Visible Lisp

Language implementation for the Raspberry Pi and other small computers

Arthur Norman

Contents

Chapter 1

The purpose of this book

This book is for people who like to understand how things work. It is about getting to grips with computers and understanding how programming languages really work by seeing not just what they do but how they do it. The main language used here is Lisp. This is a simple enough language that a very large part of its internal workings can be explained – but it is general enough that learning about how Lisp works can give insight into the structure of almost all other programming systems. The aim is to make the Lisp that is explained here Visible, and so perhaps not surprisingly the version that is to be discussed is known as Visible Standard Lisp, abbreviated as `vsl`.

The term "visible" as applied here is intended to indicate that all of its the inner workings are available for inspection. Lisp is one of the oldest computer languages that is still in use (albeit in niche applications). It it is also one that has been seriously influential in the development of more modern languages, and so understanding it can give useful insight into how we have got to where we are. The key application area that Lisp dominated for many years was Artificial Intelligence, and perhaps in particular the automation of logic, reasoning and symbolic mathematics. A second area where it has been important has been as a notation embedded within some other large program and used to describe customisation or scripting capabilities there. The vsl system discussed here could serve quite well in either of these broad areas.

Before explaining the organisation of the book here are a few taster fragments that show what can be done with Lisp. You will find a fuller explanation of each later on.

(1) Arithmetic on big numbers works, so you can type in

```
(expt 7 510)
```

and obtain the output

```
Value:
```

```
10000009377765355041159521080236291875500397545959064338020
91046611830071747256290251642675754113510381906538657851472
80048260730260174481888634961017465832153767083905055973115
17146314730511796799136227470799861707690686485640934223257
99898508099347892972049510888294887959312673650548919269730
26731194699108560193240351188856787007789047064042449190113
93827146105377053911155370062040100951494893263068969133321
6918712082217175249
```

Note how close to a power of ten this is, and so we have to a really good approximation $7^{510} = 10^{431}$. From this we deduce that the logarithm of 7 is very close to $431/510$. Checking this with a calculator reveals it has around 8 significant figures correct.

Of course big-number arithmetic has applications in encryption and in keeping track of the bank balances of the very very rich as well as for computing logarithms.

(2) One of the sample programs included here uses turtle graphics to draw snowflake patterns. The key directives are `turn` which changes the heading of the turtle by a given number of degrees, and `draw` which makes it march forward a given distance leaving a visible trail. The particular graphics capabilities shown here are not standardised across all versions of Lisp, but the simple line-drawing capbility used here is available with `vsl` and is fully explained later on.

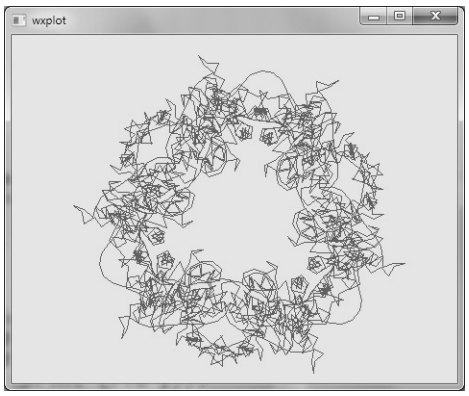

Figure 1.1: A snowflake pattern

The full code has a few lines that reduce turtle graphics to drawing instructions in ordinary cartesian coordinates, but the central part of the Lisp to draw the image shown here is just

```
(setq a 5 b 0)
(dotimes (i 2160)
   (draw 12.0)
   (setq a (plus a 7))
   (setq b (plus b a))
   (turn b))
```

and there is obviously scope for experimentation to see how changing the numbers used alters the image created.

(3) The above two examples only use short fragments of Lisp code. The final one here is based on use of many thousands of lines of Lisp to implement an algebra system capable of integrating, differentiating and performing all manner of other symbolic calculations. Here is a short transaction that evaluates an integral:

```
}
1: int(x*(log x)^4, x);


  2           4            3            2
 x *(2*log(x)  - 4*log(x)  + 6*log(x)  - 6*log(x) + 3)
-------------------------------------------------------
                            4
```

The first part of this book is a gentle introduction to Lisp and its use. There will be those who view the term "Lisp" as referring to some specific dialect. If that line were taken it would have been hard to avoid the use of Common Lisp[38]. However Common Lisp is a large and ambitious interpretation of the Lisp central ideas, and making its implementation visible would be unfeasible. So the version of Lisp used here preserves the key aspects of the language (for instance a notation that uses unspeakable numbers of parentheses!), but is styled after the Standard Lisp developed at the University of Utah[28], which allows its scale to me much more reasonable. A section later in the book provides some commentary about adapting everything here to match Common Lisp, and the associated sample programs are provided in both Standard and Common dialects.

The second section explains how the Lisp implementation used here works. As well as making the source code accessible as a basis for learning about Lisp and language implementation in greater generality it has comments about the implementation issues of some features not included in the code used here. As a result it amounts to a challenge to add some of those, or to rework the existing code following alternate fundamental choices. In this respect the aim is not to make this a book for passive readers, but a starting point for active engagement with Lisp code. It should make it possible both to write code in Lisp to do useful things, and also to adjust and extent the implementation of Lisp to add extra capabilities or tune it for particular needs. The source code of much of `vsl` is coded in the language C, and so as well as providing an introduction to Lisp this book can be viewed as a resource showing C in use. But those who are not fully fluent in C should still be able to follow the key aspects of how `vsl`'s implementation works by reading the quoted C code as if it were a sort of abstract pseudo-code and relying on explanations in the text to sort out any places where C syntax could seem obscure.

Then there are a collection of miniaturised Lisp applications. These are intended as starting points for more development rather that complete polished code,

but they illustrate the use of Lisp for a range of tasks – some frivolous and some serious.

Lisp has always tended to be the sort of language that a minority of programmers get intensely attached to. Perhaps by learning it and observing how simple it is, while at the same time how powerful it is, will let anybody reading this join that minority! Discovering its history, structure and use can give new insights into features of other programming languages, and in many cases make them seem either terribly limiting or dreadfully over-complicated.

Chapter 2

What is Lisp?

In order to understand a programming language you need to know something of its history, and of the context in which it has developed. So you need to be aware of other languages it has been in competition with, and trace influences. In the case of Lisp this leads to a need to make an amazingly broad survey of the entire history of computers, since Lisp was one of the very earliest programming languages and its influence since has been huge. Lisp was invented by John McCarthy in the late 1950's and was originally envisaged for use in Artificial Intelligence. Indeed the very term "Artificial Intelligence" had been coined by McCarthy at about the time that his first thoughts the led to Lisp were arising. While AI and related areas have continued to be important for the language, it has found a new core area as an embedded scripting language to support various large programs, including editors, computer aided design packages, musical notation systems and accounting software! Serious widely used applications (for instance the Yahoo Store) run in Lisp, but of course their users just get to see their behaviour not their Lispy internals.

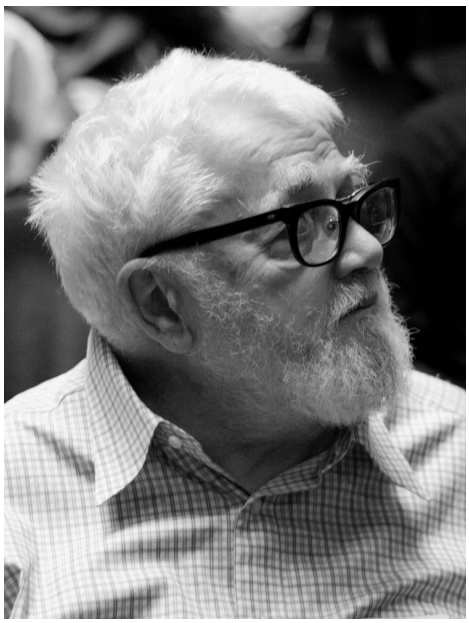

Figure 2.1: John McCarthy (photo courtesy null0 on flickr)

The Wikipedia entry for Lisp gives a list of other programming languages derived from or influences by Lisp: CLU, Dylan, Falcon, Forth, Haskell, Io, Ioke, JavaScript, Logo, Lua, Mathematica, MDL, ML, Nu, OPS5, Perl, Python, Qi,

Figure 2.2: and IBM 704 computer. This photo is from the Columbia University Archive.

Rebol, Racket, Ruby, Smalltalk and Tcl. Lisp enthusiasts will then wish to follow the references through to see what Lisp indirectly influenced. So for instance the languages that Smalltalk has had an impact on include Objective-C, Self, Java, PHP 5, Logtalk, Dylan, AppleScript, Lisaac, NewtonScript, Groovy, Scala, Perl 6, Common Lisp Object System, Fancy and Dart. Obviously many of these names will not be familiar, either because they are now defunct or because they are only used in specialist contexts, but there may be enough easily recognisable languages to show that the Lisp legacy has been truly broad.

The language design for Lisp dates from 1958 and built on ideas that John McCarthy had been working on over the previous couple of years. The first implementation was completed 1959. This original work was done on an IBM 704 computer. It could execute up to 40,000 instructions per second. At the time of their introduction IBM 704 computers had 4K (36-bit) words of memory – in other words 18 Kbytes of memory. By the time that model was withdrawn, and transistorised computers took over, the available memory option had grown to 32K words (144 Kbytes). Most such computers were rented (possibly for "under $50000 per month") but to buy one would have set you back several million (1950s) dollars. Inflation since the mid 1950s means that any prices from then need to be multiplied by about 8 to get a realistic figure for today.

If I compare the IBM 704 from 1957 with one of the cheapest options available

in 2012[1] today's machine gas not far short of 2000 times as much memory, runs perhaps 15000 times faster and the cost has gone down by a factor of perhaps half a million!

There are many variants on time-lines for the development of programming languages. Some people treat the "idea" as setting a date that establishes priority, some look at the initial reasonably complete implementation, while yet others view publication as the key milestone. But however you look at things you will find that FORTRAN came first, with Lisp and Algol almost in a tie for second place and COBOL third among languages that have in any sense lasted.

These early languages have all obviously had influences on developments that have followed. However there is scope for disagreement and interpretation about just where the links have been Strong enough to be worth noting. So what indicate here is my view, slanted by the fact that I am wishing to promote Lisp at present.

FORTRAN has as its linguistic successors FORTRAN IV, '66, '77, '90, '95 and now 200, 2003 and 2008. Each of those dates indicates a formal international standard, and the language has clearly remained vibrant and supported. However about the only language that can really be held to have been derived from or even dramatically influences by FORTRAN was PI/I (now no longer in use). It could perhaps be said that FORTRAN should take it share of the blame for BASIC. However even if the FORTRAN language has not proved influential its implementation has been. In particular there are two crucial legacies from it. The first is the concept of being able to compile bodies of code independently and form a library containing code that you can re-use with your next complete program. The second is the technology for seriously clever optimising compilers: ideas pioneered in FORTRAN form the basis of a large proportion of modern compilers.

Algol was originally the International Algorithmic Language. Algol 58 was soon succeeded by Algol 60, which might reasonably be characterised as the first "modern" programming language. Algol as such has fallen out of use, but there is a rich stream of languages that are unambiguously successors. In one direction Pascal took Algol and emphasised particular structures styles of programming. Pascal has Modula and Delphi among its more obvious successors. In a parallel set of developments CPL extended Algol with a view to becoming a universal language capable of supporting all areas of computation. The CPL project spawned BCPL (a severe simplification of CPL initially mainly for use in writing the CPL compiler), and this let via B to the major language C. C++ builds on and extends C, and Java can be seen as a reaction to C++, and yet beyond that C# and Scala are both reactions to Java. All in all Algol has been a truly major influence on program language syntax, and on the manner in which variables are declared and the scope within which they may be accessed.

[1]The Raspberry Pi model B

That leaves Lisp. As a language Lisp can be viewed as being something of a living fossil, inhabits niches and being used in slightly specialist areas. Well actually Lisp has always been something of a language for special purposes! With both Fortran and Algol there were substantial changes to syntax made between the original languages and the ones that are their current descendants. Lisp on the other hand still looks very much as it did. Various dialects and variants have arisen (and in many cases then vanished) but they all share a truly distinctive syntax where programs are held together with lots of parentheses. A modern Lisp is liable to be bigger and meaner and have many more teeth than the primordial Lisp 1.5, but the family resemblance will be strong. One reason for this is that Lisp has never had much special syntax for anything – it just uses its standard parenthesis-rich notation. This leads to huge flexibility in that adding new features will not conflict with existing notation. One way in which this has been described is attributed to Joel Moses and appears in the archives in a number of forms. There are also assertions that Moses denies having ever said it – but that does not that seriously undermine its essential truth:

> *A language such as APL is like a diamond. It is perfectly symmetric, and shines brightly. However, if you wish to add a new feature to the language, the symmetry is smashed, and the diamond cracks and shatters.*
>
> *Lisp, on the other hand, is a ball of mud. Essentially shapeless, you can easily add new extensions and ideas, and all you get is a larger ball of mud ready and able to accept more and more. Lisp is infinitely extensible: you can add new functions that have exactly the same importance to the system as the built-in commands, or even redefine the built-in commands. You can, if you feel that way inclined, redefine the whole syntax. Imagine the possibilities if C allowed you to redefine the while loop, or Fortran let you introduce exactly that form of DO loop that you required for your application.*

The above is supposed to be alluded to in Steele and Gabriel's paper on Lisp Evolution[37], but it is not visible in the copies I have located to date! The version quoted comes via Padget[33].

The key features that make Lisp what it is are

Primitive syntax: In Lisp pretty well the only syntax used is one where the name of a function and the sequence of its arguments are enclosed in parentheses. Thus $3 * x + 5$ is expressed as `(plus (times 3 x) 5)`. This scheme favours consistency over conciseness. When Lisp was first being invented it was imagined that a more conventional human-friendly notation would be provided, but the "all brackets" notation caught on and has remained basically unchanged.

Lists as data: The sorts of data you can work with in Lisp include numbers and strings, but also symbols and lists of items. Lists can be nested arbitrarily. List structures are used to model records, trees and pretty well any other sort of data you might find in a more modern language. This means that to get started with Lisp you only have to think about a few simple sorts of data. Alan Perlis[34] suggested that "It is better to have 100 functions operate on one data structure than 10 functions on 10 data structures" and Lisp follows this recommendation.

Code can be treated as data: It goes beyond coincidence that the syntax of Lisp uses parentheses and its notation for list data is the same: Lisp code can be regarded as Lisp data and vice versa. Thus the language makes it easy for programmers to think about code analysis or synthesis. Modern languages such as Java have introduces standard ways in which one piece of code can inspect another (calling if *reflection*): Lisp has always made that not just possible but natural and easy.

Types only matter at run-time: Lisp will complain if you try to add a number to a string, or to otherwise breach good taste as regards the use of its data-types. But this complaint only happens when you run your code. There is typically no checking made at the time you define functions or otherwise present your code to the system. Depending on your point of view this either gives you a lot of flexibility or represents a serious risk of allowing bugs to remain undetected. If you are using Lisp as the inner basis for the implementation of a new programming language the flexibility arguments can really win.

Mostly Functional: A complete Lisp program is created by defining a collection of little functions. A tradition and style in Lisp has grown up where often all of these really behave as functions in the mathematical sense of that word. They accept arguments and deliver a result, and that result only depends on the arguments not on any context or history. This style of programming has been found useful by those who wish to formalise things to the level of producing mathematical proofs of their code, but along with the use of recursion it turns out to lead to nice human-comprehensible ways of expressing the solution to many programming tasks.

Automatic storage management: Lisp programs create new lists, process them and eventually move on such that they do not need that data any more. It is fundamental to Lisp that the system takes care of recycling discarded data and hence the programmer never needs to worry about it. This is an

arrangement that has only rather recently found its way into other widely-used languages.

An extensible language: When you start up Lisp and define a collection of new functions they take their place alongside all the existing ones that were built into the system from the start. If you have a fragment of Lisp code such as `(obscure_function_name argument_1)` you are just using the standard Lisp notation for a function invocation. There is no distinction at all made as to whether `obscure_function_name` was a built-in function or one you had defined yourself. So as you define new functions you can often think of what you are doing as extending the language. If you need new capabilities from the operating system you can often merely implement a function in Lisp to access them, and again once it is there its status is just the same as that of all other Lisp functions. Compare this with other typical programming languages where for instance characters such as "+" and words such as "`while`" probably have rather special status.

There are some things that Lisp perhaps does not emphasise. Well on saying that it will perhaps be safer to say "Lisp as discussed here does not emphasise" since there can be particular Lisp implementations with all sorts of specialisms! The topics listed here could sometimes be represented as disadvantages or limitations of Lisp, and so in each case I will indicate the way in which a dedicated Lisp fanatic turns the point on its head to reveal a silver lining.

Human-readable syntax: There are some who believe that the Lisp notation is at best antiquated and could reasonably be described as barbaric. Valid Lisp code can end up looking really bad. For instance and as a very small example here is a factorial function written first in Lisp and them in the tidy language SML[30].

```
(de factorial (x) (cond ((equal x 0) 1) (t (times
x (factorial (difference x 1))))))
```

and then

```
fun factorial 0 = 1
  | factorial n = n * factorial(n-1);
```

The Lisp version is bulkier, it uses long-winded names for even basic arithmetic operations and ensuring that you get exactly the right number of close brackets (six in this case) at the end could confuse almost anybody.

There are several responses that a Lisp fanatic will give to this. The first is that Lisp code should be properly laid out, and that a competent text editor

can help with both indentation and bracket matching so that neither are a real burden on any serious user. There will be shorter functions to cover some common cases, so the Lisp code should have been shown as

```
(de factorial (x)
   (cond
      ((zerop x) 1)
      (t (times x (factorial (sub1 x))))))
```

This is still perhaps bulky, buts its structure is now easy to see. Indentation and bracket counting are so easy to mechanise that it is silly to fuss about them – but anybody who can not cope with the bracketed notation could layer a parser on top of Lisp so that their favourite syntax gets mapped onto Lisp's internal form. Almost any compiler will have an internal form of parse-trees that is essentially very much like native Lisp code anyway so this is not a big deal. As a concrete example of a case where this approach has been taken, the Reduce algebra system[24] has its own language, known as `rlisp`, used in exactly this way – here is our factorial example written in `rlisp`.

```
symbolic procedure factorial x;
   if x = 0 then 1
   else x * factorial(x-1);
```

Absolute code portability: There are many programming languages where if you write a program you can expect (or at least hope!) it will run everywhere without modification. Java is a fine example of a language where this virtue was made a key marketing point. With Lisp there is an international standard – Common Lisp[39] – but if you fetch and install an implementation of Common Lisp you will find that at some stage you will need to find out about the limitations and the extensions applicable to the version you selected. The earlier and perhaps more primitive Standard Lisp[25][28] are the basis for the `vsl`system used here. Lisp is used as an embedded extension language for the widely-used `emacs` editor, and a web-search will easily reveal a number of other versions with subtly or crudely differing capabilities.

The Lisp fanatic response to this is that the exact names that a Lisp system uses for some operation is very much a secondary issue. A few Lisp definitions can provide whatever *name* you need provided that the underpinning capability is present. So for instance (and as an approximation to what would really be needed) you could go

```
(de + (a b) (plus a b))
```

and subsequently use "+" rather then "`plus`" to add numbers. If you thing of building Lisp code as extending Lisp then if you start from a slightly different Lisp dialect then you merely start with a few definitions that are there to provide whatever compatibility you need. This attitude then reduces the whole discussion to merely a stand-off between those who believe that a "proper" Lisp should have almost every capability that is in any way supportable by your computer and those who prefer a smaller simpler system that just provides enough for the application area that interests them.

Strong data types: Software Engineers will emphasis the benefits of strict compile-time type checking as a crucial aid to the construction of reasonably reliable code. A Lisp fanatic[2] responds by citing the convenience of Lisp for rapid prototyping. They can point out that Lisp's natural ability to inspect Lisp code (because code and data share a single representation) leads to ease in creating custom tools to analyse code, and that at least sometimes that has been deployed to help with code robustness. And finally they will point to the way in which all Lisp data has a printed representation, and so tracing and debugging Lisp code will be so much nicer than the corresponding experience when using other languages – so a few extra bugs may not matter that much!

Performance: Is Lisp a "fast" programming language? Well one thing is unquestionably the case: The `vsl`implementation is *slow* compared to most other languages you might use – but that is a consequence of it emphasising conciseness over almost everything else. If Lisp code is written without regard to performance it will probably not run especially fast, but masters of the language using top-end Lisp implementations can coax benchmarks in Lisp to beat almost any other language. Looking at the raw power of even the cheapest computers today it seems reasonable to accept the Lisp realist's view that most Lisp code will run somewhat slower than code written in C or Fortran, but that these days it will be truly rate to find a situation where this matters.

Linguistic correctness: From the very start Lisp has been a pragmatic language providing capabilities to let people "get the job done". In its early days it was towards the forefront of adhering to principles of correctness. One early illustration of this attitude was the Ph.D. work of Michael Gordon[18] who noted that a relatively obscure feature (`label`) in Lisp 1.5 had a semantically inconvenient (and so arguably incorrect) definition. However mainstream Lisps have continued to include various imperative and destructive

[2]There are some languages that tend to breed or collect fanatics, and Lisp is undoubtedly one such.

capabilities, and to varying extents have left in ugliness. Scheme[36] is a spin-off language that made a serious attempt to tidy things up, and Common Lisp insists (for instance) that interpreted and compiled code should behave in exactly the same way – while other systems accept incompatibilities over this and over the dynamic scoping of variable bindings so as to achieve simpler or faster implementations.

Object Orientation: The Common Lisp Object System is a part of ANSI Common Lisp, however it is rather substantially different from the Object models present in other programming languages. Lisp fanatics naturally view it is greatly superior! Perhaps the most specific claims for it will be that its meta-object protocol provides exceptional flexibility when developing and them maintaining code. To others its support for multiple inheritance and the fact that it does not enforce encapsulation mean that many of the features that could make Object Orientation a foundation for reliable code and missing from it. So on the spirit of Lisp being a ball of mud, many versions of it have embraced object orientation, but it still seems fair to suggest that the core ways of using Lisp are closer to being Functional then Object Oriented.

Real-time and Windowed: Some (but not all) implementations of Common Lisp support the Common Lisp Interface Manager which is a window managing library allowing programmers to create modern user interfaces. Other systems may have their own support for events and windows. Again in the spirit of the ball of mud one can never say that Lisp is incapable of anything! However historically this is not one of the areas of Lisp key competence, and even more not one where smooth portability from one Lisp to another can be guaranteed. In the early years Lisp was heavily used in Artificial Intelligence work, specifically including the control of robots. For these any pause in its working could be extremely visible as a pause or judder in the robot's movements. This led to severe concern about real-time responsiveness, and some Lisp implementations put much effort into achieving it. As computers have become (much) faster most of the worries on this front have faded, but it is still the case that if you need strict real-time responses you will need not only a specialist Lisp implementation but a specialist real-time operating system for it to run on top of.

Lisp was originally envisaged as a notation for writing programs in the area of Artificial Intelligence. While its use spread some way beyond that, it is still the case that some of the most striking examples of big programs written in Lisp have been to manipulate algebraic formulae, perform logical reasoning, control autonomous robots and analyse natural language. Since then it has been applied

in almost all areas. As a final characterisation of its status and importance consider Greenspun's Tenth Rule of Programming[21]:

> *Any sufficiently complicated C or Fortran program contains an ad hoc, informally-specified, bug-ridden, slow implementation of half of Common Lisp.*

so understanding Lisp makes sense for when you end up needing to write or maintain that complicated other program that is notionally written in a quite different language. And `vsl`certainly contains a slow implementation of at least parts of Common Lisp.

Chapter 3

Versions of Lisp

The two most widely referenced Lisp-family dialects today are Common Lisp[39] and Scheme[36]. There are proper standards documents that define each of these. Common Lisp is huge and attempts to encompass absolutely everything, while Scheme is sometimes described as "minimalist".

Neither of these dialects is used here. That situation clearly needs justification. The short form of this is that Common Lisp has so many facilities that a subset short enough to explain will omit so much of what would make it "Common" as to be ridiculous. Scheme on the other hand is a fairly compact language, but it insists on "correct" treatment of various key features (notably variable scopes, tail recursion and continuations) that tend to make an implementation larger, slower and harder to explain.

There have been many other Lisp dialects and implementations. Common Lisp can be seen as the successor to Maclisp and its variants embodied in various special-purpose Lisp computers built at MIT. HLISP[19] was developed by Eiichi Goto in Japan with a key feature that memory was saved by using hashing to implement the `cons` operation so that as much data as possible could be shared. HLISP led to the development of the FLATS[20] computer which combined high performance for both Lisp-style symbolic computation with good numeric capability – with hardware

Figure 3.1: Eiichi Goto and the FLATS machine

acceleration for hashing
and associative operations
that made HLISP special.

At the other extreme of the size and performance scale, muLisp (in due course distributed by the Soft Warehouse Inc) started as a 2 Kbyte implementation manually toggled into the memory of a 4 Kbyte computer based on an 8080 processor[26].

One of the notable implementations of Common Lisp is `gcl` (GNU Common Lisp). Its full source can be fetched via (for instance) the Free Software Foundation's website at `http://www.gnu.org/software/gcl`. The source archive contains around 180,000 lines of Lisp code, as well as significant amounts of C code that among other things provide support for arbitrary precision arithmetic. The "Visible" Lisp used here (`vsl`) has as its main source just over 3000 lines of C code, which is already quite enough to try to explain.

Scheme is smaller, to the extent that some would claim that the basic version was only suitable for teaching not for real use. There are an enormous number of implementations of if: one list names 78. But the vast majority of these are also much bulkier than `vsl` and hence would be harder to explain.

The alternative approach used here has been to go a little further back in history and use the Standard Lisp dialect that originated at the University of Utah. There were two stages in its definition. The report defining an initial version of it was by Tony Hearn in 1969[25] and was his attempt to codify a Lisp dialect that could provide portability for his Reduce algebra system[24]. A decade later Marti, Hearn and Griss produced a substantial update[28]. Griss, Benson and Maguire released PSL[22] as a Portable implementation of Standard Lisp a couple of years later, and others involved with the Reduce project followed on with at least Cambridge Lisp[16], then CSL[31] and a Java-coded version known as Jlisp[32]. These days PSL, CSL and Jlisp can all be found on the website `http://reduce-algebra.sourceforge.net` and all are both in use and continued development.

By following Standard Lisp the code and explanations here steer a line between excess complexity and being so cut down and simple that they have no real relevance. The simplicity of `vsl`can be assessed by just looking at the fairly small amount of C code that implemented it, and this code implements the most important facilities that any Lisp variant will require. The power of `vsl` can be seen from the fact that it can be used to build the roughly 350,000 lines of code that make up the whole of the Reduce algebra system, with almost every part of that code working as expected. There are three key limitations that mean that Reduce built in top of `vsl` may not be suitable for heavy use. The first is that `vsl` only supports integers with values up to 9223372036854775807 in a tolerably efficient manner, while Reduce needs fully arbitrary precision arithmetic in many places. The second is that `vsl` provides just an interpreter for Lisp, not a compiler, and as

a result calculations performed using it will be somewhat slow. The third issue is that `vsl` only supports very basic interaction with the user via the keyboard, and many of today's users would expect a much richer user-interface. But all those limitations admitted, it remains that `vsl` can support Lisp applications that run up in to the hundreds of thousands of lines of source code!

Despite the adoption of the Standard Lisp dialect for `vsl` it has to be conceded that Common Lisp can not properly be ignored. It remains the fact that a full implementation would be much too large to explain, and indeed even full coverage of the language from the perspective of a user would need a seriously lengthy and probably rather dull book. So three things are done here to provide support for a move onward from `vsl` into use of Common Lisp. First all the example programs are provided in Common Lisp for as part of the down-loadable resources to accompany this book. The code there has comments in it noting such special features of Common Lisp as appear. Secondly a modified version of `vsl` that provides compatibility with the core of Common Lisp is also provided for download – it is known as `vcl`. `vcl` is naturally far from complete as a Common Lisp, and its documentation here is much less complete than that for `vsl`, but it can form a bridge to use of Big Lisp. Many of the features of `vsl` are implemented as Lisp code that builds on a fairly small number of primitive operations. The final aspect of Common Lisp support is that the most interesting or important of these are described and explained as an extended set of fragments of Lisp sample code. This explanation can give both insight into some of the power of Common Lisp and some of the pain it gives any person trying to provide a high performance implementation.

A definite stance taken with `vsl` is that it is not intended to be viewed as a fully-finished Lisp system merely for use. It is more a starting point for project work that will extend it. Those who are convinced that Common Lisp or Scheme would be better can use much of the core technology within `vsl` and adapt it to implement the dialect that they are keen on. It would be essentially trivial to just change the names that `vsl` uses for the Lisp functions that it supplies. For instance Scheme uses the names `set!` and `equal?` where `vsl` uses `setq` and `equal`, and Common Lisp used `defun` where `vsl` used `de`. Adding extra functions, and even data-types should be straightforward. Deeper semantic differences between the dialects would naturally lead to more challenging re-work.

Anybody who became enthusiastic about using `vsl` for algebraic computation might retrofit full-speed arbitrary prevision arithmetic. If they wanted to match the arithmetic capability of Common Lisp they would also need to add complex numbers and a full range of elementary functions working on them.

Lisp is a very natural language for compiler-building, and a small and fairly tidy version like `vsl` could provide a good starting point for a project to build one. One of the examples given later here may provide some sort of a start, but

Appendix 3 in Berkeley and Bobrow[7] (one of many important historical Lisp books now available on-line) includes a listing of a complete real Lisp compiler – albeit for a machine now long defunct. There is no reason to believe that a compiler for `vsl` would not transform it into a system speed-competitive with all other current Lisps. The compiler in Berkeley and Bobrow is maybe 1500 lines of code, and the key parts of if that do the central interesting bits of the work are maybe only around 500 lines. This is not an outrageously huge project.

User-interface refinement is something that could generate an almost unlimited amount of activity. Before starting on a project in this area it would make sense to review what others have done. One possible place to start would be to investigate LispWorks[27], which provides an integrated cross-platform development environment.

Chapter 4

A first session with Lisp

To use the "vsl" version of Lisp discussed here you will first have to fetch it and ensure that it works on your computer. This should be possible on almost any system that presents itself as a full computer, and Appendix A gives explicit explanations for how to do this for some of the most plausible platforms. Specifically it covers the case for Raspberry Pi, an ordinary pc running Windows and a Macintosh. At present vsl is not directly usable on tablet-format machines, mobile phones or the like. That is not because it could not be ported to them, but more because the simple inner workings that it seeks to present would end up wrapped in rather a lot of messy platform-specific code. When using vsl you will generally be interacting using a command prompt and keyboard rather than any more modern-seeming graphical interface.

```
(de factorial (n)
   (cond
      ((zerop n) 1)
      (t (times n (factorial (sub1 n))))))

(dotimes (i 40)
   (prin i)
   (princ blank)
   (print (factorial i)))
```

```
0 1
1 1
2 2
3 6
4 24
5 120
6 720
```

```
7 5040
8 40320
9 362880
10 3628800
11 39916800
12 479001600
13 6227020800
14 87178291200
15 1307674368000
16 20922789888000
17 355687428096000
18 6402373705728000
19 121645100408832000
20 2432902008176640000
21 51090942171709440000
22 1124000727777607680000
23 25852016738884976640000
24 620448401733239439360000
25 15511210043330985984000000
26 403291461126605635584000000
27 10888869450418352160768000000
28 304888344611713860501504000000
29 8841761993739701954543616000000
30 265252859812191058636308480000000
31 8222838654177922817725562880000000
32 263130836933693530167218012160000000
33 8683317618811886495518194401280000000
34 295232799039604140847618609643520000000
35 10333147966386144929666651337523200000000
36 371993326789901217467999448150835200000000
37 13763753091226345046315979581580902400000000
38 523022617466601111760007224100074291200000000
39 20397882081197443358640281739902897356800000000
Value: nil
```

Chapter 5

Names, Lists, Variables and Properties

Chapter 6

Simple function definitions

Chapter 7

Tests and Comparisons

Chapter 8

More data structures

Chapter 9

Styles of Lisp programming

Chapter 10

Macros

Chapter 11

Functions as values

Chapter 12

Inside Lisp: data representation

A Lisp system could be implemented in almost any programming language. In the very early days the core of the system would have been written directly in assembly code for the computer of interest. That could give ultimate flexible control of everything, but these days it feels too much like hard work. If you look at the Lisp 1.5 Programmer's manual[13] you will find various aspects of Lisp's behaviour being explained by showing the assembly code that implements them. Even at that stage Lisp was a language that could see itself as "visible".

A proper – indeed perhaps *the* proper way to build a Lisp is to write the Lisp system in Lisp itself. That can work if you extend Lisp with names for a suitable range of low-level operations to use machine resources such as addresses, bytes and pointers. A Lisp compiler (naturally written in Lisp) can then convert this Lisp description into machine code that is then ready to work. The PSL[22] system is an example of using this strategy. There is an obvious question about how you get started. If your Lisp system and its compiler is coded in Lisp then you need a working Lisp system before you can compile it. The practical solution is to use some existing Lisp the very first time, but then once you have got something reasonably workable you use the previous version of your system to build the next. If you decide to try to build a Lisp in this manner you could perfectly reasonably use `vsl` as the initial launchpad to get you off the ground.

A final option is to build your Lisp using some existing established programming language. That is what has been done with `vsl`, with the existing language being C. The same path was taken for the CSL Lisp[31] that `vsl` is at least partly styled after. The view taken here is that C is a language that provides most of the flexibility and power of assembly code but that looks a lot neater. C works on the basis of "You Asked For It So You Got It" so even small errors in C code can lead to corrupted data and most obscure consequences. However by and large C code runs reasonably fast and there are certainly very high quality C compilers available for most sensible models of computer. It would have been possible to

make a different selection (for instance Jlisp[32] is all coded in Java).

Once the implementation language has been selected a first challenge for a Lisp implementer is to decide how data will be represented. It is going to be necessary to have symbols, numbers, strings and lists (together with a few other oddities) and a key feature of Lisp is that a Lisp Object can be any one of these and it must be possible to have a run-time test to determine which sort it is.

The choice made in `vsl` is that every Lisp Object will be represented as a machine integer that is wide enough to double-up as a pointer. Modern implementations of C provide a type called `intptr_t` that is suitable. On some architectures (for instance the ARM processor in a Raspberry Pi) these integers are 32-bits wide, while on others (say a Macintosh) they are 64. The code in `vsl` has been written so it will cope equally well in both cases.

