An Interview with Donald Knuth 1974 ACM Turing Award Recipient

Interviewed by: Edward Feigenbaum

March 14, 2007 and March 21, 2007

Mountain View, California

DK: Donald Knuth, The 1974 ACM Turing Award Recipient

EF: Edward Feigenbaum, a professor at Stanford University

EF: My name is Edward Feigenbaum. I'm a professor at Stanford University, in the Computer Science Department. I'm very pleased to have the opportunity to interview my colleague and friend from 1968 on, Professor Don Knuth of the Computer Science Department. Don and I have discussed the question of what viewers and readers of this oral history we think there are. We're orienting our questions and comments to several groups of you readers and viewers. First, the generally intelligent and enlightened science-oriented person who has seen the field of computer science explode in the past half century and would like to find out what is important, even beautiful, and what some of the field's problems have been. Second, the student of today who would like orientation and motivation toward computer science as a field of scholarly work and application, much as Don and I had to do in the 1950s. And third, those of you who maybe are not yet born, the history of science scholar of a dozen or 100 years from now, who will want to know more about Donald Knuth, the scientist and programming artist, who produced a memorable body of work in the decades just before and after the turn of the millennium. Don and I share several things in our past in common. Actually, many things. We both went to institutes of technology, I to Carnegie Institute of Technology, now Carnegie Mellon University, and Don to Case Institute of Technology, now Case Western Reserve. We both encountered early computers during that college experience. We both went on to take a first job at a university. And then our next job was at Stanford University, in the new Computer Science Department, where we both stayed from the earliest days of the department. I'd like to ask Don to describe his first encounter with a computer. What led him into the world of computing? In my case, it was an IBM 701, learned from the manual. In Don's case, it was an IBM Type 650 that had been delivered to Case Institute of Technology. In fact, Don even dedicated one of his major books to the IBM Type 650 computer. Don, what is the story of your discovery of computing and your early work with this intriguing new artifact?

DK: Okay. Thanks, Ed. I guess I want to add that Ed and I are doing a team thing here; next week, I'll be asking Ed the questions that he's asking me today. We thought that it might be more fun for both of us and, also for people who are listening or watching or reading this material, to see the symmetrical approach instead of having a historian interviewing us. We're colleagues, although we work in different fields. We can give you some slants on the thing from people who sort of both have been there. We're going to be covering many years of the story today, so we can't do too much in-depth. But we also want to do a few things in depth, because the defining thing about computer science is that computer science plunges into things at low levels as well as sticking on high level. Since we're going to cover so many topics, I'm sure that I won't sleep tonight because I'll be saying to myself, "Oh, I should've said such and such when he asked me that question". So I think Ed and I also are going to maybe add another little thing to this oral interview, where we might want to add a page or two of afterthoughts that come to us later, because then I don't have to be so careful about answering every question that he asks me now. The interesting thing will be not only the wrong answer that pops first in my mind, but also maybe a slightly corrected thing. One of the stories of my life, as you'll probably find out, is I try to get things correct. I probably obsess about not making too many mistakes. Okay. Now, your question, Ed, was how did I get into the computing business. When the computers were first built in the '40s I was ten years old, so I certainly was not a pioneer in that sense. I saw my first computer in 1957, which is pretty late in the history game as far as computers are concerned. On the other hand, programming was still pretty much a new thing. There were, I don't know, maybe 2,000 programmers in the world at that time. I'm not sure how to figure it, but it was still fairly early from that point of view. I was a freshman in college. Your question was: how did I get to be a physics student there in college. I grew up in Milwaukee, Wisconsin. Those of you who want to do the math can figure out, I was born in 1938. My father was the first person among all his ancestors who had gone to college. My mother was the first person in all of her ancestors who had gone to a year of school to learn how to be a typist. There was no tradition in our family of higher education at all. I think [that's] typical of America at the time. My great-grandfather was a blacksmith. My grandfather was a janitor, let's say. The people were pretty smart. They could play cards well, but they didn't have an academic background. I don't want to dwell on this too much, because I find that there's lots of discussion on the internet about the early part of my life. There's a book called Mathematical People, in which people asked me these questions at length -- how I got started. The one thing that stands out most, probably, is when I was an eighth grader there was a contest run by a local TV station, a company called Zeigler's Giant Bar. They said, "How many words can you make out of the letters in 'Zeigler's Giant Bar'?" Well, there's a lot of letters there. I was kind of intrigued by this question, and I had just seen an unabridged dictionary. So I spent two weeks going through the unabridged dictionary, finding every word in that dictionary that could be made out of the letters in "Zeigler's Giant Bar". I pretended that I had a stomachache, so I stayed home from school those two weeks. The bottom line is that I found 4500 words that could be made, and the judges had only found 2500. I won the contest, and I won Zeigler's Giant Bars for everybody in my class, and also got to be on television and everything. This was the first indication that I would obsess about problems that would take a little bit of - - what do you call it? -- long attention span to solve. But my main interest in those days was music. I almost became a music major when I went to college. Our high school was not very strong in science, but I had a wonderful chemistry and physics teacher who inspired me. When I got the chance to go to Case, looking back, it seems that the thing that really turned it was that Case was a challenge. It was supposed to have a lot of meat. It wasn't going to be easy. At the college where I had been admitted to be a music major, the people, when I visited there, sort of emphasized how easy it was going to be there. Instead of coasting, I think I was intrigued by the idea that Case was going to make me work hard. I was scared that I was going to flunk out, but still I was ready to work. I worked especially hard as a freshman, and then I coasted pretty much after that. In my freshman year, I started out and I found out that my chemistry teacher knew a lot of chemistry, but he didn't know physics or mathematics. My physics teacher knew physics and chemistry, but he didn't know much about mathematics. But my math teacher knew all three things. I was very impressed by my math teacher. Then in my sophomore year in physics, I had to take a required class of welding. I just couldn't do welding, so I decided maybe I can't be a physicist. Welding was so scary. I've got these thousands of volts in this stuff that I'm carrying. I have to wear goggles. I can't have my glasses on, I can't see what I'm doing, and I'm too tall. The table is way down there. I'm supposed to be having these scary electrons shooting all over the place and still connect X to Y. It was terrible. I was a miserable failure at welding. On the other hand, mathematics! In the sophomore year for mathematicians, they give you courses that are what we now call discrete mathematics, where you study logic and things that are integers instead of continuous quantities. I was drawn to that. That was something, somehow, that had great appeal to me. Meanwhile, in order to support myself, I had to work for the statisticians at Case. First this meant drawing graphs and sorting cards. We had a fascinating machine where you put in punch cards and they fall into different piles, and you can look at what comes out. Then I could plot the numbers on a graph, and get some salary from this. Later on in my freshman year there arrived a machine that, at first, I could only see through the glass window. They called it a computer. I think it was actually called the IBM 650 "Univac". That was a funny name, because Univac was a competing brand. One night a guy showed me how it worked, and gave me a chance to look at the manual. It was love at first sight. I could sit all night with that machine and play with it. Actually, to be exact, the first programs I wrote for the machine were not in machine language but in a system called The Bell Interpreter System. It was something like this. You have an instruction, and the instruction would say, "Add the number in cell 2 to the number in cell 15 and put the result in cell 30." We had instructions like that, a bunch of them. This was a simple way to learn programming. In fact, I still believe that it might be the best way to teach people programming, instead of teaching them what we call high-level language right now. Certainly, it's something that you could easily teach to a fourth or fifth grader who hasn't had algebra yet, and get the idea of what a machine is. I was pledging a fraternity, and one of my fraternity brothers didn't want to do his homework assignment where he was supposed to find the roots of a fifth-degree equation. I looked at some textbooks, and it told me how to solve a fifth-degree equation. I programmed it in this Bell Interpretive Language. I wrote the program. My memory is that it worked correctly the first time. I don't know if it really gave the right answers, but miraculously it ground out these numbers. My fraternity brother passed his course, I got into the fraternity, and that was my first little program. Then I learned about the machine language inside the 650. I wrote my first program for the 650 probably in the spring of my freshman year, and debugged it at night. The first time I wrote the program, it was about 60 instructions long in machine language. It was a program to find the prime factors of a number. The 650 was a machine that had decimal arithmetic with ten-digit numbers. You could dial the numbers on the console of the machine. So you would dial a ten- digit number, and my program would go into action. It would punch out cards that would say what are the factors of this number that you dialed in there. The computer was a very slow computer. In order to do a division instruction, it took five milliseconds. I don't know, is that six orders of magnitude slower than today's machines, to do division? Of course, the way I did factoring was by division. To see if a number was divisible by 13, I had to divide by 13. I divided by 15 as well, 17, 19. It would try to find everything that divided. If I started out with a big ten-digit number that happened to be prime -- had no dividers at all -- I think it would take 15 or 20 minutes for my program to decide. Not only did my program have about 60 or so instructions when I started, they were almost all wrong. When I finished, it was about 120, 130 instructions. I made more errors in this program than there were lines of code! One of the things that I had to change, for example, that took a lot of work, was I had originally thought I could get all the prime factors onto one card. But a card had 80 columns, and each number took ten digits. So I could only get eight factors on a single card. Well, you take a number like 2 to the 32nd power, that's going to take four cards. Because it's two times two times two times two [and so on]. I had to put in lots of extra stuff in my program that would handle these cases. So my first program taught me a lot about the errors that I was going to be making in the future, and also about how to find errors. That's sort of the story of my life, is making errors and trying to recover from them. Did I answer your question yet?

EF: No.

DK: No.

EF: Don, a couple questions about your early career, before Case and at Case. It's very interesting that you mentioned the Zeigler's Giant Bar, because it points to a really early interest in combinatorics. Your intuition at combinatorics is one of the things that impresses so many of us. Why combinatorics, and how did you get to that? Do you see combinatorics in your head in a different way than the rest of us do?

DK: I think that there is something strange about my head. It's clear that I have much better intuition about discrete things than continuous things. In physics, for example, I could pass the exams and I could do the problems in quantum mechanics, but I couldn't intuit what I was doing. I didn't feel right being able to get an "A" on an exam without ever having the idea of how I would've thought of the questions that the person made up solving the exam. But on the other hand, in my discrete math class, these were things that really seemed part of me. There's definitely something in how I had developed by the time I was a teenager that made me understand discrete objects, like zeros and ones of course, or things that are made out of alphabetical letters, much better than things like Fourier transforms or waves -- radio waves, things like this. I can do these other things, but it's like a dog standing on his hind legs. "Oh, look, the dog can walk." But no, he's not walking; he's just a dog trying to walk. That's the way it is for me in a lot of the continuous or more geometrical things. But when it comes to marks on papers, or integer numbers like finding the prime factors of something, that's a question that appealed to me more than even finding the roots of polynomial.

EF: Don, question about that. Sorry to interject this question, behaving like a cognitive psychologist.

DK: This is what you're paid to do, right?

EF: Right. Herb Simon -- Professor Simon, of Carnegie Mellon University -- once did a set of experiments that kind of separated thinkers into what he called "visualizers" and "symbolizers". When you do the combinatorics and discrete math that you do, which so amazes us guys who can't do it that well, are you actually visualizing what's going on, or is it just pure symbol manipulation?

DK: Well, you know, I'm visualizing the symbols. To me, the symbols are reality, in a way. I take a mathematical problem, I translate it into formulas, and then the formulas are the reality. I know how to transform one formula into another. That should be the subtitle of my book Concrete Mathematics: How to Manipulate Formulas. I'd like to talk about that a little. It started out… My cousin, Earl, who died, Earl Goldschlager [ph?], he was a engineer, eventually went to Southern California, but I knew him in Cleveland area. When I was in second grade he went to Case. He was one of the people who sort of influenced me that it may be good to go to Case. When I was visiting him in the summer, he told me a little bit about algebra. He said, "If you have two numbers, and you know that the sum of these numbers is 100 and the difference of these numbers is 20, what are the two numbers?" He said, "You know how you can solve this, Don? You can say X is one of the numbers and Y is one of the numbers. X plus Y is 100. X minus Y is 20. And how do you find those numbers?" he says. "Well, you add these two equations together, and you get 2X = 120. And you subtract the equation from each other, and you get 2Y = 80. So X must be 60, and Y must be 40. Okay?" Wow! This was an "aha!" thing for me when I was in second grade. I liked symbols in this form. The main thing that I enjoyed doing, in seventh grade, was diagramming sentences. NPR had a show; a woman published a book about diagramming sentences, "The Lost Art of Diagramming Sentences", during the last year. This is where you take a sentence of English and you find its structure. It says, "It's a noun phrase followed by a verb phrase." Let's take a sentence here, "How did you get to be a physics student?" Okay. It's not a noun phrase followed by a verb phrase; this is an imperative sentence. It starts with a verb. "How did you get…" It's very interesting, the structure of that sentence. We had a textbook that showed how to diagram simple English sentences. The kids in our class, we would then try to apply this to sentences that weren't in the book, sentences that we would see in newspapers or in advertisements. We looked at the hymns in the hymnal, and we couldn't figure out how to diagram those. We spent hours and hours trying to figure this out. But we thought about structure of language, and trying to make these discrete tree structures out of English sentences, in seventh grade. My friends and I, this turned us on. When we got to high school, we breezed through all our English classes because we knew more than the teachers did. They had never studied this diagramming. So I had this kind of interest in symbols and diagramming early on -- discrete things, early on. When I got into logic as a sophomore, and saw that mathematics involved a lot of symbol manipulation, then that took me there. I see punched cards in this. I mean, holes in cards are nice and discrete. The way a computer works is totally discrete. A computer has to stand on its hind legs trying to do continuous mathematics. I have a feeling that a lot of the brightest students don't go into mathematics because -- curious thing -- they don't need algebra at the level I did. I don't think I was smarter than the other people in my class, but I learned algebra first. A lot of very bright students today don't see any need for algebra. They see a problem, say, the sum of two numbers is 100 and the difference is 20, they just sort of say, "Oh, 60 and 40." They're so smart they don't need algebra. They go on seeing lots of problems and they can just do them, without knowing how they do it, particularly. Then finally they get to a harder problem, where the only way to solve it is with algebra. But by that time, they haven't learned the fundamental ideas of algebra. The fact that they were so smart prevented them from learning this important crutch that I think turned out to be important for the way I approach a problem. Then they say, "Oh, I can't do math." They do very well as biologists, doctors and lawyers.

EF: You're recounting your interest in the structure of languages very early. Seventh grade, I think you said. That's really interesting. Because among the people -- well, the word "computer science" wasn't used, but we would now call it "information technology" people -- your early reputation was in programming languages and compilers. Were the seeds of that planted at Case? Tell us about that early work. I mean, that's how I got to know you first.

DK: Yeah, the seeds were planted at Case in the following way. First I learned about the 650. Then, I'm not sure when it was but it probably was in the summer of my freshman year, where we got a program from Carnegie -- where you were a student -- that was written by Alan Perlis and three other people.

EF: "IT".

DK: The IT program, "IT", standing for Internal Translator.

EF: Yeah, it was Perlis, [Harold] van Zoeren, and Joe Smith.

DK: In this program you would punch on cards a algebraic formula. You would say, "A = B + C." Well, in IT, you had to say, "X1 = X2 + X4." Because you didn't have a plus sign, you had to say, "A" for the plus sign. So you had to say, "X1 Z X2 A X4." No, "S," I guess, was plus, and "A" was for absolute value. But anyway, we had to encode algebra in terms of a small character set, a few letters. There weren't that many characters you could punch on a card. You punch this thing on a card, and you feed the card into the machine. The lights spin around for a few seconds and then -- punch, punch, punch, punch -- out come machine language instructions that set X1 equal to X2 + X4. Automatic programming coming out of an algebraic formula. Well, this blew my mind. I couldn't understand how this was possible, to do this miracle where I had just these punches on the card. I could understand how to write a program to factor numbers, but I couldn't understand how to write a program that would convert algebra into machine instructions.

EF: It hadn't yet occurred to you that the computer was a general symbol-manipulating device.

DK: No. No, that occurred to Lady Lovelace, but it didn't occur to me. I'm slow to pick up on these things, but then I persevere. So I got hold of the source code for IT. It couldn't be too long, because the
650 had only 2,000 words of memory, and some of those words of memory had to be used to hold the data as well as the instructions. It's probably, I don't know, 1,000 lines of code. The source code is not hard to find. They published it in a report and I've seen it in several libraries. I'm

pretty sure it's on the internet long ago. I went through every line of that program. During the summer we have a family get- together on a beach on Lake Erie. I would spend the time playing cards and playing tennis. But most of the time I was going through this listing, trying to find out the miracle of how IT worked. Ok, it wasn't impossible after all. In fact, I thought of better ways to do it than were in that program. Since we're in a history museum, we should also mention that the program had originally been developed when Perlis was at Purdue, before he went to Carnegie, with three other people there. I think maybe Smith and van Zoeren came with Alan to Carnegie. But there was Sylvia Orgel and several other people at Purdue who had worked on a similar project, for a different computer at Purdue. Purdue also produced another compiler, a different one. It's not as well-known as IT. But anyway, I didn't know this at the time, either.

The code, once I saw how it happened, was inspiring to me. Also, the discipline of reading other people's program was something good to learn early. Through my life I've had a love of reading source materials -

- reading something that pioneers had written and trying to understand what their thought processes were in order to write this out. Especially when they're solving a problem I don't know how to solve, because this is the best way for me to put into my own brain how to get past stumbling blocks. At Case I also remember looking at papers that Fermat had written in Latin in the 17th century, in order to understand

how that great number theorist approached problems. I have to rely on friends to help me get through Sanskrit manuscripts and things now, but I still…. Just last month, I found, to my great surprise, that the concept of orthogonal Latin squares, which we'll probably talk about briefly later on, originated in North Africa in the 13th century. Or was it the 14th century? I was looking at some historical documents and I

came across accounts of this Arabic thing. By reading it in French translation I was able to see that the guy really had this concept, orthogonal Latin squares, that early. The previous earliest known example was 1724. I love to look at the work of pioneers and try to get into their minds and see what's happening.

EF: One of the things worth observing -- it's off the track but as long as we're talking about history -- is that our current generation, and generations of students, don't even know the history of their own field. They're constantly reinventing things, or thoughtlessly disregarding things. We're not just talking about history going back in time hundreds of years. We're talking about history going back a dozen years, or two-dozen years.

DK: Yeah, I know. It's such a common failing. I would say that's my major disappointment with my teaching career. I was not able to get this across to any of my students this love for that kind of scholarship, reading source material. I was a complete failure at passing this on to the people that I worked with the most closely. I don't know what I should've done. When I came to Stanford from Caltech, I had been researching Pascal. I couldn't find much about Pascal's work in the Caltech library. At Stanford, I found two shelves devoted to it. I was really impressed by that. Then I came to the Stanford engineering library, and everything was in storage if it was more than five years old. It was a basket case at that time, in the 60's.

DK: I've got to restrain myself from not telling too much about the early compiler. But anyway, after IT, I have to mention that I had a job by this time at the Case Computing Center. I wasn't just growing grass for statisticians anymore. Case was one of the very few institutions in the country with a really

enlightened attitude that undergraduate students were allowed to touch the computers by themselves, and also write software for the whole campus. Dartmouth was another place. There was a guy named Fred Way who set the policy at Case, and instead of going the way most places go, which would hire professionals to run their computer center, Case hired its own students to play with the machines and to do the stuff everybody was doing. There were about a dozen of us there, and we turned out to be fairly good contributors to the computing industry in the long range of things. I told all of my friends how this IT compiler worked, and we got together and made our own greatly improved version the following year. It was called RUNCIBLE. Every program in those days had to have an acronym and this was the Revised Unified New Compiler Basic Language Extended, or something like this. We found a reason for the name. But we added a million bells and whistles to IT, basically.

EF: All on the 2000 word drum.

DK: All on the 2000 word drum. Not only that, but we had four versions of our compiler. One of them would compile to assembly language. One would compile directly into machine language. One version would use floating point hardware. And one version would use floating point attachment. If you changed
613 instructions, you would go from the floating point attachment to the floating point hardware version. If you changed another 372 instructions, it would change from the assembly language version to the machine language version. If we could figure out a way to save a line of code in the 373 instructions in one version, then we had to figure out a way to correspondingly save another line of code in the other version. Then we could have another instruction available to put in a new feature. So RUNCIBLE went through the stages of software development that have since become familiar, where there is what they
call "creeping featurism", where every user you see wants a new thing to be added to the software. Then you put that in and pretty soon the thing gets… you have a harder and harder user manual. That is the way software always has been. We got our experience of this. It was a group of us developing this; about, I don't know, eight of us worked together on different parts of it. But my friend, Bill Lynch, and I
did most of the parts that were the compiler itself. Other people were working on the subroutines that would support the library, and things like that. Since I mentioned Bill Lynch, I should also, I guess... I wrote a paper about the way RUNCIBLE worked inside, and it was published in the Communications of the ACM during my senior year, because we had seen other articles in this journal that described methods that were not as good as the ones that were in our compiler. So we thought, okay, let's put it to work. But I had no idea what scientific publishing was all about. I had only experienced magazines before, and magazines don't give credit for things, they just tell the news. So I wrote this article and it
explained how we did it in our compiler. But I didn't mention Bill Lynch's name or anything in the

article. I found out to my great surprise afterwards that I was getting credit for having invented these things, when actually it was a complete team effort. Mostly other people, in fact. I had just caught a few bugs and done a lot of things, but nothing really very original. I had to learn about scholarship, about scientific publishing and things as part of this story. So we got this experience with users, and I also wrote the user manual for this machine. I am an undergraduate. Case allows me to write the user manual for RUNCIBLE, and it is used as a textbook in classes. Here I've got a class that I am taking; I can take a class and I wrote the textbook for it already as an undergraduate. This meant that I had an unusual visibility on campus, I guess. The truth is that Case was a really great college for undergraduates, and it had superb teachers. But it did not have very strong standards for graduate studies. It was very difficult to get admitted to the undergraduate program at Case, and a lot of people would flunk out. But in graduate school it wasn't so hard to get over. I noticed this, and I started taking graduate courses, because there was no competition. This impressed my teachers -- "Oh, Knuth is taking graduate courses" -- not realizing that this was line of least resistance so that I could do other things like write compilers as a student. I edited a magazine and things like that, and played in the band, and did lots of activity. Now [to] the story, however: What about compilers? Well, I got a job at the end of my senior year to write a compiler for Burroughs, who wanted to sell their drum machine to people who had IBM 650s. Burroughs had this computer called the 205, which was a drum machine that had 4000 words of memory instead of 2000, and they needed a compiler for it. ALGOL was a new language at the time. Somebody heard that I knew something about how to write compilers, and Thompson Ramo Wooldridge [later TRW Inc.] had a consulting branch in Cleveland. They approached me early in my senior year and said, "Don, we want to make a proposal to Burroughs Corporation that we will write them an ALGOL compiler. Would you write it for us if we got the contract?" I believe what happened is that they made a proposal to Burroughs that for $75,000 they would write a ALGOL compiler, and they would pay me $5,000 for it, something like this. Burroughs turned it down. But meanwhile I had learned about the 205 machine language, and it was kind of appealing to me. So I made my own proposal to Burroughs. I said I'll write you an ALGOL compiler for $5,000, but I can't implement all of ALGOL. I think I told them I can't implement all of ALGOL for this; I am just one guy. Let's leave out procedures -- subroutines. Well, this is a big hole in the language! Burroughs said, "No, no -- you got to put in procedures." I said, "Okay, I will put in procedures, but you got to pay me $5,500." That's what happened. They paid me $5,500, which was a fairly good salary in those days. I think a college professor was making eight or nine thousand dollars a year in those days. So between graduating from Case and going to Cal Tech, I worked on this compiler. As I drove out to California, I drove a 100 miles a day and I sat in a motel and wrote code. The coding form on which I wrote this code, I now donated it to the Computer History Museum, and you can see exactly the code that I wrote. I debugged it, and it was Christmas time [when] I had the compiler ready for Burroughs to use. So I was interested; I had two compilers that I knew all the code by the end of the '60s. Then I learned about other projects. When I was in graduate school some people came to me and said, "Don, how about writing software full time? Quit graduate school. Just name your price. Write compilers for a living, and you will have a pretty good living." That was my second year of graduate school.

EF: In what department at Cal Tech?

DK: I was at Cal Tech in the math department. There was no such thing as a computer science department anywhere.

EF: Right. But you didn't do physics.

DK: I didn't do physics. I switched into math after my sophomore year at Case, after flunking welding. I switched into math. There were only seven of us math majors at Case. I went to Cal Tech, and that's another story we'll get into soon. I'm in my second year at Cal Tech, and I was a consultant to Burroughs. After finishing my compiler for Burroughs, I joined the Product Planning Department. The Product Planning Department was largely composed of people who had written the best software ever done in the world up to that time, which was a Burroughs ALGOL compiler for the 220 computer. That was a great leap forward for software. It was the first software that used list processing and high level data structures in an intelligent way. They took the ideas of Newell and Simon and applied them to compilers. It ran circles around all the other things that we were doing. I wanted to get to know these people, and they were by this time in the Product Planning Group, because Burroughs was doing its very

innovative machines that are the opposite of RISC. They tried to make the machine language look like algebraic language. This group I joined at Burroughs as a consultant. So I had a programming hat when I was outside of Cal Tech, and at Cal Tech I am a mathematician taking my grad studies. A startup company, called Green Tree Corporation because green is the color of money, came to me and said, "Don, name your price. Write compilers for us and we will take care of finding computers for you to debug them on, and assistance for you to do your work. Name your price." I said, "Oh, okay. $100,000.", assuming that this was… In that era this was not quite at Bill Gate's level today, but it was sort of out there. The guy didn't blink. He said, "Okay." I didn't really blink either. I said, "Well, I'm not going to do it. I just thought this was an impossible number." At that point I made the decision in my life that I wasn't going to optimize my income; I was really going to do what I thought I could do for… well, I don't know. If you ask me what makes me most happy, number one would be somebody saying "I learned something from you". Number two would be somebody saying "I used your software". But number infinity would be… Well, no. Number infinity minus one would be "I bought your book". It's not as good as "I read your book", you know. Then there is "I bought your software"; that was not in my own personal value. So that decision came up. I kept up with the literature about compilers. The Communications of the ACM was where the action was. I also worked with people on trying to debug the ALGOL language, which had problems with it. I published a few papers, like "The Remaining Trouble Spots in ALGOL 60" was one of the papers that I worked on. I chaired a committee called "Smallgol" which was to find a subset of ALGOL that would work on small computers. I was active in programming languages.

