Arthur C. Norman
Raffaele Vitolo

Inside Reduce

Inside Reduce

Preface

A respectable Preface should contain a "story" to tell people as to why they should read this and what it will get them ready to do. OK, so there is this thing called `Reduce` and you want to extend it a bit or get a bit further inside. It is too big and messy to understand all in one gulp, and too mysterious for you to do all the complicated stuff you dream of all in week one, but here is a way to get your toes slightly wet in the sea of confusion, and a way to let you observe that even you can do some stuff with it. We thought that embedding a number of *tiny* examples would help.

`Reduce` is well documented. The user's guide [2] contains all the basic material, while the books [3, 1] are much deeper introductions to the algebraic mode and the symbolic mode of `Reduce`, respectively. We hope that people will read this alongside the "`Reduce` Symbolic Mode Primer" by Melenk [5] which provides overlapping coverage and a different voice to explain aspects of system internals, and believe that there is more than enough to be explained to make it reasonable to have (at least) two places where the information is collected.

There are two more texts that it seems proper to note. "Reduce: Software for Algebraic Computation" [6] by Gerhard Rayna, provides broad coverage, but related to the state of `Reduce` in the run-up to 1987 and so in various details it may now not be fully up to date. Jed Marti's book "RLISP '88: An Evolutionary Approach to Program Design and Reuse" [4] describes the low-level programming language used by `Reduce`, ie `rlisp`. It can be useful for understanding both the syntax used at that level, how it related to the Lisp abstractions that underpin it and how it could be used. However if focusses on a dialect (`rlisp88`) which the bulk of `Reduce` ended up not adopting. Support for `rlisp88` is still available within `Reduce` in that the relevant parser can be loaded as one of the `Reduce` optional packages, but the main body of the code does not make use of it.

`Reduce` is free software. This implies that everybody can learn its deep structure from the source code. Unfortunately, while this kind of activity is certainly possible for every user who would like to develop a greater understanding of `Reduce`, it is not satisfactorily supported by any of the above textbooks.

Here is the point where "Inside `Reduce`" becomes useful: it is a textbook devoted to all aspects of `Reduce` which lie into its source code and are currently undocumented, forgotten, or simply unknown.

The text was born as a sequence of conversations in which one of us (RV) wanted to understand many undocumented aspects of `Reduce` and the other of us (ACN) was so kind to explain them to him.

Contents

Chapter 1

Introduction: the layer structure of Reduce

Like any well-written large software package, `Reduce` can be thought of as having a number of layers. Initially and often generally end-users will only interact with the very top one of these. This document is to explain something of the lower ones, because since `Reduce` is an Open Source project everybody is able to interact with all of them. Some may investigate low levels purely for interest, others because they find it allows them to improve performance of their code, solve problems that they otherwise could not or fix errors in the existing code.

Before discussing the global design of `Reduce` it is perhaps useful to introduce one particular topic: the distinction between "algebraic" and "symbolic" mode utterances in `Reduce`.

A normal user of `Reduce` is presented with a programming language in which they will express their calculation, perhaps defining a few procedures but building their work on primitives such as `+`, `-`, and `Reduce` built-in algebraic operators such as `int` and `df`. The values that they work with will be algebraic formulae, and the results they get will also be algebraic. It may of course happen that sometimes an algebraic form is merely a number, but at any time that number might be combined with an indeterminate such as `x` to create a polynomial or be used as an argument to an unspecified operator `f` to build up something more complicated. When names are used in this mode then if they have been given a value then their value is used – i.e. they are treated as variables. If they have not been given a value they are left present in the result as indeterminates. You could try out the

simple sequence

```
r := x + a;    % Value is the formula a + x
a := 3;        % Now give a the value 3
r^3;           % r is now x^3 + 9*x^2 + 27*x + 27
```

Inside `Reduce` these algebraic expressions have to be represented as more or less elaborate data-structures. For instance it happens that the value of r in the above turns out to be a structures that can be written as `(!*sq ((((a . 1) . 1) ((x . 1) . 1)) . 1) nil)`! This is a standardised form used inside `Reduce` as part of the process of simplification. The bulk of `Reduce` is a big pile of code that manipulates these internal representations so as to decompose, combine and reformat them as efficiently as the code authors could manage things. If you wish to look inside `Reduce` to inspect this code (and perhaps in due course to correct or enhance it, or to add your own contributions) you will naturally ask "What language is used for all the code that works with the internal form of data?". The answer is that it a language with just about the same syntax as the `Reduce` user-mode interface, but use in what is known as "symbolic" rather than "algebraic" mode. An effect is that the values you work with are the raw data-structures that are used within `Reduce`. The operators you have available are all the procedures defined in the `Reduce` source code (plus a further range inherited from an underlying Lisp system). Data can consist of a combination of numbers, symbols, strings and lists. If you write a simple symbol such as "x" then in symbolic mode that will be expected to have been given a value earlier in your code. A notation using a quote mark can be used to insert raw data into your code, so here is the symbolic mode sequence you would need to use to achieve the trivial effect of the algebraic mode example above:

```
setk('r, aeval list('plus, 'x, 'a));
setk('a, aeval 3);
aeval list('expt, 'r, 3);
```

This at first looks horrendous and discouraging, but in reality it is all tolerably straightforward once you grasp the nature of the various internal representations that `Reduce` uses. Part of the intent of this document is to help you get to grips both with symbolic mode programming in `Reduce`

and the ways in which procedures implemented at that low (but powerful) level can be provided with interfaces so that they can then be used easily by people who do not want to get inside the system and who will hence only ever use algebraic mode.

The view of `Reduce` presented here thinks in terms of the following layers of abstraction:

1. **The hardware of the computer you are running on.** At the moment `Reduce` can be compiled for 32 as well as 64 bit machines. On the other hand there can be concerns at this level of (the potential for) parallelism, the amount of memory and the speed of various low level operations. For instance at the time of writing this the existing implementations of `Reduce` do not have good ways of gaining benefit from multi-core processors, but essentially all hardware these days provides that capability. `Reduce` is of course far from being unique in this respect, but this very low level issue may eventually generate need for re-work at all the higher levels.

2. **The operating system and environment.** At present this broadly comes down to Windows, OS X, Linux and Android. Historically `Reduce` took the view that it would include capabilities that it could support on all the platforms that mattered to it. Doing so naturally led to it not taking advantage of the special unique capabilities that each operating system provided. . . a particular area where having portable code used to be really hard was in implementing graphical user interfaces. The currently stable versions of `Reduce` are able to provide (almost) the same experience on Windows and Linux but Macintosh users are liable to view what they get as at best ugly and archaic. Android is sufficiently different that there is not yet a full version of the system available, but a "technology demonstration" has been built. At present `Reduce` has no clear model for exploiting touch-screens. As with (1) there are major challenges that could call for large amounts of re-design and re-work at a level that is mostly rather remote from the "algebra" aspect of `Reduce`.

3. **A programming language kernel.** Any algebra system needs at its core facilities to interact with the operating system (eg to access files and the screen), to perform arithmetic (including arbitrary precision integer work), to manage memory in a flexible way (the technical

term "garbage collection" is key here) and to support compilation of higher level code into something that can run efficiently. `Reduce` started by using an existing language, Lisp, to provide this level of support, and this has led to confusion when many people have asserted that "`Reduce` is written in Lisp". These days the "Lisp systems" used with `Reduce` are (to a good approximation) tuned and maintained almost exclusively to be kernels for `Reduce` and not to be used by any free-standing Lisp programmers that may remain in the wild. In 1969 Tony Hearn (who created `Reduce`) documented a dialect that he dubbed "Standard Lisp" that `Reduce` was to rely on, and in 1978 this was given a major refresh. The Standard Lisp Report by J. B. Marti, A. C. Hearn, M. L. Griss, and C. Griss has since then been the official documentation for that level of `Reduce`'s internal structure. But since then the rest of the Lisp world moved seriously sideways with the introduction of Common Lisp and the subsequent fading of the language from widespread use. There are two particular implementations of Standard Lisp that support `Reduce`: PSL and CSL. In the years since 1979 these have, between them, informally agreed to support various capabilities beyond the Standard Lisp Report. These extras are not well documented anywhere! An example is that in the current systems the key built-in names are spelt in lower case, while in 1979 the style and tradition involved use of upper case everywhere (see the discussion in Section 3.7). Both PSL and CSL have been optimised for use with `Reduce`, but in the whole of the `Reduce` sources there are only a few hundred lines of Lisp-syntax code visible, and those are just used while bootstrapping `Reduce` into life. So the two Lisp systems exist to provide the capabilities noted at the start of this section and to isolate the bulk of `Reduce` from most operating system issues.

4. `rlisp`. Almost all of `Reduce` is coded in its own private language `rlisp`. This is sometimes also referred to as "symbolic mode `Reduce`". It provides a syntax that is styled after the programming languages in common use in the 1970s where the primitive operations available are the ones inherited from its Lisp underpinnings. There are two distinct issues in becoming fluent in the use of `rlisp`. The first is the syntax of the language. It is in fact (deliberately) almost identical to the normal user-level notation that `Reduce` provides, and since it is

a simple language getting started with it ought to be easy. Here are some examples illustrating some of the more important constructs:

```
% This procedure takes four arguments (which must all
% be numbers) and evaluates them as a quadratic in x.
symbolic procedure quadratic(a, b, c, x);
   a*x^2 + b*x + c;
```

```
% Procedures with only one argument can omit
% paretheses around it.
symbolic procedure moan_if_negative z;
% The "error" function has a first argument that
% is supposed to be an error code,
% and a second that is a message.
if z < 0 then error(0, "value is negative") else sqrt z;
```

```
symbolic procedure reverse_a_list u;
  begin % Start a block. Think "{" from C or Java
    scalar v; % declare a local variable
    while not null u do
    <<
      v := car u . v;
      u := cdr u
    >>; % "<<...>>" is a light-weight block
    return v
  end;
```

The above used functions `not`, `null`, `car` and `cdr` from Lisp. So here is a moment to talk about the data types available in `rlisp`. Note that the `scalar` notation that introduces a local variables, and all the procedure headers that specify arguments do not declare types. `rlisp` does some basic type-checking at run-time, so if you passed non-numeric values to the quadratic function there would be a run-time complaint. The available types are:

integer	(with effectively no limit to magnitude)
floating point	(using IEEE 64-bit machine precision)
strings	(eg "Hello" or "value is negative". Mainly just used for printing not in computation)
symbols	(see below)
dotted pairs	(see below)
other things	(for specialist use! Eg vectors)

A symbol is something that has a name. Procedure and function names and all the variables in `rlisp` code are in fact symbols, so `quadratic`, `b`, `v`, `null` and so on are. Even the keywords like `begin` and `end` and operator and syntax marks like `+`, `*`, `:=`, `<<` and `>>` really name symbols at some level. If you just write `X` in your `rlisp` code that will refer to (the value of) the symbol whose name is `X`. If you want to talk about `X` itself you have to quote it. So

```
v := 'X;
```

sets a variable `v` to have the symbol `X` as its value. If you want to use a symbol whose name includes characters beyond letters and digits you need to escape them using an exclamation mark:

```
v := '!<!<;   % symbol whose name is "<<"
w := '!2;     % symbol called "2" as distinct from
              % the number 2
x := '!X;     % an upper case X, since by default
              % input is folded to lower case.
y := '!"not! a! string!"; % Compare with "is a string"!
```

A (dotted) pair is an ordered pair of items. The function that creates pairs is called `cons`, but `rlisp` allows an infix (or an operator whose two arguments are written to the left and to the right of its symbol) `.` to be used for that.

```
v := 1 . 3;  % Not the floating point value 1.3!
w := 'left . 'right;
x := 'first . ('second . ('third . nil));
```

The final example there suggests that pairs can be used to model lists. The convention is that the atom `nil` (which has itself as its

value when you use it as a variable) terminated a list. This is such a common usage that special syntax and functions support it:

```
v := '(a fixed list of length six);
w := list('a, 'variable, 'list, 'including, v);
x := list('a . 1, 'b . 'value_to_go_with_b, 'w . w);
```

The functions `car` and `cdr` retrieve the two parts from a pair. The names of these functions derive from the early history of Lisp. But they lend themselves to a convenient shorthand where e.g. `car cdr x` can be replaced by `cadr x` and in the context of the current paragraph that might be `(b . value_associated_with_b)`.

Basically all of `Reduce`'s internal workings are built up using nested lists and pairs with symbols and numbers as leaf elements.

Chapter 2

Building Reduce

2.1 Ready-to-use binaries

A `Reduce` distribution, ready to be run, can be downloaded for several architectures (Windows, Mac, GNU/Linux) at the official `Reduce` website, hosted at Sourceforge [7].

Simple users will often be content with whatever pre-built binaries are available there – regardless of exact details of how up to date or how neatly packaged they are. However there are at least four reasons that can make it seem desirable to build your own copy of `Reduce` from source code:

1. Errors may have been corrected in the `Reduce` sources or new features added since the last set of binaries was put in the release area. If the bugs or features relate to a style of use of `Reduce` that does not concern you (for instance the main enhancements have been in one of the specialist packages that implement algebraic transforms that are to do with an area of mathematics remote from the ones you are involved with) this may not matter, but sometimes your particular use will really *need* the latest version;

2. You report a problem to the `Reduce` mailing list and get a response that asks if the behaviour you describe still occurs using the very latest revision;

3. You are minded to make changes to the `Reduce` source code. This could be to improve its clarity or performance, to correct deficiencies

or to add brand new capabilities. The existing `Reduce` developers would like to encourage more people to join in in this manner;

4. You adhere to the philosophy that if you are going to rely on the results calculated by some software package then in principle at least you should be in a position to understand and check its behaviour through and through.

Using `subversion` to fetch and update the source code and building everything for yourself can ensure that you have the very latest set of additions and bug fixes. This chapter is intended to supply the instruction for building `Reduce` from its source code.

2.2 Fetching the source code

The source code of `Reduce` is in the `sourceforge.net` hosting website [7]. All its files are under a revision control system, Subversion [8]. In order to get the latest version of all files you shall have the program `subversion` installed on your computer. The program can be used from a terminal, by command-line instructions, or by a graphical user interface. There are many of them, see the links at [8].