To arrange that machine integers can be used to stand for all the various Lisp data-types `vsl` uses the bottom three bits of each value as tag-bits. A consequence of this design is that these bits are not available when the object is used as a pointer – and when it represents a list or a symbol (or indeed a string, vector, floating point number...) it has to. Fortunately that is not a disaster, because it is easy to arrange that every valid item that `vsl` ever need to have a pointer gets allocated at an address that is a multiple of 8. The "trickery" of this is just the sort of thing that is possible in C but that many other languages protect you from being able to do.

The important parts of the C code making up `vsl` are all included and explained here. The full version is naturally available for download from the associated web-site. If you are not already an expert at C coding you should still be able to see what is going on, and as a side-effect of reading this you may well find you have learned quite a lot about C as well as Lisp!

The start of the code merely introduces the various tag bits and provides facilities with name such as `isCONS` that check for each case. The first few cases will already make sense, but the ones named `tagATOM`, `tagFORWARD` and `tagHEADER` will be explained later.

```
// A Lisp item is represented as an integer and the low
// 3 bits contain tag information that specify how the
// rest will be used.

typedef intptr_t LispObject;

#define TAGBITS    0x7

#define tagCONS    0     // Traditional Lisp "cons" item.
#define tagSYMBOL  1     // A symbol.
#define tagFIXNUM  2     // An integer (29 or 61 bits).
```

```
#define tagFLOAT   3     // A double-precision number.
#define tagATOM    4     // Something with a header word.
#define tagFORWARD 5     // Used by garbage collection.
#define tagHDR     6     // Header word of an atom .
#define tagSPARE   7     // Not used!

// Note that in the above I could have used tagATOM to
// include the case of symbols (aka identifiers) but as
// an optimisation I choose to make that a special case.
// I still have one spare code (tagSPARE) that could be
// used to extend the system.

// Now I provide macros that test the tag bits.
// These are all rather obvious!

#define isCONS(x)     (((x) & TAGBITS) == tagCONS)
#define isSYMBOL(x)   (((x) & TAGBITS) == tagSYMBOL)
#define isFIXNUM(x)   (((x) & TAGBITS) == tagFIXNUM)
#define isFLOAT(x)    (((x) & TAGBITS) == tagFLOAT)
#define isATOM(x)     (((x) & TAGBITS) == tagATOM)
#define isFORWARD(x)  (((x) & TAGBITS) == tagFORWARD)
#define isHDR(x)      (((x) & TAGBITS) == tagHDR)
```

Each of these cases can be considered individually. The first is the one that is most fundamental to Lisp, and is used for the `cons` cells that make up lists. A `LispObject` that is tagged as `tagCONS` is a bit pattern that can be used as C pointer to the two further `LispObject`s that will be the `car` and `cdr` of the item. Accessing them is an exercise in telling C to ignore all the proper static type analysis that robust programming languages love. It converts the object into a pointer and accesses it as if it was a little array with two elements. That "cheating" is wrapped up in a pair of C macros so that throughout the rest of the code it looks neat and natural.

```
// Macros the extract fields from LispObjects ...

#define qcar(x) (((LispObject *)(x))[0])
#define qcdr(x) (((LispObject *)(x))[1])
```

That is pretty well all that needs to be said about `cons` cells and hence lists! Well we will need to have functions that can create them, but that comes later. What we can do already is test if we have one (using `isCONS`) and access the `car` and `cdr` components. It is perhaps worth noting that a `cons` cell will consist of two adjacent `LispObject`s, and so on a 32-bit computer it will occupy 8 bytes and on a 64-bit one 16 bytes. In each case this size is a multiple of 8, and so

laying `cons` cells side by side will naturally maintain the 8-byte alignment that was required to make the tagging system in `vsl` work.

Something rather similar is done for symbols. However rather than having just two fields called `car` and `cdr` a symbol is implemented in `vsl` with space for six sub-parts. These first of these will hold various flag bits. For instance one of these flags is used to mark the name as one subject to tracing. Then there are fields that store the current value associated with the symbol it is is viewed as a variable, its property list (as used with `put` and `get`) and a reference to the string that is the name of the symbol. The name is kept at arms-length in this way because the names of different symbols can be different sizes, and it is convenient to wrap the handling of that up in the mechanism used for strings. Then there are two fields that are to do with having a function definition associated with the symbol. `qdefn` holds a C pointer, and when used it will be the entry-point of a C function to be used. `qlits` is for Lisp data associated with any way the symbol is to be used as a function.

```
#define qflags(x) (((LispObject *)((x)-tagSYMBOL))[0])
#define qvalue(x) (((LispObject *)((x)-tagSYMBOL))[1])
#define qplist(x) (((LispObject *)((x)-tagSYMBOL))[2])
#define qpname(x) (((LispObject *)((x)-tagSYMBOL))[3])
#define qdefn(x)  (((void **)     ((x)-tagSYMBOL))[4])
#define qlits(x)  (((LispObject *)((x)-tagSYMBOL))[5])

// Bits within the flags field of a symbol.

#define flagTRACED    0x080
#define flagSPECFORM  0x100
#define flagMACRO     0x200
// There are LOTS more bits available for flags here.
```

Observe that when accessing these the tag-bits (`tagSYMBOL`) have to be removed to get something C will be happy interpreting as an little array. It should already be possible to see that a key part of evaluating a Lisp form could involve a line of code somewhat like

```
if (isSYMBOL(x)) return qvalue(x);
```

to detect the case of a symbol being used as a variable and return its value. In `vsl` when a symbol is first created its value-cell is filled with a special value that marks it as not having a useful value. This is in fact a symbol with the deliberately unwieldy name `~indefinite-value~` and in the C code a reference to that loves in a variable called `undefined`. If this value is encountered it means that the user looked in the value of a variable without setting it first, and a diagnostic will be generated.

Small integers are coped with by `vsl` by using the part of the `LispObject` not used by the tag bits to store the numeric value directly. This means that the term "small" is in fact not that limiting! C provides symbols like `INTPRT_MAX` that give the limiting size of its `intptr_t` data type, so getting a handle on the range for Lisp small numbers (known as "fixnums") is straightforward. Floating point numbers can not afford to have any bits stolen for use as tags, so go back to the pattern of using a pointer to memory where the value is stored.

```
// Fixnums and Floating point numbers are  easy!

#define qfixnum(x)      ((x) >> 3)
#define packfixnum(n)   ((((LispObject)(n)) << 3) \
                        + tagFIXNUM)

#define MIN_FIXNUM      qfixnum(INTPTR_MIN)
#define MAX_FIXNUM      qfixnum(INTPTR_MAX)

#define qfloat(x)       (((double *)((x)-tagFLOAT))[0])
```

The next tag value shown is `tagATOM` and this is used for strings, bigger integers and vectors. The data in this case is a pointer to a block of memory, which must be a multiple of 8 bytes long. To make it possible to have lots of different sorts of data collected together under this single tag value the first word of the memory pointed at will be tagged with `tagHEADER` and additional bits in it will show both what sort of entity is involved and how long it is. The choice made in `vsl` is to allow for up to 16 sorts of data. This means that a header word has 3 bits of tag (that confirm that it is a header) and 4 bits to explain what it is a header for. That leaves the rest of the word to contain length information. On a 32-bit computer and given that the length is recorded in bytes this means that the largest object is limited to around 32 Megabytes. For Lisp use this is really unlikely to be a problem – especially for a small system such as `vsl`. However whenever a limitation gets built into a system by some early design decision it can make sense to consider how it could be lifted. In this case the most obvious way out is to note that any user needing amazingly long strings, huge arrays or other data that could approach this limit should just move to a 64-bit computer where `vsl` length-codes are 57 bits rather then 25: they should then not have troubles in the foreseeable future. If it was vital to change `vsl` to cope with bigger items it would be possible to make a modest expansion by making the length-code count in words rather than bytes, or a big one my relocating length information to the word following the header so that nothing was lost via packing.

The code-fragment the follows shows type-codes that go within headers and that identify Symbols, Strings, Vectors, Big numbers and some varieties of Hash table. And as the comments note there are quite a few codes left available for

expansion, but perhaps not enough to allow coverage of all the various sorts of object a full version of Common Lisp would have required. The type-code for symbols exists so that the initial word of a symbol (previously just described as containing flags) can have mark bits that identify it as a header, and that helps the garbage collector.

```
// In memory CONS cells and FLOATS exist as just 2-word
// items with all their bits in use. All other sorts of
// data have a header word at their start. This contains
// extra information about the exact form of data present

#define qheader(x) (((LispObject *)((x)-tagATOM))[0])

#define TYPEBITS    0x78

#define typeSYM     0x00
#define typeSTRING  0x08
#define typeVEC     0x10
#define typeBIGNUM  0x18
#define typeEQHASH  0x20
#define typeEQHASHX 0x28
// Codes 0x30, 0x38, 0x40, 0x48, 0x50, 0x58, 0x60, 0x68,
// 0x70 and 0x78 spare!

#define veclength(h)  (((uintptr_t)(h)) >> 7)
#define packlength(n) (((LispObject)(n)) << 7)
```

When the length of a vector is unpacked from a header word it is shifted using the type `uintptr_t` which is a C type that is "unsigned". This avoids any potential bad effects associated with large length values overflowing into a sign-bit and ending up appearing to be negative.

Given the above, it is now easy to see how to represent strings and big numbers. Each live in memory as a header word followed by the sequence of bytes that make up their data. In the initial version of `vsl` the numbers always just hold an `int64_t` value, so it makes sense to keep that data aligned at an 8-byte boundary. Access to the two varieties of data happens as follows:

```
#define isBIGNUM(x) \
   (isATOM(x) && ((qheader(x) & TYPEBITS) == \
   typeBIGNUM))
#define qint64(x) (*(int64_t *)((x) - tagATOM + 8))

#define isSTRING(x) \
   (isATOM(x) && ((qheader(x) & TYPEBITS) == \
```

```
   typeSTRING))
#define qstring(x) ((char *)((x) - tagATOM + \
   sizeof(LispObject)))
```

All other data types are handled in very similar ways. The code presented so far is in fact just declarations and C macros that are there to bridge between the human and Lisp view of objects and the detailed bit-packing representations that `vsl` has adopted. If somebody needed to change the representations then they could alter these macros and at least a great deal of the rest of the `vsl` code could remain intact.

The native `vsl` representation of a big-number is as as atom that could hold an arbitrary amount of data, but as has been explained the code is kept simple by using just 64-bits. This means that the largest integer that can be handled is only 19 digits long - specifically it is 9223372036854775807. For a real Lisp system this is dreadfully limiting! But providing C code for higher precision arithmetic would have enlarged `vsl` and distracted from its use in presenting aspects of Lisp implementation. As a compromise the code arranges to interpret a list structure of the form `(! bignum 000 000 000 ...)` as not quite a list, but as if it were a new special data-type. The number of places the code has to be aware of this is amazingly small! The code for the predicate `atom` has a tiny adjustment that detects the special case, and the code to print things notices it and takes special action. Finally a special built-in function called `bignump` checks if its argument is a list structure with the special tag at its head. With these very minor adjustments full support for unlimited precision arithmetic can be provided as Lisp code in the `vsl` library. It is perhaps useful to have this arrangement in that it illustrates how easily an existing Lisp system can be extended to add new data types – even if at first sight they would seem to need support at a really low level.

To further illustrate how this C-based low-level way of representing data is being used it is worth showing what would be more natural to do in a stricter and more modern language. The example used here is Java where classes are used to represent each sort of Lisp object, and sub-classing to establish their hierarchy:

```
class LispObject  // A base class
{
}

class Pair extends LispObject
{
    LispObject car, cdr;
}

class LispFixNum extends LispObject
```

```
{
    int n;
}

class LispString extends LispObject
{
    String s;
}

class Symbol extends LispObject
{
    LispObject value, plist;
    LispString pname;
    ...
}
```

The Java version is really a lot cleaner, and when in use Java will police access where that helps ensure system integrity. Furthermore methods relating to each sort of Lisp data can be put in as part of the class that describes the data. However Java will be needing to do some work behind the scenes to cope with its class hierarchy, and it is most unlikely that it will use the carefully designed hand-optimised assignments of bit-patterns that the C code uses to discriminate between types. You can expect a trade-off with the Java version looking nicer but the C version being able to make full use of machine resources.

The C code that makes up `vsl` continues to expose low level detail when it comes to allocating new `cons` cells. It has a block of memory that starts at `heap1base` and ends at `heap1end`. It allocates `cons` cells, symbols, strings and most other things sequentially up from the start and floating point values downwards from the top. It also has a block of memory it uses as a stack to save pointers to Lisp objects at some stages when that is necessary.

```
#define push(x)         { *sp++ = (x); }
#define TOS             (sp[-1])
#define pop(x)          { (x) = *--sp; }
#define push2(x, y)     { push(x); push(y); }
#define pop2(y, x)      { pop(y); pop(x); }

LispObject cons(LispObject a, LispObject b)
{   if (fringe1 >= fpfringe1)
    {   push2(a, b);
        reclaim();
        pop2(b, a);
    }
```

```
    qcar(fringe1) = a;
    qcdr(fringe1) = b;
    a = fringe1;
    fringe1 += 2*sizeof(LispObject);
    return a;
}

LispObject boxfloat(double a)
{   LispObject r;
    if (fringe1 >= fpfringe1) reclaim();
    fpfringe1 -= sizeof(double);
    r = fpfringe1 + tagFLOAT;
    qfloat(r) = a;
    return r;
}
```

It should be clear from the above that {\tx fringe1} records the upper limit of active material, and {\tx fpfringe1} the lower limit of allocated floating point data. When these collide the function {\tx reclaim} is called to tidy up memory recycling any that is no longer needed. The use of the macros {\tx push2} and {\tx pop2} arrange that the garbage collection process does not lose track of the values that are to be placed in the new {\tx cons}.

Code rather similar in style to that for {\tx cons} allocates symbols, strings and the like, and so is not included here, but when you check it in the full \vsl{} sources it should be easy to understand.

```
#define printPLAIN   1
#define printESCAPES 2

int linelength = 80, linepos = 0, printflags = printESCAPES;

#define MAX_LISPFILES 30
FILE *lispfiles[MAX_LISPFILES];
int32_t file_direction = 0, interactive = 0;
int lispin = 0, lispout = 1;

extern LispObject lookup(const char *s, int n, int createp);

void wrch(int c)
```

```
{
    if (lispout == -1)
    {   char w[4];
// This bit is for the benefit of explode and explodec
        LispObject r;
        w[0] = c; w[1] = 0;
        r = lookup(w, 1, 1);
        work1 = cons(r, work1);
    }
    else if (lispout == -2) boffo[boffop++] = c;
    else
    {   putc(c, lispfiles[lispout]);
        if (c == '\n')
        {   linepos = 0;
            fflush(lispfiles[lispout]);
        }
        else linepos++;
    }
}

int rdch()
{   LispObject w;
    if (lispin == -1)
    {   if (!isCONS(work1)) return EOF;
        w = qcar(work1);
        work1 = qcdr(work1);
        if (isSYMBOL(w)) w = qpname(w);
        if (!isSTRING(w)) return EOF;
        return *qstring(w);
    }
    else
    {   int c = getc(lispfiles[lispin]);
        if (c != EOF && qvalue(echo) != nil) wrch(c);
        return c;
    }
}

int gensymcounter = 1;

void checkspace(int n)
{   if (linepos + n >= linelength && lispout != -1) wrch('\n');
```

```
}

char printbuffer[32];

extern LispObject call1(const char *name, LispObject a1);
extern LispObject call2(const char *name, LispObject a1, LispObject a2);

void internalprint(LispObject x)
{   int sep = '(', i, esc, len;
    char *s;
    LispObject pn;
    switch (x & TAGBITS)
    {   case tagCONS:
            if (x == 0)    // can only occur in case of bugs here.
            {   wrch('#');
                return;
            }
            while (isCONS(x))
            {   i = printflags;
                if (qcar(x) == bignum &&
                    (pn = call1("~big2str", qcdr(x))) != NULLATOM &&
                    pn != nil)
                {   printflags = printPLAIN;
                    internalprint(pn);
                    printflags = i;
                    return;
                }
                printflags = i;
                checkspace(1);
                if (linepos != 0 || sep != ' ' || lispout < 0) wrch(sep);
                sep = ' ';
                push(x);
                internalprint(qcar(x));
                pop(x);
                x = qcdr(x);
            }
            if (x != nil)
            {   checkspace(3);
                wrch(' '); wrch('.'); wrch(' ');
                internalprint(x);
            }
```

```
        checkspace(1);
        wrch(')');
        return;
    case tagSYMBOL:
        pn = qpname(x);
        if (pn == nil)
        {   int len = sprintf(printbuffer, "g%.3d", gensymcounter++);
            push(x);
            pn = makestring(printbuffer, len);
            pop(x);
            qpname(x) = pn;
        }
        len = veclength(qheader(pn));
        s = qstring(pn);
        if ((printflags & printESCAPES) == 0)
        {   int i;
            checkspace(len);
            for (i=0; i<len; i++) wrch(s[i]);
        }
        else if (len != 0)
        {   esc = 0;
            if (!islower((int)s[0])) esc++;
            for (i=1; i<len; i++)
            {   if (!islower((int)s[i]) &&
                    !isdigit((int)s[i]) &&
                    s[i]!='_') esc++;
            }
            checkspace(len + esc);
            if (!islower((int)s[0])) wrch('!');
            wrch(s[0]);
            for (i=1; i<len; i++)
            {   if (!islower((int)s[i]) &&
                    !isdigit((int)s[i]) &&
                    s[i]!='_')
                    wrch('!');
                wrch(s[i]);
            }
        }
        return;
    case tagATOM:
        if (x == NULLATOM)
```

```
{   checkspace(5);
    wrch('#'); wrch('n'); wrch('u'); wrch('l'); wrch('l');
    return;
}
else switch (qheader(x) & TYPEBITS)
    {   case typeSTRING:
            len = veclength(qheader(x));
            s = qstring(x);
            if ((printflags & printESCAPES) == 0)
            {   int i;
                checkspace(len);
                for (i=0; i<len; i++) wrch(s[i]);
            }
            else
            {   esc = 2;
                for (i=0; i<len; i++)
                    if (s[i] == '"') esc++;
                checkspace(len+esc);
                wrch('"');
                for (i=0; i<len; i++)
                {   if (s[i] == '"') wrch('"');
                    wrch(s[i]);
                }
                wrch('"');
            }
            return;
        case typeBIGNUM:
            sprintf(printbuffer, "%" PRId64, qint64(x));
            checkspace(len = strlen(printbuffer));
            for (i=0; i<len; i++) wrch(printbuffer[i]);
            return;
        case typeVEC:
        case typeEQHASH:
        case typeEQHASHX:
            sep = '[';
            push(x);
            for (i=0; i<veclength(qheader(TOS))/sizeof(LispOb
            {   checkspace(1);
                wrch(sep);
                sep = ' ';
                internalprint(elt(TOS, i));
```

```
                        }
                        pop(x);
                        checkspace(1);
                        wrch(']');
                        return;
                    default:
                        //case typeSYM:
                        // also the spare codes!
                        disaster(__LINE__);
                }
        case tagFLOAT:
            {   double d =  *((double *)(x - tagFLOAT));
                if (isnan(d)) strcpy(printbuffer, "NaN");
                else if (finite(d)) sprintf(printbuffer, "%.14g", d);
                else strcpy(printbuffer, "inf");
            }
            s = printbuffer;
// The C printing of floating point values is not to my taste, so I (slig
// asjust the output here...
            if (*s == '+' || *s == '-') s++;
            while (isdigit((int)*s)) s++;
            if (*s == 0 || *s == 'e')  // No decimal point present!
            {   len = strlen(s);
                while (len >= 0)       // Move existing text up 2 places
                {   s[len+2] = s[len];
                    len--;
                }
                s[0] = '.'; s[1] = '0'; // insert ".0"
            }
            checkspace(len = strlen(printbuffer));
            for (i=0; i<len; i++) wrch(printbuffer[i]);
            return;
        case tagFIXNUM:
            sprintf(printbuffer, "%" PRId64, (int64_t)qfixnum(x));
            checkspace(len = strlen(printbuffer));
            for (i=0; i<len; i++) wrch(printbuffer[i]);
            return;
        default:
//case tagFORWARD:
//case tagHDR:
            disaster(__LINE__);
```

```
    }
}

LispObject prin(LispObject a)
{   printflags = printESCAPES;
    push(a);
    internalprint(a);
    pop(a);
    return a;
}

LispObject princ(LispObject a)
{   printflags = printPLAIN;
    push(a);
    internalprint(a);
    pop(a);
    return a;
}

LispObject print(LispObject a)
{   printflags = printESCAPES;
    push(a);
    internalprint(a);
    pop(a);
    wrch('\n');
    return a;
}

void errprint(LispObject a)
{   int saveout = lispout, saveflags = printflags;
    lispout = 1; printflags = printESCAPES;
    internalprint(a);
    wrch('\n');
    lispout = saveout; printflags = saveflags;
}

LispObject printc(LispObject a)
{   printflags = printPLAIN;
    push(a);
    internalprint(a);
    pop(a);
```

```
    wrch('\n');
    return a;
}

int curchar = '\n', symtype = 0;

int hexval(int n)
{   if (isdigit(n)) return n - '0';
    else if ('a' <= n && n <= 'f') return n - 'a' + 10;
    else if ('A' <= n && n <= 'F') return n - 'A' + 10;
    else return 0;
}

LispObject token()
{   symtype = 'a';           // Default result is an atom.
    while (1)
    {   while (curchar == ' ' ||
               curchar == '\t' ||
               curchar == '\n') curchar = rdch(); // Skip whitespace
// Discard comments from "%" to end of line.
        if (curchar == '%')
        {   while (curchar != '\n' &&
                   curchar != EOF) curchar = rdch();
            continue;
        }
        break;
    }
    if (curchar == EOF)
    {   symtype = curchar;
        return NULLATOM;     // End of file marker.
    }
    if (curchar == '(' || curchar == '.' ||
        curchar == ')' || curchar == '\'' ||
        curchar == '`' || curchar == ',')
    {   symtype = curchar;   // Lisp special characters.
        curchar = rdch();
        if (symtype == ',' && curchar == '@')
        {   symtype = '@';
            curchar = rdch();
        }
        return NULLATOM;
```

```
    }
    boffop = 0;
    if (isalpha(curchar) || curchar == '!') // Start a symbol.
    {   while (isalpha(curchar) ||
               isdigit(curchar) ||
               curchar == '_' ||
               curchar == '!')
        {   if (curchar == '!') curchar = rdch();
            else if (curchar != EOF && qvalue(lower) != nil) curchar = to
            else if (curchar != EOF && qvalue(raise) != nil) curchar = to
            if (curchar != EOF)
            {   if (boffop < BOFFO_SIZE) boffo[boffop++] = curchar;
                curchar = rdch();
            }
        }
        boffo[boffop] = 0;
        return lookup(boffo, boffop, 1);
    }
    if (curchar == '"')                     // Start a string
    {   curchar = rdch();
        while (1)
        {   while (curchar != '"' && curchar != EOF)
            {   if (boffop < BOFFO_SIZE) boffo[boffop++] = curchar;
                curchar = rdch();
            }
// Note that a double-quote can be repeated within a string to denote
// a string with that character within it. As in
//   "abc""def"   is a string with contents   abc"def.
            if (curchar != EOF) curchar = rdch();
            if (curchar != '"') break;
            if (boffop < BOFFO_SIZE) boffo[boffop++] = curchar;
            curchar = rdch();
        }
        return makestring(boffo, boffop);
    }
    if (curchar == '+' || curchar == '-')
    {   boffo[boffop++] = curchar;
        curchar = rdch();
// + and - are treated specially, since if followed by a digit they
// introduce a (signed) number, but otherwise they are treated as punctua
        if (!isdigit(curchar))
```

```
        {   boffo[boffop] = 0;
            return lookup(boffo, boffop, 1);
        }
    }
// Note that in some cases after a + or - I drop through to here.
    if (curchar == '0' && boffop == 0)  // "0" without a sign in front
    {   boffo[boffop++] = curchar;
        curchar = rdch();
        if (curchar == 'x' || curchar == 'X') // Ahah - hexadecimal input
        {   LispObject r;
            boffop = 0;
            curchar = rdch();
            while (isxdigit(curchar))
            {   if (boffop < BOFFO_SIZE) boffo[boffop++] = curchar;
                curchar = rdch();
            }
            r = packfixnum(0);
            boffop = 0;
            while (boffo[boffop] != 0)
            {   r = call2("plus2", call2("times2", packfixnum(16), r),
                          packfixnum(hexval(boffo[boffop++])));
            }
            return r;
        }
    }
    if (isdigit(curchar) || (boffop == 1 && boffo[0] == '0'))
    {   while (isdigit(curchar))
        {   if (boffop < BOFFO_SIZE) boffo[boffop++] = curchar;
            curchar = rdch();
        }
// At this point I have a (possibly signed) integer. If it is immediately
// followed by a "." then a floating point value is indicated.
        if (curchar == '.')
        {   symtype = 'f';
            if (boffop < BOFFO_SIZE) boffo[boffop++] = curchar;
            curchar = rdch();
            while (isdigit(curchar))
            {   if (boffop < BOFFO_SIZE) boffo[boffop++] = curchar;
                curchar = rdch();
            }
// To make things tidy If I have a "." not followed by any digits I will
```

```
// insert a "0".
            if (!isdigit((int)boffo[boffop-1])) boffo[boffop++] = '0';
        }
// Whether or not there was a ".", an "e" or "E" introduces an exponent a
// hence indicates a floating point value.
        if (curchar == 'e' || curchar == 'E')
        {   symtype = 'f';
            if (boffop < BOFFO_SIZE) boffo[boffop++] = curchar;
            curchar = rdch();
            if (curchar == '+' || curchar == '-')
            {   if (boffop < BOFFO_SIZE) boffo[boffop++] = curchar;
                curchar = rdch();
            }
            while (isdigit(curchar))
            {   if (boffop < BOFFO_SIZE) boffo[boffop++] = curchar;
                curchar = rdch();
            }
// If there had been an "e" I force at least one digit in following it.
            if (!isdigit((int)boffo[boffop-1])) boffo[boffop++] = '0';
        }
        boffo[boffop] = 0;
        if (symtype == 'a')
        {   int neg = 0;
            LispObject r = packfixnum(0);
            boffop = 0;
            if (boffo[boffop] == '+') boffop++;
            else if (boffo[boffop] == '-') neg=1, boffop++;
            while (boffo[boffop] != 0)
            {   r = call2("plus2", call2("times2", packfixnum(10), r),
                          packfixnum(boffo[boffop++] - '0'));
            }
            if (neg) r = call1("minus", r);
            return r;
        }
        else
        {   double d;
            sscanf(boffo, "%lg", &d);
            return boxfloat(d);
        }
    }
    boffo[boffop++] = curchar;
```

```
    curchar = rdch();
    boffo[boffop] = 0;
    symtype = 'a';
    return lookup(boffo, boffop, 1);
}

// Syntax for Lisp input
//
//   S ::= name
//     |   integer
//     |   float
//     |   string
//     |   ' S   | ` S  | , S  | ,@ S
//     |   ( T
//     ;
//
//   T ::= )
//     |   . S )
//     |   S T
//     ;

extern LispObject readT();

LispObject readS()
{   LispObject q, w;
    while (1)
    {   switch (symtype)
        {   case '?':
                cursym = token();
                continue;
            case '(':
                cursym = token();
                return readT();
            case '.':
            case ')':     // Ignore spurious "." and ")" input.
                cursym = token();
                continue;
            case '\'':
                w = quote;
                break;
            case '`':
```

```
                w = backquote;
                break;
            case ',':
                w = comma;
                break;
            case '@':
                w = comma_at;
                break;
            case EOF:
                return eofsym;
            default:
                symtype = '?';
                return cursym;
        }
        push(w);
        cursym = token();
        q = readS();
        pop(w);
        return list2star(w, q, nil);
    }
}

LispObject readT()
{   LispObject q, r;
    if (symtype == '?') cursym = token();
    switch (symtype)
    {   case EOF:
            return eofsym;
        case '.':
            cursym = token();
            q = readS();
            if (symtype == '?') cursym = token();
            if (symtype == ')') symtype = '?'; // Ignore if not ")".
            return q;
        case ')':
            symtype = '?';
            return nil;
        // case '(':  case '\'':
        // case '`':  case ',':
        // case '@':
        default:
```

```
            q = readS();
            push(q);
            r = readT();
            pop(q);
            return cons(q, r);
    }
}

LispObject lookup(const char *s, int len, int createp)
{   LispObject w, pn;
    int i, hash = 1;
    for (i=0; i<len; i++) hash = 13*hash + s[i];
    hash = (hash & 0x7fffffff) % OBHASH_SIZE;
    w = obhash[hash];
    while (w != tagFIXNUM)
    {   LispObject a = qcar(w);        // Will be a symbol.
        LispObject n = qpname(a);      // Will be a string.
        int l = veclength(qheader(n)); // Length of the name.
        if (l == len &&
            strncmp(s, qstring(n), len) == 0)
            return a;                  // Existing symbol found.
        w = qcdr(w);
    }
// here the symbol as required was not already present.
    if (!createp) return undefined;
    pn = makestring(s, len);
    push(pn);
    w = allocatesymbol();
    pop(pn);
    qflags(w) = tagHDR + typeSYM;
    qvalue(w) = undefined;
    qplist(w) = nil;
    qpname(w) = pn;
    qdefn(w)  = NULL;
    qlits(w)  = nil;
    push(w);
    obhash[hash] = cons(w, obhash[hash]);
    pop(w);
    return w;
}
```

