Chapter 8
Limits of Technology

The Analytical Engine has no pretensions to originate anything.
It can do whatever we know how to order it to perform.

Ada Lovelace

That materialistic and mechanistic science is driving humanity into an evolutionary cul-de-sac is nowhere clearer than in the business world. There is a widespread belief in society today that technological development can drive economic growth indefinitely, that technology is the solution to all human woes. But is it? Since soon after the invention of the stored-program computer in the middle of the last century, computer scientists have claimed that they could create artificial intelligence, artificial consciousness, and even artificial life in their machines. We can see from Chapter 4, 'Transcending the Categories' that we human beings are not machines and nothing but machines, for we are Divine Cosmic beings.

It is thus abundantly clear that the invention of the stored-program computer requires us to make fundamental changes to the work ethic and the way we run our businesses. For those still not convinced of the fact that both capitalism and communism are incompatible with the invention of the stored-program computer, this chapter shows beyond any doubt that technology is limited, that human beings are the leading edge of evolution, not computers. We do so by asking the question, "Could computers program themselves without human, that is, divine intervention?"

Part I, 'Integral Relational Logic' showed how we can model the task of an information systems architect in developing a comprehensive model of the processes and entities of a business enterprise and hence of the Universe. In this chapter, we use IRL to model the job of a computer programmer. This subject is rather technical; it requires a good knowledge of computers and computer programming languages to fully understand. But this is not knowledge that computer scientists normally highlight. For if they did, it would be only too obvious that it

is not possible to program a computer to perform all the cognitive activities performed by human beings in the workplace.

Can machines think?

The key to this issue is the possibility that machines might be able to think, which is a fairly recent subject in Western philosophy. As Vernon Pratt has pointed out, it arose from the introduction of the Cartesian-Newtonian mechanistic paradigm in the seventeenth century.[1] Since the invention of the programmable computer in the middle of the last century, this question has come under the auspices of cognitive science, defined by Howard Gardner as psychology, philosophy, artificial intelligence, linguistics, anthropology, and neuroscience.[2]

Alan Turing, often considered the father of modern computer science, made a major contribution to this debate in his much-quoted article published in the philosophical journal *Mind* in 1950,[3] which began with the words, "I propose to consider the question 'Can machines think?'"[4] Turing himself was of the opinion that the answer to his question is yes, for he went on to conclude in his article that "I believe that at the end of the century the use of words and general educated opinion will have altered so much that one will be able to speak of machines thinking without expecting to be contradicted".[5]

Since then, despite all the efforts of computer scientists to create artificial intelligence,[6] consciousness,[7] and even life,[8] Turing's prediction does not seem to have come to pass. I know of no machine in the world that has passed the Turing test, that claims that it is at least as intelligent as human beings. Why is this? Well, we can get an inkling for this 'failure' from the insightful memoir on Charles Babbage's Analytical Engine,[9] written in 1843 by Ada Lovelace,[10] the poet Byron's daughter, which Turing quoted in his article.[11] Ada wrote:

> The Analytical Engine has no pretensions to originate anything. It can do whatever we know how to order it to perform. It can follow analysis; but it has no power of anticipating any analytical relations or truths. Its province is to assist us in making available what we are already acquainted with.[12]

So what is the truth? Even today opinions vary.

The many computer scientists working in the field of artificial intelligence obviously favour Turing's opinion. This can be seen most clearly from the Dartmouth Conference in 1956, when the foundations of AI were laid down by a number of leading computer scientists, among them Marvin Minsky and John McCarthy. For the latter stated the fundamental hypothesis of AI as follows: "Every aspect of learning or any other feature of intelligence can in principle be so precisely described that a machine can be made to simulate it".[13]

On the other hand, the management scientists in the American business schools who classified all activities in business into structured, semi-structured, and unstructured tasks,[14] seem to favour Ada Lovelace's view. For if it were possible to discover the deep structures that underlie semi-structured and unstructured tasks in a formal manner, as the AI scientists claim,

all tasks would be structured and automatable, and there would be no need to make a distinction between the different types of task.

So if the aims of the AI scientists are achievable, the fundamental principle of all monetary economies would break down. While human beings might wish to consume the products of a materialist society, it would be machines that would be the principal creators of goods and services. Thus the loop between human beings as workers and consumers in the economy would be broken. As virtually no economist or politician is looking at the possibility of such a situation, we must assume that they do not believe that it will happen.

The reason why scientists and philosophers cannot agree on this vitally important issue is that science, as it is practised today, is not capable of resolving the issue. The arguments tend to oscillate between the Turing test, described in his 1950 *Mind* article, and the consequences of Gödel's Incompleteness Theorems,[15] identified by J. R. Lucas in his article 'Minds, Machines and Gödel' first published in the journal *Philosophy* in 1961.[16]

He argued that Gödel's metamathematical reasoning could not possibly be done by a machine, because human consciousness is necessary to see the truth of Gödel's statement, "This theorem is unprovable". Another philosopher, John R. Searle,[17] has used his famous Chinese room thought experiment to refute the possibility of what he calls 'Strong AI'. He argues that it is quite possible for an English-speaking human being, who has no knowledge of Chinese, to mechanistically process a string of Chinese characters fed to him, without any understanding of what he is doing. Nevertheless, the answer produced at the end of the process is the correct one from the perspective of a Chinese who understands the symbols. Both Lucas and Searle have been much attacked for their endeavours by Hofstadter and Dennett.[18]

Roger Penrose, Rouse Ball Professor of Mathematics at Oxford University, has taken up J. R. Lucas's arguments in his best-selling book, *The Emperors New Mind: Concerning Computers, Minds, and the Laws of Physics*. A key point he makes is that mechanical processing is essentially algorithmic. So, are all human cognitive activities algorithmic—insight, for example? The answer he comes to is a tentative no; tentative because Penrose is aware that his understanding of the relationship of the mind and consciousness is still limited. As he says in the final paragraph of his first book, "For the answers to such questions [about the nature of human existence] to be resolvable in principle, a theory of consciousness would be needed. But how could one even begin to explain the substance of such problems to an entity that was not itself conscious ... ?"[19]

In a review of his book in *Time* magazine in June 1990, Penrose encapsulated his view with the statement: "Computers will never think because the laws of nature do not allow it." Marvin Minsky is quoted as saying in response, "As far as I can tell, he is just plain wrong."

