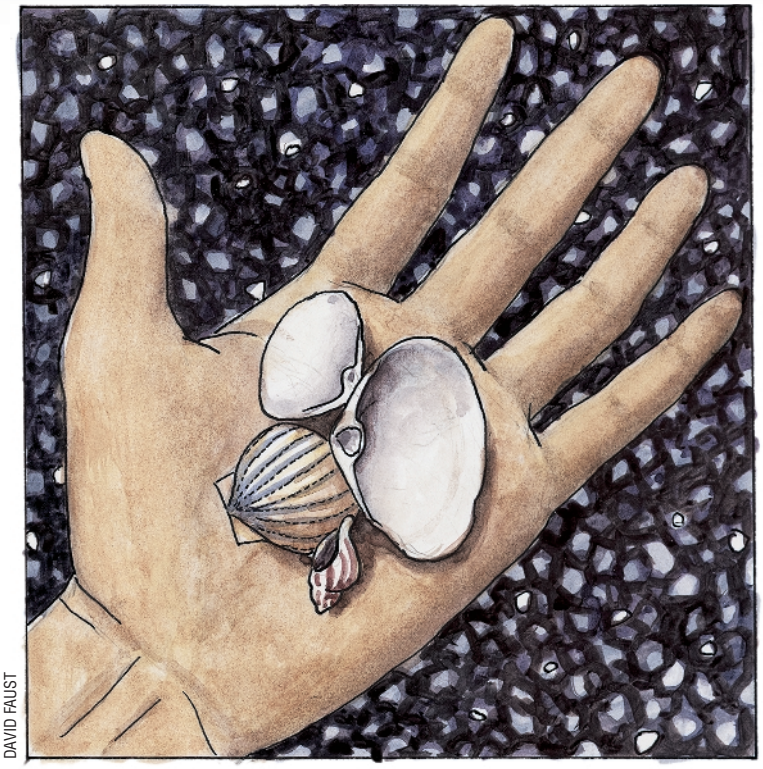
by Peter Collinson, Hillside Systems



DAVID FAUST

Programming ... and Shells

A strength of UNIX has always been the ability to create scripts that allow the user to tailor the system to their needs. It's fundamental to the system that the commands the user types into the shell from their keyboard can also be placed in a file and executed by the same shell, repeating the sequence of actions time after time. The shell script created by these means is a program and shells are really programming languages designed by programmers, and originally intended for programmers. When you take the first steps of creating a shell script, you are unwittingly entering a world that's full of implicit assumptions about how programs operate and how programmers do their work. I will examine some of those assumptions.

Programs

Most programs are not scripts, but are written using text source files, and then "compiled" into a low level set of instructions that run directly on the machine hardware. Your UNIX system is full of binary images whose source is in C or C++. The compilation step also allows the compiler that's processing the source to check the validity of the syntax the author used to express the program.

Languages for computers are designed to have a specific regular syntax that can be validated. Also, languages often contain elements used to double check on the programmer's intentions. The basic idea is to use the compiler to weed out typographical errors at the compilation stage. Compilers take in source and output a binary image intended to run on a particular machine architecture supported by a specific operating system. To alter the binary, you need to get back to the source, change it and create a new binary image. Actually, UNIX adds a new dimension to this story. UNIX programs are portable (or should be portable) at the source stage, so to get a new program running on your machine, you will need to compile it from the source. Prior to UNIX, programs often needed extensive work to move them from one machine to another. Of course, the story is not quite that simple, but this was the original aim of the standardizers of C and the POSIX work.

Scripting languages such as shells execute source text directly without compilation. They are programming language interpreters, reading commands and executing them. Shells interpret the source directly, and each command sequence is scanned for errors and then executed. I've used the word "interpreter" as jargon here. An interpreter takes instructions, perhaps in text or as binary values, and executes them one at a time.

Some languages employ both compilation and interpretation. They compile the source code into a set of opcodes that are executed by a runtime interpreter. These opcodes are generally simple operations and the interpreter can be small and efficient. The Java language works like this: it compiles the program into a set of instructions for a machine defined by software–a virtual machine. The opcodes are well defined and don't

change from architecture to architecture. The virtual machine interpreter is then implemented on different machines allowing the program to be portable to many different systems. The program is shipped in compiled form, so the result is a traditional "compiled" program where the executable form is stored in a binary image for the virtual machine.

Java programs are shipped in "binary" form, but run on a universal virtual computer that needs to be implemented for each target machine architecture and operating system. The neat trick is to write the compiler in Java, so once you have implemented the virtual machine, the compiler itself is portable. Although Java is a relatively recent language, many of these ideas of portability are not. C was based on a language called BCPL, and the compiler for this language worked in a similar manner. It compiled the program into a set of opcodes for a virtual machine, and then implemented a back-end to the compiler that translated from the virtual machine opcodes into something that would run on the target computer. The new idea in Java was to separate the two stages of the compiler and make the set of opcode instructions portable.