Chapter 13

Evaluation

```
#define qflags(x) (((LispObject *)((x)-tagSYMBOL))[0])
#define qvalue(x) (((LispObject *)((x)-tagSYMBOL))[1])
#define qplist(x) (((LispObject *)((x)-tagSYMBOL))[2])
#define qpname(x) (((LispObject *)((x)-tagSYMBOL))[3])
#define qdefn(x)  (((void **)     ((x)-tagSYMBOL))[4])
#define qlits(x)  (((LispObject *)((x)-tagSYMBOL))[5])

// Bits within the flags field of a symbol. Uses explained later on.

#define flagTRACED    0x080
#define flagSPECFORM  0x100
#define flagMACRO     0x200

int unwindflag = 0;

#define unwindNONE      0
#define unwindERROR     1
#define unwindBACKTRACE 2
#define unwindGO        3
#define unwindRETURN    4

int backtraceflag = -1;
#define backtraceHEADER 1
#define backtraceTRACE  2

LispObject error1(const char *msg, LispObject data)
{   if ((backtraceflag & backtraceHEADER) != 0)
    {   linepos = printf("\n+++ Error: %s: ", msg);
        errprint(data);
    }
```

```
    unwindflag = (backtraceflag & backtraceTRACE) != 0 ? unwindBACKTRACE :
                 unwindERROR;
    return nil;
}

typedef LispObject specialform(LispObject data, LispObject x);
typedef LispObject lispfn(LispObject data, int nargs, ...);

LispObject applytostack(int n)
{
// Apply a function to n arguments.
// Here the stack has first the function, and then n arguments. The code is
// grim and basically repetitive, and to avoid it being even worse I will
// expect that almost all Lisp functions have at most 4 arguments, so
// if there are more than that I will pass the fifth and beyond all in a list.
    LispObject f, w;
    int traced = (qflags(sp[-n-1]) & flagTRACED) != 0;
    if (traced)
    {   int i;
        linepos = printf("Calling: ");
        errprint(sp[-n-1]);
        for (i=1; i<=n; i++)
        {   linepos = printf("Arg%d: ", i);
            errprint(sp[i-n-1]);
        }
    }
    if (n >= 5)
    {   push(nil);
        n++;
        while (n > 5)
        {   pop(w);
            TOS = cons(TOS, w);
            n--;
        }
    }
    switch (n)
    {   case 0: f = TOS;
            w = (*(lispfn *)qdefn(f))(qlits(f), 0);
            break;
        case 1:
        {   LispObject a1;
            pop(a1);
            f = TOS;
```

```
    w = (*(lispfn *)qdefn(f))(qlits(f), 1, a1);
    break;
}
case 2:
{   LispObject a1, a2;
    pop(a2)
    pop(a1);
    f = TOS;
    w = (*(lispfn *)qdefn(f))(qlits(f), 2, a1, a2);
    break;
}
case 3:
{   LispObject a1, a2, a3;
    pop(a3);
    pop(a2)
    pop(a1);
    f = TOS;
    w = (*(lispfn *)qdefn(f))(qlits(f), 3, a1, a2, a3);
    break;
}
case 4:
{   LispObject a1, a2, a3, a4;
    pop(a4);
    pop(a3);
    pop(a2)
    pop(a1);
    f = TOS;
    w = (*(lispfn *)qdefn(f))(qlits(f), 4,
                              a1, a2, a3, a4);
    break;
}
case 5:
{   LispObject a1, a2, a3, a4, a5andup;
    pop(a5andup);
    pop(a4);
    pop(a3);
    pop(a2)
    pop(a1);
    f = TOS;
    w = (*(lispfn *)qdefn(f))(qlits(f), 5,
                              a1, a2, a3, a4, a5andup);
    break;
}
```

```
        default:disaster(__LINE__);
            return nil;
    }
    pop(f);
    if (unwindflag == unwindBACKTRACE)
    {   linepos = printf("Calling: ");
        errprint(f);
    }
    else if (traced)
    {   push(w);
        prin(f);
        linepos += printf(" = ");
        errprint(w);
        pop(w);
    }
    return w;
}

LispObject interpret(LispObject def, int nargs, ...);
LispObject Lgensym(LispObject lits, int nargs, ...);

LispObject eval(LispObject x)
{   while (isCONS(x) && isSYMBOL(qcar(x)) && (qflags(qcar(x)) & flagMACRO))
    {   push2(qcar(x), x);
        x = applytostack(1);  // Macroexpand before anything else.
        if (unwindflag != unwindNONE) return nil;
    }
    if (isSYMBOL(x))
    {   LispObject v = qvalue(x);
        if (v == undefined) return error1("undefined variable", x);
        else return v;
    }
    else if (!isCONS(x)) return x;
// Now I have something of the form
//     (f arg1 ... argn)
// to process.
    {   LispObject f = qcar(x);
        if (isSYMBOL(f))
        {   LispObject flags = qflags(f), aa, av;
            int i, n = 0;
            if (flags & flagSPECFORM)
            {   specialform *fn = (specialform *)qdefn(f);
                return (*fn)(qlits(f), qcdr(x));
```

```
            }
// ... else not a special form...
            if (qdefn(f) == NULL) return error1("undefined function", f);
            aa = qcdr(x);
            while (isCONS(aa))
            {   n++;             // Count number of args supplied.
                aa = qcdr(aa);
            }
            aa = qcdr(x);
            push(f);
// Here I will evaluate all the arguments for the function, leaving the
// evaluated results on the stack.
            for (i=0; i<n; i++)
            {   push(aa);
                av = eval(qcar(aa));
                if (unwindflag != unwindNONE)
                {   while (i != 0)  // Restore the stack if unwinding.
                    {   pop(aa);
                        i--;
                    }
                    pop2(aa, aa);
                    return nil;
                }
                aa = qcdr(TOS);
                TOS = av;
            }
            return applytostack(n);
        }
        else if (isCONS(f) && qcar(f) == lambda)
        {   LispObject w;
            push(x);
            w = Lgensym(nil, 0);
            pop(x);
            qdefn(w) = (void *)interpret;
            qlits(w) = qcdr(qcar(x));
            return eval(cons(w, qcdr(x)));
        }
        else return error1("invalid function", f);
    }
}

LispObject interpretspecform(LispObject lits, LispObject x)
{   // lits should be ((var) body...)
```

```
    LispObject v;
    if (!isCONS(lits)) return nil;
    v = qcar(lits);
    lits = qcdr(lits);
    if (!isCONS(v) || !isSYMBOL(v = qcar(v))) return nil;
    push2(qvalue(v), v);
    qvalue(v) = x;
    lits = Lprogn(nil, lits);
    pop2(v, qvalue(v));
    return lits;
}

// Special forms are things that do not have their arguments pre-evaluated.

LispObject Lquote(LispObject lits, LispObject x)
{   if (isCONS(x)) return qcar(x);
    else return nil;
}

LispObject Lcond(LispObject lits, LispObject x)
{
//   Arg is in form
//      ((predicate1 val1a val1b ...)
//       (predicate2 val2a val2b ...)
//       ...)
    while (isCONS(x))
    {   push(x);
        x = qcar(x);
        if (isCONS(x))
        {   LispObject p = eval(qcar(x));
            if (unwindflag != unwindNONE)
            {   pop(x);
                return nil;
            }
            else if (p != nil)
            {   pop(x);
                return Lprogn(nil, qcdr(qcar(x)));
            }
        }
        pop(x);
        x = qcdr(x);
    }
    return nil;
```

```
}

#define ARG0(name)              \
    if (nargs != 0) return error1s("wrong number of arguments for", name)

#define ARG1(name, x)           \
    va_list a;                  \
    LispObject x;               \
    if (nargs != 1) return error1s("wrong number of arguments for", name); \
    va_start(a, nargs);         \
    x = va_arg(a, LispObject);  \
    va_end(a)

#define ARG2(name, x, y)        \
    va_list a;                  \
    LispObject x, y;            \
    if (nargs != 2) return error1s("wrong number of arguments for", name); \
    va_start(a, nargs);         \
    x = va_arg(a, LispObject);  \
    y = va_arg(a, LispObject);  \
    va_end(a)

#define ARG3(name, x, y, z)     \
    va_list a;                  \
    LispObject x, y, z;         \
    if (nargs != 3) return error1s("wrong number of arguments for", name); \
    va_start(a, nargs);         \
    x = va_arg(a, LispObject);  \
    y = va_arg(a, LispObject);  \
    z = va_arg(a, LispObject);  \
    va_end(a)

#define ARG0123(name, x, y, z)                          \
    va_list a;                                          \
    LispObject x=NULLATOM, y=NULLATOM, z=NULLATOM;      \
    if (nargs > 3) return error1s("wrong number of arguments for", name); \
    va_start(a, nargs);                                 \
    if (nargs > 0) x = va_arg(a, LispObject);           \
    if (nargs > 1) y = va_arg(a, LispObject);           \
    if (nargs > 2) z = va_arg(a, LispObject);           \
    va_end(a)
```

```
LispObject Lcar(LispObject lits, int nargs, ...)
{   ARG1("car", x);  // Note that this WILL take car of a bignum!
    if (isCONS(x)) return qcar(x);
    else return error1("car of an atom", x);
}



#define qfixnum(x)      ((x) >> 3)
#define packfixnum(n)   ((((LispObject)(n)) << 3) + tagFIXNUM)

#define MIN_FIXNUM      qfixnum(INTPTR_MIN)
#define MAX_FIXNUM      qfixnum(INTPTR_MAX)

#define qfloat(x)       (((double *)((x)-tagFLOAT))[0])

#define isBIGNUM(x) (isATOM(x) && ((qheader(x) & TYPEBITS) == typeBIGNUM))
#define qint64(x) (*(int64_t *)((x) - tagATOM + 8))


LispObject boxint64(int64_t a)
{   LispObject r = allocateatom(16);
    qheader(r) = tagHDR + typeBIGNUM + packlength(16);
    qint64(r) = a;
    return r;
}

// I will try to have a general macro that will help me with bringing
// everything to consistent numeric types - ie I can start off with a
// mix of fixnums, bignums and floats. The strategy here is that if either
// args is a float then the other is forced to that, and then for all sorts
// of pure integer work everything will be done as int64_t

#define NUMOP(name, a, b)                                                \
    if (isFLOAT(a))                                                      \
    {   if (isFLOAT(b)) return FF(qfloat(a), qfloat(b));                 \
        else if (isFIXNUM(b)) return FF(qfloat(a), (double)qfixnum(b));  \
        else if (isBIGNUM(b)) return FF(qfloat(a), (double)qint64(b));   \
        else return error1("Bad argument for " name, b);                 \
    }                                                                    \
    else if (isBIGNUM(a))                                                \
    {   if (isFLOAT(b)) return FF((double)qint64(a), qfloat(b));         \
        else if (isFIXNUM(b)) return BB(qint64(a), (int64_t)qfixnum(b)); \
        else if (isBIGNUM(b)) return BB(qint64(a), qint64(b));           \
```

```
        else return error1("Bad argument for " name, b);                \
    }                                                                   \
    else if (isFIXNUM(a))                                               \
    {   if (isFLOAT(b)) return FF((double)qfixnum(a), qfloat(b));       \
        else if (isFIXNUM(b)) return BB((int64_t)qfixnum(a),            \
                                        (int64_t)qfixnum(b));           \
        else if (isBIGNUM(b)) return BB((int64_t)qfixnum(a), qint64(b)); \
        else return error1("Bad argument for " name, b);                \
    }                                                                   \
    else return error1("Bad argument for " name, a)

#define UNARYOP(name, a)                                                \
    if (isFIXNUM(a)) return BB((int64_t)qfixnum(a));                    \
    else if (isFLOAT(a)) return FF(qfloat(a));                          \
    else if (isBIGNUM(a)) return BB(qint64(a));                         \
    else return error1("Bad argument for " name, a)

// Similar, but only supporting integer (not floating point) values

#define INTOP(name, a, b)                                               \
    if (isBIGNUM(a))                                                    \
    {   if (isFIXNUM(b)) return BB(qint64(a), (int64_t)qfixnum(b));     \
        else if (isBIGNUM(b)) return BB(qint64(a), qint64(b));          \
        else return error1("Bad argument for " name, b);                \
    }                                                                   \
    else if (isFIXNUM(a))                                               \
    {   if (isFIXNUM(b)) return BB((int64_t)qfixnum(a),                 \
                                   (int64_t)qfixnum(b));                \
        else if (isBIGNUM(b)) return BB((int64_t)qfixnum(a), qint64(b)); \
        else return error1("Bad argument for " name, b);                \
    }                                                                   \
    else return error1("Bad argument for " name, a)

#define UNARYINTOP(name, a)                                             \
    if (isFIXNUM(a)) return BB((int64_t)qfixnum(a));                    \
    else if (isBIGNUM(a)) return BB(qint64(a));                         \
    else return error1("Bad argument for " name, a)

// This takes an arbitrary 64-bit integer and returns either a fixnum
// or a bignum as necessary.

LispObject makeinteger(int64_t a)
{   if (a >= MIN_FIXNUM && a <= MAX_FIXNUM) return packfixnum(a);
```

```
    else return boxint64(a);
}

#undef FF
#undef BB
#define FF(a) boxfloat(-(a))
#define BB(a) makeinteger(-(a))

LispObject Nminus(LispObject a)
{   UNARYOP("minus", a);
}

#undef FF
#undef BB
#define FF(a, b) boxfloat((a) + (b))
#define BB(a, b) makeinteger((a) + (b))

LispObject Nplus2(LispObject a, LispObject b)
{   NUMOP("plus", a, b);
}

#undef FF
#undef BB
#define FF(a, b) boxfloat((a) * (b))
#define BB(a, b) makeinteger((a) * (b))

LispObject Ntimes2(LispObject a, LispObject b)
{   NUMOP("times", a, b);
}

#undef BB
#define BB(a, b) makeinteger((a) & (b))

LispObject Nlogand2(LispObject a, LispObject b)
{   INTOP("logand", a, b);
}

#undef BB
#define BB(a, b) makeinteger((a) | (b))

LispObject Nlogor2(LispObject a, LispObject b)
{   INTOP("logor", a, b);
}
```

```
#undef BB
#define BB(a, b) makeinteger((a) ^ (b))

LispObject Nlogxor2(LispObject a, LispObject b)
{   INTOP("logxor", a, b);
}

#undef FF
#undef BB


#define NARY(x, base, combinefn)      \
    LispObject r;                     \
    if (!isCONS(x)) return base;      \
    push(x);                          \
    r = eval(qcar(x));                \
    pop(x);                           \
    if (unwindflag != unwindNONE)     \
        return nil;                   \
    x = qcdr(x);                      \
    while (isCONS(x))                 \
    {   LispObject a;                 \
        push2(x, r);                  \
        a = eval(qcar(x));            \
        if (unwindflag != unwindNONE) \
        {   pop2(r, x);               \
            return nil;               \
        }                             \
        pop(r);                       \
        r = combinefn(r, a);          \
        pop(x);                       \
        x = qcdr(x);                  \
    }                                 \
    return r

LispObject Lplus(LispObject lits, LispObject x)
{   NARY(x, packfixnum(0), Nplus2);
}

LispObject Ltimes(LispObject lits, LispObject x)
{   NARY(x, packfixnum(1), Ntimes2);
}
```

```
LispObject Llogand(LispObject lits, LispObject x)
{   NARY(x, packfixnum(-1), Nlogand2);
}

LispObject Llogor(LispObject lits, LispObject x)
{   NARY(x, packfixnum(0), Nlogor2);
}

LispObject Llogxor(LispObject lits, LispObject x)
{   NARY(x, packfixnum(0), Nlogxor2);
}

LispObject Lnumberp(LispObject lits, int nargs, ...)
{   ARG1("numberp", x);
    return (isFIXNUM(x) || isBIGNUM(x) || isFLOAT(x) ? lisptrue : nil);
}

LispObject Lfixp(LispObject lits, int nargs, ...)
{   ARG1("fixp", x);
    return (isFIXNUM(x) || isBIGNUM(x) ? lisptrue : nil);
}

LispObject Lfloatp(LispObject lits, int nargs, ...)
{   ARG1("floatp", x);
    return (isFLOAT(x) ? lisptrue : nil);
}

LispObject Lfix(LispObject lits, int nargs, ...)
{   ARG1("fix", x);
    return (isFIXNUM(x) || isBIGNUM(x) ? x :
            isFLOAT(x) ? boxint64((int64_t)qfloat(x)) :
            error1("arg for fix", x));
}

LispObject Lfloor(LispObject lits, int nargs, ...)
{   ARG1("floor", x);
    return (isFIXNUM(x) || isBIGNUM(x) ? x :
            isFLOAT(x) ? boxint64((int64_t)floor(qfloat(x))) :
            error1("arg for floor", x));
}

LispObject Lceiling(LispObject lits, int nargs, ...)
```

```
{   ARG1("ceiling", x);
    return (isFIXNUM(x) || isBIGNUM(x) ? x :
            isFLOAT(x) ? boxint64((int64_t)ceil(qfloat(x))) :
            error1("arg for ceiling", x));
}

LispObject Lfloat(LispObject lits, int nargs, ...)
{   ARG1("float", x);
    return (isFLOAT(x) ? x :
            isFIXNUM(x) ? boxfloat((double)qfixnum(x)) :
            isBIGNUM(x) ? boxfloat((double)qint64(x)) :
            error1("arg for fix", x));
}

#define floatval(x)                    \
   isFLOAT(x) ? qfloat(x) :            \
   isFIXNUM(x) ? (double)qfixnum(x) : \
   isBIGNUM(x) ? (double)qint64(x) :  \
   0.0

LispObject Lcos(LispObject lits, int nargs, ...)
{   ARG1("cos", x);
    return boxfloat(cos(floatval(x)));
}

LispObject Lsin(LispObject lits, int nargs, ...)
{   ARG1("sin", x);
    return boxfloat(sin(floatval(x)));
}

LispObject Lsqrt(LispObject lits, int nargs, ...)
{   ARG1("sqrt", x);
    return boxfloat(sqrt(floatval(x)));
}

LispObject Llog(LispObject lits, int nargs, ...)
{   ARG1("log", x);
    return boxfloat(log(floatval(x)));
}

LispObject Lexp(LispObject lits, int nargs, ...)
{   ARG1("exp", x);
    return boxfloat(exp(floatval(x)));
```

```
}

LispObject Latan(LispObject lits, int nargs, ...)
{   ARG1("atan", x);
    return boxfloat(atan(floatval(x)));
}


// Now some numeric functions

#undef FF
#undef BB
#define FF(a, b) ((a) > (b) ? lisptrue : nil)
#define BB(a, b) ((a) > (b) ? lisptrue : nil)

LispObject Lgreaterp(LispObject lits, int nargs, ...)
{   ARG2("greaterp", x, y);
    NUMOP("greaterp", x, y);
}

#undef FF
#undef BB
#define FF(a, b) ((a) >= (b) ? lisptrue : nil)
#define BB(a, b) ((a) >= (b) ? lisptrue : nil)

LispObject Lgeq(LispObject lits, int nargs, ...)
{   ARG2("geq", x, y);
    NUMOP("geq", x, y);
}

#undef FF
#undef BB
#define FF(a, b) ((a) < (b) ? lisptrue : nil)
#define BB(a, b) ((a) < (b) ? lisptrue : nil)

LispObject Llessp(LispObject lits, int nargs, ...)
{   ARG2("lessp", x, y);
    NUMOP("lessp", x, y);
}

#undef FF
#undef BB
#define FF(a, b) ((a) <= (b) ? lisptrue : nil)
```

```
#define BB(a, b) ((a) <= (b) ? lisptrue : nil)

LispObject Lleq(LispObject lits, int nargs, ...)
{   ARG2("leq", x, y);
    NUMOP("leq", x, y);
}

LispObject Lminus(LispObject lits, int nargs, ...)
{   ARG1("minus", x);
    return Nminus(x);
}

LispObject Lminusp(LispObject lits, int nargs, ...)
{   ARG1("minusp", x);
// Anything non-numeric will not be negative!
    if ((isFIXNUM(x) && x < 0) ||
        (isFLOAT(x) && qfloat(x) < 0.0) ||
        (isATOM(x) &&
         (qheader(x) & TYPEBITS) == typeBIGNUM &&
         qint64(x) < 0)) return lisptrue;
    else return nil;
}

#undef BB
#define BB(a) makeinteger(~(a))

LispObject Llognot(LispObject lits, int nargs, ...)
{   ARG1("lognot", x);
    UNARYINTOP("lognot", x);
}

LispObject Lzerop(LispObject lits, int nargs, ...)
{   ARG1("zerop", x);
// Note that a bignum can never be zero! Because that is not "big".
// This code is generous and anything non-numeric is not zero.
    if (x == packfixnum(0) ||
        (isFLOAT(x) && qfloat(x) == 0.0)) return lisptrue;
    else return nil;
}

LispObject Lonep(LispObject lits, int nargs, ...)
{   ARG1("onep", x);
    if (x == packfixnum(1) ||
```

```
        (isFLOAT(x) && qfloat(x) == 1.0)) return lisptrue;
    else return nil;
}

#undef FF
#undef BB
#define FF(a) boxfloat((a) + 1.0)
#define BB(a) makeinteger((a) + 1)

LispObject Ladd1(LispObject lits, int nargs, ...)
{   ARG1("add1", x);
    UNARYOP("add1", x);
}

#undef FF
#undef BB
#define FF(a) boxfloat((a) - 1.0)
#define BB(a) makeinteger((a) - 1)

LispObject Lsub1(LispObject lits, int nargs, ...)
{   ARG1("sub1", x);
    UNARYOP("sub1", x);
}

#undef FF
#undef BB
#define FF(a, b) boxfloat((a) - (b))
#define BB(a, b) makeinteger((a) - (b))

LispObject Ldifference(LispObject lits, int nargs, ...)
{   ARG2("difference", x, y);
    NUMOP("difference", x, y);
}

#undef FF
#undef BB
#define FF(a, b) ((b) == 0.0 ? error1("division by 0.0", nil) : \
                  boxfloat((a) / (b)))
#define BB(a, b) ((b) == 0 ? error1("division by 0", nil) : \
                  makeinteger((a) / (b)))

LispObject Lquotient(LispObject lits, int nargs, ...)
{   ARG2("quotient", x, y);
```

```
    NUMOP("quotient", x, y);
}

#undef BB
#define BB(a, b) ((b) == 0 ? error1("remainder by 0", nil) : \
                  makeinteger((a) % (b)))

LispObject Lremainder(LispObject lits, int nargs, ...)
{   ARG2("remainder", x, y);
    INTOP("remainder", x, y);
}

#undef BB
#define BB(a, b) ((b) == 0 ? error1("division by 0", nil) : \
                  cons(makeinteger((a) / (b)), makeinteger((a) % (b))))

LispObject Ldivide(LispObject lits, int nargs, ...)
{   ARG2("divide", x, y);
    INTOP("divide", x, y);
}

#undef BB
#define BB(a) makeinteger((a) << sh)

LispObject Lleftshift(LispObject lits, int nargs, ...)
{   int sh;
    ARG2("leftshift", x, y);
    if (!isFIXNUM(y)) return error1("Bad argument for leftshift", y);
    sh = (int)qfixnum(y);
    UNARYINTOP("leftshift", x);
}

#undef BB
#define BB(a) makeinteger((a) >> sh)

LispObject Lrightshift(LispObject lits, int nargs, ...)
{   int sh;
    ARG2("rightshift", x, y);
    if (!isFIXNUM(y)) return error1("Bad argument for rightshift", y);
    sh = (int)qfixnum(y);
    UNARYINTOP("rightshift", x);
}
```

Chapter 14

Storage management and Garbage Collection

```
void allocateheap()
{   void *pool = (void *)
        malloc(ROUNDED_HEAPSIZE+BITMAPSIZE+ROUNDED_STACKSIZE+16);
    if (pool == NULL)
    {   printf("Not enough memory available: Unable to proceed\n");
        exit(1);
    }
    heap1base = (LispObject)pool;
    heap1base = (heap1base + 7) & ~7; // ensure alignment
    heap1top = heap2base = heap1base + (ROUNDED_HEAPSIZE/2);
    heap2top = heap2base + (ROUNDED_HEAPSIZE/2);
    fringe1 = heap1base;
    fpfringe1 = heap1top;
    fringe2 = heap2base;
    fpfringe2 = heap2top;
    stackbase = heap2top;
    stacktop = stackbase + ROUNDED_STACKSIZE;
    bitmap = stacktop;
}

extern void reclaim();

LispObject cons(LispObject a, LispObject b)
{
    if (fringe1 >= fpfringe1)
    {   push2(a, b);
        reclaim();
```

```
        pop2(b, a);
    }
    qcar(fringe1) = a;
    qcdr(fringe1) = b;
    a = fringe1;
    fringe1 += 2*sizeof(LispObject);
    return a;
}


#define swap(a,b) w = (a); (a) = (b); (b) = w;

extern LispObject copy(LispObject x);

int gccount = 1;

void reclaim()
{
// The strategy here is due to C J Cheyney ("A Nonrecursive List Compacting
// Algorithm". Communications of the ACM 13 (11): 677-678, 1970).
    LispObject *s, w;
    printf("+++ GC number %d", gccount++);
// I need to clear the part of the bitmap that could be relevant for floating
// point values.
    int o = (fpfringe1 - heap1base)/(8*8);
    while (o < BITMAPSIZE) ((unsigned char *)bitmap)[o++] = 0;
// Process everything that is on the stack.
    for (s=(LispObject *)stackbase; s<sp; s++) *s = copy(*s);
// I should also copy any other list base values here.
    for (o=0; o<BASES_SIZE; o++) bases[o] = copy(bases[o]);
    for (o=0; o<OBHASH_SIZE; o++)
        obhash[o] = copy(obhash[o]);
// Now perform the second part of Cheyney's algorithm, scanning the
// data that has been put in the new heap.
    s = (LispObject *)heap2base;
    while ((LispObject)s != fringe2)
    {   LispObject h = *s;
        if (!isHDR(h)) // The item to be processed is a simple cons cell
        {   *s++ = copy(h);
            *s = copy(*s);
            s++;
        }
        else              // The item is one that uses a header
```

```
            switch (h & TYPEBITS)
            {   case typeSYM:
                    w = ((LispObject)s) + tagSYMBOL;
                    // qflags(w) does not need adjusting
                    qvalue(w) = copy(qvalue(w));
                    qplist(w) = copy(qplist(w));
                    qpname(w) = copy(qpname(w));
                    // qdefn(w) does not need adjusting
                    qlits(w)  = copy(qlits(w));
                    s += 6;
                    continue;
                case typeSTRING:
                case typeBIGNUM:
// These only contain binary information, so none of their content needs
// any more processing.
                    w = (sizeof(LispObject) + veclength(h) + 7) & ~7;
                    s += w/sizeof(LispObject);
                    continue;
                case typeVEC:
                case typeEQHASH:
                case typeEQHASHX:
// These are to be processed the same way. They contain a bunch of
// reference items.
                    s++; // Past the header
                    w = veclength(h);
                    while (w > 0)
                    {   *s = copy(*s);
                        s++;
                        w -= sizeof(LispObject);
                    }
                    w = (LispObject)s;
                    w = (w + 7) & ~7;
                    s = (LispObject *)w;
                    continue;
                default:
                    // all the "spare" codes!
                    disaster(__LINE__);
            }
    }
// Finally flip the two heaps ready for next time.
    swap(heap1base, heap2base);
    swap(heap1top, heap2top);
    fringe1 = fringe2;
```

```
    fpfringe1 = fpfringe2;
    fringe2 = heap2base;
    fpfringe2 = heap2top;
    if (fpfringe1 - fringe1 < 1000*sizeof(LispObject))
    {   printf("\nRun out of memory.\n");
        exit(1);
    }
    printf(" - collection complete\n");
    fflush(stdout);
}

LispObject copy(LispObject x)
{   LispObject h;
    int o, b;
    switch (x & TAGBITS)
    {   case tagCONS:
            if (x == 0) disaster(__LINE__);
            h = *((LispObject *)x);
            if (isFORWARD(h)) return (h - tagFORWARD);
            qcar(fringe2) = h;
            qcdr(fringe2) = qcdr(x);
            h = fringe2;
            qcar(x) = tagFORWARD + h;
            fringe2 += 2*sizeof(LispObject);
            return h;
        case tagSYMBOL:
            h = *((LispObject *)(x - tagSYMBOL));
            if (isFORWARD(h)) return (h - tagFORWARD + tagSYMBOL);
            if (!isHDR(h)) disaster(__LINE__);
            h = fringe2 + tagSYMBOL;
            qflags(h) = qflags(x);
            qvalue(h) = qvalue(x);
            qplist(h) = qplist(x);
            qpname(h) = qpname(x);
            qdefn(h)  = qdefn(x);
            qlits(h)  = qlits(x);
            fringe2 += 6*sizeof(LispObject);
            qflags(x) = h - tagSYMBOL + tagFORWARD;
            return h;
        case tagATOM:
            if (x == NULLATOM) return x; // special case!
            h = qheader(x);
            if (isFORWARD(h)) return (h - tagFORWARD + tagATOM);
```

```
            if (!isHDR(h)) disaster(__LINE__);
            switch (h & TYPEBITS)
            {   case typeEQHASH:
// When a hash table is copied its header is changes to EQHASHX, which
// indicates that it will need rehashing before further use.
                    h ^= (typeEQHASH ^ typeEQHASHX);
                case typeEQHASHX:
                case typeSTRING:
                case typeVEC:
                case typeBIGNUM:
                    o = (int)veclength(h);  // number of bytes excluding the hea
                    *((LispObject *)fringe2) = h; // copy header word across
                    h = fringe2 + tagATOM;
                    *((LispObject *)(x - tagATOM)) = fringe2 + tagFORWARD;
                    fringe2 += sizeof(LispObject);
                    x = x - tagATOM + sizeof(LispObject);
                    while (o > 0)
                    {   *((LispObject *)fringe2) = *((LispObject *)x);
                        fringe2 += sizeof(LispObject);
                        x += sizeof(LispObject);
                        o -= sizeof(LispObject);
                    }
                    fringe2 = (fringe2 + 7) & ~7;
                    return h;
                default:
                    //case typeSYM:
                    // also the spare codes!
                    disaster(__LINE__);
            }
        case tagFLOAT:
// every float is 8 bytes wide, regardless of what sort of machine I am on.
            h = (x - tagFLOAT - heap1base)/8;
            o = h/8;
            b = 1 << (h%8);
// now o is an offset and b a bit in the bitmap.
            if ((((unsigned char *)bitmap)[o] & b) != 0) // marked already.
                return *((LispObject *)(x-tagFLOAT));
            else
            {   ((unsigned char *)bitmap)[o] |= b; // mark it now.
                fpfringe2 -= sizeof(double);
                h = fpfringe2 + tagFLOAT;
                qfloat(h) = qfloat(x);              // copy the float.
                *((LispObject *)(x-tagFLOAT)) = h; // write in forwarding addres
```

```
                return h;
            }
        case tagFIXNUM:
            return x;
        default:
//case tagFORWARD:
//case tagHDR:
            disaster(__LINE__);
            return 0;  // avoid GCC moans.
    }
}
```

Chapter 15

Checkpoint and restore

```
// A saved image will start with a word that contains the following 32-bit
// code. This can identify the byte-ordering and word-width of the system
// involved, so protects against attempts to reload an image on a machine
// other than the one it was build on. If I was being more proper I would
// include a version number as well.

#define FILEID (('v' << 0) | ('s' << 8) | ('1' << 16) |   \
                ('0' + sizeof(LispObject)) << 24))

static const char *imagename = "vsl.img";

LispObject Lpreserve(LispObject lits, int nargs, ...)
{
// preserve can take either 0 or 1 args. If it has a (non-nil) arg that will
// be a startup function for the image when restored.
    FILE *f;
    ARG0123("preserve", x,y,z);
    if (y != NULLATOM || z != NULLATOM)
        return error1s("wrong number of arguments for", "preserve");
    restartfn = (x == NULLATOM ? nil : x);
    f = fopen(imagename, "wb");
    if (f == NULL) return error1("unable to open image for writing", nil);
    headerword = FILEID;
    reclaim(); // To compact memory.
// I write this stuff out as a bunch of bytes, since I only intend to
// re-read it on exactly the same computer.
    saveinterp = (LispObject)(void *)interpret;
    saveinterpspec = (LispObject)(void *)interpretspecform;
    fwrite(nonbases, 1, sizeof(nonbases), f);
```

```
    fwrite(bases, 1, sizeof(bases), f);
    fwrite(obhash, 1, sizeof(obhash), f);
    fwrite((void *)heap1base, 1, (size_t)(fringe1-heap1base), f);
    fwrite((void *)fpfringe1, 1, (size_t)(heap1top-fpfringe1), f);
    fclose(f);
// A cautious person would have checked for error codes returned by the
// above calls to fwrite and close. I omit that here to be concise.
    return nil;
}

jmp_buf restart_buffer;
int coldstart = 0;

LispObject Lrestart_csl(LispObject lits, int nargs, ...)
{   LispObject save = lispout;
    ARG0123("restart-csl", x, y, z);
    if (z != NULLATOM)
        return error1s("wrong number of arguments for", "restart-csl");
    coldstart = 0;
    if (x == nil || x == NULLATOM) coldstart = 1, x = nil;
    if (y == NULLATOM) x = cons(x, nil);
    else x = list2star(x, y, nil);
    boffop = 0;
    lispout = -2;
    prin(x);
    lispout = save;
    longjmp(restart_buffer, 1);
}


struct defined_functions
{   const char *name;
    int flags;
    void *entrypoint;
};

struct defined_functions fnsetup[] =
{
// First the special forms
    {"quote",      flagSPECFORM, (void *)Lquote},
    {"cond",       flagSPECFORM, (void *)Lcond},
    {"and",        flagSPECFORM, (void *)Land},
    {"or",         flagSPECFORM, (void *)Lor},
```

```
    {"de",          flagSPECFORM, (void *)Lde},
    {"df",          flagSPECFORM, (void *)Ldf},
    {"dm",          flagSPECFORM, (void *)Ldm},
    {"setq",        flagSPECFORM, (void *)Lsetq},
    {"progn",       flagSPECFORM, (void *)Lprogn},
    {"prog",        flagSPECFORM, (void *)Lprog},
    {"go",          flagSPECFORM, (void *)Lgo},
// The following are implemented as special forms here because they
// take variable or arbitrary numbers of arguments - however they all
// evaluate all their arguments in a rather simple way, so they
// could be treated a sorts of "ordinary" function.
    {"list",        flagSPECFORM, (void *)Llist},
    {"list*",       flagSPECFORM, (void *)Lliststar},
    {"iplus",       flagSPECFORM, (void *)Lplus},
    {"itimes",      flagSPECFORM, (void *)Ltimes},
    {"ilogand",     flagSPECFORM, (void *)Llogand},
    {"ilogor",      flagSPECFORM, (void *)Llogor},
    {"ilogxor",     flagSPECFORM, (void *)Llogxor},
// Now ordinary functions. I have put these in alphabetic order.
    {"iadd1",       0,            (void *)Ladd1},
    {"apply",       0,            (void *)Lapply},
    {"atan",        0,            (void *)Latan},
    {"atom",        0,            (void *)Latom},
    {"bignump",     0,            (void *)Lbignump},
    {"boundp",      0,            (void *)Lboundp},
    {"car",         0,            (void *)Lcar},
    {"cdr",         0,            (void *)Lcdr},
    {"char-code",   0,            (void *)Lcharcode},
    {"iceiling",    0,            (void *)Lceiling},
    {"close",       0,            (void *)Lclose},
    {"code-char",   0,            (void *)Lcodechar},
    {"compress",    0,            (void *)Lcompress},
    {"cons",        0,            (void *)Lcons},
    {"cos",         0,            (void *)Lcos},
    {"idifference",0,             (void *)Ldifference},
    {"idivide",     0,            (void *)Ldivide},
    {"eq",          0,            (void *)Leq},
    {"equal",       0,            (void *)Lequal},
    {"error",       0,            (void *)Lerror},
    {"errorset",    0,            (void *)Lerrorset},
    {"eval",        0,            (void *)Leval},
    {"exp",         0,            (void *)Lexp},
    {"explode",     0,            (void *)Lexplode},
```

```
    {"explodec",   0,           (void *)Lexplodec},
    {"ifix",       0,           (void *)Lfix},
    {"ifixp",      0,           (void *)Lfixp},
    {"ifloat",     0,           (void *)Lfloat},
    {"floatp",     0,           (void *)Lfloatp},
    {"ifloor",     0,           (void *)Lfloor},
    {"gensym",     0,           (void *)Lgensym},
    {"igeq",       0,           (void *)Lgeq},
    {"get",        0,           (void *)Lget},
    {"getd",       0,           (void *)Lgetd},
    {"gethash",    0,           (void *)Lgethash},
    {"getv",       0,           (void *)Lgetv},
    {"igreaterp",  0,           (void *)Lgreaterp},
    {"ileftshift", 0,           (void *)Lleftshift},
    {"ileq",       0,           (void *)Lleq},
    {"ilessp",     0,           (void *)Llessp},
    {"load-module",0,           (void *)Lrdf},
    {"log",        0,           (void *)Llog},
    {"ilognot",    0,           (void *)Llognot},
    {"iminus",     0,           (void *)Lminus},
    {"iminusp",    0,           (void *)Lminusp},
    {"mkhash",     0,           (void *)Lmkhash},
    {"mkvect",     0,           (void *)Lmkvect},
    {"null",       0,           (void *)Lnull},
    {"inumberp",   0,           (void *)Lnumberp},
    {"oblist",     0,           (void *)Loblist},
    {"onep",       0,           (void *)Lonep},
    {"open",       0,           (void *)Lopen},
    {"plist",      0,           (void *)Lplist},
    {"preserve",   0,           (void *)Lpreserve},
    {"prin",       0,           (void *)Lprin},
    {"princ",      0,           (void *)Lprinc},
    {"print",      0,           (void *)Lprint},
    {"printc",     0,           (void *)Lprintc},
    {"put",        0,           (void *)Lput},
    {"puthash",    0,           (void *)Lputhash},
    {"putv",       0,           (void *)Lputv},
    {"iquotient",  0,           (void *)Lquotient},
    {"rdf",        0,           (void *)Lrdf},
    {"rds",        0,           (void *)Lrds},
    {"read",       0,           (void *)Lread},
    {"readch",     0,           (void *)Lreadch},
    {"readline",   0,           (void *)Lreadline},
```

```
    {"iremainder", 0,           (void *)Lremainder},
    {"remhash",    0,           (void *)Lremhash},
    {"remprop",    0,           (void *)Lremprop},
    {"restart-csl",0,           (void *)Lrestart_csl},
    {"return",     0,           (void *)Lreturn},
    {"irightshift",0,           (void *)Lrightshift},
    {"rplaca",     0,           (void *)Lrplaca},
    {"rplacd",     0,           (void *)Lrplacd},
    {"set",        0,           (void *)Lset},
    {"sin",        0,           (void *)Lsin},
    {"sqrt",       0,           (void *)Lsqrt},
    {"stop",       0,           (void *)Lstop},
    {"stringp",    0,           (void *)Lstringp},
    {"isub1",      0,           (void *)Lsub1},
    {"symbolp",    0,           (void *)Lsymbolp},
    {"terpri",     0,           (void *)Lterpri},
    {"time",       0,           (void *)Ltime},
    {"trace",      0,           (void *)Ltrace},
    {"untrace",    0,           (void *)Luntrace},
    {"upbv",       0,           (void *)Lupbv},
    {"vectorp",    0,           (void *)Lvectorp},
    {"wrs",        0,           (void *)Lwrs},
    {"zerop",      0,           (void *)Lzerop},
    {NULL,         0,           NULL}
};

void setup()
{
// Ensure that initial symbols and functions are in place. Parts of this
// code are rather rambling and repetitive but this is at least a simple
// way to do things. I am going to assume that nothing can fail within this
// setup code, so I can omit all checks for error conditions.
    struct defined_functions *p;
    undefined = lookup("~indefinite-value~", 18, 1);
    qvalue(undefined) = undefined;
    nil = lookup("nil", 3, 1);
    qvalue(nil) = nil;
    lisptrue = lookup("t", 1, 1);
    qvalue(lisptrue) = lisptrue;
    qvalue(echo = lookup("*echo", 5, 1)) = interactive ? nil : lisptrue;
    qvalue(lispsystem = lookup("lispsystem*", 11, 1)) =
       list2star(lookup("vsl", 3, 1), lookup("csl", 3, 1),
                 cons(lookup("embedded", 8, 1), nil));
```

```
    quote = lookup("quote", 5, 1);
    backquote = lookup("`", 1, 1);
    comma = lookup(",", 1, 1);
    comma_at = lookup(",@", 2, 1);
    eofsym = lookup("$eof$", 5, 1);
    qvalue(eofsym) = eofsym;
    lambda = lookup("lambda", 6, 1);
    expr = lookup("expr", 4, 1);
    subr = lookup("subr", 4, 1);
    fexpr = lookup("fexpr", 5, 1);
    fsubr = lookup("fsubr", 5, 1);
    macro = lookup("macro", 5, 1);
    input = lookup("input", 5, 1);
    output = lookup("output", 6, 1);
    pipe = lookup("pipe", 4, 1);
    qvalue(dfprint = lookup("dfprint*", 6, 1)) = nil;
    bignum = lookup("~bignum", 7, 1);
    qlits(lookup("load-module", 11, 1)) = lisptrue;
    qvalue(raise = lookup("*raise", 6, 1)) = nil;
    qvalue(lower = lookup("*lower", 6, 1)) = lisptrue;
    cursym = nil;
    work1 = work2 = nil;
    p = fnsetup;
    while (p->name != NULL)
    {   LispObject w = lookup(p->name, strlen(p->name), 1);
        qflags(w) |= p->flags;
        qdefn(w) = p->entrypoint;
        p++;
    }
}

void cold_setup()
{
// version of setup to call when there is no initial heap image at all.
    int i;
// I make the object-hash-table lists end in a fixnum rather than nil
// because I want to create the hash table before even the symbol nil
// exists.
    for (i=0; i<OBHASH_SIZE; i++) obhash[i] = tagFIXNUM;
    for (i=0; i<BASES_SIZE; i++) bases[i] = NULLATOM;
    setup();
// The following fields could not be set up quite early enough in the
// cold start case, so I repair them now.
```

```
    restartfn = qplist(undefined) = qlits(undefined) =
        qplist(nil) = qlits(nil) = nil;
}

LispObject relocate(LispObject a, LispObject change)
{
// Used to update a LispObject when reloaded from a saved heap image.
    switch (a & TAGBITS)
    {   case tagATOM:
           if (a == NULLATOM) return a;
        case tagCONS:
        case tagSYMBOL:
        case tagFLOAT:
            return a + change;
        default:
//case tagFIXNUM:
//case tagFORWARD:
//case tagHDR:
            return a;
    }
}

void warm_setup()
{
// The idea here is that a file called "vsl.img" will already have been
// created by a previous use of vsl, and it should be re-loaded.
    FILE *f = fopen(imagename, "rb");
    int i;
    LispObject currentbase = heap1base, change, *s;
    if (f == NULL)
    {   printf("Error: unable to open image for reading\n");
        exit(1);
    }
    if (fread(nonbases, 1, sizeof(nonbases), f) != sizeof(nonbases) ||
        headerword != FILEID ||
        fread(bases, 1, sizeof(bases), f) != sizeof(bases) ||
        fread(obhash, 1, sizeof(obhash), f) != sizeof(obhash))
    {   printf("Error: Image file corrupted or incompatible\n");
        exit(1);
    }
    change = currentbase - heap1base;
// Now I relocate the key addresses to refer to the CURRENT rather than
// the saved address map.
```

```
    heap1base  += change;
    heap1top   += change;
    fringe1    += change;
    fpfringe1  += change;
    if (fread((void *)heap1base, 1, (size_t)(fringe1-heap1base), f) !=
              (size_t)(fringe1-heap1base) ||
        fread((void *)fpfringe1, 1, (size_t)(heap1top-fpfringe1), f) !=
              (size_t)(heap1top-fpfringe1))
    {   printf("Error: Unable to read image file\n");
        exit(1);
    }
    fclose(f);
    for (i=0; i<BASES_SIZE; i++)
        bases[i] = relocate(bases[i], change);
    for (i=0; i<OBHASH_SIZE; i++)
        obhash[i] = relocate(obhash[i], change);
// The main heap now needs to be scanned and addresses in it corrected.
    s = (LispObject *)heap1base;
    while ((LispObject)s != fringe1)
    {   LispObject h = *s, w;
        if (!isHDR(h)) // The item to be processed is a simple cons cell
        {   *s++ = relocate(h, change);
            *s = relocate(*s, change);
            s++;
        }
        else              // The item is one that uses a header
            switch (h & TYPEBITS)
            {   case typeSYM:
                    w = ((LispObject)s) + tagSYMBOL;
                    // qflags(w) does not need adjusting
                    qvalue(w) = relocate(qvalue(w), change);
                    qplist(w) = relocate(qplist(w), change);
                    qpname(w) = relocate(qpname(w), change);
                    if (qdefn(w) == (void *)saveinterp)
                        qdefn(w) = (void *)interpret;
                    else if (qdefn(w) == (void *)saveinterpspec)
                        qdefn(w) = (void *)interpretspecform;
                    qlits(w)  = relocate(qlits(w), change);
                    s += 6;
                    continue;
                case typeSTRING: case typeBIGNUM:
// These sorts of atom just contain binary data so do not need adjusting,
// but I have to allow for the length code being in bytes etc.
```

```
                    w = (LispObject)s;
                    w += veclength(h);
                    w = (w + sizeof(LispObject) + 7) & ~7;
                    s = (LispObject *)w;
                    continue;
                case typeVEC: case typeEQHASH: case typeEQHASHX:
                    s++;
                    w = veclength(h);
                    while (w > 0)
                    {   *s = relocate(*s, change);
                        s++;
                        w -= sizeof(LispObject);
                    }
                    s = (LispObject *)(((LispObject)s + 7) & ~7);
                    continue;
                default:
                    // The spare codes!
                    disaster(__LINE__);
            }
    }
    setup(); // resets all built-in functions properly.
}
```