In fact, this is an argument that nobody can win. It is not so much a win-lose game as a lose-lose one. For if the AI scientists are right our economic system will collapse and all will

be losers. And if they are wrong, why spend so much effort in doing something that is impossible? Isn't genuine human intelligence much more energizing and life-enhancing than its false pretender?

So perhaps I can explain a little more how we can answer the question "Can machines think?" Like Alan Turing's Turing Test and John Searle's Chinese room, my approach is based on the experiment in learning that I described in Part I. At the core of this experiment is one simple rule of concept formation: to notice carefully the similarities and differences of the data patterns of our experiences. In this chapter, we apply this rule to examine carefully the job of a computer programmer and then look to see what the analogous activities are in the human psyche.

Some computer background

In the early days of the data processing industry, computer systems were predominantly designed from the system out to the user; they were technology-driven. In the 1970s this situation began to change as an increasing number of general users, being unable to obtain a satisfactory service from the traditional methods of their data-processing departments, began to seek ways of doing their own personal computing.

Originally, the principal tools that these intrepid pioneers had available to them were mainframe timesharing systems, designed, not for non-technical users, but for systems and applications programmers. With the introduction of the personal computer in the 1980s, this situation changed radically. It is now generally recognized that if the full productivity potential of computer systems is to be realized, then they need to be designed from the human user inwards. This is as true for computer professionals as it is for users whose primary function is in finance, personnel, and other divisions not directly involved with information systems development.

The two most important factors that have led to computer systems being designed from the human perspective are the windows interface introduced by Apple in the 1980s, and object-oriented modelling, which develops systems closely related to the structure of the human mind, reflected in the Macintosh's innovative desktop metaphor.

However, these interfaces don't just happen by magic. Programmers have had to learn to develop programs in quite new ways from those used during the early years of the data-processing industry. As these methods use structures that closely model the deep underlying structure of the Universe, programming becomes much more natural, as professional programmers using these methods know only too well.

Programmers have also had to learn to look much more closely at the way human beings actually work at a computer interface. In the 1980s and 90s, a number of interface design methods evolved to help them, which provide guidelines for a human-oriented approach to

computer systems design. These include Apple's *Human Interface Guidelines*[20] and IBM's *Object-Oriented Interface Design: Common User Access [CUA] Guidelines*,[21] quoted in Chapter 1, 'Starting Afresh at the Very Beginning' on page 50. There are similar design guides for UNIX systems[22] and many for Microsoft Windows systems.[23]

While these guidelines differ somewhat in detail, they all agree that the starting point for sound interface design is the development of a conceptual model or metaphor of the interface that is expressible in terms that are familiar to users' experiences and matches, as closely as possible, their thought processes as they communicate with the machine. So we have seen the metaphor of the desktop come into being, together with folders, documents, and other familiar objects represented as icons.

These developments have led to the term *user-friendly* entering the English language, used, not only for computer systems, but also for any device that is comparatively easy to use. The reason why non-computer specialists can now use computer systems is that there has been a fundamental semantic change in the interface between human users and the computer. Using a technical term from the IT industry, the semantic gap between the technology and the user has narrowed in the trend from technology-driven to human-oriented design. At least, that's the theory. With developers putting more and more complexity into their products, arrogantly trying to second-guess what users want to do, practice actually falls far short of this ideal.

I began to look at the problem of modelling the human-computer interface in earnest during the winter of 1979-80, shortly after I realized that our capitalist economic system held the seeds of its own destruction within it, but long before the modern computer interfaces we see today. At that time, I was wondering how to model the computer programmer's job in the process-entity matrix of an enterprise business model, such as that used in IBM's Business Systems Planning modelling technique, described on page 21 in Part I, 'Integral Relational Logic'.

As such models are developed independently of whether human beings or computers are doing the work of the business, I needed to look at what is common to human thinking and computer programming. This was essential if I were to discover whether a computer could perform a programmer's job without any intervention from a human being.

The central problem that I faced in developing this model is that there is no clear-cut distinction in computer systems between what is a process and what is an entity being processed. As a human being interacts with a computer, what is an entity being processed by the computer sometimes becomes a process acting on some other entity. In the case of personal computer users, this change takes place within a few milliseconds, a change that I found extremely difficult to represent in a process-entity matrix, which is more concerned with operational procedures that take minutes, hours, or even days. Now, many years later, I have found a solution to this problem, which I describe in this chapter.

We can look at the human-machine relationship from three perspectives:

1. A human programming a machine.
2. A computer programming itself.
3. A human 'programming' herself or himself.

We can take it that a machine programming a human being is a 'man-bites-dog' type of problem, and does not need to be considered explicitly. We are not just concerned with how professional computer programmers in software development labs and IS departments interface with the computer; we also need to consider the work of information assistants in user departments using query languages, users themselves doing their own personal computing, and any other use of a computer that can, in any way, be construed as programming.

From here, we shall then look at how a machine rather than a human being might perform the programming task; that is how a computer might program itself. This will lead us to consider how human beings could 'program' themselves, an activity that can be regarded as humans teaching themselves to think.

Computer structure

Before we begin to look at the way human beings program computers, we need to look at the overall data processing function in a computer in the most general manner. We can best begin by looking at the computer as a black box, without considering its internal design. This shows us the basic mechanism of data processing: data is fed in, it is processed by the machine in some way, and data comes out. Figure 2.1, 'Basic data processing function' on page 177, illustrates this process quite simply.

The key point about the basic data-processing function is that it operates through linear time. The input always exists before the output can be produced. It is vitally important to remember this point for it leads to the explanation why it is possible for human beings to program a computer, while a computer cannot do this for itself.

Of course, if we now look inside the computer, we can see that what is actually happening is that there is a program being executed in the computer. So we can revise Figure 2.1 with Figure 8.1.

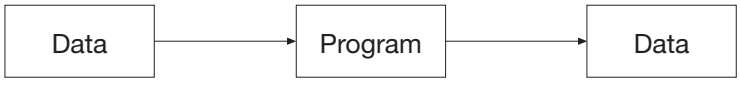

Figure 8.1: *Program execution*

Now in the computer's memory there is no essential difference between programs and the data that they process; they are both data. The central processing unit (CPU) can interpret a bit pattern as either a computer instruction or data to be manipulated. Which it is, is dependent on the context. If the bit pattern is presented as an instruction, the CPU attempts to ex-

ecute it. If the bit pattern is fetched as data to be processed, the CPU acts on it according to its current instruction.

