WEBVTT

00:00:00.000 --> 00:00:00.000

Yes, I'll do that, I'll do that and so what I'll do is I'll, I'll read the questions out from the chat so don't worry about the q amp a or the chat or anything Allah, Allah be the intermediary for the questions.

00:00:00.000 --> 00:00:00.000

Good.

00:00:00.000 --> 00:00:00.000

o'clock I'm going to.

00:00:00.000 --> 00:00:08.000

Yeah, go ahead and stop the living out and start the recording. Okay. Okay.

00:00:08.000 --> 00:00:12.000

Are we live

00:00:12.000 --> 00:00:14.000

their lives now.

00:00:14.000 --> 00:00:16.000

Okay.

00:00:16.000 --> 00:00:18.000

So I'll go ahead and so.

00:00:18.000 --> 00:00:27.000

Greetings, everybody and welcome to the. The computer information science and Engineering director it's Distinguished Lecture for today.

00:00:27.000 --> 00:00:46.000

It's my great pleasure and honor to welcome Professor Alfred ao to talk to us today. It's, um, he's a he's a. I hope he doesn't blush when I say this but he's, he's a true giant in this in our field and we're delighted that he said that he has the time

00:00:46.000 --> 00:00:55.000

to speak with us. He is the largest go spin professor emeritus of computer science at at Columbia.

00:00:55.000 --> 00:01:06.000

And here the for his research achievements in our field, he was awarded in 2020, the Turing Award, along with Jeffrey omen.

00:01:06.000 --> 00:01:15.000

The Turing Award is widely perceived to be the Nobel Prize for computing, and this was originally deserved award.

00:01:15.000 --> 00:01:29.000

So to give you a little bit of context about his accomplishments. I want to tell you a little bit about compilation. Now if you're if you're not a computer scientist, you might be scratching your head about what I mean by compilation.

00:01:29.000 --> 00:01:35.000

And what I mean by that is that, um, the way computer programs are written.

00:01:35.000 --> 00:01:56.000

It's become so routine that people almost take this for granted, is that computer programmers write programs, and then they use a compiler to convert the program that they wrote into the actual computer instructions that the computer hardware can execute.

00:01:56.000 --> 00:02:12.000

So back in the old days before compilers engineers had to write these very low level detailed programs by hand. And not only that, if they had an idea for a computer program, and the, and they wrote the program and then the hardware changed, they don't

00:02:12.000 --> 00:02:17.000

have to rewrite it. because the the low level instructions will be different.

00:02:17.000 --> 00:02:37.000

So, people were interested in in writing and writing, enabling computer programmers to write at a higher level, write their programs at a higher level so that they could be wrong, so that they can be used across different computing hardware platforms

00:02:37.000 --> 00:02:43.000

and compilers are what enable this now routine practice to take place.

00:02:43.000 --> 00:02:52.000

The first compilers, including the famous one written for Fortran were required 10s of person years of effort to construct.

00:02:52.000 --> 00:03:10.000

Nowadays we expect graduate students and computing, to produce a compiler for a extra More, more, feature of a more Richard language of the original Fortran in a semester.

00:03:10.000 --> 00:03:27.000

And so you might ask, well what happened how could we go from 10s of years of of effort to students semesters worth of out of time. And I'll say that what happened was a professor ao happened and also colleagues that were working in this in the, in the

00:03:27.000 --> 00:03:31.000

same field as his as he.

00:03:31.000 --> 00:03:38.000

They are on the one hand articulated the sort of foundational principles of the forms of programming language.

00:03:38.000 --> 00:03:54.000

And in a, in a platform independent fashion and a mathematical fashion, and they developed tools that permitted these mathematical programming language descriptions to be automatically converted into the compiler software.

00:03:54.000 --> 00:04:14.000

And so, in a very real sense, the computing revolution that we're still in the middle of is due to the, the foundational efforts on automatic generation of compilers and associated theoretical advances that Professor eo and his and his colleagues in this

00:04:14.000 --> 00:04:19.000

area were developed.

00:04:19.000 --> 00:04:35.000

So, I could go on and on about Professor a hose accomplishments and awards I will note that he's a member of the National Academy of Engineering and has been the serve in the past, as the chair of the computer science and engineering section within that

00:04:35.000 --> 00:04:40.000

organization. He's also a

00:04:40.000 --> 00:04:46.000

member of the American Academy of Arts and Sciences and the Royal Society of Canada.

00:04:46.000 --> 00:05:01.000

And there are many more awards I could go on, but I know you're here to listen to him, rather than me. So I'll finish my introduction. I actually commenting on the fact that Dr.

00:05:01.000 --> 00:05:21.000

Ray who is a recipient of several teaching awards at since he's been at Columbia, and I can say that graduate students of my vintage and computing have benefited massively from his pedagogical Palace in the form of the very influential textbooks that

00:05:21.000 --> 00:05:33.000

he's written, so I, my colleagues and I in graduate school, we use the famous dragon book for for learning how to write compilers which Dr a there was a co author of.

00:05:33.000 --> 00:05:54.000

We used his algorithms textbook that he co authored with john hopkins often Jeffrey Coleman, and we used to joke that that we were all students of of our a job because of the the tremendous influence of these very well written and.

00:05:54.000 --> 00:06:07.000

had. So, I will turn the floor over now to fester a hoe and thank you again for for talking to us today.

00:06:07.000 --> 00:06:09.000

Good morning everyone.

00:06:09.000 --> 00:06:21.000

It's an honor to speak to this distinguished audience, and I would like to thank rands for his overly generous, but wonderfully informative introduction.

00:06:21.000 --> 00:06:40.000

I think he could give my talk just as well as I can. And maybe more informative Lee, but I was asked to show one slide about me. So here it is.

00:06:40.000 --> 00:06:58.000

I've been interested in abstractions and algorithms throughout my entire life, and I have been very fortunate in my career by being in the right place at the right time to have had these two interests nourished and amplified.

00:06:58.000 --> 00:07:13.000

I went to Princeton to study for my PhD, because as an undergraduate at the University of Toronto I became fascinated with how the abstraction of Boolean algebra, could be used to model digital logic circuits.

00:07:13.000 --> 00:07:21.000

I had started exploring how algorithms for minimizing Boolean functions, could be used to optimize logic circuits.

00:07:21.000 --> 00:07:39.000

And I did my senior thesis in this area. And as a consequence, Professor Ed McCluskey at Princeton, grow personal letters, suggesting to me suggesting that I should come to Princeton to study with him for my PhD instead of going to MIT.

00:07:39.000 --> 00:07:58.000

So I thought this was a no brainer that I could go and study with this master of the field of optimizing Boolean functions at Princeton at Princeton I met Jeff Coleman, as a fellow incoming graduate student, but shortly after I arrived at Princeton McCluskey

00:07:58.000 --> 00:08:08.000

was hired by a Stanford, but Princeton retaliated by hiring john Hopcroft from Stanford to replace them.

