Interview Transcript - Donald Knuth

Professor Donald Knuth is an emeritus professor at Stanford and the author of the celebrated monograph The Art of Computer Programming*, as well as numerous other books. Knuth is the creator of TeX and METAFONT for typesetting, the WEB and CWEB programming systems, and the composer of* Fantasia Apocalypitca *for pipe organ. Knuth, 82, is one of the most award-winning computer scientists working today, with accolades including a Turing Award, the first ACM Grace Murray Hopper Award, and the National Medal of Science.*

This interview was prepared as a companion to Writing for Software Developers and is the first interview conducted in preparation for writing a second version, which I've barely begun to work on and will be released in Q2 2022 at the earliest. If you enjoy this interview, and haven't yet read the current edition of Writing for Software Developers, *I recommend that you do so.*

Why are typesetting and layout important? Why, as an author, do you care so deeply about the way your words look on the page?

There's no arguing taste—different people have different takes on this—but certainly as an author I want to be proud of what I wrote, and so I don't want it to look awful. I don't want to pick up something and say, "My gosh, look how ugly this is" before I even get to read the words. You are an author, so you know that you do a lot of proofreading and when you're reading something that you wrote you are thinking, "Oh boy, I wish I had changed it, made it a little different."

I'm hypersensitive to typesetting and layout. Even after I did TeX … Volume 2 of *The Art of Computer Programming* was the first big test case. I was looking forward so much to the day when I got my first copy and I could open it up and I could see, "Oh, yes, my project has succeeded." But it was maybe the worst day of my life because it looked awful to me.… I'd screwed up on the design of the numbers, you know, 0, 1, 2, 3, 4, 5, but I got a, b, c, d. I had a really ugly "5" and I said, "Oh no, I've worked so many years on this and now I hate it." I went

back to the drawing board and I went through a period where I couldn't pass a speed limit sign for "25 miles an hour" without saying, "Oh, is this a good 5 or a bad 5?" I'm very sensitive to style in the stuff that I see. The printing industry had gone through a terrible time where they started making books look worse and worse every year. They were modernizing stuff, and they fixed the things that most people read, the magazines and newspapers, but math took a big hit because it wasn't bringing in much money and so they didn't care about math.

Aside on a book recommendation

I have a little book called *Mathematical Writing* that you might want to take a look at. It is a paperback that you can get from the Math Association of America, and it is basically the notes of a class that I taught at Stanford. We had a lot of guest lectures and it talks about different styles of writing.

Regarding *The Art of Computer Programming*, it's such a fascinating project. As I was researching I realized that you started it before either of my parents were even born. When you're considering a project of that scope, how do you go about outlining and organizing and planning?

Well the best thing is to be a very bad estimator of how much time it's going to take. At one point I thought I would have it done before my son was born; he was born in 1965. If I had known how much work it was going to be, I would have been pretty stupid to have started because here we are almost sixty years later and I'm basically a little more than half done.

So when you originally set out to write it, what was that very first day like? What was the first artifact? Was there just a piece of paper with a few notes on it? Was there an outline?

I'm a grad student in my second year of grad school. I had been working part-time writing compilers and people knew that I was fairly good at writing compilers. So a guy came to me from the publishing house Addison-Wesley and said that his editor had recommended that they ask me to write a book on how to write compilers. I was thrilled because Addison-Wesley had published my calculus book and my physics book you know, a bunch of books that were my favorite textbooks. I was about your age, but the guy says, "How would you like to write a book for us about compilers?" Well, I was blown awayI was still in my second year of grad school. Anyway, I went home from that lunch and I wrote down on a piece of yellow paper a list of twelve chapters that I thought ought to be in a book on how to write compilers. Chapter 12 was actually "Compilers." The first eleven chapters were techniques that you wanted to know in general that would be of help. Somewhere I have that piece of paper that was the way I started. But, I wasn't able to do much about it for a year. During my third year of grad school, Caltech asked me to teach a class and so I was able to try out some of the ideas. Some of the notes that I had from that class went into chapter 1, eventually. I started working on it sort of full-time after I got my thesis done, that was 1963.

