Chapter 5 of ***Using Computers in Linguistics: A Practical Guide***
Routledge, 1998. Edited by John Lawler and Helen Aristar Dry
(Terms in bold italics are to be found in the Glossary, references in the Bibliography.
Both are available online, along with other resources; see Page 24 below for URLs)

The Unix™ Language Family

John M. Lawler

Linguistics Department
and Residential College
University of Michigan

1. General

The **Unix**™[1] operating system is used on a wide variety of computers (including but not limited to most workstation-class machines made by Sun, Hewlett-Packard, MIPS, NeXT, DEC, IBM[2], and many others), in one or another version. If one is around computers almost anywhere, one is within reach of a computer running Unix, especially these days, when Linux, a free version of Unix, may be found on many otherwise ordinary-looking PCs. Indeed, more often than not Unix is the **only** choice available for many computing tasks like E-mail, number-crunching, or running file servers and Web sites. One of the reasons for the ubiquity of Unix is that it is the most influential operating system in history; it has strongly affected, and contributed features and development philosophy to almost all other operating systems.

Understanding any kind of computing without knowing anything about Unix is not unlike trying to understand how English works without knowing anything about the Indo-European family: that is, it's not impossible, but it's far more difficult than it ought to be, because there appears to be too much unexplainable arbitrariness.

In this chapter I provide a linguistic sketch[3] of the Unix operating system and its family of "languages". I use the word *language* here in its usual sense in computing contexts; since computer languages are not at all the same kind of thing as natural human languages, clearly this is a metaphorical usage. However, modern linguistic theory, strongly influenced as it is by computer science, is capable of describing the Unix language family rather well, because these

---

[1] ***Unix*** is a registered trademark (at press time, it was a trademark of Santa Cruz Operation, Inc.) whose provenance and ownership, along with the traditions and variations of its use, is part of its history as a language, in much the same way that, say, *Indo-Germanic* is a term with roots in the history of linguistics, and of the study of the Indo-European language family. This point having been made, we do not hereinafter use the "™" symbol with the word *Unix*. For the etymology of *Unix*, see Salus (1994).

[2] All of these proper names are also registered trademarks; hereinafter we do not distinguish trademarks from ordinary proper nouns.

[3] My models in this sketch, to the extent practicable, are the excellent language sketches in Comrie, ed. (1987).

"languages" possess some of the ideal characteristics posited by linguistic theories: they are completely regular, they exist in a homogeneous community, they are unambiguous, they are context-free, they are modular in design and structure, they are acquired (by computers, if not by humans) instantaneously and identically, they are universally interpretable in an identical fashion (barring performance details), and there is in principle no difference between one user and any other. Consequently the metaphor has considerable utility here for anyone familiar with linguistics. This situation is not in fact coincidental, since Unix was designed in the first place by people familiar with modern syntactic theory and its computer science analogs, and it shows. As a result, linguists will find much here they can recognize, though perhaps in unfamiliar surroundings. To that extent this chapter is simply applied linguistics. But Unix is also useful for applying linguistics, as I attempt to demonstrate.

2. History and Ethnography of Computing

Unix is an ***operating system (OS).*** This is a special type of computer program that is, in a very important sense, a syntactic theory that completely constrains (i.e, defines, enables, and limits) all the programs that can run on a particular computer. In effect, the computer *per se* runs only the OS, and the OS runs everything else. Up until the 1970s, and for some time thereafter, it was normal in the computer industry for an operating system to be ***proprietary***; that is, it was typically developed and sold by the makers of a particular computer along with the computer, and was limited to running on that computer alone. Any program that was to be run on that computer would have to be compatible with its OS, which varied markedly from computer to computer, limiting the possibility of widespread use of any program. Apple's Macintosh-OS has been a proprietary operating system for most of its existence, for instance, and the same is true of DEC's VMS; thus a program that runs on a Macintosh will not run on any other machine[4]. MS-DOS, on the other hand (which has been influenced significantly by Unix), is an example of a non-proprietary (or ***open***) OS. Unix was the first successful open operating system.

Unix began in 1969 at Bell Laboratories in New Jersey. Ken Thompson, a member of the technical staff, put together a small operating system, and over the next several years, modified and developed it in collaboration with his colleagues, notably Dennis Ritchie and Brian Kernighan. This group[5] was also instrumental in developing at the same time two programming phenomena that have become totally integrated into Unix, and vice versa: the ***Software Tools*** movement, often called a 'philosophy'[6], and the ***C*** programming language.[7] They produced a number of enormously influential books[8] still to be found almost three decades later on the desk of most serious programmers and system designers; this is a signal accomplishment in a publishing era where one year's computer books are inevitably the next year's landfill.

The Software Tools 'philosophy' gives an idea of why Unix is the way it is. The metaphoric image is that of a matched set of hand- or machine-tools that are capable of being snapped together *ad lib* into any number of super-tools for specialized work on individual problems. If you had to make table legs, for instance, you might, with this set of tools, in this virtual reality,

[4] This has recently changed with the licensing of the Mac OS to other manufacturers.

[5] This team did not rest on its Unix laurels. They have been working on a successor, and it is now being released under the whimsical name ***Plan 9***.

[6] Technically, this is a pervasive metaphor theme, with accompanying social movement, rather than a philosophy per se. Software Tools and Unix predate modern metaphor research and terminology by about a decade.

[7] So-called because it was the successor of the ***B*** programming language.

[8] Kernighan and Plauger: *The Elements of Programming Style*, *Software Tools* (both 1976), and *Software Tools in Pascal* (1981); Kernighan and Ritchie: *The **C** Programming Language* (1978); Kernighan and Pike: *The Unix Programming Environment* (1984).

hook up your saw to your plane, and then to your lathe, and finally to your sander, *just so*, feed in sticks of wood at one end of this *ad-hoc* assemblage, and receive the finished table legs at the other end. Real tools don't work that way, alas, but software tools can, if they're designed right. The basic principle is to make available a number of small, well-crafted, bug-free programs (***tools***) that:

- do only one well-defined task
- do it intelligently and well
- do it in a standard and well-documented way
- do it flexibly, with appropriate user-chosen options available
- take input from or send output to other program tools from the same toolbox
- do something safe, and if possible useful, when unanticipated events occur.

Linguists are familiar with at least the spirit of this concept as the principle of modularity in syntactic theory. Modular design is a watchword in computer science as well as syntax, however, since it allows easy construction of *ad-hoc* assemblages of tools for individual tasks, just as English syntax allows easy construction of *ad-hoc* assemblages of ideas for individual purposes, i.e, the proverbial infinite number of sentences.

For example, consider the task of preparing a lexical *speculum*. This is simply a wordlist in ***reverse alphabetic order***, so that *bring* and *string* might be adjacent, for instance; in a suffixing language like English, such lists have obvious utility for linguists. (See the previous chapter for more discussion of wordlists.) They are immensely difficult to prepare by hand, however, and they can be tedious to program even on a computer. Below I present the Unix solution to this problem in the form of a linguistic data analysis problem; the answer follows. First, some preliminary information: `word.big` is an old wordlist from the University of Michigan's *MTS* mainframe system, salvaged from its demise. It was used by faculty and students for 20 years to spellcheck their email and papers, so it's full of unusual words, in all their paradigmatic forms. To be precise as to the quantity, if not the quality, `wc` reports that it contains 70,189 words on 70,189 lines, for a total of 681,980 bytes (including 70,188 newline characters):

```
% ls -l word.big ↵
-rw-r--r--  1 jlawler    681980 Mar 17  1995 word.big

% wc word.big ↵
   70189   70189  681980 word.big
```

And now the problem. The numbering is added for reference; the rest is verbatim. If you type A-D at the '%' Unix prompt, you get back 1-10. Describe the syntax and semantics of A-D. Are there any generalizations?

```
A. % head word.big ↵ ..... B. % head word.big | rev ↵
      1. a ..... a
      2. A ..... A
      3. aardvark ..... kravdraa
      4. aardwolf ..... flowdraa
      5. aba ..... aba
      6. abaca ..... acaba
      7. abaci ..... icaba
      8. aback ..... kcaba
      9. abacterial ..... lairetcaba
     10. abacus ..... sucaba
```

Figure 1a. Data Analysis Problem (A-B)

