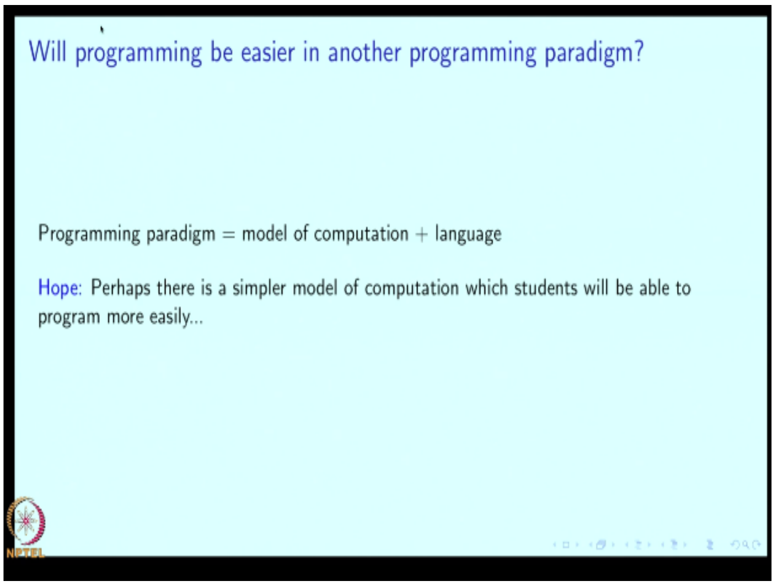
Design and Pedagogy of the Introductory Programming Course

Prof. Abhiram G. Ranade
Department of Computer Science and Engineering
Indian Institute of Technology – Bombay

Lecture - 04
Introduction and Survey.2: Alternative approaches, Summary, and Conclusion

Hello and welcome again to the course on design and pedagogy of the introductory programming course. This is the final lecture of the first part, introduction and survey. So the question that we are going to consider in this part is will programming be easier in another programming paradigm?

(Refer Slide Time: 00:43)



Will programming be easier in another programming paradigm?

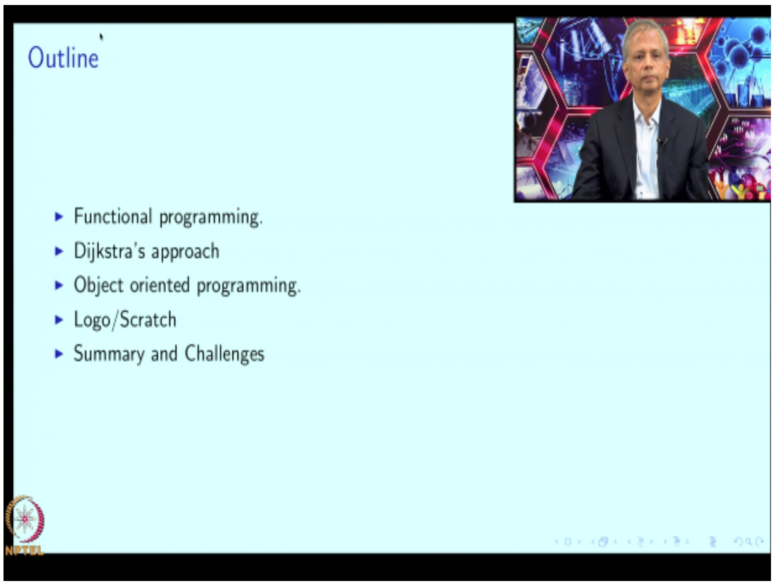
Programming paradigm = model of computation + language

Hope: Perhaps there is a simpler model of computation which students will be able to program more easily...

Programming paradigm basically means a model of computation and a language as well, but it means more a model of computation and you will see what exactly I mean by model of computation in a minute. The reason for asking this question is the hope that perhaps there is a simpler model of computation that is a model of computation which is easier to understand and therefore which will enable students to program more easily.

So as we have been seeing that programming seems to be really difficult, so it is worth looking at whether we can simplify it and so people have tried doing this and people have looked at alternate models of computation. Here is what I am going to do.

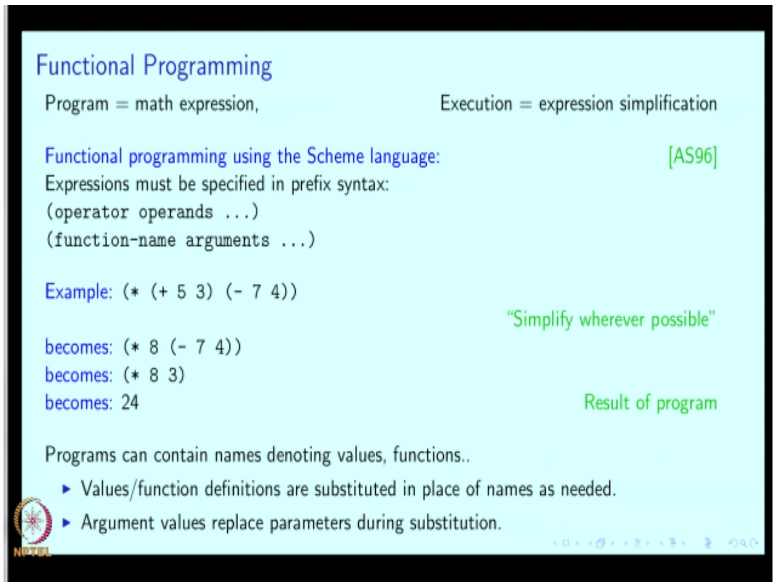
(Refer Slide Time: 01:33)


Outline

- Functional programming.
- Dijkstra's approach
- Object oriented programming.
- Logo/Scratch
- Summary and Challenges

I am going to first talk about functional programming, then I will talk about Dijkstra's approach. This is not so much a model of computation as it is a view of programming. How should programming really be taught. Then I will talk about object oriented programming and then I will talk about logo and scratch and then I will conclude with a summary of this first part and list out the challenges.

