# Contents

# AI lecture 6

Jocelyn Ireson-Paine

February 14, 1996

## 1 Introduction — Computation and computers

In this lecture, I'll describe how computers work, and introduce some ideas from computer science. You won't be expected to know how a computer works for Finals. However, it's worth doing so because: they're useful tools; the mind-as-computer metaphor has invaded psychology, and you'll find pieces of jargon from it; and many researchers believe that thought is computation, and that a simulation of thought would itself be thinking. Compare e.g. a simulation of an ice-cream factory, which may be isomorphic to the real thing, but does not itself produce ice-creams.

## 2 What's in a computer?

Most of today's computers are *serial*. Their memory contains a large array of instructions, and they obey these one at a time. This is true of PC's, Macs, the University's main computers (a Vax and a Dec 5500), etc. One can also build *parallel* computers, which have more than one processor and so can obey more than one instruction at a time. These are less common, although big parallel computers are widely available as research facilities: Connection Machines, Crays, etc.

The design of parallel computers varies widely between machines, and I won't say more about them. Serial computers are less varied. Here's an account of the main components of a typical serial computer.

- *Memory*. Or *store* if you want to be less anthropomorphic. Used to hold instructions (the program) and data.

  Consists of a sequence of *words* — nothing to do with words in English! Words are just sequences of binary on-off switches, each switch being able to store one binary digit or *bit*. All words contain the same number of switches (usually 16 or 32). Altogether, the settings of the switches in a word make up a binary numeric value.

  Each word in memory has its own *address*, determined by the electronics. The addresses run sequentially, usually from 0 upwards.

- *Program counter*. This is a one-word register, separate from the rest of memory. It "points" at a word in memory. I.e. it holds a number which is understood by the processor as the address of a word in memory. This number is the address of the instruction currently being executed.

- *Instruction register*. Another one-word register, used to store the instruction currently being executed.

- *Accumulator*. Used to hold data from memory. A kind of very small STM.

- *Central Processing Unit* or CPU.

- *Peripherals*. Terminal, discs, printer, etc.

## 3 Machine code

Program instructions are represented as numbers in memory. Each type of computer has its own way of encoding these: its own *machine code*.

Example. I'll use this in the next section.

- 41xx. "Clear". Store zero in memory location xx.
- 42xx. "Read". Read a number typed at the keyboard and store it in the accumulator. The value of xx is ignored.
- 43xx. "Print". Display on the screen the number in memory location xx.
- 44xx. "Load". Copy into the accumulator, the number in memory location xx.
- 45xx. "Load". Copy from the accumulator into the number in memory location xx.
- 46xx. "Add". Add the accumulator to the number in memory location xx.
- 47xx. "Goto". Copy xx into the program counter.
- 48xx. "Jump if zero". If the accumulator is zero, then copy xx into the program counter, otherwise do nothing.
- "Stop". Stop.

In this example, I've allocated the first two decimal digits for the instruction part, and the last two for the "operand" — an address or data value needed by some instructions. When decoding the instruction, the processor has to split the first two digits off from the last two. This would be easy if it worked in decimal. However, no processors do, and an actual machine code would rely on a binary representation. For example, the first five binary digits might be reserved for the instruction part. I'm using decimal only because it's easier.

## 4 The fetch-decode-obey cycle

The idea is that the computer keeps running through a cycle which consists of three stages:

1. **Fetch**. Copy into the instruction register, the contents of the memory location addressed by the program counter.
2. **Decode**. This contents will be a number. Separate the part which describes the instruction itself (first two digits in our example) from the rest.
3. **Obey**. Obey the instruction. Then, unless it has already changed the program counter, add one to it so it points at the following location in memory.

Assume the contents of memory is as follows:

**0** . 4107.

**1** . 4200.

**2** . 4805.

**3** . 4607.

**4** . 4701.

**5** . 4307.

**6** . 4900.

**7** . 3458.

Assume that the program counter initially contains zero. We now repeat the fetch-decode-obey cycle as follows, where ACC = accumulator, IR = instruction register and PC = program counter.

