Contents

1 Operating systems **2**
1.1 What happens to your input — inside the operating system 3
1.2 What happens to your input — inside Prolog 3
1.3 Who are you talking to? 4

Who am I talking to? — operating systems and the computer

Jocelyn Paine

February 15, 1996

In this handout, I say a bit about how the computer system is controlled, about why you need to hit RETURN after each line, about the different programs you may be talking to, and so on. To run Eden, it is not essential that you know this material. However, I do find that beginners are sometimes confused by the number of different programs they have to interact with when doing the course. If you're not used to computing, it's easy to believe that "the computer" is a single-minded entity, speaking one language that is the same in all situations. In fact this is far from the truth, as I go on to show in section 1.3. The system comprises a host of different programs. These all expect different types of input, and the grammatical forms appropriate for one will not suit another.

One thing that most of these programs do have in common is the need for you to hit RETURN at the end of each line. Beginners often forget to do this. In section 1.1, I explain why it's necessary to do so, and what happens to your input if you don't.

1 Operating systems

When you log into the VAX, you get a start-up message saying something like `This is VMS version 5.1`, together with a lot of extraneous garbage. Modern multi-user computers all come with a master control program or *operating system* which sits in the machine continuously looking after all the other programs. On the VAX, this program is called *VMS*.

Generally, such computers are designed so that they can only run one program at a time. To give the appearance of dealing simultaneously with the people in this class plus all the other users elsewhere, the operating system *time-shares*. It picks one terminal, sees which program is to be run there, and runs it for a few milliseconds. It then interrupts this, saving the state the program had got to, and picks another terminal. It runs that program for a few milliseconds, interrupts it, saves it, and picks yet another terminal. Eventually, it gets back to the first. If each program is only run for a very short time on each round, you get an effect like that of film frames, where the operating system comes back to you so frequently that your program appears to be running all the time. This resembles some theories of attention in psychology, and indeed they were probably inspired by this idea of a master controller sharing out resources amongst lots of subordinate programs.

If there are many other users, the operating system can devote less proportion of its time to each, so the programs appear to be running slower. This happens especially in mid-morning and mid-afternoon, and accounts for all those irritating pauses.

1.1 What happens to your input — inside the operating system

To remain in control — to give everyone a fair share of the machine — the operating system has to take charge of everything that goes into it, including your input. Generally, when you press a key, the character you typed gets sent to a storage area called a *buffer* inside the operating system. When you press another key, the same happens. It continues happening until you hit RETURN. When this happens, the operating system has collected a whole line of input, and it now sends this all at once to your program.

Why does the operating system store input in this way? Partly for efficiency. It's quicker to batch up the characters and send a whole batch to the terminal in one go than it is to send them one-by-one, because there's a constant overhead associated with every transmission. Partly for historical reasons, in that early computers worked on punched cards, and programming languages were designed to fit one sentence on each card. With such languages, you might as well wait until the whole card has been read before sending any of it.

The important point then is that, most of the time, whatever you type gets held by the operating system, and will not get to Prolog (or whatever program you are talking to) unless you press RETURN. You can visualise this if you imagine each character moving along a conveyer belt from the terminal to the operating system. The latter puts each character into a box. Only when it sees a RETURN does the system send the entire box (the line) on to Prolog.

If you forget RETURN, it is easy to think that Prolog is just being very slow to deal with your input, when in fact it never even receives it. This is why I keep emphasising the need for RETURN.

Having said all this, it's worth noting that programs can, if absolutely necessary, bypass the buffering process and respond to characters as soon as they're typed. Because it's inefficient to do so, most programs don't bother. However, some do. These include editors (such as Ved), where it's essential that you see each character as soon as you've typed it. The Eden "manual bug" does the same — you can type actions such as `f` and `b` without having to hit RETURN afterwards. When learning to use a new program, always make sure you know whether RETURN is needed after a command.