(Refer Slide Time: 02:02)


Functional Programming

Program = math expression, Execution = expression simplification

Functional programming using the Scheme language: [AS96]

Expressions must be specified in prefix syntax:

```
(operator operands ...)
(function-name arguments ...)
```

Example: `(* (+ 5 3) (- 7 4))`

"Simplify wherever possible"

becomes: `(* 8 (- 7 4))`

becomes: `(* 8 3)`

becomes: 24 Result of program

Programs can contain names denoting values, functions..

- Values/function definitions are substituted in place of names as needed.
- Argument values replace parameters during substitution.

So let us begin with functional programming. In functional programming a program is a mathematical expression. So just as 1 + 2 is a mathematical expression, a program is pretty much like 1 + 2 and execution is expression simplification. So if I have an expression 1 + 2, I simplify it and make it 3. So something is changing, something is evolving and that is exactly captured in functional programming and that is thought of as an execution of a program.

Functional programing can be done by using several languages and only one such language is the scheme language. Scheme language is described in a really nice book by Abelson and Sussman written in 96. It is a beautiful book and I am going to use this notation. In this notation that the scheme language represents expressions must be represented in prefix syntax. So prefix means the operators must come first followed by the operands.

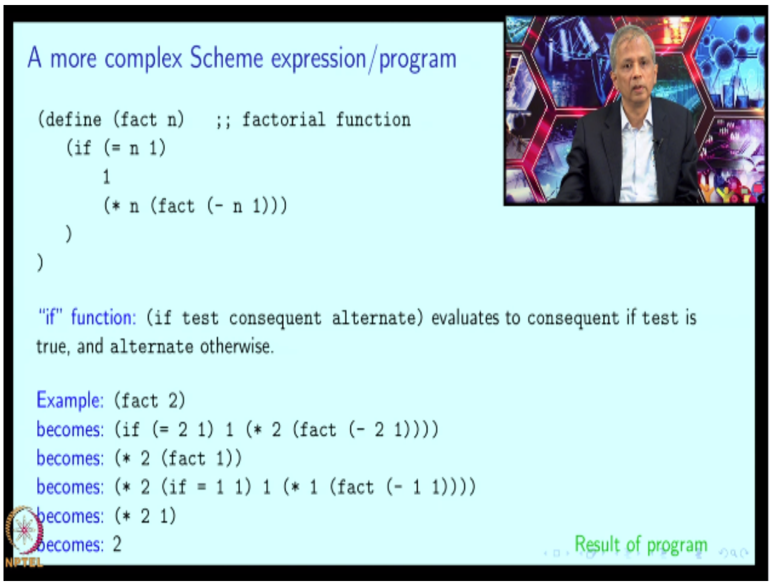
So operator followed by operands enclosed in parenthesis or function-name followed by arguments enclosed in parenthesis. So here for example is an expression. So this expression is to be read as sum of 5 and 3 multiplied by the difference of 7 and 4. So + is an operator over here and these are the operands and the result of this are multiplied together. Okay so what do we do with this.

The basic idea is we are going to simplify wherever possible. So for example + 5 3 can be simplified to 8. So in that case the entire expression now changes and it becomes times 8 – 7 4. So this is to be thought of as an execution. We are doing computation, and we are executing and the expression is changing. So in fact we can think of it as we have a program that has been given to us and we sort of keep on simplifying that program.

Then this expression in turn can become * 8 3 because 7 - 4 is 3 and then that becomes 24 because 8 times 3 is 24 and this 24 is the result of that entire programme. So as you can see evaluation is something that you already know. A program is something that you already know and therefore this is considered to be a really attractive model of how to describe programs. Of course this is not all there is to it.

So 2 things I need to tell you more. So programs can contain names denoting values as well as functions. So wherever there is a name it is replaced by the definition. So if there is a name denoting a value, the value will replace that name. If it is name denoting a function, then the function definition is substituted in place of the name whenever needed and of course if it is a function definition then the argument values replace parameters during substitution. So we will see an example of this right away.

(Refer Slide Time: 05:46)



A more complex Scheme expression/program

```
(define (fact n)   ;; factorial function
   (if (= n 1)
       1
       (* n (fact (- n 1)))
   )
)
```

"if" function: `(if test consequent alternate)` evaluates to consequent if test is true, and alternate otherwise.

```
Example: (fact 2)
becomes: (if (= 2 1) 1 (* 2 (fact (- 2 1))))
becomes: (* 2 (fact 1))
becomes: (* 2 (if = 1 1) 1 (* 1 (fact (- 1 1))))
becomes: (* 2 1)
becomes: 2
```

Here is a more complex scheme expression or program. So this program is just a definition, but let us read it first. This is an expression which defines a function fact. Fact is the name of the function. It takes one argument n and it is supposed to compute the factorial. Inside it there is an expression if. There is if function. So what does the if function do. If function has 3 parts to it.

