UNIX® Evolution: 1975-1984
Part I – Diversity

Copyright © 1984, 1985, 1986
Ian F. Darwin

SoftQuad Inc.

Geoffrey Collyer

University of Toronto

ABSTRACT

This article traces some of the intermediate history of the UNIX Operating System, from the mid nineteen-seventies to the early eighties. It is very slightly updated from an article that appeared as ‘‘The Evolution of UNIX from 1974 to the Present, Part 1’’ in *Microsystems*, November, 1984 (Vol 5, Number 11), page 44. It was intended as part 1 of 3; unfortunately that issue was also the last issue of *Microsystems*. This part discusses ‘‘Research UNIX’’; v6, v7 and v8, and tells the tale of many programs and subsystems that are today part of 4BSD and/or System V.

1. Introduction

Nobody needs to be told that UNIX is suddenly popular today. In this article we will show you a little of where UNIX was yesterday and has been over the past decade. And, without meaning in the least to minimise the incredible contributions of Ken Thompson and Dennis Ritchie, we will bring to light many of the others who worked on early UNIX versions, and try to show where some of the key ideas came from, and how they got into the UNIX of today.

Our title says we are talking about UNIX evolution. Evolution means different things to different people. We use the term loosely, to describe the change over time among the many different UNIX variants in use both inside and outside Bell Labs. Ideas, code, and useful programs seem to have made their way back and forth – like mutant genes – among all the many UNIXes living in the phone company over the last decade.

Part One looks at some of the major components of the current UNIX system – the text-formatting tools, the compilers and program development tools, and so on. Most of the work described in Part One took place at ‘‘Research’’, a part of Bell Laboratories (now AT&T Bell Laboratories, then as now ‘‘the Labs’’), and the ancestral home of UNIX. In the next part, we look at some of the myriad versions of UNIX – there are far more than one might suspect. This includes a look at Columbus and USG UNIXes and at Berkeley UNIXes. You’ll begin to get a glimpse inside the history of the UNIXverse.

2. Basic Sources

Since we can’t say *everything* about UNIX in this article, we’ll give some pointers for those who want to read more. Full acknowledgements will be found at the tail end of each installment. But some basic sources must be mentioned.

It is a truism that the final source of information about UNIX is UNIX itself. And this, of course, requires that you have a source license. And to get a source license, you must sign in blood that you will not divulge the source code of UNIX in any way, shape or form. So, in preparing this article, we have stayed clear of looking at source code. But there are times when you want to do so, not so much to find out

how some feature evolved as to see how it really works.

The UNIX Manuals are a prime source of information about UNIX. Now it's trendy to deride these manuals, but for their intended purpose and audience they are for the most part examples of good technical writing. The *Programmer's Manual* or *User's Manual* as it is variously called, more colloquially known as ''Volume 1'', summarises in a standard format each command, system call, library function, and many special files (in the technical sense!), system file formats, games, miscellany, and maintenance information. Comparing a series of manuals of different vintages offers the student of UNIX evolution a good view on changing conditions.

Volume Two of the Manual set (beginning with PWB and V7; before that it all fit in one binder) is a series of short papers. These range from notes on installing the system to reference manuals on compilers to introductory tutorials. These, too, are typically well-written but occasionally incomplete. They are concise and to the point; some people find this obscure. Remember the audience and the background. The papers are written for the benefit of someone with the source code and with some knowledge of the system. It was always assumed that you would have somebody around to help you – a wizard. Or you would go to the conferences and ask others about problems. A careful reading of the manuals was (and is) required to become a wizard, along with hands-on time spent *using* (and eventually modifying) the system, learning by doing.

These papers, and others written at Research, established an interesting tradition, so counter to mainstream computerdom: *You write the program, you write the documentation.* In almost every case, the authors of the program are the authors of the paper describing its details. And in almost every case, acknowledgement is made to those who contributed significant ideas, advice or moral support to the project. This, of course, has made our work in this paper easier. It also speaks volumes about management and about programmers – both those programmers who write effective summaries of their programs, and those who don't condescend to do so.

And the UNIX manuals are sometimes derided for the ''BUGS'' section. This is the place where the author(s) of a program list its design limitations. One UNIX critic said of this policy: ''If they know about the bugs, why don't they fix them?'' The point is that the early UNIX authors established the beneficial habit of documenting limits to the program, rather than always letting the end user find them. Dennis Ritchie comments: ''Every other manual has bugs sections; they just aren't published.'' Many of the BUGS sections were intended as pointers for further development of the programs, rather than as warnings to the user. Ritchie adds: ''Our habit of trying to document bugs and limitations visibly was enormously useful to the system. As we put out each edition, the presence of these sections shamed us into fixing innumerable things rather than exhibiting them in public. I remember clearly adding or editing many of these sections, then saying to myself "I can't write this," and fixing the code instead.'' [Ritchie, personal correspondence]

After the manuals, another important series of papers in a similar vein appeared in the *Bell System Technical Journal*, July-August 1978. This special issue - part 2 of the July-August 1978 issue - is often referred to as ''the blue book'' for its blue binding. (In reality all issues of the BSTJ from this time period are blue, so the handle is a bit misleading.) The magazine is now called *Bell Labs Technical Journal* and did another special issue on UNIX in October, 1984. This issue is must reading for the serious UNIXophile.