00:08:08.000 --> 00:08:18.000

Tom inherited me as a PhD student for my PhD thesis I created index grammars and nested stack atomic.

00:08:18.000 --> 00:08:26.000

After graduating from Princeton I joined the newly created computing Sciences Research Center at Bell Labs.

00:08:26.000 --> 00:08:44.000

As a member of technical stuff at that at that time. Bell Labs, was a researchers paradise. It was staffed by brilliant researchers, and it was teeming with fascinating research problems, and its reputation for hundreds of world changing discoveries like

00:08:44.000 --> 00:08:50.000

the transistor information theory and the Big Bang was legendary.

00:08:50.000 --> 00:09:03.000

My first boss, Doug McElroy, the inventor of pipes and Unix and the CO inventor of macros and programming languages hired me by saying, why don't you work on what you think is important.

00:09:03.000 --> 00:09:09.000

And I said, I think I couldn't handle that job charter

00:09:09.000 --> 00:09:22.000

Unix and see we're being created in the computing Sciences Research Center so the atmosphere was very fertile for somebody who was interested in abstractions and algorithms for programming languages, and compilers.

00:09:22.000 --> 00:09:34.000

My Bell Labs experience led me to recommend to new students for their first job. They should choose a place where they can work with the best people in their chosen field.

00:09:34.000 --> 00:09:43.000

You learn so much. By working and talking to the people who are at the forefront of their fields.

00:09:43.000 --> 00:09:48.000

You are then we'll position for selecting your second and third jobs.

00:09:48.000 --> 00:09:51.000

Okay, enough about me.

00:09:51.000 --> 00:09:59.000

Let's turn to abstractions and algorithms, the two pillars of computer science and computational thinking.

00:09:59.000 --> 00:10:00.000

Throughout my career.

00:10:00.000 --> 00:10:07.000

I have felt that abstractions are a valuable way for describing any field of human endeavor.

00:10:07.000 --> 00:10:15.000

In the early 1990s Jeff Olson and I wrote a textbook for the first course in computer science.

00:10:15.000 --> 00:10:23.000

The first chapter was titled, computer science, the mechanization of abstraction.

00:10:23.000 --> 00:10:43.000

In it, we said, computer science is a science of abstraction, creating the right model for thinking about a problem, and devising appropriate recognizable techniques to solve it, the appropriate recognizable techniques are of course algorithms, the subject

00:10:43.000 --> 00:10:59.000

of a textbook that john Hopcroft Jeff Holden and I had written two decades earlier this book, the design and analysis of computer algorithms had become one of the most cited books and computer science, and was widely used a newly created algorithms courses

00:10:59.000 --> 00:11:01.000

and universities around the world.

00:11:01.000 --> 00:11:15.000

Bill Baker the president of Bell Labs in the 1970s had me teach a televised 10 week algorithms course, out of this book to all the engineers at Bell Labs.

00:11:15.000 --> 00:11:25.000

He was very prescient, he said to me, I'll keep working on those algorithms, they will be important someday.

00:11:25.000 --> 00:11:45.000

In 2006, Jeanette when a former ad of size, and now an executive director for research at Columbia wrote a highly influential paper in the Communications of the ACM titled computational thinking in it, she said computer computational thinking is as important

00:11:45.000 --> 00:11:49.000

as the three R's to every child's education.

00:11:49.000 --> 00:12:01.000

Since that time, computational thinking has become a topic of considerable interest in educational circles, particularly in STEM education.

00:12:01.000 --> 00:12:06.000

The precise definition of computational thinking is still evolving.

00:12:06.000 --> 00:12:12.000

This is how I define computational thinking in a paper published in the computer journal, a decade ago.

00:12:12.000 --> 00:12:28.000

I said, computational thinking consists of the thought processes in formulating problems using abstractions, so there, there are solutions can be represented as computational steps and algorithms.

00:12:28.000 --> 00:12:31.000

In order to facilitate communication.

00:12:31.000 --> 00:12:49.000

I like to use terms that have precise definitions.

00:12:49.000 --> 00:12:59.000

Unfortunately, although abstractions are used in almost every field of human endeavor. the term abstraction does not have a standard definition.

00:12:59.000 --> 00:13:07.000

So here is the definition of an abstraction that Jeff Coleman and I used in our touring the word paper and the CCM.

00:13:07.000 --> 00:13:18.000

We defined a method, an abstraction as a mathematical model with two components, a data model, and a set of operations for manipulating the data in that model.

00:13:18.000 --> 00:13:25.000

You can think of the set of operations as the programming language of the abstraction.

00:13:25.000 --> 00:13:38.000

What differentiates abstractions and computer science from other fields, is that in computer science, an abstraction comes with the set of operations.

00:13:38.000 --> 00:13:45.000

And I think a simple example should clarify what we were trying to get up with our definition of an abstraction.

00:13:45.000 --> 00:13:51.000

Let's consider the dictionary abstraction and computer science.

00:13:51.000 --> 00:13:59.000

It has a data model, consisting of a universal set of elements you an A subset, you.

00:13:59.000 --> 00:14:18.000

And there are three operations, inserting an element into the subset deleting an element from the subset, and looking up to see if an element is contained in that subset, the programming language associated with the dictionary consists of straight line

00:14:18.000 --> 00:14:37.000

sequences of these three operations, and the dictionary abstraction as widely used. It is commonly used the model the symbol table, inside a compiler, where you is the set of all possible character strings and SS the set of currently valid variable names

00:14:37.000 --> 00:14:48.000

in the program. As the program has been compiled.

00:14:48.000 --> 00:14:51.000

Let's talk about algorithms for a minute.

00:14:51.000 --> 00:15:00.000

I'm sure all of you know what an algorithm is informally, you can say an algorithm is a recipe for doing something.

00:15:00.000 --> 00:15:10.000

My mother like that definition in computer science, we can formally defined an algorithm as a Turing machine that holds all inputs.

00:15:10.000 --> 00:15:21.000

I note with some chagrin, that some computer science textbooks, do not require an algorithm to halt on all inputs, but in all of the books that I have written.

00:15:21.000 --> 00:15:33.000

I have always used on canoes definition, which requires an algorithm to hold on all inputs in are the books that I have written.

00:15:33.000 --> 00:15:42.000

I called an algorithm a procedure that did not halt on all inputs, a procedure rather than an algorithm.

00:15:42.000 --> 00:15:58.000

And this distinction between algorithms and procedures is very much like the distinction between total recursive functions and partial recursive functions and recursive function theory.

00:15:58.000 --> 00:16:01.000

After defining an abstraction.

00:16:01.000 --> 00:16:08.000

Jeff and I started to create a rudimentary taxonomy for computer science abstractions based on their purpose.