C.`% head word.big | rev | sort ↵` D.`% head word.big | rev | sort | rev ↵`

```
 1. A ..... A
 2. a ..... a
 3. aba ..... aba
 4. acaba ..... abaca
 5. flowdraa ..... aardwolf
 6. icaba ..... abaci
 7. kcaba ..... aback
 8. kravdraa ..... aardvark
 9. lairetcaba ..... abacterial
1. sucaba ..... abacus
```

Figure 1b. Data Analysis Problem (C-D)

Using software tools (specifically the Unix programs `sort` and `rev`, and the Unix conventions of ***input-output (I/O) redirection***), and given a wordlist file (with one word to a ***line***, easy enough to prepare from any text via other Unix tools[9]) named `word.big`, the following ***command*** will produce a file named `speculum`:

```
% rev word.big | sort | rev > speculum
```

A parse of this command line shows it to be very straightforward:

% the Unix C-shell (`csh`) prompt; this is the context for the command that follows
`rev` ..... send to output a reversed copy of each line in source file:
`word.big` ..... name of source (unmarked ablative) file, which is to be read only
`|` ..... a ***pipe*** marker, connecting the output of `rev` to the input of:
`sort` ..... sort input alphabetically by line
`|` ..... another pipe, linking the output of `sort` to the input of:
`rev` ..... (another copy of) the same program invoked in the first clause
`>` ..... dative case marker, indicating where the output stream should be stored
`speculum` ..... name of goal (dative) file containing final output; to be written only.

Figure 2. Parse of command line:
`rev word.big | sort | rev > speculum`

The command is executed by sending the line to `csh`, which interprets and executes it. This in turn is accomplished by pressing RETURN at the end of the line, which may be considered a performance detail.

The programs `sort` and `rev` are both ***filters***[10]; i.e, they belong to a class of programs that read a file and do things sequentially to what they find, sending their output to a ***standard output***.[11] This in turn can become the ***standard input*** to the next program in the pipeline. This

---

[9] See sec. 6 of this chapter on *filters* for an example.

[10] See sec. 3 of this chapter for further discussion of the *stream* metaphor. Note that *pipe* and *filter* are usefully coherent with it.

[11] The ***Standard Input*** (and ***Output***) are abstract *streams* that are associated with every Unix program by the OS. They are the ablative source (and dative goal) of any filter program. The unmarked (default) Standard Input is the keyboard (and the unmarked Standard Output is the screen) unless they are redirected; i.e, unless overtly referenced with a ***pipe*** "**|**", as in *`egrep 'umich.edu' $web.log | wc -l`*
or a case marker (**<**, **>**), as in *`mail jim@somewhere.edu < job.talk`*

is not unlike the kind of processing that linguistic theories posit for various components of a derivation, and is directly related to the modularity inherent in the Software Tools design philosophy. `rev` simply reverses the ***characters*** on each line it encounters, while `sort` sorts files alphabetically by line.

The first part of the command above tells the OS to use `rev` on the file `word.big`, producing a stream of individually reversed lines. In Figure 1 above, the stream was limited to ten lines by using the head program, which simply shows the first few lines of a text file, defaulting to ten; in this command, however, the full word.big file would be the stream.

This stream is piped as input to `sort`, and `sort`'s output is repiped to another copy of `rev`, this time re-reversing the (now sorted) strings. Finally, the resultant stream is parked in a file called `speculum`; the original file `word.big` is not affected by this operation, since it is only read, not written. In a test on a Sun workstation, with a `word.big` of 79,189 words, production of a `speculum` file by this method took less than one second.

The success of this combination of Unix, Software Tools, and C is evident from the facts:

- that C is the programming language in which Unix, the most widely-used operating system in the world, is written;
- that all the Software Tools programs are available, in C, on Unix, which is designed for their use and fits them best (though they are also available elsewhere, and in other languages);
- that C is the most widely used professional programming language in the world; any popular microcomputer program, for example, was almost certainly written in C.

Many of the software tools on Unix had their origin in the Tools movement, all were written in C, and all shared a common interface language, differing only occasionally in details of semantics and grammar. In addition, many of these tool programs (e.g, `awk`, `sed`, `perl`; see section 6 below) evolved sublanguages of their own with a common core of structure, and these in turn came to influence Unix. A well-thought-out set of tools, and ways of combining them into useful programs, has many similarities to a well-thought-out set of phrases, and ways of combining them into useful speech. And, while their complexity does not approach that of a real natural language, the structure can be apprehended in similar ways, and this fact was not lost on the developers: Unix has been oriented from the start toward the written word, with numbers only incidental. Indeed, its first user was Bell Labs' word-processing department.

Gradually, the fame of Unix spread outside the lab. AT&T, Bell Labs' parent company, was at that time enjoined as a regulated monopoly from engaging in the software business, and thus the unlooked-for advent of a popular software product with its attendant demand was something of an embarrassment to the company. Their solution to this problem was almost as remarkable as its origin: AT&T essentially gave away Unix. For educational institutions, AT&T granted inexpensive licenses to run Unix on appropriate machines (originally PDP, later Digital's VAX line, eventually machines designed especially for Unix), with full source code (in C) included. This meant that not only were the universities (in the personae of the students and staff of computing centers and computer science departments, and interested others) able to run Unix, but they were also able to modify it by changing its source code. Development proceeded rapidly at a number of sites, most importantly at the University of California at Berkeley, resulting eventually in the various releases of the **Berkeley Standard Distribution** of Unix (***BSD***), which was also free, and rapidly became the standard operating system for many computers.

This was particularly significant for American academic computing, since the late 1970s and early 80's was the period in which most universities switched over from large mainframe centralized computing services to distributed departmental minicomputers, frequently running

Unix. Many of the design decisions in BSD Unix and its successors were thus made by academics, not businessmen, and this strongly influenced subsequent developments. Perhaps more importantly, the on-line culture that grew up around Unix, and proliferated into Usenet and then the World Wide Web, was an academic culture, not a business culture, with significant differences that were to become far more evident and important.

By the time 4.2BSD was released in 1983, AT&T had been become free under the law to do something commercial with the rights it still held to the Unix operating system. However, the commercial, business-oriented **System V** version of Unix (***SysV***) released by AT&T that year to take advantage of this opportunity had serious incompatibilities with the BSD Unix that had grown up in academe in the previous decade, and an anxious diglossia ensued. Decreolization of these and other Unix versions in the form of eventual standardization of the competing versions is now being pursued and in many cases has been effectively achieved; but to this day, every version of Unix has its idioms, its gaps, its own minor examples of *Traduttore, traditore*. In this survey I do not treat dialectal variations, but rather concentrate on the many mutually-intelligible characteristics found in every version of Unix.

There are many fuller accounts available of the diachronic and dialectal development of Unix. The best and most thorough is Salus (1994), which has the additional virtue (for the purposes of this chapter) of having been written by a historical linguist who was personally involved with the development of Unix. For synchronic analyses, the best source is Raymond (1995), the printed version of an ongoing electronic lexicography project of impressive linguistic sophistication.

3. Bits and ASCII: Phonetics and Phonology

Unix, like all computing systems, makes use of the concept of the ***bit***, or ***binary*** digit[12]. This is what linguists know as the concept of **binary opposition**, e.g, *voiced/voiceless*. Computing exploits binary oppositions in electronic data to form its infrastructure, just as language exploits binary oppositions in perceived phonation. Unix also exploits several important elaborations of the bit: the ***byte***, the ***line***, and the ***byte stream***. These etic units, which are literally built into the hardware, are structured by an emic system of byte interpretation called ***ASCII***.[13]

ASCII Chart about here

Figure 3. ASCII Chart

In computing, just as in distinctive-feature theories, all oppositions are binary: **plus** and **minus** are the only choices for any feature. In computing, these are represented by **1** and **0**. Since these are also digits, and the *only* digits needed in the representation of ***Binary*** (Base-2) integers, the possibility arises of combining these feature specifications in a fixed order to form sequences of digits, or numbers. The fixed order is fixed by the manufacturer and may vary considerably, but virtually all Unix machines assemble bits into convenient groups of eight, which are called ***bytes***. These are convenient because they are sufficient to define a useful-sized set.

All linguists learn that in Turkish, eight phonologically distinct vowels are possible, because there are three significant binary features, and $2^3 = 8$; that is, there are eight different ways to combine all possible values of the three features. With bytes, the relevant equation is $2^8 =$

[12] Besides being a genuine acronym, *bit* is also a remarkably apposite English name for the smallest possible unit of information.

[13] `/'æski/` in American English; an acronym of **American Standard Code for Information Interchange**.

