Computer circles

Thursday 26 February, 2004 6pm
Harold Thimbleby, Gresham Professor of Geometry

Computer languages let us tell computers what to do. And we use computer languages to tell computers how to understand the languages we use to tell them what to do. It's possible to use the same language for everything, and we end up with fascinating circular explanations of what computers do. And while it all looks remarkably neat, and it is, we may end up hiding big problems and making our programs very unreliable! Computer viruses and Trojans are one possible problem (as I talked about in my Gresham lecture on 21 March, 2002), but other problems are just getting in a mess by mistake.

Today's Gresham Lecture explores these circular ideas by looking at the programming language called LISP. LISP was designed in the early 1960s, so it seems very old by today's computing standards, but being old, it is one of the simplest programming languages. However, LISP is very versatile and it is still widely used. In particular, LISP is ideal for talking about itself and getting into 'computer circles' — as we shall see. I hope this lecture will communicate the idea that circularity in programming is not just a very useful idea, but also one that is addictively and endlessly fascinating.

These lecture notes go into more detail than the lecture itself, as well as providing good references you can follow up. The LISP system we are using is available on the internet if you get excited and want to play with it.

Introduction

You will be familiar with many sorts of computer programs, from word processors to spreadsheets, or perhaps you think about the programs in your cars, which are programs that control engines or brakes. To get this range of behaviour from computers, the computers are programmed: they follow instructions, and have different instructions for different purposes. These instructions are readable by humans (indeed, they are written by humans): the instructions are written in a language. Just as when we talk to somebody who is French, we talk French so they understand, so we talk to computers in computer languages, programming languages.

To make programming languages work, we need programs that understand programs: so-called interpreters (or compilers). That makes sense until you wonder how the interpreters got written in the first place. An interpreter must obviously follow instructions itself to tell it what to do, so there is or was an interpreter for it too (unless the programmer went to the extreme, tedious and error prone process of writing the program in binary). So how is any interpreter written without an 'infinite regress'? Curiously, they are very often written in the same language as they interpret! For example, a Java system is written in Java, and a LISP system is written in LISP.

This is rather like the fact that people who speak French usually learn it from people who speak French. If you speak English in French company, you need an interpreter who can translate your English to French. French is probably the ideal language for teaching French. In much the same way Java is the ideal language for writing Java interpreters.

Quite likely: you speak the same language as your parents, but if we go back in time over the past generations, going back a century or more ago, the language your ancestors spoke would probably have been very different from what you are speaking today.

Similarly, Java interpreters (programs that 'speak' Java) today are written in Java, but if we go far enough back through their ancestry they would have been written in earlier languages, like C++; and in turn the 'ancestor' C++ before that was written in C.

It turns out that as soon as you have written an interpreter it is very convenient to keep the language of the interpreter the same as the language it defines. This is rather like the fact that my conversations with my parents will be easiest if we speak exactly the same language, regardless of what language or dialect my great grandparents spoke. Or once we have learnt a little French, it will be easiest to teach us the rest — to become fluent — by being in French immersion.

What's the problem?

The story above makes as much sense if your English ancestors had moved to a remote village:

you would speak the same language as your parents. But it's possible that over the generations some changes or even faults crept into your tongue. Now, you are speaking a dialect. Your parents are speaking the same dialect as you, so you cannot tell you are not speaking English.

What would happen if somehow in your dialect, "while" meant "until"? You might see a road sing "Wait while barriers are down" that should stop you crossing a railway line; but you'd understand it to mean, wait until the barriers are down and then you can cross. You might get hit by a train. Or another case: Shakespeare's word *anon* now means *sometime*, a much more leisurely interpretation than it has in his plays when he wrote centuries ago.

In computer languages we have exactly the same problems. If we are using the same language to define the language we are using, there may be subtle errors in it that we cannot detect — except when things go wrong.

