

Interview with Brian Kernighan

MSM: Since we were talking about translations, and since you have watched this book be translated, I suspect you're one of the people that translators come back to when C goes into another setting-let's say into another language. It struck me, the more I converse with people-I want to get to this-that Unix is as much an ethos as it is a programming system. It's a way of thinking about computing. Have people had trouble with that aspect of the translation of Unix? What has been the response when people have tried to adopt Unix into other environments?

Kernighan: I hate to give a non-answer to the question, but, in fact, I have had relatively little feedback from any of the translators, with about three exceptions, I think. The guy who has translated several books into German really does program for a living. He writes; although he's a professor in the university, he seems to enjoy writing really grimy books on how to do things, and so he and I have technical arguments about things at one time or another. I don't think he has any trouble translating from English into German, or vice versa. He does his own translation of German words into English. So, there's a non-barrier there, if you like.

There are two people that I know, not as well, who have done the translations into Japanese of different books. The guy who's done them into-particularly the Unix book and a C book, is-runs the computer center at the University of Tokyo. Although he probably doesn't program on a day-to-day basis, I think he genuinely understands what's going on. My guess is that he doesn't have a heck of a lot of problems translating. Although one Japanese lifestyle, culture, or whatever, is enough different in some areas, that maybe it makes a difference. My bet is that it doesn't make a heck of a lot of difference. I suspect the computing aspects are pretty close to the same there, as they are here.

MSM: Let's take the English-actually, let me rephrase the question, because I had in mind more of an aspect of the portability of C. You wrote the books, and I suspect-maybe the premise is wrong-that people or institutions have tried to go over to you to adopt Unix; that you're one of the people to whom they come back and say, "I don't quite understand how this works." The queries came to you, or have come to you. Am I right on that premise? Have there been a sort of set of difficulties people have encountered when they tried to go over from one system into Unix? Has Unix proved difficult to port as a way of doing things, as opposed to simply a piece of software?

Kernighan: I think in a way it has, although memory grows dim. A lot of this stuff is something that I don't think about much anymore. Fortunately, I'm not involved in it as much. But I think there has always been a problem when somebody new comes to the system. There's a set of shared, I don't know, conventions; things that, because they work the same throughout all the Unix operating system and its programs, tend not to get written down. They're just like folklore; they're customs. Like keeping to the right as you drive almost. Everybody knows that kind of thing, and therefore, it tends not to be written down as part of every single description. It's in one place and then forgotten. People who come new to this system don't know that that sort of particular behavior is pervasive, and therefore, they are perhaps befuddled by it; they don't expect that it will work. And you could see lots and lots of examples of it. Oh, fairly narrow technical issues, but for example, the fact that most programs read from a list of files, or there are no files from the standard input. And I still remember seeing somebody who had taken one of my programs-*ratfor*, the FORTRAN pre-processor-reproduced the manual page, and had added a bug section which says that if you just type *ratfor* nothing happens, not realizing that if you type *ratfor* it's waiting to read the standard input. Okay, so there's an example of somebody who just had missed totally an important aspect.

An analogous one which is probably more modern: If you look at DOS, MS-DOS, the notion of wildcard characters for filename expansion, is part of DOS, but in a very strange way. It's really not done by the command interpreter, COMMAND.COM or whatever, the way it would be in Unix, by a shell. But, rather, it's a service provided by individual programs, and that means that its properties-

Interview with Brian Kernighan

some programs don't have it, its properties might be somewhat different from one program to another, whereas in Unix it's known that it's done by a single program, in a single place, and therefore its properties are somewhat more uniform. That notion that there's the right place to do things is something that you maybe take for granted in Unix; it doesn't quite come up the same in other systems. And so explaining that to people, maybe is difficult.

Another area that I remember endlessly, when we were doing in what I will best describe as popularizing Unix, back in the mid and late '70's: People would come in and they'd say, "Yeah, this is nice, but does the system do X?" for some X, and the standard answer for all of this was, "No, but it's easy to make it do it." Unix has, I think for many years, had a reputation as being difficult to learn and incomplete. Difficult to learn means that the set of shared conventions, and things that are assumed about the way it works, and the basic mechanisms, are just different from what they are in other systems. Incomplete means, because it was meant as a program development environment, it doesn't have all the finished products necessarily. But, as a program development environment, it's very easy to build a lot of these things. It's sort of like a kit. And if you want a new thing, you can take the pieces out of the kit and assemble them to make your new thing, rather more rapidly than you would be able to do the same thing in some other kind of environment. So, we used to say that. "Does it do X?" "No, but it's real easy. Do you want one by tomorrow? I'll give you one by tomorrow." Every once in awhile, someone would call your bluff. (Laughing) Sometimes that was fun, and sometimes it wasn't.

MSM: In what you just described, you used the term "kit," which could be expanded to "toolkit." And existing notions of tools, and Unix as a set of tools, where did that image come from? Doug credits you for it in the collection of research manuals.

Kernighan: I honestly don't know where it came from. It was in the air in several different ways. I think maybe Plauger and I were the people who put a name on it, and maybe some coherence, when we wrote this off our tools book, and tried to take ideas out of Unix and put them somewhere else. But I think the ideas had been kind of floating around anyway. Doug, of course, had published these two papers, in the-I don't even know where they were papers in, the Garmisch, and wherever else software engineering conferences in the late '60s, '68 or '69, that kind of time-where he was talking about things; not quite tools, but components. If you're going build components, you need tools, so that's part of it there. I have a vague notion that Bob Morris was talking about those kinds of things too, because I remember him giving us a fairly coherent talk at one of the spring or fall joint computer conferences fairly early. Beyond that, I don't know. I think that I started giving talks about tools, and combining programs, and so on, fairly early. '73, '74 more likely, that kind of time. And then Plauger and I started to work on the software tools book. And it was, well, what are we talking about? These are the tools you use when you write programs. And that came out in early '76. So, presumably, we were working on that in '74 and '75. I might conceivably take credit for the name, but probably not the real intellectual idea underneath it anyway.

MSM: Is this something you learned when you got here?

Kernighan: I think probably yes. I had spent the summer of 1966 working at MIT, in the group that was the MIT component of the Multics effort. I spent a summer there, working ostensibly for Corby, but practically for-I don't even know for who, officially, but, you know-I was there and I was working on something, and in fact, it was a tool that took stuff from the CTSS machine, the 7090, and made a coherent tape which you could then load on the 645, which had just arrived, and actually run it as a sort of Multics-y kind of job, although Multics didn't really do anything. I've forgotten, it was the merge editor or something; it was basically just, take a bunch of stuff from here and put it over there. So, I got used to both CTSS as an environment and some of the noise that was in the air about Multics, although I didn't actually have much to do with the pure Multics side of it.

Interview with Brian Kernighan

Then I came back and spent the summer of '67 and '68 here at Murray Hill, first summer working for Doug, and I built myself-typical summer job, I think-Doug suggested an idea and I got sidetracked into the tools that were underneath the idea, or might have been helpful in building the idea, and I never went off and did anything. He wanted to investigate storage allocation algorithms, and I started to build myself some list processing stuff and never finished that. That was much more fun than (laughing) trying to figure out the properties of storage allocation algorithms. I think that beyond that, the notion of tools, or languages, or anything like that, did not show up in my consciousness until noticeably further on, probably when Unix was actually running on the PD11, which would be '71, '72, that kind of time. And even there not really.

MSM: Before or after Doug did pipes? Was pipes a trigger for this notion?