256; that is, there are 256 different ways to combine the eight binary digits in a byte. 256 is an order of magnitude larger than the size of the English alphabet, and indeed the English alphabet is quite useful, even at that size. In fact, of course, the English alphabet (upper- and lower-case, separately coded), punctuation marks, ***diacritics***, and a number of other symbols are all commonly coded in bytes, and that is by far the most common use of the byte, so much so that it is useful mnemonically to think of one byte as one English letter.[14]

Here is a ***byte*****:** **`0 1 1 0 1 0 1 0`** This is the Binary number that corresponds to the Decimal[15] number **106**. It represents, in a textual context, the (lowercase) letter "**`j`**", which is number 106 in ASCII. In a different context, this byte might represent the Decimal integer **106** itself, or memory address 106, or instruction 106, or part of a more complex number, address, or instruction. Computers **use** binary notation; writing numbers graphically is for humans, and computers will write numbers any way they are told. This byte is therefore likely to exist, as such, not as one of a series of marks on paper, but rather as a series of magnetic charges in ferrite or silicon, or as a series of microdots on a compact disk (***CD***, or ***CD-ROM***).

All Unix systems are built on ASCII, and all Unix ***files*** (or streams) are byte files which can be interpreted in ASCII, whether they are intended to be or not. The history of ASCII will not be treated here, but it would not be unfair to stress the fact that the *"**A**"* in the acronym ***ASCII*** stands for *American*, with all that that entails from a linguistic viewpoint. That is, ASCII represents just about everything that an early twentieth-century American engineer might have thought would be useful in a character code. This includes the upper- and lower-case English alphabet (coded cleverly to facilitate alphabetization), the Arabic numerals, ordinary punctuation marks, a potpourri of non-printing ***control characters*** (like Line Feed), and virtually no "foreign" letters or symbols.

There *is* provision for representing some diacritics, as separate letters: thus Spanish, French, German, Italian, and other languages which use diacritics that have rough ASCII equivalents (circumflex [caret ^], umlaut [quote mark "], acute [apostrophe '], grave [backquote `], tilde [**~**]) can be represented, though with some difficulty, and not always completely (there is no ASCII character except comma that can function as a cedilla, for instance). Languages like Turkish, Hungarian, Polish, or Czech, which use letters or diacritics that have no ASCII equivalents, are very difficult to represent properly. Languages with completely different alphabets, like Russian, Arabic, or Hebrew, require heroic measures. And non-alphabetic writing systems like Chinese are out of the question; they require a completely different approach. Which is not to say that ASCII Romanization is impossible, of course.

Within its limitations, however, ASCII is very well-designed; a number of structural characteristics are worth pointing out. There are, to begin with, two parts of ASCII: ***Low ASCII***, represented in the chart above, from Decimal, Hex, and Binary 0 through 127 (= 2^7-1: Hex 7F, Binary **01111111**); and ***High ASCII***, from Decimal 128 (= 2^7: Hex 80, Binary **10000000**) through 255 (2^8-1: Hex FF, Binary **11111111**). Only the Low ASCII characters are completely standard; High ASCII characters vary from machine to machine.

---

[14] This is certainly as true (and mnemonically as useful) as the rough equivalences of one meter with one English yard or of one liter with one English quart.

[15] This byte is also expressible as number **6A** in ***Hexadecimal*** (base 16) notation. **A** is a digit in Hexadecimal notation, representing the number after **9**, which is called *ten* in Decimal notation. The capital letters **A**–**F** are single Hexadecimal digits representing Decimal **10** through **15**, respectively; Decimal **16** is written **10** in Hexadecimal. It is customary to add **H** after writing a Hexadecimal number (e.g, **6AH**) to indicate the base; but there are other conventions as well, such as **$6A**, with sources in a number of languages.

For instance, many of the same additional characters are included in both DOS/Windows and Macintosh text files, but not all; and they appear in different orders, with different numbers. This is one reason why DOS and Mac text files are different. Unix cuts this Gordian knot by not using High ASCII at all to represent characters; only Low ASCII, the first 128 characters, are meaningful in Unix, and we will restrict our attention henceforth to these.

The most recognizable characters in ASCII are the ***alphanumerics***, that is, the letters of the (English) Latin alphabet plus the (English) Arabic numerals. Since the upper-case letters and the lower-case letters are etically different, they are coded separately; since they are emically related, they are coded analogously. The upper-case letters begin with *A* at Hex 41 (Decimal 65, Binary 01000001) and proceed alphabetically through *Z* at Hex 5A (Binary 01011010), while the lower-case letters go from *a* at Hex 61 (Decimal 97, Binary 01100001) through *z* at Hex 7A (Binary 01111010). It can easily be seen that the difference between any given upper- and lower-case letter is always exactly 32; in Binary terms, it's even simpler: an upper-case letter has **0** in the third-highest bit, while a lower-case letter has **1** there. Otherwise, they are identical; this fact makes it simple to design software to be case-sensitive, or case-insensitive, as desired.

One of the important facts about Unix, which often disconcerts novices, is that it is case-sensitive by default, since it uses Low ASCII fully. Individual programs may (and often do) ignore case, but unless told otherwise, Unix does not. This means that a directory named `News` is not the same as one named `news`, and will not be found or referenced by that name. And since sort order is determined by ASCII order, and uppercase letters precede lowercase, this also means that `Zygote` will appear in a sorted list before `aardvark`, unless the sorting software is told to ignore case. One convention that results from this fact is that the unmarked case for Unix commands, filenames, directories, and other special words is lower-case. Capitalized and ALL-CAP terms are normally reserved, by convention, for special situations and system software, though there is no absolute prohibition imposed. For instance, most Usenet newsreaders (like `rn` or `trn`) expect to use (and will create if they don't find one) a directory named `News`. (A further behavioral modification produced by this convention is the decided predilection of some Unix users to eschew upper case in ordinary written communication, even when not modulated by Unix.)

Another feature of ASCII worthy of note are the Control Characters, which are non-printing, and represent an **action** of some sort; these may be considered supra-segmental analogs. Control characters have their own official names, taken from their original purpose (usually on teletype machines), which are normally acronymic or mnemonic in English. For instance, character no. 7, BEL (`^G` or *Bell*), originally rang the bell on a teletype, and now it often produces a noise of some sort on a computer, while no. 8, BS (`^H` or *Back Space*), originally moved the print head on a teletype back one space; now it is the code produced by the "BackSpace" ***key*** on most[16] keyboards.

The control characters occupy the first two columns[17] of ASCII; thus their **most** significant

[16] But not all. This is the source of much frustration (see next note), and explains why communication programs like `telnet` include provisions to use either BS or DEL as the destructive backspace character.

[17] With one exception. No. 127, DEL, is at the very end of the chart. This is binary "1111111" and represents an original convention in (7-hole) paper tape, an early input medium. If one made a mistake in punching the tape, one simply punched everything out and it was skipped. This later became a standard in the early Unix community for keyboarding; thus the Back Space key on many workstation keyboards produces no. 127, DEL (`^?` or *Delete*). This has not been completely integrated with the other early convention of using no. 8, BS (`^H` or *Back Space*), that persists in most microcomputer applications.

bits are "0000" or "0001". Their **least** significant bits are the source of their more common names, however. Just as the last four bits of "J" and "j" are identical ("1010"), so are the last four bits of no. 10, LF (`^J` or *Line Feed*), which originally moved the teletype print head down one line, and is now found as the ***newline*** character in Unix, among other uses[18]. Since, like all Control characters, this can be produced on a keyboard by pressing the "Ctrl" or "Control" shift key simultaneously with another key – in this case the "`J`" key – LF is often called simply "Control-J", or "Ctrl-J", and frequently abbreviated, like all control characters, with a caret as "`^J`".

All computer media, like writing or speech, imply a serial order. In print, we are used to the convention of lines of serially-ordered characters arranged horizontally on the page. For readers, a line is a more or less natural phenomenon, governed by paper and type size. In a computer, however, there is no physical page, nor any type to have a physical size. So lines, if they are to be defined at all, must be defined, like everything else, by bytes. Text files are line files; they consist of strings of bytes with newline characters inserted wherever a text line should be displayed. An ASCII text file with 1000 lines of 60 characters each would thus have 61,000 bytes:[19] 60,000 letter bytes plus 1000 newline characters. Many of the tools in Unix, like `rev`, work at the line level; others, like `sort`, work on whole files (though with reference to lines).[20]

Files are often called ***streams*** in Unix. Since a text file (and Unix is almost entirely composed of text files) is simply a string of bytes (some of which are probably newline characters), it is often convenient to view the file itself as a single string, and this is always the case whenever anything like reading, writing, or modification has to be done to a file. In a computer, since there is no question of moving the perceptor, conceptually it must be the bytes that are streaming past.