Forty years of history

Lisp is one of the earliest universal programming languages, devised by John McCarthy in the 1960s. It is still widely in use, mostly in its two major dialects, called Scheme and Common LISP. The 1965 edition of the book claims LISP has a "perhaps unique history": the LISP system is a LISP program: exactly the sort of computer circle this lecture is about.

I first wrote a LISP system when I was at school in the 1970s, working from John McCarthy's book on LISP (see references). I wrote another LISP when I was a student at university. Later, I read Peter Henderson's excellent book (see references) and I wrote another LISP. I've written yet another for this Gresham lecture. I wrote my various versions of LISP in other programming languages: BCPL, C and Java.

The LISP we will use in the lecture is written in a combination of Java and LISP itself. The part written in Java provides a very basic LISP, which is then it is extended in LISP itself to provide the full LISP that we will use. Almost all of this lecture is about what we can do as we use LISP to extend and change itself.

John McCarthy, 1978

My LISP is available on the web together with a full definition of the dialect I devised, as well as some sample programs; please see my web site at `http://www.uclic.ucl.ac.uk/harold/lisp` My LISP runs on MacOSX directly. Of course, the core code is written in Java and the core LISP can also be used on other machines if you want to port the user interface.

These lecture notes finish off with a description of how my LISP was built. You *might* have thought that programming LISP in LISP was a sort of game and that the sorts of issues we raise in this lecture are special and 'not real programming.' In fact, my LISP is just a straightforward program, but — like everything else — it is implemented on top of all sorts of language interpreters. The issues we talk about in this lecture can appear anywhere in a typical computer system. Think of using a spreadsheet or a word processor — they are all programmable in languages not far from LISP, and some of them are programmable in several languages, as well as the ones they are implemented in. One reason computer viruses are so prevalent is that email programs have programming languages embedded in them; once a virus has got to running a program of its choosing it can do what it likes. These programming languages are typically based on Basic, rather than LISP, but the principles are the same (although Basic isn't a good language to write interpreters in so it is on top of the food chain, as it were).

LISP as a language

LISP is a programming language concerned with *lists*. LISP itself stands for **LIS**t **P**rocessing. There are lots of good descriptions of the language (see the references); I'll only give a very quick overview here, so that the notation doesn't seem too alien. It is much simpler than it looks, though for most people it is an unfamiliar way of working.

A list is anything written between brackets, and the elements of a list are separated by blank space. Here are some lists:

```
()                          ; an empty list, also written ∅
(1)                         ; a list of one element, namely 1
(a b c)                     ; a list of three elements, a, b and c
```

```
((a) (a a) (a (a) a 17))    ; a list of three lists, including other lists
```
To help read our examples, anything written after a semicolon is a comment and not part of the LISP program. I used that convention above, so in fact the *whole* lines are acceptable LISP, not just the lists themselves.

LISP is a flexible language because we can use lists to write down structured knowledge like this: `(family Thimbleby (married (father Harold) (mother Prue)) (son Will Sam Isaac) (daughter Jemima))`. Or LISP can do mathematical expressions like `(= (+ (* a (square x)) (* b x) c) 0)` meaning $ax^2+bx+c=0$.

It is a standard joke that LISP *really* stands for **L**ots of **I**rritating **S**illy **P**arentheses.

What does LISP do?

When LISP looks at a list, it looks at the first element and treats it as a 'what to do,' a function, which it then applies to the remaining elements of the list. This idea might be most familiar with conventional functions like sin or tan. Instead of writing sin 30 as you would normally do, in LISP you write `(sin 30)`, and expect LISP to 'do' `sin` to the `30`. In LISP *everything* is consistently written this way, even conventional things like addition which are not normally written first. We usually write 2+3 not `(+ 2 3)` as required by LISP, which is evaluated by LISP to `5`. Similarly `(* 2 3)` is `6`, because `*` in LISP means multiplying. A more interesting case is `(+ (* 2 3 4) 1)` which is `25` — it's just another way, the LISP way, of writing 2×3×4+1.

