_std, 1);
 dup 1 max 1 swap u/mod-fx32 drop \ 32 frac bits

 num_rects \ start countdown
 begin
  1 - \ rects—
  dup sum2 @
  over sum @
  pick2 inv_area
  mean_std \ global_mean global_std inv_global_std rectind mean std
  rot dup { rectind } 2 NUM_STATS * * stats_arr OFT 2 * + + { stats }
  \ stats[OFT+0] = (short)( ((mean - global_mean) * inv_global_std) >> (32 - FRAC) );
  \ stats[OFT+1] = (short)( std * inv_global_std >> (32 - FRAC) );
  pick2       um* mean_std_stat stats 2 + h! \ global_mean global_std inv_global_std mean
  pick3 - over m* mean_std_stat stats h!
```

```
   rectind ?dup 0 = \ quit at rect 0
  until
  drop 2drop
;
```

I had a bunch of those short definitions, and yet I couldn't get rid of heavy functions with DUP and OVER and PICK and "C" comments to make any sense of it. This stack business just wasn't for me.

> Stacks are not popular. It's strange to me that they are not. There is just a lot of pressure from vested interests that don't like stacks, they like registers.

But I actually had a vested interest in stacks, and I began to like registers more and more. The thing is, expression *trees* map perfectly to stacks: (a+b)*(c-d) becomes a b + c d - *. Expression *graphs*, however, start to get messy: (a+b)*a becomes a dup b + *, and this dup cluttering things up is a moderate example. And an "expression graph" simply means that you use something more than once. How come this clutters up my code? This is *reuse*. A kind of *factoring*, if you like. Isn't factoring *good*?

In fact, now that I thought of it, I didn't understand *why stacks were so popular*. Vested interests, perhaps? Why is the JVM bytecode and the .NET bytecode and even CPython's bytecode all target stack VMs? Why not use registers the way LLVM does?

Speaking of which. I started to miss a C compiler. I downloaded LLVM. (7000 files plus a huge precompiled gcc binary. 1 hour to build from scratch. So?) I wrote a working back-end for the stack machine within a *week*. Generating horrible code. Someone else wrote an optimizing back-end in about two months.

After a while, the optimizing back-end's code wasn't any worse than my hand-coded Forth. Its job was somewhat easier than mine since by the time it arrived, it only took 1 cycle to load a local. On the other hand, loads were fast as long as they weren't interleaved with stores - some pipeline thing. So the back-end was careful to reorder things so that huge sequences of loads went first and then huge sequences of stores. Would be a pity to have to do that manually in Forth.

You have no idea how much fun it is to just splatter named variables all over the place, use them in expressions in whatever order you want, and have the compiler schedule things. Although you do it all day. And that was pretty much the end of Forth on that machine; we wrote everything in C.

What does this say about Forth? Not much except that it isn't for me. Take Prolog. I know few things more insulting than having to code in Prolog. Whereas Armstrong developed Erlang in Prolog and liked it much better than reimplementing Erlang in C for speed. I can't imagine how this could be, but this is how it was. People are different.

Would a good Forth programmer do better than me? Yes, but not just at the level of writing the code differently. Rather, at the level of doing *everything* differently. Remember the freedom quote? "Forth is about the freedom to change the language, the compiler, the OS or even the hardware design".

…And the freedom to change the *problem*.

Those computations I was doing? In Forth, they wouldn't just write it *differently*. They wouldn't implement them *at all*. In fact, we didn't implement them after all, either. The algorithms which made it into production code were very different - in our case, *more* complicated. In the Forth case, they would have been *less* complicated. Much less.

Would less complicated algorithms work? I don't know. Probably. Depends on the problem though. Depends on how you define "work", too.

The tiny VLSI toolchain from the epigraph? I showed Chuck Moore's description of that to an ASIC hacker. He said it was very simplistic - no way you could do with that what people are doing with standard tools.

But Chuck Moore *isn't* doing *that*, under the assumption that you *need not to*. Look at the chips he's making. 144-core, but the cores (nodes) are tiny - why would you want them big, if you feel that you can do anything with almost no resources? And they use 18-bit words. Presumably under the assumption that 18 bits is a good quantity, not too small, not too large. Then they write an application note about imlpementing the MD5 hash function:

> MD5 presents a few problems for programming a Green Arrays device. For one thing it depends on modulo 32 bit addition and rotation. Green Arrays chips deal in 18 bit quantities. For another, MD5 is complicated enough that neither the code nor the set of constants required to implement the algorithm will fit into one or even two or three nodes of a Green Arrays computer.

Then they solve these problems by manually implementing 32b addition and splitting the code across nodes. But if MD5 weren't a standard, you could implement your own hash function without going to all this trouble.

In his chip design tools, Chuck Moore naturally did not use the standard equations:

> Chuck showed me the equations he was using for transistor models in OKAD and compared them to the SPICE equations that required solving several differential equations. He also showed how he scaled the values to simplify the calculation. It is pretty obvious that he has sped up the inner loop a hundred times by simplifying the calculation. He adds that his calculation is not only faster but more accurate than the standard SPICE equation. … He said, "I originally chose mV for internal units. But using 6400 mV = 4096 units replaces a divide with a shift and requires only 2 multiplies per transistor. … Even the multiplies are optimized to only step through as many bits of precision as needed.

This is Forth. Seriously. Forth is *not* the language. Forth the language captures nothing, it's a moving target. Chuck Moore constantly tweaks the language and largely dismisses the ANS standard as rooted in the past and bloated. Forth is the approach to engineering aiming to produce as small, simple and optimal system as possible, by shaving off as many requirements of every imaginable kind as you can.

That's why its metaprogramming is so amazingly compact. It's similar to Lisp's metaprogramming in much the same way bacterial genetic code is similar to that of humans - both reproduce. Humans also do many other things that bacteria can't (…No compatibility. No files. No operating system). And have a ton of useless junk in their DNA, their bodies and their habitat.

Bacteria have no junk in their DNA. Junk slows down the copying of the DNA which creates a reproduction bottleneck so junk mutations can't compete. If it *can* be eliminated, it *should*. Bacteria are small, simple, optimal systems, with as many requirements shaved off as possible. They won't conquer space, but they'll survive a nuclear war.

This stack business? Just a tiny aspect of the matter. You have complicated expression graphs? *Why* do you have complicated expression graphs? The reason Forth the language doesn't have variables is because you *can* eliminate them, therefore they are *junk*, therefore you *should* eliminate them. What about those expressions in your Forth program? Junk, most likely. *Delete!*

I can't do that.

I can't face people and tell them that they have to use 18b words. In fact I take pride in the support for all the data types people are used to from C in our VLIW machine. You can add signed bytes, and unsigned shorts, and you even have instructions adding bytes to shorts. Why? Do I believe that people actually need all those combinations? Do I believe that they can't force their 16b unsigned shorts into 15b signed shorts to save hardware the trouble?

OF COURSE NOT.

They just don't want to. They want their 16 bits. They whine about their 16th bit. Why do they want 16 and not 18? Because they grew up on C. "C". It's completely ridiculous, but nevertheless, people are like that. And I'm not going to fight that, because *I* am not responsible for algorithms, other people are, and I want them happy, at least to a reasonable extent, and if they can be made happier at a reasonable cost, I gladly pay it. (I'm not saying you can't market a machine with a limited data type support, just using this as an example of the kind of junk I'm willing to carry that in Forth it is not recommended to carry.)

Why pay this cost? Because I don't do algorithms, other people do, so I *have* to trust them and respect their judgment to a large extent. Because you need superhuman abilities to work without layers. My minimal stack of layers is - problem, software, hardware. People working on the problem (algorithms, UI, whatever) can't do software, not really. People doing software can't do hardware, not really. And people doing hardware can't do software, etc.

The Forth way of focusing on just the problem you need to solve seems to more or less require that the same person or a very tightly united group focus on all three of these things, and pick the right algorithms, the right computer architecture, the right language, the right word size, etc. I don't know how to make this work.

My experience is, you try to compress the 3 absolutely necessary layers to 2, you get a disaster. Have your algorithms people talk directly to your hardware people, without going through software people, and you'll get a disaster. Because neither understands software very well, and you'll end up with an unusable machine. Something with elaborate computational capabilities that can't be put together into anything meaningful. Because gluing it together, dispatching, that's the software part.

So you need at least 3 teams, or people, or hats, that are to an extent ignorant about each other's work. *Even if* you're doing everything in-house, which, according to Jeff Fox, was essentially a precondition to "doing Forth". So there's another precondtion - having people being able to do what at least 3 people in their respective areas normally do, and concentrating on those 3 things at the same time. Doing the cross-layer global optimization.

It's not how I work. I don't have the brain nor the knowledge nor the love for prolonged meditation. I compensate with, um, people skills. I find out what people need, that is, what they *think* they need, and I negotiate, and I find reasonable compromises, and I include my narrow understanding of my part - software tools and such - into those compromises. This drags junk in. I live with that.

> I wish I knew what to tell you that would lead you to write good Forth. I can demonstrate. I have demonstrated in the past, ad nauseam, applications where I can reduce the amount of code by 90% and in some cases 99%. It can be done, but in a case by case basis. The general principle still eludes me.

And I think he can, especially when compatibility isn't a must. But not me.

I still find Forth amazing, and I'd gladly hack on it upon any opportunity. It still gives you the most bang for the buck - it implements the most functionality in the least space. So still a great fit for tiny targets, and unlikely to be surpassed. Both because it's optimized so well and because the times when only bacteria survived in the amounts of RAM available are largely gone so there's little competition.

As to using Forth as a source of ideas on programming and language design in general - not me. I find that those ideas grow out of an approach to problem solving that I could never apply.

102 comments ↓

#1 **JeanHuguesRobert** on 09.10.10 at 3:08 pm

> I bought a commodore VIC20 Forth cartridge when I was 16 I believe, it must have been in 1982.

Even though I am totally fascinated, to this day, I, too, never managed to understand how the damn thing would work "in real life".

Mister Chuck Moore is of the weirdest kind of genius I can relate to. I don't get Einstein at all, with Forth I feel like the stars are not so far.

Thank you for telling us your journey with Forth so nicely and my deepest congratulations for you refreshing humility.

#2 **Anton Kovalenko** on 09.10.10 at 3:41 pm

In my personal [imaginary] categorized language list, Forth is in the section named "cute things that doesn't scale", somewhere near Scheme (which they call Lisp when pushing it to unsuspecting students) and _hand-written_ PDP-11 assembly (or hand-written PDP-11 machine code: MACRO-11 is way too powerful for this category). For some years of my youths, I used to appreciate cute things as such, having no way and no desire to know whether they would scale, neither caring about it. Forth *is* amazing when you look at it this way, and the ease of implementing it (naïvely) even more so.

I have changed (hopefully, _corrected_) my opinion on cute little things and Forth in particular several times. Today I think that they're *still* deserve to be appreciated, but some care should be taken by any self-educating programmer not to mistake them for the Language of the Future etc.. For some people, it's way useful when a fairy tail starts with a disclaimer, like "don't try to use as if it were a real-life story". There is a place for cute little things in `real life' too; e.g. sometimes I just don't _want_ something to scale, and adding the `artificul disincentive' by choosing a cute solution automatically becoming ugly when you *try* to expand it may be a useful precaution.

Forth community is criticized frequently for failing to standardize (ANS Forth lacked even a shadow of credibility last time I checked). Today I don't regard _this_ kind of criticism as a smart thing: the entire Forth `fairy-tail' is not about standardization, it's about rolling your own. If I'd decide to implement an über-cute subset of Scheme, I'll approximate R4RS or the like, unless there is a reason not to; if I'd decide to roll my own Forth, I simple won't care and be happy [both things *may* be reasonable sometimes; e.g. I'd probably start with Forth on a system with ~16-32K RAM, no C compiler and no GCC port [yet]].

#3 **Sergey** on 09.10.10 at 5:25 pm

A great read, thanks.

#4 **Mitchell** on 09.10.10 at 6:51 pm

You might look at Factor http://factorcode.org/ .

It's sort of "scalable Forth". What you might get from a dare "if Forth is so great, then give me a SLIME and a Smalltalk IDE, modern language features, a real compiler, doing an SBCL-quality job targeting real x86, on all major OSen".

It's a work in progress. Rough edges. No MOP. Weak concurrency story. Limited type system. But quite a few nice features, some good people, and a community. A hopeful project. If Perl 6 could be next gen CL, Factor feels headed towards CL on a stack.

Another entry in the glacier race of "we know what language features are needed to start really getting work done - the only question is, given our utterly dysfunctional language ecosystem, in what decade will we finally get them".

Primary dev's blog: http://factor-language.blogspot.com/ Planet: http://planet.factorcode.org/

#5 **Joshua Noble** on 09.10.10 at 7:26 pm

This is a beautiful post, and a delight for (admittedly far less well learned) language nerd. Well done and thanks!

#6 **Yossi Kreinin** on 09.10.10 at 11:49 pm

Thanks for your comments!

Regarding ANS Forth - didn't seem to me like it "lacked credibility", in the sense that many implementations were compliant or largely compliant (more so than, say, many of the C++ implementations for many years), and in the sense that the docs seemed comprehensive and well thought-out (say, you had a discussion of what happens with cross-compilers, which isn't trivial in Forth, more generally there seemed to be much awareness to the possible differences in implementation and how semantics shouldn't restrict that.) At least that's how it seemed; didn't try to run large ANS Forth apps on different implementations and see what happens.

But - when I rolled my own, I did change lots of stuff… A lot of core words were the same, but not only was plenty omitted, some things were different - to better integrate with assembly and, um, "C". Say, there was a linker, and you could reference the address of a symbol with &sym - very un-Forth-y. You had #define and #include, and C-style comments, by virtue (vice?) of running cpp on the source before compilation. And so on. Exercised my freedom to change the language…

#7 **Antiguru** on 09.10.10 at 11:56 pm

You should write a book. You have the most interesting thoughts and experiences with programming, most of the big experts today are puffed up jokes. I feel sort of dumb for some earlier comments about pipelining and multithreading as I can see you probably know vastly more than me on the subject (though I do know pipeline size and unit size are multiples of 4 due to number of steps in executing instruction being 4).

My guess is that it's 18 bits so that he can have 2 metabits all to himself and still appease the 16 bit crowd.

It's probably absolute nonsense to try to compare productivity in languages in any measured sense but where the functional languages shine is in the meta sense, but reality requires a definite solution. It is sure easier to do string work with lisp but all of those are things in a general language you would build a tool to handle which did the job for you entirely. So it's true that if you do everything yourself it will be more efficient and usable - to you.

That is where the metalanguage nonsense fails. Adding in more complexity is obviously not desirable, let alone to have a language that is completely self defining like forth. It just becomes useless to every person but yourself.

Also it's interesting that forth sees itself as simplifying things, when it guarantees maximum complication. At least with C++ if you keep the 720 special case rules in mind and don't use its own metaprogramming features too much it's easy to see what's going on. But what does frog(aspy(grant(tumor)))) mean? All the language is defined by this guy's thinking. Which of course is never going to work when you have an API you want someone to actually use.

You can't learn a new language every single day any more than you could expect everyone else to talk to you in your own made up babble, no matter how much it makes sense to you. Even with somewhat clear APIs I usually find it easier to just code every single thing myself than to rely on an API because it takes longer to figure out how to use the latest amazing abstract library of orthogonal feature sets than it does to just make my own code.

#8 **Yossi Kreinin** on 09.11.10 at 12:04 am

Regarding Factor - to the extent of my understanding, it's a modern dynamic language, its single similarity to Forth being the stack (for example, its http-get call is followed by nip in their link scraping example.) Perhaps the single isolated feature of Forth most easy to take away to an altogether different environment, but the least favorite of mine (sounds moronic to like Forth and dislike stacks; well, I like Lisp but dislike its mutable, tail-sharing, cons-exposing lists - here, Clojure seems to have done a nice job of salvaging the right things, like macros, and not the wrong but iconic things, like naked cons). Forth's approach to metaprogramming (parsing words) is absent from Factor - not that it's necessarily a great thing if you want to scale, but AFAIK nobody checked (the extent to which you can salvage Forth's metaprogramming without dragging in the rest of Forth is unclear to me.)

#9 **Jon** on 09.11.10 at 12:10 am

Awesome as always. Thanks for the wonderful write-up. I've said in the past that Chuck Moore is the most extreme case of the NIH syndrome there is, he ended up designing his own chips. I love your "superhuman" take on this.

#10 **Eric Normand** on 09.11.10 at 2:53 am

Nice article. Thanks for sharing your story.

Just a minor point: people used to roll their own OS/language all the time, back before computing was commercialized. It was what graduate students had to do to get computers up and running.

I like the idea that Forth is a very minimal way of getting something running on a bare system (no OS) very quickly. You boot into this very small program that can quickly be expanded to be very productive. It's a very important idea in computer science.

On the other hand, I think I would want to write another layer on top of Forth as soon as I could. That is to say, a layer with garbage collection and data types and maybe a file system. I would like to see this shortest path from bare metal to modern operating system studied. I think it is important to computer science as a science. We should all have to write our own OS, just as any master painter has to create a masterpiece. Somehow, I think Forth would be at the beginning of that path.

The reason I think Forth would not serve past the initial layers is found in several interviews with Moore. He talks about programming in Forth being the best puzzle (better than Sudoku or Crosswords) for keeping your mind fit. I've also read blog posts of people reasoning out algorithms in Forth, and it looks to be very time consuming. The reason it is a puzzle is that it is not obvious and the smallest bits have to all fit together. This is not productive programming.

At the same time, I think his quote about reducing the code size of problems by 90% is accurate. He is the kind of person who will write an MPEG codec in 1kb (I exaggerate). It asks the question: what are the codec writers doing wrong? Why do I have to download a 50meg file when there is an implementation in 1kb?

It reminds me of a quote by Alan Kay. I have to paraphrase: Moore's law has given us 50,000 fold improvement in computer speed, but we've squandered that with a 5,000 fold decrease in software speed. So we've only gained 10x.

#11 **Yossi Kreinin** on 09.11.10 at 7:38 am

Regarding 18b: gotta be all data. What metadata - type info? We don't need no stinking type info.

Regarding the 10x vs 50000x: we seem to have got pretty much everything we could out of Moore's Law where we had to - Quake or Google Search, not? That WP wastes server cycles around the world is sad, but apparently economical (that is, it's better to spend the money on looking for a cure for cancer than on the obscene amount of work that would go into optimizing all software for which that doesn't matter.)

#12 **Ivan** on 09.11.10 at 3:00 pm

Yossi: Factor does have user-defined parsing words. There's also a comparison here: http://cd.pn/FactorVsForth.pdf (incomplete, I think).

#13 **Doug** on 09.11.10 at 3:01 pm

Actually, Factor has parsing words in the Forth style. For example:

SYNTAX: HEX: 16 parse-base ;

You can do { HEX: a HEX: b HEX: c } and it'll do what you want. Note that { is also a parsing word!

More:
http://docs.factorcode.org/content/article-parsing-words.html

#14 **Shimshon** on 09.11.10 at 3:31 pm

This is a great article. I LOVE Forth. I actually had a real job (paid and everything!) programming with it when I was a student. I got the job because out of the 30+ interviewees, I was the only one who had even heard of the language, let alone used it (which I also did). I worked in astrophysics lab (heaven for this kind of gig), writing the code to control the entire apparatus and record and analyze all the data. I had about 250 screens of code, which included a GUI (this was 1990) along with lots of direct hardware interaction with poorly documented components and some very hairy math. The PC was a 386 with a then-huge 8MB of RAM, plus a 387 math coprocessor, running LMI 386 Forth. It was AWESOME fun!

#15 **Kragen Javier Sitaker** on 09.11.10 at 5:49 pm

I remember when I thought Forth was the next big language, too. And like you, my experience trying to program in it was disillusioning.

A big part of the problem, I think, is that I tried to use the stack instead of variables. Forth has always had variables (at one point in your post, Yossi, you say it doesn't; but I think you mean local variables.) It doesn't have to be any harder than C to write. You can translate directly from C to Forth; C:

