Abstract

This volume consists of three separate reports:

**Part One - Project Report**

The project report is written as a chronological account of my work on porting APL to the Macintosh. It explains the main difficulties which I encountered during the work but does not specify how to use APL nor any details about how MacAPL works internally.

**Part Two - User Manual**

The MacAPL user manual is basically a port of the APL\11 manual, adapted to the new Macintosh version. It gives an introduction to using MacAPL but does not attempt to explain the APL language. References to books about APL are given in the Project Report.

**Part Three - Technical Manual**

This is a short introduction to the inner workings of MacAPL. It explains the basic operation and mentions some of the data structures used. Hopefully it will be of some use to those who wish to modify and enhance MacAPL.
Part One

Project Report

MacAPL
# Contents

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Subject</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Introduction</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>A brief overview of APL</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>The initial attempts</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>Issues in developing large applications</td>
<td>14</td>
</tr>
<tr>
<td>5</td>
<td>Macintosh I/O</td>
<td>16</td>
</tr>
<tr>
<td>6</td>
<td>Workspaces and User-defined functions</td>
<td>19</td>
</tr>
<tr>
<td>7</td>
<td>Cleaning up the loose ends</td>
<td>22</td>
</tr>
<tr>
<td>8</td>
<td>Ideas for the future</td>
<td>23</td>
</tr>
<tr>
<td>9</td>
<td>Conclusion</td>
<td>24</td>
</tr>
</tbody>
</table>

Bibliography                                                                 26

Appendix A - the MacAPL keyboard                                              27

Appendix B - some example functions                                           28
Section 1 - Introduction

1.1 Problem definition

The "official" specification for this project is as follows:

APL for the Macintosh - APL is an unusual programming language in its use of a specific surface notation including a specific character set. It was designed in the early sixties and is still used today, particularly in some scientific and mathematical application environments. However, because of the unusual character set, and the difficulty of providing it on standard terminals, the use of APL has never been widespread. But the possibilities of the bit mapped screen, and user-defined fonts, as well as the reasonably powerful processor, make the Macintosh seem a potentially good host for APL. There is a simple APL interpreter written in C for Unix that is available, and the project would be to portage that to the Macintosh and make it seem at home. Both the portage and the provision of a good Macintosh interface would both be considered important.

This was tendered by Robert Biddle, who confesses to have fallen in love with both APL, Unix and the Mac. Thus, a possible collaboration seemed to offer the best of all worlds. Not having the time to do it himself, he had to resort to the standard practice: offering the idea as an Honours project!

I was attracted to the project for similar reasons. i.e. interest in Unix, C and Macs. Working in this environment promised an interesting year. APL looked tempting and learning a new language (even one as weird as APL!) may be useful in the future.
1.2 Philosophy of the project

The main idea behind this project was to experiment with the Macintosh user interface. As this style of computing had only recently become available to the masses it presented a relatively new programming environment. APL was a language which could benefit greatly from this (see Section 2).

The general method of programming on such machines as the Mac is also different to traditional techniques. On the Mac, the user can do almost anything anytime, such as click the mouse, type at the keyboard or select an item from a pull-down menu. This is totally different from conventional programs which say things like read, whereupon the user MUST type something before the program will continue. All this is geared towards making computers easier for people to use. After all, computers are only tools and if they are not easy to use then no-one will use them!

The fact that programs on the Mac are (supposed to be) easier to use means that they are usually harder to write! Instead of the usual methods of program control, a program is written in an event-oriented manner. i.e. the main loop looks for an event such as a mouse press or a key press, and then acts upon it. The main thing to note is that the program must be able to handle any event at any time.

Another feature which makes the Mac easier to use is the availability of a large body of subroutines called the Toolbox. This provides, among other things, a set of standard procedures to handle the user interface. Because these are fast and available, many programs use them and therefore present the same interface to the user. As a result, once a user has mastered one, he can use them all.

Thus, programming the Mac is certainly a challenge and can result in programs which are very easy to use.
1.3 Implementation alternatives

Apart from porting the entire APL interpreter, there was a second alternative. This involved using the Mac as a front-end to APL\11 running on the Vax. Thus, the project would be more of a specialised terminal emulator, providing (at least) an APL font and other "user-friendly" features. This would allow more time to be spent on the user interface, because the complete port would require a non-trivial effort just to get the basics working. On the other hand, not every Mac owner has access to a Vax running APL. Thus, during the initial deliberations, I preferred the idea of a complete port.

1.4 Choice of tools

As APL\11 was written in C, this was to be the language used for the Macintosh version. There seemed no good reason to change which would justify the considerable problem of converting it to another. At Canterbury, we had two alternatives - the Megamax C development system or SUMacC, a Macintosh cross-compiler running on the Vax. After talking with a few users of both, it was decided to use Megamax, mainly to reduce dependence on our Vax, which was often unavailable. After several months' work, I think that it was the right choice, although there were many problems along the way (see section 4). Along with the Megamax system of compiler, editor and linker, I had access to other useful tools:

- **Resource Editor**: this allowed editing of icons and fonts, which proved invaluable.
- **Switcher**: the Switcher allows several programs to be in memory at the same time, using memory partitioning. Switching between them requires only a single keystroke.
Section 2 - A brief overview of APL

2.1 Birth of a notation

APL (A Programming Language) was initially developed by Ken Iverson around 1957 for use as a mathematical notation. It was based upon standard algebra, but extended this with new symbols representing more complex operations. e.g. + and - indicated addition and subtraction as usual, while $\phi$ was used to rotate or reverse a vector. Even more powerful operators were included, such as $\phi$ and $\Phi$ for sorting, and $\mathbb{E}$ representing matrix inversion. Iverson, 1962, introduced his new notation to the world in the book *A Programming Language*.

2.2 Adapting APL for the computer

Because of the concise nature of the APL notation, it appealed to many. For instance, algorithms discussed in a paper could be included in context using APL, instead of being relegated to the Appendices when using, say, FORTRAN. Soon people hit upon the idea of using APL to communicate with a computer. As Falkov and Iverson wrote in [FI73] *our work in 1964 had been focused on the language as a tool for communication among people... we never doubted that the same characteristics which made the language good for this purpose would make it good for communicating with a machine*.

The first computer implementation was performed by Breed and Abrams in 1964 on an IBM 7090. However, it was not until 1966, when APL became available on the IBM System 360, that its popularity started to spread. So much so that 3 years later IBM started supporting it officially on the 360 and (later) 370 series mainframes. Soon after, APL was adopted by many companies, such as Xerox, DEC and Burroughs.
2.3 APL\11

The APL interpreter on the Department's Unix system is APL\11, written originally by Ken Thompson at Bell Labs. It later found its way to Yale and then to the Electrical Engineering department at Purdue University, where is currently maintained by J.D. Bruner and A.P. Reeves. Written in C, it runs on either DEC PDP-11's and Vax-11's using Unix.