These two types of data we can call active and passive data respectively, as illustrated in Figure 8.2. These correspond to what Charles Babbage called the mill and store in his Analytical Engine designed over 150 years ago, although not built in his lifetime.[24]

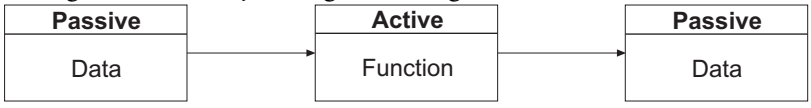

Figure 8.2: *Distinguishing active and passive data*

This pattern is the essential mechanism of mechanical data processing: passive data is transformed into another form of passive data by an active process in some way or other.

However, the modern computer is not like the flat tape of Turing's Universal machine, being directed to move backwards and forwards by the instructions on the tape.[25] A computing system contains many levels of structure arranged hierarchically from high-level languages to machine level. And at each level the data processing mechanism occurs. So it does not matter at which level we view the system; we still see a process taking an input and producing an output. This table shows some examples of what these processes look like on each level.

Level	**Example**
High level language	a = b + c;
Assembler language	ar r_0, r_1
Micro-order	mbr = a + c;
Elementary logic	$(C\wedge(A\vee B))\wedge(A\vee B)$
Sheffer stroke	|ab

Table 8.1: *Hierarchical levels of computer languages*

Starting with a program written in a high level programming language, such as C or C++, this consists of instructions that are translated into machine-level or assembly-level instructions. Each of these instructions is then broken down further until finally the chip is operating on individual bits of data using the basic logic elements, AND, OR, and NOT gates.

An example of how arithmetical operations can be represented by logic circuits is given in Figure 8.3.[26] Here two bits, A and B, are added to a carry over, C, from a previous operation. The result is S, with a new carry over, D. Using the notation of Boolean logic, described in Subsection 'The laws of thought' in Chapter 9, 'An Evolutionary Cul-de-Sac' on page 650:

$$S = (C\vee(A\vee B))\wedge((\sim((C\wedge(A\vee B))\vee(A\wedge B)))\vee((A\wedge B)\wedge C))$$

$$\text{and } D = (C\wedge(A\vee B))\vee(A\wedge B).$$

However, conceptually there is no need to stop at this point. It is quite possible to represent all logical operators in terms of just one, known as the Sheffer stroke, after Henry Maurice Sheffer, who in 1913 discovered this operator, which he called 'rejection', corresponding

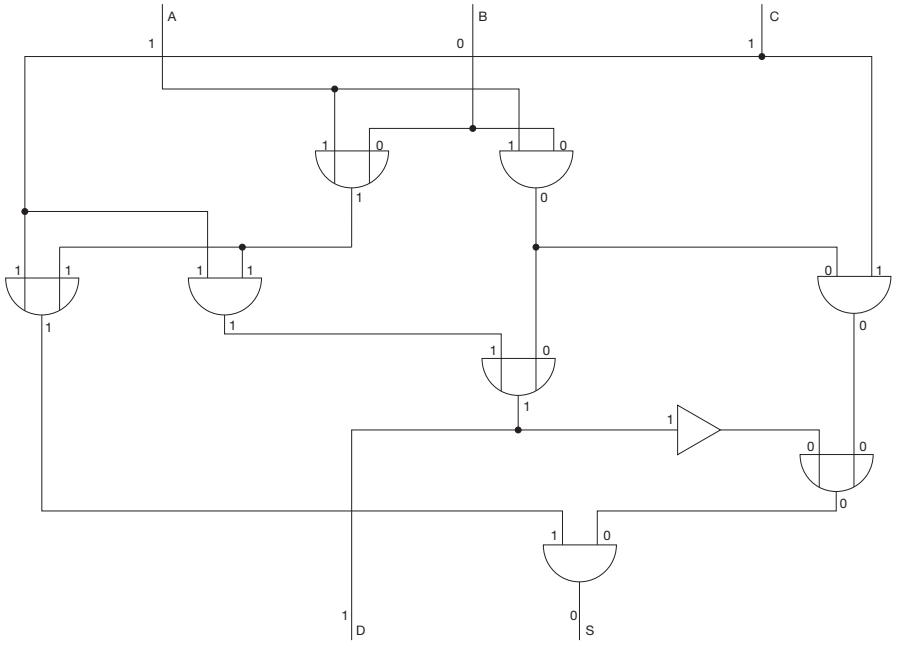

Figure 8.3: *One-bit adder using basic logic constructs*

to the NOR (not-OR) operator, the opposite of disjunction. Sheffer found this function when attempting to reduce the primitive ideas and propositions of Whitehead and Russell's *Principia Mathematica*[27] to the minimum possible, rather like the way the superclass **Being** is the most fundamental concept in IRL, described on page 167 in Chapter 1, 'Starting Afresh at the Very Beginning'. For as Sheffer said, "not all propositions can be proved and not all non-propositional entities can be defined, some logical constants must be primitive, that is, either unproved or undefined."[28]

In a similar paper four years later, Jean Nicod used the stroke as a sign for non-conjunction (NAND),[29] which has since become current practice.[30] In computer science, this is known as a NAND gate, giving:

$$a|b = \sim(a \wedge b)$$

To avoid the use of parentheses, the Sheffer stroke is today written as |ab, using a prefix rather than an infix operator.[31] The basic three operators in propositional (Boolean) logic can then be written like this:

$$\sim a = |aa$$
$$a \vee b = ||aa|bb$$
$$a \wedge b = ||ab|ab$$

So all computer programs could, in principle, be reduced to a long string of Sheffer operators processing individual bits of data, although such a program would, of course, be quite

indigestible. The vitally important semantics that human beings need to understand the function being performed is missing at this level, and, indeed, all levels lower than the semantic model of the program itself.

Douglas R. Hofstadter has called this way of looking at systems at different conceptual levels 'chunking'.[32] For example, chess masters don't look at a chess board in terms of the individual pieces, but as groups of pieces, which have significance from the master's perspective. In a computer, groups of bits or groups of machine instructions can be chunked so that they are meaningful to engineers and programmers working at these levels.

We can thus see that there is no essential difference between hardware and software, they can be considered as a continuum, as Andrew Tanenbaum has pointed out.[33] This is most obviously seen with the operating system (OS), which acts as the control program for the computer, for parts of the OS might be 'hard-wired' into the computer, while other parts are loaded in from external sources.

Indeed, some programming languages can also be implemented in the microcode of the hardware, more usually called firmware. For example, the BASIC interpreter on the early IBM PCs and many home computers and the APL interpreter on the IBM 370/145 mainframe,[34] although these were implemented in different ways.