The command to get the files will be given on the sourceforge web-site under the "code" tab. For a read-only copy at the time of preparation of this document the command was[1]

```
svn checkout \
  svn://svn.code.sf.net/p/reduce-algebra/code/trunk \
  reduce-algebra
```

If you obtain the source code in some other way then provided it is up to date and complete the same instructions should apply. In general the biggest challenge when building `Reduce` tends to be in ensuring that all the necessary tools and development libraries are installed: once they are in place the actual building of `Reduce` will usually be straightforward.

It may happen that the latest development version gets broken for some reasons (usually mistakes/misprints in the last changes to the code) and

[1] if you look carefully at the sourceforge site you can find slightly different URLs to use that could be better if you were joining the `Reduce` maintenance team and so would be contributing updates.

it is not possible to successfully build `Reduce`. Then a good idea is to use a previous version (or 'revision') that it is known to work. In order to download the revision number '1234' the following command shall be issued

```
svn checkout \
  svn://svn.code.sf.net/p/reduce-algebra/code/trunk@1234 \
  reduce-algebra
```

if you need to do a full fresh checkout, or

```
  svn -r 1234 update
```

to update an existing set of files to the given revision number; here 'update' can also mean 'go back to a previous revision'.

2.3 GNU/Linux

The recipes given here may also apply to various versions of Unix and systems in the BSD family. But various GNU tools might need installing and in the worst case you will need to do that by fetching their source files and building them first.

At a command prompt select the `Reduce` trunk source directory as current. If you look you should see it contains further directories called `csl`, `psl`, `packages` and `scripts` (as well as various other stuff). If you are going to build the CSL version of `Reduce` you should ensure that you have at least

- the GNU C/C++ compiler `g++`;
- the GNU `make` utility for automating compilation of source code;
- `autoconf` and `automake` for the automatic generation of configuration files and `make` scripts;
- the X development libraries `libx11-dev`, `libXext-dev`, `libXft2-dev`;
- the screen addressing/cursor control library `libncuses5-dev`.

When testing `Reduce` it is also desirable to have the GNU version of a `time` command installed. The names for packages listed here are representative and may vary from one Linux installation to another: the key issue is that header files and other material needed for developing C and C++ code using various libraries will be required. As an attempt to make things easier to set up there is a script that can be run as

```
scripts/csl-sanity-check.sh
```

that checks (most of) what `Reduce` requires. If it succeeds it should compile some code and end up displaying a window on your screen. If you have trouble with the main body of `Reduce` and can not understand the messages you see go back and try this script since it may provide clearer messages from a simpler attempt to build things!

With prerequisites satisfied the building of `Reduce` can be done in a small number of steps.

1. `svn update`

   This command is used if you have fetched `Reduce` using subversion, and it will ensure that the version you have locally is brought up to date with any recent corrections or upgrades from the sourceforge site. If you have altered any files locally then subversion tries to merge updates from the central repository with ones you have made, and often that works well. If not then you need to read the subversion documentation to see how to recover, but if you had altered a file (say `packages/subpackage/altered.red`) you can try the recipe

   a) `cp packages/subpackages/altered.red somewhere-safe.red`

   b) `svn revert packages/subpackage/altered.red`

   c) merge your changes back from `somewhere-safe.red`...

2. `scripts/stamp.sh`

   When `subversion` fetches or updates files it concerns itself just with their contents, not with date-stamps. This can sometimes confuse the `autoconf` tools, and at the very least can lead to some redundant work. To end up slightly saner run the script mentioned here which simply resets timestamps on some build files so that there is (much) less probability that your computer will try to re-run `autoconf` and

`automake`. Actually you only even slightly need to do this after either a first checkout of the code or if fetching an update that changed one of the files called `configure.ac`, `configure`, `Makefile.am` or one of their close relatives.

3. `./configure --with-csl` (and/or `--with-psl`)

   The standard arrangement for building many open source programs involves running a script called `configure`. This detects all sorts of details about the computer you are building on and creates files called `Makefile` in a number of places. The configure line can use `--with-psl` instead to build a PSL-based version of `Reduce`, and there are a number of other options that can go on the end of the line, of which perhaps the most important is `--enable-debug`. This make the C/C++ compiler build with debug options so that it becomes possible for low-level programmers to use `gdb` or another debugger to hunt bad crashes way down at that level.

   Running the configure script with `--help` should list all the options it is aware of, but please do not randomly try obscure combinations of options – some may be little more than traces of earlier experiments.

   Note that merely running the configuration script can take a minute or two (and substantially longer on Windows), and that if you do not have all the necessary development libraries installed it could fail in a way that may initially appear obscure. In addition of the previously mentioned `csl-sanity-check.sh` script please look in the log file(s) that configure creates[2] and look for evidence of missing header files or libraries. Install them and try again!

   In general you only need to run configure once when you first install the `Reduce` sources. On subsequent occasions and even after use of "`svn update`" to refresh things you can omit this step.

4. `make`

   The main part of building `Reduce` occurs here. The very first time you do this it will be necessary to build various sub-libraries that `Reduce` relies on, and that can feel painfully slow – so have patience or several cups of coffee.

[2]principally any file called `config.log` in any part of the tree of build directories

If all goes well you can then launch `Reduce` using either `bin/redcsl` or `bin/redpsl` (depending on whether you build a CSL or PSL version).

5. Recovering if `make` fails.

   The CSL version of `Reduce` builds all its components within a directory called `cslbuild` whereas the PSL version uses the directory `pslbuild`. In each case they put things in a subdirectory of that place names for the current computer's configuration, so for instance you may find a directory `cslbuild/x86_64-pc-windows-debug` or `pslbuild/i686-unkown-ubuntu14.04`. The first and easiest thing to try if things go wrong is to get a fresh start by just deleting that directory and re-running the configure script. If the failure had been because you had not (the first time) had enough build tools or libraries installed but you have now corrected things that may help.

   Otherwise you will need to find the build log files and use them to track just what the problem was so you can either fix it yourself or report it to the central maintainers. In either `cslbuild/xxx` or `pslbuild/xxx` the file config.log should end up as record of what the `configure` script did. For `csl` there are also `config.log` files in the further sub-directories `crlibm`, `fox` (or `wxWidgets`) and `csl`. If the failure happened at configure time the evidence is liable to be hidden in one of those, interleaved with a lot of other material. If configuration worked then `make` will run scripts that leave log files in `pslbuild/xxx/buildlogs` or `cslbuild/xxx/csl/buildlogs`. Identifying the most recent file in those directories can sometimes let you home in on where the failure arose, and with luck there will be a message that gives some clue as to what had gone wrong and hence what might be done to correct it. Until you find the key error message you do not know what to try to do to correct things.

   In general you want to find the first error that arose during the build attempt – once one thing has gone wrong it is reasonable to expect a cascade of follow-on confusion. Looking at date-stamps on files can sometimes help you see what order things happened in.

   Pretty much anything that is listed here as a "potential trouble spot, so watch out" will (we hope!) be addressed so that soon it is not. It may nevertheless be useful to comment a little on the build process in case that helps you navigate and track down an issue. With `psl`

the build progresses by fetching a read-build binary of the `psl` Lisp system and some of its modules, and it uses that to create a provisional version of a subset of `Reduce` in `red/bootstrap.img`. The log file from this will be in `buildlogs/bootstrap.blg`. This initial version of `Reduce` is then used to compile further `Reduce` packages and eventually `red/reduce.img` is created.

For `csl` everything is built all the way from source. This obviously takes longer, but you may take a view that it puts you in complete control of everything. Firstly the `crlibm` and `fox` libraries have to be build and their binaries installed in the correct place within the `Reduce` build tree. If you are uncertain about their state you should find test or sample for both – with luck when you look for it the location and usage will be obvious. When that has been done building continues in the `cslbuild/xxx/csl` directory. On the first time you build or if `Reduce` source files have changed a version called `bootstrapreduce` is built – see `buildlogs/bootstrapreduce.log`. That is used to translate some of the most critical parts of `Reduce` into C code (`buildlogs/c-code.log`). This C code is kept in `cslbuild/generated-c`. Then the main `Reduce` executable can be compiled (using the generated C) and it finally creates `reduce.img` leaving `buildlogs/reduce.log` as a record. If there is some chance that the C code has got out of step then `make full-c-code` will force it up to date. Sometimes when files are removed from the main `Reduce` sources or renamed the `make` system can retain dependency information and fail to work. If you get a complaint saying that a file is needed and on inspection it is one that used to exist in the sources but a recent update renamed or deleted it then check the files `*.dep` and delete the line that mentions the offending file. Of course if you merely deleted the whole `cslbuild` directory and re-ran the `configure`/`make` sequence that would also achieve the desired effect – but it could be a lot slower.

For the Windows version of `Reduce` there are additional complications in that there is an attempt to build up to six versions of the `Reduce` executable. This dreadful situation is to cover both 32- and 64-bit Windows, to cope with people who wish to launch `Reduce` by double clicking on an icon and those who may run of from either a native Windows console (i.e. command prompt) or a Cygwin terminal. If

something goes wrong (for instance one of the builds is interrupted before it completed) it can sometimes help to enter each of the separate build directories in turn and go `make` in each. If trying that it will be best to build the Cygwin version before the native Windows one.

As with all debugging, if things do not work first time you need to look to find out where the evidence is and apply calm reasoning to narrow down the cause! At least you have access to all the source code and all the build scripts so that you (or an expert friend) is in a position to investigate in detail!

It is perhaps proper to be aware that a version of the `Reduce` sources fetched from the subversion repository will be the latest and most up to date set of files you can obtain, and is liable to have corrections to issues that have been raised – however sometimes it may be in a transitional state when a change has been made that is intended to fix problems but that in fact introduces others. If you encounter difficulties you may wish to browse the repository records on sourceforge to see which files have changed recently and consider whether the problem you observe might relate to that. It has also been the case in the past that if you update or replace your operating system that can occasionally introduce incompatibilities. This is particularly true of you install a very new version of an operating system that other developers may not yet have had time to investigate. Please report problems so that they can be fixed not just for you but for everybody.

6. `scripts/testall.sh`

   Having built `Reduce` it is prudent to check it. The `testall` script tries to run all the standard tests and compares the output created on your machine with reference logs deposited at Sourceforge. There can sometimes be minor timing-related differences in output that show up as discrepancies and from time to time the Sourceforge reference logs may not quite keep up with the main source files kept there, but if this script runs it can give you good confidence that things are basically in a good state, and can help direct you towards any areas that need investigation. The argument `--with-csl` or `--with-psl` instructs this script to check just one version of `Reduce`, and thus if you have only configured and built one version you should specify

it. However if you are writing code that you hope to be useful for everybody you should test it on both the CSL and PSL versions of `Reduce`, and when you run the `testall` script you omit any arguments and it checks both versions.

2.4 OS X (Macintosh)

On a Macintosh the recipe is basically as for Linux. However you should be aware that the current version of `Reduce` uses `X11` when it attempts a windowed interface, and some of the development libraries it requires may not be supported by Apple. The first thing you will need will be the command-line tools for `Xcode`. You can either then fetch and install the `macports` package and use that to help you install the other components that `Reduce` relies on – or you can do all that manually yourself. The `trunk/csl/cslbase` directory contains a file `macports.my.list...` that lists the collection of ports that one of us installed on a Macintosh at one stage to get to a state where `Reduce` could be built. Since OS X and `Xcode` will change from time to time there is no guarantee that this list will be absolutely definitive, but it should provide a good starting point.

The configure script provides an *experimental* option that is not useful for end-users but that I hope some Macintosh enthusiast will feel motivated to help work on. This is `--with-wx`. It is intended that eventually that will build a version of `Reduce` using the `wxWidgets` toolkit rather than the `FOX` one, and `wxWidgets` provides native Macintosh support. So far the code there is seriously incomplete and in need of helpers to move it forward so that eventually a proper Macintosh (possibly including iOS) port of `Reduce` can exist. Volunteers please step forward!

2.5 Windows

For Windows the build system that is supported uses the `cygwin` package, freely downloadable from `www.cygwin.com`, to provide a shell environment similar to the Linux/BSD/Unix one where `Reduce` can be build using `configure` and `make`. The commands in `scripts/cygwin-sanity-check.sh` are an attempt to verify that sufficient of `cygwin`'s optional packages have been installed that you have a good

chance of success. As with the list of `macports` in the Macintosh case there will always be a possibility that this list falls out of date, but despite that it can greatly reduce initial pain by getting you at the very least close to a state where everything should work.

The build sequences on Windows are especially convoluted because there are a number of ways in which Windows may be used. The file `csl/cslbase/gui-or-not.txt` explains some of this, but the existence of both 32 and 64-bit versions of Windows itself and of cygwin (eg one can have either a 32-bit or a 64-bit cygwin shell on a 64-bit Windows) makes life messy. As a result of this the build sequences for `Reduce` are set up so that they (automatically) configure and build several slightly different versions of `Reduce`, and when you try to launch `Reduce` it should probe its context and decide on which one is best to use in the circumstances. Thus you may end up using not just `gcc`, but also `i686-w64-mingw32-gcc` and `x86_64-w64-mings32-gcc`. Recent versions also like to have an installation of the 64-bit cygwin variant present, but note that most testing is done using the 32-bit cygwin so that is what you shall try first.

Also for technical reasons the configure scripts tend to run a lot slower under Cygwin than they do on Linux. This relates to issues in simulating a Linux "fork" operation on Windows. So especially for a first build you will need plenty of patience!

There is no special reason why the code for the CSL kernel should not compile using Microsoft Visual C or the Intel C compiler (or indeed any other up to date compiler) however at present the `Reduce` project does not provide configuration files for building in any context other than the one using (GNU) `make`. If you construct some project build files in a form where they would be likely to be of use to others then please consider contributing them back to the project – but when you do so think hard about how they may need to cope with varying computer configurations, with files stored in different locations on disc and with the issues that arise when compilers and the like are updated.

2.6 On portability of Reduce source code

The list of architectures where `Reduce` has been successfully built would be quite a long one. Since `Reduce` is quite portable it is worth just to try to build `Reduce` on a particular architecture and, in case of problems (eg if

there was a rather recent operating system upgrade), refer to the developers' mailing list to try to get them sorted out.

The PSL version keeps its sets of ports in `trunk/psl`, and `trunk/psl/dist/kernel` is liable to be the best place to watch. Sometimes a new port will arrive there in an unfinished or "alpha" state. Sometimes an old port will remain there with little testing and maintenance. There will be other (older) ports that could be revived if there was sufficient demand.