LISP has a wide range of functions it knows about. For example, the function `if` lets LISP make decisions. Thus we can write `(if 'true 1 0)` to be `1` and `(if ∅ 1 0)` to be `0`. More usefully (for some purposes!) we can write`(if (< x 0) –1 (> x 0) 1 0)` and it will have the value `–1` if x is negative (notice how `(< x 0)` means what we normally write as *x*<0), `1` if x is positive, and zero otherwise.

Typically, a list like `(family Thimbleby (married (father Harold) (mother Prue)) (son Will Sam Isaac) (daughter Jemima))` won't mean anything to LISP unless we define what we want it to mean. And that's easy.

Example LISP programs

Here is a function that tells us the length of a list. Later examples will be much more interesting, but this is a good place to start.
```
(new (length s)
    (if
        (= s ∅) 0 ; an empty list, written ∅, has length zero
        ; otherwise its length is one plus the length of the rest of it
        (+ 1 (length (cdr s)))
    )
)
```
So now we can write `(length '(1 stuff x 3))` and get `4`.

Or suppose we wanted to define × (i.e., multiplication) we can write:
```
(new (× a b) (* a b))
```
Which just says doing × to two things is to be worked out by doing * to them. If we wanted to, we could define it from scratch, as follows:
```
(new (× a b)
    (if
        (= a 0) 0
        (< a 0) (– 0 (× (– 0 a) b))
        (+ (× (– a 1) b) b)
    )
)
```
which defines multiplication in terms of repeated addition, without relying on any other definition of multiplication. Thus 3×*b* is worked out by doing *b*+*b*+*b*+0. It works as follows: to multiply *a*×*b*, if *a*=0, then *a*×*b* is zero; if *a* is negative, then *a*×*b* is minus (–*a*)×*b*, otherwise *a*×*b* is *b*+(*a*–1)×*b*, and on this sum we use the rules all over again. Notice how *a*×*b* is defined in terms of (|*a*|–1)×*b*, so eventually we hit 0×*b* which is easy.

So it is easy to define a multiplication function, even if we didn't have one to start with.

Did you notice that these simple programs are themselves lists? LISP programs are lists, so LISP can process LISP! In other words, LISP is ideal for talking about LISP itself.

Eliza as another language

My LISP is 'complete,' meaning anything that can be programmed can be programmed in my LISP (though the interactive side of things is a bit weak). English is another language, so let's write a LISP program to recognise English. I take my inspiration here from Weizenbaum's classic program Eliza (see references).

In my version of LISP, we can read other programs in, rather than having to write them out in full:

```
(open-window '"http://www.uclic.ucl.ac.uk/harold/lisp/resources/Eliza")
```

will download from the internet my version of Eliza.

So, I ask Eliza

```
(Eliza '(what can I write in my lecture notes))
```

and she replies:

```
(Are you really interested in this lecture ?)
```

Here we are using `Eliza` as a standard LISP function, because that is how we defined it to work. But in LISP we can easily go on to the next stage: what is interesting is we can change our whole language to become Eliza. There is a LISP function that makes LISP apply any function to anything you say to LISP. By writing `(language Eliza)` our LISP effectively turns into Eliza: everything that used to be read as LISP is now read by Eliza directly.

After changing the language from LISP to Eliza, we can just write English (albeit in list form), without having to ask for Eliza explicitly. I ask,

```
(what other examples should I give in my notes)
```

and Eliza answers

```
(Please can I have your notes ?)
```

I don't think Eliza understands these notes...

```
(would you be able to understand my notes)
```

and the conversation, such as it is, continues between us:

```
(What else comes to mind when you think about your notes ?)
(talking about more examples)
(Please continue)
```

The language is not LISP anymore, but a program LISP is running, namely `Eliza`. Pretty deep stuff for a short program, isn't it?

There's a wonderful true story about Eliza being mistaken for a real person; but you'll have to wait and see how it works in the lecture.