The first is the test, the next is the consequent and then the last is the alternate. So for example in this if, this part is the test, okay. This test is going to check is n = 1. This part is the consequent and this part is the alternate. So if n = 1 then this entire expression represented by the if statement is going to be, this entire expression is going to be resulting in 1. If n is not equal to 1 then this entire expression is going to be resulting in this expression okay.

So let us see now how an expression which cause the factorial function will execute. So suppose we have written down an expression fact of 2 what happens. So as we said in the last slide the name fact is going to be replaced by the definition of fact and 2 is the argument and 2 will replace the parameter in the function. So this will become, so this entire function body is going to be placed over here.

So notice that n is 2. So if instead of, if = n1 we will write = 2 1. So 1 is going to come as such and then times n fact of –n1. So everywhere n is going to be replaced by 2. Okay, now this expression that we started off with has now become this expression. It looks longer, but actually it is simpler because we have substituted for a definition. What happens next well the idea is that wherever we can simplify we will. So for example in this case we can simplify this expression is 2 = 1.

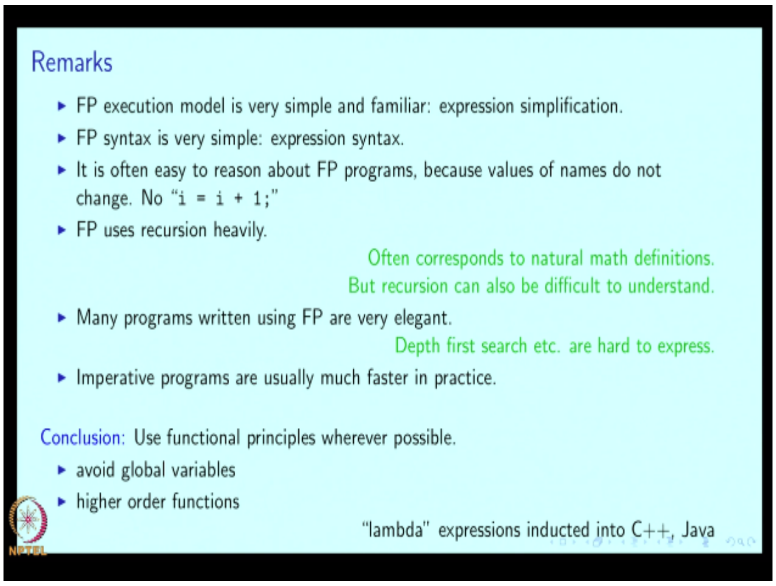
So it is not equal to 1 so which means we can simplify this entire expression. So if this 2 is not equal to 1 then it will evaluate to the alternate. So it will evaluate to this. So as a result this big expression will become just the alternate which is over here. Now in this can we evaluate anything? well we cannot. So therefore we will substitute for fact of 1. So fact of 1 again will go back to the definition and will put the body of the function inside this.

So this will be our new expression. We are going to have n, this n is going to have the value 1 because we called fact with 1. So our expression is going to be if = 1 1 okay, so there should be a parenthesis over here checking = 1 1 then it should be a 1 otherwise it should be this expression. Now we can easily check that = 11 is true and therefore this time the consequent is returned as a result.

So our new result is going to be times 2 1. So this now can be evaluated and it has become 2. So you see what is going on. We have expressions and we are only simplifying the expressions and of course we are substituting for function definitions wherever be needed, but that all those seem like standard mathematical operations. All those seem like students already know this.

So the hope over here is that since students already know simplification of expressions they will be able to understand programs or they will be able to even write programs in a much more easy fashion.

(Refer Slide Time: 10:19)



Remarks

- FP execution model is very simple and familiar: expression simplification.
- FP syntax is very simple: expression syntax.
- It is often easy to reason about FP programs, because values of names do not change. No "`i = i + 1;`"
- FP uses recursion heavily.

  Often corresponds to natural math definitions.
  But recursion can also be difficult to understand.
- Many programs written using FP are very elegant.

  Depth first search etc. are hard to express.
- Imperative programs are usually much faster in practice.

Conclusion: Use functional principles wherever possible.

- avoid global variables
- higher order functions

  "lambda" expressions inducted into C++, Java

So first remark which I just made, functional programming executional model is very simple and familiar, expression simplification. It is syntax is very simple, expression syntax and here is one more interesting point that it is often very easy to reason about functional programming programs. So scheme programs for example, because just as in mathematics the value of the name never changes.

Even here if I have a name n, it always means the same thing. Well in a different context it might mean something different, but within a given context I cannot do anything like i = i + 1. So wherever I see, n, I know exactly what it is referring to. So this is another idea which is considered to be easy, to make functional programming easy. Functional programming uses recursion heavily.

In fact, there is no iteration as such. Recursion is sort of a standard way in which iterative actions happen. You may note that we can always use recursion to implement iteration though not exactly vice versa. So in any case in functional programming we use recursion and very often it corresponds to whatever happens in mathematics. Many mathematical definitions use induction.

So many definitions are inductive and functional programming correspondingly uses recursion. Now recursion is maybe very natural, but recursion can also be a little bit tricky to understand. Many programs written using functional programming are very elegant. They can often be very small, very compact okay. However, there are some classes of programs where this notion of just using recursion becomes very tricky.

Or the idea that we cannot change the, we do not change the value of a variable, becomes very tricky. So for example if you are doing depth first search on graphs, it is quite hard to express this using functional programing. Finally, it should be noted that imperative programs are usually much faster in practice. So what do we conclude and what has the field concluded. So functional programing is not accepted worldwide.