For CSL the system is intended to run on any computer that will support the GNU build tools (gnu make, automake, autoconf) and a reliable C/C++ compiler. It is generally tested using `gcc` and `clang`. While there should be no fundamental reasons it could not be built using Microsoft, Intel or some other compiler the integration of those into its build scripts has not been investigated. For use with a GUI it will use the Windows normal arrangement on a Microsoft platform or X11 elsewhere (including at present on a Macintosh). Over the years it has been built and run at least once on a rather large range of Unix-like platforms including ones from HP, SGI, Sun and a number of smaller vendors. Also in the past customised versions (which were created as technology demonstrations not for serious use) ran on a Linksys router and on a HP iPaq organiser! Successful tests were also run using the Hercules emulator for the IBM z/Architecture. But anybody wishing to use any of these or some new specialist platform might expect to need to be ready to make minor adjustments to the code.

At the time of writing the main testing is done on (64-bit) Linux, Windows (8.1), a Macintosh and slightly less frequently on a Raspberry pi. It is expected that BSD-family systems will not cause trouble. The Windows version can be built for either 32 or 64-bit use and to run either natively and directly under Windows or to be launched via a Cygwin (32 or 64-bit) shell. The Linux versions should build equally well on 32 or 64-bit platforms. Image files created on any of these platforms should reload on any other - regardless of discrepancies in word-length or byte order.

One of the strengths of `Reduce` is that if at any time one of the CSL or PSL variants give trouble on a particular platform it may be possible to try the other!

2.7 Less supported variants

The main versions of `Reduce` build the while of the algebra system using either the CSL or PSL Lisp system. With CSL the code uses a graphics library called FOX to support some sort of GUI. There are a range of other configurations that are possible. Some represent experimental work by the developers and are not ready for prime time – bu in some such cases it would be really good to have new volunteers to help complete the work. Others are special versions that were either created to make a technical point or to support some particular user or project, and these may not always be up to date an workable. There are brief notes about some of these in the commentary on the `Reduce` source tree, but a brief overview here may also help.

The three main categories of "specialist versions" may be illustrated as follows:

new-embedded At times some users have wanted to include the whole of `Reduce` as a component within some larger software package. This could be for teaching, as a larger algebraic/numeric package or to do with optimisation. Each case may have different needs, but nevertheless the common feature will be that the large project needs to pass algebraic calculations down to `Reduce` and get back answers. `Reduce` will not be expected to supply any user interface. The `new-embedded` directory provides a prototype for support for this and is a successor to `embedded`, an earlier attempt. It uses a dramatically simplified `Reduce` build process (ie you just compile a set of C files together) and there are entrypoints into the C library that is thereby created that allow the caller to pass things into the Lisp/`Reduce` world and to retrieve results. The limitations and oddities of interfacing may not all be immediately obvious, but there has been success in the past. Anybody wishing to use these needs to be willing to delve deep into C code and should discuss their needs and capabilities with some of the `Reduce` developers. The expectation is that because their own code is some package already bigger than `Reduce` that they are pretty competent and experienced!

jlisp and jslisp The worlds of Android and the web might also like to embed `Reduce`. Doing this could sometimes be done via a native code option that could use CSL or PSL almost directly, but there is code in the full

source tree that implements the Lisp that `Reduce` needs as Java code. This has been used to provide an initial Android application that uses `Reduce` to implement a form of algebraic calculator on Android. That is certainly sufficient to prove that if somebody were to design a more general touch-screen interface for `Reduce` that could also be delivered on Android, or indeed any other platform supporting the Java language. The Java version tends to be significantly slower than either CSL or PSL, and may not always be kept as up to date, but it could be a good basis for experimentation. The Javascript port was created by passing the Java version through an automatic translator and obviously opens up the possibility of all sorts of Web use of the `Reduce` algebra engine.

vsl CSL and PSL and reasonably complete reasonably high performance Lisp systems where their main focus has been the support of `Reduce`. `vsl` is a "baby" Lisp that has just enough capability to support (most of) `Reduce`. But its code is concise enough that a reasonable beginner could potentially understand all of it – reading and working with its code could thus provide both a stepping stone towards understanding the fuller version and a sandbox in which new ideas about Lisp implementation in a `Reduce` context could be explored at fairly low cost. Part of the original motivation behind its development was to start a re-design on CSL from the ground up based on all the experience gained from the current implementation, so there is some hope that `vsl` experiments will lead to an eventual `cslplus` new Lisp that can eventually replace CSL and that will be cleaner and more modern internally. As ever, volunteers willing to put real effort into work towards the future would be very welcome!

As can be seen none of these are for trivial routine use by people who just want to use `Reduce` to do some algebra for them, but they may be of real value to system-builders with specialist needs.

Chapter 3

A guided tour of the Reduce source code

3.1 The structure of Reduce source code

In this section we list most of the directories and files in the main directory of `Reduce` source code, with a brief explanation. We will not document every single file and every single sub-directory since even an advanced and enthusiastic developer will never need to look at everything. When the purpose of something seems sufficiently self-evident we will not comment further, and note that some directories contain files with names such as `README` that try to explain their purpose!

One of the issues that arose while drafting this chapter was how to respond to files that seemed illogically arranged or out of date. There was a substantial temptation to try to tidy things up. It can be hoped that over the coming months and years rationalisation and rearrangement will take place, but by and large what is written here will relate to the structure of the `Reduce` sources in 2014 when the subversion revision number was between 2500 and 3000. It seems very probable that the major directories will remain unchanged and even if what is documented here is aimed at something of a moving target it will perhaps be better than not having any explanation at all.

When you build your own copy of `Reduce` that will create new files and directories. Most obviously when you run the `configure`/`make` steps that will create a directory `cslbuild` (or `pslbuild` accordingly) and within that

there will be both a directory relating to the architecture of your current computer and various other files needed by the build process. When you run the `configure` script it is liable to create a `config.log` file that records what it has done (albeit the log can seem cryptic on first inspection) and it uses a directory called `autom4te.cache` to keep track of configuration information that it discovers.

That leaves some major directories that are present when a fresh system is first fetched from Sourceforge.

The directories

`bin`

The files in this directory are intended to provide an easy way to launch `Reduce`. From the `Reduce` "trunk" you can just issue a command such as `bin/redpsl` or `bin/redcsl` to use the PSL or CSL version of `Reduce` (supposing that you have configured and built it!).

There are some features of the scripts used here that may not at first seem obvious.

The first thing is that each script here can be invoked from anywhere - the directory that is current when you trigger one of these scripts does not matter. So, for instance, you can add the path to this directory to your operating system's paths to binary program files and then use `Reduce` freely. This remark may seem obvious, but the important aspect of it is that these scripts identify directories where various `Reduce` resources are to be found, and contain curious-looking code to do this. Thus you should not copy any of these scripts and place them in a directory other than here since they rely on paths relative to the place where they themselves live.

The second matter is that some people in some contexts have a single shared file-space that they access from a variety of different models of computer. The binaries that relate to different operating systems and computer architectures have to be kept separate, and the main configure and build scripts for `Reduce` achieve this by building binaries in sub-directories with name such as `pslbuild/i686-pc-windows` or `cslbuild/x84_64-unknown-suse11.1`. The scripts here automatically detect the nature of the machine that they are being run on

and on that basis link through and launch the relevant version of the code.

The main programs that are provided include:

`redcsl`

`Reduce` using the CSL Lisp system. Note that just calling `redcsl` will probably cause a window to pop up, but `redcsl --nogui`[1] causes the code to run as a console application. A range of other options are available, and a list of them with brief explanations can be displayed by starting `Reduce` with the option `--help`. Notably `-L logfile.log` will send a transcript of the output from `Reduce` to the named file for later checking.

`redpsl`

`Reduce` using the PSL Lisp system. Again there are some command-line options available – see the PSL-specific documentation in the `psl` directory for explanation.

`redpslw`

On Microsoft Windows this may run the PSL `Reduce` in a window.

The following scripts may be of use of CSL developers but are not intended to be of general use to people who are not suffering problems or debugging new code:

`bootstrapreduce`

A slower CSL version that is used while building the full version, and where the function `lisp mapstore()` may be used to collect profiling information.

`csl`

The CSL Lisp system.

`fontdemo`

tests or demonstrates the Maths fonts used here. This and the other "demo" programs provided with the CSL system have on

[1]The option `-w` can be used as a shorter alternative to `--nogui`

occasions been useful when porting the full system to a new machine, in that they represent much smaller programs that illustrate or allow testing of individual aspects of the full system and so can be built and checked first, before attempting to identify and work around issues that might make the full system harder to get going.

`fwindemo`

tests or demonstrates the user-interface aspect of CSL `Reduce`.

`showmathdemo`

tests or demonstrates maths display in the CSL version.

At present for the CSL versions there is one further complication. If an exact match for your operating system can not be found in the set of installed binaries then the first available "soft match" will be tried, with a message displayed to explain. There is no clever arrangement to cause near matches to be tested for in a special order. The idea behind this is so that eg if you had binaries for say Fedora 9 installed and were actually running a different version of Fedora (say 10) then the code will TRY the Fedora 9 binary.

A few more scripts in the `bin` folder are documented in the README file in the folder itself.

`buglist`

This directory was used in earlier versions of `Reduce` to contain descriptions of the known bugs and features in the then-current development system. Some of the items listed there may have by now been sorted out, while others may represent challenges in that sometimes issues raised by a user may not have any solution that is at all obvious or that does not seriously impact performance. Inspect the files there to get a feeling for the sort of problems that `Reduce` has had, but expect current or new bugs to be reported and discussed on the Sourceforge `Reduce` developers' mailing list. It would be useful if at some stage somebody took the time to review all the reports in this directory and sort out which have been fixed and which remain as deficiencies. At present the list is unmaintained.

`contrib`

Code that has been provided by somebody but does not yet represent stable code integrated into the main tree. See `contrib/README`. The project will accept code and put it in here more easily than into the main packages directory. Code present in this directory is not automatically built as part of `Reduce`, and should be viewed mostly as experimental or as starting points towards future supported packages.

`csl` and `cslbuild`

See Sections 2.3 and 3.4 for an explanation of the contents of these directories. `csl` and its various sub-directories contain all source file relating to the CSL Lisp, while `cslbuild` is where copies of `Reduce` built in that version get created.

`debianbuild`, `MacPorts` and `winbuild`

Only a very few people will need to package up `Reduce` for distribution, and the scripts for doing this may not always be in a stable state.

The `debianbuild` directory contains everything necessary to build Debian/Ubuntu packages for `Reduce`. To use it, you should have already installed the package `devscripts` as well as a fairly full set of TEX and development tools. You will need yet further things installed if you wish to create `.rpm` as well as `.deb` package files.

Instructions for running the scripts should be present in the directory, but since these are not intended for the casual user they should not be viewed as guarantees of effortless success.

`MacPorts` contains a `Portfile` intended to make it easy for Macintosh users to fetch and build the system using the well-established `macports` tool. This file quotes the particular subversion revision of `Reduce` that it was tested with and that will often be some way behind the most recent checked-in version. Despite the fact that `macports` attempts to keep building safe regardless of the exact details of OS X that is in use it would be prudent to approach all building on Macintoshes with a reasonably open mind after installing a significant upgrade of either the operating system or of Xcode Tools.

Finally `winbuild` contains files and scripts for creating a Windows installer (using `InnoSetup`) and all the cautions and caveats relating to Linux and Macintosh packaging apply once again.

`doc`

`Reduce` keeps the source versions of most of its documentation within the `doc` directory. Almost everything is in LaTeX. If you can not find a ready build `.pdf` file documenting some aspect of `Reduce` you are interested in it is worth checking in here to see what you can find. Historically when `Reduce` was a commercial product the source of the manual lived here but customers were sent a printed and bound version and so in general did not need to look here. As well as being usable to create a `.pdf` format manual the files here are used to create the on-line documentation that is browsable at `http://reduce-algebra.sourceforge.net/manual/manual.html`.

`generic`

Material that is in general not coded in `Reduce` but which may be useful in certain contexts. For instance an Emacs customisation file is there, also the code for `redfront` that provides a terminal interface including local editing and colouring of prompts (in particular it provides richer history and command completion facilities than the line editing built into earlier eaw versions of `Reduce`, so may appeal to console-mode users who are used to the `bash` keyboard shortcuts).

`jlisp`

The main versions of `Reduce` run on top of either the CSL or PSL Lisp systems. Jlisp is yet another Lisp that can support `Reduce`, and is entirely coded in Java. It is liable to be less well maintained than either of the two main Lisps and very distinctly slower, however anybody with a special need for Java compatibility or interfacing may find it useful. The source code here can also count as a reasonably clear and easy to read presentation of how a Lisp system might be implemented. At present an Android version of `Reduce` might wish to exploit this, and enough code for that is provided to show that it would be feasible to provide either a full-featured or a calculator-style algebra package on that platform. In general you will only want to try to use this version of `Reduce` if you are willing to dive in and maintain all of the Java code. That should not be hard for a reasonably experienced programmer but is not for novices.

`jslisp`

Jlisp has been converted from Java into Javascript to produce this. The consequence is further performance worry, but in return for that the ability to embed `Reduce` in a web context. This is not for casual use by ordinary users but for the benefit of system builders constructing large scale web-based applications that need to incorporate `Reduce`. Even more than Jlisp from which it is derived this is for use by serious programmers who have specialist needs.

`libedit`

The GNU readline code that provides convenient local editing, command completion and a history mechanism is maintained by its owners under as fully strict a GNU license as they can as part of a deliberate policy to try to encourage people to license their own (much larger) programs under the GPL. The `Reduce` project has chosen to use a BSD license and explicitly not the GPL, and hence is unable to use readline. Fortunately there is a BSD-licensed work-alike, libedit. To ensure that the source of that is available whenever any relevant part of `Reduce` is used a copy of said source is included in the `Reduce` distribution tree.

`packages`

Apart from the file `package.map` that lists all the packages that comprise `Reduce`, this contains directories that are in general one per loadable package. There are a few cases (eg `misc`) where one directory is used to store a number of smaller packages, and the directory `redlog` is itself subdivided into further folders that contain the constituent parts of that one particularly large `Reduce` sub-project.

`psl` and `pslbuild`

There are to the PSL Lisp system what `csl` and `cslbuild` are to CSL.