00:16:08.000 --> 00:16:24.000

We began with four categories that are useful in the design of programming language translators fundamental abstractions abstract and publications declarative abstractions and computational abstractions.

00:16:24.000 --> 00:16:30.000

I will say a few words about each of these four categories

00:16:30.000 --> 00:16:47.000

or first category. the fundamental abstractions are basic standalone abstractions like stacks and queues were abstractions like graphs and trees that are often embedded in a general purpose, programming language

00:16:47.000 --> 00:16:52.000

or second category we called abstract implementations.

00:16:52.000 --> 00:17:09.000

These are data structures like hash tables and search streams that are useful for implementing fundamental abstractions and abstract implementation. They have several different instantiate options with wildly different running times at the machine level

00:17:09.000 --> 00:17:12.000

or third category.

00:17:12.000 --> 00:17:20.000

We called declarative abstractions. Their purpose is to raise the level of programming.

00:17:20.000 --> 00:17:34.000

Good examples from compiler design a regular expressions context free grabbers and full graphs, and I will see a few words about each of these three obstructions shortly.

00:17:34.000 --> 00:17:41.000

Our fourth category, we called the computational abstractions.

00:17:41.000 --> 00:17:52.000

These include models of actual computers like the random access machine or theoretical models of computation like Turing machines Boolean circuits, and quantum circuits.

00:17:52.000 --> 00:18:06.000

I should point out that there is enormous diversity in computer science abstractions, and our taxonomy is far from complete. We invite you to add useful categories to it,

00:18:06.000 --> 00:18:23.000

and important property of abstractions is that they can be layered, we can implement one abstraction, in terms of other abstractions. For example, we can implement a dictionary, in terms of other abstractions, such as limitless or hash tables, we can

00:18:23.000 --> 00:18:32.000

implement an abstraction on a computer with different algorithms which may impute vastly different running times for the abstraction.

00:18:32.000 --> 00:18:46.000

For example, regular expressions can be implemented in linear time with non deterministic finite a Tama or an exponential time with deterministic finite of Tamika

00:18:46.000 --> 00:18:51.000

to highlight the importance of abstractions and algorithms.

00:18:51.000 --> 00:19:07.000

I will illustrate how they have transformed the field of compiler design from an art to a science substantiating the comments that rents made in his introduction to simplify the discussion.

00:19:07.000 --> 00:19:18.000

Let's define a compiler as a program that translates a source program in one programming language into a semantically equivalent target program in some other language.

00:19:18.000 --> 00:19:37.000

He said, significant significant contribution to compiler design occurred when we started thinking of a compiler as a composition of phases that translate the source program into a sequence of semantically equivalent representations, from which the target

00:19:37.000 --> 00:19:48.000

program is finally generated the phases shown in violet. And this slide are called the front end of the compiler, and the phases in blue the backend.

00:19:48.000 --> 00:20:01.000

The modern compiler framework lol VM allows one of several different front ends to translate a given programming language into a single intermediate representation, called lol VM IR.

00:20:01.000 --> 00:20:17.000

The front end is followed by a code optimizer that transforms the intermediate representation into a version, from which a better target program can be produced by one of several different target language code generators that can be used in the back end

00:20:17.000 --> 00:20:24.000

of each of these phases of the compiler uses distinctive abstractions.

00:20:24.000 --> 00:20:37.000

I will know describe in more detail how the three declarative abstractions regular expressions context free grammar errors and photographs are used in compilers.

00:20:37.000 --> 00:20:50.000

Let's begin by looking at regular expressions as a declarative abstraction for lexical analysis. The first phase of the competitor.

00:20:50.000 --> 00:21:04.000

The data model of the regular expression abstraction is sets of strings over some finite alphabet, the operations of the abstraction are union concatenation and cleaning star on sets of strings.

00:21:04.000 --> 00:21:22.000

Regular expressions were invented by Stephen cleaning and 1951, and they are no routinely used in a variety of forms, and many string pattern matching programs in languages, such as grip lock Lex and Pearl.

00:21:22.000 --> 00:21:31.000

There are several different algorithms for implementing regular expression pattern matches with wildly different running times.

00:21:31.000 --> 00:21:45.000

In the late 1960s, Ken Thompson at Bell Labs, use the Justin time simulation of a non deterministic finite Tom from a script program to search for regular expression patterns in an input string, and time.

00:21:45.000 --> 00:21:52.000

That is linear and the product of the length of the regular expression, times the length of the input string.

00:21:52.000 --> 00:22:12.000

In the early 1970s, I created a regular expression search program on Unix called e grab that used a dynamic lazy construction of DFA transit deterministic finite atomic and trends positions to recognize regular expression patterns and time.

00:22:12.000 --> 00:22:33.000

That was observed in practice to be linear in the sum of the length of the regular expression plus the length of the input string of my boss Doug McElroy was interested in, had a program that dealt with dates, and he had a very large collection of regular

00:22:33.000 --> 00:22:52.000

expression patterns that could be used to specify a date in different formats. And when I had originally, use the standard subset construction to convert an MFA to a DFA to implement my grip program.

00:22:52.000 --> 00:23:05.000

It took a long time to construct the deterministic finite I told him attend. But when it was constructed it, the deterministic finite atomism could be run very very quickly.

00:23:05.000 --> 00:23:20.000

However, the process was somewhat like constructing a Boeing 747 to go across the street, you spend all your time constructing the 747, and very little time, flying it.

00:23:20.000 --> 00:23:34.000

So when I put the dynamic construction of the DFA transitions into my grip program, the Iraq brand amazingly fast, and this made my boss.

00:23:34.000 --> 00:23:57.000

Very happy, which was always good to do. Later in the 1970s, Michael last, and Eric Schmidt at Bell Labs implemented a regular lexical analyzer generator called licks that takes a specification of a lexical analyzer and automatically constructs Alexa

00:23:57.000 --> 00:23:58.000

from.

00:23:58.000 --> 00:24:11.000

Let's use regular expressions to specify the syntax of tokens and Eric Schmidt, as a young intern at Bell Labs, one of my regular expression pattern matching algorithms and two legs.

00:24:11.000 --> 00:24:20.000

And as you know, Eric Schmidt went on to become the CEO of Google.

00:24:20.000 --> 00:24:37.000

Lex is a tool that allows users to create efficient lexical analyzers from regular expression based specifications, without having to learn any algorithms for efficient regular expression pattern matching as an example.

00:24:37.000 --> 00:24:44.000

Here is a simple Lex specification for recognizing integer numbers

00:24:44.000 --> 00:24:52.000

fed into Lexis specification produces a lecture that will recognize a sequence of one or more digits in an input string.

00:24:52.000 --> 00:25:01.000

Install the sequence of digits into a symbol table and return a pointer to the sequence of digits in the symbol table, along with the token known.