```
static int mac80211_hwsim_start(struct ieee80211_hw *hw)
{
struct mac80211_hwsim_data *data = hw->priv;
printk(KERN_DEBUG "%s:%s\n", wiphy_name(hw->wiphy), __func__);
data->started = 1;
return 0;
}
```

Forth, just a straight translation of the C:

```
variable hw variable data
: mac80211_hwsim_start hw !
hw @ priv @ data !
KERN_DEBUG s" %s:%s" hw @ wiphy @ wiphy_name printk
1 data @ started !
0 ;
```

This function doesn't happen to be recursive. If it did happen to recurse, we'd have to explicitly save the values of its "local" variables on the stack before the call and restore them afterwards. On the other hand, most functions aren't recursive.

Now, maybe you can optimize that Forth a little bit; for example, you can probably dispense with the variable "data" and just use the top-of-stack, and actually that variable is only read once and surely the debug message line doesn't modify hw->priv, etc. etc. Maybe I should have randomly picked a different piece of C. But at some point along the path of "optimizing" and "simplifying" the Forth code, you find yourself getting into the kind of nightmarish code you demonstrated above, where you have four or five things on the stack and they move all the time, and understanding it is just a nightmare. But you don't have to do it that way. You can use variables, just like in C, and the pain goes away. You never have to use any stack-manipulation operations, not even once. They're always there, tempting you to use them, taunting you; and you have to exercise a great deal of restraint to avoid writing your code as cleverly as possible, because that will make it impossible to debug.

I think in this case the painless-but-optimized version looks like this:

```
: mac80211_hwsim_start
dup wiphy @ wiphy_name log priv @ start ;
```

Here I figure that "log" is locally defined as something that printks a KERN_DEBUG message with the calling function name followed by a colon and then the argument, and that : start started 1 swap ! ;.

But when I said "it doesn't have to be any harder than C to write", I lied a little bit. It doesn't have to require detectably more code, but per line, Forth is still a bit more error-prone than C, in my experience. In C you get enough static type checking to immediately detect something like 90% of your type errors; you get a warning if you forget to pass an argument to a function, or pass too many; the data flow of any subroutine is easily approximable without having to know the stack effect of every subroutine you call; and so on. But maybe if I programmed enough in Forth, these errors would become less significant.

(I suspect that global variables are less of a problem in Forth than in other languages: you can have multiple global variables with the same name without accidental sharing; you're very unlikely to have mutually recursive functions without knowing it (and, in any language, recursive or mutually recursive functions require special care to ensure termination anyway (that is, recursion is error-prone); etc.)

On the other hand, as you pointed out, Forth is easily extensible. I think, as Eric Normand pointed out, that you want to use that extensibility to bootstrap into a less error-prone, more problem-level language/system as quickly as possible, and in fact, this is the standard approach using Forth promoted by the likes of Chuck Moore and Elizabeth Rather, as I understand it. It's just that the next layer up they're talking about is a more traditional domain-specific language, something like FoxPro or Rexx or sh, rather than something with garbage collection and data types.

In theory, at least, it seems like as you scale to a larger program, Forth's advantages over C would become more significant, as your program looks more like a tower of DSLs — up to the point where you split your C program into multiple programs that the OS protects from each other.

There's a quote from Jeff Fox in my quotes file:

[In C under Unix] Bugs are planned, and the whole picture is all about
the planning for bugs.

Forth is about planning for good code where the bugs don't happen. If
you say BEGIN AGAIN damn it, you mean BEGIN AGAIN not twenty other
possible meanings based on C insisting that it is one of twenty
different bugs that need extra hardware and software to be handled
properly.

– Jeff Fox , in a discussion on
comp.lang.forth, inadvertently explaining why Forth is not
widely used, 2006-05-20, in message-id
,
subject "Re: hardware errors, do C and Forth need different
things in hardware?"

My own take on Forth is that it's by far the simplest way to build a macro assembler, and you can do anything with it that you can do with any other macro assembler, perhaps a little bit more easily and with more portability, and syntax that's not quite as nice. At some point you want a high-level language on top of your macro assembler, though. The Forth theory is that implementing a domain-specific high-level language has a better cost/benefit ratio than implementing a general-purpose high-level language, and a macro assembler is a perfectly adequate way of implementing a domain-specific high-level language.

Where this approach falls down is that it's true that implementing a domain-specific high-level language has a better cost/benefit ratio than implementing a general-purpose high-level language if you're implementing it for a single user, such as NRAO. But if your program memory isn't limited, and you can share the language with all the computer users in the world, a general-purpose high-level language like Lua or Python or Tcl or Ruby is a more economically efficient tradeoff, because whenever you implement some optimization, they all get the benefit.

(Also, I actually think it's easier for me to write bug-free assembly than bug-free Forth, but that may be a matter of experience.)

With regard to 18-bit words, I guess the advantage over 16-bit words is that you can fit four instructions per word instead of three. (The low-order bits of the last instruction are necessarily 0; fortunately that is true of NOP.)

Once I asked a famous CPU designer (who shall remain anonymous, since this wasn't a public conversation) what he thought about Chuck Moore. He said something to the effect of, "Chuck Moore? I used to work under him at AMD. He's great!"

"No," I said. "The Forth guy."

"Oh, HIM!" he said. "In my DREAMS I could do what he does."

I think the GreenArrays people are making a big mistake by marketing their chip as competition for microprocessors and microcontrollers. What they're really competing with is FPGAs and PALs. Unfortunately, they don't have a VHDL or Verilog synthesis system for their chips, and I don't think they're going to build one.

I can't claim to be a good or even an adequate Forth programmer. So take all of this with a grain of salt.

#16 **Kragen Javier Sitaker** on 09.11.10 at 7:12 pm

Oh, I forgot to say: I think the reason that almost every bytecode system under the sun is stack-based is that low-level bytecode is basically a bad idea these days, but it made a lot of sense in days when code density was really important. (Bytecode as a compact representation of an AST might still make sense.)

And stack-based bytecode is a lot denser than purely register-based bytecode. Hybrid bytecodes like Smalltalk-80's, where you have 8 or 16 or 32 bytecodes that fetch and store to registers, are even denser.

#17 **Yossi Kreinin** on 09.12.10 at 12:50 am

@Doug: thanks for the info. I'm not sure I understand the picture - it's not exactly Forth metaprogramming - but it sure has parsing words.

@Ivan: thanks, and - ROTFL! The book, I mean. Love how the world is split into "real world computing" (microcontrollers) and "desktop computing" (probably includes cell phones as well as high-end embedded DSPs as well as anything where it would be reasonable to use, say, malloc). The prediction of Factor soon replacing C++ and Python (Factor is faster than Python, clearer than C++) testifies of fairly little exposure to "desktop programming" indeed…

"This is one of the pitfalls of integers - that they don't represent numbers less than one." Yeah, that's definitely one of them pitfalls of integers! Then he says you absolutely need floating point to deal with this, which is kind of strange considering his apparent experience with precision; why not scale 1 to something like 0×10000 ? 0×10000/sum([0x10000/x for x in resistors]) works just fine. Perhaps he's doing it for didactic purposes, to show Forth's floating point words.

As to the Forth program where a sequence is represented as a zero-terminated list of items on the stack - doesn't seem like idiomatic Forth; you'd have trouble passing the sequence - or anything else - around until you got rid of it. That Factor apparently disallows such things (as functions are supposed to have stack effects known at compile time) is IMO a good thing.

Unfortunately, the file seems truncated or something - I didn't get anything past the "par" examples.

@Kragen:

"In my DREAMS I could do what he does" - did it mean he dreamed about being able to do that, or that he claimed to be able to do that with his eyes closed?

I'm not sure how dense stack-based bytecode really is, what with all the stack manipulation and named locals you need; it's the same question of "flow". Sure, every command is short due to implicit operands, but you end up with more commands. In my experience, the difference in density isn't that big (then of course it depends on the specifics and I only have experience with one particular stack vs RISC encoding).

As to vars - Forth has globals and locals indeed (at least in most dialects), don't think I claimed otherwise anywhere.

#18 **Mate Soos** on 09.12.10 at 3:24 am

Great blog entry. I only wish my command of hardware, maths and metaprogramming was even near yours…

#19 **Yossi Kreinin** on 09.12.10 at 4:02 am

Oh no you don't. Especially math.

#20 **Kragen Javier Sitaker** on 09.12.10 at 11:18 am

I'm pretty sure he meant he dreamed about being able to do the kind of stuff Chuck did, because he seemed awed.

It's true that you still have to specify operands sometimes, whether it's in the form of three register numbers in every instruction (like in SPARC or Lua) or in the occasional dup, swap, or pushTemp:3 operation. My gut feeling is that, on a stack machine, you probably end up with about a quarter to half of your instructions being used to fetch and store arguments and results. So your program is 133% to 200% of the size it would be if it magically had to only specify what operations to invoke, and not what to invoke them on. This compares quite favorably to the SPARC/Lua approach, where the corresponding number is 400%, and even the traditional fixed-width single-accumulator-machine approach where you get one operand per instruction, where it's about 200% or 300%.

I know it sounds like I'm pulling these numbers out of my foreskin, so I invite you to read the notes I posted at http://lists.canonical.org/pipermail/kragen-tol/2007-September/000871.html, where I compare Squeak bytecode, threaded 16-bit Forth, trees of conses, Lua, the MuP21, the F21, SWEET16, the JVM bytecode, OCaml, the BASIC Stamp, PICBIT, and BIT, to widely varying levels of detail. If you were bored by my comment, you would completely detest the full-length post.

I think I wrote the code in that post before I had my epiphany about how you should use the Forth stack for expression evaluation, not instead of variables, so some of the code in it is pretty awful.

I also, at the time, hadn't heard about the HP 9100A calculator, which could store one instruction (= keyboard keypress) per six-bit digit of memory; I suspect that the HP RPN calculators used this approach until at least the 1980s.

But, anyway, that's where my conclusion about stack-based bytecode being tops for density comes from. It could still be wrong, but it's based on some examination of real systems.

#21 **Vlad Patryshev** on 09.12.10 at 7:21 pm

My success story with Forth was that there was a year to develop a drilling station simulator (that's deep drilling, and it was in Russia). The Engineer assigned to that had wasted 11 months trying to write something smart in c. Then we were out of time. I took over, spent, sorry, 3 weeks writing a Forth interpreter for that specific chip, with all the blackjacks regarding the inputs etc. The remaining week (before New Year) I wrote the simulator; as you guess, it was in Forth, and it was just several pages of code that drilling technology engineers could read (and fix) - that including a formula interpreter. Since then, Forth was feeding me and my friends for several years.

#22 **Kragen Javier Sitaker** on 09.12.10 at 7:37 pm

Vlad: that sounds awesome. Is there some Forth code somewhere public that you think is exemplary of good style? I mean, I've read Thinking Forth, but it's been a long time; and there are of course the classics like F-83.

Also, what's a blackjack?

#23 **Yossi Kreinin** on 09.12.10 at 10:47 pm

Great story, and - about when did that happen (I assume that quite some time ago), and how did Forth even reach Russia back then? Couldn't you have done it in C in those 4 weeks though, or in whatever language? It sounds like you were the better programmer by far. Or, is there something deeply Forth-y that went on there, the way Armstrong showcased Erlang as a good fit for telephony projects (a C project collapsed, replaced with working Erlang code because of features X, Y and Z)? Such as, say, interactivity (one great thing about Forth I didn't really need where I dragged Forth in)?

#24 **Kragen Javier Sitaker** on 09.13.10 at 8:49 am

I suspect that if I did more of my Forth programming in an interactive Forth environment, with shorter words, I'd have less trouble with parameter-passing and type-checking bugs too. Not that I wouldn't have the bugs, but I'd find them and fix them immediately.

Of course, if I was writing words that were five lines long, the equivalent of ten or twenty C statements, I'd probably still have trouble.

#25 **Michael Clagett** on 09.13.10 at 2:05 pm

I wonder what you would think of the custom Forth programming environment I have put together over the past couple years working extremely part time in fits and starts as my busy life's time permitted.

I would have to call this Forth-inspired, as it makes no attempt to conform to any standard and it's only measure of success is whether it has been useful to me in developing the platform I am trying to develop. From this standpoint alone, it has been immensely successful; the thing I think that makes Forth worth all of its hassles for me is the fact that I have complete control over my environment and can pretty much have my way with a intel machine.

The trick here is to tell you enough about what I've been doing to give you a feel for it without bogging you down with so many details that my post becomes a 10-page article. The platform that I'm trying to put together is a foundation for a series of products (if I don't die first) that give their users the ability to build mixed visual/textual domain-specific languages. I happened upon Forth only because that's what I was taught twenty years ago by a friend who decided to help me come up the programming learning curve. Not your average beginning programming experience, but I credit it for introducing me early to seriously good design aesthetics and principles that only about ten years later did I encounter in the mainstream in my study of other languages.

So what is it about Forth that is so promising for a Domain-Specific Language development environment of the kind i am building? Two things really: it's natural extensibility and the fact that languages can be constructed (if you wish) without a lot of detailed understanding of abstract syntax trees and parsing and the like; and the absence of formal parameter binding and stack frames of the kind that you find in most other languages.

The latter can lead to very intuitive and natural-language-like syntax for expressing computational capabilities. And Forth's basic concatenative and compositional character makes it possible to mix and match different language constructs from different surface languages (like, for example, mixing snippets of C++ syntax with snippets of SQL query syntax) as long as you can figure out how to implement them in some underlying Forth implementation. As long as your implementations merge seamlessly in their stack manipulation you can combine language constructs to your heart's content.

I myself am building formal grammar and AST functionality on top of this basic foundation so that I can give myself the ability to work in a traditional syntax tree or more free-form treeless fashion, whichever the need dictates.

So a bit about my platform. It is built as a software virtual machine implementation of one of Charles Moore's later Forth processor concoctions. It has at its core what Moore calls a "minimal instruction set machine" — 32 (in my case bytecode primitive) instructions, the two standard Forth stacks and a single address register. Moore of course implemented all of this in hardware, while I do it in a software vm that is intended to run on an Intel machine and is mapped to the Intel registers. (Top of Data Stack = eax; Forth instruction pointer = edx; Top of Return Stack = edi; Forth address register = esi). Then I have two stacks in memory (to which I have added a third in the code I build on top of this to manage object scope and a "this" pointer in the type system I construct).

I've made a few minor changes to Moores's basic 32 instructions to suit the machine better for its 32-bit host environment. Like Moore I have an internal 24-bit address space (I think his was 20 bits, if I remember correctly) as well as the external intel host 32-bit addressing. And whereas Moore was packing four five-bit instructions into a single memory access, I pack four eight-bit instructions into a memory access. This aligns better with the host 32-bit architecture and leaves room for growth of the instruction set.

The 24-bit internal memory addresses are added to a 32-bit base address in all of my memory access primitives to actually reference some piece of the intel's memory. This is nice, as it allows me 16Mb of memory addressing that doesn't need to be fixed up during a loading sequence. I can persist these memory addresses and load them back up in a flash each program execution (very fast!). Mine is a subroutine-threaded Forth with most Forth words consisting of lists of call instructions that embed an internal 24-bit Forth address in instruction slots two three and four. Since 'call' is just one of the 32-bit primitive instructions, these calls can be mixed freely with other byte codes so that high-level and low-level Forth code is mixed seamlessly.

All of this basic stuff is implemented in a small C++ program that writes the intel code for all of this to memory along with code implementing basic Forth parsing, name/code dictionary management and interpret/compile mechanisms. This program initializes all this stuff by assembling intel bits to memory and then jumps to it as soon as it is able to. From there in typical Forth fashion everything is bootstrapped in the environment itself.

Things I have built on top of this for myself include a byte-code assembler for the underlying machine and an inheritance-based type system with interfaces and generic types (I use a very C++-like angle bracket syntax). In this respect I am a complete violator of Moore's "less is more" ethos and carry no shame around my need for more C++-like facilities.

Like any good Forth I have external function call access to the C++ runtime library as well as operating system functions. At the moment I have to pass any of these that I want to use into the initial machine setup as function pointers, but one of the next things on my list is to build me a "load library" capability. This runtime library access includes much of the STL facilities — although I make no use of the compile time type safety features. Rather, I call out only to instantiations of STL containers and funtions that take generic ints as their type parameters. I use these as generic addresses to reference my own types and code. A bit kludgy, but it actually works pretty well and gives me access to a lot of STL container and algorithm functionality (not to mention libraries like Boost and Blitz Numeric).

Finally (and I will stop soon) I have built an intel assembler on top of this that allows me to mix assembly code inline with the byte code and higher-level forth code already described. As with the 'call' instruction, one of my other 32 instructions is asm!, a primitive that will grab an intel address from the top of the data stack and execute assembler code located there.

This ability to move back and forth fluidly between assembler and Forth leads to a very powerful programming capability that is completely dynamic (I look at Forth's compile mode as being dynamic rather than static in that you can switch back and forth between compile and interpret without stopping your running program). Sooner or later I will get around to implementing some more type safety checking that will at the very least check stack transforms during compilation and data types of stack contents during execution to help provide better safety when and where it might be desired.

Okay, this is enough. I'm sorry. I knew this is what I would do, but I couldn't stop myself. Finding an active discussion of Forth's potential just unleashed all this pent-up geekitude that has been building inside of me during the solitude of this unwieldy and effort-intensive development effort. I hope you will forgive me.

My basic bottom line though is that in addition to the qualities of Forth that make it a great Domain-Specific Language vehicle, the overall control that one has is tremendous. My very next task is to try to build the capability of generating .NET MSIL bits as part of my word implementations and then a mechanism to transition between managed and unmanaged code. Part of the product vision for the visual language capability is to make these visual languages dynamic and executable in a comparable way to the textual ones that we know and love so well. When I get to that point I will want the use of facilities like those available in the .NET Framework library to be part of the language implementations I allow people to create.

Okay slap me down for being an undisciplined comment hogger. But as you can see, I really love Forth!!!!

#26 **Yossi Kreinin** on 09.13.10 at 2:16 pm

If I'd been doing that much work outside of my day job, I'd be certain to publish it, without even minding the state of its usability, just for getting feedback and generating buzz and stuff (Subtext/Coherence is a great example of high-end vaporware that most language geeks know about and follow sympathetically - which will definitely be good for it when/if it materializes, and will do no harm if it doesn't.) At the very least, I'd set up a site with a source repository, basic docs and a blog.

#27 **Shimshon** on 09.13.10 at 4:38 pm

Michael, I second Yossi's suggestion. Post the source (please)! This sounds like a fascinating project.

#28 **Michael Clagett** on 09.13.10 at 5:10 pm

Yossi –

Thanks for your prompt response. You're almost certainly right about setting up the site. I think what prevented me from doing this early on was some sort of screwed up pride of ownership — that I wanted it to be my creation. The flip side of that, of course, is the fear that the way I've done all this is stupid and naiive and that it will elicit bemused criticism from those more adept and knowledgeable than myself.

Also, some of it is just plain ugly — like the C++ code that initially kicks things off. In my early rush to attempt to see if the basic concept was doable I was very undisciplined and have scads of very fragile assembly language generation code that uses hand-calculated jump instructions and the like and that is not at all structured or factored the way I would go back and do it today. I believe I am afraid of being embarrassed by this sort of thing and so am at the very least waiting until I can just go back and refactor this mechanism.

Finally, there are pieces of the foundation that are missing that I would really like to create before giving the world a peek at this. I've gone back and ripped apart some of the foundations a couple of times (although I don't dare touch the shit I alluded to above) and so I'm really pretty satisfied with much of it — at least as a credible first pass. But two things that I really would like to add (and I don't think I'm that far away) are 1) replacing the linked list dictionary mechanism with a hash table; I've already implemented (but not yet tested) an assembly language hash table that I got from a great book by Rick Booth. So I really would like to get that in place. And 2) extending my dynamic loading mechanism to handle separate modules, adding cross-module memory fixup in the process. Up to this point I do have the capability to save to disk an image of what's been built and then to load it from disk on subsequent runs. This will do for the core foundational stuff that I'm always going to want to have present. But I am now getting into functionality that will need to be optional and modularized.

Finally, one of the hazards of departing from accepted language processing has been that I don't have a really good mechanism for exception handling in place — just a kind of crude abort feature that will print out a message to a host window in Visual Studio. Stack traces and diagnostics are in the same boat.

So I am concerned about exposing the work with all of this stuff to do and getting sidetracked with feedback that might be submitted on it. Also, while I don't flatter myself that anything I've been doing is really all that original and valuable as intellectual property per se, there is still the issue of wanting to retain control (at least for now) over the direction this development takes. I'm really not experienced and don't have any idea how people handle these kinds of issues in the public domain.

Of course I recognize counterbalancing all these fears and concerns is the potential for real valuable feedback and serious improvement of my effort. Moreover, if it turns out that what I am doing is interesting, even if it is the basis for something that I might want to commercialize at some point, I guess there are models for working even on that basis in the public's eye.

Do you have any thoughts about this. I'm really quite naiive and inexperienced in this arena.

Thanks again for your feedback.

#29 **Kragen Javier Sitaker** on 09.13.10 at 7:10 pm

Michael: it sounds like an interesting project. I think you should publish it ASAP because, you know, you never know when you could get hit by a bus. It sounds like it might not yet be useful to other people (I mean, most people who want to program in mixed x86 assembly and Forth will probably just pick up Win32Forth until yours is better) but if you talk about it then other people might take notice, and you could get some feedback. If people don't like it, that may or may not be useful. But maybe they will.

#30 **Yossi Kreinin** on 09.13.10 at 11:44 pm

Yeah, it's SO embarrassing when your assembly generation code uses hand-calculated jump instructions. Seriously - you realize that in this profession, at least every second practitioner couldn't implement sorting properly for the life of them? It's as if you're a rocket scientist embarrassed to speak about your latest ballistic missiles because they don't even have the feature of disintegrating into several parts to deal with counter-missile weapons, something all major superpowers have been doing since the 80s, based on the assumption that everyone in the room is obviously a rocket scientist who has followed the trend and will therefore find your efforts laughable. C'mon.

Most language design efforts "fail" in the sense of never gaining a large user community and never fully achieving their initial (typically ambitious) goals, but trying and failing in the closet is extremely depressing and psychologically devalues all knowledge and understanding gained in the process - I know this, for example, based on my own closet work on computer graphics and vision (hacked on dense optic flow and on extending feature-based morphing to support Bezier splines; learned shit about discontinuities you get when warping those splines - have I been doing it publicly, I could have easily got comments that could help me move forward but the way I did it, I just quit working on it at some point after getting stuck and now I barely remember what I've been talking about). But what can become a complete failure when done in solitude can become a success of a not entirely expected kind when done publicly - there's potentially much more directions work could evolve at based on feedback, and much more motivation to proceed given feedback.

Of course there's the question of how to present your work, and the longer one refrains from doing it and has one's own increasingly complicated relationship with it, the harder it becomes. What I'd do is just look at how others are doing it. The Factor language is as good an example as any - check at how their docs and their blogs look like; or you could pick any other successful attention-grabbing project without a very serious user community at this point (of course Ruby gets attention these days - got users, bad example) and without something else to be responsible for the attention (Arc is Graham's, Graham got rich from programming - bad example). Just look at how they do that. That's how I forced myself into blogging - it's really hard for me, at the basic level, to talk to unknown people, where you don't know what they think of you, or whether there are any (going to be any) to care at all and stuff. I just copied the style of what I liked to read as a starting point.

Fact is, there are people out there who'd love to discuss language design questions; one of every 20 comments you get is likely to be unexpectedly valuable. I wouldn't miss out on the opportunity.

As to commercializing - by far the most likely way to ever get any money out of this kind of work is as a side effect of attention you get. I mean, you can't sell anything in this area - all the competition gives everything away; take Python - a runaway success and all its designer got is worldwide fame and some sort of good position at Google as a result. So this is the last argument for staying in stealth mode in this domain, really.

#31 **Michael Clagett** on 09.14.10 at 2:44 am

Well, no doubt you're right about all of this. And I have been toying with the idea of doing just what you suggest. I think I will probably take the time to at least fix the nasty bug that just surfaced with one of the last things I did. A while back I ripped out the foundations and rebuilt everything on top of new underpinnings and have been slaving to just get back up to the point that I was at before doing this. I'm very close to getting it all working again and would like to at least submit some code that compiles and runs.

I'll be honest with you. I still find this extremely difficult. I realize how screwed up that is, but I'm just being honest. I have never envisioned what I am doing as something that I wanted other people to adopt and use. It's always been very comfortable being my own private little thing. But then why do I go and comment about it at some blog, if I'm not interested in some feedback and in sharing my effort. I never said I wasn't crazy.

#32 **Mike L** on 09.14.10 at 9:39 am

Yossi: Great blog post. When I code Forth (which I don't do a lot any more), I also sometimes get too wrapped up in stack gymnastics. A few well-chosen global variables help a lot. I have used some versions' local-variables feature a few times, but it's not kosher, right? ;-) So, I'm greatly influanced by Forth, even if I do most of my programming in C/C++.

Michael:

I'm a long-time fan of Forth, and have not used it much professionally. I am a fan of Chuck Moore and find him inspirational, but I'm not so much an extremist. I have two accomplishments in Forth for which I am somewhat proud:

1) I wrote a CPU simulator for the Motorola (nee Frescale) HC11 that ran on Windows 3.1 and later nicely. I did it mostly on free time, but did use the simulator for some professional embedded development work. This was never released to the public and sits in my personal archives only, helping no one else.

2) I wrote a Java-applet version of Forth, "jeForth" that made a servicable demo of Forth running in a web browser. I wrote it up for Forth Dimentions magazine and promoted it a bit on comp.lang.forth. The first version had a limited-rights copyright notice, but I later GPL'ed it. Then some people in Europe (UK, mostly), picked up on it and improved it greatly and made a bunch of Forth tutorial pages. Maybe you've seen it. (http://www.figuk.plus.com/webforth/Index.htm). That was before "blogging" became so easy and common, but comp.lang.forth served in a similar way.

I'm most comfortable and profficient at C (and as little C++ as I can get away with), and that is how I code most of my professional work. On my own time, I started making tight little apps in Linux with the FLTK GUI library for the TinyCore Linux platform I like. Again, I first publicized and opened my code to the user base through a public forum (tinycorelinux.com). Recently, I created a Google sites website with a blog and basic file download features (https://sites.google.com/site/lockmoorecoding/). Comments are enabled, but I don't get much traffic yet. I think this website will encourage a few more people to try out my stuff and make suggestions. Most of the feedback so far is at the "feature request" level. Some contains specific code suggestions. Maybe someday I will team up with someone withwhome I can closely work, but for now, the "work openly with an open mind" is fine solo.

Some of my posted code is not pretty, some may be fragile, and a lot is suboptimal. But its out there, getting some users (a few of my apps are in the official TinyCore Linux repository), and getting better from the feedback. I encourage you to do something similar.
–
Mike "Lockmoore"

#33 **Michael Clagett** on 09.14.10 at 5:34 pm

Thank you all for your encouraging comments. I will probably post something eventually, but am working pretty non-stop at the moment at getting myself up to the next point. I'll try to take some time out soon though to share my work.

But getting back to the original theme of the post, I found myself adding a local variable capability to my own Forth and I wanted to share my reasoning. The facility could be considered a bit kludgy; I'm really too close to it to judge. It works basically like this:

```
string printfDelimiter "%"
115 constant 's'
100 constant 'd'

// (outStream — ) followed by string in input stream
code _printf
>>locals
std::stringObj formatStr
std::stringObj subStr
int pNextStart
assign_Ptr() drop

// start searching at beginning of str
0 pNextStart !A
begin
// get/save start offset
pNextStart @A dup push
```

```
// string to search for (intel addr)
printfDelimiter toHost
// takes (offset strPtr) as params
formatStr L-> findFirstOf_PtrOff()

dup npos !=
if
// stack now: (… eosFlag nextDelim)
swap
else
2purge

// stack now: (… eosFlag strLen)
formatStr L-> length()
then

// update pNextStart with next pos past %x
dup 2 + pNextStart !A
// data: (… eosFlag delim cnt) ret: (… startPos)
dup rGet -

if
// dstack: (… count startPos)
pop
// get the substr to output
subStr toHost formatStr L->
substr() fromHost

// data: ( … stream substr ) ret: ( eos delim )
a! push push a

// output substring to stream
std::string-> < operator[]() a! b@

switch
case 's'
drop toHost

// output host string ptr on top of stack
[ operator<<Str ]
callFarLogged
case 'd'
drop

// output int on top of stack
[ operator<<Int ]
callFarLogged
caseDefault
drop
-1 abort "Currently printf only supports strings and ints\n"
endSwitch
repeat