It does not have that much acceptance but there are some very, very strong proponents. So what we can say is that we have learnt something valuable about programming when we study functional programming and we should perhaps be using those principles whenever possible. So what are those principles? So we should not be using global variables. So the idea here is like what has been mentioned above that we want to make it easy to understand programs.

So then if we have a name we should expect that name to have the same value everywhere. If we have a global variable, then a function call can change the value of a global variable and so we should avoid it. So whenever we write a statement like i = i + 1 we should regard that as a potential mental overhead.

So i = i + 1 is not possible in functional programming and that is because that imposes a mental overhead and we should even avoid things like this and global variables are sort of a very strong example of that where the name for which we thought a certain value assigned suddenly changes and especially if the change is happening inside some other function it is very difficult to understand what is going on.

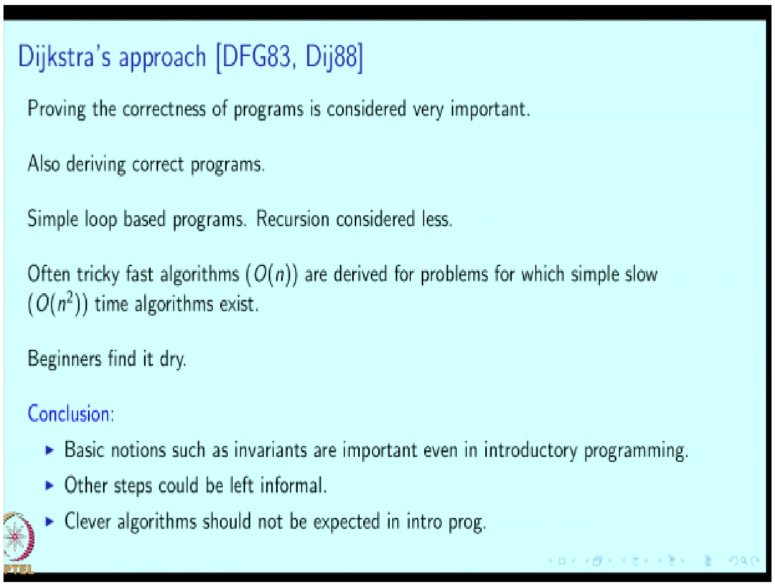
Functional programming also encourages functions themselves to be passed as arguments. In fact, functional programming does not make a distinction between functions and variables. The term used over here is that functions are first class citizens. Early programming languages, early popular programming languages had a very, very big distinction between functions and ordinary variables, but now things are changing.

An idea is like higher order functions which sort of evolved in functional programming are now coming into standard programming languages. So for example the lambda expression which is an important idea that arises in functional programming which I have not had the time to go through but this idea has been inducted into C++ java and other standard programming languages.

So what we are going to say is and how we are going to proceed is that we are not really going to use functional programming, but wherever possible we will try to use functional programming principles. So if it makes sense we will remember that we can pass a function. We can pass different functions as arguments to another function and inside that function, the function that we pass might get called or we might make sure that we do not use global variables.

So these are some of the lesions that we can take away from functional programming, but we are not going to use functional programming. We are not going to change over to functional programming.

(Refer Slide Time: 16:35)



Dijkstra's approach [DFG83, Dij88]

Proving the correctness of programs is considered very important.

Also deriving correct programs.

Simple loop based programs. Recursion considered less.

Often tricky fast algorithms ($O(n)$) are derived for problems for which simple slow ($O(n^2)$) time algorithms exist.

Beginners find it dry.

Conclusion:

- Basic notions such as invariants are important even in introductory programming.
- Other steps could be left informal.
- Clever algorithms should not be expected in intro prog.

Next we will discuss an approach due to Dijkstra. This is not quite a programing model, but it is an approach in which Dijkstra says that proving the correctness of programs is extremely important and also deriving complete programs. So the idea over here is you are given the statement of the problem, you decide what values we need to calculate and then you say okay if I want to calculate these values well then I must have these values.

Or if I have these values, I can calculate these values and use step by step in a logical fashion you decide what you can calculate, what needs to be calculate and if the two meet if what you can calculate and what needs to be calculate at some point become equal then you have a program, but the idea is that at each step you have a proof that if I can calculate this I can calculate this next thing or if I need to calculate this it is a (()) (17:30) if I calculate this.

So at every point you have a proof. Now simple loop based programmes are derived and recursion is not considered so much in this approach, but there are variations on this approach or there is work which looks at recursion as well and this approach can be considered successful in the sense that just by reasoning in this manner and sticking to having proofs for whatever you do.

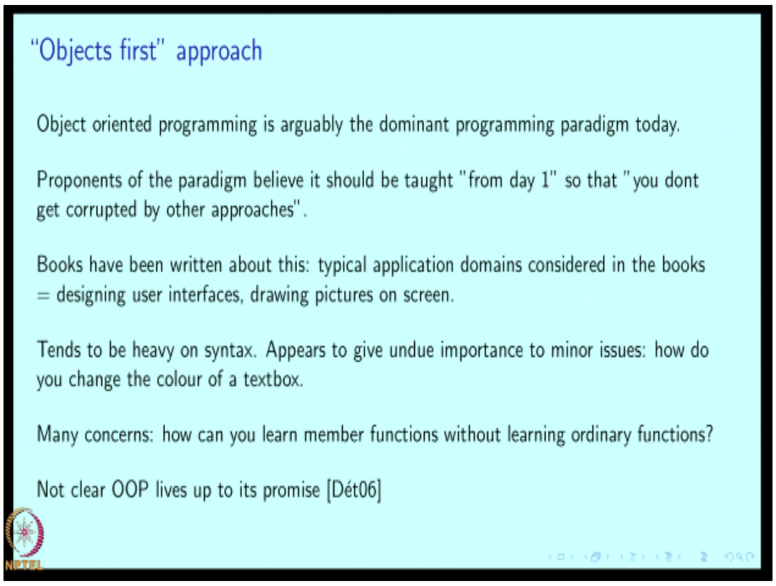
You can often derive quite tricky and fast algorithms so often time linear time algorithms are derived for problems in which simple algorithms might give you n squared time. Unfortunately, the kind of logical manipulation that is required in this approach tends to be a little bit cumbersome and beginners find it dry and this is probably the reason that this approach has not caught on.