EF: Was McCarthy on Smallgol? DK: No.

No, I don't think he was. EF: Or Klaus

Wirth?

DK: No. There was a big European group, but this was mostly Americans. Gosh, I can't remember. We had about 20 people as co-authors of the paper. It was Smallgol 61? I don't know. It was so long ago I can't remember. But all the authors are there.

EF: You were still a graduate student.

DK: I was a graduate student, yeah. But this was my computing life.

EF: What did your thesis advisors think of all this?

DK: Oh, at Case they thought it was terrible that I even touched computers. The math professor said, "Don't dirty your hands with that."

EF: You mean Cal Tech.

DK: No, first at Case. Cal Tech was one of the few graduate schools that did not have that opinion, that I shouldn't touch computers. I went to Cal Tech because they had this [strength] in combinatorics. Their computing system was incredibly arcane, and it was terrible. I couldn't run any programs at Cal Tech. I mean, I would have to use punched paper tape. They didn't even have punch cards, and their computing system was horrible unless you went to JPL, Jet Propulsion Laboratory, which was quite a bit off campus. There you would have to submit a job and then come back a day later. You couldn't touch the machines or anything. It was just hopeless. At Burroughs I could go into what they called the fishbowl, which was the demonstration computer room, and I could run hands-on every night, and get work done. There was a program that I had debugged one night at Burroughs that was solving a problem that Marshall Hall, my thesis advisor, was interested in. It took more memory than the Burroughs machine had, so I had to run it at JPL. Well, eight months later I had gotten the output from JPL and I had also accumulated the listings that were 10 feet high in my office, because it's a one- or two-day turnaround time and then they give you a memory dump at the end of the run. Then you can say, "Oh, I'll change this and I'll try another thing tomorrow." It was incredibly inefficient, brain damaged computing at Cal Tech in the early '60s. But I kept track with the programming languages community and I became editor of the programming languages section of the Communications of the ACM and the Journal of the ACM in, I don't know, '64, '65, something like that. I was not a graduate student, but I was just out of graduate school in the '60s. That was definitely the part of computing that I did by far the most in, in those days. Computing was divided into three categories. By the time I came to Stanford, you were either a numerical analyst, or artificial intelligence, or programming language person. We had three qualifying exams and there was a tripartite division of the field.

EF: Don, just before we leave your thesis advisor: your thesis itself was in mathematics, not in computing, right?

DK: Yes.

EF: Tell us a little bit about that and what your thesis advisor's influence on your work was at the time.

DK: Yeah, because this is combinatorial, and it's definitely an important part of the story. Combinatorics was not a academic subject at Case. Cal Tech was one of the few places that had it as a graduate course, and there were textbooks that began to be written. I believe at Stanford, for example, George Danzig introduced the first class in combinatorics probably about 1970. It was something that was low on the totem pole in the mathematics world in those days. The high on the totem pole was the Bourbaki school from France, of highly abstract mathematics that was involved with higher orders of infinities and things. I had colleagues at Cal Tech that I would say, "You and I intersect at countable infinity, because I never think of anything that is more than countable infinity, and you never think of anything that is less than countable infinity." I mostly stuck to things that were finite in my own work. At Case, when I'm a senior, we had a visiting professor, R. C. Bose from North Carolina, who was a very inspiring lecturer. He was an extremely charismatic guy, and he had just solved a problem that became front page news in the New York Times. It was to find orthogonal Latin squares. Now, today there is a craze called Sudoku, but I imagine by the time people are watching this tape or listening to this tape that craze will have faded away. An N-by-N Latin square is an arrangement of N letters so ever row and every column has all N of the letters. An orthogonal Latin square is where you have two Latin squares

with the property that if you put them next to each other, so you have a symbol from the first and a symbol from the second, the N squared cells you get have all N squared possibilities. All combinations of A will occur with A somewhere. A will occur with B somewhere. Z will occur with Z somewhere. A famous paper, from 1783, I think, by Leonard Euler had conjectured that it was impossible to find orthogonal Latin squares that were 10 by 10, or 14 by 14, or 18 by 18, or 6 by 6 -- all the cases that were twice an odd number. This conjecture was believed for 170 years, and even had been proved three times, but people found holes in the proof. In 1959 R. C. Bose and two other people found that it was wrong, and they constructed Latin squares that were 10 by 10 and 14 by 14. They showed that all those cases where actually it was possible to find orthogonal Latin squares. I met Bose. I was taking a class from him. It was a graduate class, and I was taking graduate classes. He asked me if I could find some 12 by 12 orthogonal Latin squares. It sounded like an interesting program, so I wrote it up and I presented him with the answer the next morning. He was happy and impressed, and we found five mutually orthogonal Latin squares of the order of 12. That became a paper. Some interesting stories about that, that I won't go into it. The main thing is that he was on the cutting edge on this research. I was at an undergraduate place where we had great teaching, but we did not have cutting edge researchers. He could recommend me to graduate school, and he could also tell me Marshall Hall is very good at combinatorics. He gives me a good plug for going to Cal Tech. I had visited California with my parents on summer vacations, and so when I applied to graduate school I applied to Stanford, Berkeley and Cal Tech, and no other places. When I got admitted to Cal Tech, I got admitted to all three. I took Cal Tech because I knew that they had a good combinatorial attitude there, which was not really true at Stanford. In fact, [at] Stanford I wouldn't have been able to study Latin squares at all. While we're at it, I might as well mention that I got fellowships. I got a National Science Foundation Fellowship, Woodrow Wilson Foundation Fellowship, to come to these place, but they all had the requirement that you could not do anything except study as a graduate student. I couldn't be a consultant to Burroughs and also have an NSF fellowship. So I turned down the fellowships. Marshall Hall was then my thesis advisor. He was a world class mathematician, and had done, for a long time, pioneering work in combinatorics. He was my mentor. But it was a funny thing, because I was such in awe of him that when I was in the same room with him I could not think straight. I wouldn't remember my name. I would write down what he was saying, and then I would go back to my office so that I could figure it out. We couldn't do joint research together in the same room.

We could do it back and forth. It was almost like farming my programs out to JPL to be run. But we did collaborate on a few things. The one thing that we did the most on actually never got published, however, because it turned out that it just didn't lead to the solution. He thought he had a way to solve

the Burnside problem in group theory, but it didn't pan out. After we did all the computation I learned a lot in the process, but none of these programs have ever appeared in print or anything. It taught me how to deal with tree structures inside a machine, and I used the techniques in other things over the years. He also was an extremely good advisor, in better ways than I was with my students. He would seem to keep track of me to make sure I was not slipping. When I was working with my own graduate students, I was pretty much in a mode where they would bug me instead of me bugging them. But he would actually

write me notes and say, Don, why don't you do such and such? Now, I chose a thesis topic which was to find a certain kind of what they call block designs. I will just say: symmetric block designs with parameter Lambda equals 2. Anybody could look that up and find out what that means. I don't want to explain it now. At the time I did this, I believe there were six known designs of this form altogether. I had found a new way to look at those designs, and so I thought maybe I'll be able to find infinitely many more such designs. They would be mostly academic interest, although statisticians would justify that they could use them somehow. But mostly, just, do they exist or not? This was the question. Purely intellectual curiosity. That was going to be my thesis topic: to see if I could find lots of these elusive combinatorial patterns. But one morning I was looking at another problem entirely, having to do with finite projective geometry, and I got a listing from a guy at Princeton who had just computed 32 solutions to a problem that I had been looking at with respect to a homework problem in my combinatorics class. He had found that there are 32 solutions of Type A, and 32 solutions of Type B, to this particular problem. I said, hmm, that's interesting, because the 32 solutions of Type A, one of those was a well known construction. The

32 of Type B, nobody had ever found any Type B solutions before for the next higher up case. I remember I had just gotten this listing from Princeton, and I was riding up on the elevator with Olga Todd, one of our professors, and I said, "Mrs. Todd, I think I'm going to have a theorem in an hour. I'm going to look at these two lists of 32 numbers. For every one on this page I am going to find a corresponding one on this page. I am going to psyche out the rule that explains why there happen to be 32 of each kind." Sure enough, an hour later I had seen how to get from each solution on the first page to the solution on the second page. I showed this to Marshall Hall. He said, "Don, that's your thesis. Don't worry on this Lambda equals 2 business. Write this up and get out of here." So that became my thesis. And it is a good thing, because since then only one more design with Lambda equals 2 has been discovered in the history of the world. I might still be working on my thesis if I had stuck to that problem. I felt a little guilty that I had solved my PhD problem in one hour, so I dressed it up with a few other chapters of stuff. The whole thesis is 70 some pages long. I discovered that it is now on the internet, probably for peoples' curiosity, I suppose: what did he write about in those days? But of all the areas of mathematics that I've applied to computer science, I would say the only area that I have never applied to computer science is the one that I did my thesis in. It just was good training for me to exercise my brain cells.

EF: Yeah. In fact for your colleagues, that is kind of a black hole in their knowledge of you and understanding of you, is that thesis.

DK: The thesis, yeah. Well, I was going to say the reason that it is not used anymore is because these designs turn out… Okay, we can construct them with all this pain and careful, deep analysis. But

it turned out later on that if we just work at random, we get even better results. So it was kind of pointless from the point of view of that application, except for certain codes and things like that.

EF: Don, just a footnote to that story. I intended this would come up later in the interview, but it's just so great a point to bring it in. When I've been advising graduate students, I tell them that the really hard part of the thesis is finding the right problem. That's at least half the problem.

DK: Yeah.

EF: And then the other half is just doing it. And that's the easy part of it. So I am not impressed by this one hour. I mean, the hard part went into finding the problem, not in the solving of it. We will get to, of course, the great piece of work that you did on The Art of Computer Programming. But it's always seemed to me that the researching and then writing the text of The Art of Computer Programming was a problem generator for you. The way you and I have expressed it in the past is that you were weaving a fabric and you would encounter holes in the fabric. Those would be the great problems to solve, and that's more than half the work. Once you find the problems you can go get at them. Do you want to comment on that?

DK: Right. Well, yeah. We will probably comment on it more later, too. But I guess one of the blessings and curses of the way I work is that I don't have difficulty thinking of questions. I don't have too much difficulty in the problem generation phase -- what to work on. I have to actively suppress
stimulation so that I'm not working on too many things at once. But you can ask questions that are… The hard thing, for me anyway, is not to find a problem, but to find a good problem. To find a problem that has some juice to it. Something that will not just be isolated to something that happens to be true, but also will be something that will have spin offs. That once you've solved the problem, the techniques are going to apply to many other things, or that this will be a link in a chain to other things. It's not just having a question that needs an answer. It's very easy to… There's a professor; I might as well mention his name, although I don't like to. It would be hard to mention the concept without somebody thinking of his name. His name is [Florentin] Smarandache. I've never met him, but he generates problems by the zillions. I've never seen one of them that I thought any merit in it whatsoever. I mean, you can generate sequences of numbers in various ways. You can cube them and remove the middle digit, or something like this. And say, "Oh, is this prime?", something like that. There's all kinds of ways of defining sequences of numbers or patterns of things and then asking a question about it. But if one of my students say "I want to work on this for a thesis", I would have to say "this problem stinks". So the hard
thing is not to come up with a problem, but to come up with a fruitful problem. Like the famous problem of
Fermat's Last Theorem: can there be A to the N, plus B to the N equals C to the N, for N greater than 2.
It has no applications. So you found A, B and C. It doesn't really matter to anything. But in the course of working on this problem, people discovered beautiful things about mathematical structures that have solved uncountably many practical applications as a spin off. So that's one. My thesis problem that I solved was probably not in that sense, though, extremely interesting either. It answered a question whether there existed projective geometries of certain orders that weren't symmetrical. All the cases that people had ever thought of were symmetrical, and I thought of unsymmetrical ways to do it. Well, so what? But the technique that I used for it led to some insight and got around some other blocks that people had in other theory. I have to worry about not getting bogged down in every question that I think of, because otherwise I can't move on and get anything out the door.

EF: Don, we've gotten a little mixed up between the finishing of your thesis and your assistant professorship at Caltech, but it doesn't matter. Around this time there was the embryonic beginnings of a multi-volume work which you're known for, "The Art of Computer Programming." Could you tell us the story about the beginning? Because soon it's going to be the middle of it, you were working on it so fast.

DK: This is, of course, really the story of my life, because I hope to live long enough to finish it. But I may not, because it's turned out to be such a huge project. I got married in the summer of 1961, after my first year of graduate school. My wife finished college, and I could use the money I had made -- the

$5000 on the compiler -- to finance a trip to Europe for our honeymoon. We had four months of wedded bliss in Southern California, and then a man from Addison-Wesley came to visit me and said "Don, we would like you to write a book about how to write compilers." The more I thought about it, I decided “Oh yes, I've got this book inside of me.” I sketched out that day -- I still have the sheet of tablet paper on which I wrote -- I sketched out 12 chapters that I thought ought to be in such a book. I told Jill, my wife, "I think I'm going to write a book." As I say, we had four months of bliss, because the rest of our marriage has all been devoted to this book. Well, we still have had happiness. But really, I wake up every morning and I still haven't finished the book. So I try to -- I have to -- organize the rest of my life around this, as one main unifying theme. The book was supposed to be about how to write a compiler. They had heard about me from one of their editorial advisors, that I knew something about how to do this. The idea

appealed to me for two main reasons. One is that I did enjoy writing. In high school I had been editor of the weekly paper. In college I was editor of the science magazine, and I worked on the campus paper as copy editor. And, as I told you, I wrote the manual for that compiler that we wrote. I enjoyed writing, number one. Also, Addison-Wesley was the people who were asking me to do this book; my favorite textbooks had been published by Addison Wesley. They had done the books that I loved the most as a student. For them to come to me and say, ”Would you write a book for us?", and here I am just a second- year gradate student -- this was a thrill. Another very important reason at the time was that I knew that there was a great need for a book about compilers, because there were a lot of people who even in 1962

-- this was January of 1962 -- were starting to rediscover the wheel. The knowledge was out there, but it hadn't been explained. The people who had discovered it, though, were scattered all over the world and they didn't know of each other's work either, very much. I had been following it. Everybody I could think of who could write a book about compilers, as far as I could see, they would only give a piece of the

fabric. They would slant it to their own view of it. There might be four people who could write about it, but they would write four different books. I could present all four of their viewpoints in what I would think was

a balanced way, without any axe to grind, without slanting it towards something that I thought would be misleading to the compiler writer for the future. I considered myself as a journalist, essentially. I could be the expositor, the tech writer, that could do the job that was needed in order to take the work of these brilliant people and make it accessible to the world. That was my motivation. Now, I didn’t have much time to spend on it then, I just had this page of paper with 12 chapter headings on it. That's all I could do while I'm a consultant at Burroughs and doing my graduate work. I signed a contract, but they said "We know it'll take you a while." I didn't really begin to have much time to work on it until 1963, my third year

of graduate school, as I'm already finishing up on my thesis. In the summer of '62, I guess I should mention, I wrote another compiler. This was for Univac; it was a FORTRAN compiler. I spent the summer, I sold my soul to the devil, I guess you say, for three months in the summer of 1962 to write a FORTRAN compiler. I believe that the salary for that was $15,000, which was much more than an assistant professor. I think assistant professors were getting eight or nine thousand in those days.

EF: Well, when I started in 1960 at [University of California] Berkeley, I was getting $7,600 for the nine-month year.

DK: Yeah, so you see it. I got $15,000 for a summer job in 1962 writing a FORTRAN compiler. One day during that summer I was writing the part of the compiler that looks up identifiers in a hash table. The method that we used is called linear probing. Basically you take the variable name that you want to look up, you scramble it, like you square it or something like this, and that gives you a number between one and, well in those days it would have been between 1 and 1000, and then you look there. If you find it, good; if you don't find it, go to the next place and keep on going until you either get to an empty place, or you find the number you're looking for. It's called linear probing. There was a rumor that one of Professor Feller's students at Princeton had tried to figure out how fast linear probing works and was unable to succeed. This was a new thing for me. It was a case where I was doing programming, but I also had a mathematical problem that would go into my other [job]. My winter job was being a math student, my summer job was writing compilers. There was no mix. These worlds did not intersect at all in my life at that point. So I spent one day during the summer while writing the compiler looking at the mathematics of how fast does linear probing work. I got lucky, and I solved the problem. I figured out some math, and I kept two or three sheets of paper with me and I typed it up. ["Notes on 'Open' Addressing', 7/22/63] I guess that's on the internet now, because this became really the genesis of my main research work, which developed not to be working on compilers, but to be working on what they call analysis of algorithms, which is, have a computer method and find out how good is it quantitatively. I can say, if I got so many things to look up in the table, how long is linear probing going to take. It dawned on me that this was just one of many algorithms that would be important, and each one would lead to a fascinating mathematical problem. This was easily a good lifetime source of rich problems to work on. Here I am then, in the middle of 1962, writing this FORTRAN compiler, and I had one day to do the research and mathematics that changed my life for my future research trends. But now I've gotten off the topic of what your original question was.

EF: We were talking about sort of the.. You talked about the embryo of The Art of Computing. The compiler book morphed into The Art of Computer Programming, which became a seven-volume plan.

DK: Exactly. Anyway, I'm working on a compiler and I'm thinking about this. But now I'm starting, after I finish this summer job, then I began to do things that were going to be relating to the book. One of the things I knew I had to have in the book was an artificial machine, because I'm writing a compiler book but machines are changing faster than I can write books. I have to have a machine that I'm totally in control of. I invented this machine called MIX, which was typical of the computers of 1962. In 1963 I wrote a simulator for MIX so that I could write sample programs for it, and I taught a class at Caltech on how to write programs in assembly language for this hypothetical computer. Then I started writing the parts that dealt with sorting problems and searching problems, like the linear probing idea. I began to write those parts, which are part of a compiler, of the book. I had several hundred pages of notes gathering for those chapters for The Art of Computer Programming. Before I graduated, I've already done quite a bit of writing on The Art of Computer Programming. I met George Forsythe about this time. George was the man who inspired both of us [Knuth and Feigenbaum] to come to Stanford during the '60s. George came down to Southern California for a talk, and he said, "Come up to Stanford. How about joining our faculty?" I said "Oh no, I can't do that. I just got married, and I've got to finish this book first." I said, "I think I'll finish the book next year, and then I can come up [and] start thinking about the rest of my life, but I want to get my book done before my son is born." Well, John is now 40-some years old and I'm not done with the book. Part of my lack of expertise is any good estimation procedure as to how long projects are going to take. I way underestimated how much needed to be written about in this book. Anyway, I started writing the manuscript, and I went merrily along writing pages of things that I thought really needed to be said. Of course, it didn't take long before I had started to discover a few things of my own that weren't in any of the existing literature. I did have an axe to grind. The message that I was presenting was in fact not going to be unbiased at all. It was going to be based on my own particular slant on stuff, and that original reason for why I should write the book became impossible to sustain. But the fact that I had worked on linear probing and solved the problem gave me a new unifying theme for the book. I was going to base it around this idea of analyzing algorithms, and have some quantitative ideas about how good methods were. Not just that they worked, but that they worked well: this method worked 3 times better than this method, or 3.1 times better than this method. Also, at this time I was learning mathematical techniques that I had never been taught in school. I found they were out there, but they just hadn't been emphasized openly, about how to solve problems of this kind. So my book would also present a different kind of mathematics than was common in the curriculum at the time, that was very relevant to analysis of algorithm. I went to the publishers, I went to Addison Wesley, and said "How about changing the title of the book from 'The Art of Computer Programming' to 'The Analysis of Algorithms'." They said that will never sell; their focus group couldn't buy that one. I'm glad they stuck to the original title, although I'm also glad to see that several books have now come out called "The Analysis of Algorithms", 20 years down the line. But in those days, The Art of Computer Programming was very important because I'm thinking of the aesthetical: the whole question of writing programs as something that has artistic aspects in all senses of the word. The one idea is "art" which means artificial, and the other "art" means fine art. All these are long stories, but I've got to cover it fairly quickly. I've got The Art of Computer Programming started out, and I'm working on my 12 chapters. I finish a rough draft of all 12 chapters by, I think it was like 1965. I've got 3,000 pages of notes, including a very good example of what you mentioned about seeing holes in the fabric. One of the most important chapters in the book is parsing: going from somebody's algebraic formula and figuring out the structure of the formula. Just the way I had done in seventh grade finding the structure of English sentences, I had to do this with mathematical sentences. Chapter ten is all about parsing of context-free language, [which] is what we called it at the time. I covered what people had published about context-free languages and parsing. I got to the end of the chapter and I said, well, you can combine these ideas and these ideas, and all of a sudden you get a unifying thing which goes all the way to the limit. These other ideas had sort of gone partway there. They would say "Oh, if a grammar satisfies this condition, I can do it efficiently." "If a grammar satisfies this condition, I can do it efficiently." But now, all of a sudden, I saw there was a way to say I can find the most general condition that can be done efficiently without looking ahead to the end of the sentence. That you could make a decision on the fly, reading from left to right, about the structure of the thing. That was just a natural outgrowth of seeing the different pieces of the fabric that other people had put together, and writing it into a chapter for the first time. But I felt that this general concept, well, I didn't feel that I had surrounded the concept. I knew that I had it, and I could prove it, and I could check it, but I couldn't really intuit it all in my head. I knew it was right, but it was too hard for me, really, to explain it well. So I didn't put in The Art of Computer Programming. I thought it was beyond the scope of my book. Textbooks don't have to cover everything when you get to the harder things; then you have to go to the literature. My idea at that time [is] I'm writing this book and I'm thinking it's going to be published very soon, so any little things I discover and put in the book I didn't bother to write a paper and publish in the journal because I figure it'll be in my book pretty soon anyway. Computer science is changing so fast, my book is bound to be obsolete. It takes a year for it to go through editing, and people drawing the illustrations, and then they have to print it and bind it and so on. I have to be a little bit ahead of the state-of-the-art if my book isn't going to be obsolete when it comes out. So I kept most of the stuff to myself that I had, these little ideas I had been coming up with. But when I got to this idea of left-to-right parsing, I said "Well here's something I don't really understand very well. I'll publish this, let other people figure out what it is, and then they can tell me what I should have said." I published that paper I believe in 1965, at the end of finishing my draft of the chapter, which didn't get as far as that story, LR(k). Well now, textbooks of computer science start with LR(k) and take off from there. But I want to give you an idea of…

EF: Don, for historical reasons, tell the audience where the LR(k) paper was published so they can go look it up.

DK: It was published in the journal called Information and Control, which has now changed its name to Information and Computation. In those days, you can see why they called it Information and Control. It was the journal that had had the best papers on parsing of languages at the time. It's a long paper, and difficult. It's also reprinted in my book "Selected Papers on Computer Languages", with a few corrections to the original. In the original, I drew the trees with the root at the bottom. But everybody draws trees with the root at the top now, so the reprint has trees drawn in a more modern notation. I'm trying to give the flavor of the way things were in 1965. My son was born in the summer of '65, and I finished this work on LR(k) at Christmastime in '65. Then I had, I think, one more chapter to write. But early in '66 I had all 3000 pages of the manuscript ready. I typed chapter one. My idea was, I looked at these pages -- the pages were all hand-written -- and it looked to me like my handwriting, I would guess, that was, I don't know how many words there were on a page. I had chapter one and I typed it and I sent it to the publishers, and they said "Don, what have you written? This book is going to be huge." I had actually written them a letter earlier as I'm working on sorting. I said to the guy who signed me up, I signed a contract with him; by this time, he had been promoted. No, I'm not sure was about this, but anyway, I wrote to him in '63 or '64 saying, "You know, as I'm working on this book on compilers, there's a few things that deserve more complete treatment than a compiler writer needs to know. Do you mind if I add a little bit here?" They said "Sure, Don, go right ahead. Whatever you think is good to write about, do it." Then I send them chapter one a few years later. By this time, I guess the guy's promoted, and he's saying "Oh my goodness, what are we going to do? Did you realize that this book is going to be more than 2,000 pages long?", or something like this. No, I didn't. I had read a lot of books, and I thought I understood about things. I had my typed pages there, and I was figuring five typed pages would go into one page of text. It just looked to me, to my eyes, if I had five typewritten pages -- you know, the letters in a textbook are smaller. But I should have realized that the guys at the publishing house knew something about books too. They told me "No, no, it was one and a half pages of text makes a book [page]." I didn't believe it. So I went back to my calculus book, which was an Addison Wesley book, and it typed it out. Sure enough, they were absolutely right. It took one and a half pages. So I had three times longer. No wonder it had taken me so long to get chapter one done! I'm sitting here with much, much more than I thought I had. Meanwhile computer science hasn't been standing still, and I knew that more still has to be written as I go. I went to Boston, and I happened to catch a glance at some notes that my editor had written to himself for the meeting that we were going to have with his bosses, and one of the comments on there was "Terrific cost bind" or something like that. Publishing houses all have their horror stories about a professor who writes12 volumes about the history of an egg, or something like this, and it never sells, and it just is a terrible thing that they have a contract that they've signed. So they have to figure out how to rescue something out of this situation coming with this monster book. We thought at first we would package it into three volumes instead of one. Then they sent out chapter one to a dozen readers in a focus group, and they got comments on it. Well, the readers liked what they saw in that chapter, and so at least I had some support from them. Then after a few more months we decided to package it. They figured out that of the 12 chapters there were seven of them that would sell, and we could stick the other five in some way that would make a fairly decent seven-volume set. That was what was finally announced in 1966 or something: that it would come out in seven volumes. After typing chapter one I typed chapter two, and so on. I kept working on it. All the time when I'm not teaching my classes at Caltech, I'm typing up my notes and polishing the hand-written notes that I had made from these 3000 pages of rough draft. That sets the scene for the early days of The Art of Computer Programming.

EF: What year are we at now?