`scripts`

The various scripts used to build and test `Reduce` are collected in this directory. The ones that are perhaps most liable to be useful are `testall.sh` (which can run tests on all known `Reduce` packages) and `test1.sh` which tests just one package. People embarking on development using CSL may find `csl-sanity-check.sh` and (on Windows)

`cygwin-sanity-check.sh` useful helpers while making certain they have enough development tools installed. When checking that directory as part of the process of writing this documentation it became clear that a number of the scripts there relate to old ways of doing things, while others are used as part of the configuration process. Of the scripts used when `Reduce` is configured the main one that may need periodic review is `findos.sh` which is intended to generate a string identifying the operating system that is in use. If you install the `Reduce` sources on a new platform you may need to insert a few lines into your copy of that file so that your port can have a good name.

`vsl`

The `vsl` directory contains yet another Lisp that is capable of (just) supporting `Reduce`. `vsl` is hardly a serious Lisp, in that its entire source code is only just over 3000 lines of rather plain C code. Being small it may fail to support quite all of `Reduce` and it will certainly be slow, but it provides a lead-in for anybody who would like to find out a bit about Lisp implementation, and could possibly form a really compact body of C that could be used to support `Reduce` as a component of some other program or running on some otherwise awkward computer. The best characterisation of this is that it could appeal to those how have a good old-fashioned hacker mentality and can appreciate being provided with a sketchy and possibly ill-documented framework that they can cannibalise to form part of whatever their own project is. See also `csl/embedded` and `csl/new-embedded` for other toolkits that an experiences programmer could use as part of the process of including all of `Reduce`'s capabilities within their own product, and note over and over again that these are not supported with any careful hand-holding or explanation: they are for people who can pick them up and make use of them with at best a few hints!

`xmpl`

This is a snapshot of all the test scripts for `Reduce` and the corresponding reference logs. The definitive versions of these are held within the packages directory, but users who do not need the full set of `Reduce` source files may find the copies here useful.

Every so often there will be a maintenance task refreshing this set from the master versions. In case of uncertainty please fetch the packages directory from subversion and inspect the version there.

This directory exists to make it easier to support the creation of binary-only distributions of `Reduce`. By its nature a binary distribution would not contain the `packages` directory (because that mostly contains source code): but `xmpl` can be shipped instead. For some people it may also be convenient to have a complete set of test/example files collected neatly in one place since those tests can act as supplementary documentation of `Reduce`'s capabilities and an illustration of how package originators expected their code to be used.

The files

In the main directory there are various files which are needed for the configuration and compilation of `Reduce`: `configure`, etc.

Some of the other files and directories are historical hang-overs and some contain notes about possible future work. Others may be being used by developers to communicate with one another. Besides these files, a few other files of key importance are listed below.

`ACN-projects.doc`

Almost any truly interesting software project will never be complete, and `Reduce` offers many opportunities for new blood to join in and contribute. The document you are reading now is intended to provide some sort of a route-map to get you started. As well as having your own ideas about what developments based on `Reduce` or extensions to the system it would be good to work on it may help to have a list of some of the ideas that others have either considered or started but not yet finished. The file `ACN-projects.doc` lists some starting points that one of the developers is interested in and where volunteers to help would in general be welcome. One might imagine other collections of project ideas from others being collected, and if enough arrived they would be consolidated into a `reduce-projects` directory. As well as a source of tasks that people wishing to contribute to `Reduce` might work on a list like this also serves as a basis where users who are themselves unable to contribute to the code can get some sense

of what might be possible, and on that basis it could be useful to provide encouragement to the active developers so that they know what sorts of extensions to `Reduce` would be most appreciated by the community.

`Contributor-Release.txt`

At the time that `Reduce` was a commercial product contributors who provided code submitted a Release form to Tony Hearn confirming that they were willing for him to use and distribute their code. The wording that was used in this process is preserved as part of the trail that backs up the use of the BSD License terms for the `Reduce` sources.

`BUILDING`

The contents of this file are the subject of Chapter 2. The current document is intended to update and replace this file, which may well be out of date.

`README`

This file contains licensing information about `Reduce` and about the various components which are used to build `Reduce` (`gnuplot`, FOX graphical libraries, `wxWidgets` graphical libraries, ...). Both as a matter of politeness and as part of a discipline in ensuring that license terms of all software involved are adhered to it is strongly desirable that this be updated whenever any new third-party software is brought into the `Reduce` tree.

3.2 An overview of how Reduce gets built

Building `Reduce` starts with having a Lisp system. The dialect of Lisp used is known as Standard Lisp and is not at all the same as Common Lisp as used elsewhere. You are generally expected to start with either the CSL or PSL implementation of Lisp. In the case of CSL that is itself built from sources held within the `Reduce` file-set with much of the code written in C or C++. For PSL the main sources of the Lisp are coded in Lisp and are again held within the `Reduce` tree, but building `Reduce` normally starts by using a ready-compiled binary version of PSL.

A carefully written script starts with a Lisp-coded version of a parser for the `rlisp` syntax, and uses that to read various key source files from the `packages` directory. Some of these have had to be written in a stilted subset of `rlisp` because they are read in at a stage where the full language is not available. The order in which files are compiled can matter because later ones can rely on capabilities provided by earlier ones. The identities of these most central files can be seen in `packages/package.map` where they are marked with the tag `core`, and the order in which they are processed is the order in which they are listed there. Although there are differences between the details of how CSL and PSL treat these in both cases a "bootstrap" version of `Reduce` is created incorporating just these core packages. This bootstrap version can then be used to compile all the other packages. Ideally the order in which other non-core packages were compiled would be unimportant, but in reality there are dependencies between some of them and the order in which they are listed in the `package.map` file should be adhered to.

Once all packages have been built it is possible to create a snapshot of the state of `Reduce`'s memory in such a way that it defines the state `Reduce` finds itself in when it is subsequently started. Packages can have three different states relative to this heap image file. Some of the most central ones will be loaded into memory and are fully and immediately ready for use. For instance the parser and very basic algebra come in this category. Others will have the keywords that call for their use set up to be triggers that cause the package to be loaded into `Reduce` when first used. The integration and factorization code are handled in this way, and the file `packages/support/entry.red` contains the catalogue of auto-load triggers. Further packages exist and are available to all `Reduce` users, but need to be activated using a statement of the form `load_package groebner;` (to load the groebner-base package) before use.

3.3 The PSL Lisp system

PSL (Portable Standard Lisp) was originally built specifically for the support of `Reduce`. At a later stage it was under the care of Hewlett Packard, but it is now available under the BSD Licence as so may be freely copied or built on.

The key idea it followed was that essentially all of the Lisp system should

be coded in Lisp. To make that feasible it defined a set of definitions known as Syslisp that support operations on machine words and pointers, and the PSL kernel is written in this. A bootstrapping process compiles this code to create a usable but minimal PSL kernel that can be used to build the rest of the system. It is normally necessary to have some sort of working PSL implementation available in order to perform the bootstrap process. PSL has compilers that can generate native code for a wide range of processors, but sometimes variations there are required not only to account for hardware but also for the operating system being used. While all relevant source file for all of PSL are available as part of the `Reduce` distribution the normal way of building `Reduce` starts by having pre-made PSL binary files to use. These are provided for a range of targets (mainly using Intel processors) but machines by SGI and Sun (at least) are also reasonably easy to find.

There is documentation of the PSL in the `psl/doc` directory of a full `Reduce` file-set, and so perhaps little more is required here.

3.4 The CSL interpreter

CSL (Codemist Standard Lisp) is an alternative Lisp. Its development started in the late 1980s when it was built with a kernel coded in C to replace Cambridge Lisp that had been built using BCPL. The key files relating to it are present in the `csl/cslbase` subdirectory of a `Reduce` tree. Normally CSL is built entirely from source as part of the process of configuring and compiling `Reduce`. This increases the chances of it building on a new or unusual architecture, and on today's faster machines it still only takes a few minutes for everything to build.

CSL uses the FOX toolkit to support a windowed interface under Microsoft Windows or X11. There has been some work towards a transition to the use of wxWidgets which would also support OS X, but that is not at all complete. There is a directory `csl/support-packages` that contains original versions of the third-party software that a CSL build relies on, together with commentary explaining how their license terms interacts with that used by CSL itself.

The full process of building `Reduce` using CSL happens in a number of stages:

1. The `configure` scripts probe the current machine environment and so set up file-paths and options that will allow CSL to build.

2. The CSL kernel is compiled. This is just a few dozen C files for CSL itself, but the first time CSL is built on any particular platform it will also involve the compilation of all the C++ files making up the FOX library.

3. In normal use the CSL executable needs a ready-built heap image to load. This image will contain saved definitions of extra Lisp-coded functions. To create such an image the system is launched with a "`-z`" flag that instructs it to perform a cold start. It then needs to read in a file containing various important bits of Lisp that upgrade it from being a Lisp kernel into a fully usable system. One of these parts is a Lisp compiler that compiles raw Lisp into a compact bytecoded format that the kernel can execute rapidly. When building just a Lisp an image file `csl.img` containing this is dumped. When building `Reduce` a version of the whole system that represented compiled code in bytestream format is created. This is called `bootstrapreduce.img`, and if forms a complete version of `Reduce` but one that is slower that the final version (by perhaps a factor of two).

4. `bootstrapreduce` is used to compile the top 3000 (roughly) functions that make up `Reduce` from Lisp into C code. The generated C is put in `cslbuild/generated-c` in files `u01.c` to `u60.c`. A file `profile.dat` created by running a measuring script tracks which functions are liable to be the ones most important for overall `Reduce` performance. A main `Reduce` executable is created by adding compiled versions of `u01.c` etc to the files that made up the basic Lisp kernel.

5. A new image file `reduce.img` is created leaving the most important functions to be implemented as C code in the kernel but putting all the less important ones as bytecode streams in the image file. This is the final version of the system that normal users will interact with.

Since the inner parts of CSL are all coded in C one might hope that it would be easy to adapt it to interface with almost any other C code. One variation on that is when a user needs to embed `Reduce` as a component in some larger product. For those purposes the GUI code in `Reduce` is liable to be irrelevant and anything that makes the build process more complicated is going to be unwelcome. Anybody with that sort of need is directed to the `csl/embedded` and `csl/new-embedded` directories that provide a

framework for a cut-down simplified system without a serious user interface but potentially much more convenient for use as a component.

A key idea that CSL uses to try to achieve respectable performance is to compile sections of `Reduce` into C, and to pick the most important bits to treat this way. This is achieved by profiling the system. When you run `bootstrapreduce` you can use the statement `lisp mapstore t;` to display information showing which `Reduce` functions had been most heavily used. Well actually the mechanism used sorts functions based on how many bytecode operations where obeyed within them but scaled to give higher priority to short functions. For instance after running the test script from `packages/alg/alg.tst` the output it generates starts off with

```
2: lisp mapstore t;

  Value  %bytes (So far) MBytecodes Function name
   4.11    1.05 (  1.05)         0: simpcar
   3.56    1.43 (  2.48)         0: noncomp
   3.45    1.01 (  3.49)         0: reval
   3.21    1.29 (  4.78)         0: exchk
   2.87    0.84 (  5.62)         0: argsofopr
   2.63    1.15 (  6.77)         0: terminalp
   2.63    2.11 (  8.88)         1: smember
   2.29    1.25 ( 10.13)         0: getrtypeor
   2.18    0.64 ( 10.77)         0: delcp
   ...
```

This indicates that the function `simpcar` was liable to be the most important function to select for compilation into C. Of all the byte-code operations performed running the test 1.05% of them were in `simpcar` and because it is a short function it got allocated a merit score of 4.11. The function `noncomp` came second with a lower merit score even though it used more (1.43%) byte-codes. Looking further down `smember` used even more byte-codes, executing just over a million of them. The top 8 functions between them account for fully 10% of all the byte-codes executed, and looking further down it is possible to see that the top 65 functions use 50% of all byte-codes. This style of profile information can sometimes be useful when trying to understand the performance of your own code, but of course as with most similar schemes it can take time to make sense of it and just

knowing that `simpcar` (say) is heavily used might not of itself suggest ways to change that fact.

Every so often the full set of `Reduce` test scripts are subject to this profiling to create a file `profile.dat` that guides the process of compiling things into C. You can create a fresh version of this file by selecting the directory where `Reduce` has been built as current and going `make profile`. It should not be necessary to do this at all often, but following significant modification of the central parts of `Reduce` or after adding a substantial new package that is liable to be such that its speed is critical it can be useful.

3.5 Algebraic and Symbolic mode

In normal circumstances when `Reduce` is started it presents the user with a world where the main data type is one that represents algebraic formulae. It is possible to write programs at this level, for instance

```
procedure legendre(x, n);
  begin
    scalar r;
    r := (x^2-1)^n;
    for i := 1:n do
      r := df(r, x)/(2*i);
    return r;
  end;

for i := 1:5 do
  write legendre(v, i);
```

Given a formula it is possible to extract components from it, as in

```
w := (x-17)^3;
length w;          % 4
part(w, 0);        % plus
part(w, 1);        % x^3
part(w, 2);        % -51*x^2
part(w, 2, 0);     % minus
part(w, 2, 1);     % 51*x^2
part(w, 2, 1, 0); % times
```

Using `Reduce` in this mode is referred to as "algebraic mode" usage. It provides an easy transition from mere interactive or simple scripted use of the system and if the operations used are mostly calls to expensive operations from existing `Reduce` packages it can be fully as efficient as anybody could ever need. But some tasks will require direct access to the `Reduce` data structures in order to be viably efficient or to have flexible enough control to achieve what they need to. For this one descends to "symbolic mode". This can be achieved either by prefixing an individual `Reduce` statement with the word "`symbolic`"[2], or by using the directive "`symbolic;`", which sets the system into low-level mode until a matching "`algebraic;`" command is issued.

While the syntax used to write code is almost exactly the same in symbolic mode its meaning is different. The values stored in variables and manipulated are now items of Lisp data. This means they can be symbols, strings, numbers or lists. The functions that operate on them are no longer the algebraic operations that `Reduce` as an algebra system perform but the low level functions provided by Lisp, augmented by all the symbolic mode functions that form the implementation of `Reduce` itself. An immediate consequence is that in symbolic mode the "`+`" operator can only add numbers together, and a name is expected to be the name of a variable unless it is prefixed with a quote mark (in which case it will be treated as data in the form of a Lisp symbol). The set of Lisp functions available are documented in the Standard Lisp Report[9], but note that the `Reduce` developers have made small changes and extensions to that since 1979 when it was written.

More of this document is concerned with coding at the symbolic level, and arranging that what you implement there becomes available to ordinary users who do not need to delve that deep.