So what is it that we conclude from this? Well, we believe that basic notion such as invariants are important even in introductory program. So invariants are a very, very strong idea that comes out of Dijkstra’s approach and we feel that that is important; however, we do not want to go all the way and only derive programs or prove everything about a program very, very formally, okay.

So what we are going to take away from Dijkstra’s approach that we should be stressing invariants when we talk even about introductory programing, but perhaps other steps could be left informal and Dijkstra’s approach is often about deriving very clever algorithms and such and clever algorithms perhaps should not be expected in introductory programming. What I mean by that is we should not really expect beginners to derive clever algorithms.

Because clever algorithms will require cleverness and often we tend to think of cleverness as something that happens in a second, but before it happens in a second you need to have experience and you need to spend some time so that you develop the intuition to generate clever algorithms and there is just not enough time in the introductory programming course that we should be stressing clever algorithms.

(Refer Slide Time: 20:24)


"Objects first" approach

Object oriented programming is arguably the dominant programming paradigm today.

Proponents of the paradigm believe it should be taught "from day 1" so that "you dont get corrupted by other approaches".

Books have been written about this: typical application domains considered in the books = designing user interfaces, drawing pictures on screen.

Tends to be heavy on syntax. Appears to give undue importance to minor issues: how do you change the colour of a textbox.

Many concerns: how can you learn member functions without learning ordinary functions?

Not clear OOP lives up to its promise [Dét06]

A very common approach is the object first approach; object oriented programing is arguably the most dominant paradigm today. Now proponents of this paradigm believe that it should be taught from day 1, okay, why because otherwise you might get corrupted by other approaches. Books have been written about this and about teaching object oriented programing and typically not always the application domains considered in these books are things like designing user interfaces, drawing pictures on screen okay.

And for whatever reasons this paradigm tends to be heavy on syntax. It often appears to give undue importance to minor issues. How do you change the colour of a text box? This is of course not an inherent aspect of object oriented programing, but somehow when it is (()) (21:19) these are the issues that get importance; however, there are many concerns. So in some sense object oriented programming is not an alternative paradigm for something like C based programming or the so called imperative style programming.

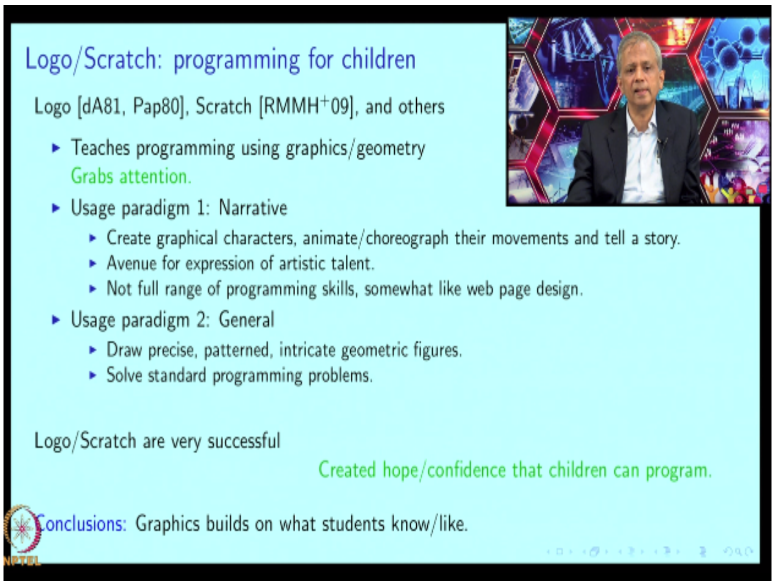
It is something in addition to introductory programming or to protect more directly how can you learn member functions without learning ordinary functions, are not member functions necessarily more complicated than ordinary functions. So in member functions there is a privileged argument. There is sort of a special argument which is the object on which you are invoking the function.

There is no such special or privileged argument in ordinary functions. So to that extent ordinary functions are easier. So when we talk to beginner should not we be teaching them the ordinary functions first before they can be expected to understand member functions. In general object oriented programming proponents says that it gives a more natural view of the world.

However, there are research papers and have mentioned one (()) (22:42) over here which say we have done surveys and studies on students and they has some suspicion or aspersions on whether object oriented programming lives up to it is promise. So what is our conclusion, well, we think that object oriented programming is strictly harder even if it is useful, it is strictly harder and since we seem to be in a crisis situation let us not make things more difficult for our students.

Let us first make sure that they understand ordinary functions and ordinary programing before we get into objects and all the issues related to objects, okay. The last programming paradigm that I want to survey is the so called logo and scratch paradigm. Again it is not a paradigm as such, but it is a bunch of projects or programs that have been developed very very systematically if I may say so for teaching programming to children.