DK: What happened is, I'm at Caltech. I'm a math professor. I'm teaching classes in algebra and once in a while combinatorics at Caltech. Also one or two classes connected with computing, like sorting, I think I might have taught one quarter. But most of the things I'm teaching at Caltech are orthogonal to The Art of Computer Programming. My daughter is born in December of '66. I've got the entire manuscript of volume one to the publisher, I think, during '66. I'm working on typing up chapters three and four at the beginning of '67. I think this is approximately the way things stand. I was trying to finish the book before my son was born in '65, and what happened is that I got… I'm seeing now that…Volume one actually turned out to be almost 700 pages, which means 1,000 type-written pages. You can see why I said that my blissful marriage wasn't quite so blissful, because I'm working on this a lot. I'm doing most of it actually watching the late late show on television. I have also some earplugs for when the kids are screaming a little bit too much. Here I am, typing The Art of Computer Programming when the babies are crying, although I did also change diapers and so on.

EF: I think that what we need to do is talk about… This is December '66, when your daughter was born.

DK: Yeah.

EF: That leads sort of directly into this magical year of 1967, which didn't end so magically. Let's continue on with 1967 in a moment.

DK: Okay.

EF: Don, once you told me that 1967 was your most creative year. I'd like to get into it. You also said you had only a very short time to do your research during that year, and the year didn't end so well for you. Let's talk about that.

DK: Well, it's certainly a pivotal year in my life. You can see in retrospect why I think things were building up to a crisis, because I was just working at high pitch all the time. I think I mentioned I was editor of ACM Communications, and ACM Journal, in the programming languages sections. I took the editorial duties very seriously. A lot of people were submitting papers, and I would write long referee reports in many cases, as well as discussing with referees all the things I had to be doing. I was a consultant to Burroughs on innovative machines. I was consumed with getting The Art of Computer Programming done, and I had children, and being a father, and husband. I would start out every day and I would say "Well, what am I going to accomplish today?" Then I would stay up until I finished it. I used to be able to do this. When I was in high school and I was editor of the paper, I would do an all-nighter every week when the paper came out. I would just go without sleep on those occasions. I was sort of used to working in this mode, where I didn't realize I was punishing my body. We didn't have iPods and things like that, but still I had the TV on. That was enough to kill the boredom while I had to do the typing of a lot of material. Now, in 1967, is when things came to a head. Also, it was time for me to make a career decision. I was getting offers. I think I was offered full professorships at North Carolina in Chapel Hill, and also at Purdue, I think. I had to make a decision as to what I should do. I was promoted to Associate Professor at Caltech surprisingly early. The question is, where should I spend the rest of my life? Should I be a mathematician? Should I be a computer scientist? By this time I had learned that there was actually possible to do mathematical work as a computer scientist. I had analysis of algorithms to do. What would be a permanent home? I visited Stanford. I gave a talk about my left-to-right parsing. I discovered a theorem about it sitting in one of the student dormitories, Stern Hall, the night I gave the lecture. I came up there, I liked George Forsythe very much, I liked the people that I met here very much. I was thinking Stanford would be a nice place, but also there were other places too that I wanted to check out carefully. I was also trying to think about what to do long-term for my permanent home. I don't like to move. My model of my life was going to be that I was going to make one move in my lifetime to a place where I had tenure, and I would stay there forever. I wanted to check all these things out, so I was confronted with this aspect as well. I was signed up to be an ACM lecturer, ACM National Lecture Program, for two or three weeks in February of 1967, which meant that I give a list of three talks. Each ACM chapter or university that wants to have a speaker, they coordinate so that I have a schedule. I go from city to city every day. You probably did the same thing about then.

EF: Yep.

DK: Stanford and Berkeley were on this list, as well as quite a few schools in the east. That was three weeks in February where I was giving talks, about different things about programming languages, mostly. When I'm at Caltech, I've got to be either preparing my class lectures, or typing my book and getting it done. I'm in the middle of typing chapter four at this time, which is the second part of volume two. I'm about, I don't know, one third of the way into volume two. That's why I don't have time to do research. If I get a new idea, if I'm saying "Here's a problem that ought to be solved", when am I going to do it? Maybe on the airplane. As you know, when you're a lecturer every day goes the same way. You get up at your hotel, and you get on the plane. Somebody meets you at noon and you go out to lunch and then they have small talk. They ask you the same questions; "Where are you going to be tomorrow, Don", and so on. You give your lecture in the afternoon, there's a party in the evening, and then you go to your hotel. The next morning you just go off to the next city. After three weeks of this, I got really not very good. I skipped out in one case. There was a snowstorm in Atlanta, so I skipped my talk in Atlanta and I stayed an extra day. I'm trying to give you the flavor of this. But on this trip in February, also, it turned out to be very fruitful because one of my stops was in Cornell, where Peter Wegner was a visiting professor. We went out for a hike that weekend to talk about the main topic in programming languages in those days: how do you define the semantics of a programming language. What's a good way to formalize the meaning of the sentences in that language? When someone writes a string of symbols, we wanted to say exactly what that means, and do it in a way that we can prove interesting results about, and make sure that we've translated it correctly. There were a lot of ideas floating in the air at the time. I had been thinking of how I'm presenting it in The Art of Computer Programming. I said, well, you know, there were two basic ways to do this. One is top down, where you have the context telling you what to do. You start out and you say, "Oh, this is supposed to be a program. What does a program mean?" Then a program tells the things inside the program what they're supposed to mean. The other is bottom up, where you just start with one symbol, this is a number one, and say “this means one”, and then you have a plus sign, and one plus two, and you build up from the bottom, and say “that means three”. So we have a bottom-up version of semantics, and a top-down version of semantics. Peter Wegner says to me "Don, why don't you use both top-down and bottom-up? Have the synthesized attributes from the bottom up and the inherited attributes that come down from the environment." I said "Well, this is obviously impossible. You get into circular reasoning. You can't define something in terms of itself." We were talking about this, and after ten minutes I realized I was shouting to him, because I was realizing that he was absolutely right. You could do it both ways, and define the things in a way that they would not interfere with each other; that certain aspects of the meaning could come from the top, and other aspects from the bottom, and that this actually made a beautiful combination.

EF: Don, we were speaking about semantics of programming languages and you were shouting at Peter Wegner.

DK: I’m shouting at Peter Wegner because it turns out that there’s a beautiful way to combine the top-down and bottom-up approaches simultaneously when you’re defining semantics. This is happening on a weekend as we’re hiking at Cornell in a beautiful park by frozen icicles. I can remember the scene because this was kind of an “aha” moment that doesn’t happen to you very often in your life. People tell me now no one’s allowed in that park in February because it’s too risky that you’re going to slide and hurt yourself. It was when all of a sudden it occurred to me that this might be possible. But I don’t have time to do research. I have to go on and give more lectures. Well, I find myself the next week at Stanford University speaking to the graduate students. I gave one of my regular lectures, and then there was an hour where the students ask questions to the visitor. There was a student there named Susan Graham, who of course turned out to be a very distinguished professor at Berkeley and editor of Transactions on Programming Languages and Systems, and she asked me a question. “Don, how do you think would be a good way to define semantics of programming languages?” In the back of my mind through that week I had been tossing around this idea that Peter and I had talked about the week before. So I said, “Let’s try to sketch out a simple language and try to define its semantics”. On the blackboard, in response to Susan’s questions, we would erase, and try things, and some things wouldn’t work. But for the next 15 or 20 minutes I tried to write down something that I had never written down before, but it was sort of in the back of my mind: how to define a very simple algebraic language and convert it into a very simple machine language which we invented on the spot to be an abstract but very simple computer. Then we would try to write out the formal semantics for this, so that I could write a few lines in this algebraic language, and then we could parse it and see exactly what the semantics would be, which would be the machine language program. Of course there must have been a lot of bugs in it, but this is the way I had to do research at that time. I had a chance while I’m in front of the students to think about the research problem that was just beginning to jell. Who knows how bad it was fouled up, but on the other hand, being a teacher, that’s when you get your thoughts in order best. If you’re only talking to yourself, you don’t organize your thoughts with as much discipline. It probably was also not a bad way to do research. I didn't get a chance to think about it when I got home to Caltech because I'm typing up The Art of Computer Programming when I'm at home, and I'm being an editor, and I'm teaching my classes the rest of the time at Caltech. Then in April I happened to be giving a lecture in Grenoble, and a Frenchman, Louis Bolliet, asked me something about how one might define semantics, in another sort of a bull session in Grenoble in France. That was my second chance to think about this problem, when I was talking with him there. I was stealing time from the other things. That wasn't the only thing going on in '67. I wasn't only thinking of what to do with my future life, and editing journals and so on, I'm also teaching a class at Caltech for sophomores. It's an all year class, sort of an introduction to abstract mathematics. While I was looking at a problem, we had a visitor at Caltech named Trevor-- what's his last name-- Evans, Trevor Evans. He and I were discussing how to work from axioms, and to prove theorems from axioms. This is a basic thing in abstract mathematics. Somebody sets down an axiom, like the associative law; it says that if parentheses "ab" times "c" is equal to "a" times parentheses "bc." That's an axiom. I was looking at other axioms that were sort of random. One of the things I asked my students in the class was, I was trying to teach the sophomores how to do mini research problems. So I gave them axioms which I called the "axioms of a grope." They were supposed to develop "grope theory" -- they were supposed to grope for theorems. Of course the mathematical theory well developed is a "group", which I had been teaching them; axioms of groups. One of them is the associative law. Another axiom of groups is that an element times its inverse is equal to the identity. Another axiom is that the identity times anything, identity times "X", is "X". So groups have axioms. We learned in the class how to derive consequences of these axioms that weren't exactly obvious at the beginning. So I said, okay, let's make a "grope." The axiom for a grope is something like "x" times the quantity "yx" was equal to "y". I give them this axiom, and I say to the class, what can you derive? Can you find all gropes that have five elements? Can you prove any theorems about normal subgroups, or whatever it is? Make up a theory.

As a class we came back in a week saying what theorems could you come up with. We tried to imagine ourselves in the shoes of an inventor of a mathematical theory, starting with axioms. Well, Trevor Evans was there and he showed me how to define what we called the "free grope," which is the set of all… It can be infinite, but you take all strings of letters, all formulas. Is it possible to tell whether one formula can be proved equal to the other formula just by using this one axiom of the grope, "x" times "yx" equals "y"? He showed me a very nice way to solve that problem, because he had been working on word problems in what's called universal algebra, the study of axiom systems. While I was looking at Trevor Evans' solution to this problem -- this problem arose in connection with my teaching of the class -- I looked at Trevor Evans' solution to this problem and I realized that I could develop an actual method that would work with axioms in general, without thinking that a machine could figure out. The machine could start out with the axioms of group theory, and after a small amount of computation, it could come up with a set of 10 consequences of those axioms that would be enough to decide the word problem for free groups. And the machine was doing it. We didn't need a mathematician there to prove, to say, "Oh, now try combining this formula and this formula." With the technique I learned from Trevor Evans, and then with a little extra twist that I put on it, I could set the machine going on axioms and it would automatically know which consequences of these things, which things to plug in, would be potentially fruitful. If we were lucky, like we were in the case of group theory axioms, it would finally get to the end and say, "Now, there's nothing more can be proved. I've got enough. I've got a complete set of reductions. If you apply these reductions and none of them applies, you've got it." It relates to AI techniques of expert systems, in a way. This idea came to me as I'm teaching this basic math class. The students in this class were supposed to do a term paper. In the third quarter, everybody worked on this. One of the best students in the class, Peter Bendix, chose to do his term paper by implementing the algorithm that I had sketched on the blackboard in one of the lectures at that time. So we could do experiments during the spring of '67, trying out a whole bunch of different kinds of axioms and seeing which ones the machine would solve and which ones it would keep spinning and keep generating more and more reductions that seemed to go without limit. We figured out in some cases how we could introduce new axioms that would bring the whole thing back down again. So we're doing a lot of experiments on that kind of thing. I don't have time to sit down at home and work out the theory for it, but I knew it had lots of possibilities. Here I had attribute grammars coming up in February, and these reductions systems coming up in March, and I'm supposed to be grinding out Volume Two of The Art of Computer Programming. The text of volume one had gone to Addison-Wesley the previous year, and the copy editor had sent me back corrections and told me, "Don, this isn't good writing. You've got to change this," and he'd teach me Addison-Wesley house style. The page proofs started coming. I started going through galley proofs, but now it was time to get page proofs for volume one. Volume one was published in January of 1968, but the page proofs started to be available in the spring also.

EF: So it's layer, upon layer, upon layer.

DK: Right. There's a conference in April in Norway on simulation languages; that was another of the things that I'd been working on at Burroughs. We had a language called SOL, Simulation Oriented Language, which was an improvement of the state-of-the-art in systems simulation, in what they called discrete simulation languages. There was an international conference held in Norway by the people who had invented the Simula language, which wasn't very well known. They organized this conference and I went to that, visiting Paris and Grenoble on my way because Maurice Nivat and I had also become friends. His thesis was on theory of context-free grammars, and no one in France would read it. He found a guy in America who would appreciate his work, so he came out and we spent some time together in '66 getting to know each other and talking about context-free grammar research. I visited him in Paris and then I went to Grenoble, and then went to Norway for this conference on simulation languages where I presented a paper about SOL, and learned about Simula, and so on. My parents and Jill's parents are taking care of our kids while we're in Europe during this time in April. I'm scheduled in June to lecture at a summer school in Copenhagen, an international summer school. I'm giving lectures about how to parse, what's called top-down parsing. "LL(k)" is the terminology that developed after these lectures. This was a topic that I did put in my draft of Chapter 10. It was something that I understood well enough that I didn't have to publish it at the time. I gave it for the first time in these lectures in June in Copenhagen. That was a one-week series of lectures with several lectures every day, five days, to be given there. The summer school met for two weeks, and I was supposed to speak in the second week of that summer school. All right. What happened then in May is I had a massive bleeding ulcer, and I was in the hospital. My body gave out. I was just doing all this stuff, and it couldn't take it. I learned about myself. I had a wonderful doctor who showed me his textbook about ulcers. At that time they didn't know that ulcers are related to this bacteria. As far as they were concerned it, was just acid.

EF: Stress.

DK: Yeah. People would get operations so that their stomachs wouldn't produce so much acid, and things like that. Anyway, he showed me his textbook, and his textbook described the typical ulcer patient; what other people call the "Type A" personality. It just described me to a "T", all of the things that were there. I was an automaton, I think, basically. I had all been all my life pretty much a test-taking machine. You know, I saw a goal and I put myself to it, and I worked on it and pushed it through. I didn't say no to people when they said, "Don, can you do this for me?" At this point I saw, I could all of a sudden get to understand, that I had this problem; that I shouldn't try to do the impossible. The doctor, I say he's so wonderful because doctors usually talk down to patients and they keep their secrets to themselves. But here he let me look at this textbook so I could know that he wasn't just telling me something to make me feel good. I had access to anything I wanted to know about my condition. So I wrote a letter to my publisher, framed in black, saying, "I'm not going to be able to get the manuscript of volume two to you this year. I'm sorry. I'm not supposed to work for the next three weeks." In fact, you can tell exactly where this was. I was writing in a part of volume two when the ulcer happened, when it started to burst or whatever. I was working out the answer to a problem about greatest common divisors that goes about in the middle of volume two. It was an exercise where the answer had a lot of cases to it, so it takes about a page and a half to explain the answer. It was a problem that needed to be studied and nobody had studied before, and I was working at it. All of a sudden, bingo. The reason you can find it is if you look in the index to volume two under "brute force," it refers you to a page, an answer page. I was solving this problem by brute force, and so you look at that page, you can see exactly what exercise I was working on. Then I put it away. I only solved half of the exercise before I could work on it again a few weeks later. I went into the hospital. It wasn't too bad, but the blood supply… I took iron pills and got ready. I could still go to Copenhagen to give my lectures in June. However, the first week was supposed to be lectures by Nicholas Wirth, and the second week was supposed to be lectures by me. But Klaus had just gone on an around the world tour with his wife and had come down with dysentery in India and was extremely ill, and had to cancel his lectures. So I was supposed to go on in the first week instead. But I was stealing time so bad, I hadn't really prepared my lectures. I said, oh, I have a week. I'll go to Copenhagen, listen to Klaus and I'll prepare my lectures. I hadn't prepared. So I'm talking about stuff that has never been written down before, never been developed with the students. I get to Copenhagen with one day to prepare for this week of lectures. Well, one thing in Copenhagen, there's wonderful parks all over the city. I sat down under a big tree in one of those parks on the first day, and I thought of enough things to say in my first two lectures. On the second day I gave the lectures, and I sat down under that tree and I worked out the lectures for the next day. These lectures became my paper called "Top-Down Syntax Analysis." That was the story of the first part of June. The second part of June I'm going to a conference in Oxford, one of the first conferences on discrete mathematics. There I'm presenting my paper on the new method that I had discovered, now called the Knuth-Bendix algorithm, about the word problems in universal algebra. After I finished my lectures at Copenhagen I had time to write the paper that I was giving at Oxford the following week. There at Oxford, I meet a lot of other people and get more stimulated about combinatorial research, which I can't do. Come back to Caltech and I'm working as a consultant as well. I resigned from ten editorial boards at this time. No more ACM Journal, no more Communications. I gave up all of the editorships that I was on in order to cut down my work load. I started working again on volume two where I left off at the time of the ulcer, but I would be careful to go to sleep and keep a regular schedule. In the fall I went to a conference in Santa Barbara, a conference on combinatorial mathematics. That was my first chance to be away from Caltech, away from my teaching duties, away from having to type The Art of Computer Programming. That's where I had three days to sit on the beach and develop the theory of attribute grammars, this idea of top-down and bottom–up. I cut out of the whole conference. I didn't go to any of the talks. I just sat on the beach and worked on the theory of attribute grammar. As it turned out, I wasn't that interested in most of the talks, although I met people that became lifelong friends at the meals and we talked about things off-line. But the formal talks themselves, I was getting disappointed with mathematical talks. I found myself, in most lectures on mathematics that I heard in 1966 and '67, I sat in the back row and I said, "So what? So what?" Computer science was becoming much more exciting to me. When I finally made my career decision as to where to go, I had four main choices. One was stay at Caltech. They offered me full professor of mathematics. I could go to Harvard as a full professor in applied science, which meant computer science. That was as close as you could get to computer science there. At Harvard my job would have been to build up a computer science department there. Harvard was, in Floyd's term, an advanced backwater at that point in time for computer science, and Caltech was as well. Because Caltech and Harvard are so good at physics and chemistry and biology, they were thinking of computers because they can help chemists and physicists and biologists. They didn't think of it as having problems of its own interest. Stanford, where we had the best group of computer scientists in the world already there, and knowing that computer science had a great future, and also the best students in the world were there to work with, the program was already built up. I could come to Stanford and be one of the boys and do computer science, instead of argue for computer science and try to do barnstorming. Berkeley was the fourth place. I admired Berkeley very much as probably the greatest all around institution for covering everything. Everything Stanford covered it covered well, but it didn't have a professor of Sanskrit, and Berkeley had a professor of Sanskrit, that sort of thing. But I was worried about Berkeley because Ronald Reagan was governor. Stanford was a private school and wouldn't be subject to the whims of politicians so much as the University of California. Stanford had this great other thing where the faculty can live on campus, so I knew that I could come to Stanford and the rest of my life I would be able to bike to work; I wouldn't have to do any commuting. And Forsythe was a wonderful person, and all the group at Stanford were great, and the students were the best. So it was almost a no-brainer, why I finally came to Stanford. My offer from Stanford came through in February of '68, which was the end. The other three had already come in earlier, but I was waiting for Stanford before I made my final decision. In February of '68 I finally got the offer from Stanford. It was a month after volume one had been published, and George said, "Oh yes, everybody's all smiles now."

EF: Everyone was all smiles because they had gone out on a limb to offer you a full professorship?

DK: No, because the committees were saying, "This guy is just 30 years old." You know, I was born in '38 and this was January of '68. But when they looked at the book, they said, "Oh, there's some credibility here." That helped me. I got through '67 and learned how to slack off a little bit, right? I've always felt after that, hearing many other stories of people of when did they get these special insights that turned out to be important in their research thing, that was very rarely in a settled time of their life, where they had a comfortable living conditions and good – the word is escaping me now - but anyway, luxury; set up a nice office space and good lighting and so forth. No, people are working in a garret, they're starving, they've got kids screaming, there's a war going on or something. But that's when they get a lot of their most… almost every breakthrough idea. I've always wondered, if you wanted to set up a think tank where you were going to get the most productivity out of your scientists, wouldn't you have to, not exactly torture them, but deprive them of things? It's not sustainable. Still, looking back, that was a time when I did as much science as I could, as well as try to fulfill all my other obligations.

EF: Don, to go back to the Stanford move. A couple of questions come up, because I was around. I remember sitting in George Forsythe's office, just a handful of us people considering the appointment of this young guy from Caltech who had this wonderful outline of books. One of the things that we were discussing was [that] Don Knuth wanted us to also hire Bob Floyd. It turns out that hiring Bob Floyd was a wonderful idea. Bob Floyd was magnificent. But it hadn't occurred to us until you brought it up, and then we did it. Can you go into that story?DK: Yeah, because Bob was a very special person to me throughout this period. As I said, I'd been reading the literature about programming languages avidly. When I was asked to write a book about it in '62, I knew there were these people who had written nice papers, but nobody knew how to sort out the chaff from the wheat. In the early days, like by 1964, my strong opinion was that five good papers about programming languages had ever been written, and four of them were by Bob Floyd. I met Bob the first time in summer of '62 when I was working on this Fortran compiler for Univac. At the end of the summer I went to the ACM conference in Syracuse, New York, and Bob was there. We hit it off very well right away. He was showing me his strange idea that you could prove a computer program correct, something that had never occurred to me. I said I was a programmer in one room, and I was a mathematician in another room. Mathematicians prove things. Programmers write code and they hope it works, and they twiddle it until it works. But Bob's saying, no, you don't have to twiddle; you can take a program and you can give a mathematical proof that it works. He was way ahead of me. There were very few people who had ever conceived of putting those two worlds together at that time. EF: [John] McCarthy was one of them, though.DK: McCarthy, exactly, right. John and Bob were probably… I don't know if there was anybody in Europe yet who had seen this right. Bob tells me his thoughts about this when I meet him in this conference in Syracuse. Then I went to visit him a year later when I was in Massachusetts at the crisis meeting with my publishers. He lived there, and I went and spent a couple of days in Topsfield where he lived. We shared ideas about sorting. Then we had a really exciting correspondence over the next time where letters go back and forth, each one trying to trump the other about coming up with a better idea about something that's now called sorting networks. Bob and I developed a theory of sorting networks between us in the correspondence. We were thinking at the time, this looks like Leibnitz writing to Bernoulli in the old days of scientists trying to develop a new theory. We had a very exciting time working on these letters. Every time I would send a letter off to Bob, thinking, "Okay, now this is the last result," he would come back with a brand new idea and make me work harder to come up with the next step in our development of this theory. We weren't talking only about programming languages; we were talking also about a variety of algorithms. We found that we had lots of common interests. He came out to visit me a couple of times in California, and I visited him. So when I was making my career decision, I said, "Hey Bob, wouldn't it be nice if we could both end up at the same place?" I wrote him a letter, probably the same letter where I was describing to him my idea about left-to-right parsing. As soon as I discovered it, I wrote immediately to Bob a 12-page letter with ideas of left-to-right parsing after I had come up with the idea. He comes back and says, "Oh, bravo, and did you think about this," and so on. So we had this going on. Then at the beginning of '67 I said, "You know, Bob, why don't we think about trying to get into the same place together? What is your take on the different places in the world?" At that time he was at Carnegie. He had left Computer Associates and spent, I think, two years at Carnegie. He was enjoying it there, and he was teaching and introducing new things into the curriculum there. He wrote me this letter assessing all of the schools at the time, the way he thought their development of computer science was. When I quoted him a minute ago saying Harvard was an advanced backwater, that comes out of that letter that he was describing the way he looked at things. At the end of the letter he says -- I had already mentioned that Stanford was my current number one but I wasn't totally sure -- and at the end he ended up concurring. He said if I would go there and he could go there, chances are he would go there, too. I presented this to Forsythe, saying why don't we try to make it a package deal. This meant they had to give up two professors to replace us with. They couldn't get two new billets for us, and so it was a lot of work on Stanford's part, but it did develop. Except that you had to lose two other good people, but I think Bob and I did all right for the department. EF: Maybe that was your first great service to our department was recruiting Bob Floyd.

DK: Well, I don't know. I did have to work a little bit the year after I got here. To my surprise they had appointed him as an associate professor but me as a full professor. It was understandable because he didn't have a Ph.D. He had been a child prodigy, and I think he had gotten into graduate school at something like age 17, and then dropped out to become a full time programmer. So he didn't have the academic credentials, although he had all the best papers in the field. I had to meet with the provost and say it's time to promote him to full professor. The thing that clinched it was that he was the only person that had gotten -- this was 1969 -- he was the only person that had been invited to give keynote addresses in two sessions of the International Congress in Ljubljana.

EF: In '71.

DK: '71, yeah. That helped.

EF: That was IFIPS.

DK: Yeah, IFIPS: Information Processing.

EF: Don, maybe we could just say a little more about Bob [Floyd] and his life at Sanford.

DK: Right. As it turned out, when we got together we couldn't collaborate quite as well as when we were writing letters. I noticed this was true in other cases. Like sometimes I could advise my students better when I was on sabbatical than when we were having weekly meetings. It's not easy to work face- to-face all the time, but rather sometimes offline instead of online. I told you my experience with Marshall Hall -- that I couldn't think in his presence. I have to confess that there are some women computer scientists that when I'm in their presence, I think only of their brown eyes. I love their research, but I'm wired in certain ways that mean that we should write our joint papers by mail, or by fe-mail. Anyway. We did a lot of joint work in the early '70s, but also it turned out that when Bob became chair of the department… I'm not sure exactly when that was; probably right after my sabbatical.

EF: I think 1972.

DK: Yeah. I went on leave of absence for a year in Norway and then I came back and Bob was chair of the department. He took that job extremely seriously, and worked on it to such an extent that he couldn't do any research very much at all during those three or four years when he was chair. I don't know how many years, five years.

