Why I Use Forth

First, there are at least two sorts of reasons:

- the real reason, which happens to be some accidental coincidents,
- and the motivations, which are "reasons" my brain thought of to show that it has full control (which it hasn't).

Accidental Coincidents

In 1986, I bought an Atari ST. Compared to other home computers of that time, it was bleeding fast, and had a wonderful GUI. It had virtually no software, but this was supposed to change. The main reason I bought it was to become a hacker (in the sense of ESR's "New Hacker's Dictionary"), and the main excuse was to computerize a comic strip a friend and I had been drawing for three years.

Well, there was nothing like a comic construction set or such like which would allow me to draw the comic on the screen. The only two programming languages I got with the ST were ST Basic and Speed Logo. ST Basic was soon forgotten, and Speed Logo was dog slow, and a toy. I used it three months to learn programming, though, and I wrote two little games in it.

After three months, I decided to get a real programming language. There was the C development system, which was very expensive, needed a hard disk (very expensive, too), and in fact, I didn't have that money.

However, I read that there was a PD Forth system (volksFORTH), and it was considered to be a machine oriented language, and since it won't cost me anything except the floppy (10 Deutschmarks! They were expensive those days), I got it. I got a book about Forth, too (Leo Brodie's "Starting Forth"), and I liked that language from the start.

I soon started to write my "Comic Construction Set" (CCS), and it turned out that I had to do a lot meta-work. There was a GUI, but it wasn't easy to program. So I wrote a half-ways object oriented package to cope with it. This brought the program forward, and soon it could handle multiple windows, user defined objects, and many things I needed. I've even drawn some "art" to test the program. However, the work on the comic strip stalled in the meantime.

After my window library stabilized, I sent it to the authors of the Forth system, because I though this would bring Forth forward, and I felt obliged to give the hacker community something back. It was the only major contribution the authors got, and since their motivation had erroded quite a bit, instead of incorporating my changes into their system, they gave the rights of what they did to me. So I had an improved 32 bit native code system with some flaws (volksFORTH was 16 bit, threaded code), no documentation and some work to put into it, and I was alone with it.

I decided to make it commercial, as the experiences of the PD authors were not very good. People seemed to want printed documentation (I never had any documentation for the system except the source), and therefore they must pay something, anyway.

Therefore I stalled the CCS until I got the new system to a state where it would be ready to use. Thus I

never looked back. And that's how I became a Forth implementer by accident. At least I didn't want to write my own Forth, and in fact, I never wrote a Forth system from scratch.

Motivations

Like Basic and Logo, Forth is an interactive language. You have a command line and you can just type commands in and try them out. This is very helpful for the beginner and the impatients, and it turns out that it is helpful for everyone, especially for debugging. You don't need to write a test program to test your program.

Unlike Basic and Logo, Forth isn't slow. Though it has an interactive command line, programs are really compiled to quite efficient code. Forth programs are also called "words", since each program has a name, like a Logo/C/Pascal function/definition. You call it just by typing in that name. There is no difference between words from the language and words you defined yourself, you can even define words using an inline assembler. The whole system is written this way.

In this respect, Forth is quite similar to Lisp and its descendents, like Logo. With my three months Logo experience, it was easy for me to pick Forth up.

Forth differs from Lisp in that it doesn't use lists, neither for calling, nor for storing multiple values. Forth uses a stack to pass data between words, and it uses the raw memory (as seen by the assembler programmer) for more permanent storage. It's much lower leveled than Lisp, and that's one of the reason why it is fast. It's not only fast, the simplicity makes it very small, too.

There's another feature in Forth which is almost unique: the compiler for definitions (or words) is not a single program, which handles parsing, control structures, statements and so on; it's split up into a simple loop which does parsing, and Forth words which are responsible for control structures. As said above, there is no difference between Forth system words, and user defined words, thus you can write your own control structures.