3.6 Central parts of Symbolic Mode Reduce

`Reduce` as a whole is "just" a reasonably large `rlisp` program. In total there are roughly 400,000 lines of code amounting to 13 megabytes of text.

The size of the code means that when you start working with it it can seem hard to know where to find things. Because `Reduce` has grown organ-

[2]the word `lisp` can be used as an alternative

ically over a period of fifty years the arrangements are not fully uniform, but there are nevertheless workable policies about where particular bits of code will live.

The source code for `Reduce` itself lives in a directory called `packages`, and that contains a file `package.map` that lists the modules that exist, and then for each sub-part of `Reduce` a directory that contains the source and test files for that component. Some of these modules may be referred to as "core". What this means is that during the process of building `Reduce` the "core" packages are built first to create a subset of the full system that is sufficient for use while building the rest. While building the core there are issues that mean that the order of reading in packages can be quite fragile, and the list in `package.map` specifies the sequence that is used. Ideally the remaining packages could be built in any order, but for a variety of reasons this is not quite the case at present.

The hope is that the names of the directories that contain packages will generally indicate what is contained there, and certainly for the higher level and more specialised modules it will be clear how to find material. However the code to implement advanced functionality will tend to call a lot of functions that perform lower level operations, and it may not always be instantly apparent where to find that for review. One way of dealing with this is to just search, as in using the Unix-style commands

```
cd packages
grep "procedure somefunction" */*.red
```

to find a definition of `somefunction`.

As has just been explained, the `Reduce` source code is partitioned so that the parts that implement some particular module or algebraic transformation live together. For instance `poly` contains the code that manages the representation of polynomials and that provides basic operations such as addition and multiplication, while `specfn` is where Bessel and other special functions are implemented. However the nature of algebra is that once there are fundamental domains such as polynomials and rational functions in place there will be many higher level operations that work on the same sort of data. Thus unavoidably there are bodies of code that perform polynomial operations in many places beyond the `poly` directory. For instance `factor` and `groebner`, with `ncpoly` adding support for polynomials with non-commuting variables and many other packages using polynomials as

their key data. Similarly rational functions and general prefix forms may have their most fundamental support in `alg`, but they are used everywhere and so functions that act on them are present within almost every module. The package `rlisp` contains the bulk of the `Reduce` parser and so is where to look to understand in detail how syntax is dealt with, but it provides hooks so that other packages can add their own special infix operators or other notations.

This all really means that while it is generally easy to find the code that implements a high level operation (it is liable to be in a directory named after the module providing that operation: that may be a little cryptic (eg `susy2` deals with supersymmetry, `scope` is a Source-Code OPtimization packageE...) but will usually not involve too much searching. Similarly the lowest level of fundamental support is well localised (eg in `rlisp`, `arith`, `poly` and `alg`). But for intermediate levels textual searching probably is the best basis for navigation.

3.7 On the legacy feature of upper case coding

`Reduce` started life around 50 years ago! At that stage everything was `UPPER CASE`. Some while later computers started to provide lower case input options, and some people liked to write code in lower case. To survive with that Standard Lisp and `Reduce` introduced a flag called `*raise` that (when set) caused all input to be folded to upper case. That was set by default and about the only time `*raise` got changed was that when Reduce was parsing a string it made temporary adjustments so that strings preserved case.

Hypothetically users could switch `*raise` off and write their own code in a case sensitive style but almost nobody did. It was more the case that different contributors adhered to different capitalisation conventions and so overall throughout the source there is not total consistency. A decade or so later having all internal names in `UPPER CASE` started to feel archaic and the Lisps were altered so that the internal functions became `car` and `cdr` not `CAR` and CDR (etc).

CSL and PSL supported this in different ways. PSL kept the one flag `*raise` and when set in modern versions that folds input to lower case.

CSL introduces a new flag `*lower` so that if `*raise` is set that input up folded up, if `*lower` is set it is folded down and if both are set then who knows... that situation was not thought about. *Neither* of these schemes are in the Standard Lisp Report (which was not updated) so at some level relying on either is "delicate". Anyway in both cases the internal names of almost all items within `Reduce` are stored to be in lower case. If anybody needs to use a symbol whose name has both cases in they can escape it as in `!Camel!Case!Word`. The exclamation marks here are of course a bit ugly, so some may feel that they wish to disable `Reduce`'s case folding so that words remain just as they appear in the source whether they are in upper, lower or mixed case. The notes here explain on why that option has not been taken throughout.

`Reduce` has collected contributed modules from a range of authors, and they adhered to a range of different conventions. If you look in the code right now you will see in e.g. `matrix/matpri2.red` (and a bunch of other places) code that goes `SYMBOLIC PROCEDURE ...` in upper case. In `misc/lie1234.red` it says `HE:=SYMBOLIC READ()` and the function `READ()` must mean the internal Lisp function `read()`. There are other modules where people have chosen to use mixed case names for their own functions and variables but where I know that they have not been quite consistent. So if `Reduce` was abruptly case sensitive by default there would actually be a significant amount of review needed with changes in many many files. It is also not clear that this would be trivial to do using a simple script in that the treatment of words in comments as distinct from code would need sensitivity, and where possible it would be desirable to clear changes with original package authors to preserve the best prospect of them being willing to continue with some support.

Probably there should be a long term objective of making `Reduce` case sensitive. Fairly recently the CSL and PSL sides have been in discussion about having *one* variable to control case not two and making the two Lisps behave the same in that area. If we started again that might remove both `*raise` and `*lower` and introduce a new flag `*input_case_control` (say). However for compatibility reasons in a load of places that would hurt. So the current *discussion* (nothing has yet been agreed or done) has as one prospect that `*lower` goes away (or at least becomes deprecated) and that `*raise` is then expected to hold not `nil/t` but one of `lower/nil/raise` as its value. If then setting `*raise` to `t` meant "fold case to whenever the underlying Lisp uses natively" that would be pretty upwards compatible with PSL usage.

Note that we do not support Common Lisp but it seems it tends to be `UPPER CASE` internally. Thus `!*lower` is under some threat and while the `Reduce` build process tries to set flags that inhibit inspection of `.redurerc`, setting `*raise` or `*lower` there is something potentially dangerous.

As an example, one might like the name `FandG` but in `Reduce` that must be input as `!Fand!G`, or conversely it can be accepted that it really means just `fandg` internally. A mixed case world might be more "modern" but it seems that while individual users can turn off `*raise` and still be CSL/PSL compatible those who turn off `*lower` are limiting themselves to CSL. And people who might contribute code to the community are way safest using `!Fand!G` (note that the exclamation mark is not part of the name – it is just a notation saying not to mess with the next character). So the name as a STRING is then just `"FandG"`. As an alternative much code in `Reduce` would use a name like `f_and_g` instead.

If a user put something that resets `!*lower` into the `.reducerc` file that is OK but really risks confusion when the user sends code to another user for testing and they do not have that option globally set. Furthermore reliance on `*lower` may tend to lock the code out from use with PSL... ensuring compatibility there takes a little more care.

Thus in my opinion is safer to accept that for historical reasons `Reduce` should be viewed as a single-case platform at present. The definition might be left as `procedure FandG x;` ... with an upper case letter written, but it should then be avoided having functions with names like `FandG`, `fandg`, `fANDg`, `FaNdG` etc. since it should be expected that all ends up as `fandg` internally.

All that said the user could put `off lower; off raise;` at the start of each of their source code files if desired. The `off raise;` is because that is what PSL needs to defeat its default behaviour of folding things to lower case, while the `off lower;` is for CSL's benefit. This whole area is just one of the many where a keen volunteer willing to put time into not just improving the code but also in contacting and coordinating with the authors of packages they change (for instance to ensure that those authors will still feel able to maintain their own packages) would be a real help to the overall project.

3.8 Data structure in Reduce

Here we will describe the main data-structures used by `Reduce` since those are important for all code, new and old.

Firstly when `Reduce` or `rlisp` code is read in either from the terminal or from a file it is parsed into a tree representation that is in effect just Lisp code. By using the `Reduce` directive "`on defn;`" it is possible to get this form displayed. You then need to go "`off defn;`" to return to a mode where what you type is actually obeyed. For the normal person using `Reduce` as an algebra system the `Reduce` parser adjusts input in this generated Lisp so it contains explicit calls to the `Reduce` functions that simplify expressions. After the directive "`symbolic;`" the code is left as Lisp that does just what the input code indicates and this is the mode in which most new `Reduce` bodies of code will be written. The code-base contains two options that are worth mentioning here if only to avoid confusion. `rlisp88` is an alternative extended version of the `rlisp` syntax that was, as its name might suggest, being worked on in around 1988. Despite the potential benefit of the language extensions the bulk of `Reduce` code is written in the earlier dialect (known as just plain `rlisp`) and it is probable that anybody who wanted to experiment with `rlisp88` would need to review and possibly update the code that implements it (which is stored in an obvious location in the packages directory). In a rather similar manner `Reduce` 4 represents research into what could be described as a more object oriented implementation of the inner algebraic structures of `Reduce`, built in a way that could lead to better code clarity and long-term reliability. Its foundations lie in order sorted-logic and the code involved remains in `packages/reduce4` but it exists at present as a starting point for a new project to bring `Reduce` internals up to date rather than as a finished viable part of the current working system.

Much of the time `Reduce` will be storing and working with algebraic formulae. There are two separate representations for these. The first is called "prefix form" and is mainly used for input and output, and is hardly ever used for serious computation. This structure is a fairly direct mapping of formulae onto trees that are built out of `rlisp` lists. For each algebraic operator there will be a symbol used to stand for it. The symbols are generally neatly spelt words, so `plus` is used instead of `+`. Each part of a prefix form is a list whose first element is its operator and the remaining ones are arguments. Thus

```
(difference (plus (expt (sin x) 2) (expt (cos x) 2)) 1)
```

The `rlisp` parser can read in a sequence of characters like `sin(x)^2 + cos(x)^2 - 1` and build this structure, and other parts of `Reduce` can convert the prefix notation back into something suitable to display to the user. To find out what format is involved, and specifically to see what name is used within `Reduce` to indicate some particular operator, either of the following recipes may assist:

```
% Observe the input...
on defn;
int(1/sin x, x);
off defn;
% Observe output...
share xxx; % xxx is "shared" between algebraic and Lisp
xxx := int(1/sin x, x);
lisp prepsq simp xxx;
```

There will be further commentary about the functions `simp` and `prepsq` later on here.

The bulk of `Reduce` uses a different representation for formulae. This notation starts as one for representing polynomials and fractions so that they are normalised or standardised. Variable are kept ordered and the highest degree terms in a polynomial come first. This canonical representation makes many internal algebraic operations fast and reliable. Following on from the previous illustration one can observe example of this internal representation using an interaction of the following style:

```
share xxx;
xxx := (a-b+12345667)^3;
lisp simp xxx;
```

Many users of `Reduce` will not investigate at a low lever that sees details of internal representations, but when debugging new code (even in algebraic mode) it can sometimes be useful to understand it since trace and debug output may display raw data structures.

The first representation I will explain is known as a "standard form" (SF) and it is used to represent polynomials. There are very many functions built in to `Reduce` to perform operations on these polynomials, and often their names end in `f`, as in `addf` and `multf`. These are defined in `packages/poly/polrep.red` with backup from other files in the `poly` directory.

A SF can be `nil` (representing the value zero here), an integer or a prefix form identifiable because its `car` (ie first item) is a symbol. This last case is used to cope with polynomials whose coefficients are things more elaborate than just integers – for instance complex numbers or extended precision floating point values: for much of the time you can ignore that case. A function `domainp` is the official way to detect this case.

When a SF is not a domain element it represents the sum of two parts - a "leading term" and a "reductum". Internally this is just a simple `rlisp` or Lisp level pair, so creating and accessing the parts is cheap and simple. `Reduce` provides notation to build and access things so that rather than using an infix `.` and functions `car` and `cdr` you use infix `.+` and functions `lt` (leading term) and `red` (reductum).

A leading term is a power paired with a coefficient, where the power is a variable paired with an exponent and the coefficient is just another SF. Polynomials in several variables are handled by having a polynomial in one selected "main" variable whose coefficients are polynomials in the subsidiary variables. All terms that have the same main variable are sorted so that the highest degree one comes first. It is important that every polynomial keeps variables arranged in consistent towers, and the function `ordp` is used to specify a normal ordering. For cases where the default ordering is unsatisfactory there are functions like `reorder` and `setkorder` but when those are used it is vital that everything is set back to a default state afterwards. See `polrep/reord.red` for the implementation.

An example of creating a simple polynomial `(3*x^3-4)` from scratch could be

```
(mksp('x, 3) .* 3) .+ (-4);
```

and functions with names such as `mvar`, `ldeg`, `lpow`, `lc`, `tvar`, `tdeg` are provided for those who need to navigate and (for instance) find the degree stored within a term. In data structure terms the above expression would display as

```
(((x . 3) . 3) . -4).
```

Formulae such as `sin(x)` that are not polynomials are dealt with by using the prefix representation of the formula as a variable in a SF, so `sin(x)` as a polynomial will be

```
((((sin x) . 1) . 1) . nil)
```

I believe that the best way to learn about standard forms and to discover the key functions relevant to their use is to read the code in `packages/poly/polrep.red` where the code that performs such operations as addition and multiplication is kept. You will find there indications of extra concern so that `Reduce` can cope with non-commuting variables, see that sometimes pattern-patching rules may be applied during basic arithmetic, and spot how some flags alter behaviour – eg a flag called `!*exp` (set using "`on exp;`" and cleared using "`off exp;`") disables a large proportion of the normal simplification of expressions.

Standard Quotients (SQ) represent the quotient of two SFs, reduced to lowest terms. They are created using a constructor `./` and their components are extracted using `numr` and `denr`. SQs are the main widespread way of storing big formulae, and if an expression is in fact a polynomial it will just be stored as a SQ whose second component is 1. Just as there are many functions that handle SFs there are functions with names such as `addsq` and `multsq` that combine SQs.

The top level of `Reduce` works by reading in an expression in prefix form, converting it to a SQ (and in doing so it simplifies and standardises it) and if the result is to be displayed the SQ is converted back to prefix form as part of the printing process. Key functions that perform these transformation include `simp` which takes a prefix from and returns a SQ, and `prepsq` which takes a SQ and returns a prefix form. A much more detailed description of the evaluation, simplification and conversion functions can be found in Section 4.5.