EF: I started being chair in '76, so like four years.

DK: Okay, so it was four years. That included very detailed planning all aspects of our new building. When he came back, then he had two years of sabbatical. That's one credit that you get. So there was a break in our joint collaboration. Afterwards, he never quite caught up to the leading edge of the same research topics that I was in. We would work on things occasionally, but not at all the way we had done previously. We wrote a paper that we were quite pleased with at the end of the '80s, but it was not the kind of thing that we imagined originally, that would always be in each other's backyard. In fact, I'm a very bad coworker. You can't count on me to do anything, because it takes me a while to finish stuff and I think of something else. So how can anybody rely on me as being able to go with their agenda? Bob, during the ‘70s, came up with a lot of ideas, like his method for half-tone, for making gray-level pictures, that is in all the printers of the world now. That was done completely independently. I didn’t even know about it until a couple years after he had come up with these inventions. But I’m dedicating a book to Bob. My collected works are being published in eight volumes. The seventh volume is selected papers on design of algorithms. That one is dedicated to Bob Floyd, because a lot of the joint papers, joint work we did, occurs in that volume. He was one of the few people in my life that really I consider one of my teachers, the gurus that inspired me.

EF: Don, I’m going to call that the end of your first period of Stanford. I wanted to move into some questions about what I call your second Stanford period. This is very different. I’ve sort of delineated this as a very different time. I saw you shifting gears, and I couldn’t believe what was happening. You became, in a solitary way, the world’s greatest programmer. It was your engineering phase. This was TeX and METAFONT. All of a sudden, you disappeared into just miles of code, and fantastic coding ideas just pouring out, plus your engineering. We were in the new building and you were running back and forth from your office to where this new printing machine was installed. You’d be debugging it with your eyes and with your symbols and pulling your hair out of your head, because it wasn’t working right, and all that. You were just what the National Academy of Engineering would call an engineer. Tell me about that period in your life.

DK: Okay, well, it ties in with several things. There was a year that you didn’t see me when I was up at McCarthy’s lab...

EF: Well, I heard all about it.

DK: …starting this. One of the first papers that I collaborated with Bob Floyd on in 1970 [had] to do with avoiding go-to statements. There was a revolutionary new way to write programs that came along in the ‘70s, called structured programming. It was a different way than we were used to when I had done all my compilers in the ’60s. Bob and I, in a lot of our earliest conversations at Stanford, were saying, “Let’s get on the bandwagon for this. Let’s understand structured programming and do it right.” So one of our first papers was to do what we thought was a better approach to this idea of structured programming than some people had been taking. Some people had misunderstood that if you just get rid of go-to statements you had a structured program. That’s like saying zero population growth; you have a numerical goal, but you don’t change the structure. People were figuring out a way to write programs that were just as messy as before, but without using the word “go-to” in them. We said no, no, no; here’s what the real issues are. Bob and I were working on this. This is going on, and we’re teaching students how to write programs at Stanford, but we had never really written more than textbook code ourselves in this style. Here we are, being full professors, telling people how to do it, having never done it ourselves except in really sterile cases with not any real world constraints. I probably was itching… Thank you for calling me the world’s greatest programmer. I was always calling myself that in my head. I love programming, and so I loved to think that I was doing it as well as anybody. But the fact is, the new way of programming was something that I didn't have time to put much effort into.

EF: The emphasis in my comment was on the solitary. You were a single programmer doing all this. No team.

DK: That's right. As I said, it's hard for me to have somebody else doing the drumming. I had to march to my... I had The Art of Computer Programming, too. I could never be a reliable part of a team that I wasn't the head of, I guess. I did first have to get into that mode, because I was forced to. I was chair of the committee at Stanford for our university reports. We put out lots and lots of reports from all phases of the department through these years. We had a big mailing list. People were also trading their reports with us. We had to have a massive bookkeeping system just to keep the correspondence, so that the secretaries in charge of it could know who had paid for their reports, who we were sharing with. All this administrative type of work had to be done. It seemed like just a small matter of programming to do this. I had a grad student who volunteered to do this as his master's project; to write-up program that would take care of all of the administrative chores of the Stanford tech reports distribution. He turned in his term paper and I looked at it superficially and I gave him an A on it, and he graduated with his master's degree. A week later, the secretary called me up and said, "Don, we're having a little trouble with this program. Can you take a look at it for us?" The program was running up at the AI lab, which I hadn't visited very often. I went up there and took a look at the program. I got to page five of the program and I said, "Hmmm. This is interesting. Let me make a copy of this page. I'm going to show it to my class." [It was] the first time I saw where you change one symbol on the page and you can make the program run 50 times faster. He had misunderstood a sorting algorithm. I thought this was great. Then I turned to the next page and he has a searching algorithm there for binary search. I said, "Oh, he made a very interesting error here. I'll make a copy of this page so I can show my class next time I teach about the wrong way to do binary search." Then I got to page eight or nine, and I realized that the way he had written his program was hopelessly wrong. He had written a program that would only work on the test case that he had used in his report for the master's thesis, that was based on a database of size three or something like this. If you increased the database to four, all the structures would break down. It was the most weird thing. I would never conceive of it in my life. He would assume that the whole database was being maintained by the text editor, and the text editor would generate an index, the way the thing did. Anyway, it was completely hopeless. There was no way to fix the program. I thought I was going to spend the weekend and give it to the secretary on Monday and she could work on it. There was no way. I had to spend a month writing a program that summer -- I think it was probably '75, '76 -- to cover up for my terrible error of giving this guy an A without seeing it. The report that he had, made it look like his program was working. But it only worked on that one case. It was really pathetic. So I said, "Okay, I'll use structured programming. I'll do it right. This is my chance to do structured programming. I'll get a learning experience out of it." I got a good appreciation for writing administrative-type programming. I used to think was trivial, [but] there was a lot to it. After a month I had a structured program that would do Stanford reports, and I could install that and get back to the rest of my life. Meanwhile, I'd been up at the AI lab and I met the people up there. I got to know Leland Smith, who is a great musician professor. Leland Smith told me about a problem that he had. He was typesetting music. He says, "I've got a piece of music and it maybe has 50 bars of music. I have to decide when to turn the page. I know how many notes are in each bar of the music, and I know how much can fit on the page. But I like to have the breaks come out right. Is there any algorithms that could work for this?" He described the problem with me. He had the sequence of numbers, how many notes there are, and try to find a way to break it into lines and pages in a decent way. I looked at the problem and said, "Hey Leland, this is great. It's a nice application of something we in computer science call the dynamic programming algorithm (method). Look, here's how dynamic programming can be used to solve this problem." Then I'm teaching Stanford's problem seminar the next fall, and it came up in class. I would show the students, "Look how we had this music problem, and we can solve it with dynamic programming." One of the students, I don't remember who it was, raised his hand and said, "You know, you could also use that to text, to printing books. You could say, instead of notes into bars, you could also say you've got letters and words into lines, and make paragraphs choosing good line breaks that way." I said, "Hey, that's cool. You're right." Then comes, in the mail, the proof sheets for the second edition of volume two. I had changed a lot of pages in volume two of The Art of Computer Programming. I got page proofs for the new edition. During the '70s, printing technology changed drastically. Printing was done with hot lead in the '60s, but they switched over to using film in the '70s. My whole book had been completely retypeset with a different technology. The new fonts looked terrible! The subscripts were in a different style from the large letters, for example, and the spacing was very bad. You can look at books printed in the early '70s and it turns out that if it wasn't simple -- well, almost everything looked atrocious in those days. I couldn't stand to see my books so ugly. I spent all this time working on it, and you can't be proud of something that looks hopeless. I'm tearing out my hair. I went to Boston again and they said, "Oh, well, we know these people in Poland. They can imitate the fonts that you had in the old hot lead days. It's probably not legal, but we can probably sneak it through without…" You know, the copyright problems of the fonts. "They'll try to do the best they can, and do better". Then they come back to me, at the beginning of '77, with the new version done with these Polish fonts which are supposed to solve the problem. They are just hopelessly bad. At the very same time, February of '77, I'm on Stanford's comprehensive exam committee, and we're deciding what the reading list is going to be for next year's comp. Pat Winston had just come out with a new book on artificial intelligence, and the proofs of it were just being done at III Corporation [Information International, Incorporated] in Southern California; at [Ed] Fredkin's company. They had a new way of typesetting using lasers. All digital, all dots of ink. Instead of photographic images and lenses, they were using algorithms, bits. I looked at these galley proofs of Winston's book. I knew it was just bits, but they looked gorgeous. They looked absolutely as good as anything I'd ever seen printed by any method. By this time I was working at the AI lab, where we had the Xerox Graphics Printer, which did bits at about 120 dots per inch. It looked interesting, but it didn't look beautiful by any stretch of the imagination. Here, with I think this was 1,000 dots per inch at III, you couldn't tell the difference. It was like: I come from Wisconsin and in Wisconsin we never eat margarine. Margarine was illegal to bring into the State of Wisconsin unless you didn't color it. I'm raised on butter. It's the same thing here. With typography, I'm thinking: okay, digital typography would have to be like margarine. It couldn't be the real thing. But, no! Our eyes don't see any difference when you've got enough dots to the inch. A week later, I'm flying down with Les Earnest to Southern California to III, and finding out what's going on there. How can we get this machine and do it? Meanwhile, I planned to have my sabbatical year in '77-'78. I was going to spend my sabbatical year in Chile.

EF: Don, can I interrupt you just a second?

DK: Yeah.

EF: I don't know if Fredkin was still involved with III at that time. But III never gets enough credit for those really revolutionary ideas.

DK: That's right.

EF: Not just those ideas, but the high speed graphics ideas.

DK: Oh yeah. That's when I met Rich Sherpel [ph?] down there, and he was working on character recognition problems. They had been doing it actually for a long time on microfilm, before doing Winston's book. This was the second generation. First they had been using the digital technology at really high resolutions on microfilm. And so many other things [were] going on. Fredkin is a guy who--

EF: Right at the beginning, Fredkin revolutionized film reading, using the PDP-1. Anyway, I interrupted you. You were on your Chile.

DK: Ed's life is ten times as interesting as mine. I'm sure that every time I hear more about Ed, it adds just another… He's an incredible person. We got to get 20 oral histories.

EF: I think Ed may be a subject for one of these oral histories of the Computer History Museum.

DK: Yeah, we've got to do it. Anyway, I cancelled my sabbatical plan for Chile. I wrote to them saying I'm sorry; instead of working on volume four during my sabbatical, I'm going to work on typography. I've got to solve this problem of getting typesetting right. It's only zeros and ones. I can get those dots on the page, and I've got to write this program. That's when I became an engineer.

EF: I'm going to let you go on with this, but I just wanted to ask a question in the middle here, just related to myself, actually. How much of this motivation to do TeX related to your just wanting to get back to being a programmer? Life was going on in too abstract a way, and you wanted to get back to being a programmer and learning what the problems were, or the joy of programming.

DK: It's a very interesting hypothesis, because really you can see that I had this. The way I approached the CS reports problem the year before was an indication of this; that I did want to sink my teeth into something other than a toy problem. It wasn't real large, but it wasn't real small either. It's true that I probably had this craving. But I had a stronger craving to finish volume four. I did sincerely believe that it was only going to take me a year to do it.

EF: Maybe volume four wasn't quite ready. Maybe…

DK: Oh, this is true.

EF: …it was still cooking.

DK: No, no, absolutely. You're absolutely right. In 1975 and '76, you can check it out. Look at the Journal of the ACM. Look at the SIAM Journal on Computing. Look at, well, there's also SIAM Review and there's math journals, combinatorial journals, Communications of the ACM, for that matter. You'll find more than half of those articles are things that belong in volume four. People were discovering things right and left that I knew deserved to be done right in volume four. Volume four is about combinatorial algorithms. Combinatorial algorithms was such a small topic in 1962 when I made that chapter seven of my outline that Johan Dahl asked me, when I was in Norway, "How did you ever think of putting in a chapter about combinatorial algorithms in 1962?" I said, "Well, the only reason was, that was the part I thought was most fun." I really enjoy writing, like this program for Bose that I did overnight. It was a combinatorial program. So I had to have this chapter just for fun. But there was almost nothing known about it at the time. People will talk about combinatorial algorithms nowadays [and] they usually use "combinatorial" in a negative way. In a pejorative sense, instead of the way I look at it. They say, "Oh, the combinatorial is going to kill you." "Combinatorial" means "It's exploding. you can't handle it, it's a huge problem." The way I look at it is, combinatorial means this is where you've got to use some art. You've got to be really skillful, because one good idea can save you six orders of magnitude and make your program run a million times faster. People are coming up with these ideas all the time. For me, the combinatorial explosion was the explosion of research. Not the problems exploding, but the ideas were exploding. So there's that much more to cover. It's true that I also in the back of my mind I'm scared stiff that I can't write volume four anymore. So maybe I'm waiting for it to simmer down. Somebody did say to me once, after I solved the problem of typesetting, maybe I would start to look at binding or something, because I had to have some other reason [to delay]. I've certainly seen enough graduate student procrastinators in my life. Maybe I was in denial.

EF: Anyway, you headed into this major engineering problem.

DK: As far as I knew, though, it was going to take me a year. I was going to work and I was going to enjoy having a year of writing this kind of a program. The program was going to be just for me and my secretary, Phyllis; my super-secretary, Phyllis. I was going to teach her how to do it. She loved to do technical typing. I could write my books and she could make them; dotting I's and crossing T's and spit and polish that she did on my math papers when she always typed my math papers.

Session 2: March 21, 2007

EF: My name is Edward Feigenbaum. I am a Professor of Computer Science Emeritus at Stanford University and a colleague of Donald Knuth's since the day that he showed up at Stanford. This is session No. 2 of an oral history in which Don has been discussing his early work and what we call the first Stanford period. We're now about to go into what we're calling the second Stanford period, in which he discusses his work on typesetting, printing, and font design, TeX and Metafont. But Don, before we start on that, I want to mention in the week since we met for the first session, the news came out that John Backus, the leader of the team that developed Fortran for IBM, died. John's work intersects to some degree with the work that you spoke about last week in the first part of the interview, the work that Alan Perlis did at Carnegie Mellon, at that time Carnegie Tech, on IT, Internal Translator, and the work that you did on RUNCIBLE. I happened to be around both places at that time. I was a summer student working for IBM at the time that the Fortran group was working in New York, and interacted with them. Then I came back to Carnegie Tech and was startled, actually, seeing Perlis's compiler up and running on the [IBM] 650. It looked to me as if Perlis's work was up and running somewhere around 6 months before, maybe 9 months before, Fortran was running on the IBM 704 in New York. The question for you is: is priority important in computer science research as it is in many other disciplines, like chemistry or physics? Is there a priority discussion to be had here on the question of Fortran and IT?

DK: Okay, those are great questions. It's funny you'd ask that, because the number one thing on my mind as I was walking into the building this morning was thinking about John Backus's death. It was a shock to me to learn about it yesterday, but then I was just thinking, "Oh yeah, he always wore clothes like this." [Motions to himself] Whenever I saw him he was wearing a denim jacket. I can say just a few orthogonal things about the whole situation that strike me first. In the first place, when I was a student we had no information at all about Fortran. I didn't hear about it until after I had been using IT, and I think after RUNCIBLE. It was, like, 1959 when people were coming out with something called FORTRANSIT, which was a translator from Fortran to IT, to IT, so that people on the 650 could use the Fortran language. You see, the IBM 650 was the world's first mass-produced computer, the first time there were more than 100 of any one kind of computer. Fortran was developed for a 704, which there were several dozen of those, but it was aircraft industries and so on. It was people who could afford a much bigger kind of machine than the 650. It was a different world. You were lucky as a summer student. You could see the other world, but I was more in the boondocks. I told you last time that priority was so far from my mind when I wrote this article about RUNCIBLE for the Communications [of the ACM] that I failed to mention any of the people who were working with me. We had this team at Case, but we didn't name any names in our story as to who came up with the improvements that we made, because it wasn't something that we knew anything about. But that might have been my naiveté as a college undergraduate. The first time I learned about upsmanship or something -- academic priority -- was, in fact, from my teacher Bose, the man who worked on Latin squares. The reason he wanted me to get this program working overnight is because he was in intense competition with another group in

Canada that was also trying to find Latin squares of order 12. It turned out it was approximately a dead heat between the two groups. That amazed me, that there could be so much competition for being first at the time. It wasn't part of the culture that I grew up in. All I can report is that I was amazed later on to find out also, when people are talking about the discovery of DNA and all this, how much passion went into these things, because it just wasn't something that I personally experienced. But it just might be, again, my naiveté. I presented a paper at the ACM Conference in 1962 in Syracuse, which was the summer that I wrote my Fortran compiler for the Univac solid-state machine. At the end of that summer I gave this paper at the ACM called, "A History of Writing Compilers." I guess I didn't call it "The History of Writing Compilers." Basically I was trying to explain in my talk what I knew about the various developments that had come up with in technology for writing compilers. Of course I mentioned Fortran, and the ways in which they had dealt with the question, for example, of operator precedence. That was, if you write AxB+C without parentheses, Fortran would recognize that as first multiply A by B, and then add C. I'll go back and give you a better example. AxB+CxD, but you write that without any parentheses. Now what would happen in Fortran, is Fortran would know that the mathematicians usually mean by that that you take A and multiply it by B, and you take C and multiply by D, and then you add the two things together. But IT wouldn't do it that way. IT would require you to put parentheses if you want to do it, and, if my memory is correct, otherwise it would associate to the right. So it would take A times the quantity B plus the quantity C times D. So Fortran had to invent a way to do this. The way they did it was rather clever. They replaced the times sign by right parenthesis times left parenthesis, and they replaced the plus sign by two right parentheses plus two left parentheses. Then they put a whole bunch of parentheses around the whole thing. The result is that you had an expression that was fully parenthesized, but since you had guarded the plus sign with two parentheses and the times with only one, the times was done first. It was a clever idea. It's just one of the things I mention in my paper in this Syracuse thing. Well, a reviewer of my paper afterwards said, "He didn't talk about the history of writing compilers. He just talked about the history of him writing one particular compiler." Well, if you look at my paper you'll see it's not a fair criticism. The reviewer was undoubtedly tee'd off that I had not mentioned his compiler. I gave a history of many ideas that were used in building compilers, but I didn't give a history of what people had done in compilers. As years went on, I got more interested in history. In 1962 I was 24 years old. It's like Mark Twain or somebody said, that "When you're a teenager you think your parents are the stupidest people in the world. Five years later you wonder how they could learn so much in five years." You get more interested in history and the overall thing. Well anyway, this criticism, that I hadn't given a very comprehensive history of compilers, weighed on my mind. So the next few years after '62, I actually started looking into the history of compilers, trying to get a real understanding as to who did what when, and first, and so on. Where did the ideas come from before the little excrescences of the story that I knew. In fact, the main lecture I was giving on my ACM lecture tour in 1967 was the real early history of writing compilers. By that time I had gone through and I had studied Grace Hopper's work, and I had studied Backus's work, and the Fortran 0, and the many developments in England and Russia and so on, that had taken place in the earliest days. So the talk that I was giving when I'm making this nationwide lecture tour is mostly this talk of redeeming myself for giving a very unbalanced view of the history of compiler development that I had given in 1962. Later on I worked with my student Luis Trabb Pardo in order to really do it right, because the Harvard University Press had asked me to edit a sourcebook on computer science which was intended to print the documents from the early days that had come out before their time. I had collected a lot of these early things. They've done this with many other fields: sourcebook on logic, sourcebook on mathematics, sourcebook on chemistry, and that kind of thing. Harvard had a big series. I was asked to do a sourcebook on computer science. In the course of this I worked with Luis to get a really thorough history of programming languages, their early development. We presented this as a paper at a big conference in Los Alamos in 1976. 1976 was the year everybody had history on mind, because it was the bicentennial of America. We had a big conference where almost all the computer pioneers were living were assembled there. People like [Konrad] Zuse came, who I met from Europe, and the people who had worked on the Colossus computers. All these pioneers were there. The paper that I presented at that time was "The Early History of Computer Languages." This was one of the most difficult papers to write, in the sense of total amount of work expended, because what I presented in this talk was 20 predecessors to Fortran. Not only was Fortran not number 1, but it was number 21, basically. Although one of the 20 preceding Fortran was the preliminary specs of Fortran, which wasn't implemented, but people were using it in mockups and trial runs. Going to Zuse's work, for example, Zuse had a high level language, his PlanKalkul. Many, many other pioneers [attended]. I brought that picture all together. I'm quite proud of the paper now, because of all the work I put into it. As I was writing it I found out actually I only had 19 predecessors of Fortran. Just a week before the conference I learned about another one at Livermore that had been developed. I went out to Livermore, and right in my own backyard was one of the first. So there was a great amount of activity going on. IT was part of this, for sure. But most of the people didn't know [of] the existence of the others. Fortran itself was strongly influenced by a compiler for the Whirlwind computer that John Backus learned about when he went to a conference at MIT in 1954. Then John got his team together and did that. I had a great admiration for John. I remember that the first time I came to Stanford, which was about 1964, was when I first met him. We had corresponded. He and Barbara invited me to their house, and he also introduced me to topless bars at the time. It was interesting as a nice phenomenon in San Francisco, you know. We had a pleasant evening together. That was on the same trip that I visited Stanford at Forsythe's invitation. John and I always hit it off well, and I admired his breadth of interest in all these things. But your question was mostly about priority. I think it cuts two ways. In the first place, I don't like to think of it as saying somebody did it before somebody else. That's the popular interpretation of the priority. But the opposite is where you just have an idea and you have no idea where it came from.

That's very bad, I think, just to assume that ideas have no connection to each other or they didn't spring from somewhere. Because how are we going to get another idea tomorrow if we don't have a lot of case studies as to how ideas can germinate? I go out of my way in my books The Art of Computer Programming to try to track down the sources of the concepts that we have in computer science. Sometimes I tell people I only do this in order to make computer science respectable, to show that it's not a fly-by-night thing, but it's deeply rooted in ancient history and so on. Well, of course, it's nice to have computer science a little bit respectable. We are the new kid on the block. But that's not really the point. The point is that really there were people who would've been computer scientists, if computers had been around, that were living a hundred years ago. They just happened to have been born at the wrong time, but they had the same kind of strange way of looking at things that I do. I can see that in their writings. I was reading last year a manuscript from 14th century India, and I felt the guy was talking to me. I doubt if any of his contemporaries really knew what he was, but here it was. I said to my wife, "This guy is a computer scientist. I know exactly what's going on because I went through the same kind of a thought process when I was looking at a similar problem when I was younger." So the idea of priority is more, instead, really learning the human element of it. How somebody was able to combine ideas and then make a non-obvious leap that would then influence somebody else. For this reason I love to read source documents instead of [reading] somebody boiling down a source document. I boil it down myself in my books. I try to recommend that. I try to give places so that people can check out the originals when they can. We'd have much less of this cutthroat idea of competition in the field than I read about when I study the novel by my friend who invented the birth control pill.

EF: Carl Djerassi.

DK: Yeah, Carl Djerassi's novel, "The Bourbaki Gambit". Or something like this, right? It's all about a world of science that I don't feel computer science inhabits. It's a different…

EF: Yeah, Carl's a chemist.

DK: Yeah.

EF: I was going to bring a quote from a chemist to this interview but I didn't have it exactly right so I didn't do it. But in chemistry, the knife is sharp.

DK: Yeah. I worked on open source publishing a few years ago, and I was surprised to find out that… I was looking at some of the general policy, and in other fields than computer science when you submit your paper, you can list people that you don't want to be referees of your paper. It blew my mind.

I said, "Why do you do this?" and he said, "Well, because they think the other guys are going to steal their ideas." <laughs> We share ideas. The whole Silicon Valley culture, the venture capitalists get together for lunch every Tuesday and say, "These are the startups I'm thinking of starting," and somebody will say, "Well why don't you change it a little bit?" They share ideas openly because they know that there's a half- life of these ideas, and in six months they'll get better. The companies are even better because of it. The biology community would never think of such a thing, of sharing their plans for new development. It is quite a different culture. I think you're right.

EF: Don, just a small follow-up question for this. As you were speaking, it was bubbling in my mind that in various sciences, including ours, the big prizes are sometimes given for what are considered breakthrough ideas. So a young person can win a big prize. Sometimes it's given for career contributions. The Nobel Prizes are like this too. Sometimes a brilliant thing flashes up on the screen, like the CT scan. The Nobel Prize was given to a EE guy for the CT scan, in medicine. But often the prize is given to someone for a career's worth of work. Do you think that we have breakthrough ideas in computer science of that sort?

DK: Yeah, but I minimize their importance, in a sense. We do have landmark ideas that sort of all of a sudden… Something like in theoretical field, the idea of NP-completeness. All of a sudden we had thousands of people inspired by this idea. But how many of them are there? It was interesting. I wrote a letter to Allen Newell when I was starting to write "The Art of Computer Programming." I think I wrote it to him in 1963 or something like this. I said, "Allen, I'm struck by the fact that all good ideas in computer science were invented before 1960 and we've just been rediscovering the wheel since then." I'm not sure, but sort of that was the thrust of my letter. Allen replied to me, "Oh no, Don, you're suffering from the bow wave phenomenon," Or something like this. Then I had another conversation with Juris Hartmanis, who was the head of the department at Cornell. Juris was a wonderful leading person in Automata Theory, and he happened to have been a student of Marshall Hall as well. He was recruiting me to come to Cornell at the same time I was considering Berkeley, and Stanford, and other things. He visited me, and I visited Cornell with the serious idea of going there, and didn't put it on my final list because people don't drop into Cornell the way they drop into Stanford. I would have to go to them, and I don't like traveling that much. But we had this conversation, and one of the questions that struck me, he said, "Don, what was the most important new idea in computer science during the past year?" I couldn't think of a single thing. Say 1965, or something like that. I couldn't think of any breakthrough. For the next ten or so years, I asked myself the same thing at the end of every year. What was the breakthrough that occurred this year? I couldn't come up with anything. Almost never could come up with anything.