00:25:01.000 --> 00:25:13.000

Lexi Lex and its descendants like flux on Linux are still used today to build lexical analyzers more than five decades after the first Lex tool was created.

00:25:13.000 --> 00:25:28.000

Let us now consider the next phase of the compiler, the syntax analyzer reinforcer as it is often called the declarative abstraction of context free grammar is is routinely used for specifying the grammatical structure of programming languages.

00:25:28.000 --> 00:25:48.000

Surprisingly, the idea that context free grammar is all. It gets attributed to an Indian scholar named Panini who used a context free grammar somewhere between 400 bc and 200 bc to specify the syntax of Sanskrit.

00:25:48.000 --> 00:25:57.000

Today, however, various forms of context free grammar errors are used to specify the syntactic structure of programming languages.

00:25:57.000 --> 00:26:10.000

In, 1973 at the first possible conference, Steve Johnson, Jeff Coleman and I presented a formalism inspired by canoes lrk grabbers for specifying syntax analyzers.

00:26:10.000 --> 00:26:28.000

The formalism was based on a syntax directed translation scheme with somatic actions attached to the production production so they shift reduce possible context free grammar for translating a token stream into some kind of output, usually a syntax tree.

00:26:28.000 --> 00:26:43.000

In our formalism the production so the ambiguous context grammar, could be augmented with this and big just invigorating rules lrk grammars are unambiguous by definition.

00:26:43.000 --> 00:27:01.000

However, the grammar that we used with the added rules, could be used to parse an input string unambiguously using a direct deterministic shift produced purser and Steve Johnson Jeff Coleman and I described such a parsing algorithm.

00:27:01.000 --> 00:27:06.000

In our 1973, Pablo paper.

00:27:06.000 --> 00:27:25.000

Steve Johnson had implemented a software tool called Yak for constructing efficient syntax analyzers from this formalism Yak allowed compiler builders to use, easy to write grammatical specifications for programming languages, without having to learn

00:27:25.000 --> 00:27:29.000

the details for constructing the purchasers for them.

00:27:29.000 --> 00:27:35.000

Yeah Can its descendants like bison are still widely used for creating passers.

00:27:35.000 --> 00:27:44.000

I might mention it was Jeff Olson who coined the name Yak it stood for yet another compiler compiler

00:27:44.000 --> 00:27:49.000

Lexcen yeah it can be used together to easily create the front end of a competitor.

00:27:49.000 --> 00:28:07.000

As an example, here's a yak specification for a desk calculator that evaluates arithmetic expressions involving the addition and multiplication of numbers by compiling this Yak specification, with the Lex lexical analyzer created from the previous Lex

00:28:07.000 --> 00:28:13.000

specification for integer numbers, we get a working desk calculator.

00:28:13.000 --> 00:28:21.000

Excuse me.

00:28:21.000 --> 00:28:24.000

knows runs mentioned.

00:28:24.000 --> 00:28:41.000

It has become very easy for students and compiler courses to create compilers using tools like Lex and yak, and I will see a few more words about my experiences and teaching students how to use these tools.

00:28:41.000 --> 00:28:56.000

Let us now look at photographs and important declarative abstraction that is often used in the code optimization phase of the compiler flow graphs are used to model the flow of control and computer programs.

00:28:56.000 --> 00:29:00.000

also at the 1973 powerful conference.

00:29:00.000 --> 00:29:15.000

Gary killed all presented a paper in which he showed how flow graphs could be used to facilitate the implementation of useful code optimizations on the intermediate representation of programs, but computing reaching definitions.

00:29:15.000 --> 00:29:21.000

So let's look at reaching definitions for a minute.

00:29:21.000 --> 00:29:40.000

We say the definition of a variable v from a statement s that defines the variable reaches a statement T, if there is an execution path from the statement s to the statement T, in which the variable b is not redefined the reaching definitions problem

00:29:40.000 --> 00:29:50.000

is to determine what definitions may reach what program points in the intermediate representations.

00:29:50.000 --> 00:30:07.000

The flow graph abstraction for reaching definitions, the data model is a semi lattice of subsets of variable definitions and the intermediate representation of the program, together with a directed graph represent the pool of control within the program,

00:30:07.000 --> 00:30:26.000

the nodes are the directed graph or the basic blocks of the program. And the edges indicate which blocks can follow which other blocks associated with each block or two sets of variable definitions, one representing the definitions that can reach the

00:30:26.000 --> 00:30:26.000

beginning of a block, and the other set of definitions that are live at the end of the block.

00:30:26.000 --> 00:30:40.000

of a block, and the other set of definitions that are live at the end of the block. There are two operations in the flow graph abstraction for reaching definitions.

00:30:40.000 --> 00:30:49.000

One is a transfer function that computes the data flow values at the end of the basic walk from the data flow values at the beginning of the block.

00:30:49.000 --> 00:31:05.000

The other is a confluence operator that computes the data flow values associated with the beginning of the basic block from the data flow values at the ends of all the predecessors have that basic block and the flow graph.

00:31:05.000 --> 00:31:20.000

The flow graph abstraction allows a number of different algorithms, with different running times to you to be used for solving the data flow equations associated with reaching definitions.

00:31:20.000 --> 00:31:40.000

I won't go into the details of this algorithm, because you can find the details of this algorithm and many other useful abstractions and their associated algorithms for compiler design in a sequence of books on comp compiler design that Jeff all and like

00:31:40.000 --> 00:31:43.000

author and then span of three decades.

00:31:43.000 --> 00:31:49.000

The first book by Jeff and me was published in 1977, and it had 600 pages.

00:31:49.000 --> 00:31:57.000

The second book in 1986 added Robbie city as a co author and had 800 pages.

00:31:57.000 --> 00:32:09.000

The third book in 2007 added Monica Lem as a fourth author co author and it had 1000 pages, none of us had the heart to write a fourth book.

00:32:09.000 --> 00:32:22.000

These books quickly became known as the dragon books, because all of the covers had a drawing of a fierce dragon conceived by Jeff, symbolizing the complexity of compiler design.

00:32:22.000 --> 00:32:32.000

I might mentioned, the Red Dragon book was briefly shown in the 1995 movie hackers, starring, a young, Angelina Jolie.

00:32:32.000 --> 00:32:42.000

When my two children, sold the movie. For the first time, They thought they're all men was really something.

00:32:42.000 --> 00:32:48.000

The impact of abstractions and algorithms presented in these books has been substantial.

00:32:48.000 --> 00:32:54.000

They have transformed the field of compiler design from an art to a science.

00:32:54.000 --> 00:32:59.000

And just to reiterate what rent said in the introduction.

00:32:59.000 --> 00:33:07.000

The first Fortran compiler built by IBM and the 1950s took 18 experienced staff years.