Kernighan: I think it probably was the capstone or whatever. I'm not sure what the right image is, but it's the thing that makes it all work, in some sense. It's not that you couldn't do those kind things, because I had already written redirection; it predates pipes by a noticeable amount. Not a tremendous amount, but it definitely predates it. That's an oldish idea. That's enough to do most of the things that you currently do with pipes; it's just not notationally anywhere near so convenient. I mean, it's sort of loosely analogous to working with Roman numerals instead of Arabic numerals. It's not that you can't do arithmetic, it's just a bitch. Much more difficult, perhaps, and therefore mentally not-more constraining. But all that stuff is sort of now squashed into such a narrow interval that I don't even know when it happened.

I remember the preposterous syntax, that ">>" or whatever syntax, that somebody came up with, and then all of sudden there was the vertical bar, and just (snaps fingers) everything clicked at that point. That was the time, then, I could start to make up these really neat examples that would show things like doing, you know, running *who*, and collecting the output in a file, and then word counting the file to say how many users there were, and then saying, "Look how much easier it is with the word count. With the *who* into the word count, and running *who* into *grep*," and starting to show combinations that were things that were never thought of, and yet they were so easy that you could just compose them at the keyboard and get them right every time. That's, I think, when we started to think, probably consciously, about tools, because then you could compose the things together if you had made them so that they actually worked together. And that's when people went back and consciously put into programs the idea that they read from a list of files, but if there were no files they read from the standard input, so that they could be used in pipelines. People went back and did that consciously in programs, like *sort*. *Sort*-an example of a program that cannot work in a pipeline, because all the input has to be read before any output comes out-it doesn't matter, because you're going to use it in a pipeline, right? And you don't care whether it piles up there briefly; it's going come out the other end. It's that kind of thing, where we say, "Hey, make them work together. Then they become tools." Somewhere in there, with the pipes, and maybe somewhere the development of *grep*-which Ken did, sort of overnight-the quintessential tool, as I guess Doug refers to it. A thing which, in a different environment probably you don't see it that way. But, in the Unix environment you see it as the basic tool, in some sense.

MSM: There are several directions to go here. One that this brings to mind is, do we take *grep* and *yacc* and get *awk*?(not clear)

Kernighan: As a bit of an oversimplification, yeah. I'm not sure that *yacc* is the right model. I think in fact *sed* is the right model. That-because the patterns are much simpler, and there's less of a sort of-there's a more sequential structure to the processing in *awk* which is much closer to match to way that *sed* works than to *yacc*. *Yacc* has mystical properties; I mean, certainly it's patterns and actions, but the patterns sort of spring into action, as it were, out nowhere. Whereas, it's really obvious what's

Interview with Brian Kernighan

happening in *sed* and in *awk*. So, I think that's a fairer example, or a fairer path of evolution. It's much closer. Because, I've been interested-having seen *sed*, I was kind of interested in the notion of a programmable editor, because there were lots of things people were doing that required text manipulation, and we didn't have on Unix a programmable editor. We had this thing called *qed* which came originally from Peter Deustch, I guess, or somebody like that, at Berkeley-but Ken had made it work on Multics and then I think Dennis made it work on the GE machine that we used. It was a programmable editor, but it was programmable in some formal sense. It was just awful, and yet it was the only thing around that let you manipulate text in a program without writing a hell of a lot of awkward code. So I was interested in programmable editors, things that would let you manipulate text with somewhat the same ease that you can manipulate numbers. I think that that was part of my interest in *awk*.

The other thing is-that I remember as a trigger for me-was a very, very specialized tool that a guy named Mark Rochkind developed. He was in group that was doing, you know, genuine telephone-related stuff, and he had a program that would let you specify basically a sequence of regular expression and message-regular expression and message-and then it would create a program such that, if you pass data through this program, when it's on instance of the regular expression, it would print the message. And we'd use it for data validation. And I thought, what a neat idea. It is a neat idea. It's a really elegant idea. It's a program that creates a program that then goes off and validates data, and you don't have to put all the baggage in; some program creates the baggage for you. The only problem with it was that it was specialized, this one tiny application. And so my contribution to *awk*, if you like, is the notion that you can generalize this. Technology of making regular expressions work, to this day, remains (not clear) and many of the database, and general feeling of data processing that you might get out of (not clear) repeater, I think that holds much of the first-cut implementation. I don't remember a lot of that stuff. Again, it's pretty blurred now. MSM: It's hard for you to remember just-I mean, how did the three of you get together?

Kernighan: Well, we got together in, typically, in my office. We'd argue, and invent something in real time, and that, (laughing) that has been with us ever since. We'd throw stuff in. Peter is a very, very fast implementer. He can build things much faster that I can or AI can; and so we would get some idea and he would typically have it working almost immediately. Then over the years, I would go back and clean it up. Sometimes, we-he implemented things and we ultimately threw them out and probably should have left them in. For example, right at the earliest times he had a version of *awk* that would generate C instead of being an interpreter. That was something that didn't resurface again for almost ten years. He had in it several different regular expression machines, one of which would do one of these very, very fast regular expression searches, if the set of things that you were looking for was just key words, without metacharacters. The basic *yacc-grep* algorithm. He had implemented that, that kind of stuff. Over the years, I had tried three or four different internal mechanisms for the interpreter machine. You try a stack machine and see how that works, and you try quads or triples or something like that, and see how that works. You try interpreting the parse tree and see how that works; that's the one we stabilized on. So, I personally have used *awk*, probably excessively, as a vehicle for experimenting with various kinds of half-baked ideas, both in implementation and also in language design.

MSM: Let me back up to the beginning. You were up at MIT in '66.

Kernighan: Correct.

MSM: And there working with Corbató on moving from the 7090 to the 645. Conscious of Multics, but not part of it?

Interview with Brian Kernighan

Kernighan: In some sense, yeah. I don't think I realized quite what it was all about. I mean, I had some notion that this was interesting. I was using CTSS; that was the first time that I had used a substantial time-sharing system. I had played very, very briefly with Dartmouth BASIC, and you know, it's kind of nice to have the computer talk to you. But CTSS by comparison was very, very polished. You could really do nice things with it, and in addition you were right center of the universe of CTSS; all of the experts were there. It's very like this is the center of Unix expertise. There it's the center of the CTSS expertise, and so they could make it do wondrous things which you would have never found if you weren't nearby. So, I was interested in that aspect of it, and then there were some interesting application programs as well. Like, ELIZA was there; Joe Weizenbaum worked down the hall. Now, he was away, I don't think I've ever met him.

But-so, you know, I'd meet an interesting girl, I'd bring her in to talk to the doctor. (Laughing) But, it was just an incredibly fun place to be. But, the Multics aspect of it, I had relatively little to do with. What I was doing was writing a program that ran on the CTSS side, and it created a output that was usable by somebody else. But I didn't actually care much about the output side of it. It was just a simple data processing problem; let's just collect inputs in some sense or other and put it together in a stylized sort of way. But, it involved list processing, and it was written in MAD, which was kind of a neat language. Much better than FORTRAN, which would have been one of the other choices.

MSM: MAD's the Michigan-

Kernighan: Yeah. Right.

MSM: It was Bruce Arden's, wasn't it?

Kernighan: Yes. Arden, Galler, I don't know who else. That group.

MSM: You came here in '67, '68. Were your projects part of Multics projects?