Chapter 16

Lisp Compilation

Chapter 17

Some implementation tricks and extensions

The main version of `vsl` emphasises simplicity over capability. This chapter discusses a couple of alternative implementation strategies that could extend it. I will refer to an extended version of `vsl` including the ideas discussed here as tx vslplus.

17.1 Conservative Garbage Collection

`vsl` has a garbage collector that requires that all references into the heap can be unambiguously identified. This constraint complicates the code, leads to the need for extensive use of a special Lisp stack and will make Lisp compilation much messier than if such a constraint did not apply. In 1988 Bartlett[6] and then Appell and Hanson[4] presented garbage collection schemes that used a copying strategy but did not impose this constraint. They both control the preservation of data that is subject to (possible) reference from an ambiguous location in units of pages and by so doing they reduce the overheap of tracking where such data is. However at the same time they may tend to waste a little space and (more seriously) set a limit on the largest size that an object can have (and fit within one page).

So although I will refer back to those important early papers I will want to arrange that I can cope with almost arbitrarily large vectors in my heap. However I will not feel the need to allow for pointers that refer within an object (ie to other than an object head) and I will view values that have the `vsl` tag-bits removed as valid. So in my first implementation at least I will accept that if the user has code that writes something like

```
LispObject a = ... // eg a string
```

```
... qstring(a) ...
```

and a compiler makes optimisations that compute and later re-use the untagged address that the `qstring` macro generates then there could be pain.

Well actually it will be easy for me to take any ambiguous value and discard its tag bits before checking if it might be a pointer into the heap, and for the benefit of `qstring` and `elt` view such a value as pinning the object whose address was just before it – or possibly scan back from it to the head of the previous object in memory and then untagged references even into the middle of arrays would become safe. That could make the code safer in the context of unduly optimising compilers and would probably not lead to verty much additional memory waste, so it becomes the new policy. Whenever I talk later down about checking if a reference is to the start of an object I have to mean scan back to the start of the previous object and treat the reference as pinning] that.

17.1.1 What are the ambiguous bases?

The key problem that a Conservative Garbage Collector addresses is support for a system where some pointers into the heap are stored in locations where one can not be certain that they contain valid data. This could include cases where pointer-containing memory is interleaved with memory containing arbitrary other material and where it is not feasible to maintain maps or table showing exactly what is where. For a mark-and-sweep garbage collection strategy all that would be needed to cope with this would be to have the mark process in the garbage collector mark from every location that could possibly contain a pointer. The code would have to be robust against data that looked superficially like a pointer but was to an invalid address. That includes addresses outside the computer's memory, outside the heap, inside the heap but into the middle of an object (rather than pointing at its head) and misaligned pointers. But in that case the sweep phase of the collector would not need any adjustment at all.

For a copying collector things are harder, because if data is copied from an old half-space to a new one then any pointer to it must be changed to track it. "Ambiguous bases" are memory locations that might contain a pointer in the heap, and the value stored in one is an "ambiguous reference". Any such reference could be a pointer but might be merely an integer with an unfortunate value, or part of a string or other binary data. So for safety its value must not be changed by the garbage collector, but the heap item it (potentially) refers to must be preserved. Thus items in the heap referred to directly from an ambiguous base must be handled as for a mark-and-sweep collector.

Any item in the heap that is only referenced from locations that are not ambiguous can, however, be copied or moved.

The only ambiguous bases I need to consider in `vsl` are on the C stack. All other referencess can be taken as fully under control, and the garbage collector must and will (and can) have information showing all the static variables that can ever hold references. Indeed the `vsl` code ensures this by keeping its (static) roots in an array called `bases` making it truly easy identify and process them.

I need to have code that can be assured of finding everything that is (or that should be) on the stack. This may involve some trickery in the fact of variables that are stored in registers not (immediately) on the stack... but I hope that arranging that the garbage collector itself declared and uses a large number of variables will ensure that if flushes all earlier callee-save values to the real stack.

One thing that emerges from this is that the original `vsl` scheme for floating point numbers (which allocates floats downwards from the top of the heap and everything else upwards from the bottom) becomes untenable. To see this imagine a case where a tight loop allocated very many floats and nothing else. Almost the whole heap is full of floats when garbage collection is triggered. Now suppose an ambiguous pointer pins one of the last to be allocated (and this in fact seems a very plausible situation). Then an address low within the heap has to end up storing a float. If the segregation between floats and other things is to be retained then this will really limit the space available for anything else. So an early change to `vsl` needed to support conservative garbage collection has to be to store floats in a way where they can mingle freely with other items in the heap. There are two potential ways of doing this – either every float is stored with a header word (doubling the space it consumes and slighly slowing allocation) or a bitmap is needed that flags which words in the entire heap contain floats (probably with similar speed impact).

Let me compare the space demands of each. If there $f\%$ of the heap is consumed by floats in the current scheme then adding headers consumes an additional $f\%$. For the bitmap approach I need one bit for every 64-bits of heap, leading to a 1.5% storage overhead regardless of whether floats are being used or not. Almost all of the time in most Lisp code there will hardly be any floats at all, but when they are in use the percentage heap occupation that may matter most relates to floats that have just been allocated, used and discarded. The bitmap approach could reduce the frequency of garbage collections triggered during heavy floating point use by up to a factor of two. Heavy use of declarations in the code and a "sufficiently clever" compiler that managed to do a lot of floating point arithmetic un-boxed could also make that saving. But I end up judging that the memory-use for the bitmap is in general a low overhead during general use as it provides a useful saving during stretches of execution that allocate many million floats.

17.1.2 Overall strategy

1. For each ambiguous base, look at content. If it does not refer to heap data that I had previously allocated ignore it. Otherwise tag that data as immovable.

   To implement this I first do a range check on the address to see if its lies within the area of the current active heap-half. I need to check tag-bits and alignment for sanity. I will maintain a bit-map that has a bit set to mark the first address of every object in the heap. Because all objects are doubleword aligned this needs one bit for every 8-bytes of heap, so this is only a 1 in 64 (1.5%) burden on memory - and the cost of setting that bit during a `cons` (and other allocation operations) is not terribly severe. That gives a really simple way that I can end up certain that I know when an ambiguous pointer truly points at the head of an object. If the object was a symbols it started with a flag word, and if it was a vector-like atom it would have a header word – and in each of those cases it would be easy to find a spare bit to use to mark it as immovable (I will often use the term "pinned" here to describe that state). However cons cells and headerless floats do not have spare bits, so the bitmap scheme here is the simplest option.

   I want to mention a range of possibilities here even if I end up discarding them. It seems probably that *usually* there will only be a modest number of items that end up pinned. In that case using a hash table to record where they are could seem attractive. The reason I do not go down that route is twofold: I do not have a reliable bound on how large that table would be, and bitmap tagging is already a guaranteed unit-cost solution. However a hash table could be attractive in that if multiple ambigiuous bases contain the same value it could help some processing deal with each just once. I think at present I do not believe that to be a big enough benefit to be worth fussing with.

2. Note that I must not start copying *anything* until I have identified everything that must be pinned. Items that are pinned must never move! So run the normal copying GC except that when a pinned is reached it is just left in place. The items within the pinned item are not copied at this stage because to do so would involve stack use within the garbage collector, and that is undesirable.

   Then re-scan the ambiguous bases and copy items that are referred to from pinned items. Note that this is not moving the pinned items themselved, just their contents. Also there is a requirement here that the set of and values in ambiguous bases must not have changed during garbage collection so far.

Since the contents of an item must be copied exactly once and since several ambiguous bases may refer to the same location it will be necessary to be able to tell if something has been processed. This is not very different from the challenge already facing the existing copying garbage collector. Specifically floating point values will (still) use a bitmap.

3. After copying there are some pinned items in the old heap. These must remain identifiable. Since it will be necessary to do a linear scan of the bitmap of pinned items at some stage the neatest idea will be to retain that bitmap and make allocation inspect it. When needing to skip past pinned items this will involve looking at their headers. Because in at least very many cases of garbage collection the stack will only be a few thousand words deep but the heap might be millions of words long it can be expected that pinned items are sparse in the heap, and in particular that there will be long gaps between adjacent pinned words. If the bitmap is often inspected word by word these runs can be skipped over in blocks of 32, speeding up the sweep usefully.

17.1.3 Overview of bitmap states at start and end of collection

Active space at start

1. A map that has a bit set showing the location of every headerless floating point value in the heap.
2. A map that will be used to mark floats. Clear it at start of GC.
3. A map that indicates that start address of each item in the heap.
4. The map that had marked pinned addresses from the previous collection that were hence locations that could not be allocated during the recent phase of computation. Since the garbage collection is now starting (possibly because the heap was full) the data in this bitmap is no longer required, so the map is cleared and can be used to tag locations pinned at this time.

Second space at start

1. A map that has a bit set showing the location of every headerless floating point value in the heap. The only data that can be in this space will be material pinned during the previous collection.
2. a map that would be used to mark floats. Not used! Thus the space for maps can be reduced by having only a half-sized one for marking floats so it doe snot cover the second half-space.

3. A map available to indicate that start address of each item in the heap. Again the only items in the heap are ones that were pinned last time, so this can be a copy of the pinning bitmap left over from the previous collection.
4. The map that had marked pinned addresses from the previous collection. The data from these is still in place and must be avoided while cupying into the second space, but that purpose can be achieved via the (copy of) the data as above. Copying can proceed and set a new start-of-object bit while avoding allocating where ons is already set. So this map can be cleared at the start of collection. Well more precisely the object-start and pinned maps can be exchanged to avoid the need for any bulk copying then the map to be used as the new pinning map can be cleared.

Old active space at end

1. A map that has a bit set showing the location of every headerless floating point value in the heap. To be ready for next time I think this needs masking with the map of pinned items in effect to discard all non-pinned floats on the old space.
2. floating point mark bits. Not needed at end of collection.
3. A map that indicates that start address of each item in the heap. The data in this will no longer be needed
4. Pinned data from this collection. Required for the future since the addresses so marked must be avoided by allocation within this half-space when that done during the next garbage collection.

Second space at end

1. Floating point map, now showing the location of both long-term pinned floats and ones that have just been copied to this space.
2. (floating point mark bits)
3. Object start adddresses.
4. Pinned data. The information here is no longer required.

17.1.4 Interaction with saving heap images

There can be two situations where an image of the state of Lisp must be saved to a file. The first is when it represents a checkpoint and execution must continue

after the image has been written. In this case all pinned items must be written, and the image on disc thus ends up (potentially) sparse. The pinned items have to be written out not because the ambiguous pointers that led to their pinning will still be present in a frest run of `vslplus` but because ordinary unambigiuous references to them are also liable to exist.

The second case is when the system will stop immediatly after creating an image file, and in this case it can be arranged that a fully compacted image file results. I can do a garbage collection that avoids marking from ambiguous roots during `preserve` since at that stage there ought not to be any important stack to mark. As a result no items will remain pinned at the end of garbage collection. Well in such cases I need just a bit more – I should ensure that the new space that I copy to does not contain any items that had been pinned during the previous GC. This can be achieved by performing two garbage collections so I end up with a fully compacted heap to save.

There are further issues that arise if there is a desire to expand the heap at run-time. If this is to be done other than when the stack is empty it is important that all existing parts of heap memory remain in position so that pinned items remain accessible. So if run-time heap expansion is to be considered it becomes necessary to structure both halves of the heap into segments with initial segments allocated at system start-up and new ones potentially added dynamically. Of course the format for saved heap images then has to allow for that! The size selected for segments will limit the size of the largest vector that can be allocated, and space will sometimes end up wasted near the end of a segment when a large item will only fit in the next one.

17.1.5 Wrapping up

With the rework described here many of the existing uses of `push` and `pop` in the original code become unnecessary, since Lisp references kept in simple C local variables become garbage-collection-safe. The speed of garbage collection is slightly impacted, and around 5% extra memory is needed for bitmaps, but experience has shown that conservative garbage collection schemes manage to recycle memory very nearly as effectively as simple precise ones, and they simplify not just the rest of the inner workings of the system they are built to work with, but the ability of it to link to other bodies of code that they may wish to exchange data with. It is overall a good policy to adopt.

17.2 Managing compiled code

When a Lisp system compiles bits of Lisp into native code for the machine that is in use it finds itself first treating the generated code as data (while it fills in all the bits) and then as executable code. Modern operating systems tend to try to protect against such behaviour since the dynmamic creation of code is a technique used in various sorts of malware. So typically special operating system calls are called for to mark some region of memory as available for use in this way.

Chapter 18

Some small Lisp programs

One of the better ways to learn a programming language is to start with a body of code that somebody else has already written. Even if you can find a good copy of the material on-line or on a CD there is much to be said for typing it in yourself! There are two specific benefits that arise from this. The first is that your fingers get used to the rhythm of the language, and practised in typing the various keywords and you are obliged to read the original code carefully so that you transcribe it correctly. At a purely mechanical level that starts to get you really familiar with how the language looks and feels. The second big benefit is that however experienced you are and however carefully you type you are essentially certain to mis-transcribe a few things. You may read a digit one (1) where some character is in fact a lower case ell (l), or (especially in Lisp) you may mis-count parentheses. It is remarkably easy to lose concentration for a moment and miss out whole lines of stuff. How can your mistakes possibly count as a benefit rather than just a pain? Well because when you first try your example code it will presumably then not work, and you have just provided yourself with a nice exercise in debugging. Of course you could start by just doing a careful proof-read of your version against the "official" and tested code, but reading the messages you get and trying to uncover what was going on for yourself first will be excellent practise for when you write your own code from scratch and so do not have some known-good version to compare against. At the very least you will get to see some of the more common behaviours and messages your programming environment gives when presented with bad input.

The Lisp provided here was implemented more so that the code making up `vsl` is short and reasonably comprehensible rather than with serious emphasis on it providing good diagnostics. Perhaps you will even end up deciding you wish to extend to code in `vsl.c` so as to make it more friendly in this respect.

The example Lisp programs here illustrate some of the features of Lisp, and also some of the sorts of programs you might find it good for. Almost all of these

programs has been cut down to to almost the bare minimum, so that it illustrates a programming style or an application area. So while it is hoped that the code will be entertaining even as it is, a significant part of its purpose is to provide starting points for larger programming projects. At the end of each example there will be some comments on the technology that it has introduced, how you might extend the example and what related programs you could consider writing.

18.1 The $3n + 1$ problem

Start with a number. If it is even then halve it, while if it is odd you multiply it by three and then add one. That gives you a second number. Do that same to it, and keep going to generate a sequence. For instance it you start with 7 you get $7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1$. It is reasonable to stop when you reach 1 because from there on you would just get a dull loop $1, 4, 2, 1, 4, 2, 1, \ldots$.

The challenge here is to investigate whether there is a starting value such that the sequence that it starts does not end up at 1. You are *really* unlikely to find a case that does not, since people have tested all starting values up to at least 10^{18} and shown that in all those cases the sequence does end. However no proof that some larger starting value might cause trouble has been produced. It is amazing to see how as you alter the starting value the length of the sequence changes and how large numbers can arise along the way. So here is some Lisp code suitable for use with `vsl` that will compute and display what goes on.

It defines a function called `threenplusone` that takes two arguments. The first is the value in your sequence being considered, while the second counts down so you can avoid going on for longer than you want. If the number ever reaches 1 or if the count expires the function returns, doing a `(terpri)` which will lead to a blank line being printed. Otherwise if tests if the number is even or odd and does what it has to. Yes doing simple arithmetic in Lisp can look ugly until you have got use to it! `(add1 (times 3 n))` looks much bulkier than just $3n+1$, but despite that it is utterly unambiguous and really rather simple to write. Writing arithmetic this way makes the notation for it just the same as the notation for all other operations: Lisp can be held to be expressing a view "what's so very important about arithmetic then that it deserves very special notation?".

```
% 3n+1 experiment. Look up "Collatz sequence" on the web.

(de threenplusone (n count)
   (prin n)
   (princ " ")
   (cond
     ((or (onep n) (minusp count)) (terpri))
```

```
((zerop (remainder n 2))
   (threenplusone (quotient n 2) (sub1 count)))
(t (threenplusone (add1 (times 3 n)) (sub1 count)))))
```

Having defined a function we then need to call it. Here I just use `dotimes` to try the cases up to (but not including) 30. So `threenplusone` is called on $0, 1, \ldots, 29$. For this range using a cut-off limit of 140 to trap any calculation that might escape (as the one starting with 0 does) turns out to be about right. If you alter the range of starting values you try you are liable to need to experiment with the limit too.

```
(dotimes (n 30)
   (terpri)
   (threenplusone n 140))
```

When you have tried this small program you can try looking up more information about the $3n + 1$ problem, for instance on Wikipedia. You will find that the challenge involving it is otherwise known as the Collatz Conjecture and that many people have contributed ideas towards its analysis or have supplied the results of lengthy computer testing. You will find that there are simple generalisations of this problem (for instance consider $3n + 5$ in place of $3n + 1$) and for these discovering just how many final cycles the sequences can drop into (on the original problem the only cycle you can reach if you start with a positive number is the $1, 4, 2$ one) is interesting. There are even ways in which this sequence can be used as a basis for generating wonderful images that are related to the famous Mandelbrot set. It feels very nice to be able to make the first Lisp example included here both very short and also one that introduces you to a topic whose full analysis has defeated mathematicians for around 75 years!

If you try running this code extensively you will discover that it nests calls to `threenplusone` very many deep as it computes. If you try it on a staring number that leads to a sequence longer than a few thousand (or perhaps a few tens of thousands) you may get a calamitous crash from `vsl` when it runs out of stack space to handle this recursion. More advanced Lisp implementations would not suffer this way, and you could re-write the function using `prog` in an uglier style so as to avoid problems even with `vsl`. This can be a further exercise.

18.2 Evaluation of formulae

In `vsl` you write arithmetic using fully spelt out names such as `difference` and `plus` for all the operations.

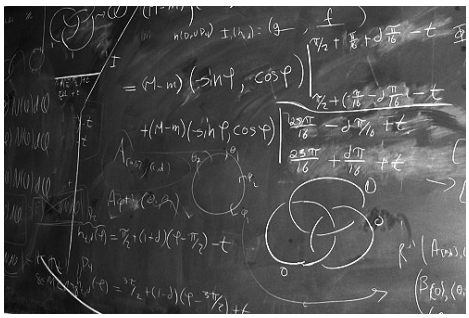

Figure 18.1: Some formulae (photo courtesy shonk on flickr)

Perhaps you feel that is really unspeakably clumsy and would rather be able to write something like `(- (* 2 8) 7)`. Still in Lisp prefix notation with the operation specified at the start, but with your own choice of short symbols to specify which operation is to be performed. The code here shows how you can take a Lisp data structure that represents a formula in your new notation and evaluate it. Now of course if the only think you needed to do was to provide an alternate name for some existing Lisp operation you could just do that using function definition, as in

```
(de - (a b) (difference a b))
```

but the scheme here could be extended to change behaviour in almost arbitrarily flexible ways. The order in which the code here is written is "top down". The main entry-point is coded first, and it makes reference to a collection of sub-functions that will be written later. The idea is that `evaluate` will be given an expression. If that is just a number then nothing much has to be done. Otherwise the expression will have an operator and two arguments, and a function `compound` will be used to process it.

```
% Evaluation of arithmetic expressions

(de evaluate (x)
   (cond
       ((numberp x) x)
       (t (compound (operator x)
                    (arg1 x)
                    (arg2 x)))))
```

Even before `compound`, `arg1` and `arg2` have been written it is possible to test the simple case where `evaluate` is given a numeric argument. Being able to test parts of your code early is one of the nice things about using Lisp.

This simple evaluator only allows for terms where there are exactly two arguments, and extracting them (and the operator) from a list merely involve use of combinations of `car` and `cdr`. It is still good style to make these accessor functions with meaningful names rather than using `cadr` and `caddr` in the higher level code directly.

```
(de operator (x)
   (car x))
```

```
(de arg1 (x)
   (cadr x))

(de arg2 (x)
   (caddr x))
```

Having got this far the code that actually does the evaluation becomes rather simple. The operands in any formula must each be evaluated and then combined.

```
(de compound (op a1 a2)
   (arithmetic op (evaluate a1) (evaluate a2))

(de arithmetic (op a1 a2)
   (cond
      ((eq op '+) (plus a1 a2))
      ((eq op '*) (times a1 a2))
      ...
```

The code shown is left incomplete since you could wish to include division or other operations. When finished, an example use would be to go `(evaluate '(- (* 2 8) 7))`.

If you were only ever using binary operators it would be trivial to alter the code to allow an infix rather than prefix notation, as in `((2 * 8) - 7)` by just adjusting the functions that extract the operator and arguments from a formula. But if you do that it becomes harder to allow for things that need just one operand, or that can accept many.

The big extension of this example that turns it into a real project is to extend it to cope with variables as well as numbers, and to provide ways for its user to define new functions. Before long it would become a Lisp interpreter coded in Lisp. That might sound unspeakably and improbably ambitious, but a basic Lisp interpret can be as short as barely a couple of pages of code, and you could find plenty of explanation of ways to arrange it in (for instance) the original Lisp 1.5 manual.

As a final way in which this example can lead on into further work, there is no special reason why the evaluation rules you implement here have to be especially Lisp-like. Yes the input needs to be given as a Lisp data-structure, but the collection of operations that are being choreographed and their exact behaviour are entirely up to you! For instance the widely used editor `emacs` uses a variety of Lisp to allow its users to customise their editing experience. The Gnu Compiler Collection (`gcc` and its friends) has an internal component that they call `MELT`[35] that is distinctly Lisp-like. You could investigate the language Prolog and create a micro-prolog by coding an evaluator for it in Lisp. There are almost no limits!

18.3 Sorting

One of the fundamental tasks that almost any large program will need at some stage is that of sorting material into order. A full-scale Lisp would have a general-purpose sorting procedure built in, but `vsl` is cut down and does not. So in this example we have code that can be used to sort a list of symbols into alphabetic order.

There are many different strategies for sorting. The one used here is known as *tree sort*. It starts by building a tree structure. Each node of the tree contains a key and has a left and a right child-tree. Every key in the left sub-tree will come before the one in the mode, and every one in the right will come after it. It should be clear that once such a tree has been built that it will be possible to flatten it to recover a simple list of sorted items. So the top-level of the sort code merely builds and then flattens a tree.

```
(de sort (items)
   (prog (tree)
      (dolist (x items)
         (setq tree (insert x tree)))
      (return (flatten tree))))
```

Now Lisp has a `cons` operator that makes structures, but it does not have a direct way of building the sort of node required here with a key and space for two sub-trees. However it is easy to write tiny functions using `car`, `cdr` and `cons` to model just what is wanted. That approach of using lists and `cons` nodes to build absolutely whatever shape data structures is required is a normal way of using Lisp. Providing the wrapper functions as done here then makes subsequent use able to think in terms of nodes and branches rather than raw `cons` cells.

```
(de mknode (v l r)
   (cons v (cons l r)))

(de getval (x) (car x))

(de leftbranch (x) (car (cdr x)))

(de rightbranch (x) (cdr (cdr x)))
```

Inserting a new item into a tree is in fact rather easy. If the tree is empty (represented by `nil`) you just construct a new node with empty children. Otherwise you need to compare the new key with the one in your top node and insert the new item in either the left or right child as appropriate.

```
(de insert (item tree)
   (cond
```

```
      ((null tree) (mknode item nil nil))
      ((orderp item (getval tree))
         (insertleft item tree))
      (t (insertright item tree))))
```

Inserting into the left or right child can be done by rebuilding a copy of the current node using the current key and one unchanged child. The code here could have been written out within the `insert` function above, but making every function really short and arranging that each function implements just one idea can help keep the code clear and easy to understand – and hence easier to debug.

```
(de insertleft (item tree)
   (mknode
      (getval tree)
      (insert item (leftbranch tree))
      (rightbranch tree)))

(de insertright (item tree)
   (mknode
      (getval tree)
      (leftbranch tree)
      (insert item (rightbranch tree))))
```

Now we have completed the job of building an ordered binary tree. So it is just necessary to flatten it. The flattened version of a tree will be the flattened left child, followed by the top item in the tree and finally the flattened right child. To make the example here fully self-contained a definition of the standard `append` function for concatenating a pair of lists is included.

```
(de append (a b)
   (cond
      ((null a) b)
      (t (cons (car a) (append (cdr a) b)))))

(de flatten (x)
   (cond
      ((null x) nil)
      (t (append (flatten (leftbranch x))
         (cons (getval x) (flatten  (rightbranch x)))))))
```

The only thing left to do is to provide a way of checking if two items are in alphabetic order. Again a full-scale Lisp might provide a built-in function to do this, but in `vsl` it is possible to use `explode` to expand any name (or in fact any other Lisp item) into a sequence of characters, and then `char!-code` to map each individual character onto its numeric code. These codes are then easy to compare using `lessp`.

```
(de orderp (a b)
   (orderp1 (explode a) (explode b)))

(de orderp1 (al bl)
   (cond
       ((null al) t)
       ((null bl) nil)
       ((equal (car al) (car bl))
          (orderp1 (cdr al) (cdr bl)))
       (t (lessp (char!-code (car al))
                 (char!-code (car bl))))))
```

Finally it is possible to test the code. Tree-sort will usually do its work rapidly, but if its input happens to be in exactly the right order already (or indeed in exactly the reverse of that) the trees that it builds end up unpleasantly lop-sided and performance drops dramatically. If you want a sorting method that can never suffer from that sort of bad behaviour you have several options. One in in fact included later in the Route-finding example here, but you could also so a web search or check a book on Algorithms (Cormen et al[11] in any of its editions is an excellent one) and adapt merge-sort for Lisp use. Using `rplaca` and `rplacd` mergesort could even avoid need for significant `cons`ing. Perhaps a good activity would be to implemented as many sorting methods as you reasonably can and compare their performance on large amounts of data (sometimes random, sometimes already sorted, sometimes with many repeated items) to discover empirically which is the winner and how you can refine your code to make it run as fast as possible.

```
(sort '(s o r t i n g))

(sort '(these things we hold to be self evident))

(sort (oblist))
```

18.4 Arbitrary precision arithmetic

A proper Lisp supports arbitrary precision integer arithmetic so that any limits on the size that values may grow to are to do with memory constraints or some other internal but huge limit. Furthermore a proper Lisp arranges that "sufficiently small" numbers are handled more efficiently than the general case. In `vsl` the internal workings support two representations for integers. One is "efficient" and supports value that have just three fewer bits in their binary representation that the word-width of the computer in use. So on a 32-bit system this is the range from -268435456 to 268435455, while on a 64-bit computer it is much larger. `vsl` stores larger numbers in arrays (in a way very similar to the manner in which it

stores strings) and so it would be possible to have code that used just the right amount of space to cope with whatever big number arose. But to keep the source code for `vsl` concise the current version actually just uses fixed precision 64-bit integers there. The `vsl` library builds on this to implement unlimited arithmetic representing huge numbers as lists of digits. This example shows a simplified version of the code used: it just supports addition and multiplication, and it will be less efficient than the full version.

For simplicity here numbers are represented as lists of simple digits in the range 0-9. To make the arithmetic easier to implement the lists will store the least significant digit of any number first, so that when you look at a list the numbers seem to be written backwards.