In more recent times, the programming language Java, which has taken Internet developers by storm, also demonstrates that there is no essential difference between hardware and software. When running on the Internet, a Java applet runs in a virtual machine (VM), with an instruction set that is different from all the physical machines it is running on. But SUN Microsystems have implemented this VM as a hardware chip that can embedded in such things as toasters and television sets.[35]

I cannot overemphasize the importance of this principle. As Tanenbaum has said, "*hardware and software are logically equivalent* ."[36] Whether a particular function is implemented in hardware or software is concerned with practical issues like cost, speed, memory, and flexibility.

So there is no need to get excited about the prospects of a DNA computer, as I have seen reported on the Internet. Such a computer would not add any functional potential to those computers that already exist vis-à-vis human potential. Neither would a computer built with the level of complexity of the human brain, although such a computer might work quite quickly. So why oh why do the computer scientists deceive the public (and themselves) into believing that the functional capabilities of a computer are a property of the hardware? As every computer scientist must surely know, what a programmable computer can do is determined by its software. Analogously, human behaviour is mostly determined by our learning, not from the proteins generated by the DNA molecule or a few chemical scurrying about in the brain.

Human program development

In the early days of data processing, programs that were being developed were punched on to cards or paper tape, which was then fed into the machine for processing. Nowadays, programs are stored within the computer itself as files of data. These programs are created by a human programmer using some form of text editor, which may or may not be part of the language translator, or, since the 1990s, through the use of a visual interface, generally called an integrated development environment (IDE), such as IBM Rational Software Architect, which is driven by the Unified Modeling Language (UML), Microsoft Visual C++ (MSVC), and Apple's Xcode for Mac OS X.

So the computer is now involved, not only in executing programs, but also in assisting in their development. How this is done in practice varies from language to language and from implementation to implementation. The first distinction we can make is between compilers and interpreters.

Whether or not a language is compiled or interpreted is not a function of the language per se, although languages generally lend themselves more to one form than the other. For example, PASCAL programs are normally compiled, although I had a PASCAL interpreter on my Macintosh Plus in the mid 1980s. Conversely, BASIC and LISP programs are usually interpreted, although there are a number of compilers for these languages available.

Compiled programs

We can begin by looking at the job of a professional programmer developing operational applications within a business enterprise or software development house. These programs are normally compiled into a concise executable form. A compiler is a program that translates procedure descriptions from a high-level language into a machine level, most usually.

A notable exception to this is Java. This language is compiled into architecture-independent bytecodes, which can run on machines with any instruction set. To speed up the execution of these programs, a Java applet running under a World Wide Web browser can be compiled when it is loaded, using what is called a Just-in-Time compiler (JIT). In these cases, Java programs go through a two-stage compilation process.

There are two distinct steps: program development and program execution. In the program development phase, the programmer will normally enter her program through a visual interface or an editor, which may check for simple syntax errors if it is part of the language. The machine-readable source program is then input to the compiler, linked with other programs and library routines to become an executable application. Figure 8.4 illustrates program development using a compiler.

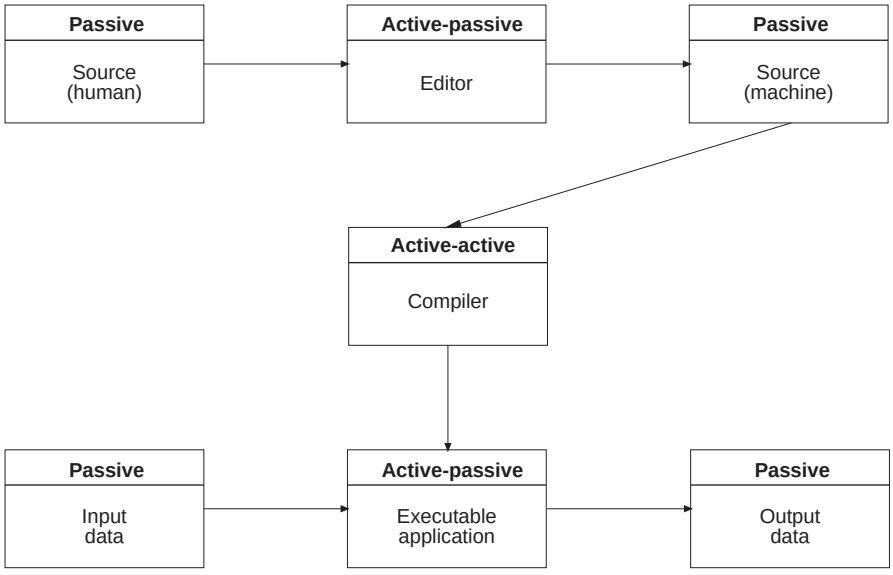

Figure 8.4: *Program development using a compiler*

It is most important to note from this diagram that there are essentially two types of program in computers, those that are used to perform some particular function for a business, like payroll and word processing, called generated programs, and others that generate these programs, like compilers in this example. Both these types of programs are active data. In order to distinguish them, we can call them active-passive and active-active respectively, a refinement of data types, as they are conventionally viewed, as described on page 584 in Chapter 7, 'The Growth of Structure'. Figure 8.5 illustrates the relationships between these more unconventional data types in a computer.

The difference between active-active and active-passive programs can be illustrated by the development and use of a consumer durable, like a washing machine. Before a washing machine becomes available on the market, designers create blueprints that describe its features and components. These are then passed on to a manufacturing plant, which actually makes the product. This is then packaged ready for distribution to the consumer, who can then use the product.

The design and manufacturing phases of developing a consumer durable are analogous to program development. When I began modelling this program development process in 1980, there was very little specific computer support for the design phase. Design documents were produced by typists, before even word processors became generally available.

