Could computers understand their own programs? To answer this question I will draw upon two ideas originally put forward by the ancient Greeks, and on two ideas from Alan Turing, the great pioneer of Computer Science. I hope my answer will shed light on traditional philosophical questions about the nature of human understanding, and the nature of the introspective self-awareness that we believe is the exclusive gift of humanity. I also hope my answer will lead to results that are useful to the designers, programmers and users of the innumerable computer programs, which now monitor and control so many aspects of modern life.
The Distinguished Speakers Program is made possible by

Association for Computing Machinery

Advancing Computing as a Science & Profession

For additional information, please visit http://dsp.acm.org/
Summary: YES

• Their understanding could be tested
  – in the same way as that of students
  – by an interactive examination
  – in which they justify their answers.

• The computer passes the test
  – if their answers to questions are useful
to professional programmers.

In summary, my answer to the question is Yes. In principle, computers could understand their own programs, and those of other computers, although their understanding is rather weak at present. It was Alan Turing who suggested that we can gather evidence of the understanding of computers if we subject them to the same kind of examination that we have inflicted on many generations students at schools and universities. An examination usually requires the student to justify each given answer by logical reasoning. The most rigorous principles of logical reasoning was first formulated by the ancient Greek Philosopher Aristotle; and they were further refined by the ancient Egyptian philosopher Euclid. Finally, Alan Turing also showed that these principles apply equally to logical reasoning about computer programs. Finally, I will suggest how to set the pass mark in the examination. A computer passes the test of understanding if the answers to the questions about its own program are useful to the human software engineer who wrote the program. I hope that one day, computers will pass this test. Then they will be able to serve as intelligent assistants to the programmer during the task of writing the programs.
Here are the sources of my inspiration. I have obtained much detailed knowledge, as well as insight and delight, from the study of these sources. They are very readable.

Acknowledgements

classics.mit.edu/Aristotle/prior.html

• [2] Euclid, Elements (~300bce)
aleph0.clarku.edu/~djoyce/java/elements.html

• [3] A.M. Turing, Checking a large routine,
Cambridge Univ. Math. Lab 67-69 (1949)

The ancient Greek philosopher Aristotle made a remarkable contribution to the history of human thought. He was the founder, director and lecturer at a private academic institution, to which he gave the name of Academy. This name is now applied much more widely, to institutions of higher learning throughout the world. His lecture notes still survive. They deal with both the sciences and the humanities, and defined almost the full range of human intellectual endeavour for next two thousand years. I will concentrate on his introduction to logic, of which he has a good claim to the title of founding father.

Ancient Athens was a democracy, in which the whole population of citizens (except women and slaves) acted as a legislative parliament. There was no party system. If any politician wanted to pass legislation, he would have to persuade more than half the citizens in assembly to vote for the law. Naturally, good powers of persuasion were essential; and what could be more persuasive than a logical argument? Ability to refute the supposedly logical arguments of your opponents were equally important. It is no surprise that the Academy, and its rivals and competitors, taught logic as a central skill, of immediate and profitable application.

The particular form of Aristotle’s logic seems well adapted to applications in biology, of which he is also recognised as the founding father.
Like all branches of science, the study of logic starts by listing and defining a notation for the range of concepts that constitute its subject matter; it then lists the range of sentences (or judgements) which may appear as a line of a proof. These definitions are given by formalising the grammar (or syntax) of a greatly simplified language. Aristotle used the familiar grammatical structure of sentences in a natural language to define the syntax of his logical sentences. Finally, he formalised the grammatical shape of each type of deductive step which may validly appear in a chain of reasoning. Any argument which follows the rules of this grammar is valid, and all other arguments are invalid.
To define his grammar, Aristotle introduced a form of syntactic definition that was rediscovered by Chomski in modern times, and developed by computer scientists for the syntactic definition of programming languages. Aristotle introduced letters like $S$ and $P$ as non-terminal symbols, to stand for the subject and predicate of a sentence. These letters are intended to be replaced by universal names of sets of individuals (e.g., sharks, fishes, animals, men). Aristotle’s logic then permitted four forms of sentence, which combine the subject and predicate using the words All, Some, None, and Not. The vowels (a), (e), (i) and (o) were introduced in later times as mnemonics for these four kinds of sentence.
Aristotle then went on to define a set of twenty four patterns of valid deduction, called syllogisms. Each syllogism consists of three sentences, written on successive lines. The first two sentences are called premises (major and minor). They are written above the line; and the third sentence below the line is the conclusion of the syllogism. The purpose of the syllogism is to check and explain the validity of a short argument, which starts with an agreement on the premises in the first two lines, and concludes that the third line is necessarily true as well. The letters S and P stand for the subject and predicate of the conclusion. The letter M stands for a middle term that is present in both the premises, but is eliminated in the conclusion.