Many of the technical reports from Research are published as Computer Science Technical Reports (CSTRs); those still in print are available from AT&T Bell Labs.

Brian Kernighan has co-written several books containing interesting historical details. We quote later from *Software Tools*, a book he wrote with P. J. Plauger, and *The UNIX Programming Environment* written with Rob Pike.

Finally, access to a nearly-complete collection of back issues of *;login:*, the journal of the USENIX Association, has been invaluable.

The only way to keep abreast of ongoing development work in the UNIX community is to attend, or at least read the proceedings of, the USENIX meetings held twice a year. Everyone doing serious technical work presents it here. Other conferences are more introductory or marketing-oriented.

3. Text Processing Tools

One of the guiding lights of the UNIX utilities or software tools has been the deeply-felt conviction that text should be stored in as simple, as general a format as possible so that any program can easily process it. This idea (it seems to have been present from the beginning) has had the widest impact possible on UNIX in all its varieties. However, there has been a regrettable tendency to move away from it in recent times, especially among commercial software developers.

We have rather arbitrarily divided the software tools into text processing tools and program development tools. Remember that UNIX makes no distinction between text files, program files and data files. Many of the same techniques can be applied to all three. But more on this later. First, an outline of the major tools and their development.

3.1. An old editor made new

The standard UNIX text editor *ed* has a lineage longer than many of us do. As early as 1969, the first assembly-language version of *ed* was in place. Although later rewritten in C, the editor is fundamentally the same program as used then. As Kernighan and Plauger wrote in 1976,

> The earliest traceable version of the editor presented here is TECO, written for the first PDP-1 timesharing system at MIT. It was subsequently implemented on the SDS-940 as the ‘‘quick editor’’ QED by L. P. Deutsch and B. W. Lampson; see ‘‘An online editor,’’, *CACM* December, 1967. K. L. Thompson adapted QED for CTSS on the IBM 7090 at MIT, and later D. M. Ritchie wrote a version for the GE-635 (now HIS-6070) at Bell Labs.
>
> The latest version is *ed*, a simplified form of QED for the PDP-11, written by Ritchie and Thompson. Our editor closely resembles *ed*, at least in outward appearance. [Software Tools, page 217]

This is not to say that *ed* is the same as the TECO found on today’s DEC computers. For one thing, TECO is character-oriented while *ed* is line-oriented. It seems rather a case of ‘‘common ancestry.’’

During the 1970’s, the editor went through countless revisions. Nearly every university had its own modified versions of *ed* and *qed*; some had several modified versions. Jay Michlin of Bell Labs wrote (in IBM assembler) a QED for IBM’s mainframe TSO; this was released to Universities in the mid-70’s. This was, in fact, one of my (Darwin) earlier exposures to the UNIX philosophy; around 1975, I heard about a ‘‘spiffy new editor’’ for TSO, so ordered and installed it on the TSO system at the University of Toronto.

Did this wide variety of editor versions lead to massive confusion? Not really. For although most of the editors added new commands and features, they seldom deleted them. The result was that you could – and this is still true – learn a basic set of *ed* commands and special characters usable on every version.

Today the Seventh Edition, 4.?BSD and System III/V versions of *ed* are all sufficiently similar that one can move freely amongst them with only minor inconvenience. (Berkeley UNIX includes both the standard *ed*, and a different editor called *ex* and *edit*. This editor, which has common code with *vi*, has similarities to the standard editor, but is not close enough that one can freely move between it and normal UNIX *ed*.) The manual pages for every current version of *ed* are all recognizably derived from the Sixth Edition document. System III/V extends the ‘u’ (undo) command, but most of the other commands are constant. If you’ve used *ed*, you’ve used an editor with a long history, and probably a long future.

3.2. roff

Having a good text editor is only half the text processing battle. Having entered your text, you still must format it neatly for presentation. That’s the function of a text-formatting program. The earliest UNIX formatter known to man is *roff*, a line-command formatter. Like *ed*, *roff* is part of a large and diverse family, one that includes the *runoff* package found on Digital Equipment computers (the latest release is called DSR, for DEC Standard Runoff). The earliest Runoff program is attributed by Kernighan and Plauger to J. Saltzer, who wrote it for CTSS. Runoff also is an ancestor of the Script programs available on IBM mainframe systems; that descent would be equally interesting for IBMers to trace (no doubt we’ll get letters from those with information to SHARE with us).

Roff was written by M. D. McIlroy, at Research. Like *ed*, *roff* was well in place by the First Edition of UNIX. It was considered static by the time of the Sixth Edition, and obsolescent by the time of the

Seventh, and dropped altogether by the time of System III.