On the other hand, in ten years the whole field had changed. I realized that what it really is, it's like a great wall, where everybody's contributing bricks to the wall, and each brick… In other words, it's the community enterprise that really has made it such a thriving field. I like to give credit to everybody who puts in one of these bricks. Of course, we've got to have the major prizes in order to get into the newspaper and things like this. But so many things go into this. The big breakthrough is not the real story, although they're wonderful when they occur. That's my take on that.

EF: Thanks, Don. We could sit here and discuss that endlessly. I'm going to resist doing that because I'd like to get on to the moment when you are sitting in that little office in [Stanford's Margaret] Jacks Hall, and you decide that you and Phyllis need a better language in which to basically, essentially, typeset your books. Listening to the early part of your conversation about this last week, it occurred to me that one level below the surface, there's something else about books. My wife just stopped being, she was a trustee, a member of the Board of the San Francisco Center for the Book. She's a book artist and she loves books.

DK: Oh, I see. I visited there two weeks ago.

EF: I get this feeling that there's something about books inside you, inside your head, that you absolutely love. Can you just tell us about your love affair with books?

DK: That goes very deep. My parents disobeyed the conventional wisdom by teaching me to read before I went into kindergarten. All of their friends said, "No, he's going to be bored in school," but I was the youngest member of the "bookworm club" in Milwaukee Public Library. I think I was two-and-a-half years old, or something like this. The Milwaukee Journal ran a little blurb about it with my picture in it because I was a member of the bookworm club at the library. I loved books from a child. In those days there weren't big drug problems and so on, and little kids could ride the streetcars downtown. I went down to the library one day, and the lights went out in the library. I went over to the window so I could see better the book I was reading. It didn't occur to me the library was closing. My parents called a couple hours later and said, "Where is he? Where's our son?", and the librarian found me in the book. I have kind of a strange love affair with books going way back. In my undergraduate years, I think I mentioned last time that a lot of my favorite textbooks were published by Addison-Wesley: the calculus book that I had, the physics book that I had, the book on number theory that I had seen. Addison- Wesley, for technical books, had a special thing. The president of the company had actually done something that other publishers… I know you were an editor for McGraw-Hill. McGraw-Hill would farm out their typesetting, but Addison-Wesley had its own house composition plant. Hans Wolf had his team of people making the type, right next door to where the editorial offices were. It was the philosophy of the company really to get a special house style, and really good designers, and they made their mark on it. They also published the first book on computer science: Wilkes, Wheeler, and Gill in 1951, or something like this. That was one of the very first books Addison-Wesley put out, at the time when it was a struggling new company. I had also this thing about the appearance of books. I wanted my books to be something that other readers would treasure the appearance of it, not just that there were some words in there.

EF: Let's go back to the time when you were planning TeX, and Phyllis was in the outside office, and the two of you needed a language.

DK: Right. Phyllis had been typing all of my technical papers. I have never seen her equal anywhere, and I've met a lot of really good technical typists. She really loved it too, and so we had a fairly good thing. She could read my handwriting. I always composed my manuscripts by hand. People ask me about this. I might as well digress yet again. I also love keyboard things. I've been playing the piano for ages. When I was in high school I learned how to run a stenograph machine, like court reporters use. I went to Spencerian College for summer class. I had the idea I'd get to college and I'm going to take notes with a stenograph machine. I tried it for two weeks at Case before giving it up. We had been taught shortcuts for how to say, "Dear

Sir," and "Yours very truly," but we didn't have any abbreviations for chemistry and all these other things.

EF: Yeah, or differential equations.

DK: But anyway, I'm fascinated by keyboards. I also took typing and I was a very good typist. I could do 70 words a minute or something like this. I got myself a Russian typewriter with a Cyrillic keyboard so that I could do my Russian homework in undergrad as well. I love keyboards. But I always compose my manuscripts handwritten. The reason is that I type faster than I think. There's a synchronization problem. I can think of ideas at about the rate I can write them down with a pencil. But with typing I'm going faster, so I have to sync, and my thoughts have to start up and stop again in a way that involves more of my brain. As a college student I found I could write a letter home much faster by hand, much faster than I could type it even though I'm a great typist. The synchronization was slowing down the total thing. Phyllis and I had this nice, symbiotic relationship. She could read my handwriting, she knew when to display a formula, make it look beautiful. You almost would think she knew more mathematics than I did, sometimes, the way she would correct a formula that I had and didn't look right to her. She would change it and also get it right. When I'm learning that typesetting is a problem of zeros and ones --just a matter of programming to get the ink where it's supposed to go -- my thought was definitely that this would be something that I would make so that Phyllis would be able to take my handwritten manuscripts and go from there. I used her as the model for the language that I was developing, and I also would be able to understand it myself.

EF: You didn't have in mind another mathematician…

DK: No.

EF: …doing his own work?

DK: That's right. I knew that Bell Labs had a system where they had been using secretaries to typeset. Bell Labs had the EQN system. I learned later that other people had developed systems where they hire and train secretaries. I used the Bell Lab system, which I knew was a working system, where somebody uses the Greek letter Alpha, they say "A-l-p-h-a." The guys in these commercial systems, the letter Alpha, they say, "Oh no, these secretaries can never learn that. They're scared by any hint that it's Greek. They just know that it's this symbol, and so they type QA for the letter Alpha, and that gives them job satisfaction because they know this code that the mathematicians don't know. My philosophy was, though, that I knew that Phyllis would like to write "A-l-p-h-a". What went into the design was her as a model. And the fact that I knew that the secretaries at Bell Labs had a language that was in existence, that it was something that secretaries could learn. At that time when I started TeX, some physics journals were already being typeset with the EQN system from Bell Labs. It looked horrible -- the spacing was just ugly -- but it was the first generation of this. But I knew that they had a language that the secretaries could learn. All I had to do was tune up the aesthetics of the final product.

EF: Don, I would like to ask you about the activities going on. You mentioned that TeX took much longer than you had anticipated. You had anticipated a one-year project. You ended up with a ten- year project. It kind of carves out a section of your life in which you were being an interface designer.

You were being a programmer. I use the term "programmer" because you yourself use it in bio material on the web, that when you were doing TeX you were a programmer. Then there's all the other things going on, both with TeX, with fonts, with the rest of your life. Can you tell us about those three things?

DK: Okay.

EF: A designer story. A programmer story.

DK: A life story. Okay, there are stories. The first part of it, I'm designing a language for my secretary. This took place in sort of two all-nighters. I made a draft. I sat up at the AI lab one evening and into the early morning hours, composing what I thought would be the specifications of a language. I had already been playing around. I looked at my book and I found excerpts from several dozen pages where I thought it gave all the variety of things I need in the book. Then I sat down and I thought, well, if I were Phyllis, how would I like to key this in? What would be a reasonable format that would appeal to Phyllis, and at the same time something that as a compiler writer I felt that I could translate into the book, because TeX is another kind of a compiler. Instead of going into machine language, instead, you're going into words on a page. That's a different output language, but it's analogous in recognizing the constructs that appear in the source file. So I went through and this day I drafted how I would typeset those 12 sample segments in a language that I thought Phyllis would understand. I also mentioned a mini-users manual for teaching this language. I wrote the draft of this one night, and I showed it to a bunch of people for their comments. Then a few weeks later I went through the same thing again. Fortunately, the Stanford AI lab, where I did this work, had a very good backup system. All of the files that were on that computer for more than 20 years, stored on archival tapes, are now being available through the internet. I found, thanks to looking at these old so-called dark tapes, I found the drafts that I made of TeX on those days when I did the design. Since I believe in source documents, as I said, I published those in my book, "Digital Typography", so the people could see what the raw thoughts were, and all the mistakes, the words that were there at the very beginning. Just as an idea of a design process. Then I showed the second version of this design to two of my graduate students, and I said, "Okay, implement this, please, this summer. That's your summer job." I thought I had specified a language. I had to go away. I spent several weeks in China during the summer of 1977, and I had various other obligations. I assumed that when I got back from my summer trips, I would be able to play around with TeX and refine it a little bit. To my amazement, the students, who were outstanding students, had not competed [it]. They had a system that was able to do about three lines of TeX. I thought, "My goodness, what's going on? I thought these were good students." Well afterwards I changed my attitude to saying, "Boy, they accomplished a miracle." Because going from my specification, which I thought was complete, they really had an impossible task, and they had succeeded wonderfully with it. These students, by the way, [were] Michael Plass, who has gone on to be the brains behind almost all of Xerox's Docutech software and all kind of things that are inside of typesetting devices now, and Frank Liang, one of the key people for Microsoft Word. He did important mathematical things as well as his hyphenation methods which are quite used in all languages now. These guys were actually doing great work, but I was amazed that they couldn't do what I thought was just sort of a routine task. Then I became a programmer in earnest, where I had to do it. The reason is when you're doing programming, you have to explain something to a computer, which is dumb. When you're writing a document for a human being to understand, the human being will look at it and nod his head and say, "Yeah, this makes sense." But then there's all kinds of ambiguities and vagueness that you don't realize until you try to put it into a computer. Then all of a sudden, almost every five minutes as you're writing the code, a question comes up that wasn't addressed in the specification. "What if this combination occurs?" It just didn't occur to the person writing the design specification. When you're faced with implementation, a person who has been delegated this job of working from a design would have to say, "Well hmm, I don't know what the designer meant by this." If I hadn't been in China they would've scheduled an appointment with me and stopped their programming for a day. Then they would come in at the designated hour and we would talk. They would take 15 minutes to present to me what the problem was, and then I would think about it for a while, and then I'd say, "Oh yeah, do this. " Then they would go home and they would write code for another five minutes and they'd have to schedule another appointment. I'm probably exaggerating, but this is why I think Bob Floyd's Chiron compiler never got going. Bob worked many years on a beautiful idea for a programming language, where he designed a language called Chiron, but he never touched the programming himself. I think this was actually the reason that he had trouble with that project, because it's so hard to do the design unless you're faced with the low-level aspects of it, explaining it to a machine instead of to another person. Maybe it was Forsythe, I think it was, who said, "People have said traditionally that you don't understand something until you've taught it in a class. The truth is you don't really understand something until you've taught it to a computer, until you've been able to program it." At this level, programming was absolutely important.

EF: Could I stop you just a second? That's exactly the same methodology that I learned from Herb Simon and Al Newell at Carnegie, which is, it's useless to spit out theories of human thinking unless you can program them. You get every detail. You have to make a decision about every detail.

DK: Yeah, and they're trying to come up with models of the brain and chess players and things like this. It becomes very clear at this point.

EF: No room for hand waving.

DK: But also in every field. Composing music. I took a class in music theory during my sabbatical year, my year in Princeton before coming to Stanford. The idea of music theory, you're supposed to decide whether or not certain combinations of notes are going to sound good or not. But if they had presented it as a programming thing -- write a program that decides whether or not these notes are going to sound good or not -- that would've focused the issue, the attention, so much more sharply. It's a dream that if I finish my "Art of Computer Programming," one of the things I want to do before I die is to spend time programming for musical composition, and see if I can come up with some good music that is developed with computer aid. I feel that in order to really understand music, it's going to help me to be able to program that. Who knows?

EF: Don, let me go back to the programming stage. I would wander into your office in Jacks occasionally, and occasionally you would jump up and down and show me something. I remember one day you were showing me something that had to do with paragraph formatting, where you had uncovered a link between that and, I think, dynamic programming or some other kind of mathematical programming. That was a very interesting story, which is told other places, but maybe you want to use that as an example to illustrate the link between one part of your life and another.

DK: I'm not sure if I mentioned that. I was telling somebody about that in the last two weeks. I don't know if I mentioned it last week.

EF: Well say it again, even if you did.

DK: Okay. I had met Leon Smith at the AI lab.

EF: Oh yeah, I think that was in the previous interview.

DK: Then in my class they said they could do this with the dynamic programming algorithm that I used for music. It turned out to also work for English texts, and that was a revelation for my student. But then when I got to actually programming it, I had to also organize it so that I could handle lots of text. I had to develop a new data structure in order to be able to do the paragraph coming in text and enter it in an efficient way. I had to introduce some ideas that are called "glue", and "penalties", and figure out how that glue should disappear at boundaries in certain cases and not in others. All these things would never have occurred to me unless I was writing the program. Edsger Dijkstra gave this wonderful Turing lecture early in the 70s called "The Humble Programmer." One of the points he made early on in his talk was that when they asked him in Holland what his job title was, he said, "Programmer," and they said, "No, that's not a job title. You can't do that; programmers are just coders." They're people who are assigned like scribes were in the days when you needed somebody to write a document in the Middle Ages.

Dijkstra said he was proud to be a programmer. Unfortunately, he changed his attitude completely, and I think he wrote his last computer program in the 1980s. At this conference, I went to in 1967 about simulation language, Chris Strachey was going around asking everybody at the conference what was the last computer program you wrote. This was 1967. Some of the people said, “I’ve never written a computer program.” Others would say, “Oh yeah, here’s what I did last week.” I asked Edsger this question when I visited him in Texas in the 90s and he said, “Don, I write programs now with pencil and paper, and I execute them in my head.” He finds that a good enough discipline. I think he was mistaken on that. He taught me a lot of things, but I really think that if he had continued... One of Dijkstra’s greatest strengths was that he felt a strong sense of aesthetics, and he didn’t want to compromise his notions of beauty. They were so intense that when he visited me in the 1960s, I had just come to Stanford. I remember the conversation we had. It was in the first apartment, our little rented house, before we had electricity in the house. We were sitting there in the dark, and he was telling me how he had just learned about the specifications of the IBM System/360, and it made him so ill that his heart was actually starting to flutter. He intensely disliked things that he didn’t consider clean to work with. So I can see that he would have distaste for the languages that he had to work with on real computers. My reaction to that was to design my own language, and then make Pascal so that it would work well for me in those days. But his response was to do everything only intellectually. So, programming. I happened to look the other day. I wrote 35 programs in January, and 28 or 29 programs in February. These are small programs, but I have a compulsion. I love to write programs and put things into it. I think of a question that I want to answer, or I have part of my book where I want to present something. But I can’t just present it by reading about it in a book. As I code it, it all becomes clear in my head. It’s just the discipline. The fact that I have to translate my knowledge of this method into something that the machine is going to understand just forces me to make that crystal-clear in my head. Then I can explain it to somebody else infinitely better. The exposition is always better if I’ve implemented it, even though it’s going to take me more time.

EF: It’s not just the exposition. It’s the understanding. That’s why I don’t do theoretical AI. I just can’t understand the thing from a theoretical point of view until I experiment with it.

DK: Yeah. That’s absolutely true. I’ve got to get another thought out of my mind though. That is, early on in the TeX project I also had to do programming of a completely different type. I told you last week that this was my first real exercise in structured programming, which was one of Dijkstra’s huge... That’s one of the few breakthroughs in the history of computer science, in a way. He was actually responsible for maybe two of the ten that I know. So I’m doing structured programming as I’m writing TeX. I’m trying to do it right, the way I should’ve been writing programs in the 60s. Then I also got this typesetting machine, which had, inside of it, a tiny 8080 chip or something. I’m not sure exactly. It was a Zilog, or some very early Intel chip. Way before the 386s. A little computer with 8-bit registers and a small number of things it could do. I had to write my own assembly language for this, because the existing software for writing programs for this little micro thing were so bad. I had to write actually thousands of lines of code for this, in order to control the typesetting. Inside the machine I had to control a stepper motor, and I had to accelerate it. Every so often I had to give another [command] saying, “Okay, now take a step,” and then continue downloading a font from the mainframe. I had six levels of interrupts in this program. I remember talking to you at this time, saying, “Ed, I’m programming in assembly language for an 8-bit computer,” and you said “Yeah, you've been doing the same thing and it's fun again." You know, you'll remember. We'll undoubtedly talk more about that when I have my turn interviewing you in a week or so. This is another aspect of programming: that you also feel that you're in control and that there's not a black box separating you. It's not only the power, but it's the knowledge of what's going on; that nobody's hiding something. It's also this aspect of jumping levels of abstraction. In my opinion, the thing that computer scientists are best at is seeing things at many levels of detail: high level, intermediate levels, and lowest levels. I know if I'm adding 1 to a certain number, that this is getting me towards some big goal at the top. People enjoy most the things that they're good at. Here's a case where if you're working on a machine that has only this 8-bit capability, but in order to do this you have to go through levels, of not only that machine, but also to the next level up of the assembler, and then you have a simulator in which you can help debug your programs, and you have higher level languages that go through, and then you have the typesetting at the top. There are these six or seven levels all present at the same time. A computer scientist is in heaven in a situation like this.

EF: Don, to get back, I want to ask you about that as part of the next question. You went back into programming in a really serious way. It took you, as I said before, ten years, not one year, and you didn't quit. As soon as you mastered one part of it, you went into Metafont, which is another big deal. To what extent were you doing that because you needed to, what I might call expose yourself to, or upgrade your skills in, the art that had emerged over the decade-and-a-half since you had done RUNCIBLE? And to what extent did you do it just because you were driven to be a programmer? You loved programming.

DK: Yeah. I think your hypothesis is good. It didn't occur to me at the time that I just had to program in order to be a happy man. Certainly I didn't find my other roles distasteful, except for fundraising. I enjoyed every aspect of being a professor except dealing with proposals, which I did my share of, but that was a necessary evil sort of in my own thinking, I guess. But the fact that now I'm still compelled to… I wake up in the morning with an idea, and it makes my day to think of adding a couple of lines to my program. Gives me a real high. It must be the way poets feel, or musicians and so on, and other people, painters, whatever. Programming does that for me. It's certainly true. But the fact that I had to put so much time in it was not totally that, I'm sure, because it became a responsibility. It wasn't just for Phyllis and me, as it turned out. I started working on it at the AI lab, and people were looking at the output coming out of the machine and they would say, "Hey, Don, how did you do that?" Guy Steele was visiting from MIT that summer and he said, "Don, I want to port this to take it to MIT." I didn't have two users. First I had 10, and then I had 100, and then I had 1000. Every time it went to another order of magnitude I had to change the system, because it would almost match their needs but then they would have very good suggestions as to something it wasn't covering. Then when it went to 10,000 and when it went to 100,000, the last stage was 10 years later when I made it friendly for the other alphabets of the world, where people have accented letters and Russian letters. I had started out with only 7-bit codes. I had so many international users by that time, I saw that was a fundamental error. I started out with the idea that nobody would ever want to use a keyboard that could generate more than about 90 characters. It was going to be too complicated. But I was wrong. So it [TeX] was a burden as well, in the sense that I wanted to do a responsible job. I had actually consciously planned an end-game that would take me four years to finish, and [then] not continue maintaining it and adding on, so that I could have something where I could say, "And now it's done and it's never going to change." I believe this is one aspect of software that, not for every system, but for TeX, it was vital that it became something that wouldn't be a moving target after while.

EF: The books on TeX were a period. That is, you put a period down and you said, "This is it."

DK: 1986 was it, in other words. Five volumes were published, "Computers and Typesetting, Volumes A, B, C, D, and E", and that was to be the end. Then we had this 1988 and 1989, changing everything from 7-bit to 8-bit, which was a major rewrite, done with the help of volunteers all over the world. But I still had to personally do everything myself in order to make sure that it wasn't going to diverge.

EF: This was at the same time it was being ported over to personal computers?

DK: It was ported over to personal computers already in 1980. It was ported to 200 different programming environments -- I'm considering the combination of operating system and language – by 1981. TeX '82 was the complete rewrite and incompatible break with TeX '78. The original design, TeX '78, had already been ported to 200 different environments before I did TeX '82. We also made sure that this could be ported.

EF: Did you have to design that porting environment?

DK: Yes. We worked on the porting environment. This was the genesis of literate programming. One of the aspects of literate programming that doesn't get top billing is the way it helps for porting a system. It's called change file mechanism. I have my master files, and nobody is allowed to touch these. It says at the top of the file, "Do not change this file unless you are D.E. Knuth." I don't know how many D.E. Knuth's there are in the world, but anyway I get to change the master file. But change files come along. The change file starts out with a line saying, "Okay, now go to the first line in the master file that matches this," and then it quotes lines from the master file, When it comes to the end, then it says, "Now replace those by these lines." This turned out to be a very flexible mechanism. It also had extra features, like you can include another change file in the midst of one change file. But anyway, there's the master files that I write, and you have everybody who's porting it. You have hundreds of these change files. Then I make a change to the master file, because I find a bug, or because I have to have a new feature before TeX is frozen. Still, the change file has very minor corrections in it. The error checking was sufficiently good that you would usually find that the people who were porting it to another environment, their ports would automatically work, even though I was changing the thing and they understood the port. So that mechanism has worked well.

EF: Don, I wanted to, while we're talking about TeX and this decade, bring in fonts. Font design, your interest in the art of font design, bringing Chuck Bigelow to Stanford. All of that, and Metafont as a program, and as a book.

DK: Yeah. Metafont. Wow, there's so many layers here. I just received in the mail two days ago a wonderful book by Herman Zapf, who's about to celebrate his 90th birthday. It tells the story of his life and everything, and I'm just thinking about it because I met so many wonderful people. The graphic designers are about the nicest people I've ever met in my life, and this came out of this group. It starts out, actually, very briefly, at Stanford. Stanford has a wonderful professor, Matt Kahn, who taught a course in basic design. Jill and I took his class – audited his class -- in 1976, I think it was. I got to rub shoulders with artists during this time. He also gave a lot of insight into the way artists do their wonderful things. Then a few years later when I'm working on TeX, of course aesthetics is very important to me. That's why I didn't like the Bell Lab system, otherwise I would've adopted the Bell Lab system. I had to have something that looked beautiful to me. Stanford has a wonderful collection of fine printing, called the Gunst Collection. I went through and I absorbed the writings of type designers through the centuries, and studied, and started to learn what makes good quality different from ordinary quality in published books. That was during the earliest time working in TeX. Before the summer of '77, I could be mostly found, like during May of that year just before my sabbatical, I could probably mostly be found in the Stanford Library reading about the history of letter forms. Before I went to China I had drafted the letters for A to Z. I'm not sure if I had gotten into all the letters. I think I had probably 26 lower case and 26 upper case letters by the time I left for China. But I had to do fonts at the same time as TeX. It wasn't something [where] I can do TeX and then I can do fonts. It's a chicken and egg problem. You can't do typesetting unless you have the fonts to work with. Structured programming gave me a different feeling from programming the old way. A feeling of confidence that I didn't have to debug something immediately as I wrote it. Even more important, I didn't have to mock-up the unwritten parts of the program. I didn't have to do any fast prototyping on something like this, because when you use structured programming methodology, you have more confidence that it's going to be right, that you don't have to try it out first. In fact, I wrote all of the code for TeX over a period of seven months, before I even typed it into a computer. It wasn't until March of 1978 when I spent three weeks debugging everything I had written up to that time. Certainly you can imagine how I'm feeling in October, November, saying, "Hmm. I wonder if this is really going to typeset a paragraph, if these data structures I have for dynamic programming are really going to work." Maybe I'm a little curious about it, but structured programming still was strong enough that I thought, "No, no. If I'm going to try to minimize my total time, then why should I have to first debug my prototype and then debug the real thing? Why don't I just do all the debugging once and save total time?" The same with fonts. I had to have fonts. I couldn't debug TeX until I had the fonts. So it's all mixed up, but working on one for a month and then going to the other for a month and coming back. I thought fonts were going to be easy. I had seen Butler Lampson playing around with fonts at Xerox PARC. He was sitting at a terminal and he had a big letter "B." I can sort of visualize it now. He was drawing splines around the edge. In my art class project I had done a project for Matt Kahn [which] taught me about splines, so I knew how to program splines. I thought, okay, I'll get the letters that are used in the old edition of "The Art of Computer Programming", and I'll do like Butler did, and I'll make my font. I was going to go over to Xerox PARC and work with their equipment. They said, “Fine. Sure, Don. We’ll give you an office over here. Of course, any fonts you design here become Xerox property. You won’t mind that?” I said, “What? All I’m going to come out with [are] my measurements, a bunch of numbers. How can you own those numbers? These are just integers. Numbers belong to God.” Well, this is a debatable point. But they said anything I do there would belong to them. So I worked instead at the Stanford AI lab, where we didn’t have anywhere near as good of precision cameras. We had a TV camera and a great amount of distortion. If you turned the light slightly up just a tiny bit, the width of the letters on the screen would grow by 25%. It was impossible to do any quality work through that. I had to learn all kind of tricks for getting around it. It became much more difficult to do fonts than I had expected. You were saying the other day that a story has to have moments of tragedy as well as success. One of the greatest disappointments in my whole life was the day that I received in the mail the new edition of volume 2 of “The Art of Computer Programming,” which was typeset with my fonts and which was supposedly to be the crowning moment of my life when I had succeeded with the TeX project. I think it was 1981, and I had gotten the best typesetting equipment, and I had written a program for the 8-bit microprocessor inside, and it had 5,000 dots-per-inch, and all of the proofs that I had coming out looked good on this machine. I went over to Addison-Wesley and they typeset it, and it came in a book. There was the book, and it was in the familiar beige color covers. I opened the book up and I’m thinking oh, this is going to be a nice moment. [But] this doesn’t look the same!

EF: You sent them film, right?

DK: I sent them film. It doesn’t look the same as my other books. I had volume 2, first edition. I had volume 2, second edition. They were supposed to look the same. Everything I had known up to that point was that they would look the same. All the measurements seemed to agree. But a lot of distortion goes on, and our optic nerves aren’t linear. All kinds of things happening. I wrote it up once, when I say I burned with disappointment. I mean, I really felt a hot flash where I “Ohhhhh!”

EF: Yeah. Probably seething anger too.

DK: I don’t know. So, I mean, I—

EF: You were saying that you put so much effort into this and it wasn’t beautiful.

DK: It wasn’t that bad. Some people didn’t notice any difference at all, but the worst was the numerals. The numbers 1, 2, 3, 4, 5 are really in a rather different style from letters, and they’re very tricky. I didn’t realize that when browsing a book our eyes jump and focus on different parts, and one of the things we focus on most, often when we’re using a book, is the page numbers. And the 2 was really ugly. And the 6 -- there is something about the 6 that it’s just not a 6. And the 5‘s! Anyway, I got to the point where I was so upset. Some of California highway signs -- the speed limit signs for 50 miles an hour, or 25 miles an hour -- the 5 is really ugly. It looks like the 5 that I used to have. I couldn’t live in Santa Rosa because they have lousy 5's on their speed limit signs in Santa Rosa. It just reminds me of this awful time. There will be a time when I would be looking at all of the 2's that I could see as I'm riding a bus, or something like this, and how am I going to get this 2 to be right, because the numbers were the worst of all. The letters were okay, but I'd seen the numbers, and I can't read my book without seeing these numbers. I'm looking up a page and I look in the index. Oh, yeah, I see, page 413. Then I have to read all these numbers in order to get to page 413.