I started working on it and I found out that the literature was pretty bad. A lot of the stuff that had been published was just wrong or was rediscovery of something that someone else had done. Two people would write about different methods, but neither one would be aware of the other person's work and there were no comparisons between the work of person A and person B. So in order to write my book I had to figure out whose work was better. That meant that I needed to use some kind of math … I found it fascinating to ask the question, "Is this method going to run twice as fast as this method or not?" As I put work into it, the book kept getting longer and longer and there was more and more to write about. I found out that there was more need for a book that discussed more than compilers, and so I wrote to my editor and said, "Hey, this book is getting a little long, are you worried about it?" and he said, "No, no, go right ahead." And so I went right ahead and I finally sent him a manuscript. I told him, "Hey, I have a first draft of twelve chapters and it's three thousand pages, handwritten." I figured that

about eight pages of my handwriting would come out in one page of the book because I'm writing by hand and the book is printed with all of these little letters. Well, all of a sudden I'm getting a letter from the guy who originally took me to lunch, but now he's been promoted to vice president … publishers have all of these horror stories about professors who write three thousand page books and nobody buys them. He said, "How can you cut this? Do you realize it is only one and a half of your pages to one printed page?" I was way off in how long my book was coming out to be, and that was basically in the middle sixties when I hadn't really gotten much started.

Going back to what I said first, if I had been a good estimator of anything, this project would never have gotten going.

How have you worked on keeping your books updated over the decades? Not much in computing that was written in the mid-sixties is still extremely relevant today, but your work is. How much of that is the topic and how much of that is you keeping it updated in subsequent editions?

I was trying to choose topics that were not going to go dead, something that was unlikely to become uninteresting. I didn't always guess right. I have a big long section in volume 3 about sorting with magnetic tapes. At the time that was one of the main things in all software, getting information sorted on tapes, because we had very small memory. Disk drives were very small, too, and core memoryI went through more than 10 years before I had more than 20 KB of memory. With magnetic tapes, people could do stuff, and twenty percent or more of all computer time was spent on sorting things with tapes. So, I had a big section in volume three about how to sort on tapes. There was a fold-out illustration that had all sorts of stuff about how to bring the theory into practiceI worked on that as it was the thing to do in 1970.

Now, I've got these parts that hardly anybody reads in volume 3, in the middle. Still, there's lots of stuff in the front and the back of the book that people use every day. I talked to my

friends. I said, “Next edition I’m going to throw out all of the stuff about tape sorting,” and they said, “Wait a minute, technology keeps changing. They could invent a new kind of memory and all of a sudden all of those methods will still be important and we love them, keep them in.”

In the preface, you write that these books are “reference works that summarize the knowledge of several important fields.” When you think about being *comprehensive* in your writing, what does that mean to you?

The first thing I want to do is try not to be biased, try to be more of a journalist than a person writing for the opinion page. One of the key things is to get the history right, not just telling the facts but telling the story of the people behind the facts. So I work very hard on checking these things and I keep challenging whatever I’ve written. I try to find mistakes in it. And of course I pay people and they find mistakes in it. That helps a lot. Also, for keeping stuff up to date, I’ve got lots of helpers who are watching for stuff to happen. I send out lots of letters asking if something needs to be fixed or changed and also I write lots of programs, I write maybe five programs every week. I can’t write about something unless I’ve experienced it personally.

Along a similar line, what does it mean for a work to be definitive in a field?

If people think that they can trust it. The way I write is to maximize my chance of error. I could write it in such a way that I would give vague statements that are basically correct, but instead I’ll say something is twelve percent better, and I’m wrong if it is only eleven percent better. As I’m writing, I’m trying to give facts that are right or wrong. The more of those that I do, the more that makes the work definitive, as long as I get the facts right. So, I’m trying to cram a lot in while still doing my best to keep the pace from seeming too intense. That’s my goal.

Speaking of intensity, something that I really appreciate about *The Art of Computer Programming* is that the sections are segmented so that the less mathematically inclined readers such as me can make it through the whole first bit of the chapter, while the more mathematically capable readers like the friend who I was speaking to in preparation for this interview are able to look at the stuff in the back of the chapters. How do you think about segmenting your readership?

