JOB-CONTROL LANGUAGES

WHAT ARE THEY ?

A language is a means of conveying information to somebody or something; it doesn't have to be text or speech, or even necessarily be parsable into words and sentences. Many deaf people can communicate very effectively by signing; at a very much lower level of sophistication, we all communicate by body language.

In computer terms, many programming languages, including traditional job-control languages, are expressed in textual forms which depend strongly on our experience of using language in everyday life : they are formalised versions of written English (usually, for historical reasons, but any language would do). As well as that, though, we communicate with computers through menus, WIMP interfaces, touch screens, pen interfaces, and so on. These are effective languages too; there is clearly a lot more to be said about computer languages than is accounted for by the Chomsky hierarchy. (Chomsky is a linguist, but concerned essentially with the development of conventional language.) These other languages can be seen as more akin to manual sign languages than to written and spoken English.

Here, we shall try to cover all the main sorts of Job-Control Language (or JCL), so we'll certainly fail. That includes textual and WIMP languages, and both interactive control of the system and control through command files. Try looking for gaps in our arguments, such as they are, and fill them in from your own experience. There's no prize, but if you find anything interesting, we'd be happy to hear about it.

WHERE DID THEY COME FROM ?

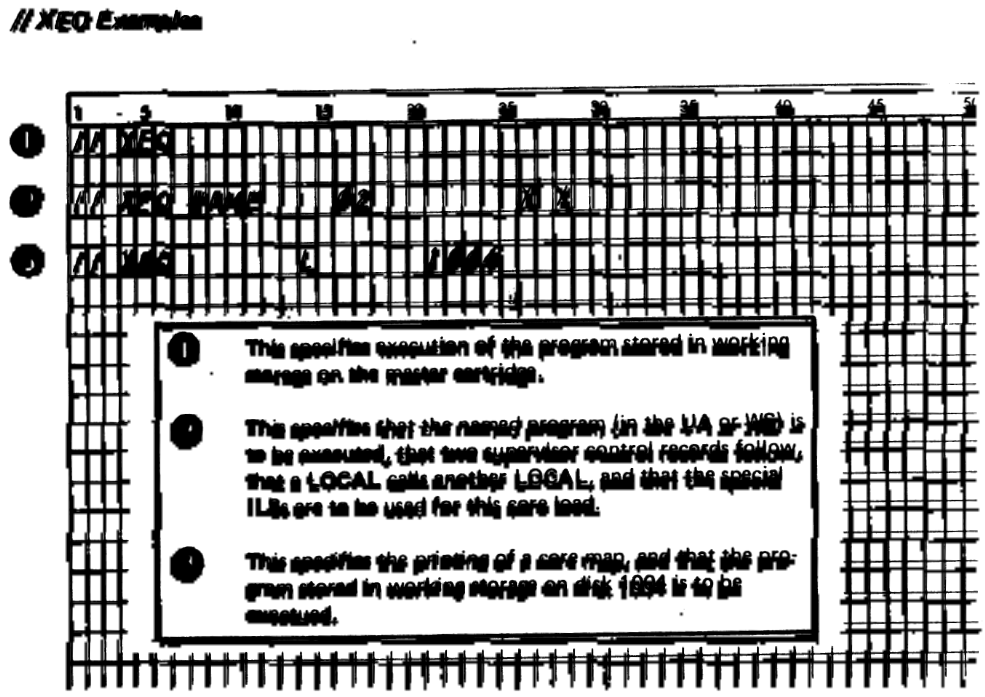
They were originally *monitor control cards.* These were used to tell a simple monitor system what to do next; they were inserted at appropriate points in the deck of cards which made up the job, and allowed you to give instructions such as COMPILE, RUN, RENAME … – but typically in more cryptic form. As monitors maintained no notion of a continuing job, there was very little in the way of conditional control.

As batch jobs became more elaborate and more demanding, the JCL had to grow. Classical control structures were found to be useful (loops, conditions, subroutines); and some sort of job variables were needed if only to carry information through the job.