The first task is to be able to convert an ordinary Lisp integer into such a list. But given that the second task will be to add two big integers the code here looks ahead to what will be needed then and provides a function that adds a small-number (`carry`) into a big number that is stored as a list (`a`). Then conversion to big format can be done by just carrying a value into an empty big-number.

```
(de carryinto (a carry)
   (cond
      ((null a) (cond
         ((zerop carry) nil)
         (t (cons (remainder carry 10)
                  (carryinto nil (quotient carry 10))))))
      (t (cons (remainder (plus (car a) carry) 10)
               (carryinto (cdr a)
                   (quotient (plus (car a) carry) 10))))))

(de makebig (n)
   (carryinto nil n))
```

Addition is now just a matter of adding corresponding digits of two numbers and keeping track of the carries. When all of the digits of one of the inputs have been processed this falls back to become the case already dealt with.

```
(de bigplus (a b)
   (bigplusc a b 0))

(de bigplusc (a b carry)
   (cond
      ((null a) (carryinto b carry))
      ((null b) (carryinto a carry))
      (t (cons (remainder (plus (car a) (car b) carry)
                          10)
               (bigplusc (cdr a) (cdr b)
```

```
                (quotient
                   (plus (car a) (car b) carry)
                   10))))))
```

Short multiplication, in other words multiplying a big number by a small one is pretty obvious too.

```
(de smalltimes (a b carry)
   (cond
      ((null b) (carryinto nil carry))
      (t (cons (remainder (plus (times a (car b)) carry)
                          10)
          (smalltimes a (cdr b)
              (quotient (plus (times a (car b)) carry)
                        10))))))
```

The key to finishing off the multiplication code is to recognise that `cons`ing a zero on the front of a list amounts to shifting all of its digits up a place, and hence multiplies it by 10. Given that long multiplication becomes just an exercise in using short multiplication and addition.

```
(de bigtimes (a b)
   (cond
      ((or (null a) (null b)) nil)
      (t (bigplus (smalltimes (car a) b 0)
                  (cons 0 (bigtimes (cdr a) b))))))
```

Having written some code it is always prudent to put in some test cases. A really nice thing about Lisp is that you can generally interleave test cases with function definitions.

```
(makebig 100)
(makebig 123456789)
(bigplus (makebig 123456789) (makebig 987654321))
```

A more interesting test will define a function that can raise numbers to a power, and use it to compute first some small powers of 2 (where we will recognise the answers and can check them easily) and then some that are big enough to show off the capabilities of this code.

```
(de bigsquare (x)
   (bigtimes x x))

(de bigpower (a n)
   (cond
      ((onep n) a)
```

```
      ((zerop (remainder n 2))
         (bigsquare (bigpower a (quotient n 2))))
      (t (bigtimes a
         (bigsquare (bigpower a (quotient n 2)))))))

(bigpower (makebig 2) 10)
(bigpower (makebig 2) 20)
(bigpower (makebig 2) 100)
(bigpower (makebig 2) 1000)
```

The simplest way to build on this example is to alter it to use a radix much larger than 10. It would be easy to use any power of ten up to 10000 and then all internal working would remain fast. If you increased the radix to 1000000000 you could probably still keep all working within `vsl`'s existing range. Using a larger radix should significantly speed things up. The code in the `vsl` library uses a power of 2 (specifically 2^{30}) rather than a power of 10. That makes printing somewhat more expensive but helps in other respects – especially with operations such as `logand` and `logor`. The library version also needs to cope with negative numbers, division and a collection of extra operations such as bit-wise logical ones. But the library code is still not highly optimised, and you could work towards producing a faster or better behaved replacement for it.

At a more radical level you could rework the code in `vsl.c` to support true arbitrary precision working in the places where the existing code just uses 64-bit values. This could be very beneficial since, as shown by one of the benchmark tests documented later here, the existing Lisp-coded arithmetic may be an order of magnitude slower then a respectable C coded version could be. You could either implement your own code to do the high precision working, or perhaps investigate existing libraries. Probably the leading multi-precision library is the GPM, which can be found via `http://gmplib.org`. However the storage management arrangements that GMP make do not fit very obviously comfortably with the ones that `vsl` uses, and care would be needed in supporting a scheme to save and later on restore Lisp images: using it would involve rather more than just linking together the two bodies of code in a naive manner.

If you write your own code then all issues of licensing it are basically up to you. The BSD license that `vsl` is covered by is permissive and perhaps its most important feature is that it reminds you that `vsl` does not come with any guarantee. But if you use GMP or any other brought-in library you will naturally wish to ensure that you understand the consequences of its license terms and adhere to them properly. For the commonly used GNU Public License and its LGPL variation you can check the Practical Guide to GPL Compliance[8] which elaborates in concrete and practical language the implications of the legal wording that apply.

Given high precision integer arithmetic there are experiments you can do in-

volving the identification of large prime numbers, performing high security data encryption (look up the "RSA" encryption method), building in high precision integers to support fractions or floating point work... the opportunities are almost endless.

18.5 Prettyprinting

This next example application is in fact included in the library of Lisp code that `vsl` expects to be provided with from the start. It is included here because it illustrates some of the benefits from the way in which Lisp code can be treated as Lisp data, and because extending and improving it can make a worthwhile project.

When working with Lisp code it is vital to keep parentheses correctly matched. A simple typing error or counting failure that leads to one getting omitted can utterly wreck the meaning of a piece of code. One way to help users with this is to provide something that can reformat their code with neat and systematic indentation. When they check what they have written they will mainly then be reviewing the way that bits of their code are grouped based on that indentation, which is perhaps easier for them than parenthesis counting. Of course one could imagine such re-layout code being built into the text editor that they use, but here is some quite short Lisp code that can print out Lisp for checking. The top-level function that people call will be called `prettyprint`, but all that will really do is to arrange to start on a fresh line and call a function `pprint` giving it and extra argument that indicates the current indentation level.

```
(de prettyprint (x)
   (terpri)
   (pprint x 0)
   (terpri)
   nil)
```

If the expression to be displayed is atomic or if it is a short list (checked for her by seeing if its printed representation has no more than 40 characters) it is just printed in the ordinary Lisp manner. Otherwise we have a list that must be split over several lines. A left parenthesis is printed, then the function (indented just one extra space). Then the job of printing everything else is passed down to `pprintail` which is told to apply an indentation three spaces deeper.

```
(de pprint (x n)
   (cond
      ((or (atom x)
           (lessp (length (explode x)) 40)) (prin x))
      (t (princ "(")
         (pprint (car x) (add1 n))
         (pprintail (cdr x) (plus n 3)))))
```

pprintail does obvious things when it reaches the end of a list, but is mostly there to start a new line, indent by the relevant number of spaces and call pprint to print each member of the list. The example finishes by using prettyprint to display the definition of part of its own code.

```
(de pprintail (x n)
   (cond
      ((null x) (princ ")"))
      ((atom x) (princ " . ")
                (prin x)
                (princ ")"))
      (t (terpri)
         (spaces n)
         (pprint (car x) n)
         (pprintail (cdr x) n))))

(prettyprint (getd 'pprint))
```

This prettyprinter does a reasonably good job for such a short body of code, but for there are still collections of cases where it could do with enhancement. The first thing is that its decisions about when it can merely print lists without indentation (based on a 40-character threshold) is crude and does not depend on how far across the page you are. Very long thin lists are displayed with each item on a separate line in a way that feels a horribly wasteful of space. As the size of the expression you are printing increases the indentation can get to be comparable with your line length, and the results and up messy. Vectors are not addressed at all by the prettyprinter, and it has no idea what to do with excessively long stings or numbers. All in all it provides a start, but there is much scope for improvement!

18.6 Tracing, errorset and backtraces

The example code here is not really to show Lisp being used to solve a problem. It is more about resolving problems that can be encountered with Lisp code. Lisp is in general a nice language to debug because it is generally possible to input data to test functions as you define them. After all the data is liable to be merely lists! But if something has been coded and is not behaving it can be a real help to see all calls to it and the corresponding results it delivers as it is exercised by a few test cases. The example given here defines a function for computing factorials and then traces both it (a user defined function) and sub1 (a function built into the vsl kernel) and runs a small test.

```
(de fact (n)
   (if (zerop n)
```

```
       1
       (times n (fact (sub1 n)))))

(trace '(fact sub1))

(fact 3)
```

The output generated by running this test follows, and you can see both all the calls to `sub1` and how `fact` recurses and then eventually unwinds to deliver a result. In this instance all the data being worked with is simply numeric, but a general good feature of Lisp is that lists as well as atomic data have well defined printable representations and so there is usually no need to do anything special to arrange that `trace` can display arguments and results.

```
(fact 3)
Calling: fact
Arg1: 3
Calling: sub1
Arg1: 3
sub1 = 2
Calling: fact
Arg1: 2
Calling: sub1
Arg1: 2
sub1 = 1
Calling: fact
Arg1: 1
Calling: sub1
Arg1: 1
sub1 = 0
Calling: fact
Arg1: 0
fact = 1
fact = 1
fact = 2
fact = 6
Value: 6
```

A second issue relevant when debugging is the concept of a backtrace. When a Lisp function fails the default behaviour of `vsl` is to print a list showing what functions were active in the run up to the exception. To illustrate that the code here defines a function whose sole real purpose is to crash – it does so by calling the `error` function. While backtraces can be invaluable when you are debugging there can be occasions when you expect errors to arise, or even wish to provoke

them deliberately to abort a path of computation you have concluded is not being productive. In such cases any backtrace could be an unwelcome distraction. The function errorset can be used to request evaluation of a Lisp expression such that any error in the evaluation is reported back to errorset rather than causing the Lisp run to terminate. errorset can also control how much information about any error gets displayed.

```
(untrace '(sub1))

(de fail (n)
   (if (zerop n)
       (error 1 "crashing")
       (times n (fail (sub1 n)))))

(errorset '(fail 3) nil nil)

(errorset '(fail 3) t nil)

(errorset '(fail 3) t t)
```

Here is the output when errorset asks for full diagnostics.

```
(errorset '(fail 3) t t)

+++ Error: error function called: (1 "crashing")
Calling: error
Calling: fail
Calling: fail
Calling: fail
Calling: fail
Value: nil
```

The trace capability was rather easy to add into vsl mainly because it is implemented as just a simple interpreted system. But the version in place does not display the names of function arguments (it just calls them arg1 and so on). It might be safer to arrange that rather than using the standard print function it used some (new) variant) that set some limit to how much it can display. This would be crucial if there was a risk that cyclic structures might get created via rplacd.

The current backtrace in vsl does not display arguments to the functions it lists. And it is probable that a nice interactive debugging capability could be developed by merging the two and adapting trace to provide a breakpoint or single stepping capability for vsl.

These changes would of course be within the implementation of vsl. An alternative project starting from the idea of tracing would be to write code that

picked up the definition of an existing Lisp function and re-wrote it to insert extra fragments of Lisp to achieve what `trace` does at present. For instance the body of a function could perhaps be replaced by something along the lines of

```
(progn
   (princ "arg1: ")
   (print a1)
   (let ((r ORIGINAL_BODY))
      (princ "= ")
      (print r)
      r))
```

It might be hard to make this Lisp-based trace code work for built-in functions (which do not have Lisp-coded definitions to edit) but it could provide extreme flexibility for when user-written code needed investigation.

18.7 An animal guessing game

Wherever there is a collection of sample applications to illustrate how to use a programming language there has to be some sort of game. The first such included here is a rather naive one that asks the user questions and on the basis of their answers guesses what animal they are thinking of. It keeps a database of questions it might wish to ask in the form of a binary tree – very similar to the one used in the previous sorting example. Here each node of the tree contains the text of a question to ask, and then the left and right sub-trees correspond to where to continue the conversation based on whether the answer to the question was "yes" or "no". When the program reaches a leaf that means it has run out of questions it can ask, so it has to make its guess. Leaves of the tree are symbols giving the name of the animals to guess at that point.

If the program guesses right it just asks the user to try again. If it fails it will get the user to provide a new question, and it build that into its search tree. The tree is updated using the potentially destructive `rplaca` and `rplacd` operations so this example provides some sort of illustration of their possible use.

The first few functions are just there to get things started.

```
(de animal nil
   (while t
      (progn (say_hello)
             (setq known_animals (guess known_animals)))))

(de say_hello nil
   (terpri)
```

Figure 18.2: Does it have sharp teeth? Is it a honey badger?

```
   (lpri '(Think of an animal and I will guess it))
   (terpri))

(de lpri (l)
   (dolist (x l)
      (princ x)
      (princ " "))
   (terpri))
```

Observe that in Lisp it may be most convenient to keep a sentence as a list of words rather than just as a single symbol or string – but it is easy to write a function like `lpri` to print it in the format you most like.

The function `guess` is a close cousin of the code that inserted a new node in a binary search tree. It is coded using `rplaca` to update the existing tree rather than making a copy of the original. You can judge for yourself if this makes it shorter or clearer.

```
(de guess (known_animals)
   (cond
      ((atom known_animals) (I_guess known_animals))
   (t (printc (car known_animals))
      (cond
         ((yesp (read))
            (rplaca
               (cdr known_animals)
               (guess (cadr known_animals))))
         (t (rplacd
               (cdr known_animals)
```

```
                (guess (cddr known_animals)))))
      known_animals)))

(de I_guess (creature)
   (lpri (list 'Is 'it 'a creature))
   (cond
      ((yesp (read)) (printc 'Hurrah) creature)
      (t (give_up creature))))

(de yesp (a)
   (if (eq a !$eof!$)
       (stop 0)
       (or (eq a 'yes) (eq a 'y))))
```

If the program fails to guess an animal it needs to extend the tree by adding a new question. A line of text to make up that question is grabbed using `readline` and ends up part of the tree.

```
(de give_up (I_thought)
   (prog (new_animal new_question)
      (lpri '(I give up))
      (lpri '(what was it?))
      (setq new_animal (read))
      (lpri '(Please type in a question that would))
      (lpri  (list 'distinguish 'a new_animal
                   'from 'a I_thought))
      (setq new_question (readline))
      (lpri '(thank you))
      (return (cons new_question (cons new_animal I_thought)))))
```

Finally it is necessary to provide an initial database. The one here is rather small and the questions that it asks may not appear subtle. So feel free to adapt it to make it bigger and cleverer. Once the database is in place you just have to invoke `(animal)` to play. The code is not at all clever so if you accidentally type an extra line it may get confused.

```
(setq known_animals
   '(Does! it! have! a! long! neck!?
       (Does! it! live! in! africa giraffe . swan)
       Does! it! have! big! ears!?
       (Does! it! have! a! big! nose elephant . rabbit)
       . crocodile))

(animal)
yes
```

```
no
no
snake
has it got a forked tongue?
yes
no
yes
yes
```

Perhaps identifying an animal like this feels as if the game here belongs in the nursery. However if the interface to this "game" was tidied up and a much larger and more serious database provided it could become useful. You may like to look at a book of plants (eg the Flora of the British Isles[10][1]) and see if you can build a decision tree for identifying flowers. The Clue Books guide to Seashore Animals[3] could also provide a starting place where your sequence of questions has already been sketched. Or almost any other field-guide could be adapted.

A rash and over-enthusiastic person would consider making the questions relate to medical symptoms and the eventual "guesses" to diagnoses of illness. I believe that a fault-finding guide to car or motorcycle engines would be safer than one the try to discern what was wrong with a human patient! For some of these examples you may need to extend the code to allow a "I do not know" response as well as just "yes" and "no", and for a large body of knowledge you will need to develop a knowledge capture program that allows you to build you database in a systematic manner.

18.8 Route-finding

Finding the shortest path from point A to point B via some transportation network leads to a classical problem in Computer Science. The network is thought of as a graph (with towns as vertexes and roads between them as edges) with each edge assigned a weight or cost. Any textbook on algorithms will present efficient ways to solve this problem. Here we have a version of one of them coded in Lisp.

A key concept needed here is that of a *heap*. A heap is a variety of tree where the lightest elements always appear towards the top. Where in the binary tree used for sorting earlier you put an arbitrary key in the root of the tree and places other items to the left or right according as they were smaller or larger than that, here we just put the smallest item at the top. Furthermore each sub-tree has its smallest item at its root, but beyond that there is no great concern about what has to go left and what right.

[1] which and "Artificial Key to Families" in a form that could almost be adapted

The start of the code here defines wrapper functions that allow nodes in the tree to gave the structure `((key . value) . (left . right))`. Adding an item to a heap then ensures that the smallest item is kept at the top but otherwise it just alternates which child-tree it inserts new data into. This keeps the two sub-trees equal in size, and hence avoids the degenerate cases that could occasionally make tree-sort slow.

```
(de node (kv l r) (cons kv (cons l r)))
(de kv (x)    (car x))
(de k  (x)    (caar x))
(de v  (x)    (cdar x))
(de left (x)  (cadr x))
(de right (x) (cddr x))

(de add_to_heap (item heap)
   (cond
      ((null heap) (node item nil nil))
      ((lessp (car item) (k heap))
         (node item
               (right heap)
               (add_to_heap (kv heap) (left heap))))
      (t (node (kv heap)
               (right heap)
               (add_to_heap item (left heap))))))
```

The heap will be used as a *priority queue*. Items will be inserted from time to time, and then at each stage the smallest will need to be identified and extracted. Well the smallest item is guaranteed to be at the top, so finding it is trivial. The trickier task is to remove it and leave the remaining items in the heap in good order.

The idea used here is that if you inspect the code that inserts items into a heap any addition always ends up with there being something new at the extreme right hand end of the structure. So by mirroring the insert code it should be possible to remove that particular item and end up with a tree that has the same shape that the original has before the most recent insert operation. This is an excellent plan in that it guarantees to keep the left and right sub-heaps neatly balanced in size. Unfortunately it removes the rightmost item, while it was the top one that needed to be deleted! So the code here saves the forcibly removed item so that it can be put back in place of the item that used to be at the top. This end up having removed the top item (as desired) and having kept the heap nicely balanced, but the item that has just been brought to the top may not in fact be the smallest value present. So observe that a function that has not yet been presented called `restore_heap` is called to fix that issue.

```
(de shrink_heap (heap)
```

```
   (cond
      ((null (right heap))
       (setq removed (kv heap))
       nil)
      (t (node (kv heap)
               (shrink_heap (right heap))
               (left heap)))))

(de remove_top_item (heap)
   (let!* ((removed nil)
           (h1 (shrink_heap heap)))
      (if (null h1)
          nil
          (restore_heap (cons removed (cdr h1))))))
```

The function `restore_heap` just has to allow the top item in the heap to sink to its correct level so that anything lighter than it floats up to be above it. Thus the top key has to be compared against the keys at the top of each sub-heap. The code is made a little messier by needing to cope with the possibility that one or even both sub-heaps could be empty. But if you trace through the tests in the following code you will see that there are four cases that end up needing treating:

1. The whole heap is empty;
2. The top of the left sub-heap is the smallest item anywhere;
3. The top of the right sub-heap is the smallest item anywhere;
4. The item you had just placed at the top of the heap is exactly where it belongs.

The code to cope with each of those cases is shorter that the sequence of tests that detect which case applies.

```
(de restore_heap (heap)
   (cond
      ((null heap) nil)
      ((and (left heap)
            (lessp (k (left heap)) (k heap))
            (or (null (right heap))
                (lessp (k (left heap)) (k (right heap)))))
       (node (kv (left heap))
             (restore_heap (cons (kv heap) (cdr (left heap))))
             (right heap)))
      ((and (right heap)
```

```
            (lessp (k (right heap)) (k heap)))
       (node (kv (right heap))
             (left heap)
             (restore_heap (cons (kv heap) (cdr (right heap))))))
      (t heap)))
```

We now have the code for heaps complete – we can add a new key into a heap and we can remove the top item. Those keen on algorithms can observe that if a heap has N items stored in it each of these operations will have a cost proportional to $\log(N)$, which means that the costs grow rather slowly even for large heaps. As a temporary diversion to the real task of route-finding it is perhaps useful to exercise and test the heap code before doing much more. Given heaps it is amazingly easy to write a function to sort lists. It takes all its input and adds each item in turn into a heap. Then for so long as the heap is not empty it repeatedly removes the top item of the heap. This naturally reads off the items in ascending order. Because it is convenient to use `cons` to put each next smallest item on the front of a list the raw result list ends up in descending order, so I just use `reverse` to put it into the state I want. The test case for sorting used here is based in digits of pi, which provide a systematic but random-seeming set of numbers.

```
(de heapsort (l)
   (let ((h nil))
      (dolist (x l) (setq h (add_to_heap (list x) h)))
      (setq l nil)
      (while h
         (setq l (cons (k h) l))
         (setq h (remove_top_item h)))
      (reverse l)))

(heapsort '(3
    1415926535 8979323846 2643383279 5028841971 6939937510
    5820974944 5923078164 0628620899 8628034825 3421170679
    8214808651 3282306647 0938446095 5058223172 5359408128
    4811174502 8410270193 8521105559 6446229489 5493038196
    4428810975 6659334461 2847564823 3786783165 2712019091
    4564856692 3460348610 4543266482 1339360726 0249141273
    7245870066 0631558817 4881520920 9628292540 9171536436
    7892590360 0113305305 4882046652 1384146951 9415116094
    3305727036 5759591953 0921861173 8193261179 3105118548
    0744623799 6274956735 1885752724 8912279381 8301194912
    9833673362 4406566430 8602139494 6395224737 1907021798
    6094370277 0539217176 2931767523 8467481846 7669405132
    0005681271 4526356082 7785771342 7577896091 7363717872
```

```
    1468440901 2249534301 4654958537 1050792279 6892589235
    4201995611 2129021960 8640344181 5981362977 4771309960
    5187072113 4999999837 2978049951 0597317328 1609631859
    5024459455 3469083026 4252230825 3344685035 2619311881
    7101000313 7838752886 5875332083 8142061717 7669147303
    5982534904 2875546873 1159562863 8823537875 9375195778
    1857780532 1712268066 1300192787 6611195909 2164201989))
```

With a priority queue (implemented as a heap) available the route-finding code itself is fairly short. It keeps its queue with cities and the length of a route to them that it knows about. This obviously starts with the source city at distance zero from itself. At each step it picks the city that the queue shows as next closest. Sometimes that will in fact be a second routing to a city that has already been met, and so it is ignored. Otherwise it sets the `distance` property on the new city so that everybody can tell how far it was from the source. So that it can reconstruct a route as well as merely calculating a distance it also sets a `previous` property that shows shows the previous city on the short path. Then for all the neighbours of the new city it inserts an entry in the queue showing how they could be reached if a path via this location were selected.

Once the desired destination has been reached all this stops and the code can backtrack through the `previous` references to reconstruct a route. The code has a couple of commented out lines that would print messages as each city was processed. These could be reinstated if that extra output was helpful for understanding or debugging.

```
(de find_route (source destination)
   (prog (queue city distance prev)
      (setq queue (node (list!* 0 source nil) nil nil))
      (while (and queue (not (get destination 'distance)))
         (setq distance (k queue)) % Next nearest place
         (setq city (v queue))     % City and predecessor
         (setq prev (cdr city)) (setq city (car city))
         (setq queue (remove_top_item queue))
         (when (null (get city 'distance)) % Seen before?
            (put city 'distance distance)
            (put city 'previous prev)
%           (princ "Distance to ") (princ city)
%           (princ " is ") (print distance)
            (dolist (x (get city 'neighbours))
               (setq queue (add_to_heap
                   (list!* (plus distance (cdr x))
                           (car x)
                           city)
```

```
                queue)))))
    (when (null queue) (error 0 "No route exists"))
    (setq distance (get destination 'distance))
    (while (not (eq destination source))
        (princ "Via: ")
        (print
           (setq destination (get destination 'previous))))
    (return distance)))
```

A test of this obviously needs some towns and some distances between them set up. The example here is rather small but still may show what can be achieved.

```
(de nput (source neighbours)
   (put source 'neighbours neighbours))

(nput 'Cambridge '((Bedford . 15) (Royston . 20)))
(nput 'Royston '((Cambridge . 20) (Watford . 30)
                 (London . 50)))
(nput 'London '((Royston . 20) (Watford . 25) (Oxford . 50)))
(nput 'Bedford '((Cambridge . 15) (Watford . 30)))
(nput 'Watford '((Bedford . 30) (Royston . 30)
                 (London . 25) (Oxford . 40)))
(nput 'Oxford '((Royston . 50) (Watford . 25) (London . 50)))

(find_route 'Cambridge 'Oxford)
```

The code as presented here is untidy in that it leaves all sorts of properties on the city-names that it processes. Specifically it leaves `distance` and `previous` information around. That would cause severe confusion, not to say incorrect results, if a second query were to be made. So a first task to develop this example code would be to clean that up.

Books on algorithms will explain alternative ways of arranging heaps, based on the use of arrays. In Lisp the function `mkvect` can make a vector (in other words a 1-dimensional array) and `putv` and `getv` can then access it. You could re-work the code to use the more standard array-based style of heap.

Those who are feeling seriously ambitious can look in a *big* algorithms book such as Cormen et al.[11] and find out about Fibonacci Heaps, a quite elaborate data structure specifically good for use in this application.

18.9 The `tak` benchmark

Lisp implementers and enthusiasts like to run benchmarks. An especially influential collection of tests was published by Richard Gabriel[17],

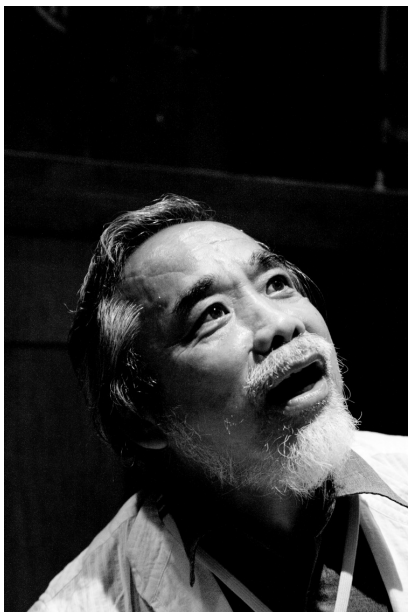

Figure 18.3: Ikuo Takeuchi (photo courtesy takai on flickr)

and it is worth going back to that to see how any single test can be measuring merely one aspect of system speed – and that timings across a range of carefully designed tests are needed before valid general conclusions can be drawn. However one tiny benchmark, known as `tak`, became perhaps especially popular. That may be because the code for it was truly short but it nevertheless ran for long enough to be a useful test.

The function used in `tak` originated with Ikuo Takeuchi, and is a ridiculously recursive function that busily calls itself merely performing trivial arithmetic operations. Indeed the only test it does is to compare two (small) integers to see which is bigger, and its only arithmetic is to subtract one from a value. The version in Gabriel's book is written as for Common Lisp, but the adaptation for the Standard Lisp dialect as used with `vsl` is not a difficult one. Note that because `tak` only ever uses small integers in its arithmetic the lower-level functions `ilessp` and `isub1` are used rather than the normal versions that could handle arbitrary-sized integers.

```
(de tak (x y z)
   (cond
      ((not (ilessp y x)) z)
      (t (tak (tak (isub1 x) y z)
              (tak (isub1 y) z x)
              (tak (isub1 z) x y))))))
```

Having defined out function we need to try it out, and see how long it tapes. The `vsl` function `time` returns a result in milliseconds that can be used to measure CPU time. The expected result of computing `(tak 18 12 6)` is just 7.

```
(setq a (time))
(tak 18 12 6)
(difference (time) a)
```

Using `vsl` on a Raspberry Pi gives a time of around 0.24 second. This is an astonishing result, since it is faster than most of the times recorded for special purpose Lisp hardware in Gabriel's book, and is only just beaten by large (and very expensive) mainframes. Only the legendary Cray supercomputer delivered a significantly better result – and these days even that is trumped when `vsl` is

run on a fast desktop machine. To give some idea of just how amazing this is I should note that `vsl` has been coded for simplicity not speed, and that the code is being run via an interpreter, while the results from Gabriel will be for highly tuned full-scale Lisp systems with state of the art compilers and even in some cases custom hardware support. With a Lisp system that is at least slightly better optimised (specifically the CSL one used with the Reduce algebra system[24]) the `tak` benchmark takes only 0.06 seconds on a Raspberry Pi, further showing how well modern hardware performs when compared with even truly expensive older computers.

This suggests that using a truly cheap computer system today puts one in a position to study problems that were cutting edge not very long ago.

If the Lisp-coded functions `lessp` and `sub1` that are capable of supporting big integers are substituted this cost increases by a factor of ten, but that is essentially alll down to the clumsy (if simple) way in which those versions are at present implemented.

Now of course having made this test there will be many ways to follow through. Perhaps the first is to track down and read almost anything that Richard Gabriel wrote, since much of it is entertaining as well as informative. His book on benchmarking has amazingly detailed information about the inner designs and working of the Lisp systems he was measuring, and insightful comments on what matters.

The next task would clearly be to try some more of his test cases. Some will be harder than others to map (fairly) from Commmon Lisp style so that they suit `vsl`. It would be possible to put `tak` into `vsl` as a built-in primitive coded in C. How much difference would that make to speed? When you get to considering a compiler for `vsl` you can see just how much difference to benchmark timings that makes.

There is also scope for investigating `vsl` and seeing exactly which parts of it contribute most to the time spent if various benchmarks, and seeing if refinements to the code can help. This test makes it abundantly clear that the existing simplistic `vsl`support for big integers is a time-sink.

I might further suggest additional historical research to find the speed, size and capability of the computers used in a range of groundbreaking past developments to compare then with what is now available. The tiny benchmark considered here may provide a starting point towards realising just how much it should be possible to do with even the cheapest option today, and that it does not just approach but often utterly dominates that which was available not very long ago.

18.10 Differentiation

Symbolic algebra was one of the early tasks that Lisp was designed for. The aspect of that considered here is differentiation, and algebraic formulae are represented as Lisp expressions in a very traditional way. Differentiation is always performed with respect to `x`, and the rules used are:

$$\begin{aligned}
\frac{dx}{dx} &\Rightarrow 1\\
\frac{dy}{dx} &\Rightarrow 0\\
\frac{d}{dx}(u+v) &\Rightarrow \frac{du}{dx}+\frac{dv}{dx}\\
\frac{d}{dx}(uv) &\Rightarrow u\frac{dv}{dx}+v\frac{du}{dx}\\
\frac{d}{dx}(u/v) &\Rightarrow (v\frac{du}{dx}-u\frac{dv}{dx})/v^2
\end{aligned}$$

where the second rule that is shown as defining the derivative of y should be interpreted as applying to all variables that are distinct from x and also all numbers. In Lisp terms to all atomic formulae other than the one for x.

Expressing these rules within a Lisp function is really rather easy, but illustrated how easy it is to find cases where composites of `car` and `cdr` such as `caddr` arise.

```
(de deriv (a)
   (cond
      ((eq a 'x) 1)
      ((atom a)  0)
      ((eqcar a 'plus)
         (list 'plus (deriv (cadr a))
                     (deriv (caddr a))))
      ((eqcar a 'difference)
         (list 'difference (deriv (cadr a))
                           (deriv (caddr a))))
      ((eqcar a 'times)
         (list 'plus
            (list 'times (cadr a) (deriv (caddr a)))
            (list 'times (deriv (cadr a)) (caddr a))))
      ((eqcar a 'quotient)
         (list 'quotient
            (list 'difference
               (list 'times (deriv (cadr a)) (caddr a))
               (list 'times (cadr a) (deriv (caddr a))))
```

```
            (list 'times (caddr a) (caddr a))))
      (t 'unknown)))
```

Here if an unrecognised operator occurs the function merely returns the literal `unknown`. It makes sense to test behaviour on a range of trivial examples before getting ambitious. One easy way to create a reasonably elaborate test is to use just these simple rules to form (say) the fourth derivative of $1/x$.

```
(deriv '(plus x y))
(deriv '(times x y))
(deriv '(quotient 1 x))

(deriv (deriv (deriv (deriv '(quotient 1 x)))))
```

Without any simplification of the result the result is amazingly clumsy and starts off as

```
(deriv (deriv (deriv (deriv '(quotient 1 x)))))
Value: (quotient (difference (times (difference (plus
(times (difference (plus (times (difference (plus
(times 0 1) (times 0 x)) (plus (times 1 0) (times 0 1)
)) (plus (times x 1) (times 1 x))) (times (difference
(plus (plus (times 0 0) (times 0 1)) (plus (times 0 1)
(times 0 x))) (plus (plus (times 1 0) (times 0 0))
(plus (times 0 0) (times 0 1)))) (times x x))) (plus
(times (difference (times 0 x) (times 1 1)) (plus
(plus (times x 0) (times 1 1)) (plus (times 1 1)
(times 0 x)))) (times (difference (plus (times 0 1)
(times 0 x)) (plus (times 1 0) (times 0 1))) (plus
(times x 1) (times 1 x))))) (plus (times (times x x)
...
```

If you inspect this it has an amazing number of sub-formulae such as `(plus (times 1 0) (times 0 1))` that would collapse dramatically when simplified.

A different differentiation program was included as one of the benchmarks in the Gabriel[17] book on Lisp performance. The code given there is explained as a Common Lisp version of a symbolic derivative benchmark written by Vaughan Pratt – the code here has been converted back out of Common Lisp into the dialect that `vsl` uses, but because the code only uses a rather small subset of Lisp this was not a difficult task. In many respects the most subtle issue is that the mapcar function takes its arguments in different orders in the two different Lisp traditions.

```
(de deriv!-aux (a) (list '/ (deriv a) a))
```

```
(de deriv (a)
   (cond
      ((atom a)
         (cond ((eq a 'x) 1) (t 0)))
      ((eq (car a) '+)
         (cons '+ (mapcar (cdr a) 'deriv)))
      ((eq (car a) '-)
         (cons '- (mapcar (cdr a) 'deriv)))
      ((eq (car a) '*)
      (list '* a
         (cons '+ (mapcar (cdr a) 'deriv!-aux))))
      ((eq (car a) '/)
         (list '-
            (list '/
               (deriv (cadr a))
               (caddr a))
            (list '/
               (cadr a)
               (list '*
                  (caddr a)
                  (caddr a))
            (deriv (caddr a))))))
      (t (error 1 'bad)))))
```

A single differentiation takes almost all time and so to get a reasonably reliable timing there is a test-harness that arranges to perform half a million simple differentiations. This number is arranged by looping 100000 times and calling the `deriv` function five times within the loop.

```
% Try once to verify results.
(deriv '(+ (* 3 x x) (* a x x) (* b x) 5))

(de run ()
   (prog (i)
      (setq i 100000) % Call deriv 500000 times
   loop
      (cond
         ((zerop i) (return nil)))
      (setq i (sub1 i))
      (deriv '(+ (* 3 x x) (* a x x) (* b x) 5))
      (deriv '(+ (* 3 x x) (* a x x) (* b x) 5))
      (deriv '(+ (* 3 x x) (* a x x) (* b x) 5))
      (deriv '(+ (* 3 x x) (* a x x) (* b x) 5))
      (deriv '(+ (* 3 x x) (* a x x) (* b x) 5))
```

```
      (go loop)))

(setq a (time))
(run)
(difference (time) a)
```

The `vsl` system has a `time` function that returns a measure of CPU time measured in milliseconds. The difference between a reading before and after the test is reported. Gabriel reports timings for this test on a number of machines, but with 5000 rather than 500000 calls. For instance on the "Lambda" special-purpose Lisp machine those 5000 calls took 3.62 or 6.40 seconds depending on options, and a VAX/780
took 23 seconds. These timings can really put the performance of modern hardware into perspective – on an fairly fast desktop pc the `vsl` timing is not much over 5 seconds for the full half million trials. A cheap machine such as a Raspberry Pi is slower than that but still dramatically faster than systems that were the mainstay of many Computer Science departments in the 1980s.