The same is true for data structures. Plain Forth has very few data structures, and it's hard to call them "structures", since they aren't. Forth has just a way to allocate some bytes of memory and assign them a name (which returns the address to allow operation on that memory).

The power comes in that you can bind some Forth code to the label. You can do almost anything with these enhanced labels; you can build arrays, you can even implement an object oriented extension to Forth, where each object, and the classes themselves are just user enhanced data structures.

Plain Forth doesn't have many abilities. It's always much work to get a complicated job done. However, Forth is the ultimate language for building extensions. Programming in Forth is generating higher levels of abstractions, until you have a language well fitted to solve your problem. The simplicity of the underlying system allows it to rely on it, which is important when you search bugs. The usual approach for application programming is to keep each layer simple, too. This is essential for rapid development of critical applications.

The strengths of Forth have therefore been mostly used in embedded control. I've never used Forth in embedded control, although a good customer uses my system to control chromatography systems. For me, Forth is the answer to the question "general purpose language", especially because of it's layered programming approach.

Why not use Forth?

There are also reasons not to use Forth. One of the most important reason is that Forth is said to be an amplifier. A good programmer programs better in Forth (and he learns to program better in other languages, too, when he masters Forth); a bad programmer programs worse in Forth, and he's spoiled when working with other programming languages, then. The sad fact is that too many programmers must be considered "bad".

I haven't believed this for quite a long time. I found quite a range of programming skills in the Forth community, and certainly I could give them labels such as "excellent", "average", and "bad". The sad thing to say is that this was all relative. Most of these programmers, whatever rank I gave them, had done a lot of very clever things, including those with the label "bad". Those which were average programmers hadn't coded anything in Forth, and therefore hadn't a label.

The Forth approach of layered programming is inherently difficult to grasp. People tend to think linear, and on the same logical level. You don't get anything done with a linear approach, since you won't have the tools you are used to. There aren't much libaries in Forth, as Forthers like it more to reinvent the wheel instead of designing one and make this one reusable. Special purpose wheels tend to be written much faster, use lesser resources, and fit to the applications. And it's easy to get a state-of-the-art wheel this way.

So IMHO Forth is an elitary language, designed and written for the best, or gifted, not for the masses.

It should be noted that the layered approach of writing programs is acknowledged elsewhere, too. However, it's usually considered that one programmer should work in one layer, i.e. write a library, an application or a user interface. A Forth programmer has to move between the layers, as a Forth program consists of several layers, even if it uses an already written library, and someone else writes the user interface.

Forth strongly encourages the hacker's paradigm of "plan one to throw away". It is often cheaper to rewrite a program once you understood what you really wanted instead of living with something that is broken by design. The assumption that it just needs someone clever enough to do the design right in the first place usually is void - noone can look into the future, and without the experience of using the beast, nobody really knows what to put in.

Anti-Elitist View

The elitist view above needs some comments: There have been successful attepts to learn Forth to childrens who have learning difficulties. These childs have shown that they can use Forth. They even use Forth in a professional manner, i.e. they write short words which do one thing right, and without any comments except stack effect and word name. It might take them longer than a Forth professional to find solutions, and they might not find as clever solutions, but they find solutions, and some are quite interesting. This leads to a quite different conclusion:

Forth is just different. Many people who know other languages say that Forth does everything upside down. The childrens mentioned above have difficulties with math - they choose the Forth course only if they are assured that they don't have to do much math in it. That also means that they probably never were good at infix notation, or that they know that by heart. They also haven't seen any other programming language. For them, Forth's way to do things is the only way they know.

Fun View

There's another view, posted by Travis S. Casey in comp.lang.forth. I don't change his wordings, since I can't say it better.

Something that strikes me is that every "non-mainstream" language I've ever looked into has claims that it "makes programmers more productive", with data to back it up. (With the obvious exception of languages that are intended to be hard to program in, like BrainF*ck.)

Recently, though, I came across an article that suggested something that makes a good deal of sense to me - that what increases productivity isn't any particular language, but "fanatic" programmers. That is, that you'll see higher productivity from people who **enjoy** using the language they're using.