**Fetch 1**. Put into IR the contents of the memory location addressed by the PC. So IR = 4107.

*bf Decode 1*. What does the IR mean? Understood as an instruction, it means "clear location 7".

*Obey 1*. Location 7 is now zero. Add 1 to the PC, so it's now 1.

**Fetch 2**. Put into IR the contents of the memory location addressed by the PC. So IR = 4200.

*bf Decode 2*. What does the IR mean? Understood as an instruction, it means "read a number from keyboard into the accumulator".

*Obey 2*. Do this, and assume the user typed 9. So ACC=9. Add 1 to the PC, so it's now 2.

**Fetch 3**. Put into IR the contents of the memory location addressed by the PC. So IR = 4805.

*bf Decode 3*. What does the IR mean? Understood as an instruction, it means "If ACC=0, then copy 5 into the PC".

*Obey 3*. Since ACC isn't zero, don't do the copy. Add 1 to the PC, so it's now 3.

**Fetch 4**. Put into IR the contents of the memory location addressed by the PC. So IR = 4607.

*bf Decode 4*. What does the IR mean? Understood as an instruction, it means "Add ACC to memory location 7".

*Obey 4*. Do this. Memory location 7 is now 9. Add 1 the PC, so it's now 4.

**Fetch 5**. Put into IR the contents of the memory location addressed by the PC. So IR = 4701.

*bf Decode 5*. What does the IR mean? Understood as an instruction, it means "Copy 01 into the PC".

*Obey 5*. Do this. The PC is now 1. Don't add 1, since it's just been changed.

**Fetch 6**. Put into IR the contents of the memory location addressed by the PC. So IR = 4200.

…and so we can go on. If you assume that the user types (say) 11, then 13, and then 0, you'll see that this program adds the non-zero numbers and stops when the user types zero.

This demo uses a very simple architecture. Real computers have many more instructions, much bigger memory, more accumulators, etc. They do not usually have one instruction just to print and another to read. Also, as I said above, real computers will *always* encode their instructions in binary, so that instead of splitting off the first two decimal digits, you split off (say) the first five binary ones. But the principle is the same.

# 5 Stored programs

There is nothing about the numbers in memory that marked some off as instructions and the others (location 7 in our example) as data. It depends on where you start executing. Early computers weren't like that: the programmer input a program by re-wiring part of the computer (perhaps by changing switch settings). The idea that a program could be stored in the same place as data was an important advance. Amongst other things, it meant that one could write programs such as compilers (see below) which operate on other programs. More generally, programs could be treated as data, or as instructions.

The idea of the stored-program computer contributed to psychology. It was a model used by Karl Lashley, who realised that the serial ordering of behaviour couldn't be explained by behaviourist S-R associations. For a brief history of early computer science, connectionism and cybernetics, including their influence on psychology, see Chapter 1 of *Brains, Machines and Mathematics* by Michael Arbib (2nd edition, Springer-Verlag 1987; RSL Comp Bd 98; chapter 1 copied as AI box photocopy A52). [1]

# 6 Hardware versus software

In describing the behaviour of my example computer, the details of the hardware are irrelevant. It's only necessary to say how it pushes the numbers around. The same behaviour could have been implemented on one of many types of chip, or mechanically, or by fluidics, or (as I did) by a person simulating it. This gives rise to the idea of *hardware/software independence*, and to the idea that one can explain the operation of computational systems in information-processing terms, without reference to the hardware.

# 7 Functionalism

So what matters is — in some sense — the abstract structure of the software, not the way it's implemented in hardware. This idea has contributed to that of *functionalism*, one attempt to answer the question "What is a mental state"?

One of the early replies: the identity theories. Mental states are physical states of the brain: each mental state is the same as a certain physical state or process. E.g. someone can have pain only if they have *c*-fibres and these are firing. However, this seems to concentrate too much on biological make-up. How could it apply to a Martian, or even to a human with a prosthetic aid which replaces part of the nervous system?