This is the APL interpreter which I ported to the Mac. A home on the Mac would be beneficial in many respects, the most important being (I believe) the facility of a specialised APL font. On most APL systems, many APL characters, such as I and $\pi$, are not present. These must be generated by overstriking 2 characters which are available. e.g. $\infty$ is produced by keying \<overstrike> O. Worst still, on standard ASCII terminals, when this overstriking is used, only the last character is displayed on the terminal, even though the overstrike has been noted internally. This is at least cryptic and at worst unreadable and is one of the factors that has limited the acceptance of APL. I feel that it is preferable to have all APL symbols available with a single key-stroke, without ANY overstriking. Using a custom-designed font on the Mac allows this.

A few weeks after the start of this project, I was told of a new version of APL for the Macintosh - PortAPL. My entire project had just been made redundant! PortAPL for the Mac is based upon and compatible with similar interpreters for the IBM PC and DEC Vax. Looking on the bright side, whatever the benefits of this new system, it seemed far too expensive (at US$275) compared with a forthcoming (free!) MacAPL.
2.4 the main features of APL

One way of looking at APL is to consider it as a kind of enormous pocket calculator. It is used interactively and an expression typed in will be evaluated and the result printed on the screen. This is similar to how Lisp operates. However, the main feature of APL that it uses a special character set to implement some very powerful operators as primitives. These inbuilt operators are divided into two basic types - monadic and dyadic. This simply means that they take either one or two arguments. To simplify matters there is a common syntax for all operators:

monadic operator : monadicop arg

dyadic operator : arg1 dyadicop arg2

On top of this is the facility for users to define their own functions, as in other languages. However, these functions are of similar form to the built-in operators, in that they are either monadic, dyadic or niladic. The first two are as above, while the third takes no arguments. Thus, a niladic function may be executed by:

niladicfun

Functions may return a result if needed. The common format for built-in operators and user-defined functions means that the by defining new functions, APL may effectively be extended and new functions can be regarded as being part of the language.

The other main feature of APL is the use of workspaces. Instead of source files to be compiled, APL keeps all its functions and variables together in a workspace. These may be saved and restored later to return APL to a previous state.
Section 3 - The Initial Attempts

3.1 Early scuffles with the compiler

To start, all 200k of source was downloaded to the Mac using Macput. This is a simple but effective communication program which allows data to be transferred from a Vax to a Macintosh. In order to decide whether to do the port or the terminal emulator, I decided to feed all the sources to the Megamax compiler and watch. If there were too many problems, I would be forced to abandon the idea of a stand-alone MacAPL.

Out of 35 source files, 16 compiled with little or no change. However, there were a large number of type errors present - more than expected. On close inspection, the Vax makefile compiled sources with warning diagnostics suppressed. Turning them on again resulted in 51 warnings! Obviously, a large number of "dirty tricks" had been employed with no consideration for portability. Unfortunately (fortunately?) the Megamax compiler was less impressed with these and produced errors instead of warnings. Indeed, as the project progressed, it became apparent that the APL\11 had been written using an intimate knowledge of the host. e.g. (on the Vax) integer variables known to be 4 bytes long were used to traverse chains of character pointers, also 4 bytes long. In another case, a string of pointers, floats and integers were passed as a single integer to another routine! The same pseudo-integer was then passed to other routines which then unpacked their parameters and continued on their way. As the Megamax compiler handles function arguments differently, all this hairy code had to be modified.

Two sources which caused some trouble were one concerning file buffering and another which implemented a memory management scheme similar to that in [KR78]. As these features were not essential, I left them out and let the Mac take care of memory management. If needed, they could easily be reintroduced later.

Although Megamax C provided some Unix calls (such as malloc, creat and printf) there were a few chunks of Unix absent which were needed by MacAPL. The main
ones were `qsort`, a sorting routine, and `ecvt`, a routine for formatted output. The sources of these were downloaded from the Vax and compiled without too much trouble.

The other main problem in compiling sources was that in APL\11, an integer variable was often used to store characters and other things as well. The Vax compiler let this pass while Megamax didn't. A few extra variables of the right type soon fixed most occurrences of this problem, while the more cryptic ones were commented out initially.

Finally, all sources compiled and I thought that MacAPL would soon be up and running after a simple loading and linking procedure. However, it was not so simple . . .

3.2 Loading and Linking

As soon as the second object file was loaded, the linker complained about duplicate definitions and variables. This was due to the format of the header file, which is interpreted differently by the Vax and Megamax compilers. Whereas the Vax compiler allowed this header file to be included with each source, Megamax requires that variables be declared only once in the entire set of source files. This meant that the header file `apl.h` had to be split into `apl.h`, containing all type and constant definitions and `apl.vars`, which contained all the variable declarations. This latter file was then included in only one source file, while `apl.h` was included in all sources. This satisfied the linker.

The next problem was that on the Mac, the largest possible piece of code is only 32 kbytes. This ridiculous restriction is something to do with the fact that relative addressing is done via a signed 16-bit quantity ($2^{15}=32768$). I had visions of dreadful thrashing, with pieces of code being swapped in and out at just the wrong times. However, overlays can be used to allow larger programs. The operation of these is as follows: whenever a routine in another overlay is needed, that overlay is loaded
from disk into memory. If there is not enough space, another currently resident overlay is **swapped out** to make room. On the other hand, if there is enough free memory, all overlays may be simultaneously resident in memory. In this case, as soon as they are all loaded, no further disk access is needed. In the current situation, the total objects easily fit into a Fat Mac (512 KB) or a Mac Plus (1 MB). This means that overlays will be almost completely transparent.

After loading all objects without complaint, the next grumble was that some names (such as sin, tan and sqrt) were undefined. Obviously, Megamax's support of Unix did not go this far! Including the Megamax maths header file soon solved this problem.

Finally, the linker completed its job successfully and MacAPL was ready to roll. It was roughly 95 KB, and consequently could easily reside in memory on anything but the now defunct 128K Mac. The size of MacAPL compared favourably, for instance MacPaint and MacWrite which are both about 70KB each. APL\11 on the Vax is about 93 KB.

### 3.3 the Trial Run

I double-clicked the MacAPL icon and waited in hope. the Megamax default window entitled `stdout` appeared, along with APL\11's initial greeting. An underscore marked the cursor, waiting for input. I typed a simple APL expression and it was echoed back exactly. More input produced the same result. Not much APL, more of a bloated Mac copy of Unix's `cat`!