This, of course, makes perfect sense: happy workers are better workers. It also explains how all these non-mainstream languages can claim better productivity; in general, the people who use non-mainstream languages are using them because they **like** those languages. In contrast, many people writing in mainstream languages are doing it not out of any particular liking for the language, but because that's what they're being required to use by management, or because that's what they think will sell.

It also explains hackers being more productive than "professional" programmers. (Using the traditional definition of hacker as someone who programs because they enjoy it, rather than the more recent definition.) Since hackers enjoy programming, of course they're going to be more productive than someone who's just doing it as a job.

In addition, I'll note that some people find that certain languages just suit them - they find it easy to think in the way that language works. Again, it seems natural that someone working with a language that "fits them" is going to be more productive than someone who's not; and here as well, non-mainstream languages seem likely to have an advantage, since many people using them are doing so because they've found that the language is easy for them to work with.

The simple upshot is: there is no "best language". Use a language that you find yourself enjoying programming in, and find it easy to program in.

Forth Future

We are now in the cambric explosion of language evolution. Lots and lots of programming languages, nice and ugly, are created out of the dust. Many are short-lived, even the important ones ride on the waves of fashion. We have languages with hard shells, without shells, mobile and immobile. There's one little language with a backbone, but without much flesh, that's Forth. If it will survive, and render flesh and shells when necessary, it will take over the world. It might be swallowed before that, but it has survived long enough to make this unlikely.

Appendix: What's a "bad programmer"?

A few people expressed doubt if they are bad programmers, or if they shouldn't try Forth because they might be part of what I considder "bad programmers", so here's a small test if you are a bad programmer. The more of the following answers you answer with "yes", the worse you are. Don't feel bad if you have a few "yes", nobody's perfect. I added also a few questions where a "no" should be quite sure:

- When you program, you wince at least once every minute, and not just because your tool crashed or displayed other bugs, but because you stare a the blank screen and don't know what to enter.
- Extra points if you avoid this by pretenting to do "design" before coding, and you just produce tons of documents and UML diagrams without ever touching the keyboard and leaving the dirty work of implementing your designs to other people.
- When you have a problem, you invite for a meeting. The more people you invite here, the more points

(and pointy hairs) you get.
- You think that a team of 10 programmers can solve tasks 10 times as complex or in one tenth of the time. Bonus points if you think 100 programmers scale as linear. Point dropped if you believe that because a set of 10 programmers contains one which is capable to outperform the average programmer by a factor of 10. Tell me how you prevent the others to mess up the job of the elite one.
- You always ask you what's so funny about these "Dilbert" strips your coworkers put on their cubicle walls. Extra points if you follow the advices in "Dogberts top secret management handbook" when managing the dolts below you.
- You do programming for money; doing it as a hobby (for fun!) is something that only madmen do.
- You believe that programming can't be self-taught, and titles are worth more than experience.
- You believe that your self-taught skills are better than anything you can learn at a university. Remove that point here if you believe that because you are the archcancellor of an elite university, and therefore know what you are talking about.
- You can't grasp how this free software, or as you call it, "Lie-nuts" system came around, other than being either all lies, or created by nuts. Bonus points if you believe both.
- Your programs tend to become an unmaintainable mess of spaghetti code after 1000 lines (add bonus points if you reach that state earlier, reduce if you reach it later, "master of bloatware points" if your programs exceed 100klocs and are still maintainable). Extra points if your programs grow wild even after they have become unmaintainable.
- You didn't know what to do with these rectangular Lego bricks when you were a child, and rather played with prefabricated toys.
- You think that one programming language (and maybe assembler) is enough to be a competent programmer. Bonus points if you believe that knowing a programming language isn't necessary at all to be a competent programmer.
- You still believe that if you think hard enough you can produce correct non-trivial code without testing, though testing in the past always proved you wrong.

Created 30dec1997. Last modified: 13feb2010 by Bernd Paysan