This repeated conversion backwards and forwards could become wasteful, so in certain circumstances `Reduce` uses `mk!*sq` to convert a SQ into a prefix form that has the special operator `!*sq` and then the undisturbed SQ as the next item in its list. This allows SQs to end up embedded within

prefix forms just as the scheme shown above to cope with sin and cos allows prefix forms to appear within SQs.

If a programmer makes a mistake and passes a prefix form or an SF to a function that expects an SQ (or any other similarly mistaken action) `Reduce` will not provide a nice informative compile-time complaint. It will almost certainly crash when the incorrect code is executed, and this crash may be messy. This lack of safety in the code is one of the consequences of `Reduce`'s age. There is at least some experimental code in the `assert` package that can help detect any such problems, but if you write code at the symbolic level you mostly just need to be careful. With a little experience you may become use to the styles of crash report that various styles of mistake tend to lead to.

3.9 Higher level Reduce packages

The majority of the packages in the directory `packages` is made up by user contributed packages. They can be of general purpose or more specialistic. To find their symbolic mode entrypoints (or names of procedures) scan their code for places that establish `symbolic operators` or set up `simpfn` or `opfn` properties on symbols (see Section 4.1 for explanations on these properties).

It goes beyond the scope of this document to give even a superficial description of the code in the "contributed" packages. On the other hand, the authors welcome the submission of extensions to this document aimed at describing features of `Reduce`'s packages that might be of general interest.

Prospective authors of new packages might wish to contact the maintainers of code on which their packages are based for possible advice or even cooperation.

3.10 Finding out specific features in the Reduce sources

In this section we will try to answer to the question: "Where, in the `Reduce` sources, is XXXX implemented?", elaborating beyond the commentary given earlier.

Well of course (eg) the bulk of the indefinite integration code is in `packages/int/*.red` and the factorizer in `packages/factor/*.red` etc. Most of the parser for the `Reduce` language in is `packages/rlisp/*.red` and the core of the bits that drive the simplifier are in `packages/alg/*.red`, but lots of the way that `Reduce` is built is such that the core bit provide mechanisms that let other packages or modules extend things, so loads of stuff is somewhat scattered!

So here is one strategy to find stuff.

Given a procedure or operator or keyword, eg "hah", go

```
lisp prop 'hah; % in PSL and CSL
```

or

```
lisp plist 'hah; % in CSL
```

and that gives the "property list" of the symbol whose name is "hah". You may spot `'simpfn` or similar properties... if so they give you the name of the function that implements the thing. Eg `plist 'int;` has an entry `(simpfn . simpint)` in it so `simpint` is an entrypoint, or the name of a procedure. Later on there will be further discussion of some of the properties that may be observed within `Reduce`.

Now you may use a program for searching string in a directory tree of text files, like `grep` in GNU/Linux environments, to search for `procedure simpint`. Files in `packages/*/*.red` have a good chance of being the place where that is defined.

The above is not that wonderful, but is a start!

3.11 Finding the documentation of Reduce

There is a lot of documentation scattered in `Reduce` sources. The purpose of this chapter is to put together a list of all potentially interesting texts inside and about `Reduce`. We will describe the available documentation within the different topics to which it pertains. It may be useful to note from the start that these days Reduce is supported by volunteers rather than being commercial product, and there are liable to be places where existing documentation is incomplete or out of date. Obviously from time

to time effort will go into improvements on that front! One general policy is that existing documents will be retained (if only for historical interest) even when or if they have become somewhat out of date.

Building and installing. The main file to start with is the file `BUILDING` in the main directory. It describes the building of `Reduce` from the source on the main platforms that are expected to be important.

The directory `debianbuild` contains a README file about building Debian packages for `Reduce`. Such packages may be transformed into RPM packages by the program `alien`, and similarly `winbuild` relates to creating a packaged binary release for Windows. In general only those concerned with making neat binary releases for others will need to concern themselves with these, but the curious may nevertheless find interest there with documentation embedded in scripts as well as in the form of separate files.

CSL. The CSL reference manual is the file `csl/cslbase/csl.pdf`. Even if this document is in a draft state it is a unique source of information about CSL's internals. In the same subdirectory also lies a series of text files that are provisional reports on the development of CSL. For example, the file `gui-non-gui.txt` describes how to start `Reduce` in windowed or terminal mode and contains a discussion and explanation of some of the design decisions that led to what initially looks like a very complicated mess of different versions.

Documentation for the FOX toolkit used to support the CSL GUI can be found in `csl/fox/doc`. It is also accessible from the file `index.html` in the directory `csl/fox`. See also the README file in the same directory. The definitive source of information about FOX is the library's own web-site `http://www.fox-toolkit.org`.

The folder `csl/reduce.doc` contains an HTML version of the `Reduce` 3.8 manual to be used with the graphical interface. In general the master copy of the manual (see later) should be consulted in preference to this.

The folder `csl/support.packages` contains several libraries and utility programs, like: TEX fonts, the `gnuplot` program for scientific drawing, the `distorm` disassembler, etc. These are provided so as to give full credit to the authors of software components written by others but used within `Reduce`, and a sequence of README files there should comment on the exact origin and license terms of each of the components that are present.

PSL. In the directory `psl` there are several documents of interest. The most important is probably `pslman.pdf`, the reference guide of PSL. Other manuals, notably `pc-install.pdf`, `pc-oper.pdf` and `unix-install.pdf`, `unix-oper.pdf`, contain instructions on how to install PSL `Reduce` under Windows and Unix systems. Some of the installation guides relate to installing PSL from the media that was distributed prior to the move to Sourceforge, and need to be interpreted in that light.

User manual of Reduce and its packages. Perhaps the first thing to note is that the main `Reduce` manual lives in the directory `doc/manual` in the `Reduce` source tree, but that what is there is initially just the LaTeX source for it. You need to go `make` to run the scripts that convert it to a .pdf file that you can view easily. This scheme means that the documentation can be updated incrementally in just the same way that the source code can, but it does mean that users need more software tools installed (*i.e.*, LaTeX) to use it! The main manual encapsulates the original `Reduce` 3.8 user guide by A. Hearn and the manuals of nearly all the contributed packages.

Standard Lisp report. This document, written by J. Marti, A.C. Hearn, M.L. Griss, C. Griss, laid down the definitions of Standard Lisp, which was meant as a standard basis for the development of `Reduce`. It is of primary importance for those who would like to understand the symbolic mode of `Reduce`.

Symbolic mode primer. This document was written by H. Melenk [5] and is contained in the directory `doc/misc`. As we said in the Preface, the document provides overlapping information about `Reduce`'s internals with respect to this text, and we encourage the reader to integrate the two to have a more complete picture.

Add-ons There are several auxiliary programs which are contained in the folder `generic`, as we already discussed. We just stress that the programs are well-documented by manuals that can be found in the relative subfolders of the folder `generic`.

Chapter 4

Low-level features of Reduce for Programmers

4.1 Extending Reduce with new operators.

There are different ways to extend `Reduce`.

New procedures can be introduced in a way which is covered by the manual, and it is conformant to the general idea of "procedure" of any modern programming language.

A `Reduce`-specific possibility is to introduce new *operators*. Operators are different from procedures in many ways, and the purpose of this section is to uncover some of the many features of operators which are hidden in the source code of `Reduce`.

If you write `f(a,b,c)` in an expression I will call `f` an "operator". One case of that is a sort of abstract operator with no special meaning so in algebraic expressions it just remains as `f(a,b,c)` with a visible `f`. Other sorts of `f` are names of built-in nouns or verbs like `df` and `int`, and `besselj` and `part` etc etc. So there are many hundreds of such and the main manual tries to cover them. Some lie between, so eg `sin(x)` stays as just `sin(x)` mostly, but in some contexts there may be rules like `sin(x)^2 => 1 - cos(x)^2` or `sin pi => 0` that are applied. There are also array names which at some level are the same, but then `v(3)` simplifies to the third component of the array `v`.

A "procedure" is one of these where somebody has given a rule for how to process uses of the operator. So if you say `procedure inc x; x + 1;`

then `(inc 1)` becomes `2` and `inc y` becomes `y+1` and you should never end up with the word `inc` visible in your output. Note that the sequence

```
procedure inc x; x := x + 1;
v := 2;
inc v;
```

displays the value 3 as the result from `inc v`, but the value of the variable `v` remains as 2 - passing it as an argument to `inc` can not lead to its value changing. Overlapping with the concept of procedures is the idiom

```
for all n let dec(n) => n-1;
```

i.e. use of an unconditional rewrite rule. The rewrite rule is activated by sort of making a full expression that might have instances of `dec` in it then pattern matching to find them and replace them. I usually expect it to be way slower. But there are times it can provide more flexibility. The RHS of a `let` statement typically does not have program-style sequences of `begin ... end` on it while procedures often do.

If you go `algebraic procedure foo x; ...;` that defines a lisp-level function called `foo` and gives `foo` the `opfn` property, so it is if you like an operator that has a special treatment by the simplifier so that when it is seen the function gets called.

If you just say `operator xx` then `xx` gets given a `simpfn` of `simpiden` that again tells the simplifier how to handle it, and in general it makes `xx 2` remain as `xx 2`. If at some stage you set up `let` rules etc then the simplifier applies them using what I expect to be slow pattern matching - but of course pattern matching can be more flexible than activating the recipe of a little procedure.

Operators can be introduced in algebraic mode or in symbolic mode.

Both algebraic and symbolic operators are implemented in `packages/alg/algdcl.red`, together with the function `remopr` that is used if something is to cease to be an operator and all the flags (boolean `put`/`get` properties) that *could* have accumulated are removed. Then `alg/reval.red` contains the definition of `aeval` that "evaluates" an expression in algebraic mode. And `alg/simp.red` is also part of the same sort of stuff.

More in detail, in `algdcl.red` at the beginning you will find

```
symbolic procedure operator u; for each j in u do mkop j;
rlistat '(operator);
```

which introduced a new syntax for a statement

```
operator a1, b1, a2, ...;
```

that works by calling the `mkop` function in each item in the list.

The "`procedure mkop`" statement can be found in `alg/simp.red`. Apart from some checks what that basically does is

```
put('a1, 'simpfn, 'simpiden);
```

where `simpiden` is a "simplification function" that behaves like the identity transformation, so that `(a1(3))` remains as `(a1(3))`. That means that from symbolic code if you have a symbol and you want it to be treated by algebraic code as the name of an `operator` in this sense all you have to go is go

```
mkop 'x;
```

Note that to a decent approximation you can see what the symbolic mode version of any algebraic mode stuff in `Reduce` is by going `on defn;` after which the Lisp-form rendering is shown (and the code is not actually obeyed)

```
1: operator a1;
2: on defn;
3: a1(x,y,z);
    (aeval (list 'a1 'x 'y 'z))
```

Since an operator applied to some arguments is stored as a list, it is accessible (even) in algebraic mode by `part` as follows:

```
1: operator a1;
2: part(a1(x,y,z),1);
    x
```

```
3: part(a1(x,y,z),0);
    a1
4: arglength(a1(x,y,z));
    3
```

Note that the function `length` will not work in the same way as `arglength` on `a1(x,y,z)`.

Let us see the differences between the parsed forms of an operator and a procedure:

```
1: on defn;
2: operator abcd;
    (operator (list 'abcd))
3: (x-3)^3;
    (aeval (list 'expt (list 'difference 'x 3) 3))
4: procedure foo(a, y); df(a, y);
    (put 'foo 'number!-of!-args 2)
    (flag '(foo) 'opfn)
    (de foo (a y) (list 'df a y))
```

while in Symbolic mode you might write

```
x := mksp('x, 1) .* 1 .+ nil; % "x"
xm3 := addf(x, -3); % x-3
p := exptf(xm3, 3) ./ 1; % now it is a quotient with denominator 1
```

now `p` is the Standard Quotient for `(x-3)^3`, which you might convince yourself of by saying

```
prepsq p;
```

That is messier than

```
aeval '(expt (difference x 3) 3);
```

which is what the algebraic mode does (in effect), but it is sort of what `aeval` causes to happen...

Even if just for testing purposes one of the first things anybody starting out with symbolic mode `Reduce` will want to do involves setting up a link

between the user-level algebraic world and the new world of data structures and low level code that they are entering. Here are some recipes for different ways to achieve this:

```
symbolic operator f1;

symbolic procedure f1 a;
  << print a;
     print list('times, list('quotient, 22, 7), a) >>;

a := (1-y)^3;

f1 a;
```

The output from print shows that the function `f1` has been passed a prefix version of the simplified form of its argument a, ie

```
 (plus (minus (expt y 3)) (times 3 (expt y 2))
    (minus (times 3 y)) 1)
```

The value returned as also a prefix form, which here is the product of `(22/7)` and the argument. So by declaring something a "symbolic operator" you get a chance to define your own function where input and output are both in prefix form. You may of course use `simp` and `prepsq` to convert to internal representations inside your own code. Sometimes it can be useful to use `mk!*sq` to wrap up a result that is in SQ form for the return here.

```
symbolic procedure simpf2 a;
  << print a;
     print (355 . 113) >>;

put('f2, 'simpfn, 'simpf2);

f2 a;
f2 ((1-y)^3);
```

If you use `put` to give some name a `simpfn` property then the function you provide will receive a list of prefix forms and here the variable `a` will just be passed as itself (while with the symbolic operator case it got expanded

to its value). The function must return a SQ this time, so in my example I use just the SQ for the number `(355/113)`.

There are extensions to and variations on the above, but the examples shown here may be enough to get you started. Perhaps a point I should explain is that when there are `let` rules in play with `Reduce` there can be an issue as to whether an expression needs scanning again to ensure that all transformations have been applied, and this can interact particularly with `mk!*sq`. Some of the variants on functions (eg `simp` vs `simp!*`) may relate to this sort of issue. I view coping with it as an advanced topic and so will not discuss it further here!

The issue of defining operators within procedures could arise; consider things like the following:

```
1: operator f;
2: x := y;
x := y
3: y := z;
y := z
4: procedure hah w; df(f w, w);
hah
5: hah x;
df(f(z),z)
```

Within `hah` does `w` stand for `w`, for `x`, for `y` or for `z`? Within `hah` is there any way to differentiate with respect to `x` or `y`? The issue is the relationship between the algebraic symbol called `x` and the programming language variable(s) called `x`. Algebraic mode does not provide a notation to be explicit about when you evaluate things, or how many levels of definition should be worked through. This is the case for operator names as well as simple variables.