The reply to this: *functionalism* (Putnam). What's important is the *rôle* of the *c*-fibre firings in the organism's psychology. So Putnam compared mental states to program states in a computer. In his words, the defining feature of a mental state is certain abstract relations between it and (a) the environment; (b) other mental states; (c) the possessor's behaviour. You can find Putnam's own articles in *Minds and Machines* from *Dimensions of Mind* edited by Hook (NYUP 1960; PSY AD:H 76) and in *The Nature of Mental States* from *Mind and Cognition: a Reader* edited by Lycan

[1] It also influenced biology. DNA acts both as a genetic program, and (when being copied) as data. Although I haven't traced the history in detail, I suspect that our ideas about DNA's function owe a lot to computer science. John Von Neumann did a lot of the early work on computers, and he also produced a theoretical analysis of self-reproducing systems (in general, not just biological ones), showing that there were certain logical principles they had to follow. See pages 26–46 of *Artificial Life* by Levy (PSY KH:L 057). This includes a description of his self-reproducing automaton, and an account of how these ideas influenced space exploration.

(Blackwell 1990; PSY AD:L 098). The latter has a summary of the ideas on page 7. Putnam's *Minds and Machines* essay shows clearly that he's based his ideas on hardware/software independence. For another account, see page 23 in *Microcognition: Philosophy, Cognitive Science and Parallel Distributed Processing* by Clark (MIT 1989; PSY KH:C 054).

We can make this clearer with our machine-code program. We might say that at a certain point in the program (just before the 4307 instruction), our machine is in the state "about to display the sum so far". The essential thing about this state is that which would remain unchanged, no matter what the hardware. Consider the same program: being simulated by me; being simulated by a program running on a Mac; running on a purpose-built silicon chip. There is some sense in which, just before displaying, each program is in the same state. That's the essence of the mental state. Note that the state is partly defined by the states that can precede and follow it.

We can have different levels of functionalism. Neural functionalism says it's the input-output behaviour of the neurons, and their connectivity, which is important. Symbol-level functionalism is that it's the *symbol level* that's important. (This carries the assumption that there is a distinct symbol level.)

# 8 Two games

Another example, of a different kind, to demonstrate this idea of abstract similarity.

## 8.1 The 15 game

In the first game, there are nine numbers, 1 2 3 4 5 6 7 8 9. You and your opponent take turns, taking one number each time. You can only take a number once, so if your opponent has taken a number, you can't take it also (and neither can your opponent, again). The winner is the first person whose numbers total 15.

Here's an example:

```
                                  You         Your opponent
    1 2 3 4 5 6 7 8 9

You take 1

      2 3 4 5 6 7 8 9             1

Opponent takes 8

      2 3 4 5 6 7   9             1           8

You take 5

      2 3 4   6 7   9             1,5         8

Opponent takes 4

      2 3     6 7   9             1,5         8,4
```

```
You take 9, and win!

      2 3     6 7                 1,5,9       8,4
```

## 8.2 A word game

Now try this one. Like the above, you play with nine things, but they're words, not numbers. In fact, they're these words:

```
BRIM     SHIP     WOES     FORM     HEAR
HOT      TIED     TANK     WASP
```

To play, you start by writing down all the words once. Then take turns crossing out a word and initialling it. The winner is the first to get three words with the same letter in common.

## 8.3 Equivalence between games

In fact, both these games are equivalent (the technical word, taken from maths, is *isomorphic*, "of the same form") to each other, and to noughts-and-crosses. See below for how they correspond. This should begin to show you how apparently wildly different ways of representing and manipulating information can be, at a deeper level, the same.