Today, in the object-oriented world we now live in, this situation has changed radically. There is increasing emphasis on model-driven architecture (MDA), tools that are becoming increasingly integrated within software development tools such as IDEs. As I have been re-

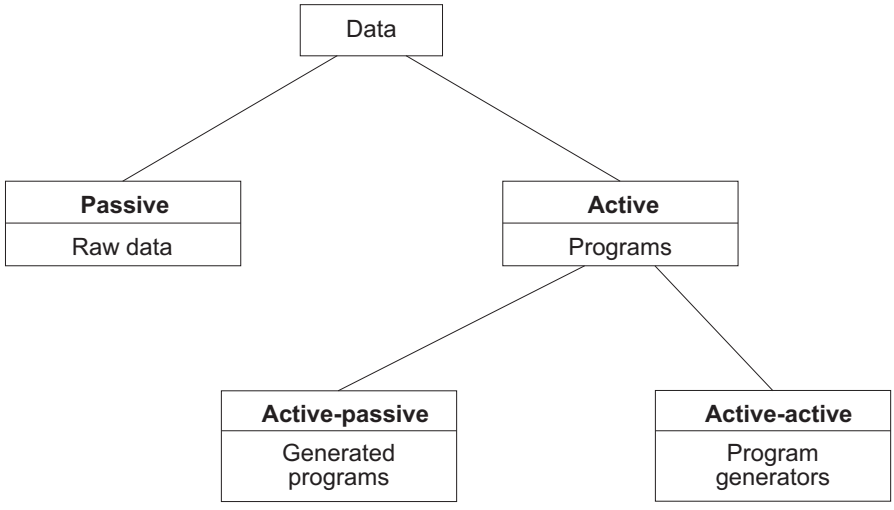

Figure 8.5: *Principal data types in a computer*

tired from the IT industry for several years, I am not up-to-date with this rapidly changing environment. Just a few products that have caught my eye are IBM Rational Software Architect, System Architect from Telelogic in Sweden, now a subsidiary of IBM, and Casewise Corporate Modeler. It was some extinct predecessors to these modelling tools that led to the development of Integral Relational Logic and the Unified Relationships Theory. However, these modelling tools have somewhat complicated the points that I want to make about modelling computer programming rather than modelling the modelling process itself. To keep the exposition as simple as possible, we can consider both modelling tools that are a source of executable code and compilers as active-active data.

Once a software house has developed a product, it is then packaged in the shrink-wrapped package we can buy at our local computer store. At first sight, this process might look like software manufacturing. But it isn't really. This can be best seen by noticing that we can now download software products that have gone through the first two phases of development directly from Internet. In this case, the packaging step is eliminated.

The product is now ready for use, just as we can use a washing machine once we have purchased it. This software product is a generated program, which I call active-passive data. It is this type of program that actually determines the functional capability of the computer.

Notice that a chess-playing program is a generated, active-passive program. Thus studying the capabilities of such programs can tell us little about the nature of the creative software development process. To determine whether a machine can think creatively or not, we need to examine closely the nature of program generators. However, because of the cumbersome way that program compilers operate, they do not really match the dynamics of the human creative process. We therefore need to turn to the other type of program generator: the interpreter.

Interpreted programs

A formal definition of an interpreter that I have picked up somewhere on my travels is "a program that follows an explicit procedure description incrementally, doing what the procedure description specifies". In other words, an interpreter executes each instruction of a high-level language as it is presented to it, without converting a sequence of instructions, or program, to machine-level first.

In the interesting case of a Java applet running under a World Wide Web browser, the compiled program is interpreted in a virtual machine. But as the instructions that the interpreter is processing have been produced by a machine rather than a human being, we don't need to be concerned with this here.

The subject of interpreters is rather complicated because there are so many different ways of interfacing with them. But as my whole approach to learning is to abstract simple patterns that underlie the complexity of the world we live in, let me do this here.

Interpreters usually work in two ways: either immediate or deferred execution, the one being non-procedural in nature and the other, procedural. In the immediate mode of execution, the computer immediately executes an instruction that we give it. The input to be executed needs to be a syntactic whole. This is normally a command or instruction, which looks like a one-statement program to the language interpreter.

One way of doing this is through a prompt in a command window, such as the Terminal application in Mac OS X or the command interface in Windows, inherited from DOS. For example, when I once entered `dir d:\docs` in the Windows command interface, the computer immediately responded by giving me a listing of the `docs` directory on my D drive. Here the command `dir` is active data operating on the passive data `d:\docs`. Similarly, in the Terminal application, which provides a UNIX interface to Mac OS/X, we can get a list of the files and folders in the Documents folder with this statement: `ls -al Documents`.

In immediate execution, the three steps of program development using a compiler illustrated above are combined into one as illustrated in Figure 8.6. In this case, there is no distinction between the instruction to be executed and any data that is to be processed. Both are contained in the one source statement.

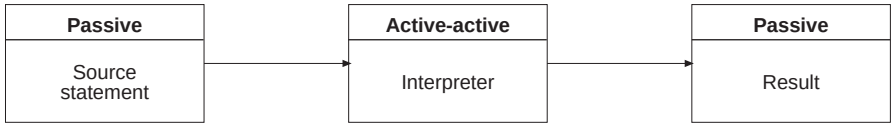

Figure 8.6: *Immediate execution of an intepreter*

In deferred execution, statements of an interpreted language are stored in the computer as a sequence of instructions, or programs, for later execution. These programs are usually given names and so become new entities that can be interpreted by the interpreter. The main dif-

ference between such programs and ones generated by a compiler is that a compiled program runs directly under the operating systems independently of the compiler, while an interpreted program executes within the environment provided by the interpreter. This is illustrated in Figure 8.7.

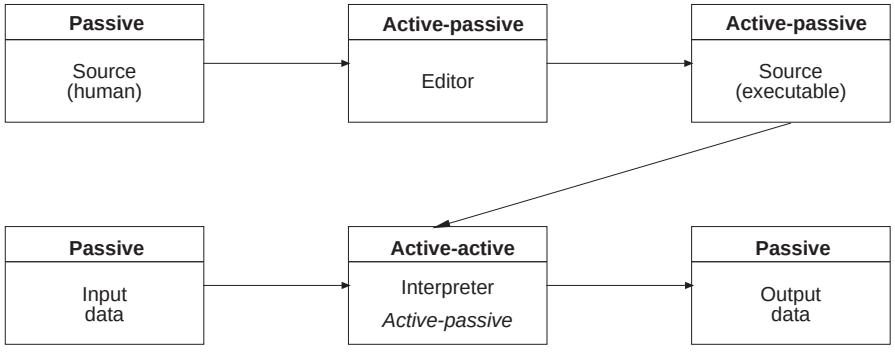

Figure 8.7: *Deferred execution of an interpreter*

How then does the interpreter distinguish between these two modes of operation? This varies widely from language to language. I collected several examples during the 1980s and 90s to illustrate this distinction, as listed in Table 8.2. No doubt others could be added with more up-to-date examples.