// discard eosFlag, delimPos and outStream
drop pop drop drop
```

<>locals, <locals and <>locals and the >locals creates a hidden class and places its symbol dictionary on top of the dictionary stack; the local variables themselves are then defined with field semantics just as if they were fields in one of my classes; <locals closes this hidden class and compiles code to do the following at runtime:

1) with assembly language create a stackframe (by manipulating esp and ebp as would normally be done).

2) iterate through the class dictionary just created replacing the execution token in each definition with that of a thunk that it creates.

3) This thunk at runtime places the runtime stackframe pointer on the top of the Forth data stack and then calls the original xt it has replaced, which then treats the stack frame pointer as an instance object base for the locals' hidden type class.

In this way, field semantics can be used and the field definitions themselves have no idea that it isn't a normal class object they are consuming from the top of the stack. (It in fact is a normal class object; it's just a special purpose type used for defining this function's local variables).

<<locals, which appears after all code has been compiled that might use the local variables, then cleans everything up and discards the local class and dictionary, which isn't needed anymore.

If any of this isn't clear (which I wouldn't be surprised at all) I would be happy to share the source for these functions with anyone who is interested.

Now why did I go to the trouble of creating all this (other than just to torture myself and add a month or two on to my development effort)? It's because I get tired of tiresome stack manipulations. Inside a function I often use (as many people do) the return stack to hold values that will be used multiple times. With my r5Get, r4Get, r3Get, r2Get and rGet stack access words, I can grab these values fairly easily. (I also have r5Pop, r4Pop, r3Pop, r2Pop and of course the normal pop to be able to get and remove these values.) But I find locals to be much more self documenting. Stretches of code with bunches of stack manipulation variables is not so easy to absorb in my opinion. Now if I were a Chuck Moore superForther, I wouldn't have any definitions that were more than a line or two and I wouldn't even have the opportunity to need such stack manipulation. But I'm not and so I do.

In my mind self-documentation is a beautiful thing and probably trumps most other competing concerns.

Additionally, in the case of printf above, it really was useful to make use of the C++ std::string class to do the heavy lifting. While theoretically I could create these dynamically and manipulate pointers to them on the return or data stacks, I found it much more straighforward to use the already programmed functionality and access mechanisms of my class semantics. An additional benefit is the ability to make use of a stack frame like more traditional languages do.

#34 **Michael Clagett** on 09.14.10 at 5:43 pm

Oh drat! The posting mechanism took out all my indentation and also ate some of my characters; there is some critical code missing above.

For those who care to do the mental work the following lines should have been posted in place of the tenth line of code above:

assign_Ptr() drop

It would of course be one of the lines with a key component of the mechanism I am discussing. I say, forget about trying to make sense of the non-indented code above and let me just mail you the sourcce. Please feel free to mail me at mclagett@hotmail.com if you wish to receive it and discuss. Or just comment here and I will mail it to you.

If anyone does want to discuss, let's continue to do it here; it's more fun that way.

Cheers.

Mike

#35 **Michael Clagett** on 09.14.10 at 5:44 pm

I give up. They're missing again! Just email and I'll send you the code.

#36 **gus3** on 09.14.10 at 9:23 pm

Seven years ago, I thought I would be a self-declared genius by designing my own programming language. I started by stating some requirements, including "stack-based" and "typeless". I put the project aside, when life intervened.

A few months later, I pulled it out and took a look at what I had put together. After just thirty seconds or so, I pointed out to myself that it was nothing new: "Congratulations. You just re-invented FORTH."

#37 **Michael Clagett** on 09.15.10 at 5:18 pm

A couple more thoughts on the original topic of whether Forth's strengths outweigh its weaknesses and how generally usefull it is as a computing framework.

Having used it now day in and day out for a couple of years — not in my day job but with some serious hours put into a side project — I can testify, probably better than most, that Forth is a mixed blessing. As I said in a previous comment, I really really love it. But that's basically because I'm a crazed power-hungry megalomaniac who wants complete control so that I can range freely up and down the abstraction hierarchy and be fairly certain that I can accomplish anything I truly feel is necessary for what I am doing.

Most programmers have no such need, however, and for them doing without scoped variables, call parameter binding and other such staples of most modern programming languages (including the functional ones based on the lambda calculus) is to do without the shared frame of reference that makes programming such a communicative exercise.

The bottom line in my mind is that programming is as much about expressing the concepts of a problem domain and solutions as it is about getting work done. And in order to express yourself and share your work with others, you have to be working in a medium that is accessible not just to you, but to the community at large as well. This is where Forth falls short. If I'm a Java programmer, I can look at something that's been done in C#, Visual Basic, C++, Pascal and pretty much understand fairly quickly what's going on. Maybe the same isn't true for JavaScript, which can be pretty inscrutable to the uninitiated, but the overall programming idea is very similar and with a few basics one can catch on fairly quickly.

It's probably even less the case with OCaml, Haskell and Scheme, etc. which do require a fairly significant mental shift, but at the end of the day they also are about passing binding values to paramters and passing them into functions. All you have to do is look at how quickly lambda expressions have been adopted and used heavily in C# to see how really familiar they end up being to traditional programmers.

Concatenative languages in general and Forth in particular are a different beast. Maybe, as some people suggest in previous comments, it's just the cultural way that many Forth programmers came to write their code. And certainly there is nothing stopping you from writing extremely expressive code in Forth and with a little care you can even make it quite natural language like. But at the end of the day, a programmer is going to want to look at it and envision the processing involved.

And it is here, in my opinion, where Forth presents one of its biggest barriers to widespread adoption, with its postfix notation, it's surfacing of the stack machine and its disdain for more familiar concepts like parameter/value binding and local variables. Not that it's less expressive without this, it's just less familiar.

But what is true for programmers is definitely not true for the public at large. For many of them, in my experience, the more familiar a language syntax is to the average programmer, the more inscrutable it is to the average layman. For these folks, who don't really need to know or care about the processing that's being done, the more baggage-free a syntax is, the more meaningful it becomes.

It still takes a lot of work to write natural-language-feeling code with Forth and to hide the whole post fix thing under the covers. But it's easier to do this in my opinion than it is to get rid of function call syntax and parameter passing in a more traditional language or to somehow make lambda expressions understandable to the uninitiated.

That's the primary reason I chose Forth for my platform. But I don't have any illusions about what I'm going to have to do to make it usable by traditional programmers — which is basically to surface some more mainstream programming language on top of it. This will have to exist side by side with other more natural language domain-specific languages I allow to be created as well. But this really just reflects the true purpose of the platform, which is to help act as a bridge between software engineers and the mere mortals that they serve.

Okay, that's enough for now.

#38 **Kragen Javier Sitaker** on 09.18.10 at 6:58 am

Michael: if you're finding it's difficult to write parsers in Forth, maybe you should check out Brad Rodriguez's article on BNF parsing in Forth, which should make it easier to write a simple backtracking recursive-descent parser in Forth than in most other languages. (It still suffers from the difficulties of recursive-descent parsers: exponential-time parsing if you're not careful, and infinite recursive loops if your grammar is left-recursive, so you have to refactor your infix grammar to not be left-recursive.)

With respect to "scoped variables": traditionally, Forth variables are kind of like C static variables, in that most of them are only visible to a small part of your source code, due to vocabularies/wordlists and the fact that their lexical scope ends at the next declaration of a variable with the same name. This is one reason Forth non-local variables aren't as bad as global variables in C.

#39 **void** on 09.26.10 at 10:46 am

I like playing around with forth, I loved the part of the binary arithemetic to shave off time. It reminds me of earlier days, people were different then indeed. I think you are doing it right by using many small words, I too sometimes feel the need to clarify things with 'c' comments but that's a sign to me that I need another or more small words to better describe the problem. Lisp and Forth, they both are nice. People often tell me why on earth one would fiddle with them but when you simplify a python loop from 20 lines down to one quite lispy line they understand why.

#40 **Yossi Kreinin** on 09.26.10 at 4:30 pm

Lisp and Forth are very similar in one way and very different in another; Lisp is much bigger but much less likely to blow one's both legs to little pieces. As to 20 LOC of Python simplified to 1 LOC of Lisp - I'd like to see that.

#41 **Jacko** on 10.05.10 at 4:17 am

Not too bad an article. Still haven't booted my own forth yet, but the priority is with other things at the moment. A language with a type structure where void is primary and is linked. Void -> (List/Symbol/Number/Vocab/Exec/Machine) as subclasses. Don't know why I'd want to duplicate the functionality of Symbol by making a String class. I'll probly have a LEL function as a dual to ADD. For that harmonic parallel…

As is often happening these days, I am not as impressed by any language, feature or hyp as I used to be. Maybe that just happens when your 40.

#42 **P.M.Lawrence** on 10.05.10 at 7:07 pm

I'll just put in one little thing, a poor man's local variable approach I've occasionally used in Forth (it's not a true local variable thing, since pointers to the variables don't work that way, it just pushes and pops old values on and off the return stack). First, set up a couple of ordinary global variables, say A and B. Then code like this:-

: DEMO

(maybe some code)

```
A @ >R ( pushing value of A to return stack )
B @ >R ( pushing value of B to return stack )

( do some work with A and B as though they were local )

R> B ! ( popping value of B from return stack )
R> A ! ( popping value of A from return stack )

( maybe some more code ) ;
```

You have to be careful to balance things, but thereafter you can forget about the fact that the code using A and B is treating them as locals, and within that you have no more overhead than globals do.

On the layering thing, the only time I ever used Forth for real was when I was given a massive project WITHOUT the right kind of flexibility. I was required to do some sophisticated high level stuff for a utility to be linked and run from Cobol programs on IBM 360 or 370 hardware, with no programming team and using only officially sanctioned languages - Cobol and assembler. I only knew Cobol, but even that only on other platforms, i.e. not all the IBM features. I ended up doing the work in quasi-Forth to minimise not only the low level but also the learning curve of needing more assembler features, i.e. not interactively but via assembler macros, with I/O in a separate Cobol module, implementing some object oriented stuff (which I had only heard of as "data driven", in a Lisp book), and reinventing co-routines from scratch as I had never heard of them. Forth let me leverage what I DID know and what I WAS allowed control over to achieve a solid and reliable result, albeit not as efficiently as if I had previously used enough Forth to be safe and confident of implementing it with better optimisation (I bought the Loeliger book "Threaded Interpretive Languages" and used the indirect threading from that). And the whole thing was pointless anyway, since it was supposed to deskill things so auditors wouldn't have to learn to read dumps, but IBM's operating systems needed them to learn just as much just to install the utility…

#43 **Hugh Aguilar** on 10.06.10 at 3:31 pm

People who want to learn Forth ought to start by writing an application in Forth.

I think that too many people become fascinated with the internal workings of Forth compilers (or in this case, of a Forth engine), and they write their own basic Forth system as a first effort. The key word here is "basic" — such a first-effort Forth system is going to lack a lot of support code necessary for writing applications. The person then tries to write an application and fails due to lack of basic programming support. The result is that the person spends the rest of his life telling everybody that he is an *expert* in Forth because he wrote his own Forth compiler (a pretty impressive story for C++ and Java programmers who can't imagine writing a C++ or Java compiler themselves). In the next breath however, the person states that it is impossible to write a serious application in Forth because Forth is just too primitive (not admitting that it was only his own Forth system that was too primitive). The result is that Forth gets a reputation for being a "cute" language (to use one of the commentator's terms), but of being impractical — essentially a science-fair project.

This was pretty much the point of my novice package — I wanted to provide novices with a library of code necessary for writing applications. http://www.forth.org/novice.html

It is not just novices who need a library of code like this, either. If a person pays $500 for SwiftForth, they get a system that is completely lacking in code necessary for writing applications. There is no support for records, arrays, lists, or anything else. The person tries to port a program over from C or Lisp or whatever, and he gets bogged down in not knowing how to emulate C's STRUCT in Forth, or Lisp's lists in Forth, and so forth. He gets blocked by the most basic problems, and he ends up deciding that his $500 was wasted and that Forth is impractical for writing applications. He really needed some help in getting started on writing applications, but everybody in the Forth community was too busy pondering the wonders of threaded-code to provide any support for the application writer.

Applications! What a concept! Why didn't we think of that before?

#44 **Hans Bezemer** on 10.06.10 at 11:05 pm

You tried to leapfrog into Forth. Don't. It will take years before you're good enough to program Forth. And frankly even those "gurus" you describe as "extreme DIY" do not always write good Forth. Just good compilers ;-) But when it comes together, it is nice, very nice. E.g. this is a mini-blackjack program. You catch my drift:

```
include lib/shuffle.4th
include lib/cards.4th
include lib/yesorno.4th

\ This is a showcase for the libraries above. No money is involved here,
\ splitting is not supported and an ace is always 11 points.