With interactive systems, the JCL contracted again. The system no longer needed to carry the information, as there was always somebody there to do it; similarly, any necessary decisions could be taken as they arose, so elaborate programmes which catered for all contingencies were no longer required. The instructions with which you control a text-driven interactive computer system are much like the old monitor control cards, though they're often much better adapted to use from a terminal; more formal JCL programmes are still used for command files and for controlling batch jobs.

WHAT ARE THEY LIKE ?

ASSEMBLY-LANGUAGE type : Here's an excerpt from a manual[REQ2] :



The example shows the typically cryptic instructions, often requiring data to be put into special positions in the record. The JCL programme was (almost ?) always executed interpretively, with no correlation between different parts of the job; syntax errors within instructions were therefore found (once the interpreter reached the offending statements), but consistency between instructions could not be checked. These were characteristic of monitor systems, but their influence persisted for a long time.

Notes :

The "master cartridge" is whatever cartridge was loaded on the default disc drive.

"Working storage" (WS in note 2) and "UA" (standing for "user area") are areas on the disc. There were only two, apart from space reserved for the system - the user area stretched from the first disc sector to the end of the allocated disc space, and working storage was the rest. The user area was always kept compact, so when you deleted a file all the files beyond the point were copied backwards. Executable programmes were built by the system linker in working storage, and could be run from there without formally declaring them to the file system.

A LOCAL is a sort of system subroutine which runs in a special overlay area - it doesn't mean "local", but "LOad on CALl".

An ILS is an interrupt-level subroutine.

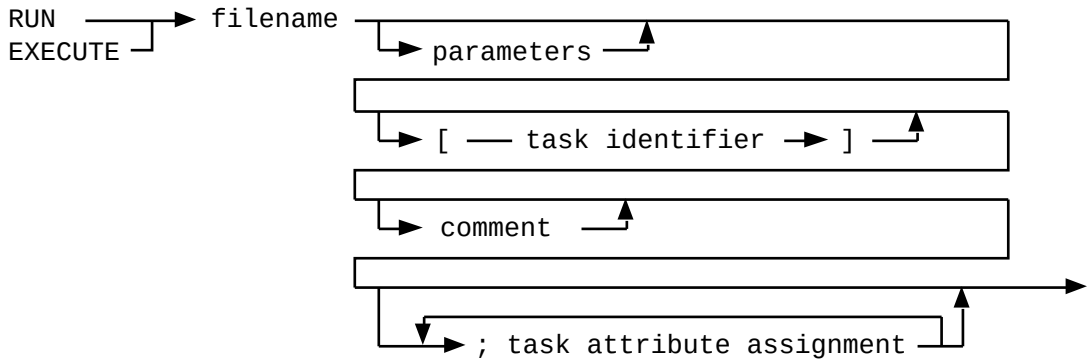
"exectued" is a misprint.

HIGH-LEVEL type : For contrast, here's an extract from a different manual of about the same vintage, describing an equivalent instruction. This one (Burroughs WFL [REQ3]) shows Algol-like structure and free form text; as an additional curiosity, it was compiled, not interpreted :

Example :

```
RUN X;
RUN A/B [T]; STACK = 500;
RUN A/B (1, I, FALSE, 3.14, 3"123", "HI THERE" )
    [T1]; FILE C := G; BDNAME=X;
    FILE F( BLOCKSIZE = 30, KIND = PETAPE ),
        G( KIND = DISK );
    FILE H = ABC/D DISK;
RUN A/B THIS IS A COMMENT; VALUE = I;
```

Run Statement :



Notes :

In the first instruction :

X is a file name, here used to identify a programme.

In the second instruction :

A/B is the programme file name.

T identifies a task variable, associated with the execution of A/B. This is essentially a *process control block*, which will turn up again in the *EXECUTION* section.

The STACK attribute of the task is defined; it governs the stack size for the task.