This metaphor is quite different from the static concept implied by ***file***: a *stream* is in motion, can be used for power, provides a continuous supply of vital resources, is all of the same kind, and is one-dimensional. A *file*, on the other hand, just sits there and waits for you to do something with it, offers little help, is entirely passive, may consist of many parts of different kinds, and is at least two-dimensional. This distinction between the metaphors of **stream** and **file** is not unlike Whorf's presentation (1956:210) of the distinction between the two Hopi words for 'water'. It turns out that the *stream* concept lends itself to convenient programming.

The result is that many Unix resources are designed around the concept of manipulating, measuring, analyzing, abstracting, modifying, sampling, linking, comparing, and otherwise fooling around with *streams* of text data, and since they share a common structure of conventions,

---

[18] Again, early conventions have resulted in variation. In DOS and Windows ASCII files, each text line is terminated by a cluster consisting of no. 13, CR (`^M`, or *Carriage Return*, which originally returned the teletype print head to the left margin **without** advancing a line), immediately followed by no. 10, LF. In Mac ASCII files, the standard ***newline*** character that terminates lines is CR **alone**, and in Unix it is LF **alone**.

[19] Or roughly 60 Kilobytes (KB). The *kilo-* prefix, normally denoting 1000, refers in computing contexts to 1024, which is 2^{10}. Similarly, *mega-* is 1,048,576 (2^{20}), rather than 1,000,000. While this is not standard metric, it rarely causes confusion.

[20] One important qualification must be made here. Text files in word-processing programs (on Unix or elsewhere) are **not *Standard ASCII*** files, and rarely mark individual lines with anything; on the contrary, most use ***newline*** to end paragraphs only, preferring to reformat lines on the fly. In fact, each wordprocessor has its own proprietary ***file format***, in which control characters and High ASCII characters are used to code information peculiar to the particular program that writes (and expects to read) the file. In general, one may assume that any two wordprocessors' files are incompatible unless special steps, like ***format translation*** or translation to a common interlanguage, such as ***Rich Text Format (RTF)***, have been taken. Virtually all wordprocessors, however, have the capability to save text as a standard ASCII file, in some cases with or without line breaks specified, and this format is universally compatible.

they can be used together in surprisingly powerful ways. This is inherent in the way the simple `speculum` example above works; further examples may be found in Section 6 below.

4. Grammar

The Unix language family is inflected. This is not common (though not unknown,[21] either) in computing languages. There is, for instance, complex clausal syntax, including clitics, marked lexical classes, a case system, and a very powerful morphological system for paradigmatic matching called ***regular expressions***.

Regular expressions permeate Unix. Originally developed by the logician Stephen Kleene (1956), they found their place as Type 3, the lowest, of the Chomsky Hierarchy (Chomsky 1963), where they are equivalent to finite-state ("right-linear") grammars, or finite automata. The most common type of regular expression morphology is the use of "`*`" (the ***Kleene closure***) to indicate "any string" in such contexts as `*.doc`, meaning (in context) all files with names ending in the string "`.doc`"; this is the ***shell*** regular expression dialect, the simplest but not the only one. The Unix program `egrep`, for instance, uses an elaborated version of regular expressions to search text files for lines containing strings matching a expression.

Suppose, for instance, one has a World Wide Web server, which stores a record of each "hit" (i.e, file request) on a separate line in a logfile with a long and unmnemonic name. Suppose further that one has decided to think of this file as `weblog` for convenience. Then one creates a ***shell variable***, stores the name in it, and then uses `weblog` to refer to that file thereafter. This could be done by putting a line like the following in one's `.cshrc` file:

```
weblog=/usr/etc/bin/httpd/log
```

Once set, this variable is available, and may be referred to, in any command. Unix makes a philosophically nice use/mention distinction here between the variable itself and the ***value*** of the variable. That is, *`weblog`* is the variable, while ***`$`****`weblog`* is its content, namely the string "`/usr/etc/bin/httpd/log`".

To return to our example: this web log file, however it is named, or referenced, is filled with automatically-generated information from the Web server program, which runs in the background[22] The ***format*** of each line is invariable, since it's generated automatically, and begins with the date in a particular format (e.g, "`01/3/96`"), followed by other information, terminating in the name of the file requested and the number of bytes served. Then the command:

```
egrep umich $weblog
```

will find and display every line in the file `web.log` containing the string "`umich`".[23] There

[21] The **apl** programming language is an example of a polysynthetic computer language, for instance.

[22] The usual metaphor is that programs like those that serve files on the Web (`httpd`), respond with personal information on the `finger` command (`fingerd`), or make ***ftp*** connections (`ftpd`), etc, are ***d(a)emons***, whence the suffixal `-d` in their names. Daemons are invisible slavey programs that run only in the background, checking every so often to see if what they're looking for has come in, responding to it if it is, and going back to sleep. This metaphor refers to Selfridge's (1958) "Pandemonium" model of perception, which is fairly close to the way many net programs work.

[23] Note that the argument immediately after `egrep` on the command line is interpreted as a regular expression, while the one following that is interpreted as a file name; we have here a system of subcategorization that specifies the lexical class and morphological properties of verbal case roles. The string to be matched by `egrep`

may be very many of these, and one may only want to know how many, so the output of `egrep` may be ***piped*** to `wc -l`[24]:

```
egrep umich $weblog | wc -l
```

which simply provides the number of lines found, instead of a listing of all of them. This works for more complex strings, too, though one is well-advised to use quotation marks to delimit the ***search string***. If, for example, one wanted to count how many requests were made for a given file, say "FAQ", on a given day; the command would be:

```
egrep '01/23/98.*FAQ' $weblog | wc -l
```[25]

Since "**.**" matches any character and "*" matches any number of the preceding character, "**.***" comprises a regular expression idiom that matches any string at all, and "`01/23/98.*FAQ`" thus matches any string[26] containing the date and the file name, in that order.

We alluded above to the analogs to the ablative (source) and dative (goal) cases, with reference to the input or output of a command on the ***command line***, i.e, whatever the user types after the Unix prompt.[27] It is worth looking at the command line in some detail, since it is the principal linguistic structure of Unix, analogous to the Sentence level in natural language. The basic syntactic structure is

```
command [-switches] [arguments]
```

That is, verb, plus optional (marked) adverbials, plus optional noun phrases; some command verbs are intransitive, some are transitive, some are bitransitive, and some vary in the number of arguments they take. These linguistic analogies are reasonably straightforward: virtually every Unix command is, as the name suggests, an imperative verb, directing some action to be taken; the arguments, like nouns, refer to a person, place or thing, generally a ***user***, a ***path*** or ***directory,*** or a ***string*** or ***file***, respectively; and the switches, like adverbials, specify optional modes and manners in which the action is to be performed.

Commands are not always simplex; they may be conjoined or embedded, and there can be quite intricate flow of information from one to another, as we have seen. Concatenation of commands is straightforward: to instruct Unix to perform several commands, one merely separates them with semicolons; when the RETURN key is pressed, each is performed in order.

```
cd ~/News; trn; cd
```

changes the current directory to one's own `News` directory (which is used by news readers like `rn`, `trn`, or `tin`), invokes `trn`, and returns to the ***home directory***[28] when `trn` exits. These are

---

need not be quoted (though it may be). However, one is well advised to 'single-quote' complex search strings containing space and other special characters, to avoid unexpected misinterpretations.

[24] From "word count"; `wc` counts lines, words, and characters; the optional `-l`, `-w`, and `-c` switches say which.

[25] It is also possible to have Unix supply the current date in the appropriate format as the search string (thus making the command indexical), by means of the *backquote convention* (see below):
`egrep` `` `date +%d/%h/19%y` `` `$weblog | wc -l`

[26] As a matter of fact, it will match the **longest** such string in the line, if there is any ambiguity in the match.

[27] The prompt is usually "%" (possibly with other information, like the name of the machine), and sometimes "$".

[28] `cd` (from "change directory") changes the directory location to the one specified; when issued without an argument, it defaults to the user's ***home directory***.

coordinately conjoined clauses,[29] unlike the subordinate complement clauses produced by input/output redirection, where each successive command depends on a previous one.