Eliza speaks bad English, and Eliza's English is so bad she is no more than a curiosity — though she played an important part in computing research when Weizenbaum invented her and saw how people fell for her wiles.

A different sort of maths

My LISP evaluates sums if they have numbers in them, so `(+ 2 3)` is 5. This is pretty conventional programming stuff. Can we extend LISP to be more useful?

In most programming languages $a+b$ or `(+ a b)` is not allowed unless `a` and `b` are both known numbers. Yet in normal maths, $a+b$ does make sense; for instance we know $a+0$ is always a, whatever a is. We can extend LISP to do this sort of symbolic maths easily.

(When I first wrote the code, LISP read it all and said `(Go on)`. I was still running Eliza! I had to reboot to get back to the original LISP.) We download the symbolic maths package I prepared earlier, that I have already put on the internet:

```
(open-window '"http://www.uclic.ucl.ac.uk/harold/lisp/
resources/symbolicmaths")
```

This gets the simple code to do symbolic maths, including being able to differentiate. Now if we change our LISP language using `(language symbolicmaths)` the whole approach to maths changes. Before `(+ a a)` would either result in an error (if a was undefined) or LISP would do the sum — if a was defined as 2, say, then `(+ a a)` would be 4 of course.

But now if we now try `(+ a a)` we will get `(* 2 a)`, even if we don't know what `a` is, rather than a useless error. If we happen to know `a`, then we would get the numerical result as before. Other examples we can work out include `(+ a 0)`, `(* a 1)` and more complex examples along the lines of `(* (- (+ 2 a) (+ a (+ 1 1))) (+ x y))`, which is zero.

The maths extension also allows us to differentiate. Normally it differentiates against x: if we try, say, `(D (* u v))` we will get results like the multiplication rule for differentiation: in this case, `(+ (* du/dx v) (* dv/dx u))`. We can differentiate against other variables too: `(D t (*`

4

`2 t))` gets us `2`, for example, because it is calculating $d/dt\ 2t$=2.

Inventing and defining new languages is a bottomless pit. We now turn to defining variations of LISP itself.

LISP in LISP

Now that we've seen we can write programs in LISP and effectively change the language we are using, let's try and write a program in LISP to run LISP — we will 'change' the language we are using to LISP itself. Thus we create a 'computer circle' where we have a new language that is LISP, which we've defined in LISP. We can do anything we like, including adding new features to the language if we wish.

Functions to define LISP

The basic structure of doing LISP in LISP is to have a function `eval` that looks at the first parameter of anything it should do, and then works out what to do with the rest of the list. Here is one simple version, where — for brevity — we only pay attention to arithmetic operators, like addition and multiplication, rather than the whole range of LISP features:

```
(new (eval e)
    (if
        (number e) e
        (= (car e) 'add) (+ (eval (cadr e)) (eval (caddr e)))
        (= (car e) 'subtract) (- (eval (cadr e)) (eval (caddr e)))
        (= (car e) 'times) (* (eval (cadr e)) (eval (caddr e)))
        (= (car e) 'divide) (/ (eval (cadr e)) (eval (caddr e)))
        (list 'error e)
    )
)
```

This defines a language where we can write arithmetic, as in ordinary LISP, except to avoid confusion (for us humans) it uses the names `add`, `subtract`, `times` and `divide` instead of LISP's built in `+`, `-`, and so on. (Notice the error warning, which is intended to cover the bits of LISP we haven't yet defined.) When `eval` reads 'plus' it does the basic LISP `+` on whatever `plus` is applied to, so circularly it does the same thing. Of course we could make the names in the two languages the same, which might have been a bit too confusing to read the first time:

```
(new (eval e)
    (if
        (number e) e
        (= (car e) '+) (+ (eval (cadr e)) (eval (caddr e)))
        (= (car e) '-) (- (eval (cadr e)) (eval (caddr e)))
        (= (car e) '*) (* (eval (cadr e)) (eval (caddr e)))
        (= (car e) '/) (/ (eval (cadr e)) (eval (caddr e)))
        (list 'error e)
    )
)
```