I do try to have a lot of pictures, and the most important thing in technical writing is to say everything two or three times, once informally and once formally, for example. Then, the reader can get it into his or her brain from two different slaps. I go and give a definition, and then I give a formula that is equal to the definition or something like that. Choosing the examples is one of the most important things; it has to be somehow wily enough that a person can understand why I was interested in writing about it. I just was reading a book that is probably the best of its kind on the subject, but it is all full of wishy-washy stuff. You think it is saying things, but it is saying, “Hey, this is a difficult problem, but here is the way people can massage it in practice.” But then the author doesn’t really say what is going on.

Gotcha, so you have to have these very specific, precise examples that are easily falsifiable.

Right, but I also try to write in such a way that it sounds like I’m not being really formal. I use American English. I try to give a few jokes that you can only understand if you get the technical point. I think that what I’m most guilty of is that I say something but I say it in such a smooth way that people don’t realize that I’m being precise. They think I’m being informal when I’m really meaning every syllable.

How do you teach people how to read what you've written in the work itself? I noticed there were those 18 instructions for reading the book—those were very funny with loops and sleeping and all of that—how do you teach people how to read something as they go along?

You can only hope for the best. Every once in a while, I'll say, "You really want to do exercise 12 now. Stop, turn to that page." Then, I'll try to say something else to make sure you did it. Or, I'll give a puzzle on one page and say, "Don't turn the page until you've solved it."

Philosophically, in a book like yours or a technical work, does the author's intent matter for a reader or does it matter what the reader wants to get out of their reading experience?

Certainly both matter, and there are various kinds of readers. Some people will start in the middle, some people will start at the beginning, and others will start at the end and skim: there is no way for me to reach all of them. At the least, a book has to read well if you go from the beginning onward, and so I try to keep in mind what the reader knows at this point and is expecting. But, I also try to be surprising. I don't want to give it all away.

A lot of people give this advice for writing: at the beginning of the chapter, tell them that you are going to cover these four points in this chapter, and then go to point one and point two and point three and point four, and at the end tell them what were the four points that you covered in that chapter. I suppose most of the people who talk about how to write will recommend that. But I think that's boring. I want the reader to have little surprises as they go, and I want to reward them so that they can have fun being surprised. I don't look at it as something that the reader is going to sit there and read because they have to; I'm trying to write for a reader that is self-motivating. And I'm not telling the reader what the reader ought to be interested in, I'm trying to say: "Isn't this cool?"

My final question about *The Art of Computer Programming* specifically is, how do you stay focused on a decades-long project like this when there are so many other interesting things that you dedicate your time to?

I couldn't do it if the subject wasn't interesting. If the subject was really boring, then there would be nothing to keep me going. In the morning, when I get started, sometimes I have to talk myself into getting going, but as soon as I get to it I'm sailing. Especially because I'm writing programs all of the time and asking myself all of these questions. I'm trying to figure out what the reader wants to know and then I answer the questions myself. I enjoy learning something as I do that and so I enjoy writing to tell somebody else what I learned.

Aside on transcribing

My computer is stand-alone. I don't have it connected to the internet, so in order to put stuff up I have to put it on a pen drive and carry it to the other room and put it on the internet. That's important to me because I catch myself making mistakes. I do everything twice, but it is better for me because it gets it in my brain twice. You could do something that would be more efficient, but it wouldn't come out with as good of a result because you didn't put as much of yourself into it.

How do you consider the impact of things that you have made? Do you look at the popularity of something, like TeX, or do you consider your personal interest in it? Do you have your own heuristics of value? How do you evaluate the importance of something that you're working on?

I always write something that I feel ought to be out there….

I'm a teacher, so I thought I knew something that I wished more people knew so I wanted to write about it. But I never said, "Now, let me create a fundamental typesetting system

for people." In fact, originally TeX was only going to be for my secretary and me. I needed something for *The Art of Computer Programming* and the printing industry was making it look terrible. Then I found out that with the new digital technologies it would be possible to make a good-looking book if you had a good computer program to say which pixels should be black and which should be white on the page. I started working on TeX not thinking what impact it was going to have but for TeX; I needed something so that my books wouldn't make me sick when I look at them. I was working up in Stanford's Artificial Intelligence lab and people would come into my office and say, "What are you doing, Don?" I'd say, "Oh, I'm working on this typesetting thing," and they'd say, "Oh, that's cool, can I use it?" So, I had ten users, and then I had a hundred users, and then a year later I had a thousand users, and then a year later I had ten thousand users, and it kept going. Every time you go up to more users, they have more things that they need. I kept having to revise it for a while, but I never looked at it saying, "What impact is this going to have?" I always looked at it as *this is what people need*.