This slide shows the first example of a syllogism, which was later given the name Barbara. The three vowels aaa in the name Barbara indicate the syntactic form of the two premises and the conclusion of this kind of syllogism.

10 mins
My second example of a syllogism is Celarent. The three lines of the syllogism are of the form e,a,e, as indicated in the vowels in the name of the rule. Again, the conclusion follows fairly obviously from the two premises. The purpose of a formal logic is to tenable the validity of the step to be checked, for example by a computer, without having to reconsider intuitively on every occasion why it is valid.
My last example is Darii. It is, perhaps, a little less obvious than the previous two examples. You can justify it by observing from the second premise that at least the $S$'s that are in $M$ must be also in $P$, because the first premise says that everything in $M$ must be in $P$. 
24 syllogisms

- Barbara, Barbari, Barnalip, Baroco, Bocardo, Camestres, Camestros, Calemes, Calemos, Celarent, Celaront, Cesare, Cesaro, Darapti, Darii, Datisi, Dimatis, Disamis, Felapton, Ferio, Ferison, Fesapo, Festino, Fresison.
- Modern logic has just one rule of inference, and a couple of axioms,
- and is much more powerful.

Here is a list of mnemonics for all the 24 syllogisms of Aristotle. They are in fact a complete list of all the valid three-line arguments that can be written in Aristotle’s syntax. In earlier times, a student of logic would know all these names, and be able to recognise and use them in disputations with other students and with his examiners.

Fortunately, modern logics have been greatly simplified since Aristotle’s day, and at the same time they are much more powerful, in the sense that they cover a wider range of valid reasoning. For example, Boolean logic, which is essential to the design of any binary digital computer, can be based on just a single deductive rule, and a few simple and obvious axioms.
The example deductions shown above are taken from Aristotle’s classificatory biology. His study of biology dates from a visit to the Greek island of Lesbos, where he got the fishermen on return from a fishing trip in the morning, to show and name for him all of the various kinds of catch in their nets. He realised that sharks and rays were similar to each other, but different from the majority of inhabitants of the sea. So he classified these two species under a distinct genus of selachians.
The intention of Aristotelian logic is to validate arguments a great deal longer than three lines. His principle is that the conclusions of the earlier syllogisms in the argument are allowed to be used as premises of later syllogisms. Here is an example of a five-line proof, containing one instance of Barbara and one instance of Darii. The middle line of the five is the conclusion of the first syllogism and the major premise of the second.
A proof is defined as a series of lines in which every line is either itself a premise, or it is the conclusion of syllogism, from two of the premises that precede it in the proof. This recursive definition of the syntax of proofs allows proofs of any finite length. The purpose of the proof is to give permission in any proof to deduce the conclusion in the last line of the proof from prior agreement to the truth of all the unproven premises contained in earlier lines.
Aristotle realised that his logic was an entirely general-purpose or universal tool, just like stored-program general-purpose computers of the present day. This is because the validity of a proof depends entirely on its syntactic form, and in no way depends on its subject matter. This principle of logic violates the deeply felt prejudices of the man in the street, who naturally believes that validity of an argument depends on whether you actually like the conclusion or not, or whether you like the person who is presenting the proof, or even whether the premises in the proof are true or not. It is the independence from these real-world concerns that makes it possible for a computer to check the validity of logical and mathematical proofs.

15 mins
Computer reasoning

• Computers easily check conformity of a proof to the given deductive rules.
• Computers discover proofs by exploring all the possible deductions from lines proved so far.
• Computers were essential to proof of Four-colour Theorem and the Kepler Conjecture.

But computers can help far more effectively than just by checking. Computers can examine far more cases than a human would have the patience or accuracy to enumerate. Computers have recently helped in generating and checking long proofs of interesting theorems in mathematics, and they have established conjectures that were made centuries ago, and which mere humans have long been unable to prove without computer assistance. Recent notable examples are the famous four-colour theorem, and the Kepler conjecture.
The four-colour theorem tells us that in a map of the constituent states of a country, every state can be painted in one of four different colours in such a way that no neighbouring states that share a border need to share a colour.