00:33:07.000 --> 00:33:27.000

In contrast, starting from the mid 1970s on scores of innovative and lead innovative languages like Brian current ahead, and we're into cherries EQN language for type setting equations have been created in just weeks, using the tools and abstractions

00:33:27.000 --> 00:33:30.000

documented in the dragon books.

00:33:30.000 --> 00:33:37.000

Don Knuth used to EQ and style mathematics and is widely used tech type setting systems.

00:33:37.000 --> 00:33:47.000

The books have also been used for decades and universities around the world to teach competitors two generations of computer science students.

00:33:47.000 --> 00:33:58.000

Let me mention an anecdote about my experiences teaching the one semester compilers course to thousands of students at Columbia, for the last 25 years.

00:33:58.000 --> 00:34:09.000

In addition to teaching the abstractions and algorithms used in compilers competitors ahead students form teams of five for semester long project that accompany the course.

00:34:09.000 --> 00:34:16.000

Each team had to create an innovative programming language of their own, and write a compiler for it.

00:34:16.000 --> 00:34:25.000

But I mentioned the semester long project at the beginning of the course, the students said this professor is out of his gourd.

00:34:25.000 --> 00:34:36.000

Not only does he expect us to create a new programming language, but he wants us to write a compiler for it, but

00:34:36.000 --> 00:34:49.000

I might mention that in 25 years that I taught the compilers course at Columbia, never did a team of students failed to deliver a working compiler for their innovative programming language.

00:34:49.000 --> 00:35:05.000

I attribute this success to the tools abstractions and algorithms that we documented in the dragon, and the dragon compiler books, and to the lightweight software engineering process that I learned working at Bell Labs and Bell core.

00:35:05.000 --> 00:35:12.000

I should mention that many of the languages created by the students far exceeded my expectations.

00:35:12.000 --> 00:35:26.000

And as a hint to those people who teach. I found that when I started teaching the compilers course I gave the students, a little language for which to write a compiler.

00:35:26.000 --> 00:35:36.000

And the students immediately came board and said, Why do we want to create a compiler for this stole language that the professor has creating created for us.

00:35:36.000 --> 00:35:45.000

But when I asked the students to work and small teams to create their own innovative programming language of their own design.

00:35:45.000 --> 00:35:56.000

The students quickly said this is the best course it ever taken, because the process of designing the language, allow the students to get to know one another.

00:35:56.000 --> 00:36:07.000

And they learned, just as much by interacting with each other as they did from the lectures that I gave him the course.

00:36:07.000 --> 00:36:18.000

One memorable language that was created and the course was called upbeat with it users could write programs to solidify streaming input data.

00:36:18.000 --> 00:36:31.000

The students who created upbeat demonstrated their language by writing an upbeat program that would play user selected music triggered by user specified patterns in the input data.

00:36:31.000 --> 00:36:45.000

They illustrated their program by taking the New York Stock Exchange ticker tape feed and having their compile target program play music fitted to the patterns and the ticker tape data, when the market was going up.

00:36:45.000 --> 00:37:00.000

The music was bouncy and lively, when the market was going down, it was gloomy and somber their theory was that a stockbroker would rather listen to music, that would indicate which way the market was going, rather than than staring at the ticker tape

00:37:00.000 --> 00:37:01.000

all day.

00:37:01.000 --> 00:37:12.000

I might mention that one of the students on the upbeat theme was Adrian Willer, who was a stockbroker stockbroker before he came to Columbia as a PhD student.

00:37:12.000 --> 00:37:20.000

Adrian is now a principal research fellow at the University of Cambridge.

00:37:20.000 --> 00:37:23.000

For the last part of my talk.

00:37:23.000 --> 00:37:29.000

I like to mention a few unusual abstractions and quantum programming languages.

00:37:29.000 --> 00:37:44.000

To show abstractions that are vastly different from abstractions appearing in traditional programming languages. However, the point I like to make is the framework and tools that we've created for designing compilers for traditional programming language

00:37:44.000 --> 00:37:54.000

can be applied equally well to create compilers for quantum programming languages. Even with these weird abstractions.

00:37:54.000 --> 00:38:10.000

I will begin by reviewing the four postulates of quantum mechanics, a fundamental theory in physics developed in the early part of the 20th century that describes the physical properties of nature at the scale of atoms and subatomic particles from these

00:38:10.000 --> 00:38:14.000

postulates we can derive

00:38:14.000 --> 00:38:19.000

the laws.

00:38:19.000 --> 00:38:29.000

Underlying quantum quantum computing and quantum computation.

00:38:29.000 --> 00:38:40.000

And I'm going to use these postulates to derive the abstraction of quantum circuits. One of the key models of computation underlying quantum programming languages.

00:38:40.000 --> 00:38:50.000

And I'm going to follow the treatment in Nielsen chunks classic book quantum computation and quantum information.

00:38:50.000 --> 00:39:09.000

The first postulate states that the state of an isolated physical system can be described by a unit vector in a Hilbert space, which is a complex vector space with an inner product from this postulate, we can derive the notion of a quantum bit cold a

00:39:09.000 --> 00:39:15.000

a cubit, which is a unit vector in a two dimensional Hilbert space.

00:39:15.000 --> 00:39:33.000

The state of a cubit sigh can be represented as shown by a linear combination of the computational basis states kept zero, and kept one where the coefficients alpha and beta or complex numbers, such as the squares of their absolute.

00:39:33.000 --> 00:39:49.000

Some of those squares of their absolute values is one, the strings, surrounded by the funny brackets represent vectors, called cats, using Paul using Paul directs widely used broadcast location in quantum mechanics.

00:39:49.000 --> 00:40:08.000

Catch zero represents the unit vector, one comma zero, and get one, the unit vector, zero comma one, these two vectors are computational basis vectors of a complex two dimensional Hilbert space.

00:40:08.000 --> 00:40:14.000

It is important to note that a cubit is fundamentally different from a bit in classical computing.

00:40:14.000 --> 00:40:18.000

It represents a quintessential property of quantum mechanics.

00:40:18.000 --> 00:40:23.000

The state of a cubit is a superposition.

00:40:23.000 --> 00:40:34.000

A complex linear combination of two basis states

00:40:34.000 --> 00:40:44.000

postulate two states that the evolution of the state of the coolest quantum system from one time to another can be represented by a unitary operator.

00:40:44.000 --> 00:41:04.000

And for the purposes of this talk, you can think of a unitary operator, just as a complex square matrix whose rules and columns are also normal, a unitary operator represents a rotation of the state vector in Hilbert space unitary operators are reversible.

00:41:04.000 --> 00:41:13.000

In contrast, operators in classical programming languages, do not have to be reversible.

00:41:13.000 --> 00:41:22.000

The single qubit had a mark operator is a useful unitary operator that appears in many quantum algorithms.