	Execution mode from terminal	
	Immediate	**Deferred**
APL	Everything else	∇ (begins function def) or program window
Basic	No statement no.	Statement no.
HyperTalk	Message box	Script editor
LISP	Well-formed list	DEFUN (begins function def)
MS/DOS	Command	Text editor to create file
MVS/TSO	Command	ISPF/PDF editor
PostScript (Host)	Executive command	Program stream from host
PostScript (Printer)	Object with executable attribute	Executable array encountered directly
Python	Statement	Text editor to create module as file
PROLOG	? (instruction for goal)	consult(user).
REXX	User-written program	Normal method
SQL	Interactive mode	Embedded in host language (Dynamic SQL in interpretative language)
VM/CMS	Command	XEDIT or other text editor

Table 8.2: *Modes of execution in interpretative languages*

Computer-driven program development

So far we have been looking at the programming process as it is performed by a human programmer. However, a number of languages provide facilities for programs themselves to generate and execute programs dynamically. The language that drew my attention to this possibility is A Programming Language (APL), a language that was used extensively in IBM in the 1970s as a personal computing language for management information.

APL was initially developed by Kenneth Iverson when he worked at Harvard University in the late 1950s as a mathematical notation to assist students in analysing various topics in data processing.[37] It became a programming language in 1966 after Iverson joined IBM. I myself learned something of the language in 1978 when I was responsible for selling IBM personal computing products in IBM's North London sales office.

APL is perhaps the most concise of languages, using just one or a few symbols for functions that would require several statements in other languages, such as matrix inversion and division (⌹) and matrix multiplication (+ . ×), where . denotes a general-purpose inner-product operator whose operands are other operators. APL was also designed from the outset as an interactive language, enabling human beings to communicate directly with the computer.

However, it is not these features of the language that I am concerned about here. APL also has some little known features that enable an APL program to create, modify, and erase other programs without the intervention of the human programmer. APL is not the only language that has these capabilities; there are a few others that I shall mention. It is not good programming practice to write programs that dynamically modify themselves, because they are notoriously difficult to debug. But there are occasions when such a facility is useful. Otherwise, they would not have been built into the language.

In the words of Gilman Rose, "This leads to application systems that can appear intelligent (in the sense of programs that write or edit other programs)".[38] As it is these facilities that are mimicking the creative programming process of a human being sitting at a computer terminal, we now need to study these languages in more depth.

The subset of programming languages that provide commands to enable a program to be created or modified from within the program and then to be executed dynamically without the agency of a human programmer we can call dynamically active. In contrast those languages that are statically active, such as BASIC, C, COBOL, FORTRAN, PASCAL, and PL/I, do not have such a facility. It is not possible, for example, to modify a program written in C within a program written in C and then to execute that program from within the program. While it is theoretically possible for a C program to modify, or even create C source statements from

some other form of data, it is not then possible to execute this program from within the C program.

As dynamically active languages are more procedural than non-procedural in nature, we can call them, in full, Dynamically Active Procedural Programming LanguagEs, or DAPPLES for short. Examples of dapples are APL, Python, HyperText, the language of Apple's HyperCard (now withdrawn), LISP, a list processing language developed by John McCarthy of MIT, and PostScript, a page description language for high quality printers created by John Warnock of Adobe Systems.[39] We can use these examples to see the nature and diversity of these types of languages. All these languages have two particular features, both of which are probably necessary for a language to be dynamically active.

The first of these is that they are all extensible. That is, they have an open grammar as opposed to the closed grammar of most conventional programming languages used in business. This means that programs written in extensible languages become new functions of the language, which syntactically are treated in exactly the same way as the primitive functions of the language. There is thus no distinction between commands and functions as there is in non-extensible languages.[40]

Secondly, dapples treat active and passive data in a similar way, to a greater or lesser extent. This can most simply be seen when using these languages interactively. If the name of a variable is presented to the translator, its value is immediately returned. Similarly, if the name of a function is entered then this is executed, and the result is again displayed on the screen. To give a simple example of this phenomenon, as addition is generally a primitive function of dapples it is possible to enter 2+2 in a syntax recognized by the language and to receive the answer 4. Dapples can thus be used as rather expensive calculators!

It was this situation that led me to the difficulty of representing active and passive data in a process-entity model. Most particularly, conventional modelling techniques are not able to satisfactorily represent the read-eval-print loop in LISP, which well illustrates the fundamental mechanism in data processing. In LISP, evaluating passive data simply means to return its value, given its name, while evaluating a function is just another name for executing it.

The key point to note is that when used in this way, dapples perform the translation and execution processes immediately and consecutively. In other words, a statement presented to it is passive data to the translator and active data to the evaluator. To separate out these two functions, it is necessary to tell the translator to defer execution in the manner outlined above.

Now, as I have said, the key characteristic of some dynamically active languages is that they are able to modify and create programs written in the language and then immediately execute them in exactly the same way as they would translate and execute a program presented by a human programmer. Rather than using the text editor built into the language, dapples do this

by regarding programs as strings of characters, which can then be manipulated by the string or symbol handling facilities of the language like any other symbols.

There are three steps in the human programming process that we need to look at:

1. Telling the computer that a program is to be entered and not executed.
2. The entering of the program at the terminal or modifying an existing program.
3. Presenting that program to be executed.

The second of these is the essence of human computer programming, the creative part. The other two functions are mechanical in nature, and act in support of the programming function. To examine how a computer might simulate the human programming function, we need to look at how these three functions can be programmed into a program.

In a dapple, a program can mechanically perform all the tasks mentioned above, but typically in a somewhat different way from the human programmer. Let us suppose, then, that a human programmer has written a program that can write new programs, and execute them dynamically.

Dynamically deferring execution

First of all, the program must tell the interpreter not to execute the instructions presented to it by the program, but to go into deferred execution mode. Whether it uses the same facility as the human programmer depends on the language and the environment. What is needed is the facility to convert programs, stored in their active, executable state, to a passive form that can be manipulated by the program. The modified program then generally needs to be converted back into its active form by some means. Alternatively, if the computer program is creating a completely new program from scratch, only the second of these facilities is required. Table 8.3 shows how active data can be converted to passive and vice-versa in the five languages we are looking at as examples.

	Deferred execution from program	
	Active→Passive	**Passive→Active**
APL	`⎕CF`	`⎕FX`
HyperTalk	*Get the script*	*Set the script*
LISP	Not known	*DEFUN*
PostScript	*cvlit*	*cvx*
Python	Not known	*def*

Table 8.3: *Converting active and passive forms of program dynamically*

For example, in APL, `⎕CF` creates a matrix from the function, whose dimensions are the number of lines and the maximum number of characters in a line. In APL*PLUS, there is also a function, `⎕VR`, which converts the function to a vector representation. `⎕DEF`, a more

powerful version of ☐*FX*, converts both matrix and vector representations to active functions.