Kernighan: No. When I came in '67, as I mentioned earlier, Doug had this notion of working on figuring out what were good storage allocation algorithms, for ML, basically, that kind of thing; how do you provide quick access to storage and then release it again. That remains a good problem, to this day, and Doug continues to work on it to this day. But I didn't get turned on to by it. But, what I did was to go off and say, well, if you're going to provide storage allocation-let's say in a FORTRAN program, which at the time FORTRAN was kind of the universal language for most people-it would be nice if you could-I had spent time at MIT working on list processing kinds of things. There was a list processor called MAD Slip, which was basically just a bunch of routines that you call from a MAD program that let you do-you know, create an arbitrary size block of storage and then link it to some other arbitrary block of storage and so on-so you could do list processing in a conventional algorithmic language. And that was kind of neat, and so what I did that summer of '67 here, was sort of take Doug's vague idea of storage allocation and go off and build myself a bunch of routines that would let you, from a FORTRAN program, do list processing. You know, create a block of storage this big, link blocks together, walk along lists and stuff like that. And in a fit of absolutely misguided craziness I spent much of the summer trying make it run absolutely as fast as possible on this specific machine. So a lot of it was assembly language programming. It was-except for a learning experience, I think ultimately a total waste of time. But it was fun.

Then the second summer, I came back, the summer of '68-I spent that summer working genuinely on my-on stuff that would be related to a thesis. I worked with Shen Lin on graph partitioning, which is just unrelated to any of the above. I mean, it did require going out and writing substantial FORTRAN programs and doing experimenting and so on, but it had nothing to do with any of the sort of system-y

Interview with Brian Kernighan

stuff. It was pure combinatorial experimentation. I did that, and then when I got back to Princeton they, in their wisdom, said, "Your money is going to run out in January. Maybe you better write a thesis." I said Ah! (laughing) So, I wrote a thesis.

MSM: What did you write it on?

Kernighan: Graph partitioning.

MSM: Graph partitioning. I was going to look it up, but I didn't get a chance to go to the library.

Kernighan: Anytime you want to hear about graph partitioning, I will be glad to tell you what I know about graph partitioning. It remains a standard problem. I think it's an interesting problem, because it shows up in a variety of guises in real life. In circuit design, which is one of the reasons that people here were kind of interested in it, and it's still part, a fundamental part, of a lot of circuit layout problems. Because, basic idea is to take the components of a circuit and cluster them-in some sense so that you don't have too many wires running from here to way over there; rather, most of the wires are short. But there are constraints on the of clusters, because the things you are clustering together have to fit on something, like a chip, or a substrate, or a circuit board, or whatever. It's a plausible model. Quite a few real problems. I actually worked on that off and on for probably the next four or five years, when I got here, and other combinatorial problems with Shen. But, I gradually drifted out of that stuff, after two or three years.

MSM: When did you come here in '69?

Kernighan: Real early. February probably. I graduated. I escaped from Princeton in January of '69.

MSM: It's the only way to describe getting out of grad school.

Kernighan: Yes. (Laughing) You understand completely. I really enjoyed Princeton as a graduate student. It was in some sense the peak of income and the minimum of responsibilities, and except for a horrific-probably eight to ten months, while I anguished over what the hell am I going do to for a thesis, I can't get out of here without a degree. Other than that, it was idyllic, it really was. Just a lovely place to be.

MSM: Yeah. I kid, but I actually had a good time as a grad student there. I was married at the time, but when we'd come back from living in a one room apartment in Germany, this Butler tract looked absolutely palatial. (laughing) But, it was like the situation my third year. An NSF fund, and I was teaching a couple of sections. I don't think I ever had so much disposable income.

Kernighan: Yep. Exactly. (laughing)

MSM: The rent was forty eight dollars a month.

Kernighan: Yeah. I remember.

MSM: What did you come here to do? Were you hired into the computing research group?

Kernighan: Oh, yeah. The same group that I'd been in all along. It's interesting, I had so much fun here in '67 and '68-just, you know, the people. It was just such a good collection of people, and I've enjoyed it. I never interviewed anyplace else; I never even thought of any other place. I simply said I'd like to work here. And Sam Morgan, in his wisdom, said, "We don't want any Ph.D. dropouts, so you

Interview with Brian Kernighan

have to get your degree finished. But, other than that, sure, we'd love to have you." That was it, and I came early in '69 and the charter, or my instructions, were-as they are for everyone else-non-existent. Do what you want. The hope is that the combination of people around you, doing things that are interesting-and perhaps ultimately relevant, but not instantaneously relevant, I don't even know-but the combination of people around you doing interesting things, and getting their jollies, I think, from having their interesting things affect other people, means that there's this sort of gentle gravitational pull towards doing the same kind of thing yourself. It's clear the reward mechanism ultimately favors those people who have an impact on the local community, impact on the Bell Labs community, impact on AT&T, impact on the scientific community, in some combination.

MSM: So, it's a question of, "Do what you want, but when we come to salary review, performance review, come talk about what the impact of what you've done has been."

Kernighan: Impact is, I think, ultimately the criterion, but there's quite a long view taken, and quite a broad view of what impact is. So, somebody who does purely theoretical work-but whose theoretical work affects the community in some sense, either because other people take it and produce artifacts, or because that person is able to shape a field, or something like that-that's fine. That's good work. That has impact. It doesn't mean that you can see it in a telephone or anything like that; it means that it has had an effect, a positive effect, on something of substance.

MSM: Now, when you arrived, one of the things going here was Multics. Were you that involved in it?

Kernighan: Yeah. I was conscious yes, involved no. I remember watching Ken and Dennis and a couple of others stroking this giant this giant machine that sat probably just around the corner here, or maybe it was one floor down, and sort of being intrigued, but from a distance. I never actually, I think, ran on it or did anything on it. At the time I came, it must have been within a few months of disappearing. I don't remember specifically. But, for all I know, it disappeared between one summer and when I came permanently. I don't know the date of that. But I never had anything to do with that. At the time, I was doing this combinatorial stuff with Shen Lin. When I first came, I continued to work on graph partitioning and, you know, I also got interested in the traveling salesman problem again. He'd been interested for a long time, and I got sucked into that. We were writing FORTRAN code for that, and I don't think there was a FORTRAN compiler on the Multics machine. So, we ran it on the standard GE machine, where there was a perfectly fine FORTRAN compiler and, you know, operators and all the other things that you needed to get the job done. So, for the first couple of years that I was here I spent essentially all my time doing combinatorial kinds of things and running right just vanilla FORTRAN programs.

MSM: So, that's why The Elements of Programming Style is basically about FORTRAN programming?

Kernighan: Well, that's not why it was. It's because FORTRAN was the dominant language, and-it really was the dominant language of scientific computation, and the only other language that you could pick on that had any overlap with it, really, was PL/1, because it had this sort of side of it that was used for scientific computation, and there was a hope that PL/1 would replace FORTRAN for scientific computation, which of course never came about.

MSM: Were you around when Multics was canceled?

Kernighan: I guess so, yeah, as I said.

MSM: Do you get any memories of that?

Interview with Brian Kernighan

Kernighan: I really don't. I just don't know-

MSM: Sense of mood or-

Kernighan: I just don't remember. It's completely gone. You know, I-my guess is that the people who were involved were probably somewhat disappointed, maybe bitterly disappointed. If nothing else, they had invested a lot of time in it. On the other hand, I think the handwriting must have been extremely clear on the wall for quite some while, that this thing was not living up to its promises. The promises made had been totally unrealistic, but it just wasn't living up to anything.