1.2 What happens to your input — inside Prolog

When the input reaches Prolog, there are more complications caused by the fact that Prolog commands need not be exactly one line long. If they were, it would be easy to tell where they ended. But Prolog is designed to be *free format*, so that spaces and newlines between words have no significance. Thus you can write either

```
eps(psbug2,psworld2).
```

or

```
eps( psbug2,
     psworld2
   ).
```

or even

```
eps
(
psbug2
,
psworld2
```

```
)
.
```

When typing commands, you normally won't want this freedom, and will use as few spaces as possible. However, when writing *programs* (such as sets of production rules), it's very useful to have it. Indentation and spacing form valuable psychological cues to a program's structure. Hence, although the older card-based languages did not allow free-format input, all the newer ones do.

This means that the only way Prolog can tell where a command ends is by looking for the dot at the end. And it uses its own buffer to do this. As soon as it receives a character from the operating system, it adds it to the buffer. It keeps doing so, until it sees a dot. It then takes the entire contents of the buffer to be the next command, and processes that.

The upshot of this is that even if you hit RETURN, if you forget the dot at the end of the command, Prolog will again just keep waiting.

Incidentally, the operating system does not need dots at the end of its commands. It assumes that all commands are one line long unless you tell it otherwise. If you type a dot, it will be treated as an error. You may also have tried the Eliza program. This assumes that all sentences are one line long, and hence doesn't need a terminator either. However, the operating system and Eliza both require you to hit RETURN after a line.

1.3 Who are you talking to?

A common source of bewilderment on this system is that although you are communicating with the same computer all the time, you are not talking to the same *program* all the time. Beginners often talk about "the computer", as though it were one indivisible entity. It is more accurate to think in terms of multiple personalities, each program having its own personality, memory, and language, and knowing nothing of the others sharing the same machine.

So although the operating system and Prolog both share the VAX, they follow completely different conventions. Prolog needs sentence terminators; the operating system does not. Beginners will often type

```
eden.
```

to the operating system, and get understandably confused when it complains about an unwanted dot. Similarly, some people forget to start up Prolog after logging in, and try typing Prolog commands at the operating system, provoking cascades of error messages. To get round this, the best advice is: watch the prompt! Prolog's prompt is `?-`; the operating system's is always `$`.

The networks and intermediate stages are another source of bewilderment. When you start up the terminals in CTC, you begin by talking to a program running on the micro-computer there. The commands you give that tell it to connect to the University master switchboard. In turn, you must tell the switchboard to connect to the VAX. Then you tell a VAX LOGIN program to log you in, typing your username and password; then you tell the operating system to start up Prolog; and finally you talk to Prolog. So you have all the sub-systems shown in figure 1 to deal with.

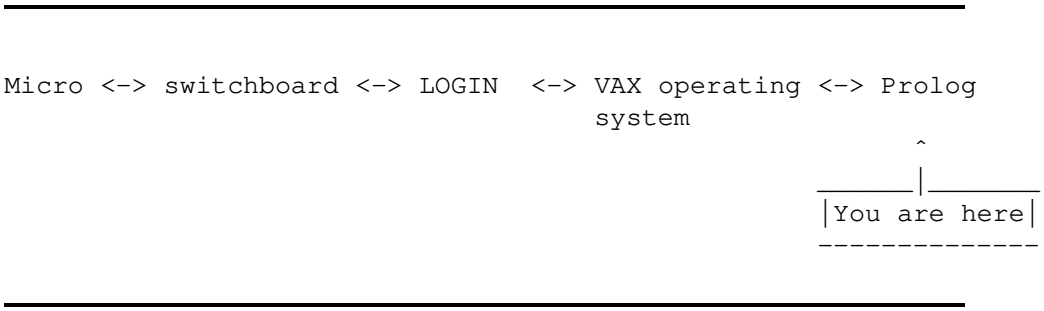

Figure 1 Subsystems on the VAX.

It is difficult at first to remember which sub-system you are talking to at any stage, but practice makes it all seem easy, eventually.