The picture on the left of this slide shows that four colours are indeed necessary for colouring a map, to ensure that the green state in the middle does not have a green neighbour. Furthermore, the three states that surround it all touch each other; so they need three different colours, all different from green. Green can be reused anywhere outside this picture. The picture on the right has one of its states split into two. So it has more states, but can be successfully coloured with only three colours.
A recent computer proof that no more than four colours are ever needed, was constructed by my colleague Georges Gonthier in the Coq programming language and proof validating system. It required examination of 633 cases, similar to the two cases shown above, but usually more complicated. Each case required typically over a million proof steps – approximately a billion steps in all. This is the shortest known completely formal proof. It was a computer program that generated all these steps, but Georges wrote the program. He had to invent a new branch of mathematics to do it.
The Kepler Conjecture

This is the way to pack the most oranges into a large container.

The Kepler conjecture tells us the best way to pack the most oranges into a large but limited space. It is the natural way that coster-mongers and green-grocers already do it. A computer-assisted proof of this was discovered in 1998 by Tom Hales. A fully automatic proof of the Kepler conjecture is still being investigated. It is estimated that it will take a further 66 years of human effort to get the computer how to check the proof completely.
The first answer to question 2 was given by the ancient Egyptian geometer Euclid. He lived and worked in Alexandria, around the beginning of the third century BCE. He systematically applied logical deduction to the proof of theorems about physical space, with applications to measurement of land (the original Greek meaning of Geometry). Ever since his day, geometry has been applied to the definition of boundaries, to the design of buildings, to the surveying of land and of seas, and to the making of maps and charts. And his textbook on the subject has been the basis of mathematical education in Europe and the middle east for upward of two thousand years.

20 mins
The logic of Euclid was very different from that of Aristotle. It did not reason just about facts. Rather, it reasoned about actions, and showed how to prove that they achieve a desired effect. The major part of a typical Euclidian proof is a construction, a program of actions which will draw a line or triangle or other figure which possesses some desired property; and the rest of the proof just proves that it does so. Euclid therefore gives a comprehensive set of rules for reasoning about the programs. He demonstrated their use in the valid proof of hundreds of geometric theorems.

So I would call Euclid the inventor of the world’s first programming language. It is similar to a modern graphics programming language, used to draw geometric pictures on the screens of all the computers, televisions, cinemas, and touch-screen mobile phones of the present day. It contains many of the essential features of the languages taught today in an introductory course for Computer Scientists. And of course, Euclid’s programs had to be carried out by people, using only instruments available to land surveyors of Euclid’s own day, perhaps pegs and strings for marking boundaries in the land, when the annual flood of the Nile has receded. So perhaps it is not surprising that the formalisation of geometry was the achievement of an Egyptian geometer. As for logic in Athens, the practical value of geometry in Egypt was never in dispute.
Euclid defines his programming language in five basic axioms or postulates. They describe all the basic actions which are provided by his programming language that he used in his proofs. For example, the first postulate allows the program to draw a finite straight line between any two points, given as parameters to the action. The third postulate is an action that draws a circle with its centre at any given point, and with a second given point on its circumference. The famous fifth postulate allows the drawing of a line parallel to a given line, and passing through a given point. These basic actions are included in all graphics programming languages today.
This animation illustrates performance of the action allowed by postulate 3.
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>A point is that which has no part</td>
</tr>
<tr>
<td>2.</td>
<td></td>
</tr>
<tr>
<td>15.</td>
<td>A circle is ...</td>
</tr>
<tr>
<td>16.</td>
<td>Its centre is ...</td>
</tr>
<tr>
<td>20.</td>
<td>An equilateral triangle has three equal sides.</td>
</tr>
</tbody>
</table>

Words of the language are related to each other and to their meaning in the real world.

