What Computers Can't Do (And Why)

Computing is approximately 35 years old. Looking back at its progress, we can ask how much it has advanced: How much better are our facilities now than 35 years ago?

Not much.

Many improvements have been made in both hardware and software; home computers are affordable; most companies use them daily. But the progress made in computing has been akin to the improvements made in horse-and-buggy technology over the centuries—impressive but not revolutionary. Nothing in computing rivals the leap that the internal combustion engine represented to transportation.

1 What Computers Ought To Be Able To Do

Philosophers debate the limits of computers: Can they think, are enormous programs impossible to write, can a computer-generated proof be trusted? We only need travel close to home to see their limits. Here are some examples of the things that computers ought to be able to do but cannot.

A computer is not aware of what it and I are doing—not in the sense of artificial intelligence, but in the sense of knowing that the current task, for example, is to write a document using a text editor, a typesetting program, and a laser writer. If there is a set of typesetting macros I generally use, the computer should know them. If I walk away from my computer for a couple of hours, it should be able to later remind me: You were editing a file, running a typesetter over it, and printing the results.

Computers have no idea what is going on. You can't hold a reasonable conversation with them, even on their own terms. Does this scenario look familiar?

```
% lpt /usr/fred/common-lisp-functions
lpt: Command not found.
% lpr                                              <long pause>
/usr/fred/common-lisp-functions                    <another pause>
^C
% lpr /usr/fred/common-lisp-fucntions
lpr: cannot access /usr/fred/common-lisp-fucntions
% lpr /usr/fred/comon-lisp-functions
lpr: cannot access /usr/fred/comon-lisp-functions
% lpr /usr/fred/common-lisp-functions               <typed slowly
                                                    and with care>
lpr: cannot access /usr/fred/common-lisp-functions
% ls common*
% ls /usr/fred/common*
/usr/fred/common-lisp-fns
% lpr common-lisp-fns
lpr: cannot access common-lisp-fns
% /usr/fred/common-lisp-fns
/usr/fred/common-lisp-fns: Permission denied.
% lpr /usr/fred/common-lisp-fns
%                                                  <success at last>
```

Maybe this looks like a novice floundering around, but the same conversation rendered into English might be worthy of Monty Python's Flying Circus:

Customer: I would like a pound of roquefort cheese.
Shopkeeper: Oh, I'm afraid we're out of roquefort cheese today.
C: Mozzarella?
(The shopkeeper is silent.)
C: (after pause) *Do you have any?*
(More silence.)
C: (aside) I forgot! The twit insists on complete sentences!
C: (to shopkeeper) Do you have any mozzarella cheese?
S: I don't have any, personally.
C: I want to buy some chee... (he sneezes)
S: We don't sell sneezes here.
C: (slowly and firmly) Sell... me... some... cheese!
S: What cheese?

C: What cheeses do you have?
S: I don't have any, personally.
C: (he consults a phrase book) What cheeses are for sale?
S: (he looks) Ummm... just cheddar.
C: Please sell me some cheddar!
S: Cheddar what?
C: Cheddar *cheese*. S: What about it?
C: *PLEASE SELL ME SOME CHEDDAR CHEESE!*
S: (cheerfully, as if nothing had gone on before) That'll be two quid.

I really hate having to tell a computer all over again something I just told it. Even worse is having to tell it something *it* just told *me*, like a sixty-character file name.

If I have a favorite way of interacting with programs, I cannot easily transfer that interface to new computers. In fact, I cannot usually customize the user interface of a general program. For example if I like to use text editor commands to move around the cells of a spreadsheet, I cannot unless my desire was foreseen by the spreadsheet authors.

I cannot easily write a program by composing three existing programs unless the composition is trivial. I cannot easily use parts of already-existing programs in new programs.

Early cameras required a lot of detailed knowledge and talent. One had to know about chemistry, needed to know how to use sophisticated light-measuring devices, and one had to use a finely honed talent to trade off shutter speed against aperture. Excellent photographs could be taken only by the skilled and knowledgeable. In recent years camera designers have concentrated on making beautiful photography easy—ordinary people can taken excellent pictures whose beauty is limited by only the photographer's taste and artistic ability.

Why can't there be an analogous story for computers?