Or, using the ***backquote convention***, whole commands may function as nouns, like complement clauses[30], with the output of the complement command functioning as the argument of the matrix command. Quoting a command inside backquotes "`" runs that command in the background and then uses its output as the argument for the main command, so that:

```
finger `whoami`
```

first runs the `whoami` program, which returns the current user's login name, then uses that string as the argument for `finger`, which provides information about a user from their name.

Unix is a ***multi-user***, ***multi-tasking*** system. This means that several (on larger systems, several hundred) people can simultaneously use the same machine, and each of them can, in theory, run several processes simultaneously.[31] With such complexity, it is obvious that there are a lot of people, places, and things to refer to, and Unix has a file and directory system that accommodates this. The basic unit in Unix, as in most computer systems, is the file, which is by default a text file. Each file resides in some directory, and every user has their own ***home directory***, usually named for their login ID.

Thus, if my login is `jlawler`, my home directory on a given Unix system might be `/usr/jlawler`, while `hdry`'s home directory would be `/usr/hdry`. A file named `wordlist` in my home directory has a full ***pathname*** of `/usr/jlawler/wordlist`, and it would be accessible from anywhere on the system with that name. Most Unix systems use a special referential convention to the effect that "`~jlawler`" means "`jlawler`'s home directory", while "`$HOME`" is an indexical, referring to the current user's (first person singular) home directory. Finally, one always has a **current** directory, which is thought of (and referred to) in locative terms: i.e, one is **in** `/usr/hdry` and **goes to** it with the `cd` command: *`cd /usr/hdry`*. Files referred to without a pathname, i.e, by the name of the file alone (e.g, `wordlist`) are interpreted as being in the current directory by default. Thus, for anyone who is in my home directory, "`wordlist`" is sufficient; for someone in `/usr/hdry`, "`~jlawler/wordlist`" is necessary; and I can always refer to it as `$HOME`/`wordlist`, no matter what directory I'm in.

Directories may contain other directories, and references to them are simply concatenated with the directory separator slash "**`/`**". A file `wordlist` that was in a ***subdirectory*** `lists` under a subdirectory `English` under my home directory would have a fully-specified pathname of `/usr/jlawler/English/lists/wordlist`, and other users could reference it this way, or as `~jlawler/English/lists/wordlist`, etc. The concept of ***hierarchical directories*** originated in Unix, but it has spread to most modern systems. Users of DOS will be familiar with this convention, although DOS uses backslash "****" instead of slash as a directory separator; in Macintosh usage, directories are called "folders", and colon "**:**" is used in pathnames.

---

[29] With the usual Western "narrative presupposition" to the effect that the conjuncts occur in the order they are mentioned. In this case, of course, it is not so much a presupposition as a performative.

[30] In particular, they are very reminiscent of conjunctive embedded questions of the form

I know who Bill invited

where in fact what I know is the **answer** to the question "Who did Bill invite?".

[31] Any Unix command, for instance, can be run "in the background", i.e, simultaneously, by suffixing "&" to it.

Unix ***filenames*** are normal ASCII strings of varying lengths[32], and may contain any alphanumeric character, and a number of non-alphanumerics. They may (but need not, except for special purposes) end with an extension of a period followed by a short string denoting the file type. Thus, C program code files end in `.c`, ***HTML*** files accessed by Web ***browsers*** end in `.html`, and compressed `tar` archive files end in `.tar.Z`. Some programs require special extensions on filenames, but most Unix tools do not, though they are often defaults.

In natural languages, imperative forms are almost always regular, frequently based on simple verb roots unless elaborated by a politeness system. In dealing with machines politeness is not an issue, hence the lack of verbal inflection *per se* in Unix. There is, however, an elaborate clitic system, called ***switches***.[33] By way of example, let us examine the common Unix command `ls`, which displays file names and information, like the `DIR` command in DOS. The online manual entry for `ls` (the Unix command to display it is *`man ls`*) starts this way:

```
LS(1)          UNIX Programmer's Manual
NAME
   ls - list contents of directory
SYNOPSIS
   ls [ -acdfgilqrstu1ACLFR ] name
```

Figure 4. Top of man page for `ls` command: `man ls | head`

The heading shows that the `ls` command is in part 1 of the Manual (most ordinary commands are there); the next part gives its name (in lower case) and its purpose. The "synopsis" then gives all the possible *switches*, each a single character, that it may take. The square brackets signal that they are optional; the hyphen character precedes any switch markers, which may be concatenated, in any order. The rest of the man page then details the operation of `ls`; in particular, the operation of `ls` with each switch is discussed separately.

For instance, here is what it says about the `-t`, `-s`, and `-r` switches:

```
-t   Sort by time modified (latest first) instead of by name.
-s   Give size in kilobytes of each file.
-r   Reverse the order of sort to get reverse alphabetic or
     oldest first as appropriate.
```

Figure 5. From man page for `ls` command:

effects of *-t, -s,* and *-r* switches.

[32] The length of filenames was one of the major differences between BSD Unix and System V; Berkeley filenames could generally be longer, and this caused many problems in adapting programs.

[33] Sometimes called ***options***. These are generally adverbial in nature.

This means that `ls -rst`[34] will present the names of the files in the directory `name`,[35] with their sizes, sorted by time, most recently modified files last. Each of the other letter switches listed in the Synopsis does something different; further, what each does is context-sensitive, in that it may vary if used with some other switch, like the combination of `-r` with `-t`, which changes the sort order from reverse alphabetic to reverse temporal.

`ls` has 18 switches in this dialect; which is a larger degree of modification than most Unix commands have. Each command has a unique set of switches, however, most of which are only infrequently needed. The syntax ranges from extremely simple to rather complex. Below are some syntactic synopses of other Unix commands. Underlining indicates variables to be supplied by the user, and square brackets optional ***elements*** – switches separately bracketed require separate "**-**" prefixes. Vertical bar "|", like linguists' curly brackets, requires a choice among elements.

rev:	reverse the order of characters in each line	**rev [file]**
cp:	copy files	**cp [-ip] file1 file2**
mv:	move or rename files	**mv [-i] [-f] [-] file1 file2**
head/tail:	give first/last few lines of a file	**head/tail [-count] [file]**
mail:	send and receive mail	**mail [-v] [-i] [-n] [-s subject] [user ...]**
uniq:	remove or report adjacent duplicate lines	**uniq [-cdu [+|-n] [inputfile [outputfile]]**
diff:	display line differences between files	**diff [-bitw] [-c [#] | -e | -f | -n | -h] file1 file2**
spell:	report spelling errors	**spell [-blvx] [-d hlist] [-h spellhist] [-s hstop] [+localfile] [file]**

Figure 6. Synopses and syntax of selected Unix commands.

In each of these, the switches may be expected to have a different meaning. All this might seem a large burden to place on the user, and it would indeed be excessive, were it not for the facts that:

- a complete list with glosses is always available via the `man` command
- some, at least, of the switches are mnemonic (in English): `-time`, `-reverse`, `-size`
- one need never learn any switch more than once, since any useful configuration can be made into an ***alias*** or ***script*** with a name chosen by (and presumably significant to) the user; thus `ls -rts` can be renamed, say, `reversedate` with the command

```
alias reversedate ls -rts
```

Any command, or sequence of commands, can be given a name, thus making it into an idiom, or a little program. This facility is provided by the Unix ***shell***, the tool that coordinates the other tools by interpreting commands. There are two principal shells, and each provides a different facility for command formation. `csh`, the "**C-shell**",[36] provides ***aliases***; it is principally used ***interactively***, where it identifies itself with a "%" prompt. `sh`, the "**Bourne shell**",[37] is used mostly to interpret files containing ***shell scripts***; it has fewer ***interactive*** features, but when it is

[34] Or the command `ls -r -st`, or `ls -rs -t`, or `ls -t -r -s`, etc. Switches need not be concatenated.

[35] In case `name` is not specified (as it isn't in the command in the previous line), `ls` assumes the current directory is intended. This is an example of the Software Tools philosophy in action; instead of requiring literal compliance with the syntax, make the defaults useful.

[36] So-called because it incorporates many features of the C programming language.