00:41:22.000 --> 00:41:34.000

If we apply the head of art operator to the computational basis state kept zero, we could get a quantum state, which is superposition. Catch zero plus get one divided by the square root of.

00:41:34.000 --> 00:41:35.000

to.

00:41:35.000 --> 00:41:42.000

And if we apply it to the head of smart operator to get one, we get

00:41:42.000 --> 00:41:50.000

the superposition cat zero minus cap one divided by the square root of two.

00:41:50.000 --> 00:42:06.000

The coo qubit control NOT operator, see not for short, is also used in many quantum algorithms. It has to input cubits, namely a control qubit C and a target qubit T.

00:42:06.000 --> 00:42:17.000

It flips the value of the target qubit if and only if the control qubit has the value one, the value of the control to but is not changed.

00:42:17.000 --> 00:42:22.000

So here you see if the sea is equal to the cube.

00:42:22.000 --> 00:42:25.000

One, and t is equal to the cube and Cheryl.

00:42:25.000 --> 00:42:34.000

Then on the output side, see retain so that you get one. But the target qubit is now.

00:42:34.000 --> 00:42:36.000

Then flip to one.

00:42:36.000 --> 00:42:45.000

The some mode to have zero and one.

00:42:45.000 --> 00:43:03.000

postulate three says the only way to get information out of a quantum system is to measure it, a measurement returns an outcome with some probability, the some of the some of the probabilities of the possible outcomes as one, a measurement is not a unitary

00:43:03.000 --> 00:43:07.000

operator because it collapses the state of a quantum system.

00:43:07.000 --> 00:43:25.000

And there are many different kinds of measurements that can be used as an example, let us consider the project of measurement of a single qubit sigh equal to alpha times kept zero plus beta times get one.

00:43:25.000 --> 00:43:36.000

You would like to get the dudes of size and outcome. But in the world of quantum physics, you can only get one classical bit a zero or one. As an outcome.

00:43:36.000 --> 00:43:47.000

The probability of getting zero is the absolute value of alpha squared, the probability of getting one is the absolute value of beta squared.

00:43:47.000 --> 00:44:08.000

Moreover, if the outcome is one, the state after measurement has collapsed to catch zero. Likewise, if the outcome is one, the state after measure measurement has collapsed to get one and measurement is a quintessential bottleneck to quantum computing,

00:44:08.000 --> 00:44:21.000

because you can only get a classical bit as output from any measurement that you can produce.

00:44:21.000 --> 00:44:32.000

The fourth postulate says the stage space of a composite system is a tensor product of the state spaces of its components systems.

00:44:32.000 --> 00:44:41.000

And just as an illustration in this slide. I've shown that if we take the tensor product of the two, two dimensional vectors, and be.

00:44:41.000 --> 00:44:44.000

Then we get a four dimensional vector.

00:44:44.000 --> 00:44:54.000

If we take the tensor product of the Three, two dimensional vectors A, B and C. we get an eight dimensional vector.

00:44:54.000 --> 00:45:03.000

What postulate for shows is that if we add a single cubit to a quantum system. We've doubled the dimension of it state space.

00:45:03.000 --> 00:45:12.000

So if we combine and single qubit quantum systems, we're going to kind of system with estate space of dimension to the end.

00:45:12.000 --> 00:45:23.000

Note that if we combine just 300 single qubit systems, we get a state space with more dimensions than the number of atoms in the universe.

00:45:23.000 --> 00:45:34.000

It's this aspect of quantum mechanics that makes the simulation of quantum systems computationally difficult.

00:45:34.000 --> 00:45:43.000

Okay we are finally ready to introduce quantum circuits. The key abstraction and quantum computing.

00:45:43.000 --> 00:45:51.000

Quantum circuits are sometimes called the gate model of quantum computing.

00:45:51.000 --> 00:46:13.000

They are a little bit like Boolean circuits, but there are important differences, a quantum circuit is an a cyclic graph consisting of lines connecting unitary operators and measurement gates.

00:46:13.000 --> 00:46:27.000

No fan out is allowed, either because of the Themis no cloning theorem of quantum mechanics, which states that you cannot make a copy of an unknown quantum state.

00:46:27.000 --> 00:46:48.000

Here is a simple but important example that illustrates what a quantum circuit looks like it has to input lines, x&y, carrying the input cubits, the x line goes into a head of art gate and the output of the head of art gate and the white line, go into

00:46:48.000 --> 00:46:53.000

a see NOT gate, as its control and target cubits respectively.

00:46:53.000 --> 00:47:04.000

This quantum circuits generate to know put what are called Einstein Podolski states, also known as spell states.

00:47:04.000 --> 00:47:14.000

These states have another quintessential property of quantum mechanics and experimentally verified phenomenon known as entanglement.

00:47:14.000 --> 00:47:34.000

As an example, if the cubits x&y both had value catch zero, then the output of the circuit has a value kept 00 plus cat one one divided by the square root of to the state has a property that it cannot be expressed as a tensor product have to basis state,

00:47:34.000 --> 00:47:38.000

such a state is said to be entangled.

00:47:38.000 --> 00:47:47.000

Einstein called the phenomenon of entanglement and quantum mechanics spooky action at a distance.

00:47:47.000 --> 00:48:04.000

I've shown on this slide, what happens to the inputs cats 0001 cat 1011. What gets produced the output and all those four states are entangled states

00:48:04.000 --> 00:48:22.000

in 1994 quantum computing recently received an enormous spools when Peter shore at Bell Labs published an algorithm combining a classical computer than a quantum computer to factor integers, with order and cube operations.

00:48:22.000 --> 00:48:34.000

Many of today's encryption schemes, like RSA on the internet. Assume factoring large integers is computational computationally prohibitive.

00:48:34.000 --> 00:48:41.000

We do not know how to factor integers, in polynomial time on a classical computer.

00:48:41.000 --> 00:48:52.000

However, if we could build large scalable quantum computers with error correction encryption schemes like RSA would be easy to crack with shores algorithm.

00:48:52.000 --> 00:49:08.000

But for the time being. No polynomial time algorithm is known for factoring integers, on a classical computer, and we cannot build large enough quantum computers to factor the sizes have been to introduce us by our essay in practice.

00:49:08.000 --> 00:49:10.000

So RSA is safe for now.

00:49:10.000 --> 00:49:26.000

However, as a precaution NIST has recently initiated a process to solicit proposals develop cryptographic systems that are secure against attacks for both quantum and classical computers.

00:49:26.000 --> 00:49:41.000

Several companies like Google and IBM have already built quantum computers with around 100 or so cubits, but to implement shores algorithm with full fault tolerance and error correction will require a million or more cubits.

00:49:41.000 --> 00:49:59.000

It is not clear when such large scale come quantum computers will become available from a theoretical point of view, if we could prove factoring integers cannot be done in all you know real time on a quantum on a classical computer.