The postulates are accompanied by twenty three definitions. Some of them (like the first one, defining a point) relate the words of the language informally to their meaning in the real world. Other definitions (like number 20) define new words in terms of previously defined words. Definitions 15 and 16 state that a circle is a figure in which every point on the circumference is the same distance from the centre. As a result, every radius of the same circle is equal in length to every other radius.
Here is a picture which illustrates Euclid’s definition of a equilateral triangle. It introduces the convention that equal lines in a diagram can be decorated by the same marking, for example, a yellow twiddle.
Here I have used the same marking convention to summarise Euclid’s definition of a circle. He defines it as a figure which has all points on its boundary at an equal distance from the same point, namely its centre.
Euclid realised that his geometry was only a particular application of more general principles of logical reasoning, which he called common notions. They relate to the mathematical and logical concept of equality, and they are built in to all modern systems of mathematical logic.

Five common notions

1. Two things that are both equal to a third thing are equal to each other.
2. If equals are added to equals, the wholes are equal
3. If equals are subtracted from equals...
4. Things which coincide are equal
5. The whole is greater than the part
The main part of Book One of Euclid’s elements is a collection of forty eight theorems, usually called propositions. The last two propositions express the well-known theorem of Pythagoras about a relationship between the lengths of the sides of right-angled triangles. The first proposition constructs an equilateral triangle, and proves that it satisfies its definition (number 20).

25 mins.
The purpose of a proposition is like that of a subroutine or function or method of a modern programming language. Just like a subroutine in a program, it can be called by its name or by its number, as a single command in any subsequent proof. The call will perform the construction described in the proof of the called proposition. The proven property of the construction can be used as assumptions of the proof of any proposition that calls it.

Subroutines

Propositions are subroutines that can be called by name repeatedly in later proofs, to perform useful constructions. The proven properties of the construction can be used as assumptions of the proof of any proposition that calls it.
To illustrate Euclid's programming and reasoning methods, I will show Euclid's beautiful proof of his very first proposition, number 1. It constructs an equilateral triangle with a given finite straight line (shown here) as one of its sides.
The first action in the proof is to choose one end of the given line.

Next, draw a circle with its centre at the chosen point, and with the other end of the line on its circumference.
This is done by a call of postulate 3, which allows one to draw a circle with any given centre and radius.

The next step is to draw a second circle with the same given line as radius, but now the centre of the circle is at its other end.
Of course, after this step, it is obvious that it did not matter which of the two circles was drawn first. The circumferences of the two circles now meet at two points, one on either side of the original straight line. Although it is obvious from the diagram, we now realise that Euclid should have proved this.
Now it is necessary to choose one of the two intersection points, whichever is more convenient. In modern geometric texts, it is usual to name the chosen point by some letter, say $C$. This naming of a particular point is like a declaration (or allocation) and initialisation of a variable in a modern programming language. However, in Euclid’s programs each such variable is assigned its value only once. Euclid’s language is essentially a single-assignment language, like the functional languages of the present day.
It is also necessary again to choose one of the ends of the given line.

The next step draws a line from the point named \( C \) to the chosen end of the original given line.
Then draw a line between them (Postulate 1).

This is justified by postulate 1.

Then do the same for the point at the other end of the given line.
That is the end of the construction. The triangle shown in this final figure is the desired equilateral triangle. It remains only to prove this fact.
The proof goes as follows. The two lines marked by blue equals signs in this diagram are equal, because they are both radii of the left hand circle, and definitions 15 and 16 tell us that all radii of the same circle are the same length.

30 mins
The lines marked ✓ are equal, being radii of the right circle (Def. 15)

And the same is true of the two lines marked with a red wedge. They are both radii of the right hand circle.
We now use the principle of equality (which Euclid has stated as a common notion). The first constructed line is equal to the original given line. The second constructed line is also equal to the given line. Since they are both equal to the same thing, they are equal to each other, as shown by the yellow twiddles.
According to Definition 20, any triangle with all three sides equal is an equilateral triangle. We have therefore proved Euclid’s proposition number 1.
The caller of this proposition does not need to reproduce or even know the construction.