To find out where the problem lay, I would usually go to a debugger. However, on the Megamax system there was no such item! This proved to be an enormous time waster throughout the project. I therefore resorted to strategically placed debugging statements to trace the error to the parser. This had been generated using Yacc and Lex on Unix and, not knowing any better, I went in head first and looked around. After getting thoroughly confused by numerous variables beginning with `yy`, I decided that the best thing to do was to read the Unix documentation, by Johnson and Lesk and
Schmidt. This improved my understanding immeasurably and I returned to the parser source again.

Turning on Yacc debugging allowed me to compare the parsing of a simple line with that generated by the Vax version. I soon discovered that there were no elements correctly recognised, and concluded that the problem might be in the size of the variables used. Sure enough, pointers to integers (4 bytes long) were being used to navigate tables of shorts (2 bytes). Incrementing such a pointer resulted in missing every second short out - not surprisingly, this produced incorrect results. Changing these tables of shorts to integers fixed the problem.

This version of MacAPL was able to return some correct results although there were a few slight problems . . . it had great difficulty in printing digits above 5, no monadic operators worked and after a screen of interaction, it fell over! And these were only the main problems! Also, the use of standard input and output meant that the mouse was ignored and thus so were all menus. Although at last I had an APL interpreter going on the Mac, there was clearly a great deal of work to be done.
Section 4 - Issues in developing large Mac applications

4.1 File organisation

By this stage, I had discovered some of the difficulties associated with software development on the Mac. Most were due to the size of the application. The problem was to organise over 200 KB of sources and a similar quantity of objects, plus an editor, compiler and linker to leave enough room for temporary files and the final application itself. On top of this were numerous Megamax header files which also had to be accommodated.

After some early combinations were tried, I decided to use the following configuration:

- **800 KB internal drive**: headers, sources, objects + application
- **600 KB RAM disk**: Finder, System, editor, compiler, linker
  Megamax library

This obviously required a Mac Plus, both for the memory and the disk capacity. Luckily, I was usually able to get access to a Plus whenever I wanted.

The only other feasible possibility was to put all the tools in the internal drive and the rest on the RAM disk. This would mean that edits, compiles and links would all be lightning fast. However, the main disadvantage with this is the lack of security. If the system crashed (and it often did!) I would lose all changes of the current session. This would naturally be unacceptable.

When I was initially compiling and editing, I used the Switcher. This piece of software allowed the compiler and editor to be in memory simultaneously and
switching between them took only a single key-stroke. This was very convenient and saved having to exit back to the Finder between each compile / edit cycle.

However after the first complete linking, a different setup was needed. Megamax includes a simple batch language with which I was able to write a make facility, similar to that on Unix. This was most useful. The batch language was based upon C, with some extra functions dealing with mounting volumes, querying the user and similar things.

4.2 Hardware Limitations

After working with this arrangement for a long time, several disadvantages showed up. As all the object files were on floppy disk, the linking process was very slow and consumed 2 or 3 minutes. Whenever a header file was altered, all sources had to be recompiled - this took close to 15 minutes. The obvious solution to this would be a hard disk drive. Apart from speeding up compiling and linking there is another benefit. Every time the Mac fell over, rebooting would be much faster. Currently, I have to boot a RAM disk and replace the startup-disk with my working disk - again, a slow procedure.

Another moderately useful feature would be a reasonable operating system which didn't fall over every time anything went wrong - who would use Unix if it had to be rebooted after a segment violation error! The single user nature of the Mac along with a lack of memory management hardware is probably at the root of this problem. In fact installing Unix underneath the Mac's user-interface might not be a bad idea!

4.3 the Software Environment

The next gripe is with the software system in use. No debugger resulted in numerous statements printing variables everywhere. These extra edit-compile-link cycles were very time consuming.
Section 5 - Macintosh I/O

5.1 low-level Macintosh I/O

After getting some parts of the APL interpreter going, the next thing to do was to replace the standard Unix I/O with event-oriented Macintosh I/O. Before even trying to use the Mac toolbox, I firstly read about 500 pages of the phonebook-sized *Inside Macintosh*. This was, I think, the first book to give comprehensive details of the routines which were used to program the Mac. Organised a little like an encyclopaedia this hefty volume covered almost all of the Mac's insides.

After reading *Inside Mac*, I had a good idea of the methods used to program the Mac. I then ordered the two volume *Macintosh Revealed* from the States and read both of these. These were more readable than the above and included a large example (a simple text editor) to show how a complete application is put together. At this stage I had enough confidence to write some actual code.

There are at least two possible ways of managing I/O on the Mac. The first involved using the TextEdit routines in the Mac toolbox. This option would take care of such things as backspacing and cut and paste editing. Also, after the screen is obscured by another (desk accessory) window, TextEdit will automatically repaint it when the APL window becomes active again. Another advantage is that the script of the entire APL session is available and can be scrolled through to review previous results. Yet another advantage is that I could make use of SnazzyWindows, a library of routines written by Greg Ewing, which looks after most window and menu management. The main disadvantage was that as the entire session was being saved, this would use up large amounts of memory. Periodically pruning this off would result in annoying scrolling of the window.

The second alternative would be to perform I/O as done in Macscreen. This involves the Toolbox routine *drawchar* and requires much more low-level screen management than with TextEdit. However, only the current line need be stored in
memory, so no space would be used up with previous results. If the session's interactions are to be remembered, a script file (see Section 7) could be used. Another reason to use Macscreen-style I/O is that drawing in bold or another text style is easier than with TextEdit.

I decided to use the latter method, mainly for the space saved by not remembering the entire session. After completing the Mac I/O work, I think that maybe it wasn't the correct choice. TextEdit would have allowed cut and paste editing which would have been useful. By the time I gave my project seminar, I had completed a simple I/O system for the Mac, including backspace but not tabs. However, it was not yet connected to the interpreter! Thus, I had two separate pieces of program and had to knit them together.

This proved to be more difficult than expected. APL\11 included a complete rewrite of I/O and I had to replace this at a low level (putchar) with Mac I/O. Unfortunately some of the output routines didn't print anything at all! I traced them right down to the putchar level and still could not discover the problem. After a few weeks I gave up and wrote my own replacements. Although they are smaller and faster, the precision of printing is fixed and not dependent upon the APL system variable \texttt{digits}, as it should be. This is something which can be worked on in the next version.