(Refer Slide Time: 23:48)



Logo/Scratch: programming for children

Logo [dA81, Pap80], Scratch [RMMH+09], and others

- Teaches programming using graphics/geometry
  Grabs attention.
- Usage paradigm 1: Narrative
  - Create graphical characters, animate/choreograph their movements and tell a story.
  - Avenue for expression of artistic talent.
  - Not full range of programming skills, somewhat like web page design.
- Usage paradigm 2: General
  - Draw precise, patterned, intricate geometric figures.
  - Solve standard programming problems.

Logo/Scratch are very successful

Created hope/confidence that children can program.

Conclusions: Graphics builds on what students know/like.

Logo has been developed by Papert and several others and scratch by Resnick and several others and scratch actually could be said to be a derivative of logo and scratch is not the only derivative, there are other derivatives also. What does logo do, so logo first of all teaches programming using graphics and geometry. Graphics and geometry grabs attention. Children and even older students love to see pictures and especially pictures that are being drawn or pictures that are moving or being manipulated.

There are 2 paradigms actually, there are 2 ways in which logo and scratch get used. The first paradigm might be called a narrative paradigm. What happens here? Well, in this students create graphical characters or they may use pre-created graphical characters and they animate or choreograph the movements of the characters and they use this to tell a story. So a program will essentially be a script.

So the program might say move this fairy from this tree to this house or something like that okay. So in each step of the program you are going to do something to the characters and something will happen on the screen. So you are programming these characters on the screen and they may do things on the screen, that they, for example they may draw lines on the screen as they move for example.

Scratch is especially beautifully crafted and in fact people who are artistic talents students who have artistic talent can create their own graphical characters. Of course this is an expression of graphical talent and many may say that look what is the programming in this and they might indeed be right, but the creators of scratch say that the point of scratch is to bring artistically talented people into programming somehow or the other, not fully but at least partially.

So that they start using programming, not sort of the kind of mathematical programming that we might want, but at least they start moving things on the screen and once they start moving things on the screen they may want to do other things, say for example they may want their characters to walk, so that means maybe they have to put some kind of a loop over there okay. So it may or may not encourage students to learn the full range of programming.

To give an analogy and only analogy it is somewhat like webpage design. So you are laying things out on the screen and if I click on this you want this to happen, you are sort of setting things up and maybe you are not writing programs in which you need to worry about whether there is termination or not, whether the program runs in linear time or not, those are things which may not get tested or you may not need to understand those things when you write narrative programs.

The second paradigm is the more general paradigm and even in this graphics plays a role, but in this case the graphics is meant to be much more precise. Your drawing patterns which might have intricate features, but there may be symmetry and so you might have to reason about that symmetry and you might have to program that symmetry and of course the languages large enough so that you can solve standard programming problems.

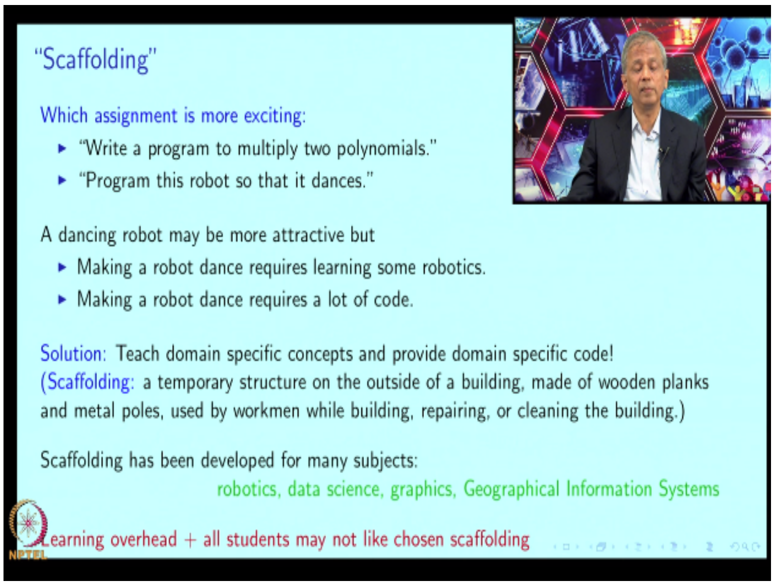
Actually the language is large enough in fact so that you can also do things like parallel programming to a little bit. Logo and scratch are very successful in that they encourage children to program, okay. So they have created the hope or the confidence that children can program; however, it must be said that their strength is in my opinion is slightly different. I do not believe that they have been successful in helping every child to program.

What they have been successful in is that they have provided an avenue for children who have inclination to program, to come out and start programming. It is not clear that they can inspire every child. They have tried to do so by bringing in this narrative model and some additional people have come in, but again will they be able to program in the sense that we want where they have to reason about termination and things like that?

It is not; however, the main takeaway in my opinion from scratch and logo is that they build on what students know and like. Students like graphics, students have learnt geometry, so students like using what they have learnt and that is what logo as well as scratch beautifully exploits and it also shows this general idea that geometry is actually extremely rich. When we look at pictures of course we can see iteration.

We can see repeated patterns, but not only that we have pictures in which there is recursion, there is recursive substructure and we can write recursive programs and in fact you can write recursive programs in logo which are very easy to understand. So I think those are the conclusions that we should take away and we should think of exploiting geometry as a means to teach programming. Another idea we have come out of logos what to say is the idea of scaffolding.