[37] Named after its inventor. `sh` was an earlier shell, superseded for interactive use by `csh`; however, its simplicity has made it the default choice for shell programming.

being used interactively, it identifies itself with a "$" prompt.[38]

The command `reversedate` could be either an alias (as in the example above), or a shell script. Generally, simple commands like this are more likely to be made into aliases, since the process is easier, and doesn't involve creating and activating a file. Of course, to make an alias permanent, it is necessary to record it; each `csh` user has a file called `.cshrc`[39] that may be customized in a number of ways, including a list of their aliases. One of the first aliases some users put in `.cshrc` is something like the following:
`alias define 'edit $HOME/aliases;unalias *;alias -r $HOME/aliases`[40]
which allows them to define new aliases on the fly.[41] A good rule to follow is that any command one notices oneself typing more than a few times should become an alias with a mnemonic name; and to keep track of these, it is also useful to have a few aliases whose purpose is to remind oneself of one's previous aliases. The Unix tool `which` is helpful here; *`which define`*, for instance, will return the following information:[42]

```
define - aliased to: edit $HOME/aliases;unalias *;source $HOME/aliases
```

`egrep` can be used to advantage as well, to refresh one's memory about previous lexicography. Suppose you have several aliases for `ls` with various switches, but you don't recall all of them; the following command will then print each line in `.cshrc` containing the string "`ls `".[43]

```
% definitions 'ls '
alias dates        'ls -sACFt | more'
alias dir          'ls -alF'
alias lf           'ls -sF'
alias ll           'ls -l'
alias lc           'ls -lc'
alias whichls      'ls -l `which \!*`'
```

Figure 7. Operation of the *`definitions`* alias.

by means of the following alias:

```
alias definitions  "egrep  \!*  $HOME/aliases "
```

---

[38] Others include `ksh`, the "Korn Shell", which combines features of `sh` and `csh`, and `tcsh`, an improved `csh`.

[39] The period prefix is obligatory; most Unix programs use such ***dot files*** containing customizations or preferences. The `ls` command does not display dot files unless instructed to with the `-a` switch.

[40] The three successive commands separated by semicolons respectively: (a) edit the user's `aliases` file, presumably to insert a new alias; (b) remove all current aliases; (c) reload the presumably modified `aliases`.

[41] Provided this is where their aliases are; the following command should be the last line in the dot file `.cshrc`: *`alias -r $HOME/aliases`* This will load the alias file when the shell starts, e.g, at login.

[42] Besides aliases, `which` will also locate any executable files (shell scripts or programs) matching a name that are in the user's ***path***. As such, a command like `which foobar` answers the question: "If I type `foobar`, will anything happen?".

[43] Note the final space, to restrict the match to commands. Quotes are used to disambiguate strings whenever necessary, as with spaces and special characters, but they are not necessary with simple strings. There is a principled pragmatic difference between single and double quotes in Unix.

"**\!***" is the C-shell code for a command parameter, i.e, whatever appears on the command line after the alias; in this example, it is translated by the shell into the string "`ls `" (note the space), and passed on to `egrep`, which dutifully searches `$HOME/aliases` [44] for lines containing this string and prints the result.

Of the various aliases above, `dates` shows multi-column output sorted by time, oldest last, and pipes the output to a file viewer that shows only a screen at a time; this is useful for directories with a large number of files. `ll` and `lc` both produce a "long directory", with all details of each file printed on a separate line; `lc` is sorted by time of last edit, most recent first. The last alias, `whichls`, uses the backquote convention; `which` finds executable programs, scripts, or aliases anywhere in the user's *path*, but it returns only the name and location, and not the size, date, or any other information. If one wants more information, one can then use `ls` to find it; but it's often convenient to combine the steps, as here.

By contrast with an alias, a *shell script*:

- is interpreted by the Bourne shell `sh` (aliases are interpreted by `csh`, the C-shell; this means that aliases and scripts use somewhat different conventions)
- consists of a file and resides on disk, like other Unix programs (aliases are loaded from a file when `csh` starts at login and are thus in-memory commands)
- is generally longer and more complex than an alias, which is usually a short sequence of commands or a mere synonym for a single command

As an example of a shell script, consider a problem one often encounters: making a simple change in multiple files. This could be done individually with an *editor*, making the change by hand in one file, loading the next file and making it by hand again, etc. But this is not only wasteful of time but also prone to error, and Unix provides better facilities for such tasks. Suppose the files in question are all ***HTML*** files in a single directory on a Web server, and that what needs to be done is to change a ***URL*** link that may occur several times in each file (or may not occur at all) to a new address, because the server that the URL points to has been renamed (this particular task that is likely to be with us for some time).

A two-step process will serve best here: first, a shell script (call it `loopedit`) to *loop* over the files and edit each one with the same editing commands, and a separate file (call it `editcmds`) of editing commands. This has the benefit of being reusable, since subsequent editing changes can be made with the same script merely by changing the contents of `editcmds`. The Unix `cat`[45] tool will print any file on the screen, so we can see the files:

```
% cat loopedit
#!sh
for i in *.html
 do
  ex - $i < editcmds
 done
```

Figure 8a. The *`loopedit`* script, with commands in *`editcmds`* file (Fig. 8b).

[44] Each user's dot and customization files are located in their home directory, to which `cd` returns when invoked without arguments, and which is contained in the ***system variable*** `$HOME`.

[45] From "catenate", since the tool will concatenate multiple files named as arguments.

The first line invokes the sh shell to interpret the script (one gets to specify the language and dialect). The next line (a "`for`" statement) instructs the shell to carry out the line(s)[46] between the following "`do`" and "`done`" markers once for each file ending in "`.html`",[47] just as if it were typed at the keyboard. At each successive iteration of the command, the shell variable "`i`" is to be set to the name of each successive file in the set of those ending in "`.html`". The fourth line is the command itself; it runs the `ex` line editor (using `ex`'s *silent* switch "`-`" that tells `ex` not to print its usual messages on the standard output for each file, unnecessary with multiple files), and the name of each file (referenced as the value of `i`, or `$i`) as its argument. `ex` is further instructed by the input redirection (ablative) marker "`<`" following the argument to take its next input – the commands themselves – from the file `editcmds`.

The contents of `editcmds` can be similarly displayed:

```
% cat editcmds
g/www.umich.edu\/\~/s//www-personal.umich.edu\/\~/g
wq
```

Figure 8b. The *`editcmds`* file, input to *`ex`* in the *`loopedit`* script (Fig 8a).

There are only two lines necessary; the first makes the changes, and the second saves ("writes") the file and quits. The second line is trivial, but the first is fairly complex.[48] There are several technical wrinkles, due to peculiarities of `ex` commands and of URL syntax, that render it more complex than usual; this makes it a good example of a number of things, and worth our while parsing out character-by-character below.

First, let us examine the precise change to be made. URLs begin with the address of the server to be contacted; in the case of the University of Michigan, there are several, all beginning with "`www.`". As the Web has grown, it has become necessary for some Web pages to be moved to different servers to equalize the load. In particular, at the University of Michigan, personal Web home pages, which are named using a tilde convention similar to the Unix home directory convention, have had to be moved from the server having the address "`www.umich.edu`" to a special server for such pages only, with the address "`www-personal.umich.edu`". Thus Eric Rabkin's home page, which used to have the URL "`www.umich.edu/~esrabkin/`", can now be found at the URL "`www-personal.umich.edu/~esrabkin/`", and this change must be made for thousands of URLs in many Web pages. The change should only be made to personal pages, however; other (e.g, departmental) pages, which are not named with the tilde convention, remain on the original server and retain the "`www.umich.edu`" address.

---

[46] There can be many lines between `do` and `done`, but we need only one for such a simple task.

[47] Thus, by default, lying in the current directory; this also applies to `editcmds`. This means that `editcmds` should be in the same directory as the files to be edited, and that that should be the current directory. `loopedit`, however, need not be there, since as an executable script it can be located anywhere in the user's ***path*** (the series of directories searched by the shell to find programs to be executed). `ex` itself resides in a system directory, where it is accessible to (but not modifiable by) ordinary users.

[48] And in fact took a couple of tries to get right. However, once debugged it can be saved and reused indefinitely, a major feature of the Software Tools philosophy.

We therefore need to search for all lines in a file that contain the URL address string "`www.umich.edu/~`", and to change each occurrence of this string on each of these lines to "`www-personal.umich.edu/~`". That is what the first line does. The "`s`" (for "substitute") command in `ex` has the syntax `s/`*`re`*$_1$`/`*`re`*$_2$`/`, where *`re`*$_1$ and *`re`*$_2$ are regular expressions; it substitutes *`re`*$_2$ for *`re`*$_1$, and is thus a variant of the *Structural Description : Structural Change* transformation notation that generative linguists put up with for over a decade. `s//`*`re`*`/` is a zero pronominal reference, and substitutes *`re`* for whatever the last search string has been; in this case, that has already been specified by the preceding search (the slash-delimited regular expression beginning the line). In the event of a search failure, the "`s`" command will not execute. However, this particular command has a special twist: slash "**`/`**" and tilde "**`~`**" are themselves both meaningful characters to `ex`, and thus cannot be searched for directly.