5.2 a font for MacAPL

It was during this work that I designed a simple APL font. I had sent off for and received one developed at Victoria University (Wellington) but this was only 9 point and not very good either! My effort began with Geneva-12; I then changed the upper-case characters to produce APL symbols. Although most early APL systems used only upper case, nowadays lower case is almost universal. This is why I decided to keep lower case for identifiers. As there were not enough keys, I had to use two option combinations as well. Initially I had hoped to have APL symbols on option combinations, but the standard Mac keyboard driver uses some combinations for
accenting e.g. option-e and option-o. This would have messed up my character set. Therefore, I decided to revert to using mainly upper case characters, as did APL\11.

However, I did manage to fit all APL characters into the font and thus overstriking became unnecessary and was abolished. This also simplified the input routine to MacAPL. With the new character set looking at least recognisable, I then changed the parser to acknowledge them.

### 5.3 Declaration problems

The next problem was that no monadic operators worked. Again, I used cunningly placed print statements to track down the problem. This turned out to be that, although presented with the correct data and calculating the correct result, it was never returned correctly! How could this happen when it worked on the Vax? After substantial scanning of the source, I noticed that the routines were declared in some places to return an integer, whereas they really returned a float. Although these were the same size on the Vax, on the Mac they were not. Changing these declarations to float soon fixed the problem. This was more evidence that APL\11 was not written with portability in mind! Reduction and inner product also needed this kind of attention to get them working.

Another set of problems were typified by one related to the APL operator \iota\ (iota). This operator is supposed to generate a list of integers from the origin (0 or 1) up to its parameter. It worked for some numbers, but not for others. It turned out that somewhere along the line a cast was being used to convert a float to an integer. A quick example showed that while the Vax truncated, the Mac rounded the float to the nearest integer. A simple fix solved this problem and similar ones.

Random numbers required some alteration, due to the different format of the random number generators on the two machines in question.
Section 6 - Workspaces and User-defined Functions

6.1 workspaces

Now that most APL operators worked, it was time to try saving and loading workspaces. Fortunately, MMC's Unix support included most file routines, such as open, creat and read. This meant that most of the code should run unmodified. A few system calls were not implemented, such as fstat (which returns file information) and dup (which copies file descriptors); however, this was no great problem. Instead of duping a file descriptor and closing it later, I just used the existing file descriptor and left it open for use later. There have been no problems with this so far.

File I/O was accomplished with the use of a union of an integer and an array of characters of the same size. This allowed both characters and integers to be moved in the same way (whether it is a good idea or not is another matter!). Due to the differing sizes of types on the Mac and Vax, a little fiddling was required to get this going. Also, no fstat meant that (to start with) the date and time were not saved in the workspace file. This is not too important, as these may be viewed from the Finder under view by name.

6.2 Design decisions

At this stage, I had about half of an APL system - the inbuilt operators. The second part, which builds on earlier work, is the facility of user-defined functions. Although most APL implementations have an in-built editor, APL\11 used a standard Unix editor such as vi. Also, ASCII files containing functions could be accessed using the )read and )write system commands.
As the Mac sports many text editors, I decided to start with the desk accessory Mockwrite and read functions into the workspace. Later, if necessary, a simple editor could be built into MacAPL. Another option is to use a word-processor such as MacWrite, and MacAPL together with the Switcher.

After rewriting a routine to accept a line of file input, I was able to read in functions with little modification to the original source.

### 6.2 Compiling functions

The next step was not so easy. When executing normal input, APL\11 firstly reads a line, compiles it into object code of a virtual APL machine and then executes this code. The process of compilation is performed by the Yacc-generated parser as mentioned in Section 3. However, when executing functions, the mode of operation is slightly different. If the function has not yet been compiled, it is compiled to produce object code together with extra code for function entry and exit. This extra code deals mainly with allocating and releasing space for local variables. When the function is compiled, the resulting object code is executed line by line.

As the function compiling routine used many coersions and dubious assignments it took a fair while to get going. Most helpful was comparing code from a faulty MacAPL with the correct results generated by APL\11 on the Vax. This was accomplished using the `code` APL system command and was used throughout the project.

### 6.3 Executing functions

The routines for executing functions needed similar close inspection to weed out the type size dependencies. After this was done, user-defined functions seemed to work correctly. One interesting point is that instead of stopping at an run-time error, functions just continue executing! This results in no suspended functions and no use for the `si` command! As far as I can see, the problem is something to do with the
lack of Unix-style interrupts. Some scrutiny is definitely required - later, because there were more important things to do first. At least I now had a useable APL system on the Macintosh.
Section 7 - Cleaning up

Although most of the main features were now working, I still had a fair amount to do - the project deadline was only two weeks away! However, most problems were small and didn't take too long:

Quad I/O - as all Unix I/O had been replaced, these routines did not work at first. Replacing the standard I/O soon had them going.

Menus - I connected up several menus, mainly one which offered some commonly used APL system commands. Another set of menus allowed direct entry of APL symbols without using the keyboard.

File types - to enable a workspace to be double-clicked from the Finder, I changed the file type and creator fields to WKSP and MAPL.

Special icons - I designed two (unoriginal) icons for MacAPL and its workspace files.

Also, an initial APL tab was added, script files were fixed, a beep operator was added, an about...menu item was constructed and a beep was added to the error routine.
Section 9 - Ideas for the future

Eventually I ran out of time and naturally still had much I would like to have done.

**bugs to fix**

There are bugs in the domino and format operators which need to be fixed.

**features to add**

Firstly, rewrite Mac output, including the `ecvt` routine. This would allow the format operator to be implemented. Adding some more I-beam functions, such as time and date, would be useful, as would low-level file I/O. Including windows and support for the Macintosh Toolbox (e.g. Quickdraw).

Emphasise the APL operators and reserved words (outline or shadow?). This would distinguish user-defined variables and function names from APL functions such as `\nl`. Include a simple editor.

Add an interactive debugging environment. This would involve two windows - one output window and another showing the source text. The currently executing line (symbol?) would be highlighted to show the exact state of the calculation.

Use resources more. I did not use them initially, as there was not enough room for the resource compiler and source on my system. The advantage of using resources is that often changes can be made without recompiling the source code of an application. e.g. changing languages or menu items. However, I have tried to make my menus reasonably easy to understand.
Section 9 - Conclusion

9.1 Points to watch

After working at MacAPL for almost a year, I have noticed some general ideas to bear in mind when attempting such a project. These have turned up repeatedly and I would have saved considerable time if they had been listed initially! They are:

1. size of types on source and destination machines. If a language such as C is used then many dirty tricks used for efficiency will depend upon the size of types on both machines.

2. action of casts. Automatic type conversions (coersions) are sometimes used instead of the above. Find out whether casting to an integer results in truncating or rounding; whether casting involves any change in the actual data as well as just its type.

3. strictness of type-checking. This often varies with different compilers and strict type-checking will show up many potential problems at compile time instead of (more expensively!) at runtime. With respect to C, possibly use Lint or a similar program.