3.3. nroff and troff – The assembler of text

Computerists are never satisfied. So after *roff* came ‘‘New Roff’’, or *nroff*, written by the late Joseph Ossanna, who throughout his career was concerned with improving the way text was handled. Ossanna’s *nroff*, as Kernighan and Pike relate,

> ‘‘was much more ambitious [than *roff*]. Rather than trying to provide every style of document that users might ever want, Ossanna made *nroff* programmable, so that many formatting tasks were handled by programming in the *nroff* language.
>
> ‘‘When a small typesetter was acquired in 1973, *nroff* was extended to handle the multiple sizes and fonts and the richer character set that the typesetter provided. The new program was called *troff* (which by analogy to ‘‘en-roff’’ is pronounced ‘‘tee-roff’’). *nroff* and *troff* are basically the same program...’’ with divergent processing appropriate to the differences in output device. [UNIX Programming Environment page 289]

They point out that *troff* is tremendously flexible, and indeed many computer books have been typeset using it. But it can be complex to use. As a result, most everyone uses one or another ‘‘macro package’’ – a series of pre-programmed commands – and optionally one of the preprocessors (such as *eqn*, *tbl*, *refer*, and more recently *pic*, *grap* and *ideal*). *Troff* was originally written in assembler, but was redone in C in 1975. Joseph Ossanna wrote both versions and maintained it until his death in 1977.

3.4. Macro Packages

The earliest macro package to come into wide use was ‘‘ms’’, for ‘‘manuscript’’. Written by Mike Lesk, the ms macros provide a powerful but easy-to-learn (by comparison with bare *nroff*) approach to document formatting. The ms macros were distributed with the Sixth and Seventh Edition UNIX and most subsequent releases. The package was picked up and extended at Berkeley.

The USG versions of UNIX include a macro package called ‘mm’, for ‘‘memorandum macros’’. These do most of the same things as ms, in slightly different ways, with the addition of numbered lists and a few other bells and whistles, but are about half again as big as ms. The startup time is such with mm that USG in 1979 had to resort to a compacted form of the macro packages; this made it into System III.

There are two versions of the ‘man’ macro package used to format the manual pages in Volume 1 of the UNIX Manual Set. One was used on V6, and the other from V7 on. If you see a manual page beginning with ‘.th’ instead of ‘.TH’, it’s from V6. System III has an (undocumented) command *mancvt*, and 4.1BSD has *trman*, to convert files from the old to the new format.

There is also the ‘mv’ macros for producing viewgraph or slide presentations. This is a USG product, and versions of the USG manuals from PWB 1 up to just before System III carried the now-famous line:

> The PWB/UNIX document entitled *PWB/UNIX View Graph and Slide Macros* is not yet available. *Viewgraph Macros* is in preparation.

System III manuals appeared with scarcely a mention of mv, and (finally) it was documented with the System V manuals.

3.5. tbl, eqn

One view of *troff* is as an assembler language for text processing. If this be true, then *eqn* and *tbl* are the high-level compilers that go with it.

Mathematics has always been an inconvenience to traditional typesetting. This observation led Brian Kernighan and Lorinda Cherry to develop *eqn* for UNIX, and would later lead Donald Knuth to write his TeX typesetting package with math capabilities built in.

The *eqn* program reads an entire *nroff/troff* input file and passes it unchanged except for ‘‘equation specifications’’ delimited by .EQ and .EN requests. Material inside these requests is used to construct equations of considerable complexity from simple input. English words such as ‘sum’, ‘x sub i’, and

'infinity' produce the expected results (a large Sigma, x with a subscript i, and the infinity symbol respectively). In most cases no typesetter wizardry is required. A list of some forty extra character definitions lives in the file */usr/pub/eqnchar*; these can be copied, extended, or altered by the knowledgeable user.

Eqn was written by Brian Kernighan and Lorinda Cherry. The ''new graphic symbols'' in */usr/pub/eqnchar* are the work of Carmela L'Hommedieu (formerly Scrocca) at Bell Labs. The first public write-up of *eqn* appears to be a paper by Kernighan and Cherry in the *CACM*, March 1975. The software was included in the Sixth Edition UNIX.

Like *eqn*, *tbl* is a preprocessor that passes over a formatter input file looking for special requests (here .TS and .TE). The material between these requests is expected to be a series of special commands to *tbl* and some tabular data. To greatly over-simplify how this program works, *tbl* replaces tab characters with explicit horizontal and vertical moves to make the rows and columns in the table align exactly under control of the table specification. It is invaluable for putting tabular material of any kind into documents.

Mike Lesk wrote *tbl* at Research; the idea for it came from an earlier table formatting program by J. F. Gimpel. *Tbl* first appeared outside the Labs with the V6 release of UNIX. It appeared in its present form on the ''Phototypesetter Version 7'' (interim V7) and PWB tapes, in Seventh Edition UNIX distributions, and in all systems since then.