MSM: When did you find out what Thompson and Ritchie were up to?

Kernighan: I don't know. I remember long discussions right across the hall there--which I think at the time that may have been Rudd Canaday's office, I don't remember now--when they would sit and draw pictures on the blackboard, file systems, different kinds of pictures and talk about it. I wasn't paying very much attention to that. I remember some distant--well, I certainly remember the PDP-7 and 9, because of the great graphics display, and you could play Space War on it. It was the best Space War, and still one of the best video games ever invented. But then Ken started to do serious computation, this Unix-related stuff on it, and at some point he put together a system. I think it was Doug who probably first pointed out that this was actually a useful thing, that you could do interesting work on it, but I still didn't actually do anything with it, certainly not while I was on the PDP-7. And I did not--I think that I, in some ways, got involved only peripherally through the B interpreter, which--B was the language that they had used to experiment in one way or another. Steve Johnson made a version of the B interpreter that ran on the Honeywell machine. I started to play around with that, and got interested, and wrote with Steve a sort of introduction to B for people, so that people could write B, presumably on the PDP--whatever--at that point I don't remember whether it moved to 11 or not--but also on the Honeywell. So, I wrote this little tutorial on B, and I wrote a couple of B programs, and I found it, you know, easier. It was like going back into MAD from FORTRAN. It was just easier, because a lot of--it was easier to manipulate characters, textual kinds of things, you didn't have to worry about goddamn labels and continue statements and so on. I mean I still use *goto*'s, but it was just cleaner. You didn't have to worry about card boundaries, all that kind of nonsense. It was just nicer. In some sense, maybe that's how I got into it, through the language side. Then, at some point, the PDP-11 came along. There was an actual useful machine there, and.... I literally don't know what I--how I got in, or what I did first. I must have been pretty early in the game, because I have a single-digit user ID, as do Ken, Dennis, and Doug.

MSM: Low serial number?

Kernighan: Low serial number. (Laughing) Mine is 9. It's the last of them. But, there's got to be some cachet in having a single-digit user ID on the lineal descendant of the original Unix machine.

MSM: Absolutely.

Kernighan: *Ossanna*, I don't know, was 4, maybe, and Robert Morris was 5, and that was it. I mean, Ken was 6, Doug was 7, Dennis was 8, or something. You know, that kind of thing. So, I was 9. So, I must have been in, in some way, fairly early, and I don't know how I achieved that.

MSM: You don't remember what you were doing, what got you on?

Kernighan: I don't. I really don't. It may have been that I said, "Gee, I'd like a login or some way to use this machine," and maybe never did anything with it. That I don't remember.

Interview with Brian Kernighan

MSM: What were you working on at that time?

Kernighan: Probably, still these combinatorial kinds of things, because Shen and I worked on graph partitioning and traveling salesmen, and I worked on a variety of circuit-related things, circuit design related things, with a guy named Dan Shweichert, that were also combinatorial, and so on, in that period of time. Those were predominantly FORTRAN programs running on the GE machine, because that was the general-purpose computing environment.

MSM: What was your first major project on Unix, and when did it start to become part of your research program?

Kernighan: The first substantial thing I can remember was eqn, which Lorinda and I did, and that, I would guess, was '73 or early '74. The initial development was very, very short, but you could probably date it almost exactly. It was written in C, so there had to be a working C compiler, which was presumably put in in '72 or '73. There had to be a working yacc, because, you used the yacc grammar. It was done-in fact, I could find out from this-there was a graduate student named Wayne Hunt who had worked on a system for doing mathematics, but had a very different notion of what it should be. It basically looked like function calls. And so, although it might have worked, he a) didn't finish it, I think, and b) the model probably wasn't right. I remember, he and Lorinda had worked on it, or she had been guiding him, or something like that. I looked at and I thought, "Gee, that seems wrong, there's got to be a better way to say it." I mean, then suddenly I drifted into this notion of, do it the way you say it. I don't know where that came from, although I can speculate. I had spent a fair length of time, maybe a couple of years, when I was a graduate student at Recording for the Blind at Princeton. I read stuff like computing reviews and scattered textbooks of one sort or another, so I was used to at least speaking mathematics out loud. Conceivably, that triggered some kind of neurons. I don't know.

MSM: EQN was the-

Kernighan: That certainly was the first substantive thing that I did that ran on Unix; was purely, specifically, only a Unix program.

MSM: Was that the point at which Unix was being developed on the grounds, on the basis, of being a text processing system?

Kernighan: It must have been well into that, because at that point *troff* existed, we had typesetter-because this stuff wouldn't be interesting if you didn't have a typesetter-so we had a typesetter, and Ossanna had a working version of *troff*. So, all of us had had this interest in text formatting for a long time. When I was at Princeton, I wrote a program to format my thesis, because I couldn't stand the idea of paying somebody a couple of dollars a page to type the stuff, because I knew I was going to do it over and over again. I hated just writing stuff out by hand, and I couldn't use a typewriter very effectively; it was just too tedious. So, I wrote a text formatter at Princeton, which was sufficiently successful that, for a decade later, there was a student *roff* agency.

MSM: I don't remember the student *roff* agency, but I do remember, back in the mid '70s, when I first became conscious there was a course on computers and society that was being taught by a guest brought in from Rutgers because no one at Princeton wanted to teach the course. It has since disappeared as a course; I keep threatening to bring it back. It would actually be a good course; I'd have no trouble getting 1500 students in for a course like that. But, I remember that the students were asked to *roff* their text. I remember about that time, one of my students, the first of my graduate students in the history of science, discovered script. He started using green bar paper, and then never

Interview with Brian Kernighan

did write a proper thesis, because he thought editing meant going in and responding to my margin area on the line, and that's not what I had in mind.

Kernighan: So, I was interested in formatting. I'd done this formatter, Joe Ossanna had done a formatter. I think it was either-it may have Ken, it was probably Ken, but it might have been Dennis-that had done a little formatter that ran on Unix, and that was the genesis of the patent department stuff. But, Ossanna took it over, I think, and made it big. I may be making that up, you'd better check with some of the protagonists. Doug was interested in formatting, and had built quite a sophisticated one for the Honeywell or GE machine-somewhere in there, it became Honeywell. So, there were a lot of us who had our hands in on formatters, and it was one of those topics that everybody was intrigued by. But, the *eqn* thing was the first-something that sat on top of, or in front of, a formatter to genuinely broaden what you could do with them. That was the first thing.

MSM: One of the things that I find quite interesting there, is-I can understand people's being interested in formatters, because you get the idea that you're going to have a system that you can type onto, and using a teletype to do it. You say, "Look, this is a typewriter, if I can get stuff into it, I ought to be able to be able to get stuff out of it, and I ought to be able to get it out in any form. Why not use this as a way of editing? Why not stop this silliness of having to count lines and footnotes as I encounter them, and keep a record of them off to the side?" and so on, and so forth. But, *eqn* and *tbl* and the others go beyond formatting, to typesetting. It's going to get you into the printer's art. I think of Don Knuth, who was in the midst of writing this marvelous bible of programmers, and then, all of a sudden, off he goes and gets into a decade of close studying of the formation of letters. He still hasn't gotten back to Volume 4. There seems to be an allure. (laughing) What is that?