: score 13 mod dup if 1+ 10 min else 11 + then ;
: next-card deal dup card space type cr score + ;
: player ." Player: " cr 0 begin next-card s" Hit" yes/no? 0= until cr ;
: dealer ." Dealer: " cr 0 begin next-card dup 16 > until cr ;
: won? dup 21 > if drop ." is bust!" else ." has " 0 .r ." ." then cr ;
: shuffle-deck new-deck deck /deck cshuffle ;
```

: minijack shuffle-deck player dealer ." Dealer " won? ." Player " won? ;

minijack

#45 **Yossi Kreinin** on 10.06.10 at 11:57 pm

@Hugh: yeah, that's why I said I didn't consider myself a competent Forth programmer, let alone an "expert", despite having implemented a Forth dialect. As to support code for writing apps - IMO very relevant to many real-world many cases but not mine since I wrote very low-level code: no dynamic allocation, no text, the most trivial data structures, etc. - the examples above show the kind of things I tried to do, struggling with the most basic traits of Forth while trying to do the most basic things. So I don't think a general-purpose support library would help me much, though I did try to write a support library for the specific stuff I worked on, not that it worked out very well for me but that's another matter.

@Hans: I don't play card games nor know the included libraries so it's hard for me to appreciate the program - wouldn't know how to write it in any other programming language, though it sure looks nice and compact. As to having to wait years before getting any good - Jeff Fox says so, too, about how Forth programming is like musicianship or martial arts taking many years to master. It seems to me, however, is that many (most?) actual programming jobs are more like driving - a pedestrian activity, if I may say so, an important one and requiring responsibility and skill but nothing an average adult can't handle and something we'd be disappointed to spend the talent and time required to master exalted art forms on.

#46 **Hans Bezemer** on 10.07.10 at 7:01 am

@Yossi Kreinin
The included libraries handle a simple Knut shuffle, simple card game primitives and asking "Yes"/"No" questions. All as tiny or tinier than the program itself.

But that's not the point here. If you consider programming to be for everybody (the Thomas Kurtz paradigm) then Forth is not for you. Forth is build around the idea (the Chuck Moore) paradigm that programming is too important to leave to amateurs. Hammers simply don't come with "don't hit your thumb" protection. The idea that programming can be left to "idiots" (industrial scale programming) is the cause of the bad situation of software today. End quote.

In short, whether Forth is suited for your situation or problem is directly related to the paradigm you support. Everything else is academic.

#47 **Yossi Kreinin** on 10.07.10 at 9:30 am

"Bad situation" compared to what, I wonder; and if the industry is filled with idiots, I'd expect the minority of master programmers to collect most of the revenue of, say, Google.

There are a lot of arguments that can be made for something despite that something having little economic significance at the time when the argument is made, but not beyond limits. The idiots ("idiots") part is well past the limit.

#48 **Hans Bezemer** on 10.07.10 at 1:36 pm

@Yossi Kreinin
Note the "end quote" part. I didn't have the time to search through all Moore's interviews and ramblings around the web. But that is really how he feels.

Dijkstra utters (or uttered I should say - he's dead) along similar lines. He thought only mathematicians should be allowed to program. BASIC made you "braindead". "It's not lines produced" he used to say "But lines spent." END QUOTE.

Note most software has around 10-20 defects per 1000 lines. Only 1% of the s/w companies around are labelled CMMI Lvl 3 or better. Over 75% of all s/w projects fail (over budget, over time, cancelled). Compare that to other industries. Personally, I don't think that is a track record to be proud of as an industry. On the other hand, in the days of Dijkstra and Moore there was a "s/w crisis": not enough s/w was produced. I think we solved that one.

But the s/w craftsman, the master, that is an entity we have lost. Probably forever. Who can teach the real trade. And can we ever get rid of that image as the secondhand car salesmen of engineering??

#49 **Yossi Kreinin** on 10.07.10 at 2:41 pm

Only 1% are CMMI level 3 or above ("worse", I'd say, where you say "better")? Good, 99% get some work done then. As to comparison to other industries - it's not just the failure rate you'd be interested in but the outcome of success. Many more than 75% potential locations of oil turn out to have no oil but it's still worth it to search for oil because of the value of what you do find.

I'm not saying we're very good at programming - we aren't, just that what gets done after all is a lot by any reasonable standard. As to programming being bad engineering - it is unclear to what extent programming is engineering at all; just as what submarines do isn't exactly swimming though not entirely unlike it.

The question is how to measure the worth of software. Clearly dollars earned is a flawed metric but definitely better than defects/LOC IMO and then if someone makes billions in an open market, without relying on patents or even network effects (google.com is thus a good example), then how imprecise the dollars earned metric has to be to prove the software is still really really bad? Unrealistically imprecise if you ask me.

Of course we can say that software that, say, isn't mathematically correct or isn't optimal in some theoretical sense ("could be done cheaper") is intolerably bad, but why does this make any sense for such an artifact? A program is a mathematical object in a sense, but its utility has different roots than that of a theorem. Even a chess move played in a real game is arguably to be judged based on the psychological effect on the given opponent in the given game as visible in the opponent's following moves, and not just based on its mathematical correctness.

#50 **Hugh Aguilar** on 10.07.10 at 2:58 pm

Yossi — I have experience with writing "very low-level code" for a Forth engine. I worked at Testra when they built their MiniForth chip on a Lattice 1048isp PLD. I wrote the cross-compiler, assembler and simulator for it (called MFX). The processor was a WISC — each opcode could contain up to five instructions, which would execute concurrently. My assembler would rearrange the instructions so as to pack them into the opcodes with as few NOP instructions as I could manage. The instructions were very low-level — addition, for example, had to be written as a function built out of lower-level instructions. An important design goal was to have a fast multiply, because this was the bottleneck in the Dallas 80c320 motion-control program (this was in the mid 1990s when the '320 dominated the micro-controller world).

I wouldn't really recommend that anybody tackle a project like MFX as a first-ever Forth project. This was the point I was trying to make — aim for something reasonable to start out. Your effort at learning Forth on a custom Forth engine was like learning how to swim by jumping into the deep end off the high-dive board.

Even simple projects can benefit from a good library. For example, you were lacking local variables, which is why you became bogged down in stack-shuffling, which seems to be what largely turned you off on Forth. I don't recommend overuse of locals, which I see as a misguided effort to turn Forth into C, but I do think that the judicious use of locals can greatly simplify some functions.

Mostly though, I think that learning Forth programming style and Forth philosophy is more important than any particular software tools (I didn't use local variables at all for many years). Consider the following points:

1.) You want to factor your functions into small functions. The worst thing you can do in Forth is write page-long functions and/or functions that do a lot of nested conditional testing (or use CASE). This kind of code is the mark of the recalcitrant C programmer, and is by far the #1 reason why people fail at Forth. You want to write short functions that do one thing, and which have no side-effects.

2.) You want to use records (see FIELD in my novice package) to bundle data. Often, when people get into trouble with stack-shuffling a lot of datums, most of those datums should have been bound together as a single record. This can greatly reduce how many datums you have on the stack (you generally don't want more than three). Any use of ROLL is a sign that you are in trouble, and any use of PICK is a sign that you are getting into trouble.

3.) You want to pass pointers to functions (called "vectors") around as data a lot, as this can simplify programs considerably (this is roughly analogous to the LAMBDA function in Lisp). See my list.4th program in the novice package as an example of how to do this. There are other examples in there too, such as SORT.

Try to think of Forth as an opportunity to program in a Lisp-like manner on micro-controllers that Lisp would be too big for. Also, try to forget everything that you know about C, as this is largely antithetical to Forth. If you do this, you will have a pretty good chance at succeeding. Download my novice package (I have an upgrade coming out in a few days) and try again! :-)

#51 **Hugh Aguilar** on 10.07.10 at 3:08 pm

Programming is not engineering; programming is art.

Programming in Forth is like painting a masterpiece on canvas.

Programming in Java is like painting a barn — it is somewhat similar to Forth, but is not really in the same category.

#52 **Yossi Kreinin** on 10.07.10 at 3:13 pm

IMO Forth is definitely a great "Lisp-like" language for a microcontroller - if you want something comparatively very efficient but don't mind a little inefficiency. Say, locals - I had locals, they'd just cost a cycle or two, not important for most purposes, important for mine. Similarly for passing around function pointers and bundling data in structures. I think I can write quite decent Forth given that this stuff is affordable, and I definitely see your point. Just saying that I was really, really anal-retentive about performance, *beyond* what typically happens on uCs. Not saying it's a good first Forth target. The thing is, a desktop machine isn't necessarily a good Forth target, either as you can afford "a safer Lisp", such as, um, Lisp. My view, and I don't claim it to be authoritative in the slightest, is that Forth is the best fit for small microcontrollers where you typically have little memory but do have a few spare cycles; I was on a coprocessor running exclusively the application bottlenecks so not only had little memory but also no spare cycles. Admittedly atypical. BTW what you described is quite different and much more complicated than what I was doing: I had a hardware stack machine so no effort went into compilation and a VLIW subsystem handled manually using inline assembly, whereas you had Forth targeting a far from trivial machine architecture. So perhaps my target wasn't that tough a Forth target after all.

BTW - anything you'd done radically differently in the examples I quoted using whatever facilities I left out? Just interesting if there's something that can be done without losing performance to make it look a teeny bit less ugly.

#53 **Yossi Kreinin** on 10.07.10 at 3:15 pm

Masterpieces sell for much more than what you'd made painting a barn. Now if the analogy extended to Forth software and Java software, it could have been taken more seriously.

#54 **Hugh Aguilar** on 10.08.10 at 10:38 am

In regard to desktop computers, Forth is often faster than Lisp. Some Lispers ported my encryption-cracking program to Common Lisp and it turned out that even GForth (which is a pretty slow Forth) was significantly faster than CLisp (even with a lot of type declarations uglifying the Lisp code). That program did a lot of integer arithmetic though, so it is not necessarily a representative example.

Do you program in Lisp? I only know a little about Lisp; I've never written a non-trivial program in Lisp. I am planning on getting into PLT Scheme soon — Forth is getting boring for me, and Scheme seems like the obvious next step. I didn't like Factor.

In regard to performance issues on a micro-controller, if you are building a custom processor, performance is the responsibility of the electrical engineer. The Lattice PLD is a pretty inexpensive part, but it ran Forth very fast. With lasers, speed is extremely important because if the laser hesitates it will burn a blotch at that place. The MiniForth was able to do this. Performance is only a problem when you are using off-the-shelf processors that were not designed to run Forth.

My painting analogy was pretty lame, so you shouldn't take it seriously. I was mostly slamming Java because of its reputation for boiler-plate code. I don't actually program in Java either, but I have seen Java code and it looks similar to C++ (which I do program in).

As for making money as a Forth programmer, forget it. When I worked at Testra and wrote MFX, I was making a wage roughly comparable to semi-skilled factory labor. Also, Forth experience is considered to be a negative when applying for work as a C programmer.

#55 **Yossi Kreinin** on 10.08.10 at 1:45 pm

Forth outperforming uglified Lisp? Very interesting. Don't see how this can be, it's probably in some details I fail to imagine. Anyway, even if Forth is faster than Lisp on the desktop, which for non-uglified Lisp code it certainly is, you're just so much less performance constrained on the desktop that you'll pay the performance to get the extra safety, reflection, etc. - or at least I'll pay.

As to performance - if you must be done in 10K cycles and you have a job that takes 200 cycles in the most optimal implementation, then you can tolerate a slowdown by a factor of 3 as long as it's deterministic. If you must be done in 10M cycles and you have a job that takes 7M in the most optimal implementation, then you can't tolerate such as slowdown. So the 10K cycles situation, which in my mind is somewhat representative of what happens on RT uCs, is really less awful in some ways than the 10M cycles situation which in my mind is somewhat representative of what happens in RT high-performance computing, therefore RT uC code might come out prettier. Hope this is clear/relevant.

Forth experience is almost certainly a positive when applying for a job at a real tech company, not? I listed Forth, Lisp (which I never really programmed at) and such as languages I'm interested in though have no real experience with and one language or other tends to grab the attention of one interviewer or another in a good way. Perhaps Forth experience is a negative in those C shops where Forth could have realistically been an alternative so they don't want to hire someone constantly itching to do things in the other way?

#56 **Hugh Aguilar** on 10.09.10 at 12:09 pm

The point I was trying to make in regard to performance, is that hardware can be 1000s of times faster than software. I heard about one case in which a 4-bit processor significantly outperformed a 32-bit computer. It was just doing one simple task repetitively, and all of the work was being done in hardware. This can't be done for every application though — it would be pretty hard to write a word-processor when you only have 16 nybbles of memory. In many cases though, a 16-bit custom-made processor can outperform a 32-bit off-the-shelf processor because more hardware resources can be dedicated to performing application-specific work.

"Perhaps Forth experience is a negative in those C shops where Forth could have realistically been an alternative so they don't want to hire someone constantly itching to do things in the other way?"

That pretty much sums it up.

#57 **Hugh Aguilar** on 10.12.10 at 11:54 am

"Forth outperforming uglified Lisp? Very interesting. Don't see how this can be, it's probably in some details I fail to imagine."

The Forth program relied heavily on mixed-precision arithmetic. Lisp is like most languages in that if any part of the calculation requires double-precision, then the entire calculation gets "infected" and has to be done at double-precision.

#58 **Yossi Kreinin** on 10.12.10 at 3:06 pm

Oh yeah. In C, I got to implement an inline assembly macro to do 32bx32b->64b (optionally immediately followed by >>shift->32b) a few times, sometimes it gets optimized properly and sometimes it doesn't. It's one thing that Forth gets Right that the vast majority doesn't.

#59 **Shimshon** on 10.17.10 at 11:45 pm

I'm not a Lisp expert, but I'm pretty sure CLisp is considered one of the slower Lisps. It uses a bytecode design, whereas others, like SBCL, run natively.

Regarding apps, Forth was used for some pretty innovative products back in the 1980s. The Canon Cat is pretty famous. The Epson QX-10 is another, lesser-known example.

And, as I mentioned above, I was responsible for a fairly large (250 screens) app to manage an astrophysics lab.

#60 **Kragen Javier Sitaker** on 11.09.10 at 3:58 pm

Here's how I think I would rewrite your mean_std word above in somewhat C-accented Forth. I hope the formatting makes it through in some form. Maybe this is more valuable than people saying things like "use short words" and "rot means you're in trouble".

```
( We have some fixed-point math here. Let's admit that, and define the fixed-point math primitives. It looks like one of sum and inv_len is a fixed-point fraction, while the other is an ordinary integer; I'm
guessing that sum2 is the sum of the squares, and inv_len is the reciprocal of the length, which therefore must be a fraction, while sum and sum2 are ordinary integers. Adopting "." as an indicator character for fixed-point arithmetic, even though that conflicts with
its usual Forth meaning of "output": )
: s.* u* ; : .>s FRAC rshift ;
: .* um* 32 FRAC - lshift swap FRAC rshift or ;

\ Now this should be quite straightforward.
variable sumsq variable sum variable len_recip variable .mean
: variance sumsq @ len_recip @ s.* .mean @ dup .* - .>s ;
: mean_std len_recip ! sum ! sumsq !
sum @ len_recip @ s.* .mean !
.mean @ .>s variance isqrt ;
```

(I believe that the only performance differences between this and the original version should be:

1. It doesn't use an integer multiply to multiply FRAC by 2 at run-time!

2. There are some functions that would benefit from being inlined. This is somewhat clumsy but doable even if the compiler doesn't do optimization; you just end up with definitions like