In the third instruction :

A/B is the programme file name, followed by a collection of parameters. (The syntax description isn't very systematic; the parameter parentheses are defined in the further syntax diagram for parameters.)

T1 identifies a task variable, associated with the execution of A/B.

C, F, G, and H are the internal names of files used by A/B. C is identified with a global file called G, and certain attributes of the others are defined.

In the fourth instruction :

A/B is the programme file name, followed by a very Algol-like comment. (The compiler knows it's a comment because it can't be anything else.)

The task's VALUE attribute is set; this can be retrieved from inside the task's programme, so provides a channel of communication between operating system and programme.

GRAPHICAL languages : instructions are given by moving a cursor of some sort around a screen with areas predefined to have some significance. Menus and WIMP interfaces are commonly used. (Menu interfaces can be used with character terminals too.)

COMPARISON.

The assembly-language sort of JCL has (we think) died. It was very easy for the system software to interpret; as everything was in a specific place, no parsing was necessary. (Could one suppose that, in their reliance on position, these languages anticipated the graphical languages ?) This ease of interpretation was very important when the language had to be handled by a resident monitor system, and space was at a premium. It was very hard to enter from a terminal – but comparatively easy from a card punch, which you could programme to tabulate to specific column positions.

Languages of the high-level type, with or without advanced features such as conditional statements and iteration, are still with us (though still not often up to WFL standard), and works well. They are in competition with graphical interfaces. You need a lot more software to drive a graphical interface effectively, but unless you're willing to waste a lot of processor time (which perhaps you are), they can be quite slow. (Comment from a colleague on first seeing a Macintosh : "It's a great achievement to make a 68000 run so slowly !" - but it's a long time since Macintoshes used 68000 processors.) So far, the only realistic way to write a script for interactive or batch use is to use a textual language, though things are changing; see our comments on "Applescript" later on in *SESSION LOGS AND COMMAND FILES*.

WHAT SHOULD THEY BE LIKE ?

- which is to say, what facilities do we need in a job-control language in order to be able to use it effectively ?

Obviously enough, that depends on what we mean by "effectively", but for once there's an easy answer : we should be able to use the language directly, through a terminal, to make the system do anything we're entitled to do with it, and in a command file to programme any sequence of actions which we might want to take if we were watching the job proceed in person. That specification for the command file includes a lot more than simply issuing one instruction after another. In any but the most routine jobs, if we're sitting at a terminal running a computer session, what we do is determined by what happens. If something goes wrong, we might decide to stop the sequence of operations, or we might try an alternative approach, or we might first look at the state of the system or the output produced by the faulty component to find out what went wrong, and then take further action according to our findings.

If we are to come anywhere near to reproducing that sort of behaviour from a command file, therefore, our JCL must include not only instructions with which we can execute programmes, but also instructions (which we won't use directly) to inspect the states of the programmes and of various aspects of the system, and take decisions, and choose what to do next according to the results of the decisions. Here is a list of examples of some of the implications.

Programme control : To control the execution of the command file, we shall require the usual control structures we find in high-level programming languages – conditional execution, iteration, composition of instructions, etc.

Communication with programmes : We require access to the current state of a process, and to its recent output, and we must be able to set parameters for processes. It is particularly important to be able to determine why a process stopped.

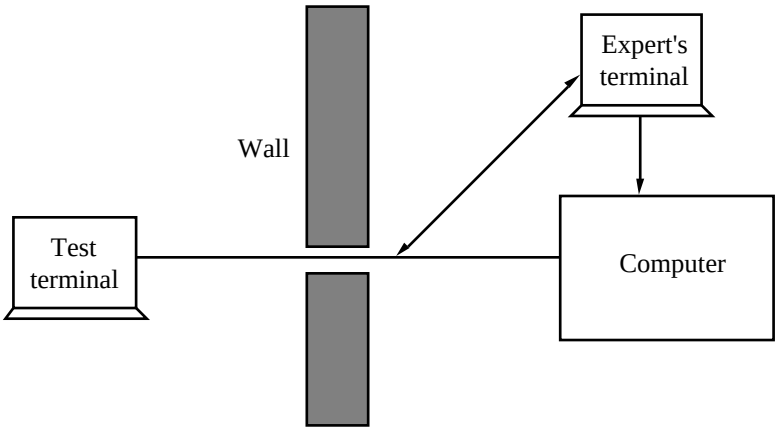
Communication with the system : We must be able to determine normally accessible facts about the state of the system – are certain files present, what time it is, etc.