Of course, we perceive the isomorphism because of the properties we're interested in. The 15 game has no analogue for the fact that one of the words in the word game is a different length. But for the properties that we consider essential to the game, they're isomorphic. This is not just a vague notion of similarity: we can define it exactly. Part, not all, of the meaning of isomorphism is that we can put the components of one game into one-one correspondance with those in the others:

| Noughts-and-crosses | 15 game | Word game |
|---|---|---|
| Top left | 1 | HOT |
| Top mid | 2 | TANK |
| Top right | 3 | TIED |
| Mid left | 4 | FORM |
| Mid mid | 5 | HEAR |
| Mid right | 6 | BRIM |
| Bot left | 7 | WOES |
| Bot mid | 8 | WASP |
| Bot right | 9 | SHIP |

But there's more to isomorphism than that. After all, we can put the board positions in one-to-one correspondance with the planets, but that doesn't mean the Sun spends all its time playing Alpha Centauri at noughts-and-crosses. We can also map properties and actions to one another:

| Noughts-and-crosses | 15 game | Word game |
|---|---|---|
| In same row | Add up to 15 | Share a letter |

```
Put a mark                 Add a number    Initial a word
Cross/                     Your pile/      Your initials/
zero                       oppt's pile     oppt's/initials
```

That's the second part of isomorphism. It too is not enough. What's important is that objects, their properties, and the actions interact in the same way in different games. We can express this by trying to make rigorous the idea that if you start off by playing one game, and then change to another, nothing has really changed.

To give an example: you start a game of noughts-and-crosses by putting a cross in the top-left cell. Your opponent makes a move. Let's say he puts a zero in the bottom-left. Now you put a cross in the centre of the board. Then your opponent puts a zero in the top-right square. And finally you put a cross in the bottom right square. You have a row! Now, you translate this final state to the 15 game:

```
Noughts-and-crosses                    15 game

You: top-left, centre, bottom-right    You: 1, 5, 9
Oppt:bottom-left, top-right            Oppt:7, 3
Row top-left to bottom-right           Sum of 1+5+9 = 15
```

What isomorphism means is that, if you'd played the 15 game all along, making moves that correspond to your noughts-and-crosses moves, you'd have got to the same state as if you'd played the noughts-and-crosses moves and *then* converted back to the 15 game.

## 9 Weak and strong AI

Taken to its limit, functionalism implies that if we can replicate in a computer the abstract relations that define a mental state, then we have replicated the state itself — including the behaviour it causes, and the feeling of subjective awareness that goes with it. This stance is known as *strong AI*.

Neural functionalism would do this at the neural level. It doesn't matter whether we replicate the input-output behaviour and connectivity in real neurons, or silicon equivalents, or (the "Chinese brain" scenario) people simulating them. Either way, we'll build a system which generates the same behaviour, and has the same subjective awareness as the brain it mimics. There's an amusing dialogue about this in *A Conversation with Einstein's Brain* by Hofstadter, from *The Mind's I* edited by Hofstadter and Dennett (PSY AA:H 067; Philosophy; Hooke). [2]

Symbol-level functionalism would do this at the symbol level: the level of classical AI.

*Weak AI* backs off from such extreme claims, and says merely that the computer is a valuable tool — e.g. in "running" theories so as to test them more rigorously.

## 10 Functional architecture

The distinction between hardware and software is also related to the idea of *functional architecture*. When we describe a Mac, what's important is not its physical

[2] The idea crops up frequently in science fiction, often in the extreme form of "copying" brain software into a computer. See for example *Dust* by Greg Egan, from *Best New SF 7* edited by Dozois. There's a copy of this as AI box photocopy E53. The story illustrates some of the points about functionalism and the progress of time that worry its opponents. One of these is Roger Penrose in *The Emperor's New Mind* (OUP 1989; PSY KH:P 038). Penrose believes that before we can understand consciousness, we need to understand quantum mechanics and time, which we certainly do not now.

architecture (layout, etc) but its functional architecture. I.e. the basic processing elements (accumulator, memory, program counter, instruction register) and operations (the machine-code instructions).