```
: s.* postpone u* ; immediate
```

and the like. It's probably also worthwhile to change `32 FRAC - lshift` to `[ 32 FRAC - ] literal lshift` for a similar reason.

3. It uses memory operations instead of stack operations. The original uses eight stack operations; this version uses ten memory operations [plus a stack operation]. Reducing that a bit by using the stack and/or the return stack would be pretty straightforward. Still, I think that after sufficient optimization, you'll always end up with something as ugly and incomprehensible as the original version.

4. It might be wrong, since I haven't tested it, and I assume Yossi's original code was copy-pasted from working code.

)

With regard to locals and performance: on most processors, fetching and storing from global variables is cheaper than fetching and storing from locals, assuming some very minimal compiler optimization (i.e. don't literally push an immediate value onto a stack and then fetch from it; peephole those two instructions together). On most processors these days, the difference is quite minimal.

With regard to GForth vs. Clisp, both are interpreters, but interpreting Forth is a lot faster than interpreting Lisp. A more interesting comparison would be Bigforth vs. SBCL: Bigforth uses a very simplistic strategy for generating machine code, while SBCL has this massive monument of a compiler, but does a lot of stuff at run-time by default. SBCL is very likely to be faster than GForth, even without any declarations, but it might be either faster or slower than Bigforth.

#61 **Kragen Javier Sitaker** on 11.09.10 at 4:00 pm

Hmm, well, the horizontal whitespace didn't make it through, which makes the code quite a bit muddier, but hopefully it's still more readable than the original.

#62 **Bernd Paysan** on 11.10.10 at 5:01 pm

Nice read, thanks for linking some of my material. I probably understand better why Forth is for me - when you describe your layering, your partitioning between hardware (often divided further into analog and digital), software, and algorithm, it reminds me on my role: I do all of that, I don't divide that up between persons - I have coworkers who are specialists and do only one part, but they support me, they don't do the architecture.

The point of taking out the things you don't need is to take out the complexity. This allows you to comprehend the whole project within one person, and that's what makes the result so efficient - no need to have three, four layers talk to each others, with the usual problems communication has - it doesn't scale, and it doesn't work well between different experts. The Forth way is: Remove that constraint, just don't do it.

#63 **Kragen Javier Sitaker** on 11.10.10 at 10:08 pm

Bernd: any thoughts on how to rewrite the standard-deviation code to be comprehensible? Is my version any good?

#64 **Yossi Kreinin** on 11.11.10 at 12:35 am

@Bernd: I work on stuff occupying a few dozens of developers - obviously some of the effort is wasted on friction but I can't think of a way for just a single person to be able to do it all by himself. If I were doing something all by myself, and it involved a complete custom hw, sw and algo design, then I can imagine how dragging C into it could add considerable weight with little gain, though it also depends on experience - I'm very familiar with bootstrapping C, much less so with Forth.

@Kragen: my compiler and target were somewhat nonstandard; in particular, I had constant folding and :inline words, and on the other hand memory was consistently more expensive than the stack.

#65 **Bernd Paysan** on 11.11.10 at 9:04 am

@Yossi: I don't say I'm doing all by myself. I'm doing the *architecture*, the specialists take care to implement a particular function or block, or provide a particular service to the team (verification and ATE tests, project plans and such). This follows the "surgical team" approach as described in "The Mythical Man-Month".

Also, keeping it simple really allows to be a lot more productive. The b16 CPU (a simple 16 bit Forth CPU, very similar to Chuck's 18 bit CPUs, just with a more conventional word width) took a few days to implement - it just has no obstacles in it to stumble over.

Jeff Fox sometimes mentions that Forth can be 1000x as productive as a conventional approach, but I think he misses to point out why. It's three-fold:

* First, you use a 10x programmer, i.e. one that's ten times as productive as the average programmer in the team. As Brooks states, this sort of productivity deviation is usual even within relatively small teams - use it, don't waste talents.

* Then, you reduce complexity by applying the Forth philosophy of leaving out everything you don't really need. This gives another 10x.

* Finally, you can reduce your team by such a great margin, that the friction between team members drops by another 10x (by basically putting all the communication-intensive stuff into one head).

#66 **Kragen Javier Sitaker** on 11.14.10 at 9:25 am

@Yossi: Maybe the lack of a decent variable facility was the problem, then. What did the C compiler do for local variables? Did it allocate them stack slots and use PICK or the equivalent?

#67 **Yossi Kreinin** on 11.15.10 at 1:29 am

@Kragen: regarding the cost of variables, how it changed and what the C compiler did - I recall that it's all there in the article, perhaps in too much detail… Eventually the hardware got to the point where you could fetch a "local" (global, actually) in a cycle - worse than getting an operand right from the stack but same as a stack manipulation word, so it was a question of how many local access vs how much stack manipulation.

#68 **John Cowan** on 11.15.10 at 3:31 pm

If by "CLisp" you mean GNU Clisp, then it's not surprising that floating-point code runs slowly: it uses interpreted floating point for portability. If you meant one of the Common Lisp compilers, then they actually only support C "double" precision.

#69 **Frank** on 01.08.11 at 6:42 pm

I think the reason why the use of FORTH tends to result either in disappointment or amazing solutions is that it forthes you to either come up with an amazing solution, or fail!
Example:
It's a well known maxim that functions should be short.
Forth makes it near impossible to write anything *but* short words!
Ergo: If you hit upon a sweet factorization of your problem and are open enough to throw a few things over board to meet that sweet spot, you win - maybe big.
If you don't? Well, you make do. Painfully.

#70 **Yossi Kreinin** on 01.10.11 at 10:55 am

I do believe that it comes down to throwing a few things overboard, and the question is why one would or would not do it. I wouldn't put it as "being/not being open enough" - which of course is the main point of disagreement here.

#71 **Frank** on 01.10.11 at 8:24 pm

Yeah, that was probably not a good choice of words. Let's say "free and willing to"… throw, that is! =)

#72 **argv** on 01.11.11 at 3:23 pm

wow. there is some occasional lucidity here, as you wade through your thoughts. but there's real honesty throughout. well done.

one point i must raise- i think bacteria are equipped to "conquer space", but just not in the sense we might envision "conquering space". i mean, bacteria can seemingly survive anywhere, why not in space? is surviving in an evironment, and even thriving, "conquering" it? the takeway idea is that bacteria's efficiency allows them to survive in more places than we can. they adapt faster.

1. people are not rational in what they want, e.g., in this case, from an efficiency perspective.
2. people like to mimic each other. they copy each other. independent thinking is rare of not entirely non-existant. so true in computing.
and
3. in life, relationships and keeping people "happy" are very important, not to mention making money. and this is more important than the implications of 1. and 2.

eventually i think more people will start copying chuck moore. eventually. as of now, there is still no pressing need to be more efficient, and a fluff economy has been built around inefficient but widely mimicked and easily marketed abstractions.

will the day ever come when we need to be more efficient? i think yes. eventually.
but for now, carry on. dismiss the thoughts of forth. stick to the familiar. make money. keep people happy.

chuck moore has been offering solutions to a problem in a time when the problem still has not been fully recognised nor appreciated- because there is no pressing need, no crisis. he is like august dvorak, but perhaps less frustrated.

#73 **argv** on 01.11.11 at 3:26 pm

/existant/s/a/e/

#74 **Yossi Kreinin** on 01.11.11 at 11:29 pm

@argv: interesting; the common view is that the need for efficiency in computing decreases with time (somewhat consistently with the relative popularity of Forth), whereas you suggest that it increases or at least is likely to increase in the future. How so? Global clock cycle depletion?

#75 **Mike** on 09.12.11 at 11:47 am

Interesting article. I also like the idea of Forth and LISP. I expect to write a pet project in LISP when I get my motivation back. The difficulty I had with Forth is getting my head wrapped around strings. Numbers were easy. I wrote a large program for a Postscript printer. The Postscript language is similar to Forth.

#76 **Kragen Javier Sitaker** on 09.13.11 at 8:16 am

Yossi: clearly there are more and more things you can succeed at with a computer without being efficient — the things you could already do with a computer in 1998. But there are also more and more things that are possible to do with a computer, and to succeed at the things that are just barely possible, you still have to be efficient.

The particular way in which Forth is efficient is peculiar, though. It's efficient in that it needs minimal hardware complexity to provide a given level of functionality — fewer gates, fewer bits of memory. But that isn't the important measure of efficiency for most applications at the moment. The smallest chip you can fabricate last time I looked was 3mm^2, through the French MPW broker CMP, in an 0.35µm process. That means you have 73 million square lambdas to play with.

Forth's strength is that you can build a CPU in 4000 transistors and run an interactive development environment, or something of similar complexity, in 32000 bits of memory. How does that apply when your smallest possible chip has room for millions of transistors? (Is that a reasonable estimate of how much space you need for a transistor — less than 70 square lambdas? I've never designed a chip, in part because CMP's lowest price was €1950 for one of those 3mm^2 chips last I checked.) Well, one way that can apply is by putting lots and lots of processors on the same chip. That's what Chuck Moore's been trying with GreenArrays (and previously Intellasys). His effort is probably doomed to failure because you have to write all your software from scratch for his chip, and you have to write it to be parallel — both to get more than the single billion instructions per second that a single core delivers, and because each core only has 1152 bits of RAM plus its stacks.

Funnily enough, putting lots and lots of processors on the same chip is also what NVIDIA and ATI do. They seem to be doing pretty well with that approach, even though it *also* requires you to write all your software from scratch for their chips and to be parallel. Two-stack machines like Moore's might be a way for a GPU company to fit more processors on a chip. NVIDIA's Fermi GF100 (year 2010) has 2.9 billion transistors with a theoretical max of 1.5 gigaflops at 1.5 gigahertz on 512 cores, which is 5.7 million transistors per core. If you could cut that down to, say, 64000 transistors per core, you could have 100 times as many cores — which is only a win if the application has enough parallelizable computation to take advantage of them, and if they're individually fast enough.

#77 **Alaric Snell-Pym** on 09.14.11 at 2:03 am

I think you've hit the nail on the head about the problems with FORTH!

I think part of the reason more people try to implement FORTH than try to use it is that they don't like the existing implementations, though. Perhaps gForth can help a bit here, but it runs on "proper PCs", which already have the resources to run "proper development environments".

The place I've really craved FORTH is in embedded programming. There's open-source FORTHs available for a few common chips that could be used, but the chip I was working with was an ATTiny15L AVR, which just didn't have the resources, so it had to be assembly…

Personally, I think that something like FORTH is an interesting model for an intermediate language; stack-based VMs like the JVM are a step in this direction, but providing metaprogramming facilities would enable compilers to generate more compact code (by, in effect, using metaprogramming as a compression engine), enable load-time conditionals (eg, when compiling the code you may not know which optional libraries/features will be available on the target VM, so you can generate VM code that will conditionally compile different code depending on knowledge obtained at that point).

And making the VM language more appealing to human coding will help with writing things like bootstrap code and compiler backends and debugging tools, too…

#78 **Yossi Kreinin** on 09.27.11 at 11:12 pm

@Mike: I think Forth doesn't have particularly good string handling facilities, so I don't know if it's wrapping one's head around them as much as banging one's head into them…

@Kragen: the optimal core size is a very interesting topic; I wrote about it but I'm not sure that I did it very well. Anyway, NVIDIA, IMO, has much less "cores" if you use normal terminology than if you use their own; what they call a "streaming processor" - composed of, say, 32 "cores" - is what most of us would call a single "core" since all the 32 sub-processors execute the same instruction at every specific cycle. So IMO their cores are even bigger than your estimate - and I think it's been the path to success everywhere up until now (picoChip being the one outlier perhaps). So "by default", I believe that few big cores beat many small ones - but I'm willing to change my mind on that one.

@Alaric: I actually think that register-based VMs are perhaps better than stack-based (LLVM is one good one) - stacks give you some sort of code compression but are otherwise gnarly to interpret efficiently. I'm not quite sure why .NET copied the JVM approach here.

#79 **Hans Bezemer** on 11.23.11 at 4:02 am

@Yossi
The beauty of Forth is that you can add whatever you like, including string handling. To prove my point: my 4tH preprocessor is entirely written in Forth and just 16K source (which is pretty long by Forth standards). It features macros (and macros within macros), rewriting rules, token concatenation, token comparison, variables, a string stack, include files - and I'm probably forgetting a few features. I think you need pretty good string handling to handle all that.

#80 **Yossi Kreinin** on 11.23.11 at 4:14 am

@Hans
First, I love Forth. Second, depends on what one means by "good string handling" :) Is C's string handling good? C++'s? Python's? Ruby's? Very substantial text processing systems are written in C (including its own compilers), but I wouldn't call its string handling "good" or even "tolerable" for my typical daily uses.

#81 **Hans Bezemer** on 11.24.11 at 12:30 am

@Yossi
Agreed, unless we have a common standard to determine what "good string handling" is, the discussion becomes pretty fuzzy. On the other hand, I have little idea what your "typical daily uses" are as well. ;-)

#82 **Yossi Kreinin** on 11.24.11 at 3:48 am

@Hans
Well, my standard, off the top of my head, is that I don't want to manually delete unused strings, and I want easy formatting, interpolation, concatenation, splitting, regexps, and, ideally, parsing (yacc-ish or something along those lines). I also want to be able to easily use strings for look-up in maps/hashes/whatever you call them, and similarly easily use them in other sorts of data structures. Symbol support is nice (Lisp/Ruby style, :sym is a unique global evaluating to itself), but, like parsing, I'm used to not having it. Unicode I usually get to happily ignore.

#83 **Hans Bezemer** on 11.24.11 at 9:18 am

- Formatting (check) .r {padding library}
- Concatenation (check) +place
- Splitting (check) {SPLIT BACK tokenizing library}
- Regexps (check) {Wildcard + Char Match library}
- Parsing (check) PARSE PARSE-WORD OMIT - sorry no recursive descent parser
- Associative arrays (check) {hashtable array library}

The point is, those are things I can usually live without. I need tools that can crunch through a large amount of massive datafiles with predictable performance and memory usage that I can convert to a single small executable that I can easily distribute. E.g. I once did a parser that automatically switched to another parser once a certain condition occurred. Easy to do with Forth. I don't need reflection in that case.

I certainly don't like Python scripts that require a Python installation of a certain version with certain libraries in order to run them properly.

I think it boils down more to programming style than to tools that are lacking. I simply solve a problem in another way than you. And may be even different problems. Tools that you find indispensable are of no use whatsoever to me - even worse: they get in my way. And vice versa. But don't blame the tool or claim that it CAN'T SOLVE problems efficiently. It can't solve problems YOUR WAY.

#84 **Yossi Kreinin** on 11.24.11 at 9:29 am

I don't think I blamed Forth or claimed it couldn't do something; and if I had an opportunity to be exposed to people who use Forth efficiently and become a part of that culture, I'd jump on that opportunity. I think what I said amounted to admitting that I wasn't able to pull Forth into my culture and my environment - which I guess is similar to "it can't solve problems my way". I think the last sentence more or less shows we're in no disagreement:

"I find that those ideas grow out of an approach to problem solving that I could never apply" - which of course doesn't imply "…*you* could never apply".

Regarding strings - I do like unused ones to get destroyed automatically, which I think isn't idiomatic Forth, though very likely doable in some way or other.

#85 **John Comeau** on 12.11.11 at 6:54 pm

Yossi, just happened upon your article and loved it. There are probably many, myself included, who wonder if "they" were the "colorforth enthusiast" to whom Jeff was referring.

#86 **Yossi Kreinin** on 12.11.11 at 9:48 pm

Seriously? So you've played with ColorForth?

#87 **Samuel A. Falvo II** on 12.23.11 at 11:37 am

I've used both ColorForth and punctuated/classic Forth systems. I prefer the latter because ColorForth lacks CREATE/DOES>. That being said, though, ColorForth has an _amazing_ quality that makes it resemble orthogonally persistent OSes at the programming level. (Of course, it's not really O/P, but still…)

That being said, Forth is a very libertarian language. Many other languages rule-by-decree what you can and cannot do. With Forth, you're expected to take responsibility for your own actions.

Evidence shows that increasing numbers of people in non-Forth communities are becoming displeased with syntax-driven languages like Java. A recent upwelling of so-called "fluent interfaces" exists precisely to overcome the limitations imposed by decree from language designers. For example, in my Java code, I wrote a fluent module to help me make and verify HTTP requests are working correctly, like this:

new
FluentTestHelper().get().requestTo("http://url.here.com").shouldYieldResponseCode(200).shouldYieldResponseBody(BODY_STRING_HERE).go();

In Forth, I would probably end up writing it like this:

S" http://some.url.com" url get request 200 responseCode BODY_STRING_HERE responseBody verify

It should be noted that words like "url", "request", and so on are just setters in disguise. They stuff global variables, so that words like "verify" have enough operational state to work with.

That brings me to another matter — you were having *the* classic beginners Forth problem — way way WAY too many items on the data stack. You should have no more than 3 items, 2 preferred, at any given time. Ruthless factoring helps, but is not sufficient as you discovered, to ensure this remains true. Remember when Chuck said that he sees the need for global variables, but not locals?

Remember that "global variable" in Forth doesn't mean the same thing as it does in C. With C, once you define a symbol, it remains universally addressible. No means exists for redefining what a symbol means, so you end up having to declare unimportant symbols "static" to keep their scopes roped into their compilation units, and no further. Not so in Forth!!! Consider this code:

```
variable apples
: setApples apples ! ;
: apples ." You have " apples @ . ." apples." cr ;
```

Notice that my colon definition for "apples" obscures the variable with the same name. Formally, this is called a "hyperstatic global environment," which means that I can redefine symbols to mean whatever I want them to mean, WHEN I want them to mean. Through this, Forth can implement information hiding and modularity without actually imposing syntax to enforce it.

Like all things libertarian, though, it can be abused. Be careful, use it wisely and judiciously. But, at the same time, don't be afraid to use it. Knowledge comes with experience, after all.

The trick to writing good Forth is knowing that you can extend the language to suit your problem domain, as you've already identified. I agree that RPN is the best syntax for Forth, but if you are dealing with expression *graphs* so often, you should probably consider extending the language to support infix arithmetic. Chuck says this is bad, but Chuck is only human, and also consider that much of what Chuck says isn't gospel, but taken out of a larger context. And, of course, he can be plain wrong too.

#88 **Samuel A. Falvo II** on 12.23.11 at 11:39 am

Also, I forgot to mention, my blog software is written in Forth. You can learn more about it here: http://www.falvotech.com/blog2/blog.fs/articles/1032

#89 **Peter da Silva** on 12.24.11 at 11:29 am

I think paying too much attention to Chuck Moore or the kinds of Forth enthusiasts who insist on squeezing code down as much as possible is a mistake.

I also think that building a hardware stack machine because you want to use a software stack language is a mistake… one of the best platforms I ever worked on Forth on, the Cosmac 1802, doesn't even have a dedicated hardware stack or subroutine calls.

But the biggest problem is that the most important lesson from Forth is the value of that clever metaprogramming stuff… the process of designing a domain-specific language for your current problem… and leaving that out is going to turn Forth from a joy to a nightmare.

#90 **Yossi Kreinin** on 12.24.11 at 11:25 pm

@Samuel: regarding "fluent interfaces": keyword arguments let you do something like this: verify(url="http://some.url.com", request="get", responseCode=200, responseBody=BODY_STRING_HERE)

This is IMO better than the Java style - which, though it certainly is a sort of an abomination, still has a big advantage over the Forth style where you have no idea who passes which parameters to what. Each of the words you used could leave and consume any number of words on the stack. The stack *is* a global variable, and I don't like its effects on readability very much.

Regarding globals vs locals vs items on the stack - how would you rewrite my examples for better readability? (In my case I also cared about speed and globals were slower than items on the stack, but suppose we mostly ignore this.)

@Peter: I didn't build a hardware stack machine because I wanted to use a stack language - I used a stack language because I wanted to build a hardware stack machine (or rather the hardware people preferred it over building a register machine).

As to the clever metaprogramming stuff - AFAIK, ColorForth doesn't have CREATE/DOES>, which brings us to the question of "paying too much attention to Chuck Moore"… I won't dwell on that one though - what I will point out is that I don't see how my problems with Forth (which are not necessarily representative) would be solved by metaprogramming. My problems were with efficiency & readability of very pedestrian code (BTW, it is reasonable to argue that Forth isn't the best choice if one is so anal-retentive about efficiency, but then when I don't care much about efficiency, I really appreciate amenities such as boundary checking, so I'd rather use some other, safer language and its clever metaprogramming stuff).

#91 **mark** on 12.25.11 at 8:28 pm

Your analogy to bacteria does not work.

Bacteria contain plasmids, which contain genes. These gens can also integrate into the genome of the Bacteria and offer resistances against drugs, viruses (bacteriophages) and so on.

Bacterias do not strive for "simplicity" per se. They strive for maximum reproduction, but that does not mean that the net result is simple or really optimized.

If you are just optimized for maximum growth, bacteriophages will take advantage of you.

And that is why Bacterias are very constrained and die fairly rapidly.

#92 **Yossi Kreinin** on 12.25.11 at 11:02 pm

Well, I don't know much about this, so this analogy wasn't supposed to be something very deep; that said, I believe simplicity is one way to speed up reproduction and speeding up reproduction is one thing they need for survival - I didn't say it was the only one.

#93 **philip andrew** on 12.26.11 at 6:35 am

When I was 14 in about 1988, I found a Emprom chip with Forth written on the top of it, thrown out on a circut board at the University of New South Wales. I opened up my Microbee computer (Australian Z80 based computer), put the chip into one slot, the expansion slot, it fit.

Then I found out how to run it by executing some command in the basic language, then it ran and there was a prompt. I had no idea what it did, so I went to the library and searched for anything on "forth", and found a book! Just lucky there was one book with cartoon pictures inside it as well, interesting.

From there I wrote some basic forth programs, well it was a brief love affair and very interesting and strange.

#94 **Yossi Kreinin** on 12.27.11 at 6:22 am

@philip: cool stuff.

#95 **mpeg encoder** on 04.10.12 at 3:22 am

I have been looking around and trying to find how i can convert verilog code into C++, this is something i would expect from forth programming really

#96 **b** on 04.12.12 at 12:05 pm

FORTH = flexibility without numerous layers of abstraction.

To get up and running with today's scripting languages requires multiple installations of various abstraction layers.

You need an OS, then you need a compiler, then you need an interpreter, then you need libraries, etc.

Even if you have access to the sources for all those layers in the event you need to reinstall them, it takes considerable time to install and configure.

Getting set up with FORTH takes seconds. Boot straight into a FORTH interpreter. Done.

Flexibility.

#97 **robv** on 06.09.12 at 3:56 am

@Yossi needlessly, caterpillars always envy butterflies
maybe i'm in the same class.

i was playing around with Mitch Bradleys forth on the atari in the 80's but i could never figure out the point of Create>Does that could otherwise be done by colondefenition, and to date i still haven't found any explanation that unlocks it for me, even though i know its essential. thus far i got and no further
however,,,,
I'm about to take delivery of a raspberry pi in the next week or so and i want to use its lightness to control a quadricopter out of wi-fi range to do its own thing on a large property in the australian bush like checking algeal blooms in a distant dam or stock movements around gates. etc.

I figure this is a good reqson to take up forth again as i dont feel like shoehorning a linux or python code listing into a small amount of ram to make the copter work out situations that it hasn't been programmed for and fly on regardless (or fly back!!)

good to see interest in Forth hasn't disappeared

regards Chuck's quote about hammers, they may not come with thumb protection but most do come with an UNDO function :)

#98 **Yossi Kreinin** on 06.09.12 at 7:55 am

Cool stuff; I guess I'd use C on the bare metal (no Linux), but Forth is fine, too (small amount of RAM - about how much? IMO Forth really shines if you only have a few K.)

#99 **robv** on 06.09.12 at 3:54 pm

256Mb ram outside of cpu and gpu

my example above was the final one i'm working to
i'll start with something a lot simpler though like a teachable camera tripod head, just replaying various things that i do, but smoother.

yeah i can buy something instead of reinventing, but where's the fun in that?

i'm very much aware of the divide between those who apply forth and those who dissect forth so i'm trying to always straddle the two, hence my putting forth aside when i did. i could easily have gone on but i didn't like the imbalance.

#100 **Yossi Kreinin** on 06.09.12 at 10:09 pm

OK, 256MB is NOT a small RAM (in fact, I don't think I ever worked on an embedded system with that much RAM to date, though it's happening now.) Anyway, good luck :)

#101 **Mike** on 08.02.12 at 3:27 am

Forth is the dodo bird of programming. It is in the process of going extinct both as a language, as a virtual machine, and as a physical machine. Dodo aficionados everywhere may lament it's passing, but there are very good reasons for it's disappearance.

1. Forth is a bad programming language:

Explicit named variables serve the very useful purpose of documenting the intent of the code! But Forth passes parameters on the stack implicitly, which makes even the simplest code segments harder to read and understand. Readable languages win over unreadable languages any day - not only in popularity, but in productivity as well. Sure, you can learn to read Forth, given enough time and practice. But doing so still burdens your mind with keeping track of the state of the stack where a more user-friendly language would free up those brain cells for the actual problem at hand. Even worse, a Forth programmer is by necessity obsessed with redefining the problem until it maps nicely to Forth. You can certainly find a way to do anything with a bunch of tiny functions with no more than three live variables at a time, but it's not a very productive effort.

2. Forth is a bad virtual machine:

Register based virtual machines execute faster on modern machines. Lua's VM is an excellent example. This is because function calls take a lot of time on a highly pipelined machine. When simply fetching the next VM instruction takes a lot of time, you might as well do something worthwhile while you're at it, such as fetch operands from virtual registers. The stack-based VM of Forth has to execute many more instructions to do the same work, and the simplicity of these instructions won't make them go any faster. The capabilities of the host processor is badly utilized when it runs a Forth VM. The situation can be improved by cross-compiling the Forth code to the target architecture, but a register-based VM maps better to the target in this case.

3. Forth is a bad physical machine:

Forth is simply too minimalistic for it's own good. You get more logic gates than Forth knows what to do with with modern manufacturing processes. Even in an FPGA implementation, it's hard to see why anyone should pick a quirky Forth core over a nice streamlined 32-bit RISC design like the Nios II. 32 general purpose registers for your local variables, with shadow register banks for task switching? Or a bothersome stack? Take your pick!

I'm one of those people who may WISH there would be a place for Forth - the simplicity of it is esthetically pleasing if nothing else. But the domain where Forth excels is shrinking steadily. If you're a hobbyist wishing to create a compiler from scratch, Forth is an excellent toy to tinker with. And if you're a hobbyist soldering together your own processor from mechanical relays in your basement, a Forth machine is definitely worth looking into. But other than that?

#102 **Yossi Kreinin** on 08.03.12 at 9:15 am

I agree about "bad VM assuming off-the-shelf modern hardware" - being a good VM for that case was never a purpose of Forth designers, and, well, it isn't a very good VM. I possibly agree about "bad physical machine" - certainly didn't work out that superbly for me - but it's not like I deeply explored the design space so I dunno. The third point - "bad language" - I'd say "not a suitable one for what I'm trying to do" - which is why I didn't explore the physical machine design space very thoroughly, BTW - but, for the tiny minority loving it, I don't think I should consider them self-delusional; Chuck Moore, in particular, is doing rather amazing circuit design all in Forth, and he feels like he couldn't do it in other ways. So here I think, different people are wired differently, and for me Forth isn't a good language and perhaps it isn't for most people and perhaps it's just a matter of education but I don't care since I'm sure not counting on having everyone reeducated; but Forth is, possibly, a real strength multiplier for some people out there who're wired differently from me, and so I wouldn't dismiss it as "bad language" - while I eagerly do dismiss it as one unsuitable for me.

Of course it's not doing very well in terms of popularity and here one question is why bother talking about it; one of my reasons is that I naturally prefer to look at the more offbeat stuff - I intend to write why this is actually sensible some time in the future.

Leave a Comment

Name

Human? (Just type "yes" or "y")

Website (optional)

Submit

- ## Search

To search, type and hi

- **Entries**

- AI problems
- A writer of the lame kind
- Low-level is easy
- Blogging is hard

Categories

- bastard
- hardware
- numerical
- OT
- software
- wetware

Blogroll

- Development Blog
- Documentation
- Plugins
- Suggest Ideas
- Support Forum
- Themes
- WordPress Planet