Slash is used to delimit search strings,[49] and in order to search for slash itself in a string, or for strings containing it, it must be ***escaped*** with a backslash "**`\`**" literal prefix. I.e, "**`\`**" quotes the next character literally, so that the string "**`\/`**" means "the character '**`/`**' "; the slash will not be interpreted by `ex` as a string delimiter. Similarly, unescaped tilde implicitly refers to the last ***replacement string*** (***`re`***$_{\mathbf{2}}$) used in a previous "`s`" command (just as unescaped ampersand "`&`" refers to the ***search string*** (***`re`***$_{\mathbf{1}}$) of the current "`s`" command), and to the empty string if there have been no previous "`s`" commands, which will be the case in this script. So the actual string we must instruct `ex` to search for is "`www.umich.edu\/\~`", with both slash and tilde ***escaped***,[50] and the replacement string is "`www-personal.umich.edu\/\~`".

The two "`g`", for "general", commands, one at the beginning and one at the end, refer to two different contexts. The initial "`g`" instructs `ex` to find **all** lines in the file with an occurrence of the following search string, and to execute the command following on those lines where it is found, while the final "`g`" refers only to the (line-oriented) "`s`" command, and instructs `ex` to perform all possible substitutions on the indicated line; this covers the case where there is more than one occurrence of the string on the line. Without the suffixal "`g`", the "`s`" command could only be executed once on any line.

[49] As in this command. Although any character may function as a string delimiter in an "`s`" command, slash is most common. Using a different character for the "`s`" command would eliminate one pair of backslashes in this command. However, slash **is** the canonical delimiter for searching and may not be changed in that sense.

[50] Actually, we could do without the escaped slash in the **search** string. Since any string containing "`www.umich.edu`" followed by one character followed by a tilde is acceptable, we could simply use a period, which will match any character, instead of an escaped slash in the search string: "`www.umich.edu.\~`". Indeed, the periods in "`www.umich.edu`" will match any character, too; the fact that they are intended to match literal periods is entirely coincidental. However, in the **replacement** string, period is not interpreted, while slash is, so the escaped slash is necessary there.

g ....."general" .....find all occurrences of the following search string in the file

/.....string delimiter .....begin search string; find a line containing the following string and make it the current line

www.umich.edu\/\~.....string to be searched for, including escaped (i.e, literal) slash / and tilde ~ characters

/.....string delimiter .....end search string

s ....."substitute"..... replace first following string with second **once** on current line

/.....string delimiter .....begin search string

(nothing) ..... use last previous search string by default

/.....string delimiter .....end search string, begin replacement string

www-personal.umich.edu\/\~ .....replacement string, including escaped (i.e, literal) slash / and tilde ~ characters

/.....string delimiter .....end replacement string

g ....."general" ..... execute the preceding "s" command as often as possible on a line

Figure 8c. Parse of the edit command in *editcmds* file (Fig 8b) interpreted by *ex* in the *loopedit* script (Fig 8a).

With these files in place, the only thing remaining is to activate `loopedit` as an ***executable*** (i.e, program) file with the `chmod`[51] command. From then on it works the same as any Unix program. One need hardly add that, with several hundred Unix tools available to be used, singly or together, plus dozens of specialized sublanguages for instructing them, shell scripts offer unlimited possibilities for automated text processing to every Unix user. For instance, the LINGUIST List is edited, distributed, archived, abstracted, and put on the Web via a large suite of Unix scripts that depend on tools like the ones discussed in this chapter.

5. Editing and Formatting

The Unix toolbox always includes an ***editor***, actually several editors, of several different kinds. Editors are programs that create and change the contents of ASCII files. They may do many other things as well, and some, for instance `emacs`, can become an entire environment. An editor is a significant part – the part that connects keyboard, screen, and disk – of the usual microcomputer word-processing programs; the usual metaphor is a typewriter, without paper, but with a memory. A wordprocessor is a large complex program with many capabilities; the usual metaphor is a typewriter that not only has paper, but also a print shop, an art studio, a type foundry, and a reference library. Wordprocessors are used to produce actual printed pages, while an editor need only fool around with bits and bytes, not fonts and footnotes. An editor is thus usually much smaller and faster, because it is a tool that only does one thing and doesn't try to do others.

[51] From "change modifiers", a reference to the executability of the file. The command that activates `loopedit` as a program is `chmod u+x loopedit`, which means that the user adds executability to the file's properties. If this seems difficult to remember (and it is hardly intuitive), an alias renders it more memorable:
`alias activate chmod u+x`

They are also especially useful in Unix, because Unix was originally invented by programmers for programmers, and its editors, though mostly used for ordinary writing, are designed to be especially useful for programmers. In order to make a shell script or save an alias, for instance, one must use an editor. Which one? That is a semi-religious matter for many. The choices include:

- **`pico`**, the screen editor that is a part of the `pine` e-mail package. Many people have found it easy to use, and the editor is available separately from email. Furthermore, `pico`'s key commands are a subset of the standard key commands for:
- **`emacs`**, the most powerful and flexible editor in the computer world. It can be found on most Unix systems in academia, though not always in business. It is the product of the Free Software Foundation and must be given away free. Though it is not simple to install, nor to learn completely, it is thoroughly programmable (in Lisp) and can do almost anything with ASCII text. `emacs`' main competitor is:
- **`vi`**, universally pronounced /viyáy/, which, growing out of a short tradition of line editing, was the first screen editor available on Unix, and, as part of the standard Unix distribution, may be found on every Unix system, along with its relatives:
- **`ex`** and **`edit`**, essentially command-line versions of `vi` (they **become** `vi` with the "`v`" command); and **`ed`**, the first Unix line editor, still a functional tool within its limitations.

All of them work, but they all work differently. In this chapter, I use the `ex` line editor, both as a least common denominator, and because it is the editor I use myself by choice for most simple file editing tasks like adding or modifying aliases, mail names, text Web pages, and writing small scripts. It is fast and convenient for these tasks, and can easily be automated. Thus the details of the editing in the transcriptions that follow are independent of the rest, in that the editing could have been done visually.

But it's irrelevant, from the standpoint of the Software Tools philosophy, or of Unix, which tools you use, as long as they work, because all of the tools work together. There is thus a wide choice of programs available for virtually any task, and editors are no exception. Indeed, editors are so important to programmers that they are constantly improving them, often on their own time, for glory; and since programming glory involves efficiency and power, among other things, this leads to some very interesting tools.

There is an important class of word-processing tool program, called a text *formatter*, which is also usually part of a wordprocessor, but may be used as a separate tool in combination with an editor. Examples are *TeX* and *LaTeX*, programs used by many scientists to produce technical text, and the Unix programs `roff` (for 'run off') and `troff` (for 'typesetter runoff'), all of which implement special printing instructions from special ***tags*** or commands embedded in a text file. Formatters and embedded commands are common with file structures that follow ***SGML*** or ***HTML***, or some equivalent ***markup*** scheme, like Web browsers (see chapters 1, 4, and 6 in this book for further discussion of markup, TeX, and SGML). In all of these, the *stream and pipe* metaphors of information flow control via tool programs can easily be discerned. Separate formatter programs are not as widely used in ordinary writing as previously, since the locus of most text construction has moved to microcomputers with full-featured wordprocessing programs with built-in formatting; but they are still a common type of program, one of the larger class called ***filters***.

6. Filters

As mentioned above, a filter program is one that takes input (prototypically textual input) from some source, performs some regular transformation on it, and sends the resulting output to some terminus. This may be sequential, like the speculum example, or interactive, like a Web browser interpreting HTML code; but frequently enough filters employ regular expressions, used in special filter languages, to specify their transformation.