So in symbolic mode the worry does not apply. You always do exactly one evaluation, and you can use `'x` to mean "`x` itself" and `x` to mean "the value of the variable `x`.

So now consider

```
algebraic procedure ttt x;
<<
  operator x;
  {x 1, x 2, x 3}
>>;
```

now try

```
load_package rprint;
lisp rprint cdr getd 'ttt$
```

to see the Lisp level version. When I try that it DOES let me go

```
ttt banana;
```

and get back `{banana 1, banana 2, banana 3}` so I *can* define new operators within a procedure... but to get an interesting new name for one may be harder in algebraic mode.

4.2 Properties of symbols and their use

In this section we will explain how properties are used at a Lisp level inside `Reduce`.

In Lisp any item is either an atom or not. The function call `(atom x)` will test if `x` is an atom. If something is not an atom it is a pair, and `car` and `cdr` access its left and right components. You can create a new pair by using the function `cons`. Thus `(car (cons p q))` gets you back p.

`(cons a (cons b (cons c (cons d nil))))` can be done using `(list a b c d)`. Lists of the shape `((k1 . v1) (k2 . v2) ...)` *i.e.* `(cond (cons k1 v1) (cons (cons k2 v2)` ...are known as association lists (function assoc scans them) and are thought of as a way to have a table with `k1`, `k2` as keys. `cons` is used often enough in the `Reduce` sources that a shorthand syntax for it is provided: an infix dot (`.`) invokes it.

There are a bunch of different sorts of atom. In each case there will be a function to detect that case and then functions that operate on that sort of data. Strings are mostly used for printing not for computation, *e.g.*

```
"string" (stringp x) (print "message")
```

Perhaps the easiest suggestion for somebody who want to get started with string manipulation is to note that the function `explode2` converts a string into a list of its constituent characters and functions like `compress` and `list2string` can be used to convert in the other direction. Any manner

of string searching or concatenation becomes straightforward when it is performed on lists of characters.

Numbers come in a number of sub-flavours, and `numberp` recognises all

1, 2, 3	`(fixp x)`	`(plus x 3)`
1234567890123456	[ditto but a bignum]	
1.234	`(floatp x)`	`(plus x 3.14159)`

Small integers may be handled with extra efficiency and if you are certain that inputs and outputs are small (in CSL up to $\pm 2^{27}$ maybe) you can use `(iplus x 3)` rather than `(plus x 3)` and go faster.

In Common Lisp there would be ratios and complex numbers built in but those are not present in Standard Lisp. Long-precision floats are used in `Reduce` but implemented as pairs of integers (mantissa and exponent) and not as primitive objects.

Vectors, hash tables, (maybe) character literals and other things may exist but are *usually* not very central to usage such as `Reduce`. Well where they are needed they are very valuable indeed, but they live in corners. Eg for vectors (and I will use `rlisp` not raw lisp syntax here) are used as in

```
v := mkvect 10;
for i := 1:4 do putv(v, i, i*i);
for i := 2:3 collect getv(v,3-i);
```

Now to the case I have been building up to - symbols. Firstly a symbol is something that has a "print name" (which is liable to be a string) but unless you do funny things that will be just one symbol with any given name. You can have two quite separate strings each with the same characters in them, but symbols are looked up in a table so that if you enter some name (eg `car` or `procedure`) you get the same symbol each time. Each symbol also has a property list, the capability to have an associated (global) value and the capability of having a function definition associated with it.

All the built in functions like `car`, `cdr`, `cons`, `plus` etc exist as symbols built into Lisp with the even lower level explanation of those functions attached to them. In `csl` any symbol where nobody has defined a function for it has `i_am_an_undefined_function` as its function definition! Well that may not be the exact internal name used but you see the idea maybe.

Any symbol that has been declared fluid or global has a (valid) global value when accessed as a variable. There are a few special cases. The symbol `nil` has itself as its value and you can not change that. `nil` is used

to represent "false" in tests. `t` has the value `t` and is by convention used for true, but in fact anything non-`nil` will be treated as true.

Now to the real question, properties. A symbol can have properties and you set one up by going

```
put('symbol 'propertyname 'value)
```

and later on retrieve the value using

```
get('symbol 'propertyname)
```

A plausible implementation is that the data structure for a symbol includes an association list known as the symbol's "property list". There is no guarantee that this is how it is done but it is a good conceptual model at the very least. So at the `rlisp` level something is given a property any time you see `put('name, 'propname, <value>)` and the property remains in place from then until you chance it with a new `put` or remove it using `remprop` or the Lisp (`Reduce`) run ends. When `Reduce` is being built it checkpoints its state at the end so that function definitions, variable values and properties are saved in the image file to be reloaded when you start up `Reduce` ...

Sometimes a function `deflist` does multiple property list settings at once. `flag` and `flagp` and `remflag` deal with (in effect) properties whose value is restricted to being either `nil` or `t`.

"Having a property" is not a metaphysical state. It is *just* a statement that if the code used `get` it will retrieve what `put` placed there.

This is a bit like "a variable has a value". After somebody has obeyed `x:=3` the variable `x` has a value (which is `3`). In general the stage at which anybody notices this is when they ask for the value of `x`! So similarly a property being associated with a symbol is only relevant or noticed when the code uses `get` to see if it is there. The function `aeval` in `Reduce` explicitly looks for `simpfn`, `opfn` etc. properties when it is trying to simplify an algebraic expression...

A symbol can have as many properties as you like, and in some sense properties are not "defined" in any one place. There will be lines of code that do `put` and lines of code that do `get` but there is no obligation for these to be close together.

There are well over 200 different property names in use within `Reduce`: some are used *e.g.* by the compiler, some just by specific packages, some more centrally. There is no central registry that lists all of them – thus their exact behaviour needs to be discovered (supposed you actually need to know) by searching the code and reading parts that used them.

Again, we suggest to read the "Primer" [5] for another view on the above topic.

4.3 A bit on parsing

How does the parser/evaluation mechanism work? At what stage the evaluator understands that a symbol has a property? How the property is effected?

Firstly parsing and syntax are a quite separate issue from evaluation. You can get *just* parsing done by going "`on defn;`" and typing some `Reduce` in. You get echoed the internal form for the stuff. In symbolic mode 2+3 will parse into (plus 2 3). In algebraic mode you may get calls to `aeval` etc. interposed.

Evaluation is then just executing the stuff you see viewing it as a prefix tree as in `(plus 2 3)`. The "`plus`" is the name of a function to call and 2 and 3 are its arguments.

Sometimes when you create a new body of `Reduce` code you may wish to extend the system's syntax. This section gives an overview of what could be involved, because different styles or levels of extension will require different techniques and amounts of work.

New (prefix) operators

Having new syntax that just amounts to a newly defined name for a function or operator has to some extent been covered already. If a symbol is tagged with `opfn` or `simpfn` then `Reduce` will know that use of that name must trigger simplification via the appropriately specified body of code. If this is can suffice for you then everything is liable to be simple!

Operators with non-alphanumeric names

Sometimes people wish to use a new bit of syntax that involves a new operator written using special characters. There are existing packages that give an interpretation to "`-->`" and "`::-`" as well as "`><`" and "`<>`". The function `newtok` is used to make new sequences of punctuation marks combine to form single symbols, and `precedence` can then be used to control how expressions group. This form of extension should not be undertaken lightly if only because we do not have any organised central way to ensure consistent use of such syntax across all available modules. However once a new (infix) operator has been introduced in this manner it can be used very much as if it had been an ordinary prefix-notated one. Search the `Reduce` sources for instances of the use of `newtok` to find examples to build from.

Syntax introduced using a new keyword

More extreme than new operators will be fully new syntax introduced by a keyword. If a symbol is given a `stat` property then that specifies a function that will be called to parse whatever material follows an instance of the given symbol. Perhaps good examples to inspect to get an idea of what might be done are in `packages/rlisp/loops.red` where this scheme is used to implement the syntax that `Reduce` uses following the keywords `repeat` and `while`. Other places in the `packages/rlisp` directory use the same scheme to set up most of the rest of the varieties of `Reduce` statements. A special case is if the `stat` property is `rlis` that makes the parsing action to be to call the function `rlis` and this just reads a comma-separated list of expressions and builds them into a list. There are many cases where that is about all you need to do to get your extra syntax supported!

Generating elaborate code from your new syntax

Simple parser extensions using the `stat` property will tend to start off by building a list structure that is a direct reflection of the syntax used. For complicated formats it may be really useful to transform or expand that before it is really used by the rest of `Reduce`. One concrete example of this is given by the `for` statement as implemented in `packages/rlisp/forstat.red`. If a symbol is given a `formfn` property that specified a function that is called to re-form expressions parsed to give

an initial structure headed by that word. The form-function is given extra arguments that tell it what variables are in scope and whether parsing is being performed in symbolic or algebraic mode, and it should return something that could have corresponded to some more verbose `Reduce` syntax. This expansion process can discard material, duplicate other bits, rearrange and generate its output as if the input had been whatever mix of symbolic and algebraic mode material you wanted. You might also use the `form` scheme to perform checks on input and reject cases that are liable to correspond to user mistakes. Searching for `formfn` will find the existing uses of this scheme.

4.4 A bit on domains

If you have a polynomial in 1 variable it has a "variable" (and I will use `x` here) and a "domain" where the domain is the ring or field that coefficients are taken from. So in simple cases with `Reduce` an integer will be a domain element. And for historical reasons zero is denoted by `nil` not by 0.

If you want a polynomial with a high precision float as a coefficient then the "domain elements" have to be composite structures since simple Lisp or machine floats have a fixed limited precision. This is coped with by `Reduce` putting a coefficient that is something like

```
(!:rd!: mantissa . decimal-exponent)
```

where the two components are each integers.

Sometimes you have polynomials whose coefficients are reduced modulo some (usually) prime, then the coefficients (domain elements) are given as `(!:mod!: . nn)` where `nn` is smaller than your modulus.

Also complex numbers etc etc etc. In the concrete representation this is OK because each domain element is either atomic (a symbol or number) or has a symbol as its `car`. A term in a polynomial will always be

```
((v . deg) . coeff) . rest-of-lower-degree-term
```

so the test

```
if atom u or atom car u then .. % a cofficient / domain element
else ..  % a term non-constant polynomial with a leading term
```

does the job. This is in fact written in the code as

```
if domainp u then ...
else ...
```

and the inline function `domainp` is just a shorthand for the longer test.

We observe that the "Primer" [5] treats the same topic in more details, and we invite the interested reader to consider also that text on this issue.

4.5 Evaluation, Simplification and Conversion

As has been explained, `Reduce` has several different representations for algebraic expressions. Prefix forms (PF) are what the parser creates when expressions are initially read in. Standard Forms (SF) are for polynomials, while Standard Quotients are pairs of SFs and are the most important and general canonical representation that `Reduce` uses. In simple terms `Reduce` performs its calculations by starting with PFs and converting them into SQs. In the process of that conversion everything is normalised – and it is hoped that that corresponds reasonably to a user's concept of "simplification". To a good first approximation printing is achieved via a route that first converts an SQ back into prefix form, which can then be rendered in some human readable form.

It is thus useful to know about the functions within `Reduce` that perform conversions between the representations. There are a number of subtleties:

PF to SQ The functions `simp` and `simp!*` accept a prefix form and evaluate it, returning an SQ. For most purposes there is little to distinguish the two, and `simp` is slightly faster. `simp!*` can perform a few more transformations, as illustrated by:

```
on expandlogs;
off precise;
```

```
lisp;
simp '(plus (log x) (log y));
simp!* '(plus (log x) (log y));
```

where the call using `simp!*` performs extra scans over the expressions to combine the two logarithms into $\log(xy)$. The code for these functions is in `alg/simp` and other differences (eg some relevant when non-commuting variables as used in the High Energy Physics package, or for some uses of complex numbers arises). Most frequently you can just use `simp`. Note that the difference illustrated above where `simp!*` converts $\log x + \log y$ to $\log(xy)$ is only activated if various flags such as `expandlogs` are set, and that transformations of that sort need very careful consideration in the light of possible issues of the multiple branches of the logarithm function!

There is a quite separate scheme that does not so much convert a PF into an SQ but embeds it within one with relatively little simplification. This is `mksq` by using the PF as a "kernel" (i.e. treating it as if it was a a variable) and building an SQ that stands for that raised to a given power... except that if the variable raised to the indicated is the subject of a substitution rule (such as `let x**5 = 0;`) the substitution will be applied. Prefix forms where the leading operator is not a simple arithmetic one (for instance the cases where it is a trig function) typically use this.

SQ to SF A Standard Quotient consists of a pair of SFs, and these can be extracted using `numr` and `denr`. This is very often useful in cases where your code expects a polynomial, because `simp` will process an argument to obtain an SQ and then `numr` can extract the (polynomial) numerator. You will sometimes want to verify that the denominator is just 1 and complain otherwise.

SF to SQ The infix operator `./` combines a numerator and denominator to be an SQ. The function `gcdchk` (in `poly/polrep.red`) can then be used to ensure that it is expressed in its lowest terms - in other words that any common factor between numerator and denominator has been cancelled. In almost all circumstances it should be considered seriously bad form to create a data structure representing an SQ that

does not have common factors cancelled or that has a negative number as its denominator!

SF or SQ to PF The functions `prepf` and `prepsq` convert SFs and SQs back into prefix forms in a full and obvious way, while `mk!*sq` wraps an SQ up in a small prefix form wrapper (using a pseudo-operator `!*sq`). This latter is used where a prefix form is required for some reason but where it is expected that the value will soon be needed as an SQ again, so it avoids both the costs of conversion to prefix form and those of converting back. By hiding expressions away like this there would be a possibility that formulae packed by `mk!*sq` would not be sufficiently re-simplified if new `let` rules (and the like) were introduced between packing and the short form unpacking that `simp` would perform to unpack. To cope with this the special `!*sq` prefix form contains a flag such that all such forms share a single fragment of lisp data. When a new `let` is performed this data is overwritten (see `alg/rmsubs.red`) in a way that will force re-scanning of previously scanned forms.

PF to PF, simplifying The simple solution is normally to simplify a form `u` by going `prepsq simp u`.

Q to SQ, re-simplifying This can need to be done if some (new) substitutions need to be applied to an expression that you had already converted into an SQ. The function `resimp` does this, and as can be seen from its implementation (in `alg/simp.red`) it merely calls `subf1` (from `alg/sub.red`). Well the term "merely" here perhaps understates all of the complications associated with pattern matching and re-writes!