4. If possible, use a hard disk!
9.2 the Current state of MacAPL

At October 7th, the official project deadline, I currently have a reasonably useful APL system for the Macintosh. However, many problems still exist:

1. domino, format and laminate are not implemented.

2. function tracing and labels do not work.

3. TEXT files containing functions often end with a funny square symbol. However, this symbol is not interpreted by MacAPL and so is irrelevant.

4. on the Mac, for some reason workspaces cannot be loaded or saved! I cannot work out why, but a Mac expert may be able to find the problem in no time.

5. on the Mac Plus, a few font characters do not come out correctly. Again, I do not know why.

These should not be too difficult to fix given more time.

9.3 Availability of MacAPL for further development

I shall release MacAPL into the Public Domain so that anyone with a Mac can get a free introduction to APL. I shall also be offering the sources to interested parties so that MacAPL can be maintained and improved as users add to it. This is similar to the Vax version of APL\11 which is currently being "looked after" at Purdue University in America.
Bibliography

Bruner, J
Reeves, A *Unix APL\11 Users' manual* unpublished

Chernicoff, S *Macintosh Revealed* Hayden Book Company. 1985

Ewing, G *Macintosh Snazzy Windows Package* unpublished

Iverson, K *A Programming Language* J. Wiley and sons, Inc. 1962

Johnson, S *Yacc - yet another compiler compiler* Unix Programmers' manual Bell labs 1978

Kernighan, B
Ritchie, D *the C programming language* Prentice and Hall. 1978

Lesk, M
Schmidt *Lex - a lexical analyser generator* Unix Programmers' manual Bell labs 1978

Pommier, S *An Introduction to APL* Cambridge University Press. 1983
This keyboard shows the characters generated when the shift key is depressed. To produce A and Y, use option-a and option-v.
Appendix B - MacAPL System Commands

The following is a complete list of APL system commands.

)`clear` This command is used to completely initialize an APL workspace. Sometimes, it is desirable to completely erase everything and this command serves that purpose. To let you know that everything has been erased, APL will set the window's title to "Untitled".

)`cls` This command will clear the screen and return the cursor to the top of the screen.

)`code fun` This command causes APL to print the compiled internal code for the function `fun`. This is intended for APL system development and not for general use.

)`copy file` This command instructs APL to locate `file` and load it into its internal workspace, similar to the `)load` command (see below). The difference between `)load` and `)copy` is that `)load` will replace the current internal workspace with the one being read in, while `)copy` merges the current internal workspace with the one being read in. Functions and variables which are loaded from the file take precedence over functions and variables of the same name existing already in the internal workspace.
This command is used to specify to APL how many digits are to be displayed when a number is printed in floating-point or exponential format. By default, APL will print 9 digits. You may specify any number between 1 and 19 for the number of digits (n). APL will respond with the number of digits it was using. Unfortunately, MacAPL currently ignores the value of )digits.

This command invokes debug mode. In this mode every action which APL takes is logged on the terminal. This mode is excellent for generating reams of hopelessly cryptic output and exists only to facilitate APL development. It is not intended for general use. Debug mode can be turned off by issuing the )debug system command a second time.

This command deletes a list of files. The names of the files to be deleted should be separated by spaces or tabs. The files may be APL workspaces, TEXT files, or any other type of file.

This command is handy when it is desirable to get rid of parts of a workspace without using )clear to eliminate all of it. A list of function and variable names (separated by spaces or tabs) may be specified. The named functions and variables will be deleted from the internal workspace. The remainder of the workspace will not be affected.
This command causes APL to list the names of all of the functions which are defined in its internal workspace.

This command causes APL to print out the function named `fun`. This is very handy for looking at a function without having to use the editor - especially when an error has occurred and you wish to look at a function without disturbing the state indicator.

This command is used to instruct APL to load `file` as a workspace. After the file is loaded, APL's internal workspace will be the same as it was when the workspace file was saved with `)`save, and that previous APL session may be resumed. If the workspace file exists and is successfully loaded, APL will print the time and date that the workspace was last saved.

This command terminates the APL session and exits to the finder.

This command is used to change the `origin`. By default, the origin is 1. The `origin` is the starting index for arrays. For example, if the origin is 0, then the first element of a 3-element array A is A[0]. If the origin is 5, the first element will be A[5]. Although standard APL permits only 0 or 1 for the origin, MacAPL allows any integer value. APL will answer with previous origin.
This command causes APL to print the contents of the workspace in a readable format. Nonscalar variables are displayed along with their dimensions; functions are displayed as via the \texttt{)list} system command.

\texttt{)read fun} 

At times it is desirable to read a function which is stored in an TEXT file into the internal workspace. The \texttt{)read} command causes APL to read the file named \texttt{fun} into the workspace as a function. Note that \texttt{)read} and \texttt{)load} (see above) are not the same thing. \texttt{)load} reads a complete workspace into APL from a workspace-format file while \texttt{)read} reads a function from an TEXT file and adds it to the current workspace.

\texttt{)reset} 

This command is used to reset the state indicator. All suspended functions are reset; the state indicator will be cleared. APL returns to the global level.

\texttt{)save file} 

This command causes APL to write its internal workspace into \texttt{file}. This allows the current session to be resumed at a later time.

\texttt{)si} 

This command is useful when something goes wrong. When an error occurs, the function that was executing is suspended. The \texttt{)si} command causes APL to print a traceback of the suspended functions. Each function is listed, in the reverse order that it was called. The current
line number for each function is also printed. Functions followed by an asterisk ("**") were suspended due to an error; these were called by functions listed on the following lines whose names are not followed by an asterisk.

)`vars` This command causes APL to list the names of all of the variables which are defined in its internal workspace.

)`write fun` This command is the complement of )read (see above). It takes the function fun from the current workspace and writes it to an TEXT file named fun. This is useful for writing functions which will be )read later into other workspaces.

)`ydebug` This command toggles the state of Yacc-based debugging statements. This is even more cryptic for most users than )debug!
Appendix C - MacAPL Quad Functions

The following quad functions are available in MacAPL:

- `cr 'name'` The result of `cr` is a character array containing the function whose name is passed as an argument.

- `fx newfn` The contents of the character array specified as an argument are fixed as an APL function.

- `nc 'arg'` This function can be used to determine what type of variable an APL symbol is. The returned value will be:
  
  0 - symbol is undefined
  2 - symbol is a label or variable
  3 - symbol is a function name

- `nl arg` The argument should be a scalar or vector with components 1, 2, or 3. This function returns a two-dimensional character array containing the names of all items whose types are specified in the vector (same type definitions as for `nc` above). The ordering of names in the matrix is fortuitous.