Lesk also wrote *refer*, a bibliographic citation and reference package, which first appeared in V7.

3.6. Typesetter Independent troff

New! Improved! Yet again! That's right. *troff* is infinitely perfectible. In 1979, Brian Kernighan at Research set out to re-write *troff*. Rather than rewrite it completely and be incompatible with the tens or hundreds of thousands of documents in existence, he chose to ''clean up'' *troff*. It turned out to be rather more akin to cleaning the Augean Stables than he had imagined, but resolve did not desert Kernighan. Finally he emerged with a tape for the Device Independent *troff*, along with revised *tbl/eqn* and two new preprocessors, *pic* and Chris Van Wyk's *ideal*. *pic* (as the name implies) does pictures. It is useful for drawing ''flow-chart''-like drawings, but there is much more to it than that. *Ideal* also draws pictures, but is somewhat more mathematical in usage than is *pic*.

This set of products forms the basis of the commercialised ''Documentor's Workbench'' package from AT&T. And work continues, of course. Kernighan has been working on cleaning up the appearance of *eqn* output. Recently, Kernighan and John Bentley have written *grap*, a graph plotting preprocessor for *pic*. The program was released with Release 2 of Documentor's WorkBench, released in early 1986. A report on *grap* is also available from AT&T Bell Laboratories as a CSTR.

3.7. Of Mice and Blits

Tired of typing at a dull, boring 24×80 screen, Rob Pike and Bart Locanthi had a better idea. Integrating the ideas of the Alto project and related work at Xerox PARC (Palo Alto Research Centre) with the UNIX approach to things, they built a special terminal called the Blit (not an acronym) with a 68000 processor, high-resolution screen, good keyboard, and a mouse, **and** software shared between the host and the terminal. Their particular combination of these ingredients makes possible a form of interaction with the computer that is not yet understood by 99% of the people working in the computer field. Pike, in addition to being a radical advocate of the UNIX approach to software development, is quite visionary. Some of his work has been described at recent USENIX conferences. And most of the audience didn't seem to grasp the essentials of what he is saying. In 1969 Steve Munro described to me somebody with what he called ''Unit Record Mentality'', meaning somebody firmly attached to the (even then obsolescent) card punches, readers, and impact printers. By analogy with this, I could describe those who don't dig Pike's terminal interaction as being possessed of a ''pre-interactive mentality.'' But that would be premature.

True Blits are available only inside Bell Labs. The Blit has been commercialised by AT&T/Teletype, and is sold (with a different processor) as the 5620 terminal.

3.8. Style, Diction, Writer's Workbench

One of us (Darwin) has long been interested in the computerised processing of text, a term I take to mean more than is commonly included as ''word processing.'' So I was quite interested to read a paper by L. E. McMahon, Lorinda L. Cherry and R. Morris entitled ''Statistical Text Processing'' in the 1978 special BSTJ issue on UNIX. I would later use several of the techniques mentioned in the paper.

At the end of the paper, Ms. Cherry describes a program *parts* for finding parts of speech in English text. This was written to be the first pass of a system to add inflection to the *speak* program written by Doug McIlroy, but the person doing the stress part left the company. Ms. Cherry wasn't interested in the stress assignment, so she documented the work done so far and went on to other things.

In the spring of 1979, W. Vesterman of Rutgers approached Doug McIlroy at Research about computerizing one of the techniques Vesterman used in teaching writing. The students had to count surface features in their text and in a sample of text written by a professional writer. That summer, Ms. Cherry expanded *parts* considerably, and added the code that turned it into *style*, a program to analyse the readability and other characteristics of a textual document. She also developed *diction* to check for awkward word uses, overused words, and other problems facing everyone who composes text for others to read. She also modified *deroff* to find the real text in a document. Vesterman consulted on this work.

And when the 4.1BSD release of the system came out, I was pleasantly surprised to see that *style* and *diction* were present. Bell Labs has a policy of sometimes releasing software to educational institutions; this probably explains the release at Berkeley.

While this was going on, the Human Factors group at Piscataway (now at Summit) was getting interested in automating document review, and Nina Macdonald of that group called Ms. Cherry about using *parts*. She had worked at Murray Hill in a Linguistics group and was familiar with the program. Ms. Macdonald took *style* and *diction*, and WWB evolved from there. Writer's Workbench (WWB) consists of *style*, *diction* and a dozen or so related programs for finding problems in written work. The ''chattiness'' level of the programs is set for the beginning user, but can easily be adjusted by the advanced user. The ideas for this work came from the Piscataway group, the Murray Hill group, and from Colorado State University, where extensive use of the Writer's Workbench (described at USENIX, Toronto, July 1983) currently puts several thousand undergraduates on WWB each year. The use of WWB is perceived to improve significantly the students' writing skills.