Kernighan: Well, I will say for Knuth, I think it's in a large part, a waste of great mind for him to spend a decade on TeX. It's not that TeX is bad, but it's a waste of Knuth. I mean, he's capable of much more. I think I understand it. It's seductive because-first the output is actually appealing. I mean, you can actually see its utility, and you get a chance to change something and see how it improves it. So, it's really appealing in that sense. The input/output relationship is really quite intriguing. The other thing is that-I think that in some sense, the problem area is a microcosm of everything you ever wanted to do with computers. It really has all of the interesting problems, but they're there on a somewhat smaller scale, so that one person can encompass it. It's relatively self-contained, and if you do anything useful, people are just really enthusiastic; they love to have it.

MSM: Were you learning things, as you were setting up this text processing system, about computing?

Kernighan: Yeah. You learn-

MSM: Or were you just saying that that had theoretical interest?

Kernighan: Well, yeah, theoretical is not the right word. But, it genuinely had computer science kind of interest, because what you're doing is, you're building compilers, right? But you're building small compilers. You're building them for relatively simple languages, which means that you don't have to face, in your language, all of the problems that may get difficult to build a compiler for a real, conventional mainstream language. For example, a compiler for *eqn* is several orders of magnitude smaller than the compiler for something like C, or FORTRAN, or something like that. So, the job is intrinsically simpler; there are a lot more things that you don't have to worry about. Yet at the same time, retains most of the interest. I think maybe that's part of it. Then the other thing is that it's kind of like many other things in computing; you get to go off and invent your own rules for the game. Especially if you're building languages, which is a lot of what I've done over the years. If you're building a new language for something, well, you can invent the rules. It's not like-if you want to go out

Interview with Brian Kernighan

and build yourself a new C compiler, you can't invent the rules. They're already defined for you. I think that's boring. So, but if somebody says "We need a new language to make it easy to talk about, you know, the way that medieval manuscripts have been illuminated," or something or like that. Well, gee, define your own rules, because nobody else has done it before. That, I think, is part of the charm as well. Before you set out to play the game, you get to define the rules, and if you don't like the way the game turns out, then you change the rules. That's always been part of the charm in computing, I think, building something out of nothing.

MSM: That's a mathematician's approach.

Kernighan: Well, in some sense, yeah. But it's different from at least pure mathematicians; there is, in addition, the reward of utility if you do it well. People come along and say, "Hey, look what I did." Maybe mathematicians don't get any jollies when somebody says, "Oh, I used your theorem." But I think people who write programs do get a kick out of it when somebody comes along and says, "Hey, I used your program." I know I do.

MSM: I think mathematicians-certainly when a mathematician's result unlocks someone else's problem for them, there's got to be an immense sense of gratification.

Kernighan: That's right.

MSM: Plus a validation that what one has developed is a profound theorem, when it begins to unlock other people's problems.

Kernighan: None of these things are profound, but, you know, they make life easier for people. They make it easier for somebody to get something done. So, I personally enjoy that.

MSM: Actually, one of the things you were talking about, that I hadn't thought about, is--typesetting things, which one tends to think of as graphical, is really about languages. Small languages. It's a question of how to capture a description.

Kernighan: The input side is small languages. There's a continuing debate between: is it better to have a linguistically, textually-based description of things, or is it better to have a visually, graphically-based description of things? People who have WYSIWYG editors will tell you it's much better to have it right there on the screen before you, and people like myself, who've grown up with, and are comfortable with, the textually-based tools, will tell you there are lots of things you can do with text that you can't do with the pictures. The truth, of course, is that it's a combination of both; each has its good and bad points. I think we started to make significant progress in a lot of these things when we started to think that what we were doing was building languages, because then you could start to think of using the tools that were around for constructing mainstream languages in these other areas. In addition, the artifacts that we built with these tools themselves, were in some sense cleaner and better, because they were linguistically-based. They became less of a collection of features and more implementation of a relatively coherent grammar and set of rules, and so they were easier. Early mathematics--there had a variety of people, I found out afterwards, who had built tools that were intended to make it easy to typeset mathematics, and they were always full of restrictions, like there shall be at most three levels of subscripts, and two of these and four of those. And the reason was that--

Kernighan: --to build yourself a language, that a lot of these funny restrictions don't happen, because the recursive structure that that tool encourages you to use means that you tend to get things that just go. So, if you need seventeen levels of subscript on something, well, you know, it's no different than

Interview with Brian Kernighan

two levels of subscript. It's just more than one. And so a lot of these arbitrary, capricious restrictions, that showed up in things where somebody was building a program, sort of go away when somebody's building a language with an appropriate set of tools, because the grammar does not really make it easy to talk about restrictions-well, one or two or three-but encourages you to think of just an arbitrary number. It's not that you can't have an infinite number of subscripts; you'd run out of something after a while. But, it's not in the structure of the language.

MSM: Little languages: when did you start learning little languages?

Kernighan: I believe it was--I've forgotten his name, but in *Le bourgeois gentilhomme* the gentleman who realized he'd been speaking prose all his life. (laughing) I should remember his name, but I can't. In some sense, it's the same phenomenon. Somewhere, somebody asked me to give a talk. I looked back and realized that there was, in some way, a unifying theme to a lot of the ways that I had been fooling around over the years, which is that I had been building languages to make it easy to attack this, that, or the other problem. In some way, make it easy for somebody talk to the machine. I started to count them up, and gee, there were a lot of things there that were languages. Some of them were absolutely conventional things, some of them were pre-processors that sat on other things, some were not much more than collections of subroutines; but, you know, you could sort of call them languages. And they were all characterized by being relatively small, as they were things that were done by one or two people, typically. And they were all not mainstream; I never built a C compiler. They were attacking sort of off-the-wall targets. So, I said, gee, well, they're little languages. Then, once you see that, you can start to look for further targets of opportunity. Over the years, that's what I've done. You can push that arbitrarily far, probably too far.

MSM: The way you were describing it before, when you were talking about yacc, and about, when you see these things linguistically, but-I forget exactly what words you used-but, what it triggered in my mind was the sense that, what's come out this research group as a whole is a body of theory that seems particularly adapted to precisely this design of languages. This is the textbooks on the subject of language design, theory based language design-combinations of three things taken two at a time. (Laughing) I should say eight or nine things taken two at a time. But that all fits together, and on the one hand we say "Yes, of course." But, on the other hand, "No, not of course."

Kernighan: I think that there's-you could see the roots of it. The theory-based part of it goes back to Al Aho, I think, starting here-having just finished a thesis at Princeton on a particular class of formal languages, having worked with John Hopcroft, having had Jeff Allman as best friend all the way through graduate school-when the analysis of formal languages was the hot topic in computer science. Computer science didn't really exist in those days, but that was the topic. People were studying the properties of languages. So, when Al got here, he was still interested in that, and I suspect that there were times when his management would wish that he'd get off this damn language stuff and do something that mattered. Fortunately, in the best tradition, he never did. Then when he and Steve Johnson got together, and realized that it was possible-I guess they weren't the first people to make compiler compilers-But they were-

MSM: They were back in books-

Kernighan: Yeah. And the fact that Steve called it yet another compiler compiler suggests that this was again in the air. The difference was that this thing was sufficiently well-engineered that it was a practical thing for other people to use. It was not the personal research vehicle of Steve Johnson or Al Aho, but rather was something other people could use. And furthermore, that there were these collection of weird people around who actually wanted to use it, who were willing to use it, and stress it in ways that hadn't been thought of, and therefore make it a better tool. If you look back, maybe it

Interview with Brian Kernighan

was yet another compiler compiler. I'll bet you can't name another compiler compiler. That's the only one that has survived. It's still used, still used extensively, continuously. All kinds of people, lots of people still discover it, because the job sufficiently well done.