Other subsidiary types Various other sorts of data tend to end up hidden within prefix or SF structures. For instance lists are stored as prefix items with the "operator" `list` to mark them. Matrices are not fully free-standing items – a (global) variable can be either a scalar or matrix value and the `avalue` property associated with the name of the variable identifies this. For scalar values a single prefix form is then kept, while for a matrix a list of lists of prefix forms is used.

Domain elements are items used in SFs as coefficients. The most important case is that where a value is a simple integer, in which case the Lisp integer value is used directly (except that as a special case `nil` is used to denote the value zero: historically it that used to be a useful

performance optimisation even though now it is probably a cost not a benefit). Other cases are stored using a SF that has an atomic first element (where one that stood for a polynomial would have a pair standing for a leading power paired with an associated coefficient). The atom `!:rd!:` there introduces a floating point value, with one variant to cover one that are stored to native machine precision and another for higher precisions. `Reduce` itself supports complex values and modular arithmetic. Extending support to provide a new domain might involve significant work but would be possible.

4.6 Substitution

Substitution is described in detail in its aspects which are of interest for users in `Reduce`'s manual [2]. Here we would like to point the reader to the files where substitution is defined, and to some of the more relevant features that could be of interest for a programmer.

The main substitution schemes are represented by `sub` and `let`. `sub` is defined in the file `packages/alg/sub.red` and `let` is defined in the file `packages/alg/forall.red`. The main features can be summarised as follows.

sub The command is *local* in scope; this means that substitution is performed only when the command is issued, and only on the expression which is passed to the command. Moreover, the command performs the substitution *only once*, eg

```
sub({x=y+1,y=-x-2},x^2+y^3);
```

returns the standard form of `((y+1)^2+(-x-2)^3)`.

let The command sets rules which apply globally from the point at which the command is issued. This means what follows. Suppose that the expression `expr` contains a variable `x` which is to be replaced by the rule `x => y+1` (issued in a previous `let` command). Then as soon as `expr` is reevaluated, for example if `expr` is used in a bigger expression, then every instance of `x` is replaced by `y+1` in `expr`.

If we are in the case when the replacement expression still depends on the variable that is being replaced through another rule, then all rules

are applied through recursive pattern matching until the situation when rules become independent is reached. Here 'independent' means that any variable on the left-hand side of any rule does not appear in any right-hand side of any other rule. For example, the command

```
let {x => y+1,y => -x-2};
```

implies a substitution by `{x => -1/2,y => -3/2}`.

Substitutions that replace simple indeterminates with values are reasonably easy to use. Things become harder when the pattern sought becomes composite.

The mechanism of `let` rules is very powerful, but abusing it leads to great memory occupation and consequent slowing of the system. This can be caused by recursion in following ways:

- a long list of rules may contain a lot of cross-dependencies between variables to be substituted. However, it should be pointed out that there are concrete examples in which pattern-matching solving of a huge system of equations is faster that the standard `solve` routine!

- `let` commands create a tree of subexpressions that are bound to each variable to be substituted. Further rules can increase the branches of the tree in a way that at least empirically can lead to high costs.

A chance to avoid the above problems is the command `where`. This command is implemented in the files `packages/alg/forall.red` and `packages/rlisp/where.red`. Internally it is implemented as a local `let` command, so that 1 - the rules are evaluated recursively until independence (as above) is reached, then 2 - the rules are applied to the one expression which is on the left-hand side of `where`. Such an expression, after that rules are applied, is returned, and immediately afterwards the rules cease to have any further effect.

The simple case of substitution is where the item being substituted for is a single literal symbol. Two more complicated cases arise and one significant further complication can intrude.

The first is that the target for a substitution can be an expression, so for instance `let x^2 + y^2 => 1` will be accepted and will at simplify

`2*x^2 + 3*y^2` into `y^2 + 2`. But it will also lead to `x^2` being converted to `-y^2 + 1`, and so it could be that using the form `let x^2 -> 1 - y^2;` will be less confusing! A common use of substitutions for expressions will be a case like `let eps^3 => 0` which leads `Reduce` to discard third and higher powers of `eps`.

The second issue arises when operators are present in the substitution pattern, as in `let sin(x)^2 => 1 - cos(x)^2;`. If written like this the rule will apply to uses of `sin` where the argument is literally `x` and so `sin(y)^2` would remain untouched. The rule can be made more general by prefixing the first use of the free variable `x` with a tilde so it ends up as `let sin(~x)^2 => 1 - cos(x)^2`. An alternative to this may be to use syntax involving the phrase "`for all x`".

A complication arises in the internal way that `Reduce` handles substitutions when there is also an existing built-in rule that deals with some of the operators present in the pattern. A particular case of this might be substitutions that add a new rule for differentiation, where arranging that the existing and new rules cooperate nicely can be delicate. The limit of what can be explained here is just that such cases may need special care – and perhaps advice on a case by case basis from experts!

4.7 Adding a new module to Reduce

This is a cook-book style set of instructions for adding a new module, while I will suppose is to be called `maud`. Add a line

```
(maud   "maud"   test csl psl)
```

to the file `packages/package.map`. You will almost certainly want to add it close to the end of that file. The first entry on your line is the name of the new module you are creating. The second item (the string) is the name of the sub-directory of `packages` that its source files live in. If the word `test` is present then you will have provided a `.tst` and a matching `.rlg` file. Your test should run in a reasonably short amount of time – say at most a few seconds. Finally the words `csl` and `psl` are present to indicate which Lisp systems can support your code. Only in extraordinary circumstances should you consider providing something that will only work with one of the Lisps.

Create a directory `maud` in packages and in there place files with names `maud.red`, `maud.tst`, `maud.rlg`, `maudsub1.red` and `maudsub2.red`. A proper package should also have a file `maud.tex` that provides a section suitable to include in the `Reduce` manual or to provide a stand-alone explanation of what the code does and how it achieves its purpose.

The file `maud.red` will contain the text:

```
module maud;
% Author: ...
% Redistribution and use ...
% <copy the BSD license from any other source file>
% ... SUCH DAMAGE.
%

create!-package('(maud maudsub1 maudsum2), nil);

% Place fluid and global declarations here

endmodule;
```

The body of your code will be in `maudsub1.red` and `maudsub2.red` (you may use one, two or as many files as makes sense for your package. I am illustrating this imagining that two make sense.

Each source file will read something like

```
% maudsub1.red
% Author: ...
% Redistribution ...
module maudsub1;
symbolic procedure ...
endmodule;
end;
```

Now to rebuild your code you can start a fresh `Reduce` and go

```
package!-remake 'maud;
```

or just arrange to recompile the whole of `Reduce`. If anything seems to have become particularly confused or damaged the simplest recovery option generally goes

```
cd <the reduce trunk directory>
rm -rf cslbuild pslbuild
./configure --with-csl (or --with-psl)
make
```

which gets rid of any previously part-built files and so is as close to guaranteed to restore sanity as anything. Of course before following this path ensure you have not left personal files anywhere within the `cslbuild` or `pslbuild` directories!

The scripts `maud.tst` is a test file, with `maud.rlg` being the expected output from running it. This will be used by the `scripts/testall.sh` scheme that runs tests of all `Reduce` packages. To avoid worrying users it is good to keep `maud.rlg` up to date as otherwise those who test their own installation of `Reduce` may fear that they have a problem.

A reasonable way to create `maud.rlg` once your package has been build is to select the `Reduce` trunk directory as current and issue the command

```
scripts/test1.sh --csl maud
```

which should try your `maud.tst` file and create a log in `csl-times/maud.rlg`. You can then copy that file to `packages/maud` and on subsequent uses of `test1.sh` or `testall.sh` output will be compared against it. So when you change your code or test script in ways that alter the test log please regenerate and install a clean version using this recipe. Of course you could test using PSL rather than CSL.

When you have got past the basics you might wish to consider more sophisticated issues like auto-loading your code or pre-loading any supporting package that your code might need just before loading your package.

If you want your code to be auto-loaded on use you will need to add line to the file `packages/support/entry.red`.

Yes that is roughly it. The `defautoload` statement can have various forms:

```
defautoload(name); % The parens are optional here!
defautoload(name, packagename);
defautoload(name, package, functiontype);
defautoload(name, package, functiontype, n_args);
```

where `name` is the name of a symbolic mode function. If other things are omitted then `packagename` defaults to `name`, `functiontype` to `expr` and `n_args` to `1`. I think that `package` can also be a list of packages.

(For algebraic mode things there are `defautoload_operator` and `defautoload_value`. And all this happens in `packages/support/entry.red`).

The `autoload` stubs set up by `entry.red` are present in `Reduce` before any other things are loaded - or perhaps better to say when only the "core" functionality is loaded. So if you want *e.g.* `cde` to be a function that people can call you may end up putting either

```
defaultoload(cde, cdiff);
put('cde, 'simpfn, ...);
```

in `entry.red`, or maybe `cde` will be a separate package related to `cdiff` but loaded separately, then you might have

```
defautoload(cde, (cdiff cde));
put('cde, ...);
```

You need the `put` because without it an algebraic mode person would not get to have the list-level function called for them. OK so loading the package puts that in place again, but that does not hurt.

See also examples there like

```
symbolic operator meminfo;
defautoload(meminfo, rltools, expr, 0);
symbolic operator fastresultant;
defautoload(fastresultant, rltools, expr, 3);
```

for things that are symbolic operators with other than just 1 argument.

When you go

```
defaultoload(abc, def);
```

it is distinctly as if you had gone

```
symbolic procedure abc(x);
<< load!-package 'def; % I REALLY hope this (re)defines abc!
abc x >>; % the real version from package def.
```

so when this version of `abc` is called it overwrites itself with the "proper" version.

When you are sufficiently happy that your new package would be of use to others (and especially if you are willing to provide at least some level of support when people try it out) contact existing `Reduce` contributors and offer your code to the project. When contributing any code to `Reduce` you need to be ready to confirm that your code does not contain components that are encumbered by license terms that would conflict with the BSD license that `Reduce` uses and that you both have the right to and are happy to release your code on those terms. If you inspect the whole of the `Reduce` source tree you will find the names of the many existing contributors.

4.8 File and Directory management, Shell access

Sometimes a `Reduce` package (or indeed some user level code) will want to interact with the file-system of the computer it is run on. It may want to discover what the current directory is or charge that. It may want to enumerate, rename or delete files. It could need to run some command or program external to `Reduce`. The Standard Lisp Report did not specify how this should be done: it thought almost entirely about support for the algebraic calculations to be done within `Reduce` and almost totally ignored issues of interfacing with the rest of the world. Thus there is some delicacy in this area! To be specific, it can not be guarantee that the CSL and PSL versions of `Reduce` will support the same functionality. It is also easy to stray into areas where Windows, Unix/Linux/BSD and Macintosh systems will have different and incompatible conventions, so producing robust portable code involved significant care.

In general the facilities to interact with the underlying computer exist at the level of symbolic (or system-level) coding and will not be immediately available to those who write pure algebraic mode code, so access to them is perhaps one of the benefits of fully understanding this document.

It is not possible to document or describe how to solve all possible problems here, and so general suggestions will be made. The first is that both CSL and PSL do provide their own set of functions for operating and file system interfacing. For PSL these can be found documented in the manual, and for CSL they are probably best identified by searching the source code in `csl/cslbase` and reading the comments. Many of those functions that do exit in CSL are broadly modelled after Common Lisp ones, so checking Common Lisp documentation may provide leads. You will certainly find that there are functions there to manipulate the current directory and to enumerate and remove files. In some cases a long-stop is provided by the function `system`. This accepts a string and attempts to interpret it as a command to be obeyed as if it has been presented from a terminal.

There are examples of some of this sort of activity, including the creation of strings to pass to `system`, in the source code in `packages/crack/crutil.red` where the code to delete individual or multiple files may be particularly illuminating. Note the customisation where different recipes are involved in Windows and as between the CSL and PSL worlds. Code that you write that you are certain will never be used by anybody else and will always be run on just one computer can avoid some of that complication – but of course we would want to encourage authors of interesting code to prepare it in a form that could me merged into the main `Reduce` distribution and would run everywhere.

If you have sufficiently strong need for some further control of your computer from within `Reduce` and have checked carefully and it can not be achieved using the current system then the Open Source nature of the code means that you could consider adding extra features in to either CSL or PSL (or both) to help you achieve your aims. If your addition was liable to be of significant use to others it may be accepted by the project to be added to the main versions. Full guidance about such steps would call for a companion to this document – perhaps called "Inside CSL" or "Inside PSL"! But perhaps especially in the CSL case you might find that reading the existing C code and patterning your new code on some existing function should not be too challenging. Over time there have been a number of experiments of this style made by the current developers where things are not necessarily tidily enough finished off for general use but could form a basis for future projects. I will particularly note that there have been experiments in parallelism using both PVM and MPI and links down to

numerical libraries. It could be that "Here be Dragons", but to a computer scientist that is a good challenge not a reason to flee.

Bibliography

[1] F. Brackx, D. Constales, Computer Algebra With Lisp and Reduce: An Introduction to Computer-Aided Pure Mathematics, Springer 1991.

[2] Anthony C. Hearn, REDUCE: User's and Contributed Packages Manual, Version 3.8 Santa Monica, CA and Codemist Ltd. July 2003. The file `manual.pdf` that should be present as part of a distribution of `Reduce` is an updated version of this.

[3] Malcolm A. H. MacCallum and Francis J. Wright, Algebraic Computing with REDUCE Lecture Notes from the First Brazilian School on Computer Algebra Vol. 1, Series Editors: Marcelo J. Rebougas and Waldir L. Roque, Clarendon Press, ISBN 978-0-19-853443-3 (1991).

[4] Jed Marti, RLISP '88: An Evolutionary Approach to Program Design and Reuse, World Scientific, 1993.

[5] Hubert Melenk, `Reduce` Symbolic Mode primer.

[6] Gerhard Rayna, Reduce: Software for Algebraic Computation, Springer 1987.

[7] Sourceforge, the official hosting website where `Reduce` can be found at `http://reduce-algebra.sourceforge.net/`.

[8] Subversion, software and documentation: `http://subversion.apache.org/`, graphical user interfaces: `http://en.wikipedia.org/wiki/Comparison_of_Subversion_clients`.

[9] J. Marti, A.C. Hearn, M.L. Griss, C. Griss: The Standard Lisp Report.