EF: How did this get by your eyes?

DK: Before.

EF: How come it didn't get caught in the process?

DK: You see, it's the context. Having it on a film… Ok, first of all, we're working with the Xerox Graphics Printer, which has a very low resolution. Everything has jaggies -- jagged edges -- in that machine. I knew about this even before I started to go into typography. We had the Xerox Graphics Printer and we were saying, "Oh, this is interesting, but it's not a book." Then I had the nice results from Pat Winston's book that looked like a book. That was professionally designed type; it wasn't done by a computer programmer. But now I was trying to match exactly the type that we had in the other [version]. I would debug my whole book looking at Xerox XGP proofs. Then I would go to my high-res machine, this expensive typesetter in the basement, and [on] that machine it was certainly crisp, and I didn't see any jaggies in those. I had no indication that when this would actually then go to be printed on paper, the ink gets a little distorted by the printing process, and even more so bound in a place that looked exactly… It's the context. It had to look right, and it didn't at that time. I'm happy to say that I open my books now and I like what I see.

EF: You're at the bottom of this trough—

DK: Even though they don't match exactly 1968, the way they differ are pleasing to me. But I had to… So then I went to all the best type designers in the world. I had learned some of their names, and I was able to invite them to participate in my research project, and I got to meet [them]. I could see, for example, that Herman Zapf, from some of the things he had written, he seemed to be a very open- minded guy. So I wrote him a letter introducing myself and saying, "Would you be interested in spending two weeks at Stanford?" And boy! He's the absolute best in the world. In my apprenticeship he's one of my great teachers. As you mentioned Chuck Bigelow, Chuck was the dean of typography in America. I worked out to get some donations that we would be able to hire Chuck and have a joint appointment with the art department. I was glad to find out that after we had gone through the process of committees and getting the appointments approved by two departments and everything, the week after he had accepted our offer he received a MacArthur Prize Fellowship, which certainly enhanced my credibility too with the art department. This was a big, new thing for them; we had never had a joint thing with the art department before. I brought Matthew Carter, who is considered definitely the leading type designer in America. There was a great article about him in The

New Yorker last year. He was out here for a quarter. Many other visitors and industry leaders from around the world helped me at the time. Finally by 1986 I was ready. I had type that I could be happy with. They said to me, "Don, that's the normal five years' apprentice as a type designer. That's the way it goes." Originally, I thought it was just going to be a matter of making a few measurements and taking a few numbers, and that would be it.

EF: That was the TeX story, the METAFONT story. Anything else going on during this time, [in] the other parts of your life?

DK: Okay. I had to work so intensively on this software that I could not keep up my normal teaching load at Stanford. I think three or four quarters… I'm not sure. Were you chair?

EF: I was chair '76 through '81.

DK: '81, yeah. So I had to approach you and say, "Can you give me a leave of absence this quarter because I'm doing software?" Also then Nils [Nilsson] probably. Do you know who? No?

EF: After me I think Gene Golub may have taken over.

DK: Gene. Okay. Anyway, I missed three or four quarters during a period of four years, because I found that writing software was much more difficult than anything else I had done in my life, in the following sense. I had to keep so many things in my head at once. I couldn't just put them down and start something else. It really took over my life during this period. I used to sort of think there were different kind of tasks: writing a paper, writing a book, teaching a class, things like that. I could juggle all of those simultaneously. But software was an order of magnitude harder. I couldn't do that and still teach a good Stanford class. Of course, I'm advising my grad students through all this period, and they're doing great theses related to typography. Mostly, not always. But the other parts of my life were largely on hold. That includes The Art of Computer Programming. Except volume 2 was my big project, to get the new edition of volume 2 done with TeX. In 1980 I spent several months just doing pure… There were new developments in the algorithms that belong in volume 2, and I wrote a lot of new material for volume 2 during this period. But then in order to get TeX and METAFONT completely finished, that was the focus. At Stanford we had a unique class taught in the spring of '84 when the new version of METAFONT was being done. I co-taught it with Chuck Bigelow and Richard Southall. Richard is not a type designer but an expert in the interface between the designer and the actual final product. He's a talented designer but he's not one of the leading designers. His main expertise is actually knowing what distortions you have to make in order to get it to look right on the page. The three of us co-taught the class. The class met three days a week, once by Chuck, once by Richard and once by me. The students in the class are learning to design fonts at the same time. It was a great quarter doing this class, and it was all recorded on videotape. Unfortunately the tapes were all erased, so we just have our memories of this class. My life was pretty much typography. When it got to The Art of Computer Programming, every three months I would take a look at the journals that had come in for those three months and I would scan the titles. For each article I would say, "Oh, this belongs in volume 4, in a certain part." I kept an index of them for a while. I started throwing the preprints that I would receive in the mail, I started first putting them into a box. All my preprints had been organized well for volume 4, into 32 compartments. But then they were starting to overflow, so then I had X1, which just had overflow from all the compartments, and X2 and X3. I got up to X15 of these preprints. Then I gave up on that and I started putting them into a big box in a room in my house. And then the box overflowed and there was a big pile on the floor.

EF: Yeah. I remember visiting you in your study when it was just a chaos of piles.

DK: Yeah. So in 1993, I think it was, I finally attacked the pile. I went through and I had accumulated, I think it was, 14 linear feet of material that I had just been saying "someday get to this for volume 4." I think it took me a year to go through all of that and organize it and get ready to write the real volume 4 after all this time. So I put that on hold. Then before 1994 I had to get ready to, well, I'm retiring. We'll probably get into my third Stanford period. But typography was it for the early part of the '80s. Then I started doing a lot of mathematical research in the late part of the '80s, analysis of algorithms, my real life's work.

EF: I'd like to do that, to move on to the third period. You've already mentioned one of them, the retirement issue, and let's talk about that. The second one you mentioned quite early on, which is the birth in your mind of literate programming, and that's another major development. Before I quit my little monologue here I also would like to talk about random graphs, because I think that's a stunning story that needs to be told. Let's talk about either the retirement or literate programming.

DK: I'm glad you brought up literate programming, because it was in my mind the greatest spinoff of the TeX project. I'm not the best person to judge, but in some ways, certainly for my own life, it was the main plus I got out of the TeX project was that I learned a new way to program. I love programming, but I really love literate programming. The idea of literate programming is that I'm talking to, I'm writing a program for, a human being to read rather than a computer to read. It's still a program and it's still doing the stuff, but I'm a teacher to a person. I'm addressing my program to a thinking being, but I'm also being exact enough so that a computer can understand it as well. And that made me think. I'm not sure if I mentioned last week, but I think I did mention last week, that the genesis of literate programming was that Tony Hoare was interested in publishing source code for programs. This was a challenge, to find a way to do this, and literate programming was my answer to this question. That is, if I had to take a large program like TeX or METAFONT, fairly large, it's 5 or 600 pages of a book--how would you do that? The answer was to present it as sort of a hypertext, where you have a lot of simple things connected in simple ways in order to understand the whole. Once I realized that this was a good way to write programs, then I had this strong urge to go through and take every program I'd ever written in my life and make it literate. It's so much better than the next best way, I can't imagine trying to write a program any other way. On the other hand, the next best way is good enough that people can write lots and lots of very great programs without using literate programming. So it's not essential that they do. But I do have the gut feeling that if some company would start using literate programming for all of its software that I would be much more inclined to buy that software than any other.

EF: Just a couple of things about that that you have mentioned to me in the past. One is your feeling that programs can be beautiful, and therefore they ought to be read like poetry. The other one is a heuristic that you told me about, which is if you want to get across an idea, you got to present it two ways: a kind of intuitive way, and a formal way, and that fits in with literate programming.

DK: Right.

EF: Do you want to comment on those?

DK: Yeah. That's the key idea that I realized as I'm writing The Art of Computer Programming, the textbook. That the key to good exposition is to say everything twice, or three times, where I say something informally and formally. The reader gets to lodge it in his brain in two different ways, and they reinforce each other. All the time I'm giving in my textbooks I'm saying not only that I'm.. Well, let's see. I'm giving a formula, but I'm also interpreting the formula as to what it's good for. I'm giving a definition, and immediately I apply the definition to a simple case, so that the person learns not only the output of the definition -- what it means -- but also to internalize, using it once in your head. Describing a computer program, it's natural to say everything in the program twice. You say it in English, what the goals of this part of the program are, but then you say in your computer language -- in the formal language, whatever language you're using, if it's LISP or Pascal or Fortran or whatever, C, Java -- you give it in the computer language. You alternate between the informal and the formal. Literate programming enforces this idea. It has very interesting effects. I find that, for example, writing a system program, I did examples with literate programming where I took device drivers that I received from Sun Microsystems. They had device drivers for one of my printers, and I rewrote the device driver so that I could combine my laser printer with a previewer that would get exactly the same raster image. I took this industrial strength software and I redid it as a literate program. I found out that the literate version was actually a lot better in several other ways that were completely unexpected to me, because it was more robust. When you're writing a subroutine in the normal way, a good system program, a subroutine, is supposed to check that its parameters make sense, or else it's going to crash the machine. If they don't make sense it tries to do a reasonable error recovery from the bad data. If you're writing the subroutine in the ordinary way, just start the subroutine, and then all the code. Then at the end, if you do a really good job of this testing and error recovery, it turns out that your subroutine ends up having 30 lines of code for error recovery and checking, and five lines of code for what the real purpose of the subroutine is. It doesn't look right to you. You're looking at the subroutine and it looks the purpose of the subroutine is to write certain error messages out, or something like this. Since it doesn't quite look right, a programmer, as he's writing it, is suddenly unconsciously encouraged to minimize the amount of error checking that's going on, and get it done in some elegant fashion so that you can see what the real purpose of the subroutine is in these five lines. Okay. But now with literate programming, you start out, you write the subroutine, and you put a line in there to say, "Check for errors," and then you do your five lines. The subroutine looks good. Now you turn the page. On the next page it says, "Check for errors." Now you're encouraged. As you're writing the next page, it looks really right to do a good checking for errors. This kind of thing happened over and over again when I was looking at the industrial software. This is part of what I meant by some of the effects of it. But the main point of being able to combine the informal and the formal means that a human being can understand the code much better than just looking at one or the other, or just looking at an ordinary program with sprinkled comments. It's so much easier to maintain the program. In the comments you also explain what doesn't work, or any subtleties. Or you can say, "Now note the following. Here is the tricky part in line 5, and it works because of this." You can explain all of the things that a maintainer needs to know. I'm the maintainer too, but after a year I've forgotten totally what I was thinking when I wrote the program. All this goes in as part of the literate program, and makes the program easier to debug, easier to maintain, and better in quality. It does better error messages and things like that, because of the other effects. That's why I'm so convinced that literate programming is a great spinoff of the TeX project.

EF: Just one other comment. As you describe this, it's the kind of programming methodology you wish were being used on, let's say, the complex system that controls an aircraft. But Boeing isn't using it.

DK: Yeah. Well, some companies do, but the small ones. Hewlett-Packard had a group in Boise that was sold on it for a while. I keep getting… I got a letter from Korea not so long ago. The guy says he thinks it's wonderful; he just translated the CWEB manual into Korean. A lot of people like it, but it doesn't take over. It doesn't get to a critical mass. I think the reason is that a lot of people don't enjoy writing the English parts. A lot of good programmers don't enjoy writing the English parts. Two percent of the world's population is born to be programmers. I don't know what percent is born to be writers, but you have to be in the intersection in order to be really happy with literate programming. I tried it with Stanford students. I had seven undergraduates. We did a project leading to the Stanford GraphBase. Six of the seven did very well with it, and the seventh one hated it.

EF: Don, I want to get on to other topics, but you mentioned GWEB. Can you talk about WEB and GWEB, just because we're trying to be complete?

DK: Yeah. It's CWEB. The original WEB language was invented before the [world wide] web of the internet, but it was the only pronounceable three-letter acronym that hadn't been used at the time. It described nicely the hypertext idea, which now is why we often refer to the internet as a web too. CWEB is the version that Silvio Levy ported from the original Pascal. English and Pascal was WEB. English and C is CWEB. Now it works also with C++. Then there's FWEB for Fortran, and there's noweb that works with any language. There's all kinds of spinoffs. There's the one for Lisp. People have written books where they have their own versions of CWEB too. I got this wonderful book from Germany a year ago that goes through the entire MP3 standard. The book is not only a textbook that you can use in an undergraduate course, but it's also a program that will read an MP3 file. The book itself will tell exactly what's in the MP3 file, including its header and its redundancy check mechanism, plus all the ways to play the audio, and algorithms for synthesizing music. All of it a part of a textbook, all part of a literate program. In other words, I see the idea isn't dying. But it's just not taking over.

EF: We've been talking about, as we've been moving toward the third Stanford period which includes the work on literate programming even though that originated earlier. There was another event that you told me about which you described as probably your best contribution to mathematics, the subject of random graphs. It involved a discovery story which I think is very interesting. If you could sort of wander us through random graphs and what this discovery was.

DK: Well, let me try to set the scene and connect it to the past a little bit. We finished the TeX project. The climax of that was 1986, although I did have to come back into it later on to make it more world friendly. But after 1986, that was a sabbatical year for me, so it was also a time when I spent the whole year in Boston. It was the year I gave to my wife as her sabbatical. It was 25 years of marriage; I thought I could help her for one year, and she's been helping me for all the rest. That was a break. I came back to Stanford after that, and I plunged into what I consider my main life's work is analysis of algorithms. That's a very mathematical thing, and so instead of having font design visitors to my project, I had great algorithmic analysts to my project, especially Philippe Flajolet from Paris. I started working on some powerful mathematical approaches to analysis of algorithms that were unheard of in the '60s when I started the field. We were excited about these developments and able to analyze a lot more algorithms that previously were untouchable. Also other visitors, like Boris Pittel and so on. I had good research funding to do work on analysis of algorithms. In fact I brought in the TeX project originally as just a minor thing on my contract. "Say, by the way, we're going to write these technical papers and we need a publishing method to present our work, so I'll spend a little time on typography." That lasted only a year, and then I got special funding for working on TeX. But throughout that time I also was doing a little bit of support, with graduate students and visitors, doing analysis of algorithms. This became a major thing again in the late '80s. I found on the web one of my progress reports from 1987 listing ten accomplishments of that year. I had to say that I don't know if any other year was as fruitful as that year, as far as my project was concerned anyway. It was certainly in full swing again finally after, from 1977 to 1986, the work on typography. So here I am in math mode, and thriving on the beauties of this subject.

The main glory of it then occurred after the new ideas had started to gel. We started to see the deeper implications. As you learn the new techniques you apply it to new problems that were previously unreachable. One of the problems that was out there that was fascinating is the study of random graphs. Graphs are one of the main focuses of volume 4, all the combinatorial algorithms, because they're ubiquitous in applications. A lot of times in order to understand what an algorithm is doing, you see what would it do if I applied it to random data of various kinds. Yesterday at our computer forum Pat Hanrahan was telling me how many people he knows that are working with random graphs to study the internet, and so on. One of the simplest models of random graphs is one that also the physicists had been interested in for many years. It connects to so-called Bose-Einstein statistics, they tell me, although I don't really understand that much about that part of physics. This model is very simple. We start out with N points that are totally disconnected from each other. These points don't exist in three-dimensional space. They exist just as N objects in any number of dimensions. Initially there's no connection at all between any objects. But you can imagine that somebody draws two random objects, totally at random. Close your eyes, find one, and each one with equal probability, 1/N. Then find another one and then put a connection between those two. "Zap." Those two are now joined. Okay. Now we have N-2 objects that are still independent, but two of them are connected together. Do it again, and maybe you'll connect two others. After you do it a few more times you might find that these two are together, and these two are together, but then you will hook them together and we'll get four. Or we might have two that get a third; a guy goes with them. Eventually we build up trees of things, meaning that they're hooked together but they don't have cycles. There's no loops. Everything in a tree is connected to everything else in the tree, but there's only one way to get from each one to each other one. There's no loops. But we keep on adding. This random process keeps going on, adding more and more connections, one at a time. Eventually cycles occur. If we keep on going on and on and on and on, eventually everything is going to be connected to everything else directly. This is called the evolution of random graphs. We can ask, at any point in time, what does the random graph look like after we've added M connections to these N groups? What does it look like? Paul Erdos and Alfred Renyi had proved in 1960 that an amazing thing happens as we add these connections. When M gets to a value which is approximately one half of N times the natural log of N, all of a sudden a "big bang" occurs, where comparatively little connection was true before the big bang, compared to a lot after the big bang. The statistics are something like this. If we say that M, the number of edges, is equal to lambda over 2 times N. If M is N over 2, if we went ahead, added half as many edges as there are points, then lambda is 1. If lambda is 10, then I've added 5N point connections. The thing is, if lambda is less than 1… So we consider a large value of N, and we have fewer than one half… Sorry. If lambda is less than log N… No. Ok. Change my definition so the number of edges is equal to lambda times natural log N times N over 2. If lambda is less than 1, then almost surely the graph consists of only trees, and the largest tree is of size something like the logarithm of N. It's almost totally dispersed. If lambda is equal to 1, almost surely there is a component of size N to the two thirds power; if N is a million, a component of size approximately 10,000. It's N to the two thirds power. It goes from log N size trees to connect the part that's big, that has N to the two thirds. If lambda is greater than 1, it's proportional to N, not N to the two thirds. So there is this jump between a very small number and no cycles. If lambda is 1 minus, if lambda is 0.999999, you still only get log N. If lambda is 1.000001, you get N. There is this bang that's occurring, and the question…

EF: By "bang" you mean a discontinuity.

DK: Discontinuity, a double jump. People who have studied the Erdos and Renyi, and physicists, could study it from the point of view of starting from zero and going up to lambda equals 1, and then their equations would blow up at lambda equals 1. Or they could study the later stages, larger lambda, and lambda gets down towards 1, and there the equations blow up. Okay? Now a Russian man in St. Petersburg who had noticed to his surprise that actually there was some similarity between the blow-ups from the top and the blow-ups from the bottom. What we proposed to study was what happens in the middle, and center on the middle, if possible. I guess we'll continue the story later.

EF: We'll continue that story.

EF: Don, we're at the discontinuity point, and you're about to explore both sides of that point, and the story's going to get really interesting here.

DK: Well, I hope so, but at about this time, Dick Karp at Berkeley was also interested in the evolution of random graphs, and this explosion phenomena. It relates in a vague way to computer algorithms, because if we have data that has a lot of connections in it, then we would want to use a different kind of data structure to represent in the computer, and certain strategies would work a lot better. Dick Karp had shown that, for example, if we want to take the transitive closure of a binary relation, you use a different method, or if you want to update the consequences of adding a new thing, depending on how big the graph is, you want to choose a different strategy. So this becomes a problem also in an analysis of algorithm as well as in physics. He had a couple of his graduate students do a simulation and try to grow a lot of random graphs and see what happened. The word we heard from this simulation -- it actually turned out we misunderstood it -- but what it seemed to imply from what we heard from what the Berkeley students had done was the following: as the graph is growing and getting more and more connections, the graph first gets to a point where it has one cycle. It's not just trees, but there's also one of the components has an extra edge in it, more than needed to connect things together. Not only are the things connected, but also there's another edge making a cycle. Eventually there will be two cycles, and three cycles, and things like this, and there'll be more things happening. What we thought the Berkeley group had discovered was that there almost never was a case where two of the connected components of the graph would have cycles. In other words, as we're adding edges, components merged together; things that used to be apart become one. You might think that actually in a graph if we have a left component and a right component, the left component might get a cycle and the right component might get a cycle, and then they might merge later. But in the Berkeley experiments, it seemed, this almost never happened. Instead, whichever component first got a cycle, it was the only one that had cycles later on. Others would merge into it, but none of these other components would grow their own cycles first. They weren't big enough to have cycles. We thought, well, if this is true, this would also have implications for data structures and algorithms. We could design our algorithms so that they could have one place for the cycle guy, and one place for the other ones. We could have our data structure and say, well, here's where the cycles are, and here's where the trees are, and then we could do faster updating. So we set out, really, not originally to understand everything about the way the graph goes through this critical point. Our original goal was to just try to prove what we thought the Berkeley group had found empirically, this phenomenon that there's sort of almost always only one main component, or one main component that has cycled.

EF: Don, can I interrupt you just a second to ask a question? What puzzles me, and puzzles maybe the audience, which is how often do analysts, mathematical analysts, do empirical experiments to discover things? Is that a usual thing, or was it special in this case?

DK: It's a fast-growing area in mathematics. The Journal of Experimental Mathematics was founded by Sylvio Levy less than ten years ago. He was my co-author with CWEB, but he's very broad. It's because computers are now there, so we can now do empirical studies with mathematics. It's not too common. My professor, Marshall Hall, was sort of famous for his observation with combinatorial things, that at the time he best expressed the wisdom of the 1960s of saying that when you're doing mathematics it's nice to do a bunch of experiments with pencil and paper. If some problem has a parameter N associated with it, you can usually go up to some value of N, like N=10 or something, by hand. Then with the computer you can go on with N=11. Combinatorial problems tend to grow faster, to the point where the computer can go beyond the hand thing. But then you can't go to N=12, because that's already too much, because the problem is growing so much. So he says computers were good for going one case beyond what you could do by hand. But now computers are better by orders of magnitude than they were there, and also the tools that we have now for examining mathematical things are much better, the software that we have.

EF: If this journal is only ten years old, this work that you were doing around 1990 must've been very much an early kind of a pioneering thing.

DK: Well, it was, actually. I guess there was another story associated with that, and that is I did empirical studies on the first cycle that occurs with a random graph. There was the paper that I wrote just previous to the one, the work I did with Philippe Flajolet. We first developed the theory, and then we wanted to have a section at the end of the paper that validated [it] experimentally, so we could see how big the graph had to be before the asymptotics would kick in. A lot of graph problems actually behave differently when the size is small. Our theorems we knew were true when N gets up to larger than the size of the universe, but how did we actually know, if N is a million, is our theory correct? So I ran experiments, sort of as a last phase of writing the previous paper, in order to test the thing in practice for small values, since our mathematics was entirely concentrated on the case where N is getting very large, the size of the graph is getting very large. I ran the program over Christmas vacation. I think I let it run a little longer than I intended, I think because of timesharing, nobody else was using it at the Christmas vacation. I didn't realize, but a week later I got a bill from Betty Scott for $60,000 of computer time, which was way more than I had in my budget of my research grant. I refused to pay it, basically. I said, "I'm sorry, I have to declare bankruptcy." The worst part of the story is that I found out, 15 years later, that I had a bug in my program and all the answers were wrong, all the $60,000 of calculation. What we had to report in our paper was that actually our theory didn't seem to be very relevant for the small values of N. And

Professor, our stat professor -- oh, what's his name? I see him in front of me, but I don't know -- he was looking at our data and he figured out another algorithm by which he could calculate things by hand. He knew that our answers were wrong. Sure enough, all this money that I wasted on this empirical calculation, no wonder it didn't agree with our theory, because my program was, in fact, wrong. In the reprint of that paper on the first cycles, which came out in my Collected Papers on Discrete Mathematics, I think it is -- I don't remember which of mine -- I recomputed this table with a correct program. Of course, it only took five minutes on a modern computer. But with the SAIL [Stanford Artificial Intelligence Lab] computer we got a whopping bill. So it wasn't very usual to do empirical calculations at that time, and it was at Berkeley that the guys do.

EF: Let's go back to Berkeley. You had interpreted, but probably misinterpreted, the Berkeley numbers.

DK: That's right. The Berkeley numbers were telling us there might be this giant component phenomenon, that the seed is planted very early, and then it stays with the thing. That was our original motivation for studying the… What we finally found out was a good explanation of the Big Bang, but our motivation -- we didn't start out in saying, "I'm going to solve this problem." That would've been a hopeless problem. That would've been too much, even for an optimist like me, to say he was going to tackle that problem. It just turned out that we stumbled on the answer. But in our course of looking at it, we did find a way to slow down the Big Bang, and that's not too hard to understand. Let's imagine again that we're watching this graph evolve. Every graph as it evolves finally gets to a point that a cycle appears in one component. Okay, we have one component containing a cycle. Now then it comes to a point where there are two cycles in the whole graph. There are two possibilities. Either the two cycles are in the same component, or one cycle in this component, one cycle in another one. So there's a fork in the road. It goes one direction or the other. Then when the third cycle appears, we have three possibilities. We could have all three in one component, or we could have one and two, or we could have one, one and one. And so on. You could draw an abbreviated history if you just look at which components have cycles. There's a branching diagram that every evolving graph goes through some path in this diagram. The Berkeley experiment, as we understood it, was that almost always we were on the upper line of this path. Almost always there's only one component that contains cycles. These other possibilities are there, but rare. We developed tools of complex analysis that I had mostly learned from Philippe Flajolet. It got to the point where I could prove that it wasn't almost always happening on the top line, because at the very first branch, if I'm not mistaken, the odds were 72 to 5 that it would take the first branch, but 5 cases out of 77, you'd take the bottom branch. It's not going to zero, but that most of the time it takes the top branch. But then maybe those two will join together and will get up to the top branch again. We started to have more mathematics so we could find the first branch, in the two cases. A few more days later, we could extend that. We could say, "Oh, what happens when the two go into three, and three go into four?" We were getting peculiar numbers, but we could calculate these probabilities by a long sequence of steps; a lot of calculus, a lot of Mathematica -- or Macsyma, I guess it was at that time -- using the symbolic algebra systems to grind out these strange probabilities. The truth actually turned out that the Berkeley experiments had sampled the graph. Say you have a million nodes. They would sample it after you had a thousand edges, and then you print out what's the state then. What about 1,100 edges, 1,200 edges? I'm sorry, the critical point occurs at 1/2 N log N. They would sample it at periodic times, but they wouldn't sample it [exactly] at the state where you get the first cycle, the second cycle, as in our mathematics. What we were doing is we were seeing the graph at a certain number of seconds at time. The truth is that these deviations from the top line disappear very quickly. There'll be a brief instant of time where there's two [cycles], where you're not on the top line, but then it jumps back up to the top line again. If you're only sampling the graph at intervals of time, you almost never see the case where you're not on the top line. That's why we misunderstood what we thought the Berkeley experiments contained. We actually were able to prove sort of a climactic theorem, to get an exact probability that it stays on the top line throughout and never ever has more than one cycle. The number was something like 5π/12. Amazing. No, it's got to be less than one, but it equals a small rational multiple of π. That was the exact probability of staying on the top line. It's kind of amazing that the number π would occur in this connection. So we had these new mathematical tools. What it finally gave us was a way to look at the Big Bang from the center of the process, and none of our equations blow up. We're able to slow down the Big Bang and watching it happen, by means of this new scale of measurement saying, "Look at it after there's a certain number of cycles in the graph."