The JCL designer should provide syntactic devices with which these operations can be carried out. To do so, the system and process attributes must be available, so these requirements have implications for the way in which system information is managed, and these implications must be addressed as the system design is further refined.

In practice, if you can precisely define what you want, it's usually quite straightforward to make it available – but if you miss out this sort of analysis, then you never find out what you want, and it can be hard to add it later when the system is already designed and constructed. We think the "usually" in that sentence is fair; it means in three of the four cases covered by our set of types of JCL. We've already mentioned the case that isn't covered, which is the preparation of command files using a WIMP language. Well, how would you do it ?

HOW WELL DO THEY WORK ?

Unix is the classic example of a traditional system; it's also, according to several usability studies, one of the least comprehensible of its type, mainly because the usual Unix shell makes widespread use of abbreviated instructions which don't obviously mean anything informative or helpful. Other textual systems of the same general type can be made much easier to use. In one experiment[REQ4], people were asked to perform a simple computing task using a terminal which (they were told) was attached to an intelligent operating system which would try to understand whatever sort of instruction they entered. In fact, the "intelligent system" was a conventional system helped by an expert, who would translate any comprehensible instruction into its equivalent operating system instruction, and engage in dialogue with the experimental subject through the terminal if an incomprehensible instruction were received. (For reasons which are not clear to us, this is called a "Wizard of Oz" experiment. "Wizard" we understand, but why "Oz" ? We know about the story, but it doesn't seem to have any connection.)



All transactions were logged, and the log later analysed to find out how people naturally formulated their instructions. Then the operating system interface was changed to cope with these more natural instructions, and the experiment repeated. In the first iteration, about 7% of the instructions which people tried were immediately acceptable to the operating system; at the end of the study, the changed operating system could accept about 76% of the "natural" instructions.

And that's not bad at all. So far as we know, though, while research on this and related topics has continued[REQ21], no manufacturer ever took up the challenge, and most of the traditional job-control languages remained obscure. One might wonder what would have happened if the textual style of input had remained fashionable through the development and popularisation of microcomputers. Certainly experiments of the sort we've described suggest that much better textual interfaces are possible, and if the amount of effort expended on graphical interfaces had been put into the improvement of textual interfaces, who knows what might have turned up ?

BATCH SYSTEMS.

The old-fashioned batch systems are still useful, even though interactive work is much more common. Plenty of routine jobs (summaries, reports, housekeeping) can be done very efficiently without intervention; some specialised jobs make great demands on the system resources (large number crunchers), and can't coexist with ordinary interactive work. In general, the more work you can do in the middle of the night, the more you can get out of your computer system; many systems encourage people (lower charges) to run jobs at night whenever possible. But it can be overdone[REQ5] :

```
IBM4341 BATCH QUEUES CLOGGED

Owing to the large number of jobs being sent to the IBM4341 overnight
batch
queue, turnaround time has lengthened to about four days. There is
little
that can be done about this.
                                                        Russell Fulton
```

The omission of such a batch or command file facility from the Macintosh system was a significant nuisance : as a simple example, a common sequence of actions which one of us used frequently after making a final copy of a file (such as this section of these notes) is to move it from a working directory into another directory, and also to copy it through a network connection to a directory on a different machine for off-line archiving. Sometimes one forgets the off-line step, so archives become inconsistent; to be able to make a command file to do it all systematically and easily would be very helpful. Now Applescript is available, circumstances have changed, so the same procedure is no longer necessary, but that's the sort of job which it should do well.