When proposition 1 is called as a subroutine from a subsequent proof, it is not necessary to repeat the construction. It is permitted just to mark that the sides of the triangle as equal, and to quote proposition 1 as a justification.
On this slide I have listed the main features of Euclid’s language, which are pervasive in modern programming. The individual primitive commands of the language are described in the postulates. New names can be given to points and other figures in the diagram. A larger diagram is drawn sequentially, by first drawing one part of it, and then drawing the rest. A construction is called a proposition, and it can be performed whenever needed, just like a subroutine in programming. The input data for the proposition describes the initial state with which the construction starts. It is often accompanied by a precondition describing the circumstances in which the construction will be successful. The purpose of the construction is described by a postcondition, stating some useful property of the completed figure.
I now come to the third of the brilliant philosophers and logicians who have contributed to the answer of the question posed in the title of my talk. The Centenary of his birth has only recently been celebrated. Like Aristotle, he was a polymath. He made contributions to number theory, developmental biology, statistics, philosophy, cryptanalysis, and logic, as well as earning a title as founding father of Computer Science.
This slide just summarises the reasons why I think that Alan Turing would agree with my positive answer that computers can indeed understand their own programs. He invented a logic for reasoning about programs, following the example of Euclid’s logic for reasoning about constructions. He realised that a computer can use logic for reasoning, which would include reasoning about its own programs. And he suggested that this could be taken as the criterion for the understanding of programs by computer.

This ideal that computer programs should be accompanied by a proof of their correctness is one that has inspired my own life-long research directions. In the first article that I published as an academic, I introduced a structured notation for writing proofs of programs, which was subsequently known as the Hoare Triple. I will use this notation to explain Turing’s method of verification as a development of the tradition of formal logic as instituted by Aristotle.
The Hoare triple: \( P \{ V \} R \)

means: When \( P \), action \( V \) ensures \( R \)
where \( V \) changes (part of) the world
- \( P \) describes an (initial) state of that part
- \( R \) describes the (final) state resulting from action \( V \)

Turing’s logic is a logic which reasons about actions which change the state of the world. A computer program is a special case of such an action, which changes the state of just a small part of the world that is inside the memory of a computer executing the program. In the case of a Euclidian proof, a construction is an action which changes the state of the diagram that illustrates the proof.

As in the case of Aristotle’s syllogism, let us introduce letters to stand for what we are reasoning about. We are reasoning about an action, denoted by a verb \( V \). The initial state of the world before the action is described by an assertion \( P \). It is the precondition of successful performance of the action. The final state after the action is described by an assertion \( R \), called the postcondition of the rule.

The meaning of the Hoare triple is that when action \( V \) is performed in a state satisfying the assertion \( P \) then when the action is completed, the assertion \( R \) will be true.

35 mins
Examples of triples

• When the radio whistles, turning the suppression knob clockwise ensures the absence of the whistle.

• When two distinct points are closer than \( r \), drawing a circle of radius \( r \) around each point, ensures that their circumferences intersect at two places.

• When the value of \( x \) is greater than three increasing the value of \( x \) ensures that \( x \) is greater than 4.

The first of these examples is taken from a human action changing the real world. The second describes the precondition under which circles around two points will have a point at which their boundaries intersect. This is a geometric fact that was not quoted explicitly by Euclid’s proof of proposition 1. The third example describes the action of a computer program which increases the value of one of the variables named \( x \), which stored in the memory of the same computer.
Here is the rule which enables us to prove things about performing actions sequentially, one after another. We have seen plenty of examples of sequential execution in the construction that implements Euclid’s proposition 1. This slide shows a rule of the logic of programming, just like the syllogism of Aristotle. As always, the purpose of the syllogism is to prove the conclusion on the third line from the premises on the first two lines.

The action (V1 then V2) means performance of action V1, followed in time by performance of action V2. Many programming languages use semicolon to abbreviate the word ‘then’. The first premise introduces an intermediate state between the two actions, and describes the state by the assertion S. It says that S is ensured by V1 under the overall precondition P. The second premise uses S as a precondition, to ensure that V2 achieves the overall goal R.

In a multistep construction of a Euclidian proof, every step changes the state of the same diagram. Consequently, every step of the proof implicitly uses this rule.
Conditional tests.

\[\begin{align*}
\text{When } P \text{ and } B \text{, } V_1 \text{ ensures } R \\
\text{When } P \text{ and not } B \text{, } V_2 \text{ ensures } R \\
\text{When } P \text{, } (\text{if } B \text{ then } V_1 \text{ else } V_2) \text{ ensures } R
\end{align*}\]

This is the rule which introduces alternative actions into the logic. Suppose \( B \) is a description of some test which the program (or other agent) can apply to the current state. The first premise of this syllogism says that if \( B \) is true in the same initial state that satisfies \( P \), then the action \( V_1 \) will ensure the overall goal \( R \). The second premise is similar, but it deals with the case that the condition \( B \) is found to be false in the initial state. The bracketed action on the third line is an action of the programming language which selects between \( V_1 \) and \( V_2 \) according to the truth or falsity of \( B \). It is usually called a conditional. The conclusion states that the runtime test of \( B \), followed by the selected action (\( V_1 \) or \( V_2 \)), will ensure the overall goal \( R \), independent of whether on any given occasion the test is true or false.