In LISP, both functions and function definitions are lists, for *LISP* stands for *LISt Processing*. For instance, a human programmer could enter this statement to define a function to square a number, which could then be evaluated:

```
CL-USER 1 > (defun square (x) (* x x))
SQUARE

CL-USER 2> (square 8)
64
```

As function definitions are lists, a human programmer could write a function definition that sets a variable to a list that defines a function as passive data. This could then be converted into active data with `eval`. Then this function could be executed. Here is an example in interactive mode:

```
CL-USER 1 > (setq funcdef '(defun square (x) (* x x)))
(DEFUN SQUARE (X) (* X X))

CL-USER 2 > (eval funcdef)
SQUARE

CL-USER 3 > (square 7)
49
```

Dynamically editing programs

When the computer creates or modifies a program, it generally needs another method of doing so from that used by the human programmer; it cannot use the text editor directly because this is intended for people. So generally there is no way of invoking the text editor from within the language.[41]

Instead, program interpreters that create or modify programs do so by using the string or symbol handling facilities of the language. The purpose often is to assist in the programming function; to use a program to manipulate other programs rather than doing this process manually, which can be tedious work. The text, of course, needs to conform to the syntax of the language in order for it to be recognized by the interpreter when it is converted into executable form. In mathematical terms, the string of symbols needs to 'well-formed'; otherwise there will be a syntax error when the interpreter tries to execute the statement.

In an extensible language, programs that are created in this way, or by a human programmer, become new instructions in the language. The power of the programming language is thus increased whenever a new function is created.

Now, two types of program can be created in this way: either the program is a simple application, which processes passive data, and so can be called active-passive, or it may contain instructions to enable it to generate new functions to be added to the language. In this latter case, the program is active-active. When this program is then executed by the interpreter, there is thus an active-active program running another active-active program beneath it.

Two key issues relating to whether computer programs can be creative or not arise from this mechanism. The first is that, in theory, a program could be written that would indefinitely go on creating new active-active programs to be added to the language. This process is somewhat like a program that calls itself recursively in that the process needs to be terminated in some way if it is not to lead to an infinite regress, a repetitive process illustrated in Figure 7.3, 'Programming iteration block' on page 583. However, it is not exactly the same.

A recursive program is active-passive and is following some well-defined algorithm in a predetermined manner. After a specified number of iterations, this process terminates, and the program returns a result. The PostScript program that produced a fractal fern in Figure 1.38 on page 134 in Chapter 1, 'Starting Afresh at the Very Beginning' was written recursively. The diagram you see is the result. The purpose of an active-active program, on the other hand, is to create new functions dynamically. This is not a mechanistic iterative process, so there is no purpose in writing a program that continues to create active-active programs indefinitely.

The second key issue is where do the symbols come from that are to form the new or modified program? Within the computer these cannot be created out of nothing. Every symbol, which is to form part of the new program, must exist before the program can start. The computer cannot create new symbols that are meaningful in any real sense. The meaning of symbols is something that is only understood by human beings.

Dynamically executing programs

Having created or modified a program, there now needs to be a means of dynamically executing it. In extensible languages, programs are executed simply by entering the name of the program. However, sometimes it is necessary to explicitly execute a program. This is necessary when the program does not have a name; for instance, it consists of just one, or a few, statements. Table 8.4 lists some of the commands that do this.

In APL, for example, ⍎ also acts like ⎕*FX* or ⎕*DEF* in that passive data can be executed directly by this execute function. This does not need to be in the form of a function. For example, ⍎ '*A*+*B*' is exactly the same as *A*+*B*. There are times when it is more convenient to express this operation using execute, rather than with the active data directly. Variables can be assigned these active pieces of code, as passive data, and then executed. The `INTERPRET` instruction and the `VALUE()` built-in function provide similar facilities in REXX.

	Immediate execution from program	
	Implicit	**Explicit**
APL	*Function name*	⍎ or ⎕*FX*
HyperTalk	Script name	Script name
LISP	Function name	*EVAL*
PostScript	Executable array encountered indirectly	*exec*
Python	Not known	*def*

Table 8.4: *Dynamically executing programs*

The whole process of computer-driven program development and execution is depicted in Figure 8.8. The first line shows a human programmer creating a program that can execute under the control of an interpreter to create another program, shown in the second line. But is it possible to dispense with the second line in the diagram so that this diagram reduces to Figure 8.7 on page 632, but with a machine providing the initial input not a human being? In other words, could a computer initiate the process of programming itself, or can a machine only do what we tell it to do, as Ada Lovelace surmised 150 years ago?

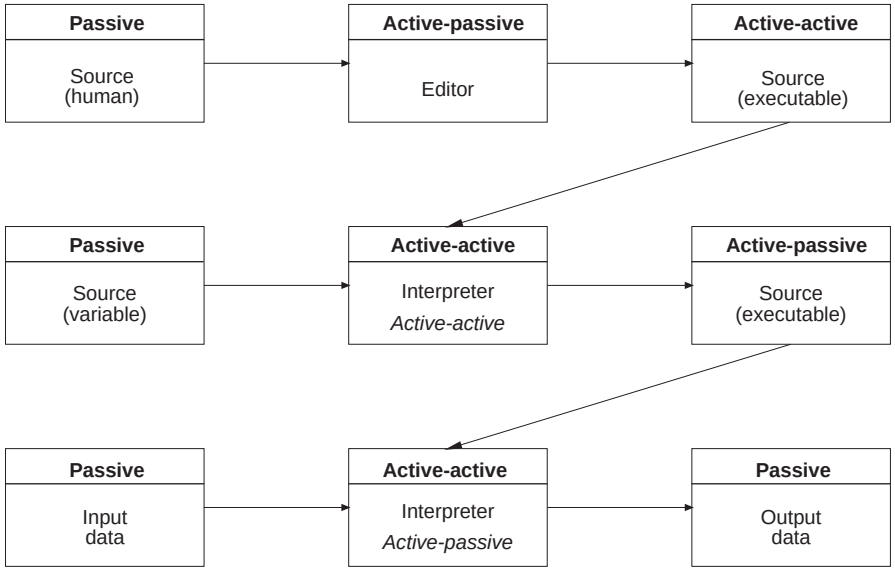

Figure 8.8: *Program development with dynamically active languages*

The key to the answer to this question lies in the fact that the basic data processing function in computers always operates through linear time. Every program that has ever been written has been helped on its way by a program generator along the lines that I have described in this chapter. There is thus a long mechanical cause and effect chain going back to when the very first computer was built. So who built the first computer? Where did this come from? How is it that we human beings can create computer programs without apparently having some already pre-existing program to get us started? To see the answer to these questions, we

must now look at the characteristics of human beings that correspond to the features of computing systems that I have been describing.