Regular expressions are far more powerful than simple string search examples would suggest. Besides "**.**" and "`*`", there are a host of other special inflections with special senses, as in any synthetic morphology:

- "`?`" means 0 or 1 of the preceding character, so "`s?`" means that "`s`" is optional
- "`+`" means a string of **one** or more of the preceding characters ("`*`" is **zero** or more)
- "`|`" indicates alternation, so "`to(day|morrow)`" matches either *today* or *tomorrow*
- "`[A-Z]`" matches any single character from the ASCII range between "`A`" and "`Z`"
- "`[^AUZ]`" matches any character **except** "`A`", "`U`", and "`Z`"
- "`[^A-Z]`" matches any character **except** an uppercase letter, so
- "`[A-Za-z0-9]`" matches any single alphanumeric character, while
- "`[A-Za-z0-9]*`" matches any string consisting only of alphanumerics, and
- "`[^0-9]*`" matches any any string that does not contain numeric characters
- "`$`" means the end of a line, and "`^`" means its beginning, so "`^$`" matches an empty line.

Figure 9a. Simple examples of regular expressions and the strings they match.

Special characters intended to be used literally, rather than interpreted like this,[52] are preceded by "`\`", thus "`\. `" matches a period with two spaces afterward, and "`\\`" matches a single (literal) backslash.

Some other examples of regular expressions, all working with `egrep` (and all delimited with `'`quotes`'`):

[52] There are other possible interpretations; for instance, the `ex` editor has a special meaning for slash "`/`". See above for examples.

`' .*is?tic( |al( |ly ) '` ..... any word ending (note the spaces) in *-itic*, *-istic*, *-itical*, *-istical*, *-itically*, or *-istically*

`' [A-Z][a-z]* '` ..... any Capitalized word (note the spaces)

`' [A-Z][a-z]*[A-Z][a-z]+ '` ..... any CapiTalized word containing one other CapiTal

`'^[^ ]*$ '` ..... a complete line containing no spaces (* matches empty line, + doesn't)

`'^[A-Z].*\.$ '` ..... a complete line beginning with a Capital and ending with a period.

`' spr?[^\.?,:;!]*[ \.?,:;!]'` ..... any word beginning with *sp-* or *spr-*;
this expression specifies that the string must begin with a space, and may not **contain** period, question mark, comma, colon, semicolon, or bang, while it must **terminate** with one of them, or with space, making it suitable for searching in normally punctuated text.

`'[A-Z][A-Za-z]* [A-Za-z]*: ?([Tt]he|[aA]n?) [A-Z]?[a-z]* of .*'`
..... a specification for certain styles of academic title like
Regular Titles: An analysis of technical paper nomenclature.

Figure 9b. Complex examples of regular expressions and the strings they match.

Besides `egrep`, many other Unix tools can use these regular expressions. The text editors `ed`, `ex`, `vi`, and `emacs`, for example, can perform very complex string manipulations based on regular expressions. In addition, the text filter languages `sed`,[53] `awk`,[54] and `perl`[55] make extensive use of regular expressions. `sed`, from "stream editor", is the simplest filter tool. It can do the same things as `ex`, but operates on the entire text stream with predetermined instructions. It is useful for repetitive editing tasks; since these are character-based editors, `sed` is best at character-level manipulations. `awk` is a more complex language, based on the concept of the word instead of the character, but still oriented toward sequential operation on each line in a filter operation. `awk` is somewhat more like a conventional programming language, and one can write quite complex programs in it, but is simple enough for useful short programs to be written on the fly. It works best for formatting repetitive and relatively predictable text data. `perl` is a general-purpose programming language oriented toward text handling, which is very widely used on the Internet, especially the Web. It is very powerful and efficient, and, though relatively easy to learn, is more complex than `awk`, and does not presuppose the filter metaphor so literally.

---

[53] From "stream editor". `sed` can do the same things as `ed`, but operates on the entire text stream with predetermined instructions, instead of interactively. It is useful for repetitive editing tasks.

[54] An acronym of "Aho, Weinberg, Kernighan", the authors of the program. `awk` is more powerful than `sed`, and is designed specifically for use as a text filter, especially for repetitively-formatted files.

[55] An acronym of "Practical Extraction and Report Language". Perl is a full programming language, oriented towards manipulating large texts. It is widely used on the Web for CGI scripts; a very simple example is the Chomskybot , whose URL is: `http://stick.us.itd.umich.edu/cgi-bin/chomsky.pl`
the URL of its Perl script is: `http://www.lsa.umich.edu/ling/jlawler/fogcode.html`
I do not consider Perl much further here, except to point out ways of learning it easily, by automatic translation.

7. Unix resources for users

There are thousands of books on Unix in press. Since it has not changed in its basics since the 1970s, even books published a long time ago can still be useful. Rather than attempt to survey this vast and variable market, I will point to a few standard references (many of which can be found in used book stores).

I have already mentioned the various books by Brian Kernighan & assorted co-authors; they remain standard, even though their examples show signs of aging in some environments. The single best source of printed information (both on Unix, and on regular expressions and their use in filters as well) for sophisticated beginners remains the first four chapters of Kernighan and Pike's classic (1984) *The UNIX Programming Environment*, which treat much the same topics as this chapter. This is pretty condensed stuff, but admirably clear; Kernighan is not only the *k* in *awk*, and one of the creators of UNIX, but also one of the best writers in information science.

For those curious about how software is designed and developed, Brooks (1995) explains a great deal about the mistakes that can be made and the lessons that have been learned. For the historically-inclined, Salus (1994) covers the territory very well. Raymond (1996) is the latest installment of an online lexicographic project called the ***Jargon File***; it contains a lot of good linguistics and ethnography, and some wonderful metaphors. Other books of historic and ethnographic interest include Kidder (1981), Levy (1984), Libes (1989), and Stoll (1990).

Regular expressions are covered in every book on Unix. They are especially well-treated in books on filter languages. A good source for all of these is the set of books from O'Reilly and Associates (the ones with the strange beasts on the cover); they publish good manuals on `sed` and `awk`, regular expressions, Perl, and many other topics, centered on Unix and the Internet.

When evaluating a Unix book for reference purposes, look for a thick book with a good index and multiple appendices. Like good linguistics, it should give copious examples of everything, and say what each is an example of. A good check for the index (a vital part to any reference grammar) is to see if it's easy to find out how to refer to command-line arguments in a C-shell alias – you should be able to find the arcane formula (`\!* or \\!*`) without looking hard. Check the index also for mentions of useful commands like `sed, ls, head, sort, uniq, rev,` and `awk`. Check the table of contents for a separate section on regular expressions near the beginning of the book; there should also be discussions (ideally, entire sections) on aliases and customization, as well as shell programming in both the Bourne shell **and** the C-shell. Both `vi` **and** `emacs` should be treated in detail, with examples, and commands for both should be listed in detail.

Marketing hype about how the book makes Unix easy, even for those unwilling to attend to details, is extremely suspect, just as it would be if it were encountered on a linguistics book; one needs reference grammars as well as phrasebooks.

For technical reference, the official source is the Unix edition of the *Bell System Technical Journal* (1979, 1987), and Bell Laboratory's *Unix Programmer's Manual*, which is largely a collection of standard `man` pages. (The online `man` system **always** provides the most up-to-date and deictically-anchored – and terse – Unix information available). Stallman (1993) is the standard reference on the editor `emacs`, by its designer and author; the result is comprehensive, though as always the author of a program is not necessarily the best possible author of its manual.

Below is a list of ***URL***s for online resources from this book.

- The top Web page for this book, at
  http://www.routledge.com/linguistics/using-comp.html
  (from which one can reach online Appendices for **all** chapters)
- Chapter 1 ("Introduction") of this book, complete online, at
  http://www.routledge.com/linguistics/introduction.html
- The Glossary of all the technical terms in ***boldface italics*** used in this book, at
  http://www.umich.edu/~jlawler/routledge/glossary.html
- The Bibliography of all references cited in this book, at
  http://www.umich.edu/~jlawler/routledge/bibliography.html
- Chapter 5 ("The Unix™ Language Family") of this book, complete online, at
  http:/www.umich.edu/~jlawler/routledge/unixlanguage.pdf
  (i.e, this file)
- The online Appendix for Chapter 5, at
  http://www.umich.edu/~jlawler/routledge/unix.html
  which points to pages on
  - General Resources, at
    http://www.umich.edu/~jlawler/routledge/unixweb.html
  - Regular Expressions, at
    http://www.umich.edu/~jlawler/routledge/regex.html
  - Filter Languages (sed, awk, and perl), at
    http://www.umich.edu/~jlawler/routledge/sedawkperl.html
  - Shells, Aliases, and Scripts, at
    http://www.umich.edu/~jlawler/routledge/alias.html