I think the reason was well done is, aside from the intrinsic brightness of the people involved, and the fact that they picked good algorithms and continued to define them over the years-I think it's the milieu of other people sort of banging against it, and trying things with it, and building things with it-build up sort of a collection of things, where you could look at it and say, "Yeah, this is actually useful stuff." It's not a academic exercise. It's genuinely a better way to do things than what you might have had before. So, this collection of people-enough critical mass, if you like-to actually prove the concepts, and prove them in the in addition to working the rough edges off them, and getting them to point were they genuinely practical tools. We use our own stuff, and I think that's a critical observation about this group here. We do not build tools for other people. We do not build anything for other people. I think it's not possible to build things for other people, roughly speaking.

MSM: In the sense that, you just sit and have other people lay out their specs, and then you build the tool?

Kernighan: Right. If I build something for you, even if you spend a lot of time describing to me what you want, and why it's the way it is, it's not going to be as successful as something where I personally face the problems. Now, I may live with you long enough that I start to understand what your problems are, and then I'll probably do a better job, but I think that we have historically done the best on building things that address problems that we face ourselves. That we understand them so well because we face them, either directly-you know, I face that problem myself-or it's the person in the next office.

MSM: This is probably one of those, either "chicken or the egg," or "a little bit this, a little bit of that" questions. What drives this, in essence, is that you're working on your own project, developing your own stuff, but you're also constantly calling on what other people are doing. So, as a result, you tend to reinforce one other's work and you're your own best critical audience. Criticism in the best sense, because you have a stake, not only in finding, but in making stuff work. Therefore when you find problems that are genuine problems-that block you-you want them fixed.

Kernighan: Yep.

MSM: One can see all sorts of things coming together now. You've got to have the right people. Those people have to be working on the right set of problems. And they have to establish a way of working with one another; one can image that happening of itself, or one can have a sense of its somehow being directed. Now, what is it that makes it work here? What is it that was making it work, back in the early '70's?

Kernighan: I don't think it was directed, or least if it was directed, it was done in an incredibly deft and unobtrusive way. Maybe management will tell you that that's worse....(laughing)

MSM: Surely.

Kernighan: If that is true then it is to their eternal credit. Now, I think, in a sense-I mean, Doug was management of at least some part of that. I guess Ken technically was in Doug's department, and Doug is superb that kind of stuff. Insofar as he manages it, he does by superlative constructive criticism at the right time, and by going out and trying your stuff and finding out where it works and where it doesn't work, and then telling you what was good about it and what didn't work. To a lesser

Interview with Brian Kernighan

degree, I suspect that, in some sense, we all do that. I don't think that this was done by any direct management, so we can dispose of that part.

Part of it is a confluence of really good people with reasonably good taste. Particularly Ken and Dennis, who, as far as I can tell, genuinely have truly deep insight, and at the same time, good taste, and at the same time, essentially very close to parallel taste, so that they don't get going in opposite directions. Part of it is happy coincidence, that technology had gotten just about the right point where you could get hardware—a machine to work on—where you didn't have to, in some sense, be beholden to other people. You didn't have to use that machine their way because they paid for it, or something like that; that you could have something that's sort of your own, so you could furnish your computing world the way you wanted it, the way you are comfortable with it. If I want to go off and run on a big IBM machine, there's no way I'm going to be able to do that, because I can't afford it. I'll have to do it the way which is provided by somebody else, because they paid for it, they call the shots. But, if it's a machine that's my own, then I could run it the way I wanted to, assuming I have the technical background to actually make it work. It's some combination of those things, and it would be nice to know precisely what makes it work so that you could clone it. But, my guess is that it's—in a sense, almost an accident. You could see other places that have had that same kind burst of stuff. I think Xerox PARC went through a phase like that, in roughly in the early seventies, where they produced a bunch of really, really good stuff, with a different working style, to some extent, but again, it was a combination of things that led them to produce genuinely innovative stuff that's had an effect on everybody.

MSM: You know, I was talking with Shedell. The guy that did the Paintbrush program, the first Paintbrush program. I just happened to meet him at a reception at the computer museum at Boston. We started talking about Xerox PARC: how did you get your assignment and what was your mission? You know, there was a great deal of similarity. Well, it was sort of a general mission to do interesting things and hope they work out, but look around for something interesting to do and get going on it. He said he was given two years to find something to do. He sat there and he played with Paintbrush, which he had done back in '73, and it was brilliant.

When did you become the group's scribe?

Kernighan: I think at some point fairly early in game, I started writing tutorials. Sort of, how do you use it? Gerry Markey, who is now Sandy Fraser's secretary, said to me one day, "I don't really understand how to use *qed*," which was the text editor, and I said, "Well why don't I write down something that will, sort of, tell you how to use it." And I wrote a tutorial on *qed*. Now, I don't know if it ever did Gerry any good or not, but a lot of people found it useful. It was a sort of, here's how you get started on this kind of stuff, and it was a particular style of writing that nobody that else seemed to be interested in doing. Which was sort of—not a manual, but a "how to do it" tutorial. No better word.

I had already written one of those for my little formatter at Princeton, in fact, so it wasn't the first time I had done it, but this was, I guess, the first time I, sort of, consciously did it. I did several like that. I did one on B; the one on B mutated into the one on C, which then mutated into a C book. And, you know, a variety of others. I did a "cave guide" to the Murray Hill computer center, an underground guide to the Murray Hill computer center, which was (laughing) sufficiently against the received wisdom that they didn't want me to publish it. But I published it anyway. Which was, sort of, one of the useful programs, as opposed to the ones you will find in the manual. That kind of stuff. I guess, from there it's not too big a step to writing things that describe what's good about the operating system, or the environment, or the way of doing things—these sort of pseudo-philosophical papers that explain what you could do when you connect programs together, and what you could do when you had a

Interview with Brian Kernighan

programmable shell, and that kind of stuff, which, where I took basically one paper and probably wrote it about fifteen times. But that kind of thing. So, I think that's part of where this writing came from.

The other aspect was book writing, which in the long run is much greater effect. I think you have much more impact if you have a successful book. I still remember quite clearly how I got started in that. I was enormously lucky when I came here the first summer to be in the office next door to Dick Hamming. I'd heard of Hamming codes because I had taken a course in coding, error-correcting codes at Princeton. I was sitting in my office the first day, as we are sitting here, and this guy came in from next door at eleven o'clock and he said to me, "Hi, I'm Dick Hamming, let's go to lunch." I said, "Well, okay," and he dragged in Vic Vyssotsky, who was right across the hall from me, and we went off to lunch. Later on I discovered, this was the famous Hamming of Hamming codes, and the numerical analysis book that I had used the year before. I think a lot of people had trouble with Dick because, he was one of these guys who was absolutely outspoken, and not afraid to give you an opinion on *anything*, and usually sort of a controversial, rock-the-foundations kind of opinion. But, I didn't mind. That was fine, and he was a nice guy.