(Refer Slide Time: 30:40)



"Scaffolding"

Which assignment is more exciting:

- "Write a program to multiply two polynomials."
- "Program this robot so that it dances."

A dancing robot may be more attractive but

- Making a robot dance requires learning some robotics.
- Making a robot dance requires a lot of code.

Solution: Teach domain specific concepts and provide domain specific code!
(Scaffolding: a temporary structure on the outside of a building, made of wooden planks and metal poles, used by workmen while building, repairing, or cleaning the building.)

Scaffolding has been developed for many subjects:
robotics, data science, graphics, Geographical Information Systems

Learning overhead + all students may not like chosen scaffolding

So consider the following 2 assignments and tell me for yourself which is more exciting. Okay, the first assignment is write a program to multiply 2 polynomials and the second assignment is program this robot so that it dances. This is hardly a fair question right, 90% of the people will say of course I want to program the robot, I mean that is so much more excited, well it might be exciting, but for example it requires you to learn robotics.

And second making robot dance requires a lot of code, so what we do? Well, here is the idea. So in our introductory programing course itself we may think of teaching some domain specific concepts. So maybe we teach a little bit of robotics and then we provide the code which will do say the higher level function. So maybe we will provide the code which makes the robot walk and then will tell the student oh now you change the codes so that instead of walking, the robot dances.

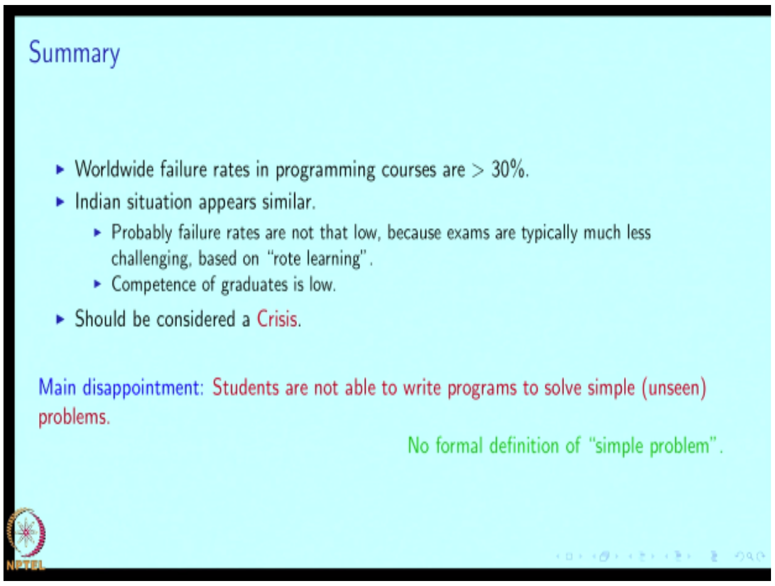
So this way we are providing additional material so that the students get drawn into this task of programming. So if you have seen buildings being built or repaired, a temporary structure is often erected on the outside, typically made of wooden planks or metal poles and things like that and it is used by workman while building or repairing or cleaning the building and that structure is called scaffolding.

So we are going to provide code which will be used by students not because they need to learn that code because that that code will make it easier or more attractive for the students to learn programming. So people have used this idea in many subjects and scaffolding has been developed for many, many domains, robotics, data science, graphics, geographical information system and even some others.

So this seems like a really nice idea, but we have to worry about 2 things. First of all if students are going to use robotics as scaffolding they need to learn robotics. So if we are saying that programming is already too difficult we have to ask the question, is it going to be more difficult because students need to learn robotics. For some students it certainly will be and some students might like robotics and some students might like data science, some students might like geographical information systems.

But if he teaches only one of them we might be leaving out some students. We might be trying to teach geographical information systems to students who wanted to learn robotics. So scaffolding might be a good idea but we have to be careful in using it okay. So now I am going to summarise the first part of this course and let us go over what we have learnt.

(Refer Slide Time: 33:55)



Summary

- Worldwide failure rates in programming courses are > 30%.
- Indian situation appears similar.
  - Probably failure rates are not that low, because exams are typically much less challenging, based on "rote learning".
  - Competence of graduates is low.
- Should be considered a Crisis.

Main disappointment: Students are not able to write programs to solve simple (unseen) problems.

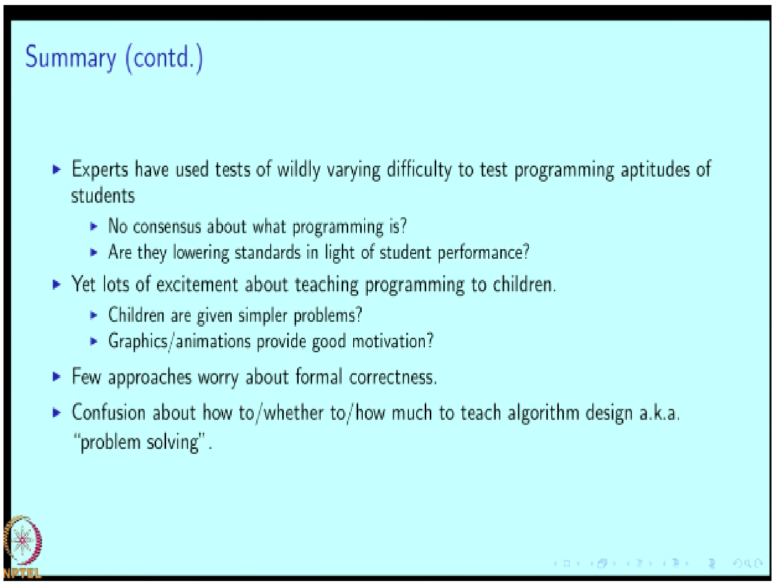
No formal definition of "simple problem".