Many writers will be thankful to all who contributed, because these programs have proven themselves useful many times over. If buying a 4.1 or 4.2BSD system, insist on *style* and *diction*. If you get a System V UNIX, consider getting the WWB add-on if you'll be doing any document preparation. Writer's Workbench is one product that should survive and prosper as UNIX continues to evolve. The next major release of WWB (3.0) is scheduled for the spring of 1985.

4. Compilers, Languages, Tools

What is an operating system without languages and utilities? Despite the limited support for FORTRAN, UNIX has always been known for the diversity of languages and tools provided. Some of these are well-known, others are less-well-known than perhaps they ought to be.

4.1. The C Programming Language

The early evolution of the C language has been described elsewhere (see Dennis Ritchie's paper in *Microsystems*, October, 1983. Dennis is rather modest, and doesn't tell you that the UNIX world has named the C compiler described there ''the Ritchie compiler'' to distinguish it from other C translators). As we pick up the threads of the story, Fifth Edition UNIX has been in the field for some time. It is May, 1975, and the new improved Sixth Edition is about to be released. Ritchie has added some support for a new data type, ''long integers'' referred to with the keyword ''long'', but this will not be documented. Not all the runtime support has been installed, and the tape goes out without it. Later Ken Thompson will announce that the support for 'longs', limited though it be, was there all along in V6. There is no support for ''short integers'', or ''shorts''.

There follows a succession of releases of the C compiler. The PWB 1.0 release of UNIX, the first outside the labs of a non-Research UNIX from Bell, goes out in 1977. And a special-release tape known

only as ‘‘Phototypesetter Version 7’’ includes a new release of *troff* as well as the C compiler, assembler, loader, archiver and bits of the C library, including the first release of ‘stdio’. These compilers seem to be from about the same vintage. Both compilers support another new data type modifier, ‘‘unsigned’’, which causes all bits of an integer to be treated as magnitude (on the PDP-11, for example, signed ints run from -32768 to +32767 while unsigned ints are from 0 to 65535). These compilers add typedefs, which allow you to generate your own names for existing data types, for a degree of independence from the machine data types. One of these compilers is somewhat buggy – the concept of ‘‘cast’’ is in the code but doesn’t work properly. Bit fields exist but are buggy; this is documented. The Phototypesetter Version 7 was primarily a release of *troff*; the compiler was included because it was necessary for *troff* (very convenient, since Research wanted to get the latest C out into the field anyway).

Finally the Seventh Edition of UNIX is released. Of course, it has another C compiler. This one, for the most part, is a ‘‘shaken down’’ version of the ‘‘Phototypesetter C’’ compiler. It is a lot more solid, although bit fields are still broken and now the bug is not documented. And there is a special kludge for *uucp* whereby casting an expression involving a character pointer to type unsigned treats the character referenced by the pointer as an unsigned character, a concept not yet in the compiler or language. This will be quietly withdrawn later. The compiler has a bug in that it treats the right, not the left, side of an assignment as the value of the assignment expression. The V7 *stdio* exploits these two bugs, thereby making it nonportable. The semantics of casts will remain unsettled until slightly after V7.

Along with the Ritchie C compiler, V7 of UNIX includes the first release outside Bell Labs of a second C compiler, bearing the impressive name of the ‘‘Portable C compiler.’’ Written by S. C. Johnson, this compiler has been in development since 1975 and uses the program development tools *yacc* but not *lex*. (A part of the *yacc* grammar for this compiler was published in the C manual with PWB in 1977.)

The portable C compiler turns out to be not as portable as desired, so a second version is developed over the next few years, called *pcc2*. At the ACM National Conference in 1983, Steve Johnson describes *pcc2* in some detail, and shows an example of its portability. Of the many ‘‘back ends’’ for it, one compiles a C language algorithm into the commands necessary to drive a VLSI fabrication process. So your program (if you work in the right part of the Labs!) can be compiled into a custom microprocessor, optimised to execute your program and nothing else! That sure out-classes the EPROM versions of Intel and Motorola microprocessors. *pcc2* has only recently been released; it is the C compiler for the Software Generation System.

One immediate beneficiary of the two-pass nature of *pcc* was the Fortran compiler, to which we will return shortly. But a second major fallout from *pcc* is a program called *lint*, which does partial compilation of C programs with much greater error checking. Like *pcc*, *lint* first appears with V7. We continue to recommend the use of *lint* to provide some reassurance of program correctness and portability.