Figure 18.4: A VAX/780 (photo courtesy vax-o-matic on flickr)

The first test program here illustrates the need for simplification. This too was something that was observed very early in the history of Lisp. A thesis by Fenichel[15] from 1966 implemented a general rewrite system that could accept new rules such as

$$\begin{aligned} u + 0 &\Rightarrow u \\ 0/u &\Rightarrow 0 \end{aligned}$$

While the system described there could simplify things, my favourite quotation from Fenichel's thesis is

> *The current FAMOUS implementation presents a mean lethal dose of inefficiency. That is, approximately half of the problems given to FAMOUS have been solve so slowly that the users have lost interest in waiting for their solutions.*
>
> *The figure "one half", moreover is probably charitable. The system's major user is overmotivated and all of the system's users have been more interested in seeing what the system can do than in what the system can do with some predetermined problem. ...*

So not only is there a challenge to simplify but also there is a challenge to make things run at a sensible speed!

As you might imagine, once the early Lisp programmers had got differentiation under control their next target was integration...

18.11 Parsing

Some people will view Lisp's parenthesised input notation as barbaric. This example shows that the language can cope with a more traditionally human notation too. Specifically it presents a parser that can read in simple arithmetic expressions and convert them into Lisp form read for further processing. The code given here is only a page long and only copes with the four arithmetic operations of +, −, * and /. But purely at the cost of some more code it could extend first to support more operators, and then to cover the whole syntax of a normal-looking programming language.

The first function needed is one to read a "symbol". The version here maintains a variable called `cursym` that holds the most recent symbol that has been read, and it takes the next single character as the next whole symbol. It then skips blanks and newlines. There is no discrimination between names, numbers and operators, no arrangement for multi-character items and in general this is pared to be as simple as it possibly could be.

```
(de nextsym ()
   (let ((prev cursym))
      (setq cursym (readch))
      (while (or (eq cursym blank)
                 (eq cursym !$eol!$))
             (setq cursym (readch)))
      prev))

(de expression ()
   (let
      ((tree (term)))
      (while
         (or (eq cursym '!+) (eq cursym '!-))
         (setq tree (list (nextsym) tree (term))))
      tree))


(de term ()
   (let
      ((tree (factor)))
```

```
      (while
         (or (eq cursym '!*) (eq cursym '!/))
         (setq tree (list (nextsym) tree (factor))))
       tree))

(de factor ()
   (cond
      ((eq cursym lpar)
         (nextsym)
         (let
            ((tree (expression)))
            (nextsym)
            tree))
      (t (nextsym))))

(de parser ()
   (let ((cursym nil))
      (nextsym)
      (expression)))

(parser)
2+(a-b)*(4/x+7);
```

18.12 Expanding macros

```
(de macroexpand_cond (l)
   (cond
      ((null l) nil)
      (t (cons (macroexpand_list (car l))
               (macroexpand_cond (cdr l))))))

(de macroexpand (x)
   (cond
      ((atom x) x)
      ((not (atom (car x)))
         (cons (macroexpand (car x))
               (macroexpand_list (cdr x))))
      ((eqcar x 'quote) x)
      ((eqcar x 'cond)
         (cons 'cond (macroexpand_cond (cdr x))))
      ((or (eqcar x 'prog) (eqcar x 'lambda))
         (cons 'prog (cons (cadr x)
```

```
            (macroexpand_list (cddr x)))))
      ((eqcar (getd (car x)) 'macro)
         (macroexpand (apply (cdr (getd (car x)))
                             (list x))))
      (t (cons (car x) (macroexpand_list (cdr x))))))

(de macroexpand_list (l)
   (cond
      ((atom l) l)
      (t (cons (macroexpand (car l))
               (macroexpand_list (cdr l))))))

(cdddr (getd 'expand_dolist))

(macroexpand_list (cdddr (getd 'expand_dolist)))

(stop 0)
```

18.13 Compiling

```
% A compiler that compiles Lisp into C, but
% that makes no attempt at optimisation.

% comval is the central dispatch that takes any
% Lisp expression and creates C code that moves
% the value of it into a variable called "w".

(de comval (x)
   (cond
      ((symbolp x) (loadsymbol x))
      ((atom x) (loadliteral x))
      ((get (car x) 'compfn)
         (eval (list (get (car x) 'compfn)
                     (list 'quote (cdr x)))))
      ((eqcar (getd (car x)) 'macro)
          (comval (apply (cdr (getd (car x)))
                         (list x))))
      (t (loadargs (cdr x))
         (callfunction (car x) (length (cdr x))))))


% Literal values used within the function being
```

```
% compiled will live in a vector. This function
% keeps track of what will go in that vector.

(de findlit (x)
   (prog (w)
      (setq w (assoc x lits))
      (cond (w (return (cdr w))))
      (setq lits (cons (cons x nlits) lits))
      (setq nlits (add1 nlits))
      (return (sub1 nlits))))

% Loading the value of a variable involved loading
% (as a literal) the name of the variable and then
% accessing its value cell.

(de loadsymbol (x)
   (princ "    w = qvalue(elt(lits, ")
   (princ (findlit x))
   (printc "));"))

(de loadliteral (x)
   (princ "    w = elt(lits, ")
   (princ (findlit x))
   (printc ");"))

% loadargs merely arranges to push all the
% arguments onto a stack.

(de loadargs (l)
   (dolist (x l)
      (comval x)
      (printc "    push(w);")))

% After loadargs you can use callfunction to
% pop args into individual variables and call
% the function concerned.

(de callfunction (f nargs)
   (dotimes (n nargs)
      (princ "    pop(a")
      (prin (difference nargs n))
      (printc ");"))
   (loadliteral f)
```

```
   (princ "    w = (*(lispfn *)qdefn(w))(qlits(w), ")
   (princ nargs)
   (dotimes (n nargs)
      (princ ", a")
      (prin (add1 n)))
   (printc ");"))

% Now I will put in a collection of functions that
% provide special-case treatment. All "special forms"
% need this, and some other code can benefit from it
% in terms of efficiency.

(de comcar (x)
   (comval (car x))
% For simplicity I make taking CAR of an atom a
% fatal error here.
   (printc "    if (!isCONS(w)) disaster(__LINE__);")
   (printc "    w = qcar(w);"))

(put 'car 'compfn 'comcar)

(de comcdr (x)
   (comval (car x))
   (printc "    if (!isCONS(w)) disaster(__LINE__);")
   (printc "    w = qcdr(w);"))

(put 'cdr 'compfn 'comcdr)

(de comcond (x)
   (cond
      ((null x) (loadliteral nil))
      (t (let ((lab1 (gensym))
               (lab2 (gensym)))
         (comval (caar x))
         (princ "    if (w == nil) goto ")
         (print lab1)
         (comprogn (cdar x))
         (princ "    goto ")
         (print lab2)
         (prin lab1)
         (printc ":")
         (comcond (cdr x))
         (princ lab2)
```

```
         (printc ":")))))

(put 'cond 'compfn 'comcond)

(de comprogn (x)
   (cond
      ((null x) (loadliteral nil))
      (t (dolist (c x) (comval c)))))

(put 'progn 'compfn 'comprogn)

(de tidylits (a)
   (mapcar (reverse a) 'car))

(de localargs (l v)
   (cond
      ((null l) nil)
      (t (princ ", ")
         (prin (car v))
         (localargs (cdr l) (cdr v)))))

(de pushargs (names vars)
   (cond
      ((null names) nil)
      (t (princ "    push(qvalue(elt(lits, ")
         (princ (findlit (car names)))
         (printc ")));")
         (princ "    qvalue(elt(lits, ")
         (princ (findlit (car names)))
         (princ ")) = ")
         (prin (car vars))
         (printc ";")
         (pushargs (cdr names) (cdr vars)))))

(de popargs (l)
   (dolist (v l)
      (princ "    pop(qvalue(elt(lits, ")
      (princ (findlit v))
      (printc ")));")))

(de compile (fn)
   (prog (def pops lits nlits)
      (setq nlits 0)
```

```
      (setq def (cddr (getd fn)))
      (terpri)
      (princ "LispObject L")
      (princ fn)
      (printc "(LispObject lits, int n, ...)")
      (printc "{")
      (printc"    LispObject w, a1, a2, a3, a4;")
      (princ "    ARG")
      (princ (length (car def)))
      (princ "(""")
      (princ fn)
      (princ """")
      (localargs (car def) '(v1 v2 v3 v4))
      (printc ");")
      (setq pops
         (pushargs (car def) '(v1 v2 v3 v4)))
      (comprogn (cdr def))
      (popargs (reverse (car def)))
      (printc "    return w;")
      (printc "}")
      (terpri)
      (princ "lits: ")
      (print (tidylits lits))))


(compile 'cadr)

(compile 'pushargs)

% This example will generate INCORRECT code because
% at present this compiler does not have a 'compfn
% for PROG (or GO or RETURN).
(compile 'comprogn)
```

18.14 Simple graphics

It is often nice to be able to draw pictures. As it stands `vsl` does not have any directly built-in graphics capabilities, but it is easy to provide some using a helper program. The one shown here is called `wxplot` and is written in C++ using a graphics library called `wxWidgets`[14]. The purpose of this helpere is to recieve a stream of simple commands from `vsl` (or indeed anywhere else) and draw something based on them. The commands that it accepts each consist of a

command letter followed by twoi or three integers. The commands provide for selection of a background colour, selection of a colour to draw in, movement of the pen to an absolute position and drawing a straight line.

As indicated, the code that creates a window and draws in it has been coded using `wxWidgets`, but there are actually a large number of other graphical toolkits that could have been used almost equally as easily. So anybody who wishes could re-work the code here to use one of the others. Perhaps `Qt`[12] is one of the leaders, but it is larger and more elaborate than `wxWidgets`, and for current purposes small is good.

```
// wxplot.cpp                                  A C Norman, 2011
// Simple plotting using wxWidgets. Works with wxWidgets 2.8
// This code is subject to the BSD license from
// the accompanying file "bsd.txt".

#include "wx/wxprec.h"
#include "wx/wx.h"
#include <iostream>
using namespace std;

DECLARE_EVENT_TYPE(FROM_THREAD, -1)
DEFINE_EVENT_TYPE(FROM_THREAD)

class wxPlot : public wxApp         // The program as a whole.
{
public:
    virtual bool OnInit();
};

class InputThread : public wxThread // Where input is read.
{
public:
    InputThread(class PlotFrame *pf)
    {   owner = pf;
    };
protected:
    virtual ExitCode Entry();
private:
    class PlotFrame *owner;
};

class PlotFrame : public wxFrame    // A window to display.
{
```

```
public:
    PlotFrame(const wxString& title);
    void OnQuit(wxCommandEvent& event);
    void OnPaint(wxPaintEvent& event);
    void OnThread(wxCommandEvent& event);
private:
    int x, y;
    wxBitmap bitmap;
    wxMemoryDC bitmapDC;
    InputThread *inThread;
    DECLARE_EVENT_TABLE()
};

BEGIN_EVENT_TABLE(PlotFrame, wxFrame)
    EVT_MENU(   wxID_EXIT,             PlotFrame::OnQuit)
    EVT_PAINT(                         PlotFrame::OnPaint)
    EVT_COMMAND(wxID_ANY, FROM_THREAD, PlotFrame::OnThread)
END_EVENT_TABLE()

IMPLEMENT_APP(wxPlot)

bool wxPlot::OnInit()
{   PlotFrame *frame = new PlotFrame(wxT("wxplot"));
    frame->Show(true);
    return true;
}

PlotFrame::PlotFrame(const wxString& title)
    : wxFrame(NULL, wxID_ANY, title),
      bitmap(480, 400)
{   bitmapDC.SelectObject(bitmap);
    bitmapDC.SetBackground(*wxLIGHT_GREY_BRUSH);
    bitmapDC.Clear();
    bitmapDC.SetPen(*wxBLACK_PEN);
    bitmapDC.SelectObject(wxNullBitmap);
    x = 480/2; y = 400/2;
    wxSize winsize(480, 400);
    SetSize(winsize); SetMinSize(winsize); SetMaxSize(winsize);
    Centre();
    inThread = new InputThread(this);
    inThread->Create();
    inThread->Run();
}
```

```
void PlotFrame::OnQuit(wxCommandEvent& WXUNUSED(event))
{   Close(true);
}

void PlotFrame::OnThread(wxCommandEvent& event)
{   char c; int u, v, w;
    bitmapDC.SelectObject(bitmap);
    sscanf(event.GetString().mb_str(),
           "%c %d %d %d", &c, &u, &v, &w);
    switch (c)
    {
    case 'x': // x red green blue  Set background colour.
        bitmapDC.SetBackground(wxBrush(wxColour(u, v, w)));
        bitmapDC.Clear();
        break;
    case 'c': // c red green blue  Set colour to draw in.
        bitmapDC.SetPen(wxPen(wxColour(u, v, w)));
        break;
    case 'V': // V x y             Move to position x, y.
        x = 480/2 + u; y = 400/2 - v;
        break;
    case 'D': // D x y             Draw to position x, y.
        bitmapDC.DrawLine(x, y, 480/2 + u, 400/2 - v);
        x = 480/2 + u; y = 400/2 - v;
    }
    bitmapDC.SelectObject(wxNullBitmap);
    Refresh();   // Get window re-painted soon.
}

void PlotFrame::OnPaint(wxPaintEvent& event)
{   wxPaintDC dc(this);
    dc.DrawBitmap(bitmap, 0, 0);
}

wxThread::ExitCode InputThread::Entry()
{   string line;
    while (cin)
    {   getline(cin, line);
        if (line[0] == 'Q') break;
        wxCommandEvent msg(FROM_THREAD, GetId());
        msg.SetString(wxString(line.c_str(), wxConvUTF8));
        wxPostEvent(owner, msg);
```

```
    }
    return (wxThread::ExitCode)0;
}
```

```
% Using the wxplot application to draw a picture...

(de draw (x y)
   (princ "D ") (princ x) (princ " ") (printc y))

(de move (x y)
   (princ "V ") (princ x) (princ " ") (printc y))

(de circle (n)
   (prog (w)
      (setq w (quotient (times n 7) 10))
      (move n 0)
      (draw w w)
      (draw 0 n)
      (draw (minus w) w)
      (draw (minus n) 0)
      (draw (minus w) (minus w))
      (draw 0 (minus n))
      (draw w (minus w))
      (draw n 0)))

(de spider (n)
   (cond
      ((minusp n) nil)
      (t (circle n)
         (spider (difference n 10)))))

(de drawit ()
   (prog (a)
      (setq a (open "./wxplot" 'pipe))
      (wrs a)
      (spider 180)
      (printc "Q")
      (close (wrs nil))))

(drawit)
```

18.15 Turtle Graphics

"Turtle Graphics" is a style of creating drawings by imagining that you have a turtle that crawls around the floor with a pen attached so that it leaves a trail. It is associated with the Logo programming language and mechanical robot turtles that have perhaps most often been used in primary schools to introduce children to geometry and computation. The key idea it applies is that the turtle will respond to commands to turn left or right by various amounts and to crawl forward. Simulating this in `vsl` just involves converting these relative movements into standard Cartesian coordinates so that information can be sent to the `wxplot` drawing code used in the previous example.

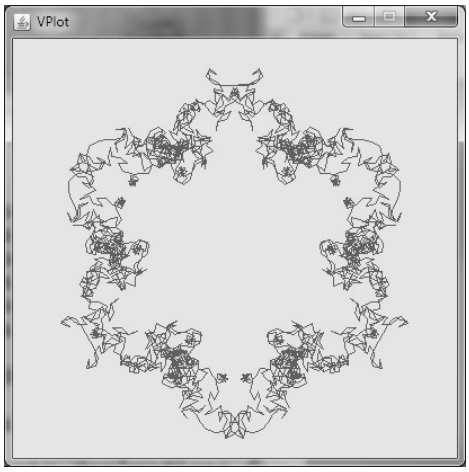

Figure 18.5: An example of graphical output from `vsl`

If a turtle alternates between moving forward by a small amount and turning right by a fixed small angle it will trace a path that is roughly circular. Making the angle turned smaller will smooth corners along the path and lead to a better circle, but the forward movement will need to be scaled down at the same time to keep the image size reasonable. If the angle turned by is changed at each step much more complicated images emerge. The code here uses a scheme that sets up a sequence of values for a variable `a` that increases as an arithmetic progression $0, 11, 22, 33, 44, 55, \ldots$ and then a second one `b` where at each stage it is increased by the current value of `a`. This value `b` is then used to instruct the turtle what angle to turn by, working in degrees.

```
(setq x 0.0)
(setq y 160.0)
(setq dir 0)

(de turn (n)    % Turn by n DEGREES (not radians)
   (setq dir (remainder (plus dir n) 360)))

(de draw (len)  % Draw line in the current dir
   (setq x (plus x
      (times len (cos (times dir 0.01745329252)))))
   (setq y (plus y
      (times len (sin (times dir 0.01745329252)))))
   (princ "D ") (prin (fix x))
```

```
   (princ " ") (print (fix y)))

(prog (o a b)
   (setq o (open "./wxplot" 'pipe))
   (wrs o)
   (printc "x 100 200 220") % Select background colour
   (printc "c 200 30 180")  % Select colour for plot
   (printc "V 0 160")       % starting point for pen
   (setq a 0 b 0)
   (dotimes (i 5000)
      (draw 8.0)
      (setq a (plus a 11))
      (setq b (plus b a))
      (turn b))
   (printc "Q")
   (close (wrs nil)))
```

There is great scope for experiment here using different starting values for the sequences `a` and `b`, and an increment other than 11 for the values that `a` takes. But perhaps the real challenge is to explain why the path that the turtle takes here, that seems so chaotic to start with, seems to close up on itself leaving a figure with threefold symmetry. The code given merely takes 5000 steps – is there a neat way of predicting ahead of time whether a path will close up and exactly how many steps are needed to achieve that? The book by Abelson and diSessa[1] may provide all sorts of other leads for worthwhile investigation, experimentation and fun in this area, and it certainly shows that Turtle paths are not just for pre-school children.

18.16 Mazes and Dungeons

```
(de adventure nil
   (printc '(Welcome to the Lisp Dungeon))
   (setq location 'start)
   (setq carrying nil)
   (display_position)
   (while (not (eq location 'finish))
      (make_move (readline))
      (display_position))
   'Congratulations)

(de display_position nil
   (terpri)
```

```
   (princ "at: ")
   (lpri (get location 'description))
   (dolist (item carrying)
      (lpri (list 'you 'are 'carrying item)))
   (dolist (item (get location 'objects))
      (lpri (list 'you 'see item)))
   (princ "exits:")
   (dolist (dir '(north south east west up down))
      (cond
         ((get location dir)
          (princ " ")
          (princ dir))))
   (terpri))

(de lpri (l)
   (dolist (x l) (princ x) (princ " "))
   (terpri))

(de make_move (word)
   (cond
      ((member word carrying) (drop word))
      ((member word (get location 'objects))
         (pick_up word))
      ((get location word) (move_to word))
      (t (lpri (list word 'not 'understood)))))

(de drop (word)
   (setq carrying (delete word carrying))
   (put location 'objects
      (cons word (get location 'objects)))
   (lpri (list 'dropped word)))

(de pick_up (word)
   (put location 'objects
      (delete word (get location 'objects)))
   (setq carrying (cons word carrying))
   (lpri (list 'got word)))

(de move_to (word)
   (setq location (get location word)))

(put 'start 'description '(The entry to a maze))
(put 'hall 'description '(A fine gothic hallway))
```

```
(put 'twist 'description '(A twisty passage))
(put 'dead 'description '(Dead end))
(put 'cave 'description '(Aladdin!'s cave))
(put 'finish 'description '(Castle splendid))

(put 'start 'north 'hall)
(put 'hall 'east 'twist)
(put 'twist 'up 'cave)
(put 'cave 'east 'dead)
(put 'cave 'west 'finish)
(put 'hall 'south 'start)
(put 'twist 'north 'hall)
(put 'cave 'down 'twist)
(put 'dead 'down 'cave)

(put 'start 'objects '(lantern lasergun))
(put 'twist 'objects '(keys map))
(put 'cave 'objects '(treasure))

(adventure)
lantern
lasergun
north
east
up
treasure
lasergun
west
```

18.17 Towards Common Lisp Compatibility

The Lisp dialect that this book uses is not Common Lisp, and many Lisp users will question this situation. So this section discusses some of the special capabilities that a full Common Lisp would require and looks at how they could be added to `vsl`. Actually each separate extension needed could end up involving a reasonably small amount of extra code, and so where there are particular Common Lisp features that matter to you it should be easy to provide them. But the total size of all extensions would not be small by any reasonable standards.

When this section gets to providing code examples that support Common Lisp compatibility it will concentrate on the features relied upon by Bark's Land Of Lisp book[5] since that is a text aimed at the beginner and containing worked

examples generally similar in scale to the ones presented here.

The discrepancies between Standard Lisp and Common Lisp come in a range of styles. There are cases where both dialects provide similar functions but use different names or specify arguments in a different order. Altering `vsl` to cope with things of that sort would be close to trivial. Common Lisp demands support for many additional data-types. These include rational and complex numbers, vectors that can hold bits, bytes, words or floating point values, character constants, user-specified record structures and a whole range of specialities that interact with the inner workings of the system (readtables, packages, pathnames, streams, ...). The data tagging scheme used in `vsl` could probably cope with most of these without too much trouble, and the trick used here to support big integers illustrates an escape route that could certainly make it possible to add as many fresh data-types as were ever needed. Of course if rational numbers such as $\frac{1}{3}$ and $\frac{17}{93}$ and complex numbers such as $1 + i$ are introduced then all the arithmetic code in `vsl` will need enlarging to cope. The dispatch code that compares the types of two values that are to be combined and selects a proper type for the result of an arithmetic operation will become significantly more complicated and risks becoming slower. When complex values are supported there are not just extra functions that are needed to support them (for instance the simple ones that extract the real and imaginary parts) but all the elementary functions – sines, cosines, square roots, logarithms and so on – have to be re-worked to behave properly in the complex domain. Common Lisp sets precise rules as to how branch cuts are to be handled in such cases, and completing a high quality implementation will be a serious (but fun) amount of work.

18.17.1 Spelling and Syntax

The fact that different Lisp dialects merely use different names for some functions is hardly a serious issue in any implementation. So for instance where Standard Lisp used the name `plus` for a function that can add up some numbers Common Lisp uses the name "+". But the use of a single character name does not alter the fact that addition is denoted with the operator first, as in `(+ 1 2 3)` rather than $1 + 2 + 3$.

Changing the exact names used by `vsl` to match Common Lisp is not a big deal. A little more work is required in that the Common Lisp reader (at least by default – it has elaborate mechanisms to make it re-configurable) treats most operator-like characters as if they were letters. Thus the input `(a+b*c)` would be a list of length 5 in Standard Lisp and a list of length 1 in Common Lisp. The symbol in that list would have had to be written as `!A!+!B!*!C` if for some reason you wanted it in Standard Lisp, because in addition to treating + and * as letters Common Lisp has a default behaviour where all input is converted to upper

case as it is read. The true Common Lisp name for two of the most fundamental Lisp functions are `CAR` and `CDR` not `car` and `cdr`. The main consequence of this is that with Common Lisp most of your output will unexpectedly appear in upper case unless you take special steps.

Thus to create `vcl` the first and rather easy things that had to be done to the `vsl` sources involved altering all the names of built-in functions to spell them in upper case and to adjust a few of their names. The tokeniser needing altering so that the escape character was "\" rather than "!", and so that most symbols could appear in names. The printing code needed corresponding changes because otherwise it would have felt obliged to insert escape characters in a lot of unnecessary places. Common Lisp has an unusual concept of "potential numbers" and so it is quite possible to have a symbol whose name starts with digits. This case is used for built-in functions such as `1+` (corresponding to Standard Lisp `add1`). Thus the tokenizer needs to start by assembling an almost arbitrary string of alphanumeric characters and then test to see if it matches the syntax of a number.

Common Lisp symbols are not recorded in a single object list but in a collection of "packages". This provides a way to avoid name-clash problems in large programs, but in general is not terribly important in small examples. However one aspect of this scheme must be supported. A name whose spelling starts with a colon (eg `:a` or `:b`) is a *keyword* and behaves as a constant. Keywords are used in some function definitions, and so `vcl` detects that syntax and treats it specially.

A complete Common Lisp would require a fairly elaborate re-programmable reader. That has not been addressed at all in `vcl`, but if the capabilities were seen as genuinely useful then extending the code to support them would be straightforward but fiddly and reasonably bulky code. And of course then if you wanted you could reconfigure it back to support the lexical structure as originally used in `vsl`!

18.17.2 Rational numbers

The second fairly simple extension to support Common Lisp is forced by the fact that as soon as you implement the four basic arithmetic operators you are faced with the fact that Common Lisp specifies that dividing one integer by another in general delivers a rational number. So even though a large proportion of real users are not at all liable to really want exact rational arithmetic one is almost forced to provide it from the start. In `vcl` this is done using the same mechanism that `vsl` uses to support arbitrary-precision integers. A symbol (in this case `~RATIO`) is reserved and any list-structure starting with it is treated as standing for a fraction. The dispatch code in arithmetic that already had to cope with combinations of large and small integers as well as floating point values now gets slowed down

with extra tests for rational numbers, and the obvious code then uses rules such as

$$\frac{a}{b} + \frac{c}{d} \quad \Rightarrow \quad \frac{ad + bc}{bd}$$

followed by a step the reduces the result to its lowest terms. Anybody wishing to complete the support for arithmetic would need to provide complex numbers as well, and this could obviously be done using just the same sort of mechanism. Full Common Lisp can support up to four different precisions of floating point number not just the one used here. Of course with each new data-type there is an opportunity to introduce a full range of functions that work with it. Since these will in general be straightforward to add in one at a time `vcl` does not rush to provide everything. Apart from such issues as that implementing high quality complex-valued elementary functions that match Common Lisp's ideas about where branch cuts must go is a delicate and time-consuming task really requiring significant numerical experience these will all be things that anybody using `vcl` could add in as and when they found a need.

18.17.3 `setf` and friends

In Standard Lisp and the old-style Lisp tradition a variable is updated using `setq`, the `car` and `cdr` parts of a `cons` can be overwritten using the functions `rplaca` and `rplacd`, properties can be attached to symbols using `put` so that `get` can retrieve them and there are pairs of accessor and mutator functions for array access, hash tables and anything else that needs them. Common Lisp seeks to unify all of these so that only the accessor functions are used as such. To update anything you would use `setf`. This takes a *place*, which is either a variable name or something that looks like a call to an acccesor function. It expands into whatever is needed to update that place. The user thus does not need to remember the names of all the mutator functions. With this instead of writing `(rplacd a b)` you would say `(setf (cdr a) b)`. Supporting this for `vsl` is mainly just a matter of listing all the accessor functions that are considered to indicate places, and arranging to map `setf` onto the relevant mutator. The code in `vcl.cl` at present only covers a basic few case, making this one of the many places where there is scope for the reader to extend and complete what they find here. A proper implementation will take care to arrange that the arguments to `setf` get evaluated in the expected order whatever convolutions the expanded version of the code involves. It will also arrange that the result delivered is as `setf` expectes – the value that was stored.

Associated with `setf` are some other macros that can update generalised places. When building lists it is frequently useful to put a new item on the from of a list or remove that first item. If the list is though of as a stack of values the names `push` and `pop` for these operations becomes reasonable. As with `setf`

itself these are fairly simple macros to implement, but a version more proper than the one in `vcl` would worry more about coping properly when the expressions used with them had side-effects and so the order of evaluation became critical: pedantry about this may sometimes lead to a much larger bulk of code in the expansion!

18.17.4 `defun` and `defmacro`

The next way in which Common Lisp goes beyond most earlier dialects of the language is in how functions indicate what arguments they specify. The relatively uncontroversial part of this allows for arguments that are optional and that possibly have default values to be used if the caller did not supply them. In Common Lisp this is indicated as follows:

```
(defun name (arg1
             &optional
             arg2
             (arg3 default3)
             (arg4 default4 supplied4)
             &rest
             arg5-and-up)
   ...)
```

Any arguments before the special word `optional` indicate ordinary arguments that must be supplied. After that word arguments are optional and they can be specified in several ways. As shown in the above example `arg2` is simply optional, and if the caller only passes one argument then `arg2` will end up with a value of `nil`. The next case shows `arg3` where `default3` stands for an arbitrary Lisp expression providing a default value (typically non-`nil`) to be used if the caller does not supply a proper value. The final case shown is for `arg4` where not only is a default provided, by the additional variable `supplied4` can be tested by the function and indicates whether the caller had supplied a genuine value or (conversely) the default had been used.

The word `&rest` allows functions to be called with excess arguments – these are collected into a list, and in this case would be made available to the function under the name `arg5-and-up`. An extreme case of the use of a `&rest` argument would be as in

```
(defun my-list (&rest l) l)
```

which would define a function that behaves like the built-in `list` one, by just returning the list of all its arguments. A function without `&rest` may not be called with more arguments than it expects.

The initial code in `vsl` does not support any of this, but to support some degree of Common Lisp compatibility a variant on it, `vcl`, has support for `&rest` implemented directly within the interpreter. The changes needed add less than 50 lines of code: it is necessary to detect the special marker `&rest` in an bound variable list and collect all subsequent actual arguments to be passed as a single list rather than as separate values.

The treatment of `&optional` is arranged by having the Lisp feature that defines functions expand definitions so that at a low level all arguments that are not mandatory are hanled by `&rest`. The macro that does this inserts code that checks just how long the `&rest` list was, and uses that information to fill in values for optional arguments.

The following example shows the code that `vcl` arranges to generate when starting from

```
(defun f (a &optional (b BB) (c CC c-p) &rest d) ...)
```

For reasons that are to do with the exact Common Lisp rules about scope (in particular a form used to provide a default value for an optional argument can depend on all the earlier arguments) the actual function defined uses freshly-generated new names for its actual arguments. Here these are shown as `G1` and tx G2. All the names that will be made available to the body of the function are introduced in a single `let*` clause. This is done for two reasons. The first is that forms that specify default values for optional variable are allowed to refer to any previous arguments. The second is that in full Common Lisp the body of a function can start with declarations that impact the treatment of all the arguments, and here those declarations now just have an effect on the variables bound by `let*`.

```
(de f (G1 &rest G2)
   (let* ((a   G1)
          (b   (if G2 (pop G2) BB))
          (c   (if G2 (car G2) CC))
          (c-p (and G2 (progn (setq G2 (cdr G2)) t)))
          (d   G2))
      ...))
```

As well as `&optional` and `&rest` it is also possible to use the word `&key` followed by specifers for keyward arguments. These can have default values and can be provided with associated variable the tell if a real value had been given or the default was used. The special feature of keyword arguments is that the caller may specify them in any order. So if the function definition had been

```
(defun f (a &key b (c 3) d e) ...)
```

then a call could be something like

```
(f 1 :d 4 :b 2 :e 5)
```

where a is an ordinary first argument, c end up with its default value and the other arguments are given the values shown after the keywords in the call.

The implementation of this is in fact a remarkably easy continuation of the style used for optional arguments. A sub-function is used to search the list of all arguments to find keywords and the exoabsion ends up much as follows:

```
(de find-keyword (k l)
   (cond
      ((or (null l) (null (cdr l))) nil)
      ((eq k (car l)) (cdr l))
      (t (find-keyword k (cddr l)))))

(de f (G1 &rest G2)
   (let* ((a G1)
          (G3)    ; a workspace variable used below
          (b (if (setq G3 (find-keyword :b G2)) (car G3) nil))
          (c (if (setq G3 (find-keyword :c G2)) (car G3) 3))
          (d (if (setq G3 (find-keyword :d G2)) (car G3) nil))
          (e (if (setq G3 (find-keyword :e G2)) (car G3) nil)))
      ...))
```

That does not arrange to detect additional unexpected keyords, and so it will not be as good at error reporting as would be ideal, but it is a good start.

Finally Common Lisp allows the use of a keyword &aux that can be followed by variables that are not actually arguments but are merely extra local variable to be made available within the function. In the macro-expansion as discussed here these trivially end up tagged on the end of the list of things that let* introduces.

What might have become clear is that use of these facilities carries a cost. If the value they bring is sufficiently high then that is clearly not a problem, and for functions that are only rarely called speed will not be an issue. However the excessive use of keyword arguments for various Common Lisp functions has attracted comments since the launch of the dialect. See for instance Brooks and Gabriel[9] and a quotation from Richard Stallman on the emacs developers list (http://lists.gnu.org/archive/html/emacs-devel/2003-08/msg00436.html) to the effect "I do not like the Common Lisp style of using keyword arguments for many common functions. I basically do not have a very high opinion of many of the decisions that were made in Common Lisp."

The vcl does not include any significant optimisations and so will be especially painful. But part of the purpose of this book is to leave projects for its readers – so finding ways of speeding things up in this area can be one of those.

defmacro defines macros in a suitable elaborate manner and needs the same sort of treatment.

So far I have not implemented thisin vcl.cl, but it should be easier than defun.

18.17.5 loop

Common Lisp provides a facility called loop that aims to make all sorts of iteration easy to specify. It is a magnificant example of how having a language like Lisp with a macro capability makes it possible to graft on almost anything. In the case of loop the addition that is added on is essentially a whole new language, with its own syntax and around seventy keywords. The explanation of what it is supposed to do in Common Lisp the Language[38] covers 38 pages. Even without serious concern for optimisation an implementation can be over 1000 lines of Lisp (the version used in clisp[23], and a more serious version (as in CMU Comon Lisp[29]) can be twice that size.

There is thus a conflict: the full version of loop is seen by many as absurdly overblown with its sub-language not even being in the broader style of Lisp. On the other hand *simple* cases of it can be very useful. So vcl supports a rather modest subset, where the exact limitations are only really discernable by inspecting the implementation in the file vcl.cl. This approach is perhaps in line with much of the rest of the book: the starter implementation can be viewed as the beginning of a project that would end up expanding it to support full Common Lisp.

As with the case of defun, the explanation given here will show the style of expansion needed rather than including full coverage of exactly how it is achieved.

There are a number of simple cases of loop that do things that almost everybody will need at least sometimes. In the examples shown here I will show the loop keywords in upper case so that there is no ambiguity as to which parts of the code are part of the loop syntax and which are names chosen by the user. First consider counting loops that simply obey a sequence of Lisp statements a number of times. This will represent a generalisation of the simpler dotimes iteration macro.

```
(LOOP FOR v FROM 5 TO 10 DO (print v))
(LOOP FOR v BELOW 4 DO (print v) (print (* v v)))
```

The first of these shows that the starting and finishing value for the user's variable v can both be specified. The second starts by default from zero, and in this case counts through 0, 1, 2 and 3. It illustrates that several Lisp forms may follow DO.