As I've said above, you could realise the same functional architecture on machines with widely differing physical structure. It's possible of course to have many different functional architectures. The differences might be big — the difference between a production system and my example computer. They might be smaller — a difference in the number of registers, or in the details of the machine code. Or they might just be a matter of different restrictions — differing size of memory.

This is where the idea of *the functional architecture of cognition* comes from: see the production-systems lecture and my notes on production systems for the Prolog practical.

## 11 The development of programming languages

In early stored-program computers, the programmer had to get programs into memory by flipping binary switches on the front panel. A little later, came the idea that you could punch your machine-code program onto cards, as numbers, and then write another program that would read the cards into memory. So my program would be entered as a sequence of cards:

```
0 4107
1 4200
2 4805
```

and so on.

This is better than flipping switches. But it would be even better to have some mnemonics.

```
0 CLR  07   ; This is where we store the sum.
1 RD
2 JIZ  05   ; Will stop and print if 0 typed.
3 ADD  07
4 GO   01
5 PRT  07
6 STOP
7 DATA 3458
```

Programmers soon developed the techniques to read and decode such mnemonic commands. This type of notation is called *assembler* or *assembly code* and the program that translates it is also called an assembler.

This still leaves the programmer worrying about where to put each instruction. He has to allocate all the addresses manually. And if he wanted (e.g.) to insert another instruction before the `PRT`, he'd have to change the 5 to a 6, the 07 to an 08, the 6 to a 7, .... So the next stage in evolution was to have a cleverer assembler that could accept names for *labels* and *variables*, allocating these to addresses by itself.

```
        DATA I
        INITIAL I 3458

        PROGRAM START 0
        CLR  I     ; This is where we store the sum.
NEXT:   RD
        JIZ  OUT   ; Will stop and print if 0 typed.
        ADD  I
```

```
        GO   NEXT
OUT:    PRT  I
        STOP
```

But that still means you have to break down your calculations into individual instructions. So the next step was to design a program called a *compiler* which could do this for you. This was the start of *high-level languages* like Fortran (one of the first) and Pascal. This is what how the program might look if written in such a language.

```
        integer N, Sum;
        Sum := 0;
Next:   read(N);
        if N=0 then goto Out;
        Sum := Sum + N;
        goto Next;
Out:    print(Sum);
```

# 12 AI languages

With the sort of hardware I've described, it's easy to manipulate numbers. Consequently, computer scientists rapidly worked out how to implement high-level languages whose only concern was numbers, vectors, matrices, etc. It's harder to see how you would implement more "symbolic" programs such as a word processor or even a production system.

So in the early days of AI, the programmer had to work out how to encode every item of data — be it a sentence, or a chess-board, or a logical statement — as numbers, and every program as a program that operated on numbers. For example, a hypothetical chess program:

```
integer From, To;
integer array Board [ 1:64 ];

if Board[From] = 1 and
   Board[To] /= 0 and
   ( To=From+7 or To=From+9 )
then
   Board[To] := 1;
   Board[from] := 0;
```

The board is represented as a 1-d array of locations 1 through 64. The piece that's just moved is at location `From`. Each position `P` on the board is associated with a number `Board[P]` representing what's there. It's 0 if the position is empty; 1 for a paawn; 2 for a bishop; etc.

The task of this program fragment is to work out whether the piece being moved is jumping onto an occupied square, and if so whether it can capture anything. The location of the target position is `To`. So the program fragment means "if the piece being moved (at `From`) is a pawn (`1`), and the target position (`Board[To]`) is not empty (`\= 0`), and it's diagonally ahead of the piece, then move the pawn to the target, and empty the square it was on". In practice, you'd need to record the capture as well.