Berkeley has taken the C language in some new directions. They have relaxed some restrictions on compiled programs. Most notably, variables can be almost any length and need not be unique in the first seven or eight characters. While this sounds handy, it is a major annoyance to the rest of the world, which has to change programs written with such ‘features’ in order even to compile them. Berkeley programmers also tend to rely to an unprecedented extent on the ‘asm’ keyword, which allows you to interpolate assembler language code into the middle of the C program. ‘asm’ buys an increase in micro-efficiency but with a tremendous loss of portability. To preserve portability, the programmer should use **#ifdef** to include in the source code both an assembler version and a portable C version. But the latter is often omitted. A fine example was shown in a talk by Mike Tilson of Toronto’s Human Computing Resources at the San Diego USENIX Conference in January, 1983 (reprinted in the November, 1984 final issue of *Microsystems*). Here’s the code:

```
to = bp->b_ptr;
asm("movc3 r8,(r11),(r7)");
bp->b_ptr += put;
```

What it does is left as an exercise to the reader. The writer of this code left no clues to how his mayhem works. As Mike says: ‘‘The variable ‘‘to’’ is one of the registers used in this VAX assembly instruction. You guess which.’’ Oh, we almost forgot. The three lines above are Copyright © 1980 by the Regents of the University of California.

Meanwhile, back at Research, B. Stroustrup has been busy adding ‘‘classes’’ to C. Classes (nothing to do with going back to school) are the interesting part of Simula 67. They provide for orderly interchange of data between modules, with no possibility of hidden dependencies. A class consists of data (normally inaccessible from outside the class) and functions which are normally accessible from outside but which may be declared as inaccessible. One typically defines a class and publishes the names of the accessible functions. Functions outside the class cannot reference the data within that class except by calling the class’s publicly accessible functions. This enforces modularity by hiding the details of a particular class’s internals from other routines. Classes can be nested, of course, so you can develop such things as queues and stacks of objects. The C compiler encompassing all recent developments, including classes, declaration of function parameters for type checking, and other recent developments is given the name ‘‘C++’’. C programmers will recognize the pun; for others, it simply means ‘‘an incremented (augmented?) form of the C language, which retains the value of the old language’’. C++ has been in use for some time within the Labs, and is now available to the public from AT&T.

In addition to the compiler, one needs a series of library routines to do Input/Output and some ‘extra-linguistic’ operations such as setexit() in V6 or longjmp() in V7. The first ‘portable C library’ was written by Mike Lesk, and was implemented on the PDP-11, the IBM 370, and the Honeywell 6000 with the GCOS operating system. It set the style for subsequent development, and in Version 7 there was a ‘‘new portable I/O library’’ written by Ritchie. This has become known as ‘‘stdio’’ (pronounced ‘‘stuh-DYE-oh’’) for the name of its header file. This is the I/O library distributed with all real UNIXes today.

The current C compilers for the PDP-11 continue to derive from the Ritchie versions. *Pcc* for the PDP-11 never worked as well as the Ritchie compiler. Most other machines use *pcc*-based C translators since the Ritchie compiler only works for PDP-11’s. Many systems integrators wait earnestly for the release of *pcc2* since porting *pcc* takes a non-trivial amount of work.

The future of the C language is not primarily in the hands of people like Dennis Ritchie and Steve Johnson and B. Stroustrup. Rather, it is in the hands of the ANSI C Language standards committee. But in the final sense, it is in the hands of programmers everywhere. Partly because ANSI is a democratic agency, and any member of the committee has as much voice as a Dennis Ritchie or a Steve Johnson. And also because C is a powerful language, and like all powerful tools it can be used or abused. This is not the place for a tutorial on C style, but the interested reader can refer to the paper by Tilson cited above. Good use of C leads to rapid development of high-level code; poor use of the language leads to code that looks like it was written in assembler. As we have seen, in a few cases it has been.

4.2. fortran

Since UNIX comes from a Computer Science research background, it is perhaps natural that Fortran, that octogenarian, reptilian but ubiquitous language should be the object of some disdain among UNIXophiles. Indeed, the V6 *how to get started* document says that ‘‘no debugger is much help for Fortran.’’ And the Sixth Edition manual set included the *C Reference* and the *C Tutorial*, but nothing on Fortran beyond the manual page for *fc*(1), a compiler for a slight variant of the ANSI Fortran-66 standard.

fc produced executable programs that used threaded code and floating-point instructions heavily; thus it ran slowly on machines without a floating-point processor, on which floating point had to be interpreted by the UNIX kernel.

The prime mover behind the next Fortran compiler was Stuart Feldman, who had been interested in compilers for some time. In 1976 he released a CSTR on ‘‘Fortlex – A General Purpose Lexical Analyzer for Fortran.’’ This program reads a Fortran program and breaks it up into lexical tokens of the appropriate type. *Fortlex* was used in the construction of various Fortran programming aids, such as a program to change all double precision variables, functions and library calls to single precision. The paper also includes the *yacc* grammar for a Fortran scanner to be used with *Fortlex*; not a complete compiler, but possibly a basis for one.

And the Fortran weakness of UNIX was remedied with a vengeance for the Seventh Edition. A compiler for the full ANSI Fortran-77 standard was included, apparently the first implementation of the 1977 standard on any system anywhere, along with a paper detailing its use and implementation. The back-end of this compiler was the same back end as the Portable C Compiler, so that it would be easy to adapt to new

computers.