One of things that he was fond of saying was that programmers don't know how to write programs. The way we teach programmers how to write programs is we give them a grammar and a dictionary and we say, "Okay, kid, you're a great writer." That's the way programming was taught according to Hamming, and there's a grain of truth in it. He said, "What we really need is a book of style; here's how to write *well*." I don't remember in detail, but his idea of how to write good programs was, I think, pretty half-baked at that point. But, one day he came into my office and he handed me a book and said, "Look at this!" (searching for book) I don't think I still have it down there. I used to have it. It's probably down there somewhere. He handed me this book, and he said, "Look at this! This is awful!" It was a FORTRAN program. It went over two pages. What he was talking about was the numerical analysis in this thing, but what I saw, when I looked at it, was this horrible piece of code, which is the first example in the *Programming Style* book. I looked at it, and said, "Jesus Christ!" I wrote on it in green, "Jesus Christ" (laughing) and then it occurred to me. I know what I can do, because I had somewhere stumbled across Strunk & White, which is basically an "awful, good, awful, good," you know, side by side. You could make a book out of that. You go off and you find awful examples and you show how to do them right. Bill Plauger at the time was in the next office, and he kind of got intrigued by the idea. So, we went off and we hunted the libraries, all the libraries, all of Bell Labs' libraries, and any other place that we went. Bookstores and everything. And we found awful examples, and it was easy to find them. There were just zillions of awful examples, because every programming textbook was awful. We went off and wrote-basically just collected these, tried to get them in a coherent form, where an example would illustrate some principle or guideline on how to write well. Then we just threw it all together and wrote a book. It was, I think, certainly a fun book to write. (Laughing) My wife said to me, "You found your ecological niche."

MSM: I've been looking for a copy. I can't find it anywhere.

Kernighan: I'll find you one. I can probably do that. (searching for book) I'm sure I have one down here, somewhere. Fun book to write, and I think that's how got I started in book writing, at least. Then Plauger and I wrote the Software Tools book shortly thereafter, and then I browbeat Dennis into working on the C book with me.

MSM: You just found you like to write?

Kernighan: I like to write-ultimately, it was deemed to be good thing at the labs. I think when I first wrote that book, it was deemed to be neutral. It wasn't like a positive thing, but I'd only spent about

Interview with Brian Kernighan

three to six months on it, so it wasn't a bad thing either. It was just a neutral thing, and it was only sort of retroactively that it was deemed to have been a good thing. I think Plauger got fired for writing it.

MSM: I want to talk about where he went in a second, but there's an obvious question. When someone takes other people's style and says, "This is the way to do it," and when you talk about its being relatively easy to find examples of bad code writing-where did you learn to write programs? Where did Plauger learn to write programs, that your style is good style?

Kernighan: (laughs) Well, it's not clear the style is good style; it was better than a lot of the programs that we found in textbooks. All of the examples came from textbooks, and they were-I mean, we clearly picked on the bad ones. It was embarrassingly easy to find bad ones, because a lot of these things clearly had never been thought of. Many of them have never been executed. I think a lot of them, the professor wrote the book and said to some of his students, "Why don't you write programs that will answer these exercises?" and then stuck them in the back of the book and never looked at them, and a lot of those were just-they couldn't possibly work, and others were just really bad. You didn't have to look very hard to realize what was bad, and from that, you could start to infer what might be good. I don't think our rules of style are very deep or anything like that. But, we gradually evolved a set of-well, rules, not in any formal sense, but little detailed rules that corresponded to getting your commas in the right places when you write sentences, and realizing between "which" and "that," those kinds of things-and move up to grander things, that relate to the structure of programs.

MSM: Was Strunk & White more than just an inspiration, or was this-did you read through Strunk & White, think in terms of Strunk & White when you were doing this? You just used an example from it.

Kernighan: Yeah. I think that Strunk & White was an excellent example of something that we didn't consciously-I think we must have consciously taken them as the model, because, the sort of side-by-side, "Here's a good one, here's a bad one, here's a good one," and what did we do to get from one to the other, is definitely what we did there. It's not the same, because English is not the same as writing programs. But there's an awful lot of commonality. I still reread Strunk & White every couple of years just to refresh my memory, what they think about good writing, because that's still the best English style book I have read. The other thing we tried to emulate was the brevity, because I think long books are not necessarily better than short books. In fact, they are sometimes worse.

MSM: Try and convince my grad students of that.

Kernighan: There's lots of reason to believe that short is better. (laughing) If nothing else, even if it isn't better, there isn't so much of it.

MSM: You started out using FORTRAN, the old FORTRAN, which is not a structured language. You spent time optimizing, writing an assembler. Ultimately you made a transition to what might be called structured programming, and Plauger went off to work for a company that is probably the standard bearer of the structured approach to things. Was that a conscious transition for you? Or, did you think of it as a transition? Was there a time in which Dijkstra was on the shelf here?

Kernighan: Yeah. It was-structured programming, and programming without *goto*'s and all that stuff was definitely very much a topic of discussion at the time when Bill and I were working on this book, and noticeably before. Quite a bit before. We wrote a couple of papers before we got into the book, or concurrently with the book, I guess. I remember, somebody or somewhere, I was introduced to the notion that you could write a program without *goto* statements if you had the right kind of language, and B was the first example that I really used where that was a possibility. And so I set myself the task of writing a particular program-something that just reported on job status on the Honeywell machine or

Interview with Brian Kernighan

GE machine-I set myself the task of writing that program without any *goto* statements. I said that this would be my test case, and it was a program that ultimately wound up with, I don't know, six or seven hundred lines of code. I would write it without *goto* statements, and I found it an *incredible* bitch of a job. I finally got it down to three, and I was quite proud of myself. In hindsight, you know, fifteen, twenty years later, I cannot image what the problem was; how could it be so difficult? Because I just don't do that anymore. You know, it's like training yourself so that you always get "which" and "that" correctly. You don't think about it anymore, and you don't realize it's an issue, you just do it correctly. In that sense it was conscious for me. I had to work very hard to get to a particular level, where then, after a while, it became completely unconscious. And there were other kinds of programming, things that go along with-

MSM: But you were rather persuaded by the virtues of doing so?

Kernighan: I think so. The first big program I ever wrote, truly big, was a COBOL program. COBOL programs are sort of the antithesis of structure; in fact, the only way you could write a COBOL program is basically to make it unstructured. It's just awful. And I still remember, the way that I wrote that program was, every time I had to make a decision about something-is this bigger than this?-then I'd say *goto*, I'd invent a label, and I had this steadily increasing frontier of things I hadn't figure out how to do yet. (laughing) The program just grew without bounds, and I never finished the damn thing. In retrospect, it's the kind of program that I could probably have written in a hundred lines in a decent language, and it was just-it was literally over ten thousand lines at the point where I quit, at the end of the summer. So, in that sense, I was convinced that, at least, just pouring in these *goto* statements was wrong. Looking at some of the examples in the textbooks, it was clear that was wrong. I'm not sure that I believed Dijkstra in any pure sense. I really don't remember when I first read his famous letter to the editor; it was probably well after I had been doing this kind of stuff for a while. A lot of that's just blurred.

MSM: I started doing this going back to school in '82. I was taking a Pascal course, and I learned to program directly, in machine language, or-I guess we had alphanumeric-no, it was 074, it was clear and out. You just had to know what the codes were. I remember the first time we were supposed to have a branch decision, having to search out, have to go see, and I said, "Where's the *goto*?"

Kernighan: (Laughing) Yeah. Right.

MSM: They said there is none. You're not supposed to use it. And then, they explained what this was all about, and then I remember getting Wirth's book, and looking at it, and it's got *goto*'s. It usually has *goto*'s, at least to the error, so he gets all his errors together. So, here's the ultimate structured programming language, and there's the author of it using *goto*'s. Like most truths, it lies somewhere in the middle.

Kernighan: There's a great line in Strunk & White about, you know, you'll find places where the rules are broken, but you'll find some compensating merit to make up for it.