A plausible expansion for the first of these would have used prog in Standard Lisp, but Common Lisp deconstructs that using block to make something that can handle return and tagbody to support go. Thus the loop might render as

```
(block nil
   (let* ((v 5))
      (tagbody
```

```
   top   (when (> v 10) (return nil))
         (print v)
         (setq v (1+ v))
         (go top))))
```

with something rather similar for the second. It is certainly clear that use of `loop` has made it possible to express the iteration in a much more concise manner.

The next two cases for `loop` iterate over a list rather than over a range of numbers. In the examples I will show the list to be traversed as a fixed one, but any Lisp expression yielding a list can be used:

```
(LOOP FOR v IN '(a b c) DO (print v))
(LOOP FOR v ON '(a b c) DO (print v))
```

One might hope that it would be obvious what each of these does: the output from the first is just

```
A
B
C
```

(note that Common Lisp mapped lower case input to upper case interal names), while the second one will produce

```
(A B C)
(B C)
(C)
```

by letting `v` range over all the non-empty parts of the starting list. The expansion here has to be a little careful if it wants to avoid taking `car` of an empty list in some cases, and it certainly needs two variables – one to run down the list and the other to be the `v` specified by the user:

```
(block nil
   (let* ((w '(a b c))  ; internal variable
          (v nil))      ; for the user
      (tagbody
   top   (when (null w) (return nil))
         (setq v (pop w))
         (print v)
         (go top))))
```

The next complication is that as well as `DO` it is possible to write either `SUM` or `COLLECT`. The more interesting of these is `COLLECT`, which arranges to build a list from the values concerned. Thus a list of the first few squares might be built using

```
{LOOP FOR v UNDER 10 COLLECT (* v v))
```

A plausible expansion works by keeping two pointers to what will end up as the result list. The first always points to its head, while the other tracks its end. A new item can then be added to the end of the list using `rplacd` top overwrite the termination. To remain in a Common Lisp style this is expressed here using `setf`, but that just expands rather directly into the primitive operation it stands for.

```
(block nil
   (let* ((v 0)          ; counting variable
          (r (list nil)) ; list being created
          (p r))         ; end of list so far
      (tagbody
   top   (when (>= v 10) (go end))
         (setq p (setf (cdr b) (list (* v v))))
         (setq v (1+ v))
         (go top)
   end   (return (cdr r)))))
```

The implementation I have of a subset that includes many of the more sensible styles of use of `loop` is of the following form

```
(dm loop (u)
   (prog (bindings prelude endtests
          body updates postlude returnform)
      ;; decode the syntax of the LOOP in u.
      ...
      (return
         `(block nil
             (let* ,(reverse bindings)
                (tagbody
                   ,@(reverse prelude)
                top  (when (or ,@(reverse endtests))
                           ,@(reverse postlude)
                           (return ,returnform))
                   ,@(reverse body)
                   ,@(reverse updates)
                   (go top)))))))
```

The missing code builds up the list of bindings, the body of code to be iterated and everything else by pushing items onto the front of lists, hence the need for all the calls to `reverse`. It follows essentially the patterns of expansion shown in the various examples give earlier.

Just supporting these few cases of `loop` perhaps provides some value for programmers: anybody finding that they need much more can just extend the macros for themselves!

18.17.6 `labels`

The Common Lisp `labels` construction provides a way of introducing local function definitions. It is discussed here since it provides a further illustration of the way that Lisp macros can be used to extend the core language in extremely flexible ways, and because it provides an opportunity to show (reasonably sane) uses of `loop` in action.

The implementation here maps local definitions into global ones that use generated symbols as their names in place of the names originally provided by the user. The function `sublisfns` defined here is intended to take a list of substitutions that are to be made and apply them to the first item in any list within a form. The idea behind that is that the first item of any sub-list may be a function name, and by only substituting there it will be possible to avoid altering variable names or raw data. A proper version of this code shoulld probably expand macros as it went and understand all possible special forms, not just `quote`, but for simple cases it should suffice.

```
(defun sublisfns (l u &aux w)
   (cond
      ((atom u) u)
      ((eqcar u 'quote) u)
      (t (setq w (assoc (car u) l))
         (cons (if w (cdr w) (sublisfns l (car u)))
               (loop for v in (cdr u)
                     collect (sublisfns l v))))))
```

Then the main `labels` macro just needs to list the names of the new local functions it is to introduce and create private names for them (using `gensym`). It uses `sublisfns` to change everything to use these new names, uses `eval` to cause the functions concerned to get defined (at macro-expansion time) and is then done.

```
(~dm labels (u)
   (prog (defs names)
      (setq u (cdr u))
      (setq defs (pop u))
      (setq names
         (loop for v in defs
               collect (cons (car v) (gensym))))
      (loop for v in defs
```

```
        do (eval (cons 'defun (sublisfns names v))))
    (return (cons 'progn (sublisfns names u)))))
```

18.17.7 `format`

The fundamental printing functions (such as `print`) in Lisp print a single item. In terms of accessing information this is not a terrible limitatiion, since if you want to display elaborate messages you can put all relevant information into a list, as in

```
(print (list "The value of" x "is" (eval x)))
```

where your messages and data all get printed – albeit with a pair of parentheses surrounding everything. But for interaction with humans it can be desirable to avoid those parentheses. Common Lisp provides a function `format` for this. It takes a specification for a destination, a format string and then any data you want merged into the output. The format string contains escape sequences each starting with "~ showing where data is to be displayed mereged in with the raw text of the format. Perhaps the three most important escape sequences are `~A` which displays material in as plain way as possible (as in `princ`). So in particular a string will just have its contents displayed. Then `~S` is similar but displays strings with their surrounding quotes, and puts escape characters into any symbol whose name could otherwise cause confusion. Finally `~%` forces a new line. Thus it could be used as in

```
(format t "~%The value of %s is %s~%" x (eval x))
```

As with `loop` the designers of Common Lisp went utterly overboard in extending the capabilities of `format`. A full implementation embeds a whole new language within format strings. It can also arrange to convert numbers to spelt-out versions (but only in American English, because of its use of the word "billion") and to pluralise words (again just in English, noy say in French or Japanese). As well as being able to generate Roman numerals it can generate them in "old style" where 4 is shown as `IIII` rather than as `IV`. All for the benefit of those who will ever find these features useful!

The first argument to `format` indicates where the generated output should go. While in general it can be the identifier for an output stream who special cases are very commonly used: `nil` indicates that the result should be collected and returned as a string, and `t` that it should go to the terminal.

The version included here is a heavily cut-down one that only supports the basics, but for very many users those are what can give the greatest benefit. As with almost everything else here it provides a natural starting place for anybody ambitious to add extensions up to and including providing full support for everything the Common Lisp standard mandates, such as the example given in "Common Lisp the Language" that reads (after the line has been split to allow it to fit here)

```
"~:[{~;[~]~:{~S~:[->~S~;~*~]~:^ ~}~:[~; ~]~ ~{~S->~^ ~}
 ~:[~; ~]~[~*~;->~S~;->~*~]~:[}~;]~]"
```

and which is intended to help “print a Xapping Data type”, coming with the comment “Are you ready for this one?”.

```
(defun format (dest fmt &rest args)
   (prog (c a res o)
      (when dest
         (setq o (wrs (if (eq dest t) nil dest))))
      (loop for i from 0 to (upbv fmt) do
         (cond
            ((eq (setq c (getv fmt i)) '~)
               (setq c
                  (char-downcase (getv fmt (setq i (1+i)))))
               (cond
                  ((eq c '%)
                     (if (null dest)
                        (push $eol$ res)
                        (terpri)))
                  ((eq c '~)
                     (if (null dest)
                        (push '~ res)
                        (princ '~)))
                  (t (setq a (if (null args) nil (pop args)))
                     (cond
                        ((eq c '|a|)
                           (if (null dest)
                              (setq res
                                 (reversip2 (explodec a) res))
                              (princ a)))
                        ((null dest)
                         (setq res
                            (reversip2 (explode a) res)))
                        (t (prin a))))))
            ((null dest) (push c res))
            (t (princ c))))
      (return
         (cond
             ((null dest) (list-to-string (reversip res)))
             (t (wrs o) nil)))))
```

18.17.8 Lexical scoping

The way that `vsl` copes with variable bindings is that each symbol has a component that holds its current value. When a variable is bound either as an argument or as a local variable this value is saved, and at the end of the function call or block it is restored. The machanism is easy to implement in an interpreter of the sort shown here. However when a function accesses some variable that it itself has not bound it can lead to anomalies. Specifically the value that will be loaded as associated with a name will be the one corresponding to whatever function most recently started execution and bound it. Language purists (and the Common Lisp designers) believe that the value used should correspond instead to a binding that textually or *statically* encloses the point where the variable reference occurs.

Illustrations of this issue tend to require that functions be passed as arguments to other functions. This arises with cases such as `mapcar`. If dynamic rather than static binding is used, as in `vsl`, code such as

```
(de testcode (a l)
   (mapcar l '(lambda (x) (cons a x))))
(testcode '1 '(78 79 80))
```

that might be expected to return `((1 . 78) (1 . 79) (1 . 80))`. And indeed it will as written. However is instead of using a variable called `a` as the first argument for `testcode` and as the free variable in the lambda-expression one instead uses the name of a variable bound by `mapcar` all sorts of bad things happen. A user writing code that uses `mapcar` should not need to know what names it happens to use so as to avoid them! Static binding makes behaviour here more stable.

It may be proper to note that in the above example I have merely quoted the lambda-expression that is passed to `mapcar`, while in a full Lisp one should write `(function (lambda (a) ...))` to arrange that everything is handled properly. In Common Lisp this can be abbreviated as `#'(lambda (a) ...)`. Following its path of minimality neither of these notations is supported in `vsl`.

The cost of this is that an interpreter has to work rather harder if it is to implement static binding, and in certain extreme cases it can make compilation much more compicated even though in the majority of simple functions it is in fact easier to compile for. Older Lisp dialects tend to take the view that using "unusual" names for the variable used in those few functions that take other functions as arguments sidesteps the issue, and note that in Common Lisp use the most commen examples (such as `map` and `mapcar`) is liable to be replaced by use of the `loop` macro. Thus they ignore the matter.

Rather than show a re-implementation of Lisp using static binding those who wish to pursue this matter should start by doing a web search on "static and

dynamic scope", or read a serious textbook about compilerssuch as the Purple Dragon Book[2].

18.17.9 Additional data-types

The existing code explained in this book illustrated the addition of a big-integer data-type to `vsl`. Common Lisp expects to have more varieties of floating point number and complex arithmetic too. It has a much more elaborate idea of what strings and arrays must support, a range of variations on the idea of a hash table, a separate data-type for characters and in general all sorts of extra stuff. For each data-type it expects provision of a range of functions to support it. In addition to all the fixed extra types that it supports it provides a facility called `defstruct` that creates new types (each distinct from all existing ones) for user-defined sorts of data. Adding and or all of these tp `vsl` would actually be quite straightforward. The tag coding scheme it uses has a number of tag values currently spare, and the values produces by `defstruct` could naturally be represented as a variation on the existing vector type but using the first entry of the array to indicate its type. The messiest thing to support would be what Common Lisp refers to as "non-simple" arrays, but even that would not be a problem for anybody fopr whom performance was not a high priority – and on today's fast machines that can be almost everybody.

18.17.10 Other additional functions and macros

Basic Common Lisp provides rather more than 1000 predefined functions and macros, and as has been seen some of these are fairly complicated and their implementation will involve dozens of sub-functions. Furthermore any real implementation of the language will need to support a range of extensions since the official standard is now somewhat old and would seem limiting. On checking two widely used Common Lisp implementations (gcl[40] amd clisp[23]) one starts off with just over 7000 names in play and the other with almost 8500.

Part of the style and purpose of this book is to provide ideas for programming challenges and projects for the reader. Each little group of Common Lisp functions can represent a separable fragment of such a challenge, and even though getting everything implemented (and debugged) would be a rather large task there is much fun to be had identifying which section of the specification to work on next and gradually making progress towards the final goal of full compliance! A reasonable way to help set goals in this regard would be to identify some body of existing (Common) Lisp code and work towards providing just the functions it relies on. In many cases you can expect that to be a rather small subset of the full standard.

18.18 The Reduce algebra system

Amazingly and even though `vsl` is rather small, a substantial proportion of the over 300,000 lines of code making up the Reduce algebra system can be loaded and run on top of it. There are some significant limitations and restrictions, but nevertheless the resulting system can differentiate and integrate, solve equations, compute high-precision floating point values and run almost all of the large number of Reduce specialist modules. Getting all that code fully stable represents a substantial exercise, and at any particular moment there are liable to be bugs – for instance arising because one of the Reduce modules relies on some Lisp feature that has not yet been implemented or simulated in vsl.

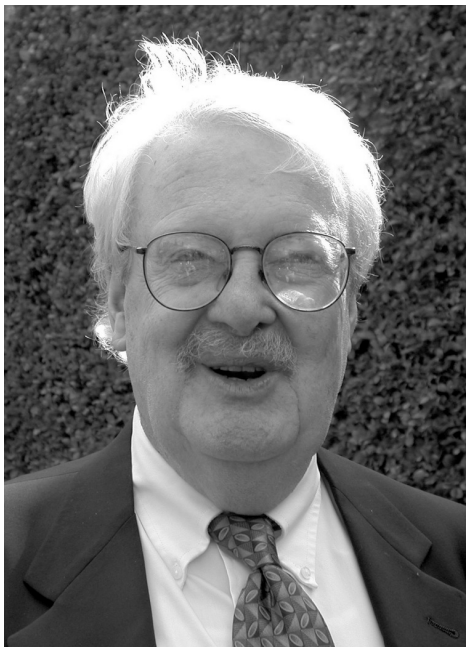

Figure 18.6: Tony Hearn, creator of Reduce

Reduce on `vsl` will also be slower than a version built on a full-sized Lisp that has a compiler. Actually it is *dramatically* slower! One of the larger bottlenecks is that `vsl` does not have a built-in function for checking whether symbol-names are in alphabetic order, and the version that works by using other Lisp features is amazingly clumsy. So one rather crude initial test showed `vsl` taking 300 times as long for a small calculation than one of the standard systems normally used with Reduce!

It is obviously not going to be feasible to include tens or hundreds of lines of Reduce source here. It is not even sensible to include the thousand lines of Lisp-code that provide a compatibility layer that builds `vsl` up to where it can be used to create a copy of Reduce. So instead here are instructions for fetching and building everything from the web.

Create a new directory and select it as current for your command-line.

First fetch a set of `vsl`sources using `subversion`. If you do not have that installed on your computer already you need to discover how to obtain it. On Linux this will be easy using a package manager (typically `yum` on Fedora or `apt-get` on Debian or Ubuntu). On Windows if you visit `www.cygwin.com` you can fetch their setup program and install their environment – ensuring that you install `make`, `gcc` and `subversion`, and probably `tex`.

Then you can go

```
U=https://reduce-algebra.svn.sourceforge.net
V=$U/svnroot/reduce-algebra/trunk
svn co $V/vsl
```

which should create a directory called `vsl` and put a collection of files in it. The above instructions build up the path at `sourceforge` to fetch this from in parts so you only have to type a modest amount on each line. The files you will fetch amount to about a megabyte, and so should not be a severe strain on anything.

So that you can build a version of the Reduce algebra system you should follow up the above subversion call with another (relying on the variable `$V` you set up to point to the subversion repository).

```
svn co $V/packages
```

This time you will end up with around 60 Mbytes in a directory called `packages`. That is the full set of sources for Reduce. Even though you are not liable to be able to make use of all of them the easiest recipe involves you fetching everything.

To build just `vsl` and then try it you can then go

```
cd vsl
make
./vsl
(oblist)
(stop 0)
```

For the build to succeed you will need to have the `gcc` C compiler installed, and to rebuild the manual the `pdflatex`[2] command is required. Again you may need to use a package manager to ensure that they are available.

In the above small example you verify that `vsl` will run by calling the function `(oblist)` to display a list of all `vsl`'s built-in symbols, then you use `(stop 0)` to exit from the system.

To try Reduce (see `http://reduce-algebra/sourceforge.net` for full information. In particular there is a manual hidden there) you go

```
make reduce
./vsl -ireduce.img
```

The "`make reduce`" step builds (much of) Reduce and saves the result in `vsl.img`. When you next start `vsl`it reloads that image file. At the time of writing this book you need to start Reduce manually by calling the Lisp function `(begin)`, but then you can try various algebraic examples.

For instance

[2]typically installed for you as part of some broader LaTeXpackage such as texlive.

```
df((x^2-1)^10, x, 10)/2^10/factorial(10);
```

should compute the 10th Legendre polynomial:

$$\frac{46189x^{10} - 109395x^8 + 90090x^6 - 30030x^4 + 3465x^2 - 63}{256}$$

The next example has had to have its output slightly adjusted to cope with line-length constraints here, but shows that the `vsl` version of Reduce can compute worthwhile integrals.

```
int(1/(x^6-1), x);

                     2*x - 1
( - 2*sqrt(3)*atan(---------)
                     sqrt(3)

                     2*x + 1           2
  - 2*sqrt(3)*atan(---------) + log(x  - x + 1)
                     sqrt(3)

         2
  - log(x  + x + 1) + 2*log(x - 1) - 2*log(x + 1))

/12
```

As a final example here, the input

```
in "alg.tst";
```

runs what was once used as the definitive test-case for Reduce and illustration of its capabilities.

This naturally leads into a series of truly major projects. Firstly to check how well `vsl` can be made to support *all* of Reduce. There are certain to be functions needed by examples not tried so far. Next to do a first pass of optimisation by identifying the most critical functions and providing C coded versions of them within `vsl`. Arbitrary precision integer arithmetic is very much needed, and comments about routes towards providing that were made earlier. Adding a compiler to `vsl` to bring its performance fully up to scratch could follow on. And finally some serious work would then be required to give `vsl` a nice windowed user-interface and to make Reduce work smoothly with it. By that stage `vsl` would have grown so it was no longer the small simple system it starts as but a fully-fledged Lisp.

Appendix A

Fetching, building and installing `vsl`

A.1 Raspberry Pi and other Linux platforms

The instructions here have been tested in a Raspberry Pi model B with a freshly installed copy of Debian Linux from the version `debian6-19-04-2012`. Everything will fit on a 4Gbyte SD card but it will be much more comfortable with one that is larger - but also significantly more pain to perform the initial installation of the operating system because of a need to re-size partitions so as to use al available space. It seems best to assume that the various Raspberry Pi starter guides and web-sites will provide careful blow-by-blow instructions for operating system installation, and merely provide a brief overview of the steps involved here. It is also probable that many users will buy an SD card that already has Linux installed on it. If that is Debian then you can skip the next few steps... and that could be a real relief!

Fetch `debian6-19-04-2012.img` It should be reasonable to fetch any later version, most plausibly by checking the downloads area on the Raspberry Pi web-site. Obviously if you use a newer version there could in principle be slight differences in your experince, so check any release notes or explanatory documentation associated with the version you fetch. You may potentially save yourself serious grief later by confirming that the image you downloaded has the correct check-sum. If you do not do that and end up with a corrupted download or a version where somehow a few bytes have been damaged then the behaviour later may be messy.

Use `win32diskmanager` to write the SD card ... if you used a Windows computer to fetch the debian image, or `dd` if you happen to have a Linux ma-

chine. On a Macintosh you may find `raspiwrite` does what you need. Detailed instructions should be available via the Raspberry Pi web site.

Re-size the partitions on the SD card This so that you can benefit from all the space on your SD card. Some Linux systems automatically arrange this when you first start them up, but at least the initial Debian release for Raspberry Pi does not. To re-size the partitions the smoothest scheme seems to be to use `gparted` from Linux – and if you have a computer that mostly runs Windows to boot it from a "live" Linux CD or DVD so you can do this. Again check websites associated with the particular operating system image that you download.

Move the SD card to your Raspberry Pi and check it It may at this stage be necessary to reconfigure it to work with the particular monitor you are using with it, or to reset it to start up with a graphical login rather than merely a text one.

You can perhaps see that obtaining a card that has been fully configured and is 100% ready for use is liable to make life a lot easier.

A.2 Windows

The instructions given here should apply for all versions of Windows from XP up as far as Windows 8. It will be possible to use either a 32-bit or a 64-bit version of the operating system without that having any major effect on results. But please note that despite the name of the operating system that is used this will be establishing a command-line based environment in which you will work! You will need an internet connection active while setting things up. Most of the steps that have to be taken only need to be done once, and so will be presented here as instructions to follow without huge amounts of explanation of the background. But you can follow them through happy in the knowledge that the software you are fetching is Free and Open Source so you will not have to pay for it, the licences that apply to it are all friendly, and if you become really keen and want to investigate everything in the most minute detail that is not just possible and legal but is encouraged.

The first thing to do will be to launch a web browser and navigate to `http://www.cygwin.com`. There you will find a link that lets you fetch a utility names `setup` that can install the "cygwin" tools and development environment. Download `setup` (or more pedantically `setup.exe`) and store it in some folder. It could make sense to save it as `c:\software\cygwin-setup\setup.exe`.

Now launch `setup`. It should announce itself as the "Cygwin Net Release Setup Program" and let you click on `Next` to proceed. Select "Install from Internet" then at the next stage you probably accept the default behaviour and make the Cygwin Root Directory `c:\cygwin` with the package installed for all users. At the next stage you can ask the installer to use the directory where you saved `setup` as your Local Package Directory – for instance this may be `c:\software\cygwin-setup`. There are options that need to be set if you connect to the internet via some proxy. If Internet Explorer works for you then you can copy and use the settings that it has. Next you need to select a download site – you should obviously scan the list of alternatives you are given and try to select one that is liable to be close to you. If your first choice gives trouble you can cancel the installation and try another.

At this stage you are presented with a screen headed "Select Packages" where you choose which components of cygwin to install. There is a search box towards the top left. Type the implausible name `mingw64-i686` into it and you then need to expand the `Devel` branch below by clicking on the `+` sign in a box. You should see a range of package names mostly tagged as `Skip`. If you click on the `Skip` you can cycle round selecting between the version numbers of any available versions, and `Keep`, `Uninstall` and `Reinstall` for packages you have already installed. Select the highest available version for `ming64-i686-gcc-core`. If you are looking ahead to trying all the `vsl` examples you should also select `mingw64-i686-gcc-g++`.

Next clear the search box and use it to find `make` and select it, and then `subversion`. Each should be in the `Devel` section. There are obviously an enormous range of other modules you could select, but it is not necessary to install all of them on your first try. You can just re-run `setup` at any later stage and get the ones you have already installed updated and pick additional ones you find you might need. So click at the bottom of the window to let `setup` install everything for you. It will download material (so just how long things take will depend on the speed of your internet connection) and you will see a series of progress bars as it installs everything. At the end it will offer to put an icon for cygwin on your desktop and in the start menu, and it might be a good idea to accept these suggestions.

If you are a developer involved with the initial work on `vsl` you also need to fetch `unifdef` and all of the packages that are parts of `texlive`. Well at least `texlive-collection` I will suggest `mingw64-i686-gcc-g` and `wget` too.

If all has gone well you will end up with a cygwin icon on your desktop. When you double-click it for the first time it does a little bit of final setting up and then presents you with a window running a terminal. There should (after a while) be a prompt visible there that ends in "`$`". The terminal window starts off by default showing white letters on a black background in a rather small window. I generally like to click on the icon at its top left and select `options` so I can change it to dark text on a light background using a font big enough that it fills a reasonable proportion of my screen. To check that things are working at all you might issue

the command "`ls /usr/bin`". `ls` is the command that displays a list of the files present in a directory, and `/usr/bin` is where `cygwin` keeps most of the commands it makes available to you (including `ls`!). If you see a reasonably long list of cryptic names that includes `ls` and `make` all is well. Note that many of the names may show up with a suffix `.exe` that is part of the full name that Windows knows them by – this is expected and not a problem. If things do not work properly you may need to try `setup` again.

There is one more once-only step you need to take. What you have done so far installs an environment within which `vsl` can be built and used. You now need to fetch `vsl` itself. So go[1]

```
svn co http://subversion.assembla.com/svn/rpistuff rpi
```

`svn` is a utility whose full name is `subversion` that can be used by collaborating groups to manage projects that several people may contribute to. Here it is being used to *Check out* (using the abbreviation `co`) the files associated with `vsl`. You should see a series of messages as individual files are fetched, and end up so that if you go just `ls` (without any specific location after the command) you are shown that you now have a directory called `rpi`. You can navigate into this by saying `cd rpi/vsl` and now `ls` should show a collection of files including ones called `vsl.c` and `Makefile`.

Now try to build `vsl` by issuing the command

```
cd rpi/vsl
make vsl
make vsl.img
```

The output from the first line using `make` should be similar to

```
$ make vsl
i686-w64-mingw32-gcc -O2 -Wall vsl.c -lm -o vsl
```

with no obvious messages or complaints. The second use of `make` digests the `vsl` library and saves it in a state where it can be available by default in the future. Now as a quick test try:

```
./vsl factorial.lsp
```

and if all has gone well you will see a table of factorials. If you manage to reach this stage you can be confident that you have a system ready to work with.

A.3 Macintosh

[1]The source listed here will NOT be the published one at all. As shown here it is just for collaborators!

Appendix B

Summary of functions in `vsl`

`!$eof!$` *predefined variable*

When the `read` function (or its relatives) detect and end of file condition it returns this value of this variable. So a code fragment such as `(eq (setq x (read)) !$eof!$))` will read a Lisp expression, store it in a variable called `x` and test if it was in fact an end of file marker.

`!$eol!$` *predefined variable*

This value of this predefined variable is a newline character, so if you use `readch` it may be convenient to compare the result against this. See `blank`, `lpar` and `rpar` for other character values where it is convenient to have a name rather than needing to enter the (escaped) character directly.

`!*echo` *variable*

When Lisp reads in any input the variable `!*echo` is inspected. If its value is non-nil then the characters that are read get echoed to the current Lisp output stream. This is often useful when reading from a file. But often if you are accepting input in an interactive manner from the keyboard you would prefer `!*echo` to be `nil`. So `vsl` arranges that if it is started up with a file to read specified on its command line it makes `echo` default to `t`, while if no command-line arguments are given it defaults to `nil`. This often results in comfortable behaviour, but the user is free to set the variable explicitly at any time to make things fit their needs.

`!,` *marker*

See the backquote entry.

`!,!@` *marker*

See the backquote entry.

` ***marker***

Lisp input can contain an ordinary Lisp expression preceded by a backquote mark. Within the expression various sub-parts can be marked with either a comma, or a comma followed by an "at" sign. This notation is commonly used when defining Lisp macros. The effect is as if a longer segment of Lisp had been written to create a structures that is the same shape as the one given, but with the comma-introduced sub-parts expanded. This may be shown with an example. The form `` `(a ,b c ,(car d)) `` might behave like `(list 'a b 'c (car d))`. The rules for backquote do not guarantee exactly what expansion will be generated – just that when it is executed it will construct the required structure. In `vsl` the code will in fact use many calls to `cons` rather than the neat use of `list` shown here. An embedded "`,@`" within a backquoted expression is expected to stand for a list, whose values are spliced into the eventual result. This `` `(a ,@(car b) c) `` might expand as `(append '(a) (append (car b) '(c)))`. In many some Lisp systems the full expansion is performed while reading the input. In `vsl` the reader leaves backquote and comma markers in the structure it returns, and macros expand those when it is time to execute them.

add1 ***function 1 arg***

`(add1 n)` is merely a shorthand for `(plus n 1)`, in other words it adds one to its argument. It is often useful when counting. See also `sub1`.

and ***function n args***

In `vsl` `and` is implemented as a special form. If evaluates its arguments one at a time, and returns `nil` if one of them evaluates to `nil`. If none of them yield `nil` its value if the value of the final argument. If you interpret `nil` as *false* and anything else as *true* then this matches a simple understanding of the of an operation that could reasonably called `and`. See also `or`. A different way of explaining `and` would be to give an equivalence: `(and a b c)` could be replaced by `(if a (if b c nil) nil)`, with similar expansion for cases with larger or smaller number of arguments. In `vsl` `(and)` (i.e. without any arguments) yields `t`.

append ***function 2 args***

If you have two lists then `append` can form their concatenation. So `(append '(a b) '(c d))` yields `(a b c d)`. The result will share structure with the second argument – a fact that only matters if you later use `rplaca` or `rplacd`. Some other Lisps may permit you to give `append` more than two arguments, and will then append all the lists into one, but `vsl` does not.

apply *function 2 args*

If you have either the name of a function or a lambda-expression (see `lambda` for more explanation) you can call it on a collection of arguments that are provided in a list. The function `apply` that does this is really there just because its capability has to be part of the Lisp interpreter. For instance since `cons` takes just two arguments you could invoke it by giving the symbol `cons` as `apply`'s first argument and a list of length two as its second: `(apply 'cons '(a b))`. This would return `(a . b)`. If the first argument to `apply` is a macro rather than an ordinary function this can be used to perform macro expansion. You should not try using `apply` on a special form (`fsubr`).

assoc *function 2 args*

An association list is a list of pairs, and each pair (`cons`) is though of as consisting of a key and a value. `assoc` searches an association list looking for a given key. If it find it then it returns the pair that contains it. Otherwise it returns `nil`. Thus `(assoc 'b '((a . 1) (b . 2) (c . 3)))` will return `(b . 2)`. The test for keys is make using the `equal` function.

atan *special form*

The arctangent function, working in radians. See `sin` and `cos`.

atom *function 1 arg*

Any Lisp object that is not a list or a pair – that is to say that could not have been created by the `cons` function, is known as an *atom*. Thus symbols, numbers, strings and vectors are all atoms. The function `atom` checks its argument and returns `t` (for *true*) if it is atomic and `nil` otherwise.

blank *predefined variable*

A predefined variable whose value is the symbol whose name is a single space character. One could otherwise refer to that symbol by writing `'!␣`, but many people find writing the word `blank` makes things clearer because it does not involve having a significant but non-printing character.

caaar *function 1 arg*

A range of names of the form `c`*xxx*`r` with the intermediate letters being either `a` or `d` are provided as functions that are merely combinations of uses of `car` and `cdr`. Thus `(caaar x)` means just the same as `(car (car (car x)))`, and `(caddr x)` means `(car (cdr (cdr x)))`. In `vsl` these are provided for up to three intermediate letters.

caadr *function 1 arg*
See caaar.

caar *function 1 arg*
See caaar.

cadar *function 1 arg*
See caaar.

caddr *function 1 arg*
See caaar. Can be though of as returning the third item in a list.

cadr *function 1 arg*
See caaar. Can be though of as returning the second item in a list.

car *function 1 arg*
Data structures in Lisp are made up with the various sorts of atom (symbols, numbers, strings and so on) as basic elements and with cons cells used to build them up into potentially large and complicated lists and trees. If you think of a structure as a list then car extracts its first element (and cdr its tail). If you think of it as a binary tree then car gets the left part and cdr the right. The basic identity is that (car (cons a b)) = a. See also cdr.

cdaar *function 1 arg*
See caaar.

cdadr *function 1 arg*
See caaar.

cdar *function 1 arg*
See caaar.

cddar *function 1 arg*
See caaar.

cdddr *function 1 arg*
See caaar.

cddr *function 1 arg*
See caaar.

cdr *function 1 arg*
(cdr (cons a b)) = b. See car.

char!-code *function 1 arg*

If you have a symbol or a string denoting a single character then the function `char!-code` will return a numeric code for it. The code used in `vsl` is ASCII so for instance the code for a blank character is 32, that for the digit "0" is 46 and a capital "A" is 65. See `code!-char` for conversion in the other direction.

close *function 1 arg*

When an input file has been opened for reading or writing you should use `close` it once finished with it. This is especially important for output files because it could be that some material will remain buffered and so will not be written until the file is closed. See `open`.

code!-char *function 1 arg*

`(code!-char 97)` would return a symbol whose name is the character with code 97 (in this case a lower case "a"). Similarly for other codes typically in the range 0 to 255.

compress *function 1 arg*

If you have a list of identifiers or strings then `compress` treats each as standing for its first character and returns the Lisp expression you would get if you read from a document that contained those characters. This would normally be used when each item in the input list was just a single character. Because in `vsl` the `compress` function is implemented by just involving `read` with input redirected from the list you can create symbols, numbers, strings and even lists this way. See `explode`.

cond *special form*

The primitive conditional operator in Lisp is `cond`. It is a special form (i.e. it does not evaluate its arguments in the standard way. Its use is as in

```
(cond
   (p1 e1a e1b e1c ...)
   (p2 e2a e2b ...)
   ...
```

where `p1`, `p2` etc. are predicates, and the sequence of expressions (for instance `e1a...`) that follow the first one that yields a non-`nil` value are computed. The result returned is the final thing that `cond` evaluates. There are many examples of uses of `cond` in the sample code here. Some people prefer to use `if` or `when` instead, but at least historically `cond` came first.

cons *function 2 args*

Lisp data is based on lists and trees, and `cons` is the key function for creating them. The term `cons`-cell is used for the object that the function creates. Such a cell has two components, is `car` and its `cdr`. The apparently strange names for these related to the architecture of an early computer on which Lisp was first developed. If you have a list structure `l` and an item `a` then `(cons a l)` is a list just one element longer than `l` was formed by putting `a` in front of the original. In Lisp it is much more expensive to attach a new item to the tail of a list. To do that would typically involve `(append l (list a))` and especially if `l` was long could be slow. So in Lisp it is normal to build up lists by successively adding items to the head using `cons`. See also `car`, `cdr` and `list`.

copy *function 1 arg*

It is normally only necessary (or indeed useful) to make a copy of a Lisp structure if you are then about to use destructive operations such as `rplaca` on the original, but this function is provided in case you do ever need to.

cos *function 1 arg*

It would be easy to extend the `vsl` implementation to provide a full set of mathematical functions, but to keep things small the initial version only provides a few key cases: `sin`, `cos`, `atan`, `exp`, `log` and `sqrt`. Each of these can be given either an integer or floating point argument but they always return a floating point result. The trigonometric functions work in radians rather than degrees.

de *special form*

To define a new function evaluate `(de name arglist body)`, the special form that is provided for this purpose. After you have defined something you could retrieve the definition you had set up using `(getd 'name)`.

deflist *function 2 args*

Sometimes when setting up data you need to perform a succession of `put` operations all using the same property name. `deflist` provides a shortcut so you can write something like

```
(deflist
   '((a A)
     (b B)
     (c C))
   'propname)
```

and have the same effect as

```
(put 'a 'propname 'A)
(put 'b 'propname 'B)
(put 'c 'propname 'C)
```

df *special form*

df is rather like de except that it allows you to defined a new special form. A special form must be defined as if it has just one argument, and this will receive the whole of the "argument" information from any call without any evaluation having happened. Often the body of a special form will thus wish to use eval to make evaluation happen. In most Lisp programs it will be unusual to introduce new special forms. See also dm for an alternative way (also sometimes controversial) for extending the syntax of Lisp.