- `beep n` This beeps for a time proportional to n and returns 0. If n is too large (100+?) then MacAPL crashes!
Appendix D - MacAPL I-Beam Functions

The following monadic i-beam functions are available:

26 This i-beam returns the line number in the function currently being executed. Thus, if it is used in line 3 of a function, it will return 3.

27 This i-beam returns a vector of the line numbers of pending functions (functions which called the current function and are waiting for its completion).

29 This i-beam returns the current origin, set by the )origin system command.

30 This i-beam returns the current width, as set by the )width system command.

31 This i-beam returns the current number of digits to be displayed, as set by the )digits system command.

95 This i-beam causes APL to print out its namelist table. It is intended for system development of APL.

96 This i-beam causes APL to dump its stack on the terminal. It is intended for system development of APL, and is probably useful only in generating a big messy display.
This i-beam returns the total number of elements on APL's internal stack. It is intended for system development and debugging of APL itself.

The following dyadic i-beam functions are implemented. They may be subject to future change. The function is specified by the right argument, the left argument is a parameter to that function. The available i-beams are:

- **29**
  This i-beam may be used to set the origin to any permitted value. The left argument specifies the new origin, and the previous origin is returned.

- **30**
  This i-beam may be used to set the terminal width to any permitted value. The left argument specifies the new width, and the previous width is returned.

- **31**
  This i-beam may be used to set the number of digits displayed to any permitted value. The left argument specifies the new number of digits, and the previous value is returned.
Part Two

Users' Manual

MacAPL
0 - Introduction

This Users' Manual is almost a complete copy of the APL\11 Users' Manual, which was written by John D. Bruner and A. P. Reeves of the School of Electrical Engineering at Purdue University, West Lafayette, IN 47907, U.S.A. I decided to do this for many reasons. Firstly, for the user familiar with APL\11, who will feel at home with this manual. Secondly the APL\11 manual gives a good introduction to this dialect of APL. Lastly because MacAPL is based upon APL\11 it seems logical to provide a similar manual.

This manual is intended to serve as a guide to the use of MacAPL. It is not intended to be a reference manual on the APL language; rather, it describes the local implementation of APL. Portions of this manual are intended for the new APL user; persons who have used other APL systems will probably want to skim most sections and concentrate on the sections concerning error messages, editing functions, the state indicator, and the appendices. For these users, differences between MacAPL and standard APL are noted where appropriate.

1 - What's a Workspace?

One of the most important concepts pertaining to the use of APL is the idea of a workspace. Simply stated, a workspace is a collection of information (including both functions and data) to be processed. Programmers accustomed to compiler languages such as FORTRAN may find this concept somewhat unusual so let's explore it a bit further.

Perhaps the best way to start describing a workspace is to define another term - file. A file is also a collection of information. On many systems, however, a
file usually contains only one type of information; e.g. a C program or data for a FORTRAN program. A standard program, perhaps written in FORTRAN, may access data in the file by opening the file and reading it. The file may also be created written, or removed.

APL differs from FORTRAN in that, while it is running, all of the information it needs is internal to itself. That is, all of the variables that the functions will require are already available. As functions are written, they are added to this internal storehouse of information. As the functions are run, and data is generated, the data is added to this internal storehouse. APL provides the facility to save part or all of this information in a special format (workspace or load format) in a file. This file can only be used by APL in later runs (it should not, for instance, be printed on the line-printer).

In addition, APL is capable of working with more conventional files, which are in character format and can be edited and displayed outside of APL. These files are in TEXT format.

2 - Getting Into and Out of APL

2.1 How do I start APL?

The answer in this case is extremely simple - double-click the MacAPL icon from the Finder. APL will print a title identifying itself. This is APL’s prompt, which means that it is expecting you to type something. Initially, you do not have anything in your workspace (no functions or variables have been defined). Equally important to getting into APL is getting out. To terminate APL, either use the )off APL system command or use Quit (command-Q) from the File menu. This will return to the Finder.
2.2 What are all those funny characters?

Part of the power of APL comes from the fact that it has so many primitive operators, usually referred to as functions in the APL literature. That is, many common functions, from addition and subtraction, to trigonometric functions, and to matrix operations, can be specified with a couple of characters. As a result, the character set for APL does look a little strange. Basically, there are two major types of primitive functions. The most common functions, such as the arithmetic operators, etc., are represented by a single character. For example, the symbol + represents addition. Because there are only so many keys on the keyboard, the other operators in APL, for example, matrix inversion, are generated by using the shift key or, occasionally, the option key to modify a key stroke. Lowercase letters are used to form variable and function names. Special symbols, such as +, and the upper-case letters are used to form the APL operators. To help the new APL user most APL symbols can be produced from the right-hand pull-down menus.

3 - APL System Commands

There are a number of system commands in APL. These commands are used to control your current session. System commands may be typed at any time when APL is prompting for input. (however, they may not be included in APL functions). As stated previously, when APL is waiting for input from the keyboard, it will type eight spaces. There are two types of system commands, those which request information and those which specify action by APL. All system commands are preceeded by a right parenthesis. Commands which specify action are composed of the command name and some other value. For instance, to set the output width (number of characters per line that APL will print) to 64, the command is:
A final warning - the command name and the right parenthesis must be together:

) width 64

is invalid and will give an error message. The available commands are described fully in Appendix B.

Many commonly used system commands are available from the system pull-down menu. These perform the same functions as their corresponding system commands:

workspace : load
save
dump
same as )prws
clear

function : read
write

clear screen

quit
same as )off
4 - Automatic Workspace Loading

MacAPL has the useful feature that you can automatically load a workspace file into the internal workspace when you start APL. Instead of double-clicking the MacAPL icon, an APL workspace can be automatically loaded by double-clicking a workspace icon from within the Finder. Alternatively, a workspace may be selected and then opened from the File menu.

5 - Editing Functions

Function editing on MacAPL differs from that on other APL systems. The `del` character is not used. Instead, an external editor such as MacWrite is used. The function is written and saved in a file with the same name. Then it can be read into MacAPL. If a function in a workspace needs to be edited, `)write` will write it out to an external TEXT file which may then be edited.

There are two main configurations for editing. One involves using the desk accessory MockWrite. This is convenient but does not allow the APL font (called GenevAPL) to be used. A better solution is to use the Switcher with MacAPL and an old memory-based copy of MacWrite. This allows the use of the APL font while writing functions. However, it must be remembered to save the MacWrite file TEXT only, as MacAPL does not understand MacWrite files.
6 - The State Indicator

Currently, MacAPL does not handle errors in functions very well and so this section is no use at all. It has been included to be of use when this bug has been fixed.