EF: Don, tell me about the words, "stumble upon."

DK: Stumble upon, yeah. EF: "We stumbled upon." DK: Right, yeah.

EF: What happened?

DK: I had these numbers now. They were numbers like 5/77. I wrote these numbers down, and they just looked like really crazy numbers. Then one day I decided to take the series, it's a power series, X + (5/77) X2, or something like this. You have a sequence of powers of X with weird rational numbers attached to each power. I realized that the mathematics that we were developing actually would simplify if we weren't using those numbers, but instead forming the exponential of these power series. You take ef(X) instead of f(X). I used Macsyma to calculate ef(X), and it has rational coefficients, too. But one of those rational coefficients was something like 23023. Or 17017, or something. It wasn't just a random number, You play with numbers [and] you know that 23023 is 23 x 7 x 11 x 13, because 7 x 11 x 13 is 1,001, and that happens to involve a lot of small prime numbers. So here's this number with a lot of small prime factors appearing. If we didn't take the exponential, the numbers just looked crazy; they didn't have small prime factors, they didn't have any nice mathematical redeeming features. But after I took the exponential, all of a sudden the numbers that I was looking at looked like old friends. They were something that, you know, there had to be a reason for it. God didn't want these numbers just to be there. There had to be some mathematical reason. You could say that's "stumbling on" something. An hour later I could see the pattern for all of the numbers, because now it was all small prime factors and I could guess what the next one in the series is. Before having this combination, it was impossible. The funny story is that I made this discovery in the middle of the night, about 4:00. I could explain why it's 5/77ths and everything, and I could draw the diagram of the transition of every graph as it goes through the beginning of the Big Bang. Bill Gates was visiting Stanford the next day, and they were trying to impress him so that he would donate some money to build a new house for the computer science department. They asked me to meet with him in the morning. I'm not sure if we had ever met before. I know he says now that he had studied my books rather hard when he was at Harvard. I was all filled with the enthusiasm about having seen a pattern in these numbers. I drew on the blackboard the branching structure of the first moments of the Big Bang, and I put my rational numbers on there, and I put my formula involving 6N factorial, or anyway all the pattern that I had noticed. Later on, Carolyn Tajnai, who was walking him around between the buildings, said to me, "Don, can you recreate for me what you put on the blackboard that day? Because Bill was really enthusiastic about this?" The next day he wrote a check to Stanford, for I don't know, $10 million, or something like this. I always use this story if somebody says, "Who says theoretical computer science has no value?"

EF: Great story. Here's a question which kind of wraps up the Stanford… Well, you were going to talk about your retirement, and then I was going to ask you about volumes five through seven.

DK: Okay.

EF: Say something about the retirement.

DK: I was afraid you'd ask me about volumes five through seven, so I'll talk about retirement. That's really when we're getting into phase 3 of Stanford. Phase 4 will be retirement, then, because phase 2 was certainly intensive software work, and then phase 3 was back into intensive analysis for "Art of..."

The next phase is going into retirement. As I said, I had my sabbatical year in Boston in '86. That was to be the climax and finishing of the TeX project. Then I come back from sabbatical and get back to speed and so on. One of the things that happens when you come back from sabbatical is people will say to you, "Oh, aren't you glad to be home?" And, you know, I say, "Yeah, it's nice", and all this. But I found that it wasn't. I wasn't really as happy as I let on. I mean, I was certainly enjoying this research that I was doing, but I wasn't making any progress at all on Volume 4. I'm doing this work, giant component, Big Bang type of explorations, and I'm learning all of this thing. But at the end of the year, how much more had been done? I've still got this 11 feet of preprints stacked up in my closet that I haven't touched, because I had to put that all on hold for the TeX project. I figured the thing that I'm going to be able to do best for the world is going to be to finish "The Art of Computer Programming". I can do cutting edge research, but maybe I shouldn't be just enjoying myself on this, but I should be getting stuff out the door that's going to be "The Art of Computer Programming," which I had promised to write in 1962, and here it is late 1980s. After two years, I started thinking about it during the summer of '88, as to what I should be doing with my life. At this point, see, I'm 50 years old. I was born in '38, this is 1988. I decided that I didn't need money anymore. I didn't need my Stanford salary. I had enough money in the bank. I didn't get any money from the TeX project -- that's in the public domain -- but "The Art of Computer Programming," you know, [is] selling by the thousands every year all the time. So I can afford to do whatever I want with my life. I don't have to be employed. I can do what's the best way to use whatever gifts I have to put out. I decided that I really wanted to do "The Art of Computer Programming," and get this done. The only way to do it was to stop being a professor full time. I really had to be a writer full time. I wrote a memo to Nils Nilsson, who is our chair, saying, "Nils, I've decided that after two more years I would like to go on leave of absence and never come back." I would love to continue an affiliation with Stanford whereby I would be giving occasional lectures, but I think the thing I really want to do is write "The Art of Computer Programming." I don't like the idea of a professor who just spends all his time writing books and getting paid for being a professor, so I shouldn't call myself any more a fulltime professor. I shouldn't be drawing my Stanford salary. I'm going to be doing only the books, except for occasional things. I'd like to be five percent time to keep participating in things, but I'll never get my books done unless I can really put fulltime into that. If I'm only going to make one day's worth of progress out of every 365, it's going to take an estimated several centuries to finish at this rate. I wrote this letter to Nils, and then we had meetings with the Dean, Gibbons, and the provost, who you know is Jim Ross.

They thought maybe they could find a donor to Stanford who'd like to endow a professorship for somebody who writes "The Art of Computer Programming." They didn't find that, but they did say that we could have an amicable way to achieve this. It looked like in a year-and-a-half we'd be able to find someone who would take over my role as leading the analysis of algorithms activities in the department. Unfortunately, that never happened. We never found a senior person to take over what I was doing. But as of January 1990 I became on leave of absence. They allowed the leave of absence to continue until I was 55 years old and I could officially retire with a pension. I didn't get any buy-out or anything like this, like people are talking about now, but I do get some of my health insurance and so on through Stanford. This is the kind of retirement that I worked out. I was able to also create my own title. I'm "Professor Emeritus of The Art of Computer Programming", with a capital "T" in "The Art of Computer Programming." I love that title. So starting at age 55, which is officially the beginning of '93, I was Professor Emeritus. The arrangement was that I give occasional lectures, which I'm now giving about three a year. We were hoping for more, like six a year, but it's on the average three because I'm out of town a lot. I have an office and a secretary, and I'm on campus a lot. But I don't have to raise funds for research projects. Unfortunately I don't have direct work with graduate students like I did before, and I don't have regular teaching. I enjoyed those things very much, and I think that the students that I had, I'm proud of every one of them. The thing is, "The Art of Computer Programming" is something I have to do my best at.

EF: Let me ask you a little easier question than Volume 5 through 7. Where are you in Volume 4?

DK: So, Volume 4: I'm on page 12 of Volume 4 right now, although I've already written 400 pages that come starting after about page 50. I've got a lot of it under my belt now, but you know as a computer programmer you don't write the initialization first. I'm at the point now I'm ready to write the initialization to Volume 4.

EF: But this volume must make you particularly nervous, because it's on combinatorial algorithms.

DK: It has [been] subject to combinatorial explosion, so it will Volume 4A, 4B, and 4C-- possibly 4D. I'm sure it won't get up to 4Z, but there will be sub-volumes to Volume 4 because of the huge growth in combinatorial algorithm. By the way, while it's in my mind, let me, because it related to a question you asked me last week and I didn't think of a response at the time. It was something about being an engineer versus being a scientist, or something like this. The way I tended to phrase that is the relation between theory and practice in my life. I always thought that the best way to sum up my professional work is that it has been a mixture of theory and practice, almost equally. The theory that I do gives me the vocabulary and the ways to do practical things that can make giant steps instead of small steps when I'm doing a practical problem, because of the theory that I know. The practice that I do makes me able to consider better, more robust theories, theories that are richer than if they're just purely inspired by other theories. There's this basic symbiotic relationship between those things that's probably central to the whole thing. At least four times in my life when I was asked to give kind of a philosophical talk about the way I look at my professional work, the title was always "Theory in Practice." I think the first time I did this you were chair of the department, and I had just gotten the "Fletcher Jones" professorship. That was the title, and I was asked to speak for five minutes on my life as I get this endowed chair at Stanford. My title was "Theory and Practice." I remember that in that talk I gave a kind of a spoof. I started out and I said, "Well, I've written so many pages of books, and I've published so many papers, and I've had so many students." I gave a lot of the numerical statistics, and I said, "And that just about sums me up. So now that I've got this chair, I'm going to follow the advice of the Fonz and 'Sit on it.'" I remember I had made a pretty compelling case for why I was tired and ready to 'sit on it.' I scared you to the point where you were really sweating blood there. Then, of course, in the next sentence I said, "And of course, you know that this is impossible, and that I couldn't possibly do this." And "Whew!" I could see you, you know, doing this. [Showing relief] That was the first time I gave a talk about "Theory in Practice." I gave another one; the next one was actually very interesting. It was given in the Theater of Epidaurus in Greece, the best preserved ancient theater. It was the keynote speech for the European Association for Theoretical Computer Science. They had their annual meeting in Greece that year. Greece is the place for philosophy, and also the words "theory" and "practice" both come from Greek words. So naturally I decided I would speak on "Theory and Practice" in Greece, and I could speak in this temple of Greek culture giving this talk. Melina Mercouri was the Greek Minister of Culture, and she introduced me in the speech. It was a great moment of my life to summarize the roles, the tradeoffs between the two. At that time I was working on TeX; it was early '80s. My main message to the theorists is, "Your life is only half there unless you also get nurtured by practical work." And, I said, "Software is hard." My experience with TeX taught me to have much more admiration for the colleagues that are devoting most of their life to software than I had previously done, because I didn't realize how much more bandwidth of my brain was being taken up by that work than it was when I was doing just theoretical work.

EF: While we have just a moment left, if that Greek lecture was written up, do you know where it is, so the audience could go look it up?

DK: I have a book called, "Selected Papers on Computer Science." George Forsythe told me early on when I came to Stanford, he said, "Don, sometimes in your life you're going to be speaking not to professionals, but you're going to be talking to a much more general audience. It's always scary to do that, because you don't understand… It's easier to give a speech to somebody that's exactly like you than to somebody who has a different way of thinking." When I wrote for Scientific American or something -- every once in a while I would write something that was not addressed to somebody in my own field. This book, 200 pages or something, contains all of the papers that I wrote in this way. There are three or four versions, takes on, "Theory and Practice," including the Fonz Winkler one, are reprinted in that volume. Thanks for asking.

EF: Okay.

EF: You've reviewed for us what you might call the chronologically-oriented themes of your career. Pre-Stanford, first Stanford period, and so on, until your retirement -- your pseudo-retirement, I should say. Cutting through all these are other kinds of themes that touch on in many different points in the chronological explanation of your life. In my field, I really call these the heuristics of leading a career. In fact, I told you once that I felt that one of the bad decisions I made in my career was leaving what was then Carnegie Tech a year too early, before I learned all I had to learn from Herb Simon. I don't mean learning the material. Not the content, but the heuristics of leading a life. Could you talk a little bit about that? If a Ph.D. program is kind of a research training apprenticeship where the students learn these heuristics, what are they learning from you?

DK: I have some slants that I would tend to emphasize. Other professors would emphasize other slants. I don't have a monopoly on wisdom of this kind. The kind of things that I would tend to emphasize are not just doing trendy stuff. In fact, I'd probably overemphasize that. If something is really popular, I tend to think maybe I back off. I tell myself and my students: really to go with your own aesthetics, what you think is important. Not what you think other people think you want to do, but what you really want to do yourself. That's sort of been a guiding heuristic all the way through. When I was working on typography, it wasn't fashionable for a computer science professor to do typography, but I thought it was important and a beautiful subject. So what? In fact, other people told me that they're so glad that I put a few years into it because they could make it academically respectable and now they could work on it themselves. They sort of were afraid to do it themselves. But all the way through, when my books came out, they weren't usually copies of any other books. They always were something that hadn't been fashionable to do, but they corresponded to my own perception of what ought to be done. Also, your word "heuristics" triggers another thing in my mind, and that is George Polya. Polya wrote this great book called, “Heuristics and Plausible Reasoning.” Of course, I know heuristics is a great word for you because you had the ”Heuristics Programming Project” and all these things. Heuristic, meaning discovery. Polya also inspired me. I had the great fortune to get to know him because he’s a Stanford professor. He came to my house many times and I spent a lot of time with him. One of the things that cuts across also many years is he had an approach to teaching that he called, “Let Us Teach Guessing.” When he was teaching his classes, he’s not saying, “Memorize this, guys.” He’s saying, “Here’s a problem. How do you solve it?”, with the idea that the students are going to make mistakes and then they’re going to learn how to recover from mistakes, as well as making guesses. These are important heuristics for my life, both in the teaching aspect and in the research aspect. Let me talk about the teaching aspect first. Polya gave a demonstration lecture that was filmed at Stanford, and I saw it when I was still at Caltech. I saw this. He presents the students with a problem, with something like, “You have a piece of cheese and you cut it with four strokes. How many pieces are you going to get?” Something like this. Then he has the students try to analyze this. He started out by looking at simpler problems, where it’s on a plane instead of in three dimensions, and you only take two cuts; things like this. At the end of the hour he has all the students understanding not only the solution to this problem, but also having taken apart and discovering the solution themselves. That’s what goes into their brain, because then they can solve other problems later on. I adopted this as a model for my own classes, already at Caltech. Whenever I taught a class that had a decent textbook, I would devote the class time to problem solving as a group, instead of reading to them or lecturing to them about what’s in the book. I would assume that they could read the book on their own. They come to class, we do things that aren’t in the book. We take a problem that’s similar to ones in the book and we try to work on it, almost like a language class. I go down the row and, “It’s your turn, your turn, your turn.” People soon learned that if they make a mistake, we all do, and we recover. I’d give a rule that nobody’s allowed to speak more than twice in the hour, so that everybody participates. My teaching assistants would take notes, so that the students could concentrate on what was going on instead of having to worry about having their notes right so they couldn’t listen fully. The teaching assistant’s notes would then be typed up later on by Phyllis and distributed to everyone. So we could record these sessions in the class as to things that aren’t in the book, and how to recover from errors. I kept that style of teaching all the way through until I retired. That was a great source of pleasure. I could use it except in the cases where there was no textbook available. In my own research, this idea of guessing is also very important. When I read a technical paper, I don’t turn the page until I try to guess what’s on the next page. Or, [say] the guy writing the paper is going to state a problem. Before I look any further, I’ll say, “Now how would I solve this problem?” I fail almost always. But I’m ready now to read the solution to the problem, so then I turn the page and I see. But by this time I’m ready for what’s happening. When I work on “The Art of Computer Programming,” over a period of 40 years I’ve gathered together dozens of papers on the same subject. I haven’t read any of them yet except to know that they belong to this subject. Then I read those papers, the first two papers extremely slowly with this “Don’t turn the page until you’ve tried to solve the problem yourself and do it yourself.” With this method, I can then cover dozens of papers. The last ones, I'm ready for. I just know what to look at that's a little different than I've already learned how to absorb.

That's been a key heuristic in my own research, based on guessing.

EF: That's a really interesting story. In fact, my little footnote to that is that I called my own project, the "Heuristic Programming Project" because I didn't want to infringe on John McCarthy's term, "Artificial Intelligence." Stanford Artificial Intelligence Laboratory. Everyone knew what programming was, but no one knew what heuristics were. When they asked me, I would just quote Polya. I'd say, "Polya says heuristic is the art of good guessing."

DK: Yeah. Okay, very good.

EF: Anyway, I wanted to ask you a little bit about the process of gathering up the literature and writing them in "The Art of Computer Programming" that you've been doing. To go back to Artificial Intelligence, part of the program is a problem solver, but then there's the part we don't understand very well, which is the problem generator. I've always thought of "The Art of Computer Programming" as some kind of a problem generator for you. In fact, I've been jealously thinking of that. As you begin to put things together, you see the holes.

DK: Yeah. The main perk that I get from working on "The Art of Computer Programming" is that I get first crack at a lot of really natural research problems. Because I'm the only person so far who's read a paper by two authors who didn't know of each other's existence. I can see where this guy's ideas fit in with this guy's ideas. They're both working on the same problem, but they don't realize it because they have different vocabularies, very often. Artificial intelligence people, you know, have a   algorithm or something like this. The electrical engineers are working on a problem with a different vocabulary, a different slant on it, but they're thinking of something else. The people in operations research are thinking of another way. Each person will take the problem and solve it in one respect. Person B will solve a similar problem in another respect. I get to be the one who solves problem A in respect to B and vice versa. Often these problems are natural and unify the subject. They tie the problems in with even more parts of the subject, which make more of a pattern instead of having page 1, page 2, page 3. Somehow it's a network instead of a branching structure. Then there are also the other problems that I can't solve. Those make good research problems. I usually know somebody in the world that I can suggest it to, and then science advances that way. But I get first crack at it. If it's an easy one, then I have a chance then. It's fun to do this. The danger is I have to know when to stop. If I couldn't go on, if I had to solve a problem before I turned to the next problem, I'd never get to the end of The Art.

EF: Another way to put it is you can't plug every hole.

DK: That's right. Very good. It's a lot of work writing "The Art of Computer Programming," but the big benefit is this chance to see patterns that other people didn't have the opportunity to see, because they just didn't spend 40 years gathering the material the way I did.

EF: I wanted to ask you, again it's a heuristics question, but it has to do with another qualitative aspect of picking problems and finding their solutions, which is the aesthetic that you mentioned. You mentioned that you had an aesthetic, that other people have aesthetics. You've mentioned to me in the past some various criteria that you use in your aesthetic. Do you want to mention any of those?

DK: Okay. For example, when I'm writing a computer program, I could have different aesthetics. I could say that the program should be the fastest possible, right? Or it could be the one that uses the smallest amount of memory. Or the one that takes up the smallest number of keystrokes to type. Or the one that's easiest to explain to a student. Or the one that's hardest to explain to a student. There are lots of different measures that you can apply to a program. Or to anything; to a piece of literature, music, whatever. You can say, "My goal is to make this best for teenagers" or whatever it is. Somehow you have an audience in mind, or some criterion. All artists are trying to optimize some constraints or other that you have in your mind, as to what you consider most beautiful or most important in this particular piece of work. In the combinatorial work I'm doing now in volume 4, the main goal tends to be speed, because we have these problems that involve zillions of cases. Every time we can save 100 nanoseconds, if we're doing it a billion times, that's an hour. We look for things like that because we know that everything we do is going to have a large payoff in that way. But other programs, I just want it to be elegant in a way that hangs together; somebody can read it and smile. There are so many different criteria of it. But in all cases, the thing that turns me on is the beauty of it and the style that goes with it. Dijkstra had a great remark about teaching programming. I find style important in programming. Like the style in IT, in Perlis' program, was not great. The program worked, but it was sort of bumpy. Another program I read when I was in my first year of programming was the SOAP II assembler by Stan Poley at IBM. It was a symphony. It was smooth. Every line of code did two things. It was like seeing a grand master playing chess. That's the first time I got a turn-on saying, "You can write a beautiful program."

I've mentioned that several times, because it did have an important effect on my life. I'm worried about the present state of programming. Programmers now are supposed to mostly just use libraries. Programmers aren't allowed to do their own thing from scratch anymore. They're supposed to have reusable code that somebody else has written. There's a bunch of things on the menu and you choose from these when you put them together. Where's the fun in that? Where's the beauty of that? It's very hard, [but] we have to figure out a way that we can make programming interesting for the next generation of programmers, that it's not going to be just a matter of reading a manual and plugging in the parameters in the right order to get stuff. I've got to say something else, too, that pops into my mind. I saw a review a year or so ago in Computing Reviews. Someone had written a book, something about tricks of the trade, or something like this. It was somebody telling how to use machines efficiently by using some of the less well-known instructions in the machine. The reviewer of the book hated this. He said, "If I ever caught any of my programmers using anything in this book, I'd fire them." Of course I immediately went to the library and got out the book, because this was the book for me. My attitude is, if there's a method that works well and it's not commonly known to students, let’s not stop using it. Let’s teach the students how to use this so that it’s understandable and it can be used in the next generation. But this guy, he was saying, “No, no. We already understand all the possible good ways to write programs. I’m not going to let anybody write for me using anything subtle.”

EF: Yeah, that was the kind of thing that I was telling you. Bob Bemer would come down when I was a graduate student and tell us about these tricks, like unintended side effects of instructions. “The designer never intended this but you can do this with it.”

DK: Of course, I told you about when I’m writing RUNCIBLE and we were saving one line of code here because we can use one constant for two different purposes. [In the 650] you could store a data address and you could store an instruction address. You could actually put one constant in there, and you could store it with one thing and it would zero out one field and store another one. So we could save; instead of having two constants we could have one, all kinds of stuff like that. That is terrible programming. I don’t recommend it at all. If you have a machine that has only 2000 words of memory, okay. But I’m not recommending tricks just because they’re tricks. Although if your aesthetic is to cram something in small, like you’re writing something for Gameboy or something, and you can put ten extra features in there without increasing the size of the cartridge, okay, that’s fine. But [for] most of the things, it’s much more important to have stuff that is not tricky to the point of breaking whenever you make a slight change to something else. With literate programming you can document this stuff very carefully, to warn people against it, but still it’s not great [or] to be recommended. But the fact is, a computer doesn’t slow down when it gets to a part of the program that was harder to understand. The computer doesn’t say, “Oh, I don’t understand what I’m doing here,” and then go faster like a human being does. So there’s no reason for us not to put subtle tricks in our programs -- unless we can’t document them enough so that the person who’s going to have to modify the program won’t be able to fathom it.

EF: Don, I wanted to ask you about another word that you have used, and lots of scientists use the word, difficult to define, but the word is “taste.” Good taste in problems, good taste in finding problems, good taste in solving problems. Do you want to say anything about good taste?

DK: Well, there’s no accounting for taste. I was going to mention how Dijkstra was talking about

style. That is, you want to teach your students that they should have taste, but you don’t want to tell them to have the same taste as you. You try to give them the idea of taste. You can imagine a music composer. If Beethoven or Stravinsky or somebody would take on students, would they be a great teacher if they told them to compose exactly like they did? Or if they said, “Here’s an example of a strong style. Now develop your own.” That’s what you really want to do. My feeling is it’s important to have taste driving yourself and to try to refine your taste, but you can’t impose it on somebody else. There’s no absolute way for me to know that what I believe is beautiful is going to appeal to somebody else. Still, if I am trying to define beauty by what other people think is beautiful than me, I think I’m making a mistake. That’s why I was talking about trendy stuff a minute ago.

EF: The other issue that you've talked about in the past is exercising some control of your problem selection by knowing what it is you don't know. Any words of wisdom about that?

DK: Well, the best way to learn what you don't know is to try to program it, as we were saying. Well, not exactly. Words of wisdom? I don't know. I often learn what I don't know by trying to program it for a computer. But also I found, like in trying to translate something written in another language, if I try to put it in my own words, then I realize that I don't know. If I read somebody else's translation, I don't get as much out of it as if I take a text and try to put it in my own words. This exercise of being a teacher, or in some way putting yourself into it, is the best way for me to discover what I don't understand. You can think you understand something until you try to program it, or do some other thing where you are really not just repeating something but you're actually processing it.

EF: When you discover some of these holes that need to be plugged, some of them are easy to solve, and some of them you just don't know what the answer is.

DK: Yeah. I remember the first time in my life when I spent more than 10 hours on a problem and actually got the answer. When you start out in life, when you start doing something that you don't know, you think of a question and then you answer it. First then you discover that oh, the Greeks already had done that. Then you learn a few more things, and you ask some more questions, and you say oh yeah, this was done in the 17th century, the 18th century, 19th century. Finally you get to something that, for the first time in your life, you discovered something that, as far as you know, nobody had discovered before. Then you're asking questions and you don't get anywhere with them. You have to go on to the next question. I do remember there was a time, when all of a sudden… Up until this day, if I couldn't get it in the first hour, I didn't get it even if I spent a week on it. But here was a time when I had actually worked on something more than 10 hours and I did get the solution. That was the big time in my life to realize that I could go that far. What I do now, though, is I try to give myself an hour on these problems, and then I say, "Well no, I'll have to pass that on to somebody else," send a letter to somebody who might do it. Unless I think I'm almost there, if I think "Well, maybe in another five minutes I'll get the answer." Then another hour later, if I still think I'm five minutes from the answer, I keep going at it. Sometimes I'm trapped in this mode for a week still. But not too often anymore. Just in the past week I sent off two problems to other people that I thought would be worth their attention, that they might like.

EF: I'd like to switch to the personal Don Knuth. We at Stanford know the personal Don Knuth. The people watching this video or the scholars of 50 years from now may know the professional and mathematical Knuth, but they won't have the privilege of knowing the personal Don Knuth. So I wanted to ask you a few questions that just relate to the Don that we know and love. You say in your various biographies, you always end by quoting or saying to the reader that your avocation is music, and if I had to write out my biography like that, I would also. I would also say my avocation is music. I get as much of a thrill every week by going to the Stanford choruses as anything else that happens in the week. But your musical background is way more extensive than mine. Could you tell us something about the role of music in your life, and if there is any connection with your work? What the role of music in influencing your work has been?