difference *function 2 args*

Subtract one number from another. If either value is floating point the result will be floating point.

divide *function 2 args*

Divide one integer from another and return the cons of the quotient and remainder. The idea behind this function was that when integers are divided it is common to require both quotient and remainder, so having a one function to return both might be helpful. In Lisp systems that support very high precision arithmetic this can indeed save time. In vsl you will probably do as well calling quotient and then remainder.

dm *special form*

A Lisp Macro is something that when evaluated produces further executable Lisp to be its replacement. The special form introduced by dm can defined a new one. In general it will be sensible to define macros before you define any functions that use them. Some people believe that extensive use of macros can make your code harder to read and debug, and so would rather you did not use them at all.

do *macro*

A perhaps over-general form of loop can be specified by the do macro or its close cousin do!*. The structure of an invocation of it is

```
(do ((var1 init1 step1)
     (var2 init2 step2)
     ..)
    (endtest result ...)
    body
    ...)
```

and a concrete example is

```
(do ((x 10 (add1 x)))
    ((greaterp x 20) 'done)
    (print (list x (times x x))))
```

The difference between do and do!* is that the former processes all its initialisation and update of variables in parallel, while the latter acts sequentially. This is similar to the relationship between let and let!*.

do!* *macro*

See do.

dolist *macro*

Simple iteration over a lisp can be performed using the dolist macro, where a typical tiny example might be

```
(dolist (x '(1 2 3) 'result) (print x))
```

which prints the numbers 1, 2 and 3 and then returns the value result. In many cases you will merely omit the result part of the expression and then dolist will return nil.

dollar *predefined variable*

A predefined variable whose value is the symbol whose name is a dollar character. See blank for another similarly predefined name.

dotimes *macro*

Counting is easy with dotimes. It starts from zero, so

```
(dotimes (x 5 'result) (print x))
```

will print values 0, 1, 2, 3 and 4 before returning result.

eq *function 2 args*

If you wish to test two Lisp items for absolute identity then eq is the function to use. If you enter the same spelling for a symbol twice Lisp arranges that you get the same symbol, but it is possible – even probable – that strings or large numbers can fail to be eq even if they look the same. Two list structures are eq only if their top-level cons cells are identical in the sense that even if you received them via different paths they are the output from the same call to cons. See equal for a more expensive but perhaps more generous equality test.

eqcar *function 2 args*

`(excar a b)` is like `(and (not (atom a)) (eq (car a) b))`.

equal *function 2 args*

Wile `eq` compares objects for absolute identity, `equal` compares them to see if they have the same structure. `equal` understands how to compare big and floating point numbers, strings and vectors as well as lists. To illustrate the difference between the two functions consider

```
(setq a (cons 1 1000000000))
(setq b (cons 1 1000000000))
(eq a b)
(equal a b)
(eq (cdr a) (cdr b))
(equal (cdr a) (cdr b))
```

In each case `eq` returns `nil` while `equal` will return `t`. Although you should not in general rely on `eq` when comparing numbers, in `vsl` all small numbers are represented in a way that will allow `eq` to handle them reliably. If `vsl` is running on a 32-bit system the range is -268435456 to 268435455. If the system had been built for a 64-bit machine it is much larger.

error *function 2 args*

If a user wants to report that something has gone wrong it can call the `error` function. This is given two arguments, and they will be displayed in any message that is printed. See `errorset` for information about how to control the amount of information displayed when an error occurs.

errorset *function 3 args*

The default situation is that when `vsl` encounters and error it unwinds from whatever it was doing and awaits the next item of input from the user. The function `errorset` can be used to trap errors so that a program can respond or continue in its own way. It can also control how much diagnostic output is generated. A call `(errorset form msg trace)` will evaluate the Lisp expression `form` rather in the way that `eval` would have. If there is no error it returns a list of length one whose element is the value that was computed. If the evaluation of `form` failed then `errorset` returns an atom rather than a list, so its caller can be aware of the situation. The argument `msg` can be non-`nil` to indicate that a short (typically one line) report is displayed on any error. If `trace` is non-nil then a report showing the nesting of function calls will also be shown. If both arguments are `nil` then the error and recovery from it should be silent. See `eval` for a sample use.

eval *function 1 arg*

An approximation to how Lisp interacts with the user is

```
(while t
    (errorset '(print (eval (read))) t t))
```

where eval takes whatever form was read and evaluates it. The eval function (and its relative apply) can be used anywhere in Lisp code where a data structure needs to be interpreted as a bit of Lisp code and obeyed.

exp *special form*

The exponential function. See log.

expand *function 2 args*

In some Lisp implementations it would be useful (for instance for efficiency) to transform some uses of functions taking an indefinite number of arguments (for instance plus) into sequences of calls to versions taking just two arguments. The expand function is intended to help with this. Its first argument is a list of expressions, the second a (two argument) function to be used to combine them. For instance (expand '(a b c) 'plus2) will yield (plus2 a (plus2 b c)). This could be useful if plus were to have been implemented as a macro expanding to multiple uses of plus2 rather than as a special form, and if vsl provided the two-argument function concerned (which at present it does not!).

explode *function 1 arg*

Any Lisp item can be processes as by prin but with the resulting sequence of characters being collected as a list rather than being printed directly. explode does this, while explodec behaves like princ. So (explode '("a" . 3)) will be (!(!" a !" ! !. ! !3 !)). explode can be useful to find the sequence of letters making up the name of a symbol (or just to make it possible to see how many there are).

explodec *function 1 arg*

See explode

expr *symbol*

The function getd can retrieve the function definition (if any) associated with a symbol. The value returned is (type . value) where the type is one of the symbols expr, subr, fexpr, fsubr or macro. The case expr indicates that the function is a normal-style function that has been defined in Lisp. The value information following it in the result of getd is the Lisp structure representing it. fexpr is for special forms

defined in Lisp (using `df`. `subr` and `fsubr` and ordinary and special functions that have been defined at a lower level than Lisp (in other words things that form part of the `vsl` kernel). `macro` marks a macro as defined using `dm`. With the default `vsl` image `(getd 'caar)` returns `(expr lambda (x) (car (car x)))`.

f *predefined variable*

`f` is a variable pre-set to have the value `nil`. This exists because at one stage people tended to want to use `t` for *true* and `f` for *false*. In most cases it will be safer to use `nil` directly if that is what you mean, and attempts to use `f` as a definitive denotation of *false* cause trouble when you try using the name as an ordinary variable, as in `(de fff (a b c d e f g) ...)`.

fexpr *symbol*

See `expr`.

fix *function 1 arg*

If you have a floating point number and want convert it to an integer you can use the function `fix`. It truncates the value towards zero while doing the conversion.

fixp *function 1 arg*

The predicate `fixp` tests if its argument is an integer, and if it is it returns `t`. See also `numberp` and `floatp`. You are permitted to test any Lisp object using `fixp` and note that `(fixp 2.0)` will be `nil` because 2.0 is a floating point number even if its value is an integer.

float *function 1 arg*

Converts from an integer to a floating point number.

floatp *function 1 arg*

Test if an object is a floating point number. See `fixp`.

fsubr *symbol*

See `expr`.

gensym *function 0 args*

Sometime in a Lisp program you need a new symbol. One that is certain not to clash with any others you may have used already. `(gensym)` will create a fresh symbol for you. Such symbols should be though of as being anonymous. In `vsl` they do not even have a name unless and until you print them. At that stage a name will be allocated, and it will be of one from the

sequence g001, g002,.... However if you type in the characters g001 that will not refer to the generated symbol that was displayed that way – you will get an ordinary symbol that you may confuse with the gensym but that Lisp will not.

geq *function 2 args*

This is a predicate that returns t if its first argument is greater than or equal to its second. Both arguments must be numbers. See leq, greaterp and lessp.

get *function 2 args*

Every symbol has a property-list, which can be retrieved directly using plist. The main functions for saving and retrieving information on property lists are put and get. After you have gone (put 'name 'tag 'value) a call (get 'name 'tag) will return value. See also remprop. An extended version of the library could define functions flag, flagp and remflag to store flags rather then more general properties, but vsl only supplies the basics.

getd *function 1 arg*

See expr.

gethash *function 2 args*

If h is a hash table then (gethash 'key h) retrieves the value stored in it under the indicated key by some previous call to (puthash 'key h 'value). See mkhash, puthash and remhash.

getv *function 2 args*

If v is a vector then (getv v n) returns the nth element of if. See mkvect, putv and upbv.

go *special form*

See prog.

greaterp *function 2 args*

(greaterp x y) is true if x and y are numbers with x larger then y. See geq, lessp and leq.

if *macro*

The fundamental conditional form in Lisp is cons, but for convenience the macros if and when are supplied. if takes two or three arguments. The first is a predicate – the condition that is to be tested. The next is the value to return, while the last is the result required if the condition is *false* and

defaults to `nil`. `when` also takes a predicate, but all its further arguments are things to be obeyed in sequence if the condition holds. Thus `(when p a b c)` behaves like `(if p (progn a b c) nil)`.

`input` ***symbol***

See `open`.

`lambda` ***symbol***

Some people will believe that `lambda` is the key symbol standing for the essence of Lisp. Others view it as a slight curiosity mostly of interest to obsessive specialists. It introduces a notation for a function that does not require that the function be given a name. This is inspired by Alonzo Church's λ-calculus. The denotation of a function is a list with `lambda` as its first element, then a list of its formal parameters, and finally a sequence of values that are to be evaluated when the function is invoked. So `(lambda (x) (plus x 1))` is a function that adds one to its argument. If you try writing lambda expressions with bodies that refer to variables other than their formal arguments then you will need to read and understand the section of this book that discusses deep and shallow binding strategies in an implementation of Lisp. This issue in fact arises with named functions defined using `de` as well.

`last` ***function 1 arg***

If you have a list then `last` can return the final element in it. Recall that the first item in a list can be extracted using `car`, and note that `last` is going to be slower, so where possible arrange what you do so that you access the start of your lists more often than their ends.

`lastcar` ***function 1 arg***

This function is just like `last` except that if `last` is called on an empty list if reports an error, while `lastcar` merely returns nil.

`leftshift` ***function 2 args***

Take an integer value, treat it as a bit-pattern and shift that leftwards by the specified amount. Return the result as an integer. Generally `(leftshift x` n`)` has the same effect as multiplying `x` by 2^n. See `rightshift`.

`length` ***function 1 arg***

This function returns the length of a list. If its argument is `nil` it returns 0.

`leq` ***function 2 args***

A test for "less than or equal". See `geq`, `greaterp` and `lessp`.

lessp ***function 2 args***

A test for "less then". See `geq`, `greaterp` and `leq`.

let ***macro***

Sometimes it is convenient to introduce a new name for some result you have just computed and are about to use. The `let` macro provides a way to do this. So as an example where four temporary values are being introduced, consider

```
(let ((u (plus x y))
      (v (difference x y))
      (xx (times x x))
      (yy (times y y)))
   (list (difference xx yy)
         (times u v)))
```

In really old fashioned Lisp this would have been achieved using `prog` as in

```
(prog (u v xx yy)
    (setq u (plus x y))
    (setq v (difference x y))
    ...
    (return (list (difference xx yy) ...)))
```

and yet another scheme would use an explicit lambda-expression

```
((lambda (u v xx yy)
    (list (difference xx yy) (times u v)))
 (plus x y)
 (difference x y)
 (times x x)
 (times y y))
```

Of all these the version using `let` is liable to be the clearest and neatest. Actually the versions using `prog` and `lambda` can have different behaviours sometimes. `let` and `lambda` introduce all their new variable simultaneously, and so the definition given for a later one can not depend on an earlier one. `let!*` is like `let` but introduces one variable at a time so that subsequent ones can depend on it, and that is closer to how the naive use of `prog` would work. Thus

```
(let!* ((x2 (times x x))
        (x4 (times x2 x2))
        (x8 (times x4 x4)))
   (times x x4 x8))
```

returns the thirteenth power of `x`, while if `let` had been used rather than `let!*` you would have received an error message about `x2` being undefined that arose when it was used to define `x4`.

`let!*` ***macro***

See `let`.

`lispsystem!*` ***predefined variable***

It is sometimes useful to allow Lisp code to adapt based on knowing something about the particular Lisp implementation it is running on. In Standard Lisp (and hence `vsl`) environment information is provided in a predefined variables called `lispsystem!*`. In `vsl` the only information put there is a symbol `vsl` to identify the Lisp system in use. But it would be easy to extent the code in `vsl.c` to put in whatever extra information about the host computer anybody felt might be relevant.

`list` ***special form***

The fundamental function for building Lisp data-structures is `cons`, but by convention a list is a chain of `cons` cells ending with `nil`. Created in the fundamental manner a list of length 4 might be built using `(cons 'a (cons 'b (cons c (cons 'd nil))))`. That is correct but clumsy!. The special form `list` accepts an arbitrary number of arguments and forms a list out of them: `(list 'a 'b 'c 'd)`. It will therefore be common to use one call to `list` in place of multiple uses of `cons` whenever possible.

`list!*` ***special form***

The structures created using `list` are always automatically provided with a `nil` termination. Sometimes a list-like structure is wanted with some other end. This may arise for instance when putting multiple items onto the front of an existing list. The function `list!*` can achieve this, and taking an example that puts 4 items ahead of the termination, the long-winded form `(cons 'a (cons 'b (cons c (cons 'd 'e))))` could be replaced by the much more concise `(list!* 'a 'b 'c 'd 'e)`. As a special case `list!*` with just two arguments degenerates to being exactly the same as `cons`.

log *special form*

The (natural) logarithm of a value. See exp.

logand *special form*

If integers in Lisp are represented in binary form (and for those who want the full story, negative values in two's complement) then a number can be thought of as a string of bits. logand takes an arbitrary number of integers and performs an logical "and" operation on each bit position, so that a "1" is present in the output only all of the inputs have a "1" in that position. The result is returned as an integer. See also logor, lognot and logxor for operations, and leftshift and rightshift for re-aligning bits.

lognot *function 1 arg*

See logand. This function negates each bit.

logor *special form*

See logand. This function yields a "1" when any input has a "1" in that place.

logxor *special form*

See logand. This yields a "1" if an odd number of inputs have a "1" in the corresponding place, and is "exclusive or".

lpar *predefined variable*

A predefined symbol whose value is the symbol whose name is a left parenthesis. See also rpar.

macro *symbol*

See expr.

map *function 2 args*

There are a number of functions whose names begin with map. These take two arguments, on a list and the second a function. Each of them traverses the list calling the given function for each position on it. map, maplist and mapcon each pass first the original list and then each successive tail of it. mapc, mapcar and mapcan pass the successive items in the list.

In each case the three variants do different things with the results returned by the function. map and mapc ignore it and in the end just return nil. This is only useful if the function that is provided has side-effects. For instance it might print something. maplist and mapcar build a new list out of the results, and so the overall result is a list the same length as the input one. Finally mapcon and mapcan expect the function to return nil or some list, and they use nconc to concatenate all those lists.

Many people find `mapc` is the most useful, but then that the `dotimes` macro (which achieves a similar effect) is easier to use: compare

```
(setq a '(1 2 3 4))
(mapc a '(lambda (x) (print (times x x))))
(dolist (x a) (print (times x x)))
```

which achieve similar effects.

mapc *function 2 args*
See `map`.

mapcan *function 2 args*
See `map`.

mapcar *function 2 args*
See `map`.

mapcon *function 2 args*
See `map`.

maplist *function 2 args*
See `map`.

minus *function 1 arg*
Negates a number.

minusp *function 1 arg*
Tests if a value is a negative number. Note that in `vsl` it is legal to call `minus` with an argument that is not even a number, in which case it will return `nil` to indicate that it is not negative, but in many other Lisp dialects you should only give `minusp` numeric input.

mkhash *function 1 arg*
The hash-table capability built into `vsl` uses `(mkhash n)` to create a table, `puthash` to insert data and `gethash` to retrieve it. `remhash` can remove data. The argument to `mkhash` is used to control the size of the table, and might reasonable by chosen to be a fifth or a tenth of the number of keys you expect to store. Searches within hash tables are based on `eq`-style identity and are expected to be faster than various alternative (if simpler) schemes that could be considered.

mkvect *function 1 arg*

In vsl a call (mkvect n) creates a vector where subsequent uses of putv and getv can use index values in the range from 0 to n. This results in the vector having $n + 1$ elements. A whole vector counts as an atom in Lisp, not as a list. If v is a vector then (upbv v) returns its upper bound – the largest subscript legal for use with it.

nconc *function 2 args*

Given two lists, nconc concatenates them using a rplacd on the final cell of the first list. This avoids some extra storage allocation that append would have had to make, but overwrites part of the first list, and unless used with sensitivity that can cause trouble.

neq *function 2 args*

(neq a b) yields exactly the same result as (not (equal a b)).

nil *predefined variable*

The symbol nil is used in Lisp to denote an empty list, or to mark the end of a non-empty one. It is used to mean *false*, with anything non-nil being treated as *true*. The value of nil is nil. In some Lisp systems (but not this one or its close relatives) it is arranged that car and cdr may accept nil as an argument and yield nil. Here you are not allowed to do that so (cdr nil) will report an error just as (cdr 'any_other_atom) would.

not *function 1 arg*

When a value is being thought of as a truth-value the function not can be used to invert it. Because *false* is represented by nil it turns out that not behaves identically to null.

null *function 1 arg*

Tests if its argument is nil. Often used to detect when a list is empty.

numberp *function 1 arg*

Returns t if its argument is a number. See also fixp and floatp that check for specific sub-categories of numbers.

oblist *function 0 args*

The term "object list" is historically used in Lisp to refer to the symbol table that keeps track of all the identifiers that are in use. Its purpose is to arrange that every time you enter the same sequence of characters you get the same symbol back. The function (oblist) returns a list of all symbols in this table. This table of symbols important to Lisp was started by taking the output from oblist and sorting and formatting it. The number of symbols

in the object list gives some idea of the size of the Lisp implementation. With `vsl` there are a couple of hundred symbols known before you start adding more. With the C and the Java coded implementations of Standard Lisp used with the Reduce algebra system there are around four times as many.

onep *function 1 arg*

Test it its argument is 1 or 1.0. See `zerop`.

open *function 2 args*

To access data in a file you first `open` the file. A file can be opened either for reading or writing, as in `(open "input.dat" 'input)` or `(open "newfile.out", 'output)`. In each case `open` returns an object that can be passed to `rds` or `wrs` to select that stream for use. When finished with any file that has been open should be tidied up by handing its descriptor to the `close` function. As well as providing access to files, `open` can be used to launch another program, with Lisp output to the associated stream made available to that program as its standard input. This is done using the construction `(open "programname" 'pipe)`.

or *special form*

See `and`.

output *symbol*

See `open`.

pipe *symbol*

See `open`.

plist *function 1 arg*

Every symbol has a property-list and the `plist` function returns it. Normally this will only be used as a matter of interest, since `put` and `get` are the proper functions for storing and retrieving information from property lists.

plus *special form*

Adds an arbitrary number of values. If any one of them is floating point the result will be floating point.

preserve *function 0 or 1 arg*

If you call `preserve` a copy of the current status of everything in your Lisp world is written to a file called `vsl.img`. When `vsl` next starts it reloads this file (unless the `-z` command line option is used). This capability

can be used to build an image file containing all the definitions and settings that make up a program so that when vsl is started they are all immediately available.

prettyprint *function 1 arg*

The ordinary print functions in vsl fit as much on a line as they can. In contrast prettyprint attempts to make its output more human-readable by indenting everything in a systematic manner. So of you want to print out some Lisp code in a format where it is easier to read it may be useful.

prin *function 1 arg*

The family of print functions supplied with vsl consists of prin, princ, print and printc. The basic behaviour of each is that they print their argument to the current output stream. The versions with a "c" omit any escape marks, and when printing strings do not print quote marks, and so the output is perhaps nice for a human reader but could not be presented back to Lisp. The ones without a "c" insert escapes (exclamation marks) in names that include characters other than letters and digits, and do put quote marks around strings. The versions with a "t" terminate the output line after displaying their argument, so that a sequence of calls to prin display all the values on one line, while print puts each Lisp form on a separate line. See terpri and wrs.

princ *function 1 arg*

See prin.

print *function 1 arg*

See prin.

printc *function 1 arg*

See prin.

prog *special form*

prog feels like an archaic feature inherited from the early days of Lisp, and provides a range of capabilities. Firstly it introduces some local variables, then it allows for the sequential execution of a sequence of Lisp forms, with a labels and a go statement to provide control. Finally a prog block only returns a non-nil value if the return function is called within it to provide one. Here is an illustration of the use of these to compute and print some Fibonacci numbers and then return the symbol finished

```
(prog (a b n c)
   (setq a 0 b 1 n 0)
```

```
top(when (greaterp n 10) (return 'finished))
   (setq c b b (print (plus a b)) a c n (add1 n))
   (go top))
```

progn *special form*

There are a number of contexts in Lisp where you can write a sequence of expressions and these are evaluated in turn with the result of the final one as the overall result. `progn` can provide this capability in any other situation where it is useful.

psetq *macro*

See `setq`, but `psetq` arranges to evaluate all the new values before updating any of the variables. A typical use of it would be to exchange the values in two variables, as in `(psetq a b b a)`.

put *function 3 args*

See `get` and `deflist`.

puthash *function 3 args*

See `mkhash`, `gethash` and `remhash`.

putv *function 3 args*

See `mkvect` and `getvec`.

quote *special form*

Normally each sub-part of a Lisp program will be evaluated – that is to say treated as program not data. The special form `quote` protects its argument from evaluation and so is used when you wish to specify data. This is so common and so important that Lisp provides special syntax for it. A Lisp expression preceded by a single quote mark `'` is expanded into an application of the `quote` function. Thus `'(a b c)` means exactly the sane as `(quote (a b c))`.

quotient *function 2 args*

Form the quotient of two numeric arguments.

rassoc *function 2 args*

`rassoc` is just like `assoc` except that it looks for a match against the second component of one of the pairs in a list rather than the first. So it is a sort of reversed-`assoc`. Thus `(rassoc 2 '((a . 1) (b . 2) (c . 3)))` returns `(b . 2)`.

rdf *function 1 arg*

rdf reads and interprets all the Lisp code in the file whose name it is given as an argument.

rds *function 1 arg*

To read data from a file you first open the file (using open) then select it as the current input stream using rds. A call to rds returns the previously selected input stream, and very often you will want to save that so you can restore it later. A reasonably complete (and slightly cautious) example would be

```
(let* ((instream (open "filename" 'input))
       (oldstream (rds instream)))
   (errorset '(process (read)) t t)
   (rds oldstream)
   (close instream))
```

This shows saving the existing input stream and restoring it at the end. It uses errorset to ensure that this happens even if processing the input from the file fails.

read *function 0 args*

This reads a full Lisp object from the current input stream (which is by default from the keyboard, but can be changed using rds). The item can be a symbol, a number, a string or a list. It can also start with a quote mark or backquote. The function read is the one that is normally used when Lisp wants to grab input from the user, so the standard Lisp top level behaviour is as if it were obeying (while t (print (eval (read)))). If read finds the end of an input file it returns !eof!.

readch *function 0 args*

This reads a single character, returning a Lisp symbol that has that character as its name. If readch finds the end of an input file it returns !eof!.

readline *function 0 args*

This reads a whole line and returns a symbol made up from the characters found. If readline finds the end of an input file it returns !eof!.

remainder *function 2 args*

This divides a pair of integers and returns the remainder.

remhash *function 2 args*

(remhash 'key table) removes a hash table entry previously inserted by puthash.

remprop *function 2 args*
`(remprop 'symbol 'indicator)` removes a property previously set up using `put`.

return *function 1 arg*
This is used with `prog`.

reverse *function 1 arg*
The ordinary function for reversing a list.

reversip *function 1 arg*
A function that reverses a list in a way that re-uses the existing `cons`-cells so as to avoid any need to allocate fresh ones. This necessarily destroys the input list by overwriting parts of it (using `rplacd`) so should only be used when you are certain that nobody else needs that list.

rightshift *function 2 args*
See `logand` and friends. This shifts the bits in a number right, and at least for positive values `(rightshift a` n`)` has the same effect as dividing `a` by 2^n.

rpar *predefined variable*
A predefined symbol whose value is the symbol whose name is a right parenthesis. See also `lpar`.

rplaca *function 2 args*
If you have a `cons`-cell you can use `rplaca` to replace the `car` field, `rplacd` to replace the `cdr` field or `rplacw` to replace both. In each case the result of the function is the `cons` cell that has been updated. Use of these functions can corrupt existing structures and create cyclic ones that lead to all sorts of trouble, and so they should only be used when there is a compelling reason to need a side-effect.

rplacd *function 2 args*
See `rplaca`.

rplacw *function 2 args*
See `rplaca`.

sassoc *function 3 args*
Here we have another variant on `assoc`. In fact `sassoc` is like `assoc` except that it has an extra argument, and if the key it is looking for is not found in the association list rather then returning a simple value `nil` it returns the result from calling that final argument as a function with no

arguments. While this has a long history of being part of Lisp I suspect that very few people use it these days.

set *function 2 args*

This behaves like `setq` except that rather than being a special form it takes exactly two arguments, and treats the first as the name of a variable and the second as a value to store into that variable. It is not very common to need to make the name of a variable that you are assigning to a computed value in this manner.

setq *special form*

When you have a Lisp variable you can change its value using `setq`. In fact `setq` allows you to make several updates, one after another, in one call. Its arguments alternate between being variable names and expressions to compute values to set them to. So for instance `(setq a 1 b 2)` sets `a` to `1` and `b` to `2`. See `psetq` for a variant that does all the assignments in parallel.

sin *function 1 arg*

This is the usual trigonometric function, accepting its argument in radians. See `cos` and `sqrt`.

spaces *function 1 arg*

`(spaces` n`)` prints n blanks.

sqrt *function 1 arg*

Computes the square root of a number, returning the result as a floating point value whether the input was floating or an integer.

stop *function 1 arg*

To quit Lisp you can call `stop` giving it an argument that is used as a return code from the system. This quits Lisp instantly and unconditionally and so should be used with some consideration.

stringp *function 1 arg*

Tests if its argument is a string. See `atom`, `numberp` and `symbolp`.

sub1 *function 1 arg*

Subtracts one from its argument.

subr *symbol*

See `expr`.

subst *function 3 args*

If you have a list or set of nested lists then `(subst a b c)` replaces every item in `c` that is equal to `b` with an `a`. In other words "substitute a for b in c". This only scans its input down to the level of atoms, so for instance vectors and hash tables do not have their components or contents looked at.

symbolp *function 1 arg*

Test if an object is a symbol.

t *predefined variable*

In Lisp the symbol `nil` is used for *false*. If there is no better non-`nil` value to be used for *true* then `t` is used, and the symbol `t` starts off as a variable that has itself as its value.

tab *predefined variable*

A symbol whose initial value is a tab character. See `blank`.

terpri *function 0 args*

This TERminates the PRInt Line. It is equivalent to `(princ !`*eol*`!)`.

time *function 0 args*

If you wish to measure the amount of CPU time that some calculation takes then you can use `(time)` to read a timer both before and after. The difference between the two values will be the processor time used, measured in milliseconds.

times *special form*

Multiplies all of its arguments together.

trace *function 1 arg*

If you go `(trace '(f1 f2 ...))` then each of the functions `f1`,... is marked for tracing. When this has happened `vsl` prints messages each time the function concerned is called and each time it returns a result. This can be a great help when your code is misbehaving: you `trace` a suitable set of key functions and try some test examples. The extra output may be bulky but with luck will allow you to understand exactly what is happening. When finished you can call `(untrace '(f1 f2 ...))` to restore things to their normal state.

upbv *function 1 arg*

Given a vector, `upbv` returns the highest legal subscript that can be used with it. See `mkvect`, `putv` and `getv`.

untrace *function 1 arg*

See `trace`.

vectorp *function 1 arg*

Tests if an object is a vector. See `mkvect`.

vsl *symbol in lispsystem!**

A predefined variable `lispsystem!*` has items in it that can give information about the Lisp system that is in use. Here the only information provided is the symbol `vsl` which (obviously) identifies the Lisp version.

while *macro*

`while` is a macro that is provided with a predicate and then a sequence of expressions to be evaluated repeatedly for so long as the predicate yields something non-`nil`.

```
(let ((n 1))
   (while (lessp n 1000000)
      (printc (list n "is too small"))
      (setq n (times 3 n)))
   n)
```

would return the first power of three that is at least a million, printing reports on its progress.

wrs *function 1 arg*

If you need to direct output to somewhere other than the terminal (for instance to a file or pipe) then you can use `wrs` to select the relevant stream as the one to print to. `wrs` returns the previously selected stream, and often you will wish to save that so you can restore it later. See `open`, `rds` and `close`.

zerop *function 1 arg*

Tests to see if its argument is 0 or 0.0. See `onep`.

Appendix C

Glossary of technical terms

Bibliography

[1] Harold Abelson and Andrea diSessa. *Turtle Geometry: The Computer as a Medium for Exploring Mathematics*. MIT Press, 1986.

[2] Alfred V Aho, Monica A Lam, Ravi Sethi, and Jeffrey D Ullman. *Compilers: Principles, Techniques and Tools*. Pearson Education Inc, 2006.

[3] Gwen Allen, Joan Denslow, and Derek Whiteley. *Seashore Animals*. Oxford University Press, 1972.

[4] Andrew W Appell and David R Hanson. Copying garbage collection in thepresence of ambiguous references. Technical Report CS-TR-162-88, Princeton University, 1988.

[5] Conrad Barski. *Land of Lisp: Learn to Program in Lisp, One Game at a Time!* No Starch Press, 2010.

[6] Joel F Bartlett. Compacting garbage collection with ambiguous roots. Technical Report 88/2, DEC Western Research Laboratory, Palo Alto, CA, 1988.

[7] Edmund C Berkeley and Daniel G Bobrow (editors). *The Programming Language LISP: its Operation and Applications*. MIT Press, 1964.

[8] Aaron Williamson Bradley Kuhn and Karen Sandler. A practical guide to gpl compliance, 2008.

[9] Rodney A. Brooks and Richard P. Gabriel. Critique of Common Lisp. In ACM, editor, *Conference record of the 1984 ACM Symposium on LISP and Functional Programming: papers presented at the symposium, Austin, Texas, August 6–8, 1984*, pages 1–8, New York, NY 10036, USA, 1984. ACM Press.

[10] A.R. Clapham, T.G. Tutin, and E.F. Warburg. *Flora of the British Isles*. Cambridge University Press, 1952.

[11] Leiserson Cormen and Rivest. *An Introduction to Algorithms*. MIT and McGraw-Hill, 1990.

[12] The Qt development team. The Qt cross-platform application and UI framework. "http://qt-project.org", 1991-.

[13] John McCarthy et al. *The Lisp 1.5 Programmer's manual*. MIT Press, 1965.

[14] Julian Smart et. al. The wxWidgets cross-platform GUI library. "http://www.wxwidgets.org", 1992-.

[15] Robert Ross Fenichel. An on-line system for algebraic manipulation, 1966.

[16] John Fitch and Arthur Norman. Implementing Lisp in a high level language. *Software, Practice and Experience*, 7(6):713–725, 1977.

[17] Richard P Gabriel. *Performance and Evaluation of Lisp Systems*. MIT Press, 1985.

[18] Michael J. C. Gordon. Evaluation and denotation of pure lisp programs, 1973.

[19] Eiichi Goto. Monocopy and associative algorithms in an extended lisp. Technical Report TR 74-03, University of Tokyo, 1974.

[20] Eiichi Goto, Tetsuo Ida, Kei Hiraki, Masayuki Suzuki, and Nobuyki Inada. Flats, a machine for numerical, symbolic and associative computing. In *ISCA 6th Symposium on Computer Architecture*, pages 102–110. ACM, 1979.

[21] Philip Greenspun. Greenspun's tenth rule of programming, 1993.

[22] Martin L Griss, Eric Benson, and Gerald Maguire. PSL: A portable lisp system. In *ACM Symposium on Lisp and Functional Programming*, 1982.

[23] Bruno Haible and Michael Stoll. clisp. "http://www.clisp.org", 1998-.

[24] Anthony C. Hearn. REDUCE: A user-oriented interactive system for algebraic simplification. In M. Klerer and J. Reinfelds, editors, *Interactive Systems for Experimental Applied Mathematics*, pages 79–90, New York, 1968. Academic Press.

[25] Anthony C Hearn. Standard lisp. *ACM Sigplan Notices*, 1969.

[26] The IMSAI 8080 microprocessor system. "http://en.wikipedia.org/wiki/IMSAI_8080", 1975.

[27] following Xanalys Corporation LispWorks Ltd. Lispworks. "`http://www.lispworks.com`", 2005-.

[28] Jed Marti, Anthony C Hearn, and Martin L Griss. Standard lisp report. *ACM Sigplan Notices*, 1979.

[29] David B. McDonald. *CMU Common Lisp user's manual: Mash/IBM RT PC edition*. Carnegie Mellon University, Computer Science Dept., Pittsburgh, PA, USA, 1987.

[30] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML*. MIT Press, 1997.

[31] Arthur Norman. Compact delivery support for REDUCE. *Journal of Symbolic Computation*, 19:133–143, 1995.

[32] Arthur Norman. Further evaluation of Java for symbolic computation. In *ACM symposium on Symbolic and Algebraic Computation*, pages 258–265. ACM, 2000.

[33] Julian Padget. How to learn (EU)Lisp. "`http://www.bath.ac.uk/~masjap/TYL`", 1995-.

[34] Alan Perlis. Epigrams on programming. *ACM SIGPLAN Notices*, 17(9), 1982.

[35] Basile Starynkevitch. GNU melt. "`http://www.gcc-melt.org`", 2011.

[36] Guy Steele. *IEEE Standard for the Scheme Programming Language*. IEEE, 1990.

[37] Guy L Steele and Richard P Gabriel. The evolution of lisp. *ACM Conference on the History of Programming Languages II*, 1993.

[38] Guy L. Steele, Jr. *COMMON LISP: the language*. Digital Press, 12 Crosby Drive, Bedford, MA 01730, USA, second edition, 1990. With contributions by Scott E. Fahlman and others, and with contributions to the second edition by Daniel G. Bobrow and others.

[39] X3J13. *ANSI INCITS 226-1994 (R2004)*. American National Standards Institute, International Standards Organisation, 1994, 2004.

[40] Taiichi Yuasa, Masami Hagiya, William Schelter, and Camm Maguire. Gnu common lisp. "`http://savannah.gnu.org/projects/gcl`", 1984-.

Index