Euclid did not have such a rule in his logic.
Repetition

- When $P$ and $\neg B$, $V$ ensures $P$
- When $P$, (repeat $V$ until $B$) ensures $(P$ and $B)$

This slide fills the remaining gap in Euclid’s logic of proof. It introduces a proof rule which needs only a single premise. The premise has the same assertion $P$ appearing both as a precondition and as a postcondition. Such an assertion is called an invariant under the action $V$ of the triple. The premise also says that $B$ must be false before the action.

The conclusion of this syllogism states that if $P$ is true initially, it will remain true as a postcondition, however many times $V$ is repeated (even no times at all). Furthermore, the tested condition will also be true, because this is the only condition under which the repetition terminates.

40 mins
Nearly all large programs which are in widespread use today, are inherently non-deterministic. The programs do not specify exactly the order in which the various actions take place. In fact this depends on essentially uncontrollable factors, like the voltage and the temperature of the hardware, or the time of arrival of a message from outside the computer. The problem with non-determinism is that the decisions may go differently each time the program is tested or run. We therefore need a proof rule to ensure that the program is correct, no matter which choice is taken.

Euclid’s programming language also contains non-determinism. In the proof of proposition 1, Euclid’s construction calls for selection of a point of intersection of the boundaries of two circles. But there are in fact two points of intersection, and Euclid’s construction does not determine which one should be chosen. Each choice leads to a different result. This means that the person carrying out the construction must exercise the choice, rather than the person who designed the construction, Euclid himself. For example a surveyor might find that one of the two points is in the middle of a river, so he would definitely choose the other point.
Syllogism for non-determinism.

- When $P, V1$ ensures $R$
- When $P, V2$ ensures $R$
- When $P, (V1 \text{ orelse } V2)$ ensures $R$

Here is a suggested proof rule for arbitrary non-deterministic choice. In the syllogism on this slide, the desired conclusion predicts the behaviour of the action which either performs $V1$ or else $V2$. The two premises state that you have to prove the same behaviour separately, both for $V1$ and for $V2$. In many cases, the two proofs share nearly all their steps, as in the example of Euclid’s proposition one.
The final link in my answer to the question of computer understanding is also due to Turing. It is described in a paper, which he published in the British Philosophical Journal Mind. In it he introduced a famous test, as a conversation, a sort of cultured small-talk, between a tester and a machine. In his example, the conversation ranged over questions of poetry and literature. The tester is just an ordinary member of the general public, not a specialist in computing. On half of the tests, the human tester is conversing with a machine, and in the other half with another human being. The tester does not know which. And there is only just five minutes in each test for the tester to make a decision whether the partner in the conversation is a machine or a human. The machine passes the test if it fools the tester does no better than chance in making the correct guess.

The Turing test

- by conversation between tester and machine
- on arbitrary topics of human interest
- for five minutes.

- The machine fools the tester that it is human
- on 30% of the tests
I beg your indulgence to suggest a modified version of the test, which is more relevant to engineering than to philosophy.

I suggest that the understanding of the computer can be tested by examination, like those to which we subject our students. It is interactive, like an oral examination, and allows the computer to ask supplementary questions for clarification. The questions are limited to just a single scientific topic, namely the computer’s own program. And there is no arbitrary time limit: the examiner decides when to stop.

45 mins
The machine answers most of the examiner’s questions, perhaps after asking questions of clarification. Of course, as Turing showed in his first publication, any computer may sometimes fail to answer a question about the termination of a program. Fortunately, as the occasional failure (or even mistake) does not invalidate a student’s or a computer’s claim to understanding.

I suggest that a definitive criterion of a computer’s understanding is the same that convinces an examiner of a human student’s understanding: the examinee can explain its own reasoning, which led it to the answers. And in the main part of my talk, I have shown how a computer or a human can explain reasons for their answer.