APL is a recursive language which supports local variables. Therefore, it is essential that it maintain some sort of information concerning its current state of execution, local variables, and status of functions. Part of this information - the current status of active functions, is called the state indicator. Users of other APL systems should note that the state indicator as implemented on MacAPL is similar in concept to the standard APL state indicator, but does not function entirely the same way. In particular, the system command )siv is not supported, and the state indicator is cleared in a different manner. Each time a function begins execution, APL keeps track of the current line number in that function. When the current function calls another function, the line number where the request was made is recorded in the state indicator. When a function terminates abnormally, it is said to be suspended. This means that APL stops execution of the function, but retains all information about its current state of execution. In some cases, the problem can be corrected and the function restarted where it left off. This discussion will not include that technique. In any event, when the function is suspended, APL will print a traceback automatically, perhaps such as:

```
varx: used before set
at fn3[5]
from fn2[12]
from main[4]
```
In this case, the error (undefined variable) occurred in the function \texttt{fn3} at line 5. \texttt{fn3} was called by the function \texttt{fn2} at line 12, which in turn was called by the function \texttt{main} at line 14. The \texttt{)si} command produces a similar traceback. For the above example, the traceback could be:

\begin{verbatim}
   fn3[5] *
   fn2[12]
   main[4]
\end{verbatim}

This shows that the function \texttt{fn3} was suspended due to error, and that \texttt{fn2} and \texttt{main} are waiting for \texttt{fn3} and \texttt{fn2}, respectively.

When an error occurs and a function is suspended, all of that function's variables are accessible from the keyboard. Thus, you can examine the current variables to help determine the problem. As stated above, in some cases it may be possible to change some variables and resume execution of the suspended function. However, in general, after you examine the variables you will probably want to restart from the beginning. In order to do this, type the \texttt{)reset} command. This causes APL to clear the state indicator, resetting suspended functions and releasing local variables.

If you do not use \texttt{)reset}, you will find two things. First, if you get another error, the state indicator may look like:

\begin{verbatim}
   fn3[3] *
   fn2[12]
   main[4]
   fn3[5] *
   fn2[12]
   main[4]
\end{verbatim}
This shows that fn3 was suspended due to an error at line 3, and it traces execution back to the start of main. Then, it shows that in a previous run fn3 was suspended due to an error at line 5, and it traces execution back to the start of main. You can still use the )reset command to get back to a clear state indicator. The other effect which you will notice if you do not )reset the state indicator is that you will be unable to edit some functions. If a function is suspended (it appears in the state indicator), it cannot be changed, because it is possible that the error condition may be corrected and execution resumed. Therefore, if you try to edit a function which appears in the state indicator, you will get the error message:

```
si damage -- type ')reset'
```

To edit the function, clear the state indicator with )reset first, and then you may proceed with an editing command.

A useful technique for finding errors is to look at the function which blew up and try retyping the line which failed, a little bit at a time. This often helps isolate the problem. Generally, you want to do this before the state indicator is )reset so that the state of APL is exactly the same as it was when the error was detected. Since you cannot run the editor to list the function without typing )reset, you will probably want to use the )list command to look at the bad function.

As an example, suppose your friend Chris (male or female, take your pick) gave you the following program to compute the reciprocal of the numbers from 1 to x:

```
r { recip x
[1] r ← ÷ 1 x
```
You are using origin 0, and your attempt to run the function yields the following:

recip 10
recip domain
at recip[1]

It looks like Chris gave you a bum steer. You decide to try to debug the function. You can't use the editor to look at it with )editf unless you type )reset first. You don't want to )reset, because that would take you back to the beginning and you would lose any temporary variables (in this case, just x). Thus, you can use the )list command:

)list recip

y { recip x
[1] y ← ÷ 1x

Now, you can experiment with the function by typing the commands yourself:

x
10
1x
0 1 2 3 4 5 6 7 8 9

At this point, you immediately spot the trouble - APL was trying to take the reciprocal of 0. The function recip was written assuming that the origin was 1, but Chris did not tell you and you were using 0 instead.
Thus, you merely say:

)reset

(now that you've found the problem, you want to reset APL to clear the state indicator) and

)origin 1
was 0

 recip
1.00000000e+00 5.00000000e-01 3.33333333e-01 2.50000000e-01
2.00000000e-01 1.66666667e-01 1.42857143e-01 1.25000000e-01
1.11111111e-01 1.00000000e-01
7 - Special Functions

7.1 Quad Functions

Quad functions are special operators that perform tasks which are closer to the system level, but are accessible within APL functions. There are many available quad functions in MacAPL, but only one is probably of interest to you at this time. The interested user is referred to Appendix C for a complete list.

□cr xxx This quad function is a standard APL quad function. The argument xxx should be a character vector containing the name of an APL function which exists in the internal workspace. □cr will return a character array containing the function definition in ASCII. This provides a very convenient way to get the listing of an APL function. For example, if you want to define the array fnlist to be the function definition of the function xyz, you can say:

\[
\text{fnlist} \leftarrow \squarecr \text{'xyz'}
\]

or, equivalently,

\[
\text{fnname} \leftarrow \text{'xyz'}
\]

\[
\text{fnlist} \leftarrow \squarecr \text{fnname}
\]
Another possible use of □cr is to list a function on the terminal without having to use )editf. (As we discussed earlier, when an error occurs you cannot edit a suspended function without using )reset first, and )reset gets rid of the current state of APL.) However, the )list system command is far more efficient for this purpose and is therefore strongly recommended.

7.2 I-beam functions

The I-beam function in MacAPL is a method of obtaining system parameters. The available I-beams are listed in Appendix D for the interested user; they are not generally required for the average APL session.
8 - What are those funny messages?

You probably figured that at some point in this manual you'd see something about error messages. Unfortunately, errors do occur, and some of APL's error messages can really throw you. Users of other APL systems have little advantage over APL novices in this area, for UNIX APL error messages are, in general, non-standard. To make matters worse, APL's status messages can be cryptic at times. Thus, in this section we'll explore some of the more common status and error messages. First, some of APL's less obvious and relatively common abbreviations are:

- **asgn** assignment
- **cat** catenation
- **crc** circle
- **dfmt** dyadic format
- **dibm** dyadic i-beam
- **exec** execute
- **fn** function
- **imm** immediate
- **iprod** inner product
- **mdom** monadic domino
- **menc** monadic encode
- **miot** monadic iota
- **oprod** outer product
- **red** reduction
- **var** variable
There are three major types of error messages which may be generated as a result of an APL session. The error message may be the result of a fatal error in APL which has caused APL to abort, the result of some internal problem in APL, or an error produced by incorrect use of APL.