00:49:59.000 --> 00:50:16.000

Then shores algorithm would disprove the extended church touring thesis, which states that any physically realizable digital model of computation, which can be efficiently simulated can be efficiently simulated on a Turing machine.

00:50:16.000 --> 00:50:35.000

So, determining the true complexity of factoring is still a great unsolved problem in computer science and mathematics,

00:50:35.000 --> 00:50:52.000

scalable quantum computers are still under their infancy, but our abstractions and algorithms for building compilers for traditional programming languages have been very successful at constructing compilers for quantum computing programming languages.

00:50:52.000 --> 00:51:01.000

A little while ago, Christmas for a graduate of Princeton University came to Columbia as a PhD student to study quantum computer.

00:51:01.000 --> 00:51:09.000

After taking my compilers course, she decided to do her PhD thesis songwriters and design tools for quantum computing.

00:51:09.000 --> 00:51:22.000

As part of her thesis. Krista demonstrated the framework we've created for implementing compilers for classical programming languages can also be used to develop compilers for quantum programming languages.

00:51:22.000 --> 00:51:25.000

After graduating from Columbia.

00:51:25.000 --> 00:51:31.000

Krista joined Microsoft Research, where she has no general manager of quantum software.

00:51:31.000 --> 00:51:48.000

A few years ago she and her colleagues at Microsoft, created a quantum programming language called q sharp. That is no part of Microsoft's quantum development kit, which can be used to experiment with quantum gal rhythms and design tools to sharp is one

00:51:48.000 --> 00:52:00.000

of 15 Quantum programming languages listed on Wikipedia is quantum programing web page.

00:52:00.000 --> 00:52:17.000

To conclude, I would like to reiterate my thesis that abstractions and algorithms, the two pillars of computer science and computational thinking, have transformed the field of compiler design from an art to a science.

00:52:17.000 --> 00:52:30.000

And I leave us an open question to determine what new fields of human endeavor, are there, that can be mechanized with abstractions and algorithms for the benefit of humankind.

00:52:30.000 --> 00:52:34.000

Thank you very much.

00:52:34.000 --> 00:52:42.000

Thank you very much. first right here this was an inspiring talk and I and I.

00:52:42.000 --> 00:52:57.000

There's a lot for us to learn from what you just told us. Um, so we have time for questions now. And please use the q amp a feature to type in your questions and what I'll do then is all related questions to do our speaker.

00:52:57.000 --> 00:53:16.000

Um, so we have one question already from Jason Hong. It relates to machine learning. So his question is machine learning seems to be changing a lot of software development from deterministic to statistical where we train system to have a behavior, rather

00:53:16.000 --> 00:53:18.000

than building or specifying it directly.

00:53:18.000 --> 00:53:25.000

What do you think are the implications for compilers, and for teaching people about computational thinking.

00:53:25.000 --> 00:53:40.000

This is a great question. And if you look at some of the work that's being done in using machine learning to create target programs.

00:53:40.000 --> 00:53:46.000

It looks very impressive and very suggestive.

00:53:46.000 --> 00:53:55.000

However, the work is still in a rudimentary stage of

00:53:55.000 --> 00:53:56.000

research.

00:53:56.000 --> 00:54:21.000

And one of the things that I've learned, sometimes from bitter experience is that the programmer and the compiler writer should use the same definition of the programming language and traditional programming languages like Fortran c c++, Java, and so

00:54:21.000 --> 00:54:30.000

on, have international standards that specify the syntax and semantics of the language.

00:54:30.000 --> 00:54:53.000

So the last thing you want is the programmer, and the compiler to have a different interpretation of the syntax and semantics of the program that's being written, because as you're more than well aware that software can have catastrophic consequences.

00:54:53.000 --> 00:55:09.000

As we have witnessed a with the software on the Boeing 747 control software or other devices, in which software has bad software's unfortunately killed people.

00:55:09.000 --> 00:55:19.000

And also, I have a soft, lyst, a software Hall of shame, where to get on to the software all of shade.

00:55:19.000 --> 00:55:29.000

You have to have spent a quarter of a billion dollars or more, and produced a software system. That didn't work.

00:55:29.000 --> 00:55:48.000

Unfortunately it's all too easy to construct malfunctioning software still, and one of the great research areas in computer sciences, how do we verify software, software, so that we know it performs correctly.

00:55:48.000 --> 00:56:03.000

And one of the difficulties that I see in using machine learning to create mission critical software. Is that how do you know what the program is generated by the machine learning program actually does.

00:56:03.000 --> 00:56:15.000

I understand it's an open problem still an AI and machine learning to determine how exactly does the machine learned. Learning Program, do its job.

00:56:15.000 --> 00:56:18.000

I liken this somewhat to.

00:56:18.000 --> 00:56:31.000

I'm puzzled very puzzled as to how the human brain computes interprets language thinks creates consciousness.

00:56:31.000 --> 00:56:44.000

And is this, again, a good problem for trying to understand AI better. and what an AI program actually does.

00:56:44.000 --> 00:56:58.000

I don't know but I think we still need to do a lot more work in understanding machine learning to know whether it can be reliably used to create target programs.

00:56:58.000 --> 00:57:02.000

That's my personal opinion.

00:57:02.000 --> 00:57:20.000

Read. Thank you. Um, So again we have a couple more minutes before hours up so send in your questions as you have them and, well, I'll ask a question, it's sort of related to the one you just mentioned, the one you just handled that has to do with the

00:57:20.000 --> 00:57:26.000

with the sort of novel computing hardware

00:57:26.000 --> 00:57:43.000

and advances that are happening now I have, for example, it's interesting to me to see analog computing making a comeback, at least in the IT driven in part by machine learning, considerations.

00:57:43.000 --> 00:57:59.000

And what the so called neuro morphic hardware which is relies on, you know, basic physics to do computations rather than the sort of Shannon style democratized operations.

00:57:59.000 --> 00:58:09.000

What do you see what impacts Do you see this, these new hardware developments having on language design and compilation.

00:58:09.000 --> 00:58:29.000

That again it's a very intriguing question because we really don't understand computation and all its forms. And as you point out, nature has been doing computation with models of computation that are very different from digital computers for.

00:58:29.000 --> 00:58:32.000

Well since the dawn of humankind.

00:58:32.000 --> 00:58:47.000

And I know with some interest that one of the great developments in biology was using machine learning to do protein folding better than biologists knew how to do.

00:58:47.000 --> 00:58:58.000

So, the impact of machine learning, I think is still to be felt. And it's still to be understood, a lot better.

00:58:58.000 --> 00:59:13.000

You mentioned, innovative types of hardware. One of the things I've noticed is that maybe we finally discovered the Maharani

00:59:13.000 --> 00:59:16.000

qubit

00:59:16.000 --> 00:59:29.000

topological quantum computing is very popular, theoretically, but people been trying to implement topological quantum computers.