So first of all it is important to remember that worldwide failure rates in programming courses are high. The Indian situation appear similar and quite possibly the failure rates may not be high in India, but that is only because our exams typically are much less demanding, but it is not at all the case that our students are being trained better than what is happening worldwide.

So we really should be considering what is going on to be a crisis. The main disappointment of teachers seems to be that students are not able to write programs to solve simple unseen problems. Our hope is that students will learn programming and then they will use it just as they use arithmetic or writing. It will be an extension of their thinking abilities, but that does not seem to be happening if we give them a new program, a new problem, students are finding it very difficult to write program to solve those problems.

And courses seem to be same that look they should be able to solve simple problems, but we are at a loss to provide a definition of what a simple problem is. So perhaps we should be doing that. We should be thinking carefully about what kinds of problems should we be asking okay. Experts have given tests of widely varying difficulty to test programming aptitude of students okay, so does it mean that there is no consensus about what programming is?

(Refer Slide Time: 35:34)



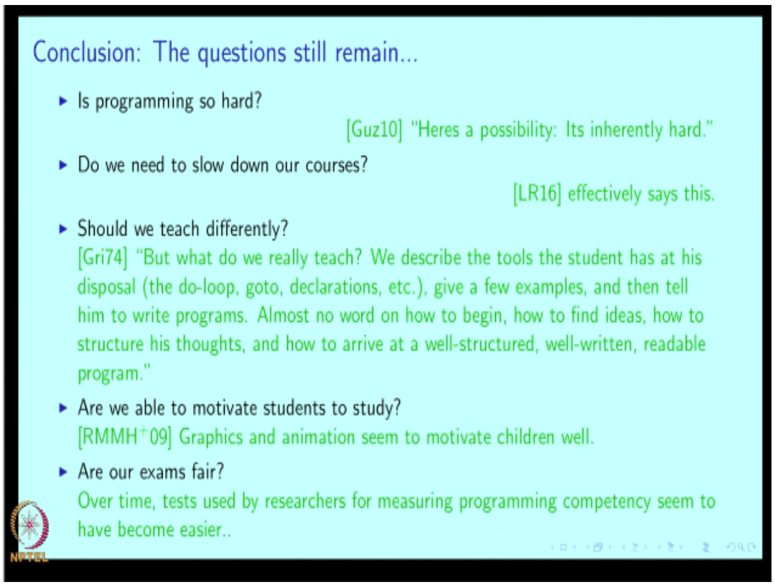
Summary (contd.)

- Experts have used tests of wildly varying difficulty to test programming aptitudes of students
  - No consensus about what programming is?
  - Are they lowering standards in light of student performance?
- Yet lots of excitement about teaching programming to children.
  - Children are given simpler problems?
  - Graphics/animations provide good motivation?
- Few approaches worry about formal correctness.
- Confusion about how to/whether to/how much to teach algorithm design a.k.a. "problem solving".

Or does it mean we are lowering, the experts are lowering standards in light of student performance. So this is on one hand as far as teaching programming to college graduates is concerned, but there is lots of excitement about teaching programming to children. So perhaps this is because children are given simpler problems, okay and perhaps graphics and animation provide good motivation.

But maybe we should be thinking from this experience as to what we can do when we teach to college students. We have seen that few approaches worry about formal correctness, but we think that they are important and there is confusion about how to or whether to or how much to teach algorithm design, also problem solving. We are relying on some kind of street smartness, but we really should be quantifying or specifying that a little bit more clearly, alright.

(Refer Slide Time: 36:40)



Conclusion: The questions still remain...

- Is programming so hard?
  [Guz10] "Heres a possibility: Its inherently hard."
- Do we need to slow down our courses?
  [LR16] effectively says this.
- Should we teach differently?
  [Gri74] "But what do we really teach? We describe the tools the student has at his disposal (the do-loop, goto, declarations, etc.), give a few examples, and then tell him to write programs. Almost no word on how to begin, how to find ideas, how to structure his thoughts, and how to arrive at a well-structured, well-written, readable program."
- Are we able to motivate students to study?
  [RMMH$^+$09] Graphics and animation seem to motivate children well.
- Are our exams fair?
  Over time, tests used by researchers for measuring programming competency seem to have become easier..

So to conclude let me just point out, let me just go back to a slide which I showed earlier and let me say that the questions raised in that slide still remain. So what are the questions? Is programming so hard, it was conjectured that it is hard and we have not really seen evidence to the contrary. Do we need to slow down our courses? Well we do not yet know, we need to think about it.

Should we teach differently? Well, it appears that we do not really teach how to design programs, how to deal with unseen problems, so something needs to happen on that ground. Are we able to motivate students to study, the piece of good news is that graphics and animation seem to motivate children well and maybe scaffolding can help okay and finally, are our exams fair?

So over time the test used by researchers for measuring programming competency seem to have become easier; however, this really means that we do not know what is the fare test. We seem to be wavering about that and my personal experience is that if you look at large number of examinations you will find suddenly there is a hard problem, suddenly teachers feel compelled to give hard problems.

So in my mind there is some question about what is a fair examination, a fair programming problem for first year students and what is not. So all these questions need to be studied, we need to think about them and that is what we are going to do in the following weeks. Thank you, we will stop over here.