Notice the four lines where the arithmetic operators are repeated: to do `+`, do `+` *etc*, is all it is saying. If we define a function `(operator (f a b))` to return the operator `f` (if `f` is indeed an operator) then we can write something that looks simpler, and more obviously does everything the same way:

```
(new (eval e)
    (if
        (number e) e
        (operator e) ((operator e) (eval (cadr e)) (eval (caddr e)))
        (list 'error e)
    )
)
```

This is the same as the earlier program, but much briefer. It is now a LISP function that does arithmetic — but it doesn't mention addition, subtraction or anything else! *What does it do*?

In fact, we can define `operator` to work with all sorts of functions, and we start to get a pretty good LISP without doing any more work. If we use another feature, `map`, we can get the code to work very nicely for any function, not just the basic binary functions we have already considered:

```
(operator e)
    (apply (operator e) (map (λ (t) (eval t)) (cdr e)))
```

This form is used in the example code on the internet supporting this lecture. This single line

of LISP defines our new LISP to have the functions `list`, `apply`, `cons`, `print` and many more too (and we don't need to say so). So we can now extend the computer circles to include our new language, and with a bit more effort, even write another LISP system in *it* too, and do it again and again!

An advanced example

The code on the internet illustrates this operator technique for the simple case of changing the new language to work out factorials — quite a change, being *two* generations of languages deep. We start by defining factorials in the usual way, but for fun using the 'paradoxical' combinator, usually called Y.

Here is how we would evaluate 8! in the basic LISP (don't worry if it looks like Greek):

```
((λ (f n) (f f n))
 (λ (f n) (if (= n 0) 1 (* n (f f (– n 1))))) 8) ⇒ 40320
```

So 8! is 40320. We can evaluate the same 8! expression in the new LISP language we've just defined:

```
(lisp '((λ (f n) (f f n))
 (λ (f n) (if (= n 0) 1 (* n (f f (– n 1))))) 8)) ⇒ 40320
```

That works straight forwardly. Now we circularly *change* the LISP system to our new LISP:

```
(language lisp)
```

…and then run the same thing again, and now we do not have to say `(lisp …)`; as it now runs directly:

```
((λ (f n) (f f n))
 (λ (f n) (if (= n 0) 1 (* n (f f (– n 1))))) 8) ⇒ 40320
```

It seems to work the same way, as if we'd done nothing. In fact, we are now running our own new version of LISP. We can confirm that we are running in the new LISP, since (by oversight?) the function `(lisp)` itself is not defined any more in the new language.

What have we done?

We have defined a new LISP on top of the original LISP. Our new LISP can do the basic things LISP can do: it can do arithmetic, comparisons, print and so forth. But our definition of the new LISP does not mention any of these features. (The LISP definition that will be used in the lecture can do more than we show here, but it takes up too much space in these notes to show it in all its glory.) Literally, the new definition inherits the original definitions (as well as any other new features we wish to introduce).

It is rather like growing up in a family that calls a certain sort of meat at dinner 'cow.' So long as everyone calls it cow, everybody knows what is meant even though it is defined more-or-less by cow=cow (or, more specifically, by my cow=parent's cow, and since this is true for everything, it's just my *x*=parent's *x*). But is it *really* cow? We manage to use English most of the time without defining much of it. If my family called lamb cow, and beef lamb, they perhaps would never know they were wrong. A more philosophical example is my use of the word 'red' may refer to a colour I see in my mind which is different from the colour you see in your mind when you think of 'red.' In this example, there really is no way to find out if we mean the same things or not.