Although the Fortran compiler is part of all standard UNIX systems, most suppliers of 68000-based UNIX boxes do not include *f77*. Whether this is so they can charge extra for it, or because they couldn't figure out how to port it, is unclear. But commercial implementations are available for most micro-based UNIXes.

4.3. yacc+lex

One of the major tools used in compiler development is the *yacc* (yet another compiler compiler) program by Steve Johnson. When this program was developed in the early 1970's, compiler generators were being generated by many universities and other research institutes. As Kernighan and Pike remark, Johnson's choice of name for his program is ironic in that his has endured while most of the others have now been retired.

yacc reads in the specification of a language, and generates a program which parses that language. Note that this is not limited to ''programming languages'', but can be applied to any input that is structured. Many applications of *yacc* are mentioned in the *yacc* manual. *yacc* is also part of the *nrws* nroff-to-Wordstar program used to translate some articles for *Microsystems*. The *yacc* manual mentions ''compilers for C, APL, Pascal, RATFOR, etc, ..., a phototypesetter system, several desk calculators, a document retrieval system, and a Fortran debugging system'' as programs which have been written in *yacc*. More recently, Cobol and Ada compilers have been constructed outside of the Labs.

But syntax analysis is only one part of compilation. Another part is lexical analysis, or scanning of the input looking for certain kinds of tokens. For this, too, UNIX has an answer. The *lex* program by Mike Lesk and E. Schmidt provides this function. Since it is part of the UNIX tradition, of course, *lex* uses many of the same conventions. In particular, *lex* uses the same notation for ''regular expressions'' as is used in the editors and elsewhere to describe the patterns to be looked for. If you've mastered commands like /[hH]e/ in the editor, you already know how to construct expressions for *lex*. And of course it works with *yacc*. The naming conventions of these two programs are such that the output of both can be loaded together to form a working unit. Indeed many programs consist of *yacc* and *lex* outputs compiled and loaded together.

Yacc was present in Version 6 (the manual page is dated late 1974); *lex* first appears outside the labs in the PWB 1.0 release.

4.4. make

It's hard to imagine UNIX without the *make* utility. *Make* is so taken for granted these days that the distribution of software in source form without a *makefile* is an event worthy of attention and inquiry. But there was a time when the name of the file with the instructions to build a system were chosen at random from the names ''build'', ''rc'', ''run'', ''runfile'' and others. And these were shell files which built the entire component.

Make builds a program or component from individual pieces, and recompiles only the minimum needed to rebuild it as changes are made. The edit-make-debug cycle is well known to all UNIX programmers. Since the topic has been treated in detail in ''The UNIX File'' by one of us, we will not expand on it here. Suffice to say that *make* was written by Stuart Feldman at Research, and first appeared outside Bell in the PWB release of the system. The barbarities of the Source Code Control System and the incompatibility of this with most of UNIX including *make* led not to the correction of SCCS, but to an ''enhanced'' *make* which appears publicly in System III and System V.

4.5. ratfor, efl

Brian Kernighan at Research realized that Fortran would not go away, so he did something about it. He fixed it. He fixed it by adding the control structures of C and the definition and inclusion capabilities of the C preprocessor. The converter which takes in ''rationalised Fortran'' and produces ugly conventional Fortran he called ''Ratfor''. Several versions were written; one in Fortran for bootstrapping onto other systems, another with *yacc* and *lex* as mentioned above. The meaning of Ratfor is best told in the book *Software Tools* co-written with P. J. Plauger. The source code for the programs in the book was made

available on magnetic tape by Addison-Wesley, in a move that was very foresighted for 1976. This lead to the formation of the Software Tools User Group at Lawrence Livermore Labs in Berkeley; this group is still active and co-sponsors meetings with USENIX.

The *Software Tools* book would later be re-done in Pascal (see 'Pascal' section). There are no plans announced for doing a ''Software Tools in C'' book; most of what you need is in Kernighan & Pike's book *The UNIX Programming Environment.*

After the Fortran-77 compiler, Stuart Feldman turned his attention to Fortran extensions, and produced the *efl* language. This combines the control structures of *ratfor* (which in turn derive from C) with the data structuring capabilities of C including the aggregates to group related data items, analogous to Pascal's *record* capability. *efl* is included in System III and some 4.?BSD systems. Some microcomputer ports (i.e., UniSoft) include *efl* even though they don't have a Fortran compiler.

4.6. awk

Aho, Weinberger, and Kernighan. The names of three authors put together in the most pronounceable way. That's what they did when they couldn't think of a more imaginative name for a wonderful program they'd devised. A is Al Aho, of compiler book fame. W is Peter Weinberger of Research. And K is Brian Kernighan, just described for his work on Ratfor. (Kernighan and Pike's book remarks that ''Naming a language after its authors also shows a certain poverty of imagination.'' [page 131]) *Awk* is not at all awk-ward, it is a great simplification. You can think of it as the combination of most of the best ideas of the other tools all rolled into one. We use it all the time. For some examples, see the review of ''Leverage'' in the August 1984 issue. You can enter the *awk* commands from the command line if they are simple enough, so that

```
awk -F: '{print $1}' /etc/passwd
```

is a complete program to print the names of all the accounts shown in */etc/passwd*, the standard place for the names of all accounts on the system. By all means learn about *awk* and use it. It will make life far less awk-ward.