DK: Okay. Well it's certainly one of my greatest loves, is music. We were just talking a minute ago about taste. I don't like all kinds of music. Like everybody, I have certain music that really touches me deeply, and other kinds that I'm not really enthused about. For example, I spent an hour-and-a-half last night playing through the score of South Pacific. More than half of the songs in there I find really beautiful. On the other hand, if I were a professor of music, I would have to find a way to distance myself from opera, because I've given opera a good shot many times, and I've seen excellent performances, but it has never turned me on. So everybody has their taste. My own musical tastes are fairly eclectic. I love jazz, I love to play things by Dave Brubeck, but other kinds of jazz don't seem to work very well for me either. I like Beatles music. I don't get too thrilled by some kinds of hip-hop and so on. Every generation also has their own favorite kind of music. It must be partly because of the records that my father played when I was growing up. Things like Brahms' Symphonies are things that are deeply satisfying to me now too, by their familiarity, by what I learned. My father was a musician. He was a church organist, and a pretty good one. He played at the Chicago World's Fair in the '30s before he got married. I started piano lessons probably when I was five years old. Throughout high school I was the accompanist for the chorus, for the choir, and I played in the band. I wanted to play bassoon, but that was taken, so I played tuba -- the sousaphone. Those were the two instruments that you didn't have to own yourself. The school owned the tuba and the bassoons, and our family was poor. We didn't have money to buy instruments. My dad earned enough money to buy a piano by teaching piano lessons himself. I did then get into the band as well as the keyboard music. I took a year of organ lessons when I was in high school, from my piano teacher. I almost became a music major, as I mentioned, in college. I went into physics but if I had gone to University, I would've been a music major there. I started looking at arranging. I made arrangements for our school band. When I got into college I wrote the music for a five-minute skit that our fraternity put on. It was called "Nebbishland." That was when nebbishes were popular in greeting cards. I don't know if anybody in the future will know what nebbishes were, but one of the lines in there was "We're all on the verge of insanity." It might bring back some memories anyway. Nebbishes. "I'm a nebbish and a nebbish isn't snobbish." I'll probably put this great musical piece of mine into the final volume of my collected papers, which is going to be called "Selected Papers on Fun and Games." There'll be a little bit of music in there that I did for fun over the years. During the '60s, at the church where I was going, I was a member of the choir. I had mentioned to the choir director and organist that I had taken a year of organ lessons when I was younger. He knew that I could do some keyboard skills. If we needed a harpsichord accompaniment or something, I could help out and he could be directing, or I could go to the console while he's directing and I can be playing. One Saturday I got a phone call from his wife saying, "My husband has just come down with a detached retina in his eye." In those days, the only way to cure this was for him to sit still for six months with a pack holding his head steady. She said, "Don, did you say that you knew a little bit about the organ? Can you play on Sunday and be our temporary organist?" That's what happened. For six months I was the organist at our church in

Pasadena. Fortunately Pasadena was the home of some of America's best organists. There was a famous teacher, Clarence Mader, and five of his students who are still located in the Pasadena area. If you look at the National Recitalists of the American Guild of Organists, five of them are from Pasadena. There are others from the east and all around, but we had a very good concentration of this. So I joined the American Guild of Organists and got to see some very excellent musicians. At that time I learned something about the literature of the organ. I thought hey, it would be cool in the future if I sort of was a college professor with an interest in organ. If I had 40 more years to look at this music, there was some neat pieces of organ that are so good I could never get tired of them and I could learn to play. When I had my year in Princeton between Caltech and Stanford, I took organ at the Westminster Choir College. I had a teacher there and I had some other classes there at the college as part of my year. My teacher, Mary Krimmel, taught me a lot about how to perform. Also, I had made some friends in Pasadena that had an organ in their own house, and that seemed kind of interesting to me. My father also had an organ. It was an electronic organ but he had an organ in our house in Milwaukee. When Jill and I were planning our dream house to be built on the Stanford campus, we decided that we would have two special rooms in the house. One was my room where I would have a music room and have room for an organ, and one was her art room, a studio, where she'd have good lighting for working on her art projects. We couldn't afford to put in an organ at the beginning, but the architect made sure that there would be enough bracing in the floor to handle several tons of weight and there was a nice 16-foot ceiling so that we would have room for a good organ. I spent the next few years thinking about what kind of an organ would be good to have in the home. Peter Naur in Copenhagen introduced me to five great organ builders in Denmark. The year that I spent in Norway, I visited him also for a week and talked to some of the world's greatest organ makers that he could introduce me to in Denmark. I found out, though, that I couldn't buy a Danish organ with any reasonable economic certainty. Because the way it works in Denmark was that they don't give you a fixed price on it. The Danish labor contracts are tied to the rate of inflation. I would have to give them a blank check and say, well, whatever it costs, I would have to pay. What happened then is that I also talked to American organ builders. I found a very fine one whose shop is near UCLA, and we hit it off very well. I started going around to all the organs around the Bay Area and all the Stanford organs and listening to each pipe and each note and making notes, and then worked with the builder, Pete Seeker [ph?], down in the Los Angeles area. It turned out then that they built an organ for my house. It's a nice company that builds about four organs a year. They made an organ for my house, and I still haven't seen another house organ that I would rather have than it. It's designed to be enjoyed by the person playing it, rather than for the audience. But it really has a lot of varieties of tone.

EF: Why don't you continue with your discussion of the organ?

DK: My main hobby had turned out to be then the focus on organs, and this had lots of interesting little side stories. I'll give you a few. In the first place, I'm making the arrangements for this organ in my home just at the time when I'm finishing volume three. I have a few jokes in the indexes to my books. Some of them haven't been discovered yet. Like one of them in the TeX

[book] that people found. There's one that I think, if you look under “ten“… No, no, I'm sorry. If you look under, oh what's her name? She was the star of the movie "Ten." Oh, goodness , you know the movie I'm talking about?

EF: Yeah, sure.

DK: Anyway, she's this beautiful woman, and I put her name in the index of the book. If you look there it will just tell everywhere the number ten appears in the textbook; you can find it indexed that way. In volume four I have a place in the index where it says "pun resisted." It refers you to a page, and you're supposed to figure out what pun that I could have made on that page that doesn't appear. I have fun with my indexes. I try to make them useful, but it takes me six weeks to write them so I have to do something to amuse myself during that six weeks. In volume three if you look under "royalties, use of" you get to a page that has a picture of organ pipes on it, because this is what allowed me to get an organ in my house. In those days it cost $35,000. Other people on the block, their house cost $35,000 in those days. You can't believe it now, but that was true. That's one little story. That was actually put in before the organ was built, but I had to sign a contract some years in advance. Through the years, then, the fact that I can play organ has given me intro to lots of the great organs of the world. I don't have to be a great organist, I just have to be pretty good for a computer scientist. Then the leading computer scientist, my host wherever I am, Mexico City or Paris or whatever, will know somebody who knows an organist. Then I get introduced, and they'll take me over there, and I get to play on the organ. I've played on the world's best organs. I played on the largest organ too. I got to a point where I had sort of given plenty of lectures, and I couldn't accept any invitations to travel to give a lecture. But a guy in Philadelphia wrote to me, and he said, "Don, we really want you to lecture at Drexel University." He says, "Now about organs." He said, "If you come I'll let you play on the Wannamaker organ, which was the largest musical instrument in the world. Then we can go to Eaton Hall, and then we can go to Benjamin Franklin…this old American organ", and so on. He arranged four great organs for me to play in the Philadelphia area, so naturally I went to speak at Drexel University.

EF: “It's a deal”, yeah.

DK: The last time I was in Paris I got to play on a really great unique organ. I had two hours to play on it. I went to Israel, I could play on the organ in the Mormon Center, wherever. This fall I'll be playing in Bordeaux. I was in Zurich -- organ -- last year, or a year and a half ago. I don't have to be a great organist in order to have these opportunities. I just have to be a computer scientist who is not too bad. I don't play in public very much. The one exception, really, was at the University of Waterloo about five years ago. They have an organ professor there, and he and I put on a concert of organ duets -- music written for two performers at one or two organs. I practiced with him several times for this. That was the highlight of my organ playing, where we put this on. The music was, I guess, broadcast a couple of times on Canadian radio as well. I got to work with a really fine organist. On my web page there's a reference to the program that we played, some very interesting music.

EF: Yeah, I was going to say -- if they broadcast it, you might have tapes, and you can put them on the website.

DK: Yeah, I think I might have a tape in my collection somewhere. Good idea to try to put it on the Internet.

EF: Don, let me move on to something which is important in your life and which we all know about. The world didn't really know much about it until you published the book "3:16". Namely, your religious belief, and your studies of religion and religious thought. Do you want to say anything about that?

DK: Well, yeah. This is the Computer History Museum, but it is part of my…

EF: We're talking about you, though.

DK: That's right. The thing is, I think computer science is wonderful, but it's not everything. Throughout my life I've been in a very loving religious community. My father -- also my mother, it wasn't her career, but she sang in the church choir for 60 years -- but my father dedicated his life to being an educator in the Lutheran school systems. I was raised in Lutheran schools, and Lutheran high school, before I went to college. I come from a Midwestern Lutheran German background that has set the scene for my life. This is something that I've gotten to appreciate, that Luther was a theologian who said you don't have to close your mind. You keep questioning. You never know the answer. You don't just blindly believe something. Also, he had ways of making it both intellectual and faith, as a combination. That's part of my background. I had a lot of exposure to it as I'm young. I'm also a scientist. On Sundays I would study with other people of our church on aspects of the Bible and other topics in religion. I got this strange idea that maybe -- the Bible is a complicated subject -- maybe I could study the Bible the way a scientist would do it, by using random sampling. Like a Gallup poll. You have a complicated thing and you want to look at a small number of samples. You talk to 1000 people and you try to find out what the sentiment is in the United States about something. I thought, well what if I did this with the Bible? This was a complicated book. There's been tens of thousands of books written about the Bible. Instead of somebody else telling me what parts of the Bible to look at, what if I just chose parts that were selected in an unbiased way? I was doing this also with other things. I wrote a paper… About this same time we had this conference which was a pilgrimage to Khwarizm [now Khiva, in Uzbekistan] . The word algorithm means "from Khwarizm"; it's an Arabic word. We went to Khwarizm, and I gave a talk there trying to analyze what is the difference between a mathematician and a computer scientist. We did this by looking at page 100 of many books. We sampled the works of mathematicians to find out, what do mathematicians do? From an AI point of view we tried to say what would we have to program in the computer in order to reproduce page 100 of these books. It's this idea of sampling. I was using it also for grading term papers. A student gives me a 50-page paper. I don't have time to read 40 of these papers and get my grades done. I only have a week to do the grading. So I would look at parts of the term paper. The student wouldn't know which parts I was going to look at, but I would look at parts of them and I would use that to assess the quality of the whole. I got in trouble with this Master's thesis about the CS bookkeeping problem, that we talked about last week. But anyway, I'm using sampling. So I said, let me do it to the Bible too. I wanted to have a rule. I would study this with my friends at our church in Menlo Park. We would, as a group, discuss randomly chosen parts of the Bible. The rule I decided on was we were going to study Chapter 3, Verse 16 of every book of the Bible. Genesis 3:16, Exodus 3:16, and it ends with Revelation 3:16. The reason is, that if any part of the Bible is known by its number, it's John 3:16. There's a verse in the Bible that people put up on Super Bowl Sunday, and it's supposed to be a capsule summary of the gospel. A lot of people knew it; 3:16 had a catchy phrase in people's minds. I said, "Okay, we all know John 3:16. Nobody knows what's Genesis 3:16." Well it turned out very interesting. It's about women's liberation. Exodus 3:16, and so on. I mentioned Peter Wegner last week. Well, his wife Judith Wegner is a great scholar of women's issues in the Hebrew scriptures. She couldn't believe it, but three of the verses that I chose are key verses in her own studies. It just turned out a really strange coincidence. Isaiah 3:16 talks about women strutting. Anyway, it's very funny. It was just serendipity, but in fact there's a nice joke about it. Somebody called it a “cross section” of the scriptures because of the cross in Christian theology. I did this with my friends in Menlo Park. Actually I had announced that we were going to meet the next Sunday and we were going to study Genesis 3:16, Exodus 3:16. Then I came down with an attack of kidney stones, and I was in the Stanford Hospital. We couldn't meet for our first session of this group. But I looked and I was in hospital room 316. So I said, "Whoa. Well, God wants me to continue with this project."

EF: A big sign.

DK: So we went through, and the class grew in interest all the way through. It could have been a real dud [if] all these are really boring, but it didn't happen that way. It was sustained all the way through, and people got inspired. Some of the women in the class were very good at calligraphy, and they would take these verses and they would write them beautifully, and we'd put them up in front of us as we're studying the things. I had this experience in the late '70s where sampling gave some insight into the complicated thing called the Bible. All of a sudden I get this “Aha!” moment in the middle of the night after I met Hermann Zapf and a whole bunch of other experts on letter forms. I'm working on the TeX project in the early '80s and I said, "Boy, this class that we did on the 3:16 turned out to be really interesting for us. It would be interesting to other people too. We could make it into a book. What if I asked Hermann if he would do a few pages of a book for me like the women in the class had been doing?" He was sort of the dean of all the calligraphers of the world. He's the god to the calligraphers and he knows all the calligraphers everywhere. I didn't really dream of asking him to do it, but I asked him to do the cover. I said, "Herman, I got this strange idea for a book on 3:16. Can you make for me the most beautiful 3 that's ever been drawn in the history of mankind, and the most beautiful colon, and the most beautiful 1 and 6 to go with it, and make it for the cover of the book?" He sends me back a letter. He says, "Don this is wonderful" and he also gives me sketches of a couple other verses that he looked at in his German Bible. He says, "Don, I know the best calligraphers in every country of the world. We could invite them, each one, to do a page." So he's on the bandwagon. I got him, and everybody loves him. To make a long story short, as I'm on my sabbatical year in Boston I also am going to the Boston Public Library several hours looking up what all the greatest writers about the Bible have said through the centuries about Genesis 3:16, et cetera, et cetera. I made my own translations of these verses. This ties in with your question a minute ago about how do you know what you don't understand? I'm thinking if I really want to understand Genesis 3:16 I shouldn't read somebody else's translation, but I should look at the Hebrew words and how those words have been used in other parts of Genesis and so on; how other people have translated these. I copied out 60 different translations of each verse, so I knew that in my own translation every mistake I made had also been made by at least ten other people. Then I made my translation, and I sent out a letter -- Herman and I signed it, both -- commissioning the best artists all over the world to do these verses as a page in the book. While I'm in Boston these artworks started coming. It's like getting a Christmas present every day, with these beautiful mailing tubes and all of that. I mean, the calligraphers also write beautiful letters; "Dear Don" and things like this. Many stories involved with individual pages later on. That's why I know I can say that the graphic artists are the best people in the world, because Jill and I have met most of these people in subsequent years. We didn't know very many of them, of course, in the beginning. Then I had to write the actual text to go with it. I could go to Harvard Theological Library, and the Boston Public[Library], and I spent a few days at Yale Divinity School, and the Graduate Theological Union at Berkeley has a great library. Here in Menlo Park we've got an excellent library in St. Patrick's Seminary. And all the theological literature is well indexed, so that I can, like Jonah 3:16, you can see what articles in the theological literature have been written that refer to Jonah 3:16. So I'm not just having a cross-section of the Bible, but all the secondary literature about the Bible. There's all these tens of thousands of books. I can just look at a few parts of them that are relevant to this thing, and I can crack open books that I would never see before. For example, John Calvin writes 90 volumes about theology. But I'm a Lutheran. Why should I ever read any of these? But, no. Now I look at a few pages of John Calvin. He wrote about Genesis 3:16. Okay, good. I find out he's got some insights that none of the other people had. I get to appreciate John Calvin. I get to appreciate St. Patrick. I get to appreciate people from early days of Christianity, different people in the 17th Century, 18th Century, 19th Century, 20th Century, all the different streams; atheist, Jews. Not too many Muslims, but there was some connection with India and so on that came up. It turned out to be really interesting. This idea of sampling turned out to be a good time-efficient way to get into a complicated subject. The result was that I actually got too confident that I knew much more. I started to feel that I knew more about the Bible than I actually had any right to do, because I'm only studying less than 1/500th of the Bible. But the thing is, people have this idea. There's a classical definition a liberal education is that you know everything about something and something about everything. Now I had the point where there were a few things that I knew everything about. I mean, I had 60 pegs of things that I had researched and I had found out just about everything that had been written about these small parts of the Bible, but these I had surrounded. There was nothing vague, so everything else in the Bible sort of could be tacked onto something solid. It gave me more of a secure feeling that I understood the Bible scholarship than I really did. But it really shows that this methodology has a lot of merit as long as you don't bias it to a particular way. It turned out to be an educational experience for me. I met these wonderful artists, and their work was shown all around the world. It was supported by the National Endowment for the Arts, and it got into many countries. It was shown in the Guinness Museum in Dublin, and greatest places like that. I saw some of the work in San Francisco in February on exhibit still.

EF: Is it still up there?

DK: The original deal was with the artists that they retained possession of the work, and I was paying them some money for the reproduction rights. What eventually developed was that the collection was so good, there was a strong feeling that it ought to be kept intact. So we wrote to them saying was it okay if the San Francisco Public Library wants to keep it in their Harrison Collection. They have the world's best collection of calligraphy, and they would like to accession these works into their collection. The artists agreed to this, so that's what happened.

EF: So it's permanently in the San--

DK: It's permanently in the San Francisco Public Library.

EF: Oh, magnificent.

DK: Yeah. So I had to come out of the closet saying, "Oh, I'm going to write a book about the Bible." Well, Isaac Asimov did this. I mentioned this for the first time to somebody when I'm living in Boston that year on my sabbatical year. There was an ACM SIGCSE convention there -- the computer science education group -- and I mentioned that I was spending my time at the library looking up these Bible verses. I thought they would say, "Oh, gosh, you're over the hill now, Don." But surprisingly, people to my face didn't really laugh at me too much. It's something that I never would talk about in a Stanford class, but this is a part of my life that integrated with it.

EF: Which is why I brought it up.

DK: Okay, thank you.

EF: I'd like to see if we can bring this full circle by getting into, finally, two aspects of your career looking back a little bit and looking ahead a little bit. We know you, and the world knows you, pretty much -- and you've said it yourself I think on the web somewhere -- that you are pretty much a lone wolf. In fact, I think you even said it last week in this interview.

DK: Yeah, could have been.

EF: You operate by yourself. We all know that. We leave you alone. You've cut yourself off from email. You work in your study for long hours. Two questions about that. Is that a myth? Because you keep talking about all the places you've traveled to and all the people you see.

And the other is, how do you feel about working with collaborators?

DK: Okay. I think I mentioned last week that the trouble… I enjoy working with collaborators, but I don't think they would enjoy working with me, because I'm very unreliable. I march to my own drummer and I can't be counted on to meet deadlines because I always underestimate things. I'm not a great co-worker. Also I'm very bad at delegating. That's why I resisted any chance, any opportunity to be the chair of the department. I knew that I would just be awful at [it]. I'd have to do it myself. I have no good way to work with somebody else on tasks that I can do myself. I'm just unable to. It's a huge skill that I lack.

With the TeX project I think it was important, however, that I didn't delegate the writing of the code, as we said before. I needed to be the programmer on a first generation project. I needed to write the manual too, a manual. I can't understand… Other users write the manual their way, but I had to write a manual too. If I delegated that, I wouldn't have realized some parts of it are impossible to explain, and I changed them as I wrote the manual realizing that it was a stupid thing that was there. So I was the tech writer of the project. I was a user of the project. I had to use TeX in order to typeset volume two. As I'm typesetting volume two, I kept track of the changes that I made to TeX as I went through volume two. It turned out almost a perfect straight line: every four pages I type, I got a new idea for how to improve TeX. For the first 500 pages of that 700-page book, I got a new idea every four pages. The last 200 pages were sheer boredom and I didn't do it. With 500 pages, if I hadn't been the user, I would not have had such a good system. I had to live with it before I gave it to somebody else. These are cases in my life where I think it's a good thing I didn't delegate. Then again, with the TeX project, once it was there, once I had this prototype out there, then we're getting more and more users. Then we would have every Friday, for two or three hours, a community meeting of several dozen people discussing questions, issues, problems with TeX, how to make it better, how to adapt it to their problems. Everybody coming to Stanford knowing about this could join our sessions on Friday. Also there was quite a team of volunteers associated with the project. There I'm working together with the group, but I'm still insisting that I be the final filter on the stuff. Now I should have mentioned, on this giant component work that we did, I mentioned that Philippe Flajolet and Boris Pittel were involved with it. But also what turned out is, I got it to a point where I couldn't prove some of the main theorems. I met Svante Janson, one of the greatest mathematicians, a Swedish guy. I was visiting Norway, so I went up to Uppsala to show him my work on this. He got enthusiastic about it, and he saw how to get me to the next level of some things that I was stumped on. Then he had a visitor from Poland who was there, so it turned out that our paper was published under four authors. He was the leading author because of alphabetical order, and so that's a joint work. Svante and Tomasz [Luczak] and myself and Boris all worked on the drafts of this. It's a giant paper too. It filled a whole issue of a journal, I don't know, 120 pages or something like this. We went through many, many drafts of this, all working together on it. So it's not that I can't ever work with anyone. In the same room with a person, I think I mentioned that I couldn't think when I was with Marshall Hall, and so on.

EF: It's too distracting, yeah.

DK: There was a guy in Princeton who was my office mate, and he and I were perfectly tuned to each other. Ed Bender. Ed and I, I mean, he could start a sentence and then I would know. Then we would work on a problem and I would take it as far as I could. Then I'd be stuck and then he would know how to do the next thing. Then he would be stuck and then I could take it over. Once in a while you find somebody where you can really do this online interaction, and the synchronization problem is nil, and it works very well. But I found that actually terrifying, because it would be your responsibility that we had to invent science whenever we were together. I already had promised to do so many other things, if I get more stimulation it'll kill me. I have to finish The Art of Computer Programming, and all these other things. So part of my being a loner is in order to fulfill the responsibilities that I have already accumulated. And knowing that I'm not that great for integrating in with somebody else's agenda; I've got too strong opinions of my own as to what I have to do. On the other hand, I came to Stanford so that I could collaborate, so that there would be a music department that I could be with. Caltech didn't have a music department. At Stanford I could be a chair of the Ph.D. thesis committee, I could chair the oral examination of music students, and students of German, and things like this. I like to come to a place where there are people who aren't clones of myself that I can learn from. Whenever I am stumped on something, I can turn to them, and they can help me learn this. I need people to help me read German and French and Japanese and Russian things, Sanskrit, and so on, when I get into a historical question where I don't have a translation handy. All kinds of parts of the university are very important that I collaborate in that way, but not so often where it's a long term project.

EF: I think, in all of that, there's probably -- I don't want to go into it here because it's your interview not mine -- but I think there are some deep issues there having to do with problem solving and concentration. When you're into something really deep, like piecing together 14 different articles on one subject to try to make sense of them, it's straining all the limited human information processing abilities, and you really can't stand a lot of input. Too many symbols change context.

DK: Yeah, it's a bandwidth question. It's easier for my left brain to communicate with my right brain than for me to communicate with another person's brain.

EF: Don, there's a little last thing I want to talk about. This is a little bit of a paradox. Well not so much; you said you like collaboration, so it's not really a paradox. But you say, I think somewhere on your website or else in one of your publications, that you're predicting that the future of computer science will be in terms of contributing pairs of people.

DK: Yeah.

EF: One a computer scientist, and one somebody from another discipline. I'd like to have you talk about that, particularly in view of the fact that you did string search algorithms, and yet you did not collaborate with a biologist; those people have the most intensive string search problems that there are today.

DK: Right. Well, to take them in last-in first-out order: the biologists didn't have those problems in 1972.

EF: That's true. That's right.

DK: The human genomes, now we've got all this data, but there wasn't such data then. Certainly if I was doing the work now, it would be a different thing. But this pairwise thing is a notion that I have that might be way off the wall. I didn't limit it to computer scientists and X. I was viewing it as a university as a whole, including humanities, medicine, everything. I'm saying knowledge in the world is exploding, and there are so many things now, that trying to look at the way a university might be 100 years from now compared to the way universities have evolved up to this point, in the following way. Up until this point we had subjects, and a person would identify themselves with what I call the vertices of a graph, where one vertex would be mathematics. Another vertex would be biology. Another vertex would be computer science, a new vertex on the block. There would be a physics vertex, and so on. Then, okay, there was biophysics and things. There was English, and Spanish, Latin. But people identified themselves as vertices, because these were the specialties. You could sort of live in that vertex, and you would be able to understand most of the lectures that were given by your colleagues. The subjects were getting bigger and bigger, but, still, we used to be able to have a computer science colloquium every week and everybody would come and we would know. But knowledge is growing and growing to the point where nobody can say they know all of mathematics, certainly. But also there's also so much interdisciplinary work now, where we see a mathematician can study the printing industry and see that some of the ideas of dynamic programming apply to book publishing. Wow! There's interactions galore wherever you look. You mentioned the electrical engineer who gets a Nobel Prize for medicine because he can do CT scanning, or whatever. My model of the way the future might go is that people wouldn't identify themselves with vertices, but rather with edges, with the connections between. Each person is a bridge. Each person is a bridge between two other areas, and that they identify themselves by the two sub- specialties that they happen to have a talent for. Then it's more of a network than a group of departments. It doesn't mean that I'm a loner, but that I'm communicating with the other people who are branches in the adjacent fields. This is the context in which that remark came up.

EF: What you're saying is that it’s an interdisciplinary world.

DK: …world. We're going to find that most of the people we talk to are people that have one foot in the same place than we do.

EF: Live on the edges, not in the nodes.

DK: Yeah.

EF: Don, thank you for sharing all of this with everyone. Not only everyone now, but everyone 50, 100, 200 years from now.

DK: Well, thank you for directing it all this way. I hope I can do half as well when I have to sit in your shoes next time.

END OF INTERVIEW