Scripting languages derive their power and usability by shipping the program in source form. However, they can still operate by compiling the source into a set of internal opcodes that are executed by an interpreter. A good example here is Perl. Perl derives a lot of its power by having both the compiler and interpreter in the same runtime environment. A Perl program can generate Perl source code and compile and run that code. The core of Perl is written in C, and its portability derives from the spread of C as the system programming language of choice.

However, shells and several other of the smaller UNIX tools are simple interpreters, which reads a command from the input file, decides what to do with that command and takes the appropriate action. Shell syntaxes of various flavors are actually complex programming languages that are directly interpreted.

Statements

We are back to where I started–placing a set of commands into a file and running that file through the shell. The notion that a sequence of commands can be run from a file maps simply onto one of the basic ideas of most programming languages: programs consist of a series of statements that are executed one after another. The programmer is responsible for the order of the statements. The system simply arranges that when it completes the execution of one statement, and automatically goes onto the next one and continues the work until it runs out of statements.

This notion extends right into the heart of the computer system. At the hardware level, the CPU spends its life pulling data from memory, interpreting the binary pattern as an instruction that deals with its hardware, and executing the instruction to change the state of memory or registers. It then automatically moves onto the next memory location to find the next instruction. Computer scientists call this action the *fetch-execute* cycle, which is fetching an instruction from memory and executing it, while maintaining the knowledge of the location of the next instruction in the *Program Counter*.

A key part of the mechanism is that there are a set of instructions that can affect the value of the program counter, so the program itself can change where the next instruction is to be found. We can load the program counter directly (a jump or goto instruction), making the program loop back to the start. However, the program might jump back to the start, run through the code until it finds the jump instruction, jump back to the start and continue forever. We don't want to create loops that don't terminate and want to "jump out" of the loop depending on the result of a test on data.

CPUs implement "conditional instructions," perhaps testing the value of a register and loading the program counter if the value is zero and allowing the CPU to move onto the next instruction if the value is non-zero. There will be a variety of tests that are implemented in the hardware to supply a wide range of testing options to the programmer.

Once we allow for the existence of jumps and tests, our linear set of machine instructions have instantly become more complicated. We can now jump around inside the code and execute different parts of the program depending on the results of tests on the data. Programs now consist of little chunks of code, each executed linearly. Each section may or may not be executed depending on the value of data and the results of tests.

Of course, these days we rarely write programs using the binary code that is fed directly into the machine. Fairly early on in the history of computing, programmers knew this was hard, and realized that they could write programs to help with the problem. "Assemblers" were created that read a textual representation of the low-level machine codes and translated it directly into the binary values. Initially, there was a close mapping between each line of text in the assembler source and the actual machine code that was generated.

Later, full computer languages emerged and these were called "high-level" languages. In a high-level language, a single statement by the programmer is mapped into several low-level instructions that run on the target machine. For example, high-level languages contain linguistic constructs that let programmers place loops in their code, which are compiled into tests and jumps in the underlying architecture.

What's interesting about computer languages is that the syntactic features of a language probably start life as a shorthand way of expressing a solution to a common programming problem. But once they are implemented in a language, they take on a life of their own. The language itself constrains or promotes the solution of a programming problem in a particular way. For example, it's not easy to handle text strings in C (or for that matter in many compiled languages); so Perl, which is good at dealing with text strings, became the preferred way of writing CGI scripts at the start of the Web. CGI scripts spend much of their life processing strings.

Routines and Functions

As time has progressed, it's become apparent that programmers fare badly when programs are large. Mistakes are made and errors introduced in almost direct proportion to the size of the program. The evolution of programming languages in the last 30 years has seen the emergence of methodologies that attempt to break programming problems into smaller digestible chunks.

At any one time, the programmer is dealing with a small problem that's comprehensible and hopefully solvable.

A key idea in the evolution of more easily used programming languages is the notion of routines. Routines, sometimes called "subroutines," are a way of specifying a section of the program that can be used and reused in several places. The code for a routine sits outside the linear code model I was discussing above and will not be used unless we jump into it. We jump into the routine using a special instruction that remembers the location in memory from where the initial jump was made. We normally term this "calling a routine." So when we get to the end of the routine we can say "return to the point in the program that's just after the point at which you jumped to the start of the routine."

We can pass values into the routine using "parameters" and routines can return a value. When they do return a value, they are often known as "functions." Functions started life in mathematical programming where it was better to get a numerical expert to write the function that computed square roots, the sines, the cosines, and the like. Once that was done, you could reuse someone else's work and hopefully always have mathematically valid answers.

The ability for programmers to define their own routines started life as a programming convenience. Programmers soon found themselves writing the same code over and over. Code is really a recipe for getting things done, and if you spot that some sections of your program have repeated patterns of operations, then you want to pull that code out of the main linear stream of the program and reuse it. Reuse cuts down on program size and typing. The code of the routine will perform a set of operations on some data and produce a result. By putting the operations into a routine or function we can easily use the same code on different data.