Awk was first described in *Software Practice and Experience* in July, 1978, and first distributed with Seventh Edition UNIX.

4.7. Pascal

Pascal did not catch on at Research. In 1981, Brian Kernighan did a paper published as a CSTR entitled ''Why Pascal is not my Favorite Programming Language.'' The note was not based solely on introspection, for he and P. J. Plauger had just converted their book *Software Tools* into *Software Tools in Pascal*, including re-coding all the programs in Pascal. In the process they came to regard Pascal as their not favourite language.

Berkeley UNIX has included Pascal for a long time. Ken Thompson wrote the first version of Berkeley Pascal at Berkeley while working there as Visiting Mackay Lacturer in Computer Science in 1975/76. He spent the academic year at UC Berkeley, and taught several courses in Computer Science. He recalls:

> ''When I arrived, the CS department shared an 11/45 with Statistics. It was 50-50 UNIX and RSTS. The first advance was an 11/70 dedicated to teaching. I put my first 155 [operating systems] course on it. Between the first and second quarters I wrote the Berkeley Pascal and talked Bob Fabry into using it on his 153 [data structures] class. It has been used for that ever since. By the time I left, there were several (2 or 3) 11/70s in the computing center providing UNIX service. CS had the 11/70 for teaching, they had almost completely taken over the stat 11/45 and there was a research 11/40'' in an AI lab [Thompson, personal correspondence].

Pascal compilers can be had for most 68000-based UNIX boxes. These are available from commercial software firms and OEMs – see the annual UNIX software directory in the April Microsystems.

4.8. S

Finally, we cannot overlook an interesting ‘‘application’’ language from Research. *S: An Interactive Environment for Data Analysis and Graphics.* The title of the 1984 book by Richard A. Becker and John M. Chambers puts it succinctly. And we put it as follows: The S package is to conventional mainframe statistics packages as the UNIX shell is to batch Job Control languages. It provides interactive, exploratory statistics, interactive and offline graphics, and data modelling and time series manipulation.

The S language was developed at Research. The first public release was in 1981, which was for V7 and 32V. There were several interim releases; the next major release was in early 1984. This release was accompanied by a change in licensing and an order-of-magnitude cost increase for non-educational users, as part of the swing to the commercialization of UNIX by AT&T Technologies.

The only remotely similar products that I know of in all of computerdom are APL and Speakeasy. APL was first implemented on IBM systems; at least one version for UNIX was developed. Speakeasy similarly arose on IBM hardware; a subset version called SpeakeC was developed at Purdue. Speakeasy was developed some time before S, but in quite different circles of influence. There appears to be no cross-pollination between the two although many of the ideas are similar. S uses *yacc* to interpret its grammar; the *yacc* specification appeared in a *CACM* paper in 1984.

5. Interlude

The Computer Science Research Group, Centre 127, at Bell Labs has had impact on computerdom far out of proportion to the number of people working there. A small group of talented people, started in motion by Ken Thompson and Dennis Ritchie with the original design of UNIX and Rudd Canaday’s file system design, aided and abetted by those mentioned here and others, developed Research UNIX and its related tools. Many reasons are given for the success of UNIX, but one we’d like to add is the consistency of the system in all its facets. As a single example, the syntax used for pattern matching (a notation for the abstract concept of Regular Expressions) allows you easily to develop tremendous skill in pattern matching. This skill once learned can be applied in the editor to find text, in *awk* to find records to be acted on, in *lex* to specify partitioning of an input, (with simple modification) in shell commands to match filenames, and in a dozen or so contexts. This kind of consistency is a rare treat in any computer system; the extent to which it permeates UNIX is exemplary.

This concludes the first installment of our history of the natural creation of the UNIX timesharing system. The next installment will cover the relationships among many, many different versions of UNIX, most of them never released or publicised outside of Bell Labs and the telephone companies. There will be a ‘‘family tree’’ diagram illustrating the descent of UNIX. We hope you enjoyed this edition, and that you’ll be looking forward to the next.

Acknowledgements

A myriad of UNIX experts helped with this paper. While it’s not possible to thank everyone who offered to help, we should single out for special thanks Dennis Ritchie, who was there from the start and helped us to find it. We also got special help from Ken Thompson, who answered many questions about the past; Henry Spencer; and Laura Creighton for comments on the work in progress. Lorinda Cherry and Nina Macdonald were helpful by describing the evolution of Style, Diction and the Writer’s Workbench. We as authors have the final responsibility for the accuracy of the material presented here.