The pass grade for the computer examinee is very high. Nearly all the answers given must be correct, and most of them should be relevant and useful. Useful to the examiner, who will usually be the software engineer whose job it is to develop the program itself.
Typical questions

• Can your program overflow a buffer?
• If so, give an execution that reveals the error.
• Generate test cases that exercise all the changes recently made to the program.
• Can this change make the program slower?
• Are all assertions made in the program valid?
• Could the airplanes under control by the program ever collide?

Here are some questions the software engineer might wish to ask of a computer.
Turing expected that a computer would successfully pass his test within fifty years. Unfortunately, the fifty years have now elapsed, and we have been disappointed. His test is one of those scientific experiments that would cost too much to carry out successfully. Maybe my experiment would also be too expensive.

Nevertheless, following Turing’s footsteps, I will make an equally optimistic prediction. I predict the fulfilment of an ambitious research goal of J Moore, which he has made the goal of his life’s research. Within fifty years from now, a design automation system for software engineering will be widely used by professional programmers in all their professional activities. It will play the role of an intelligent programmer’s assistant.

The attempt to develop a programmer’s assistant will not just be a scientific or even a philosophical experiment. The necessary research is being driven by commercial pressure to reduce the cost of development and increase the quality and reliability of computer programs.

The Intelligent Programmer’s Assistant

- Within 50 years, a design automation system for software engineering
- will be widely used by programmers
- in the analysis, design, programming, testing, delivery, and subsequent improvements
- of the ubiquitous software of tomorrow.

J Moore
Collaborative program development

- **Human understands**
  - the real world
  - the needs of program users
  - the commercial opportunities

- **Computer understands**
  - the consequences of human decisions in the context of a large and complex program.

So I predict that programs of the future will be written by a collaboration between a human programmer and a computer. Each party will contribute its own particular skills. The human has a common-sense understanding of the real world, an intuitive understanding of the needs of those who will use the program, and a good guess about the possible commercial value of the software product. The computer has an accurate understanding of the wealth of important detail that is needed in a sophisticated modern computer program. What is more, we hope that it will be able to predict accurately the consequences of changing it after delivery.

50 mins
The programmer will complain

- ‘The computer doesn’t understand what I want my program to do.
- It only understands the easy part of my job,
- and sometimes not even that’

This collaboration will lead to substantial reduction in the stress of program development, and an improvement in productivity of programmers. A real sign of achievement is that the programmer will still complain that the computer does not understand what I want my program to do. In fact the computer only understands the easy part of the job, and sometimes not even that.

Did you notice, the programmer will use the word ‘understand’ to describe the behaviour of the computer.

I hope my talk has convinced you that this usage of the word is a natural and appropriate analogy with the understanding of a program by a human being. The analogy is the same that justifies the idiom that airplanes fly, on an analogy with the flying of birds, even though it does not flap its wings.
Analogies

• Computers understand
  – by analogy with human understanding
• Airplanes fly
  – by analogy with the flight of birds
• Why don’t submarines swim
  – by analogy with fishes?

But remember, it is only an idiom, and only an analogy, and only in some languages. For example in English we do not say that a submarine swims, although its motion is very similar to that of fishes. But swimming is the word used in Polish for the motion of submarines.

My personal view is that Turing’s claim that machines can think is seriously compromised by the failure of his test. Of course, when we see the hourglass or other symbol on our computer screen, we may say that the computer is thinking. But there is no reasonable test which will distinguish the total unresponsiveness of the computer in this state from what in a human being we might call sleeping, or possibly dreaming! It is certainly possible to sleep during a conversation, or even an examination, but you will not pass by doing so.
I end with a quotation from Turing’s article on Checking a Large Routine. It summarises exactly the message of my talk. He says that a programming language forms a sort of symbolic logic. And I expect that one day a programming language will eventually be recognised, not just as a sort of logic, but as an integral part of the main stream of applied logic itself. When that happens, Computer Science will take its proper place in the History of Thought, as the latest development of one of the ancient and great intellectual traditions of the human race.
Speculation

• The logic of programming is the logic of action in space and time.

I will go beyond Turing in my speculations. I hope that programming language will not be just a sort of symbolic logic. It will be recognised as integral part of the main stream of applied logic itself. Quite independent of computers, the logic of programming applies directly to the world we live in, as the logic that governs actions in space and in time. When this is widely recognised, Computer Science will take its proper place in the History of Thought, as the latest development of one of the ancient and great intellectual traditions of the human race.

55 mins