However, once a routine is created to do something, it too "takes on a life of it's own." It becomes a building block in the program you are creating. You can forget about how it works and treat it like a primitive operation in the language. The routine will have a clear set of parameters and will possibly return a result whose value is understood. A routine will simplify the programming exercise. You no longer have to worry about how that part of the task is done, so you can concentrate on how you want to use the routine to solve the greater problem.

When you treat routines as separate building blocks in a program, you have to adopt a set of rules about what each routine can and cannot do. Programmers have learned that routines should really be self-contained, data should be passed in using parameters and results passed out. A routine should not "side-effect," do something that changes the data elsewhere in the program, unless that interaction is well defined. If the routine needs temporary data storage, then that should be kept private. If a routine needs its own routines, then those should be private too.

These ideas have lead to the development of "object oriented programming" where programmers are forced by the programming language to develop self-contained modules that have well-defined ways of interacting with each other. As we have seen again and again, "objects" have taken on a life of their own, and are now more than just a way of controlling access to code and data; they have become a way of thinking about problem solving.

If we relate the notion of routines back to shell scripts for a moment, you can think of a command that is entered into the script as a routine that does something specific. When we add a line calling the `cp` command, we don't care how the command works, we just want it to copy a file. We supply it with parameters, the source, and destination file, and it does the job. Actually, the command returns a success or failure result to the script so we can find out whether it worked or not. The commands are "well behaved" in the sense that you can only call them and get a result. They cannot affect the script that called them or the data in that script.

Variables

I've talked a great deal about the code of a program, but not about the data on which the code operates. If we return to the roots of programming–a program running in a computer–then the memory of the computer holds the program which is being executed by the CPU, and the program will refer to locations in memory that hold data. We can write a program that adds 2 to 2 to give 4, but it's more useful to create a general-purpose program that adds any two numbers. We need ways of referring to locations in memory. These locations hold varying data and are normally called "variables" to distinguish them from constants (like 2) that are placed directly in the program itself.

In the early days, when computers were programmed in binary, the code had to talk about, say, location 2001 in memory. It was soon realized that it was better to be able to give names to memory locations so when you looked at the program you could deduce something about the value that the location was holding. If you were using the location as a counter you could call it "counter." Doing this was more comprehensible and generated less programming errors. When assemblers took the strain from programming in binary, a major function was to relate names to memory locations.

When high-level languages came along, programmers were no longer worried about where a specific value was stored in memory; they just defined and used variables. The compiler took over the job of allocating memory locations for the program. Routines created a need for variables that were local to the code of the routine. When you named a local variable `i` in a routine, you wanted it to be a different memory location from the variable `i` in another routine. Languages define ways of accessing variables that were local to the routine or global to the whole program.

Variables are handled in different ways by different languages, and languages also have to cope with the type of the variable. CPUs can deal with whole numbers (integers) or fractional values (usually stored as "floating point"). There are generally different instructions used to operate on values that are stored in these different types. Computers in general don't have any intrinsic knowledge of what a particular location holds, and will depend on the program to specify what is to happen. How the program does that depends greatly on the language.

In some languages, you are forced to declare the variables that

the program will use. The declaration is used to check your typing, and the compiler will complain when a variable that has not been declared is used. The declaration is also used to assign a type to the variable. The compiler knows the variable should contain an integer and will generate the correct instructions when you add a constant to it. If you load an integer from a floating point variable, the compiler knows that it needs to compile in appropriate type conversion code when needed.

In general, scripting languages have often dispensed with the distinction between types and will attempt to do the "right thing" depending on context. Most scripting languages use text strings as their basic storage type. Some will evaluate that string into a number depending on context. Also, most scripting languages don't demand declaration of variables; you just start to use them. Of course, this means that there is no check that you typed their names correctly.

Shells

Variables in shells are preceded by a dollar character to tell the shell that "this name is a variable" and not a command name or string. Shells are really string processing languages and variables contain strings. When the interpreter finds a command line containing a dollar symbol, it will determine the name of the variable and replace the dollar and the name in the line by the contents of the variable. It then looks again at the line because a variable can contain the name of another variable. There are several "special variables." For example, in Bourne and Korn shells, `$#` is set to the number of parameters that have been supplied to the script. In all shells, the positional variables, such as `$1`, `$2`, and so on, are set to the values of the parameters.

Over time, shells have acquired similar programming constructs to the regular programming languages (`for` statements or `while` loops for example). In Bourne and Korn shells, you can also define functions and routines. Attention has been paid to make these behave like regular UNIX commands, so that they integrate seamlessly into the language. Shells can, and are used, to create complex programs. For example, Sun's patch maintenance system for Solaris is a set of shell scripts.

If you are making a start on shell programming by copying commands into a file and executing them, then the next step is wonder whether you can make your command into something more general purpose by replacing the file names with variables, and pass values into those variables from command line parameters. A good tip is to always check that you have the right number of parameters. It can save grief when you mistype the command. ⊸

Peter Collinson *runs his own UNIX consultancy, dedicated to earning enough money to allow him to pursue his own interests: doing whatever, whenever, wherever ... He writes, teaches, consults and programs using Solaris running on an UltraSPARC/10. Email:* `pc@cpg.com`.