Now, at the same time, I came out with a program called Metafont, which was for font design. The way I chose to do that was quite weird. It was way different from what everyone else was doing when they were making digital fonts. But I needed the fonts, so I wasn't worried about if people got it or not. Thirty or forty years went by and it's wonderfull go to an international conference of type designers and they love what I did forty years ago! Now they say that people are writing books about the people I talked to at that time. I did bring in four or five of the world's best type designers to Stanford so that they could advise me as I was working on it, but it was something that had almost no impact among practitioners for about thirty years. Finally it turns out that a lot of people get it and that made me happy, but that wasn't necessary for me before I did the work.

More recently, I worked on music. I spent more than fifty years thinking about it, and I spent five years working hard every weekend on a major piece of music for pipe organ. We had the world premiere and there have been several great performances of it, but it is a different kind of sound. I don't think that there are many people in the world now that think it is the

greatest music that they ever heard, but I'm pleased with it myself. It wasn't necessary for me that it would have any impact; it was necessary for me that I could test out this idea that there was a kind of music that I thought should be out there. I worked hard to get it out there and maybe fifty years from now people will get it and maybe not, but my motivation had not been to make the most impact. I felt there was something missing from the world and I wanted to try it.

Why is literate programming such a useful tool in your work, but also why don't more people use it?

It goes back to everyone's own taste. I'm a writer, and literate programming appeals to a writer. It gives me a chance to use both halves of my brain. I'm a teacher, and literate programming mirrors the way that I teach students. My attitude as I'm writing a program is not that I'm trying to get something to compile into code that works, my attitude is that I'm trying to do something so that a reader of the program will be able to understand *how* this program works. I'm addressing myself as a teacher to the person who reads the code. Now, most of the time I'm the only person in the world who's ever going to read the code, but that's still very important: I have to read this code as I'm debugging it; I have to read this code next year as I'm maintaining it; and with literate programming, everything goes better (except you might say that it's taking longer) because you say, "I'm not just writing the code, I'm also writing this explanation of all that the code is doing."

Well, that's what I would have thought, too, except that by taking a little bit of extra time to explain what the code is doing, I get my thoughts in order and I make many fewer mistakes. Furthermore, if I do make a mistake, I put that into the program too. I say, "Look, here's a mistake that you might have made." So my literate programs contain guides to what *not* to do as well as what *to do*. I come back to them all the time and modify them—make spin-offs of them. Sometimes I do little rewrites, sometimes I just add on; it's mostly the way I personally think a program ought to look. I don't expect everyone in the world to agree with this, at least with any particular kind of style. Why doesn't everybody switch to it? One person explained it this way:

“Well Don, there’s only a certain percentage of people in the world who are good at writing programs, and there’s only a certain percentage of people in the world who are good at writing English, and you’re getting a percentage of a percentage here. You’re expecting everybody to do both.”

But the most important reason is that in industry there is a certain style of code that everybody in Google will use or everyone in some division of Google or some division of Microsoft or Apple or whatever. They have their house style, and it’s not literate; it’s more traditional where you have code and then you have comments here and there and it works.

Literate programming would work better, but it wouldn’t work that much better that they would want to abandon theirs. It is a little bit like Esperanto. People say that Esperanto is a much better language than English, so much more logical. It was designed to be really efficient and so on. Why doesn’t everyone use Esperanto? Well, it’s because English works pretty well. Esperanto might be a hundred times better than English, but that’s not enough to get me to change.

Aside on literate programming

One thing I wanted to say about “literate programming,” by the way, is that even though it is not the house style at major software firms, like English is the style, it does actually seem to be sort of winning in Hollywood. There is this wonderful book called *Physically Based Rendering: From Theory to Implementation*. This book has deservedly won an Academy award given for this book that is entirely of literate programs. Major software being written at Pixar and elsewhere uses literate programming.