Making new features disappear

Let's add a completely new function to our language that is not already in LISP. We might like 'absolute value.' Normally, we would define it something like this:

```
(new (abs n) (if (< n 0) (– 0 n) n))
```

Having defined it, we can introduce `abs` into our list of operators, and now we have a LISP extended to have a built-in `abs` function. But if in *that* LISP we define a new LISP, we will get the `abs` function *without* needing the definition above: because now it is built into the language.

In fact, we can extend the language in all sorts of ways, yet our definition of it looks exactly the same! The function `abs` — or whatever we defined — has become built in and disappeared not just from view, but also from scrutiny.

It could be worrying if our original definition of `abs` was wrong; it is now invisible. It would be even worse if our definition of `abs` got maliciously corrupted. There is now no way to tell, except *after* things go wrong — by which time it might be too late. This is fertile ground for hackers.

Surprises

We have shown how some parts of a language can disappear when we define a language in a circular way. It might surprise you to know that we got this far without even mentioning some things LISP does — which we used but didn't need to talk about. For example, exactly what LISP does with numbers has nowhere been specified, and yet we have a LISP that can do arithmetic — apparently correctly. We've managed that without even thinking about what, say, division by zero might mean. Put another way: we wrote circular programs in LISP to show how it can be done, and we managed to create programs that used numbers and other features without even needing to define how numbers work in the new language definitions.

In fact, I wrote my basic LISP in Java, and I used Java's arithmetic. So if you looked at the original program that defines my LISP, you still would not know how it did arithmetic. You'd have to look up the definition of Java.

A more sophisticated example is that our LISP does "garbage collection" automatically. And so would any new LISP we defined in terms of it. It gets garbage collection *without* asking for it. Since I wrote the core of my LISP in Java, and Java has garbage collection, I got a LISP system with garbage collection for free. Even my program in Java to implement LISP doesn't explicitly provide garbage collection.

The structure of my LISP

I thought it would be interesting to finish by showing the structure of the LISP I developed for this lecture. The structure shows how the language we use in the lecture depends on itself and other languages.

The structure is rather like a linguistic genealogy for my human family: I speak English, my parents speak English (but I know a few words about, say, programming, that my parents do not — so in my generation I speak an 'extended' English). A few generations ago we spoke French (maybe), and before that something else simpler … and thousands of years ago we spoke in some primitive way probably by hitting each other. Much like if you go far enough back in the computer language genealogy you end up flicking bits.

The simplest way to explain how my LISP works is, first, that it is written in Java. The LISP system consists of an infrastructure that interprets a very basic LISP: in Java there is a LISP compiler and a runtime system, also written in Java. (The LISP runtime system interprets a special language called SECD code.) When the LISP infrastructure starts running, it first runs a LISP program called the bootstrap, for it now picks itself up by its bootstraps. The bootstrap is a LISP program that extends the basic infrastructure to the full LISP we used in the lecture. So, there are many layers of interpreters: LISP, Java, SECD, Java, Java virtual machine, hardware, and all sorts of layers from the operating system to support other important features (such as the user interface), which in turn are written in other languages.

This program…	…is interpreted by this program
Programs written in the lecture, like Eliza.	The LISP system used in the lecture.
A LISP bootstrap program that extends a basic LISP. The bootstrap is written in basic LISP, and defines all LISP primitives as proper functions. It creates a more complete set of primitives. For instance, the bootstrap program defines all these functions <, >, =, ≠, ≥, ≤.	The LISP infrastructure, sufficient to run the bootstrap. Primitives are special forms (i.e., not proper LISP functions), and only a few are available. For instance the infrastructure defines only = and ≤.
The compiler for the LISP infrastructure.	The LISP compiler is written in Java and compiles to Peter Landin's SECD code.
A Java interpreter for SECD code.	The SECD interpreter is written in Java.
Java is compiled by a Java compiler.	The Java compiler is written in Java.
Any Java system.	Any Java program runs on the Java Virtual Machine.
The Java Virtual Machine is written in machine code.	The machine code runs on the hardware, but relies on support (e.g., for drawing windows) written in other languages, such as Objective C.