Manipulating this kind of encoding is tedious and error-prone. So later languages took over more of the burden:

```
if CurrentMove.Piece = pawn and
   IsOccupied( CurrentMove.To ) and
```

```
        CurrentMove.Direction = diagonal
    then
        Board[CurrentMove.From] := empty;
        Board[CurrentMove.To] := CurrentMove.Piece;
```

## 13 List processing and Lisp

In AI, we often need to represent sequences: of words in sentences; of actions in plans; of sequences of moves in chess; .... This can be done using fixed-length arrays of the type described above, but it's not at all convenient to do so. So with the growth of AI and developments in symbol-processing computing went research into languages that let you manipulate variable-length sequences or *lists*. One of their objectives was that you should not have to worry about how to allocate storage for new items in the list, or how to de-allocate storage for old items no longer needed (something which you still have to do in Pascal and C).

There were some early attempts at this that looked like assembler with lists built in. The first general-purpose language that let you handle lists in a natural way was Lisp, developed by John McCarthy. Lisp is still one of the main AI programming languages.

## 14 Logic programming

The idea that one expresses a program in logic. The most familiar logic programming language is Prolog — see my notes for the expert systems lecture. Here's an example (somewhat simplified, so not quite correct Prolog). Lisp is one of the two main AI languages; Prolog is the other. In this example, `:-` means "if" and `,` means "and".

```
replace(Board,NewBoard) :-
    at(Board,From,Piece), is(Piece,pawn),
        %  if there's a pawn at From
    at(Board,To,_),
        %  and any piece at To
    one_diagonal_ahead(From,To),
        %  one diagonal forward
    copy(Board,NewBoard),
        %  then the new state of the board is like Board
    make_empty(NewBoard,From),
        %  but with From empty
    move(NewBoard,Piece,From,To).
        %  and the pawn at To.
```

## 15 The trend to declarative languages

Early programming languages were very *procedural*. You had to tell the computer *how* to carry out your program, in minute detail. As the examples show, the trend has been towards languages where you specify *what* to do, but not how. Such languages are said to be more *declarative*.

## 16 Interpretation versus compilation

Two different ways to implement programming languages. What I've described so far is compilation.

- *Compilation*. A program called a compiler reads your program and translates it into machine code. Then the computer obeys the machine code. Those who've learnt it may know that Pascal is implemented this way.

- *Interpretation*. A program called an interpreter looks at each line of your program in turn, works out what it means, obeys it, and then goes onto the next line.

An analogy: compilation is like translating a French knitting pattern into English, then reading the English. Interpretation is like using a French dictionary on each word in turn as you knit. If there's a loop in the knitting pattern, then the latter method means you will end up retranslating the same word over and over again.

Compilation is harder; but the result is more efficient. Also, the internal representation of compiled programs loses the original structure, so you can't print it, etc. It has been suggested that experts somehow "compile" explicit knowledge into implicit: the information's still there, more efficient, but now unverbalisable.

## 17 Virtual machines and levels of description

You could solder together a machine whose electronics obeyed Basic directly. Or you could build a mechanical system that did. Or you could compile Basic on a VAX, or PC,...; or you could interpret it. At one level, all have the same behaviour: they are *virtual machines* for Basic. Something is a virtual machine for X if it behaves like X, but that behaviour is actually implemented by software — by programming lower down — rather than by hardware. So Newell would have said in 1972 that the brain's neural hardware implemented a production-system virtual machine.

To reduce the amount of complexity they have to think about, engineers and programmers often implement complex systems in layers. E.g.

- A chess-playing system, written in a "chess language" whose elementary operations include "castle", "test for checkmate", "promote pawn".

- A compiler which converts chess-language to Lisp, whose elementary operations include "create new list", "test for element in list", "copy list".

- An compiler which converts Lisp to machine-code.

- The machine-code is run by a CPU which is implemented by connecting logic blocks together.

- Which are built up from elementary logic gates such as NAND.

- Which are built from transistors and other components.

So we have a hierarchy of virtual machines or functional architectures. See Chapter X of Hofstadter's *Gödel, Escher, Bach* (Penguin 1981: PSY KH:H 67) for further discussion.

Can we analyse biological systems this way …?