What advantages do programmers have as writers?

I had pretty good English teachers when I was in grade school. They taught me how to diagram sentences. In other words, I learned more about English grammar than a lot of kids learn from

Schoolhouse Rock. I don't know if you grew up on *Schoolhouse Rock* like my kids did, but they have all of these "Conjunction Junction what's your function?" things; it was a way to teach that English has some structure to it. Computer scientists are good at structure and good at representing stuff. So a programmer has this advantage. They can appreciate linguistic structure.

Most of our conversation has been centered around writing in English. You have travelled to many places and have a strong appreciation for many cultures. English took over computing and the Internet, even though there must be millions of programmers out there who don't speak English as a first language. How did that come to be, and how does that affect people who want to write about computing in other languages?

I don't want to give the impression that I myself am good at writing in any other language. I've got a book that I wrote in French, but that's because there were three people in the room with me doing all of the writing. They would tell me what the French words are I would say, "Oh, that sounds a little formal, what if I changed this word to another word?" They'd say, "But that's feminine, so you have to change this other adjective to agree with feminine," and stuff like that. I wrote an article once in Norwegian, too, but again I had somebody who changed all of my words and an editor, too, so at the end only about thirty percent of the words were my own.

I do think it is important to celebrate the fact that there are so many different cultures, and each language gives people the ability to think different kinds of thoughts, which is important.

So, why did English take over? I found out in the early days that it was taking over in French-speaking places, for example, because it turned out that people could get better jobs and impress their friends.… You want to write COBOL programs, and COBOL has English words in it, but if you're able to write a program in COBOL, then it looks good on your resume that you know another language. In the early days, it was kind of funny because IBM had this

arcane system control language and you would punch on a card and it would say “job in” and on another card you would punch “job out.” In France they used the same IBM software and they would say “job en” and “job au” instead of “job in” and “job out.” The fact that they could use English in their programming was something that gave them a job; people who couldn’t do that were considered less educated.

You said that spoken languages help people think different kinds of thoughts. Is the same true of programming languages?

Yeah. If somebody learns programming in APL, they’ll probably never be able to write a decent computer program if APL is their first language because you assume that what’s easy to express in a language is what’s easy to perform. If the things that are easy for you to express in a language are easy for a computer to do, that means you’re likely to be able to write programs that don’t waste much computer time. But, a lot of languages are designed not to make a computer efficient but to make the programmer spend less time expressing the problem, so there are lots of problems that can be solved now in ten minutes. There is no point in solving them in one second if you can get it done in ten minutes, but my own focus is on programs that are difficult and good ideas make them something around a thousand times faster. If you start with a language that is intended for programs that are basically easy but you only want to be able to express them quickly, that’s completely different than starting with a language that corresponds to the computations that are actually to be done.

A field like physics has schools of thought that evolve and endure over decades. Computer science schools of thought seem to be organized around languages, but do you identify any schools of thought around computer science?

Computer Science is a huge area. First off, languages are natural sources of subgroups. You can call them cults or religions too, because people who share a common language

communicate with each other better so they naturally organize into subgroups that way, by the language they use. Then, in computer science, it is the kind of algorithms that we use that separate us. People who are only working with, for example, machine-learning algorithms (that's becoming a really big part of the field now) find their methods have almost nothing in common with what I would use to solve the problems that I've been working on for most of my life.

Are there any trends that you have noticed recently that you're particularly interested in, or do you think your work is fairly unchanging?

The reason I'm working so hard now is because of ideas that have come out … there was a big revolution around the year 2000 for solving the satisfiability problem. SAT solvers became a billion-dollar industry and it revolutionized all kinds of things in hardware design with spin-offs in math and physics.. I certainly hadn't put that into the table of contents of *The Art of Computer Programming* in 1962, so I learned about it. I went to the conferences, I made friends with the best experts on SAT solvers in the world, and I wrote seven SAT solvers of various kinds that would correspond with what's going on in that world. And then I published that five years ago: three hundred pages in *The Art of Computer Programming*.