00:59:29.000 --> 00:59:44.000

For many years, and it's only recently that people seem to have been able to create hardware that can create a Maharajah quasi particle.

00:59:44.000 --> 01:00:07.000

So, this may create a resurgence in topological quantum computing. And as you may be well aware that there is another style of quantum computing called quantum annealing where the company of the wave systems, has been purveying quantum and healers to

01:00:07.000 --> 01:00:19.000

the marketplace for more than a decade. And that represents a very different style of quantum computing, where you evolve the Hamiltonian to a solution.

01:00:19.000 --> 01:00:31.000

So, I think this is why computer science is still a terrific area because we have all these different forms of computing, to understand better and to harness.

01:00:31.000 --> 01:00:34.000

So, great question.

01:00:34.000 --> 01:00:53.000

Read. Thank you. We have a another couple of questions that have arrived in the q amp a will, will take these and then I'll will let Professor he'll go eat a sandwich, so that he can rest up for the next part of his virtual visit, which is the 3pm meeting

01:00:53.000 --> 01:00:59.000

with the program directors, which I encourage all program directors to attend if you're able to.

01:00:59.000 --> 01:01:16.000

So, next question.

01:01:16.000 --> 01:01:22.000

Well, if I can give an indulgent answer to this question.

01:01:22.000 --> 01:01:50.000

Back in my youth I Ryan current ahead and Peter Weinberger created a little language for routine data processing problems, called AUC and AUC was one of the most popular programs on the Unix operating system in the first few years of Unix, because it

01:01:50.000 --> 01:01:52.000

was so easy to learn.

01:01:52.000 --> 01:02:07.000

It used a pattern action paradigm of computing. Human beings are stimulus response creatures. We hear a loud noise, we turn our head to see where did that loud noise, come from.

01:02:07.000 --> 01:02:18.000

So the programming paradigm and AUC is you write a sequence of patterns associated with each pattern is an action.

01:02:18.000 --> 01:02:22.000

And then you search a file of data.

01:02:22.000 --> 01:02:38.000

And for every line of the file that contains a pattern. The Associated action is executed. And what I found was that all sorts of non computer science people started using off because it was so easy to learn.

01:02:38.000 --> 01:02:58.000

In fact there was a manager at a at&t Western Electric plant that used to keep track of the steals that he was producing in this plant, because he said it was so easy to read the database system in order to keep track of the different products.

01:02:58.000 --> 01:03:01.000

His plant was producing.

01:03:01.000 --> 01:03:08.000

That may be an extreme example but the search for easy to use.

01:03:08.000 --> 01:03:11.000

programming languages continues unabated.

01:03:11.000 --> 01:03:33.000

And I've noted with great interest that there seem to be as many programming languages in the world as natural languages, there are about 7000 natural languages extent that natural languages in the ethanol log database, and I want saw a website that had

01:03:33.000 --> 01:03:46.000

pointers to about 7000 programming languages, so maybe there's something in the human condition that requires humanity to have 7000 have anything until they're satisfied.

01:03:46.000 --> 01:04:05.000

But I think it's a good question on what to teach. I might mention that when I taught the compilers course. I mentioned to the students at the beginning of the course that in addition to learning the principles of abstractions, and algorithms of compiler

01:04:05.000 --> 01:04:17.000

designed, you're going to learn three important life skills in doing that project associated with the course, the first port and life skill was project management.

01:04:17.000 --> 01:04:22.000

Each team has to have a project manager. And the reason.

01:04:22.000 --> 01:04:39.000

Every team produced a working compiler. At the end of the course was one of the ingredients of the grade for the project was before the project manager was timely completion of the deliverables associated with the project.

01:04:39.000 --> 01:04:49.000

So the compiler, the project manager, made sure that by the end of the semester. There was a working compiler.

01:04:49.000 --> 01:04:55.000

Another important life skill was teamwork.

01:04:55.000 --> 01:05:12.000

The students who formed the team were often strangers at the beginning of the course, but they had to learn how to create a software, non trivial software system in a distributed fashion in 15 weeks.

01:05:12.000 --> 01:05:23.000

And for that they needed to learn how to work well with one another. And the third life skill was communication, both oral and written.

01:05:23.000 --> 01:05:45.000

And as I mentioned that one of the things that is required in compiler business is you need a specification of the programming language that the programmer can use, and you need to use the same specification, when you build a compiler.

01:05:45.000 --> 01:05:48.000

So,

01:05:48.000 --> 01:06:05.000

communication amongst the team is a very important aspect, and what I did was I had the students write a specification for their language, and a user manual for their language, before they even began implementing their language.

01:06:05.000 --> 01:06:11.000

What uses there and creating a programming language that you can teach others to us.

01:06:11.000 --> 01:06:28.000

So these are. I thought some of the great benefits of the compiler course. And of course you learn abstractions and algorithms, but now with compiler frameworks like Ll BM generating the compiler is more an exercise and software engineering, because he

01:06:28.000 --> 01:06:47.000

just generate a new front end or a new back end, and inserted into lol BM rather than completely writing a compiler from scratch. But there is a great aha experience for students when they see their compiler working for the first time, and the team celebrates.

01:06:47.000 --> 01:07:00.000

So, the human aspect of creating a distributed piece of software as a team. I think was a very important component of the course.

01:07:00.000 --> 01:07:14.000

Read. So we have time for one more, one more question I know there are other questions in the chat I'm sorry we will be able to get to those but, um, the next question from Kumar balance hundred is as follows.

01:07:14.000 --> 01:07:24.000

Can you comment on the equivalence of data and function, and how our compiler would handle abstractions that allow the data that changes the function and vice versa.

01:07:24.000 --> 01:07:29.000

That seems to be the way biological systems behave, but something we avoid in practice.

01:07:29.000 --> 01:07:33.000

Oh, another great question. I love these questions.

01:07:33.000 --> 01:07:43.000

My favorite abstract programming language and not so abstract programming language is the lambda calculus.

01:07:43.000 --> 01:07:47.000

And then the lambda calculus.

01:07:47.000 --> 01:08:07.000

It's the pure functional programming language, and the separation between data, and function is very elegantly handled. So I suggest studying the lambda calculus, to understand how data and function can be elegantly combined.

01:08:07.000 --> 01:08:14.000

Some modern languages, don't do this, nearly as elegantly.

01:08:14.000 --> 01:08:27.000

Hey, thanks. So I think that's all we have time for today I want to thank you again. First row for stimulating talk in a very nice and lively question, q amp a session.

01:08:27.000 --> 01:08:55.000

And I'll just remind the program directors that at 3pm, again we have a hour set aside for sort of an informal roundtable discussion with with our speaker.