2 Why Are Computers So Bad?

Computers were once only fancy adding machines. The first programming languages provided mnemonic names for machine instructions, symbolic names for addresses, automatic subroutine linkages, and "synthetic computers." A synthetic computer was a non-existent computer that had the

features you wanted and for which you wrote code; an "automatic programming" system translated your program into the machine language of the real computer. These synthetic computers had such things as more registers than the real computers, and floating point arithmetic. Many believed that computers lacked the dexterity and imagination to "write programs" in such a way, except for isolated examples.

Fortran was invented for these glorified adding machines. Performance of compiled Fortran programs on mathematical programs was the primary design goal; language design was an afterthought. Because the prevailing attitude was that computers could not write programs, the goal of high performance was not simply a wish. It was necessary for the survival of the idea of a non-octal programming language.

Fortran became the most prevalent programming language, and much software was produced in Fortran. Hardware designers chose as their goal the production of faster computers that descended from these early ones and which were also suitable for running Fortran programs fast. Fortran and Fortran compilers were subsequently modified and improved to further take advantage of the new hardware.

The Algol model of computation was strongly derived from Fortran, and the currently popular programming languages—Pascal and C—are descendents of Algol. Good Fortran machines make good C machines.

There is a fierce standoff: programming languages emerge to take advantage of the speed of existing computers, and computers emerge to better be taken advantage of by existing programming languages.

Early operating systems were designed under similar constraints—they had to be small and out of the way of the user's program. Therefore, operating systems enjoyed minimal functionality, relied on heavy re-use of machine language code, and did not export their facilities to user programs in a convenient manner.

Because computers were slow, the operating system had to be very efficient to stay out of the way of user programs. Therefore, the most important design consideration next to size was speed. Is it any wonder that the early operating systems and their kindred spirits in use today have little functionality?

3 Traditions

Traditions are important in both culture and science. Tradition controls what we can believe, what we can do, and the thoughts we can have. In science, it controls the kind of work we can do. In computing it controls the kinds of computers, the kinds of operating systems, and the kinds of programming languages we can have.

Because the tradition of small and fast has been strong for the last 35 years, the computers, operating systems, programming languages, and programming methodologies we have are very much like the ones 35 years ago.

4 Commerce

Computer companies design complex and expensive computers, and they want to expend the least amount of time and money in making them usable. Furthermore, computer companies want to show off their hardware in the best light, which means small, fast operating systems that are just barely enough to do the job.

Computer companies do not want the cost of software to hamper the hardware sale, so they either price the software low or give it away. Today it is common to see hardware manufacturers giving away operating systems, compilers, and many utilities.

Even a company that wants to charge for software is in a quandary as to how to price it. Software is not tangible in the same way that a computer is: You cannot trip over it, you cannot weigh it—but you can accidentally erase it.

If the hardware manufacturers have no incentive to improve software technology—almost by tacit agreement—then it is up to the independent software vendors. But these developers are competing against the hardware companies, who have already de-valued software. Because they compete against the hardware companies and against each other, they must offer some edge over their competitors: Either they can lower prices or offer increased functionality. The latter is a pure risk while the former is well-understood business practice. What we see is exactly what is expected: Most companies compete using price, and some make modest technological advances.

5 Can the Situation be Improved?

Tradition and commerce form a blind conspiracy to keep the situation constant. Hardware designers have an easier time than software developers, because they must simply improve the performance of simple computers, computers just like the ones that filled rooms and multiplied a thousand times a second using one thousand memory locations. Computers are ten thousand times faster and ten thousand times bigger than they were when the first programming language was invented. Are programmers, or for that matter non-programming users, ten thousand times more productive? And if I gave you a time machine and the ability to take back with you either everything you know about hardware advances or everything you know about software advances over the last 35 years, and I suggested you make yourself rich, which would you take?

Computers are as valuable as what they do, and we see they are less valuable than they could be. John Backus is the father of Fortran, the language that froze computer architectures to this day. But he wrote in 1981:

> While it is perhaps natural and inevitable that languages like Fortran and its successors should have developed out of the concept of the von Neumann computer as they did, the fact that such languages have dominated our thinking for twenty years is unfortunate. It is unfortunate because their long-standing familiarity will make it hard for us to understand and adopt new programming styles which one day will offer far greater intellectual and computational power.

God bless him.

rpg
gls