I've been learning about new methods for constraint satisfaction, and that's been keeping me very busy. For example, there's a problem it's a purely mathematical problem; it has hardly any practical use at all, but it computes a very beautiful pattern. Somebody in the year 2001 or 2002 was able to find some instances of these patterns and they said, "Look at how all of these numbers fit together, isn't this cool?" That was considered a big victory for the field of constraint satisfaction, which is another fairly large subfield of computer science at that time. Then, ten years ago in 2010, some people came along and they were able to do all of the stuff in this earlier paper. They were able to do it all a hundred times faster. What took people several minutes, they were now doing in a fraction of a second. They found larger patterns and more beautiful patterns. They spent two days working on this one pattern that was really awesome from the standpoint of mathematical beauty. Unfortunately, they spent their whole paper talking

about the techniques they used, but they didn't bother to show the final result that they had computed. They just said, "We found this pattern and it's the unique solution." They didn't bother to say what the answer was. The paper mentioned a whole bunch of other patterns, but this particular one that I thought was the most spectacular, they had left out. So, I wrote to them and asked, "What is the answer? I want to put the answer in my book." One author had retired, and the second author said, "Maybe I can find some notes from those days," but never got back to me.

A year went by and finally I decided, "Okay, I'm going to write the program myself to discover the beautiful pattern that they found ten years ago." I got lucky and it turned out that my program was a hundred times faster again. So what they had done in several hours, I could do in a few minutes. And the original problem from 2001, I could now to in six-hundredths of a second. And, of course, I found the pattern. I showed it to my friends, and they found even more incredible patterns. This was amazing, unbelievable. This is the kind of stuff that keeps me going.

It sounds like you have a strong aesthetic appreciation for the textual results of a computer program in the same way that people enjoy, say, paintings.

Exactly. There's got to be beauty in it of some kind. If you're doing something that's boring, it's your fault. It's not the subject's fault; it's your fault for not figuring out a way to make it not boring.

Do you think that the commercialization of computing has harmed the beauty of programming?

Everything can be spoiled. We can destroy all of the great ideas on the internet, but thank goodness we still have creativity and we can find beauty. One of the best ways not to be bored is to find something that you like and can be proud of and strive for that.

One thing that I find beautiful is a great story. I know you've spoken before about programs being stories. If an English professor were here, they might say that a story involves one or more characters and then some sort of conflict. Do you think that a computing story has the same basic ingredients?

They tend to be different. There tends to not be a cast of characters, although you can personify stuff, anthropomorphize it. One of the examples in my book was about an assembly line in an automobile factory. You can talk about what the workers do and what makes their day go and things like that. But, the story is more saying, "Well, it turns out that this didn't work because you didn't put the paint on at the right time. So you have to improve on your ideas." In programs it's the sequence of ideas that keep moving along it is more like a detective story.

So a story exists to present and resolve a conflict and a program exists to present and resolve a problem?

That's right! You can make that into a challenge that you have to surmount as if you were Indiana Jones.

My final question, which I hope can encapsulate many things we have talked about today: on a daily basis, what does it mean to make progress on your life's work?

That's a good way of phrasing it. I usually don't achieve by midnight what I'd hoped to get done on that day, but if I didn't move backward, that has to be progress. If I figured out something that was within my skillset that I'm able to pass onto somebody else and is likely to have some value in the future, that's something even if I haven't finished what I thought I'd get done. For example, I decided two nights ago that one of my exercises was too hard, and so I didn't dare ask that hard a question. But, I decided that for myself I still had to know the answer, because I

wouldn't be happy unless I knew. I thought I should really get on with the book and get over this thing, but I decided that I was so close to this one that I had better, for my own sake, write the program. This is a program that's going to take fifty days [to run], so this morning I started six computers at Stanford, each working on this, and I asked a friend how I can get more computers at Stanford going on it so I'll be able to do the fifty days in three days. If I have enough computers working on the problem, I'll know the answer to that question and I'll be able to move onto the next thing.

My mother had a scheduling algorithm: look for something that you can to do and do it. Instead of planning, she just said look for something that needs to be done and do it. It worked for her for ninety years.