YET ANOTHER LANGUAGE - THE ONE WE DON'T MENTION.

What we have written so far in this chapter is what you might call the party line; there are two sorts of language which we use when communicating with a computer through a terminal, one graphical depending on the cooperation of the operating system, and selection from different possibilities diaplayed on the screen, and the other textual, essentially independent of the screen display and expressed in a "command language" which should be clear and precise and so on.

But there is another sort of language, widely used with graphical interface systems, which fits neither of those patterns, and which can be very powerful. It goes under the name of shortcuts, command key codes, and other such incomprehensible titles; it is a way of issuing instructions through the keyboard which bypasses the graphical interface, and is often a lot faster - once you know what to do.

This is not at all a respectable language. It has no syntactic structure, isn't "intuitively obvious", uses cryptic symbols, and is not easy to discover. These are all very bad things, and we wouldn't mention the language at all if it didn't make a very important point about terminal use : that is, that for all our agonising about designing languages of one sort or another, people - some people, anyway - are quite happy to get along with what amounts to a vocabulary of primaeval grunts. And they get along very well.

Why, then, do we worry so about our carefully designed respectable languages ? We'll give two answers to that question, both of them good. First, an orderly and systematic framework is valuable when you're learning to use a computer system. If you have a set of reliable rules to follow, a set of conventions which generally work, and a reasonably comprehensible vocabulary which you can use to refer to help files or manuals when you get stuck, then you can start to work independently much sooner than would be possible if all you had was a collection of grunts. Second, as with ordinary programming languages, we do need ways to write down what we want done, whether for command files or for records or for preparing instructions for other people to follow, and a language which is consistent and reasonably comprehensible is a big help. And we might also add that when you forget your arbitrary shortcut keys, it's good to have a reliable alternative on which you can fall back.

Turn the question round, then. Why do people implement the shortcuts ? Again, there are (at least) two good answers. First, if you do know them they can speed up your work considerably, if only by relieving you of the burden of spending time moving the mouse around. Second, some of them at least are indispensable for people with forms of disability which mean that they physically can't use a mouse. Keys are comparatively easy to use, either directly or by using some sort of alternative selection device, and the

key equivalents make the system accessible to many who otherwise wouldn't be able to use it at all.

Finally, we note that the idea is not new. The menu interfaces which preceded the modern graphical systems often allowed you to select an item by entering a letter or number at the keyboard, and if you knew the sequence of menus which you were going to meet (as you did if you often carried out the same task), you could remember the correct key sequence and simply type it in as a unit. Unfortunately, hardly any systems designers had noticed that, so however quick you were the system would go on laboriously displaying all the menus in the chain and reading one letter from your input string each time. So you didn't save any time at all.

COMPARE :

Lane and Mooney[INT3], Chapter 4.

REFERENCES.

REQ1 : J. Nielsen : "Traditional dialogue design applied to modern user interfaces", *Comm.ACM* **33#10**, 109 (October 1990).

REQ2 : *IBM 1130 Disk Monitor System, Version 2, Programmer's and Operator's Guide*, IBM Corporation, 10th Edition, 1972, pages 5-9.

REQ3 : *Burroughs B6700 Work Flow Management User's Guide*, Burroughs Corporation, 1973, pages 2-13 (redrawn for clarity).

REQ4 : M.D. Good, J.A. Whiteside, D.R. Wixon, S.J. Jones : "Building a user-derived interface", *Communications of the ACM* **27**, 1032 (1984)

REQ5 : R. Fulton : *Auckland University Computer Centre news*, 21 March 1986.

REQ21 : B.Z. Manaris, J.W. Pritchard, W.D. Dominick : "Developing a natural-language interface for the Unix operating system", *Sigchi Bulletin* **26#2**, 34-40 (April, 1994).