Having got a LISP compiler (written in Java) to work, we can now run LISP programs. If I wanted to, I could now write a LISP compiler in LISP to do the job of the current LISP compiler, which is in Java. If I did so, I could then throw away the LISP compiler written in Java. I would then have a LISP system written in LISP and Java, but the definition of the LISP would be all

written in LISP, with no non-LISP definition!

The previous version of LISP I implemented (which I wrote in C rather than Java), I did just this. Interestingly, my LISP compiler ran faster than the original LISP compiler written in C. That's because my LISP then (as now) had numerous optimizations (such as tail recursion optimization) that the C compiler was unable to provide. So, LISP-in-LISP was *faster* than LISP-in-C.

Conclusions

Usually, a circular argument has got holes in it. A circular definition in a dictionary tells you practically nothing. In this lecture we've seen how programming languages can be defined in circular ways, and that this is an effective and very common way of defining programming languages. In the lecture, we used LISP and showed how various languages could be defined in it: new ones for maths, simple English, and even new versions of LISP itself.

Everything running on computers depends on knowing what programming languages do. The computer must be instructed correctly using some language, and so circularity is a central concept. Circularity used in this way is often called *metacircularity*; it isn't just circular, but gives us powerful leverage. Intriguing as it is, circularity can introduce very subtle and not so subtle problems: as more and more becomes implicit, the more we benefit from its power, yet more things get harder to check and the more unreliable they may become.

Further reading

1   H. Abelson and G. J. Sussman with J. Sussman, *Structure and Interpretation of Computer Programs*, MIT Press, 2nd edition, 1996. This is the best introduction to LISP and programming by a long way; it uses the modern LISP dialect Scheme.

2   J. R. Allen, *Anatomy of LISP*, McGraw-Hill, 1978. A somewhat dated book, but about the basic implementation of LISP.

3   G. Güzeldere and S. Franchi, "Dialogues with Colorful Personalities of Early AI," *Stanford Humanities Review*, **4**(2). Updated 1995, at http://www.stanford.edu/group/SHR/4-2/text/dialogues.html. Example Eliza dialogs, some of which are used in the lecture.P.

4   Henderson, *Functional Programming: Application and Implementation*, Prentice Hall, 1980. A really clear book, which defines the SECD machine on which my LISP used in this lecture is closely based.

5   D. E. Knuth, *Selected Papers on Computer Languages*, CSLI, 2003. Provides an excellent overview of early programming languages, and the role of FORTRAN.

6   J. McCarthy, "Recursive functions of symbolic expressions and their computation by machine (Part I)," *Communications of the ACM*, 1960. Now available at http://www-formal.stanford.edu/jmc/recursive.html, this is essentially the original paper on LISP.

7   J. McCarthy, P. W. Abrahams, D. J. Edwards, T. P. Hart & M. I. Levin, *The LISP 1.5 Programmer's Manual*, MIT Press, 1965. One of the original LISP definitions. My own edition of this book cost £1.40, so you can see how even inflation has moved on, let alone programming languages!

8   G. L. Steele, Jr. & R. P. Gabriel, "The Evolution of LISP," *ACM SIGPLAN Notices*, **28**(3), pp231–270, 1993. An excellent overview of all the LISP dialects and evolution of the language.

9   L. Sterling & E. Shapiro, *The Art of Prolog*, MIT Press, 1986. Prolog is another programming language, which has a much shorter interpreter than LISP. Some might argue this means Prolog is a more powerful language.

10   J. Weizenbaum, "ELIZA — A Computer Program for the Study of Natural Language Communication between Man and Machine," *Communications of the ACM*, **9**(1), pp36–45, 1966. The original paper on ELIZA.

Notes to the knowledgeable These lecture notes don't distinguish between interpreters and compilers. A full definition of the LISP used here is on the web (see text) and it defines the static lexical binding semantics, *etc*.