MSM: Well, there's Winston Churchill's remark about prepositions, that it's nonsense up with which I were.

Another aspect of structured programming, at least the approach that you were talking about before, is this use of recursion. I remember when I was reading the proceedings of the Garmisch and Rome conferences, and I remember now, someone saying something like, everybody recognizes the virtues of recursive procedures, and yet, no operating system, no large software system program, uses them. And it was in Holmdel, someone handed me a-rules for programmers from another project, and one of

Interview with Brian Kernighan

the rules was no recursive procedures. And yet, the Unix systems were-seemed to take-what was going on there?

Kernighan: I think that there was, for a long time, a conception-perhaps originally based on fact, but laterally not at all-that recursion was in some way expensive, and therefore you couldn't use it; that a recursion procedure, if nothing else, it was another procedure call, and therefore procedure calls were more expensive and therefore you couldn't use it. Now, it wasn't a issue in FORTRAN, where you couldn't recurse, but it was in PL/1, where you could recurse, and PL/1 function calls were notoriously expensive because of the language design; it sort of forced you to carry around incredible baggage to actually say what the arguments were as you went along. So I suspected it came from that fear that, when you-any extra function call was going to cost you a lot, and recursion was intrinsically function calls in place where you didn't have to use them, and therefore it was a bad idea.

MSM: Was call by value in C? C's calls are very spare. Was that it was there for? To support recursion more easily?

Kernighan: I not sure it was to support recursion. I think it's simply to support function calls, because functions, breaking up the job into smaller pieces, I think is necessarily a better way to do things. But, if you're going to that, you want to make sure that they are efficient. I think in part it was a reaction to the PL/1 stuff, where functions were so expensive that people tended to avoid them. And they don't have to be expensive, they can be quite cheap. In fact, there was a lot of effort devoted over the years to various ways to make the function calling inexpensive so that people would be encouraged to use it. C compilers to this day go to some effort to do that. And furthermore, a lot of that is reflected in the machine architecture, where the machines themselves drive the subroutine call, mechanism cheap. Part of it was just the machine. A machine like, say the IBM 360's, 370's, the subroutine calling mechanism was actually intrinsically expensive; it was badly designed in the machine. The machines like PDP-11's and their derivatives, the function call mechanism is actually well-defined, and it's comparatively cheap, because basically, the machine supports the notion of a stack in some way-the auto-incremental structure is one, manipulated stack made it easy to get in and out, and there's even a-A del had a subroutine instruction; the question is, what do you have to do before and after it? The DEC machines just did it right, and IBM machines just did it wrong.

MSM: Have the, just following along those lines-you said earlier something along the lines that, it was the right moment, the hardware was coming along. Is there an extent to which this development, maybe its timing, or its pace, has been hardware determined? To what you had available? Or has it reached the point where it's now beginning to influence hardware design?

Kernighan: Certainly the original development and pace, I think, was very strongly affected by the hardware. The PDP-11, I never programmed it in assembly language, so I can't speak intimately of that, as Ken or Dennis could. But, my belief is that, from a programmer's standpoint, as a dream machine-that's really a very, very nice machine. And from an economic standpoint, it was kind of a dream machine too. It would fifty thousand dollars fifteen years ago, and that was manageable. That was something where a group of ten people could justify spending that much money. It was an order of magnitude cheaper than existing machines that you could buy and you could use for something So, I think-and furthermore, it was available from a real manufacturer, somebody who existed, who was in some ways comparatively easy to deal with, certainly much easier to deal with than with IBM; and a manufacturer that, perhaps, because of its origins, made it easy to connect interesting widgets to the machine. A lot of things combined together to make the 11 a very attractive machine. IBM always played it very close to the chest about what you connect to their machines; you had to have these elaborate channel things. It was just horrible. DEC machines, you could just plug in stuff. It was a whole industry of weird things you could connect to DEC machines, all of which made it substantially

Interview with Brian Kernighan

more interesting. And the fact that it was small enough that a university department could buy it-it made it spread. So, it got to the point where, the combination of the attractiveness of the hardware and the fact that there was this attractive software that you could get, almost for free, to go on it, meant that large numbers of university computer science and related departments would go off and buy the things, and run them. I think the VAX carried on the tradition, although the VAX is probably, today, excessively pricey for what it does. But, I think what we're finding now is that the portability aspect of Unix mean that in some sense the hardware is almost irrelevant. There are certain kinds of hardware where Unix and C are not going live comfortably. But, for the most part, the hardware is almost irrelevant. I think that, in some sense, is a giant leap forward.

When I first got into computing, and for the first ten or so years, up until the mid '70s, at least, the arrival of a new computer-you were going to go from GE to IBM, or from UNIVAC to Data General or something-that was just a trauma beyond words. It would months--it would take *years*-to make that adaptation. It doesn't happen that way anymore. Things are-I mean, you still spend a lot of time as you move from one machine to another. But, it's a different-it really is a different order of magnitude. We've slipped in several new machines locally, and a lot of the software, you just compile it and it runs, and the operating system environment is essentially identical. You type the same commands, they behave the same way, the shell is the same; it's the *same*. That, I think is a tremendous step forward. I don't know whether that relates to your original question, which I've now managed (laughing) to lose from my neurons, but-

MSM: No, actually, it's always seemed to me it's sort of-a bit of a contradiction that, the most popular host machine for Unix, during this period of rapid spread, was the DEC PDP-11, or the VAX, which is just an extension of the 11. You can look at the manuals, and it's one of the richest instruction sets, I think, that's ever been put on a machine. I mean, you've got an assembly language routine for evaluating polynomials, and all the packed array, decimal packed array stuff--it's fun to go and play with it. Sort of an assembler programmer's playground. I can see the attraction of doing it.

But also, the ethos of Unix is you stay out of assembler. You don't program in assembler. You use C, and you try to keep the assembler kernel down to just what's necessary to get things up and running. If I understand it correctly, it's right there in the bootstrap command. From there on in, you address it in C, and though I haven't looked at the C compiler, my sense is that, for portability concerns, if for no other, one would say they're a pretty restrictive set of assembler routines, and that in many ways the RISC architecture is better suited in the Unix system, than this extra rich instructions set. That's what led to-the instruction set on the VAX says, wasted if you run Unix. And the irony is that two-thirds of the VAXes are run on Unix.

Kernighan: Yep. Absolutely.

MSM: (Laughing) That was meant to be a question; obviously it failed.

Kernighan: Well, I believe what you say. I don't think that people write assembly language on most machines that run Unix, except for some tiny thing that they can't get at any other way. I have not written an assembly language program in eons, fifteen years probably, and never to expect to again; I hope never to have to. You're right; most machines provide a very, very rich set of instructions, most of which are inaccessible from your favorite high level language, whatever it is. Now, packed decimal is presumably there for the benefit of COBOL, right?

MSM: I don't know, (Laughing) I never could understand what that's for. I simply took Peter Huntington's challenge once, to do something in packed decimal arithmetic. Pain in the neck.

Interview with Brian Kernighan

Kernighan: But, you're right. There's no compiler in the world that's going to generate the Horner's rule instructions. It just isn't going to happen. Poly-q or whatever it's called; it's just not going to happen. So a lot of the effort that goes into building that kind of complicated architecture is totally wasted. The RISC architecture is, insofar as they are uniform, are a better match to what you likely want out of your favorite compiler.

MSM: I've run out of questions for now.

Kernighan: What are you doing for lunch?

MSM: Nothing.