Analogous human cognitive characteristics

Not unlike computer systems, we human beings, also process data. However, rather than taking in data that is input to us, the Totality of Existence provides us with the data that we process. As our inner cognitive faculties are part of the Universe, they also provide us with data. We process all this data by using our intelligence and intellect to interpret the data patterns of our experiences, which gives us the knowledge and information that we need to live in the world of practical affairs.

So our cognitive faculties can be divided into two groups, just as in a computer. Our knowledge and information correspond to passive data and intelligence and intellect to active data. Our passive cognitive faculties—our knowledge—can be further subdivided into two categories. As Gilbert Ryle has observed, human knowledge can be considered both as the facts we know and the skills we know how to perform.[42] These we can call passive-passive, which reduces to passive without any loss of meaning, and active-passive respectively. Human knowledge is thus analogous to 'raw' data and generated programs respectively.

Our intelligence and intellect also naturally fall into two parts. First of all, it is our intelligence that drives our thinking and learning, which are also skills that are quite different from our other skills. Thinking is the skill that helps us to learn skills, including thinking and learning themselves. We can see thinking as a visualization process, that sometimes, but not always, helps us with the development of our skills. Thinking is most useful in developing mental skills, such as playing chess and computer programming. But it is less useful in learning to play the piano or to speak a foreign language, for instance, where feeling is vitally important.

In general, the more that we think about the development of our skills the more proficient at them we become. Most particularly, in my case, because I have developed a model of my learning skills since 1980, I can now learn at a very rapid rate. If this were not so, it would have been quite impossible to develop a framework for a synthesis of everything. I would have been quite overwhelmed by the task.

While our intelligence lies behind our thinking skills, it is intellect that drives our reasoning skills. Sometimes, when intelligence sees a new pattern, a new concept or thought arises, which we can store as a mental image to which we can attach words and other symbols. Once we have formed these concepts, we can then arrange them in a multitude of ways according to the rules of Integral Relational Logic, which embraces traditional Aristotelian logic within it. We can call this process reasoning.

Reasoning doesn't really assist in the development of our skills. It is a mental process that enables us to derive new facts from already existing concepts, while thinking is the process of creating those concepts in the first place, as illustrated in *Integral Relational Logic (to be moved)*. Reasoning is thus more like a developed skill, which, of course, we can improve through practice. As is well known, no new knowledge is ever created through the deductive logic of mathematics.

We can therefore consider intelligence, which lies behind our thinking and learning skills, to be analogous to program generators in computers, while intellect, which drives reasoning, corresponds to developed programs. Figure 8.9 shows these relationships, which are analogous to the data types in Figure 8.5 on page 630.

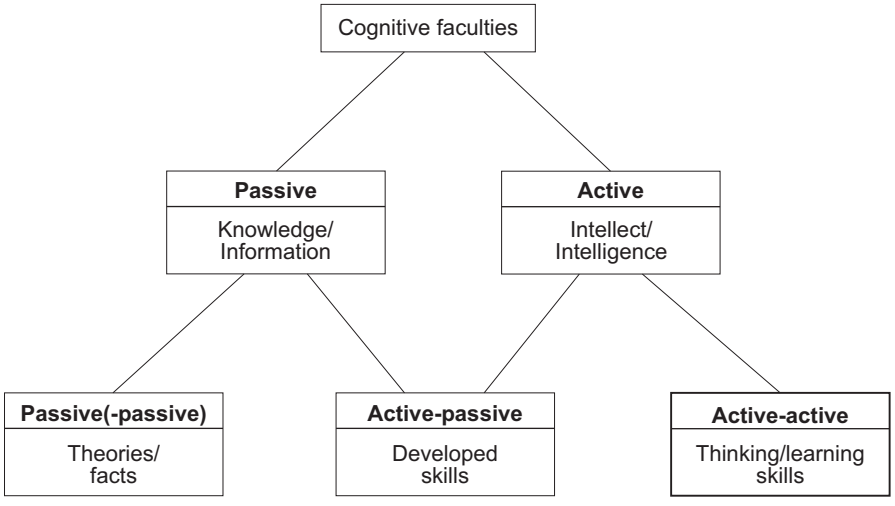

Figure 8.9: *Analgous human cognitive faculties*

So can machines think? Well, the analysis in this chapter shows quite clearly that the feature of a computer system that is closest to our thinking and learning skills is a program written in a dynamically active programming language programming itself. But we have seen that for a dapple to write programs, the symbols that it uses must already exist. All mechanical processes are trapped in the linear, horizontal dimension of time.

We can compare this perspective to the reasoning that led Aristotle to the existence of an unmoved mover, which led Thomas Aquinas to the first of his five 'proofs' for the existence of God. As Aristotle said, "That which is moved must be moved by something, and the prime mover must be essentially immovable, and eternal motion must be excited by something eternal."[43] In Thomas' words, "Now anything changing is being caused by something else. … Now we must stop somewhere, otherwise there will be no first cause of the change, and, as a result, no subsequent causes. … We arrive then at some first cause of change not itself being changed by anything, and this is what everybody understands by *God*." [44]

In terms of computer programming, every program that has ever existed has come into being through another program. So where did the first program come from? There is only one possible answer to this question. As the first program cannot have originated along the horizontal dimension of time, through an endless string of cause and effect processes, by the Principle of Unity, it must have arisen from the vertical dimension, from the timeless, formless, Absolute Whole, from God the creator. This means that Richard Dawkins' program *The Blind Watchmaker*, which is designed to show that evolution progresses without divine intervention, could not have become manifest without the creative power Life arising from our Divine Source. All scientific theories that deny the existence of Wholeness are actually being created by those very same energies whose existence the theories deny!

In general, it is only possible for us to use our intelligence to develop new concepts and to learn new skills because of the divine energies within us. This means that our creative learning is the leading edge of all evolutionary processes on this planet, not the development of computer programs. It is thus a great delusion to believe that technological development can drive economic growth indefinitely. The only way forward for humanity is therefore to focus our attention on our spiritual awakening and liberation, being guided to Oneness and Wholeness by the Principle of Unity: Wholeness is the union of all opposites.