8.1 Non-fatal Error Messages

Usually, APL can detect serious errors before they become fatal. When this occurs, it will issue an error message but will not terminate. You can then take appropriate action. Unfortunately, unlike most APL implementations, UNIX APL does not report the position on the line where the error occurred. However, a traceback from the function in which the error occurred is printed. (See "The State Indicator" for more information on function tracebacks and error recovery.)

As an example of this type of error, it is possible to completely fill up the internal workspace. Usually APL will not attempt to get more internal workspace memory if it already has the maximum permissible amount. It will then inform you:

workspace exceeded

Other possible messages of a similar nature are:

directory botch
save botch
elid botch
stack overflow
pop botch
In general, if an error message such as this occurs, you should save your internal workspace in a different file than you started from (in case it may have been corrupted or otherwise damaged by the error in APL), and then should exit APL and restart. You can then determine whether or not anything happened to your workspace and can recover any losses from the original workspace file. These type of errors generally occur as the result of some other error, such as a function which calls itself indefinitely or declaring an array which is too large for the workspace.

By far majority of non-fatal error messages produced by APL are not serious but simply indicate some error in the definition or execution of a function. These errors fall into some common types:

**conformability**

The variable which you are passing to a function or operator does not conform. That is, it is not the proper shape (e.g. array has the wrong number of dimensions or a scalar is used instead of an array, etc.) For example, trying to add a 2 x 2 array to a 2-element vector will produce this error.

**botch**

This can occur for several reasons. Attempting to read a non-existent element of an array will produce a getdat botch. (Storing into a non-existent variable or array element will produce a putdat botch.)

**domain**

A domain error means that somehow the variable passed to a function or operator cannot be used. For example, attempting to take the logarithm of a negative number will give a log domain error message.
### Index

A bad index was specified. A quick example of this error is to have origin = 1 and attempt to access A[0]. Attempting to concatenate two arrays along a non-existent dimension will produce a **cat index** error message.

### Syntax

This error message is probably painfully familiar - it indicates that the indicated APL expression was not syntactically correct. If this error occurs during the processing of a function, the traceback will show which line was incorrect.

### Rank

This type of error indicates that the argument to an operator or function has the wrong number of dimensions. As an example, if you attempt to declare an array with more than 8 dimensions, you will get a **max rank** error message.

Some other common error messages do not fall into any of the above classes. These include:

<table>
<thead>
<tr>
<th><strong>used before set</strong></th>
<th>This message means that a variable was referenced which had not yet been defined. An easy example:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$y ← 2$</td>
</tr>
<tr>
<td></td>
<td>$z ← x + y$</td>
</tr>
<tr>
<td></td>
<td>$x$: used before set</td>
</tr>
</tbody>
</table>

Page 16
no fn result
This error occurs when an attempt is made to use the returned value of a function which doesn't return a value. As an example, if the function \texttt{xyz} returns no value then the following sequence will result in an error:

\begin{verbatim}
z ← xyz
no fn result
\end{verbatim}

si damage
This error message (discussed earlier) usually occurs when an attempt is made to edit a function which is suspended due to an error. Type }\texttt{reset} and then you may edit the function.

line too long
the input line is limited to 128 characters. Watch out!

can't open file
This error message occurs when a non-existent file is used for a }\texttt{load} , }\texttt{read} , }\texttt{save} , etc.

not a fn
In most cases, an attempt to use a function which has not been defined, or which has been defined as a variable, will result in a syntax error. There are some special circumstances in which this error will slip past the ordinary error-checking code. In the event this occurs, the message name: not a fn will be printed.
8.2 Fatal Error Messages

Sometimes, something really bad happens and the Mac bomb alert comes up. There's nothing you can do - you must restart or resume to the Finder.

9 - A startup function in APL

APL has a startup facility which allows automatic execution of an APL expression when a workspace is loaded. If you define the variable \( \text{D1x} \) to be some APL expression and then save the workspace, when you next \text{load} the workspace, APL will automatically read and execute \( \text{D1x} \). For example, if you have the function \text{hello} which looks like this:

\[
\text{hello}
\]

\[[1]\] 'hi there, my friend'

\[[2]\] 'welcome to apl'

and you want APL to perform \text{hello} when you load the workspace file \text{hi}, you can say:

\[
\text{L1x ← 'hello'}
\]

\[
\text{)save hi}
\]

\[
\text{)off}
\]

Then, if you run APL and load \text{hi}, the following will appear:

\[
\text{hi there, my friend}
\]

\[
\text{welcome to apl}
\]
10 - MacAPL Limitations

MacAPL does have some limitations. Tracing currently does not work, nor do functions using labels. It is hoped that these will be fixed in later releases. Some operators have not been implemented (for example, laminate is not supported). Neither format or domino are implemented. Inner and outer product will not work with character data.

Functions which are created with Dfx should not have the same name as existing variables; occasionally APL gets confused and mistakenly zaps a newly-fixed function. Functions containing overstrikes that were created by Dfx cannot be properly edited. The only quad variable which has been implemented is Dlx, and it is always a global variable.

11 - Differences between MacAPL and APL

As MacAL was derived from APL, it seems logical and useful to compare the two:

MacAPL has: menus, an APL font, )cls, )ydebug and Dbeep.

MacAPL doesn't have: function tracing or labels, domino and format, )edit and )editf, )continue and other system commands, quad file I/O, many l-beams or overstriking.
Appendix A - The MacAPL ASCII Character Set

Note that the ASCII character set for APL uses both upper-case and lower-case and two option combinations.

<table>
<thead>
<tr>
<th>Character</th>
<th>Shift Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>a-z</td>
<td></td>
<td>letter</td>
</tr>
<tr>
<td>0-9</td>
<td></td>
<td>digit</td>
</tr>
<tr>
<td>-</td>
<td></td>
<td>negative sign</td>
</tr>
<tr>
<td>'...</td>
<td></td>
<td>string</td>
</tr>
<tr>
<td>Æ</td>
<td>shift-c</td>
<td>comment</td>
</tr>
<tr>
<td>( )</td>
<td></td>
<td>indexing</td>
</tr>
<tr>
<td>[ ; ]</td>
<td></td>
<td>quad</td>
</tr>
<tr>
<td>□</td>
<td>shift-l</td>
<td>quote quad</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+</td>
<td></td>
<td>add</td>
</tr>
<tr>
<td>-</td>
<td></td>
<td>sub</td>
</tr>
<tr>
<td>×</td>
<td>shift-x</td>
<td>mult</td>
</tr>
<tr>
<td>÷</td>
<td>shift-5</td>
<td>div</td>
</tr>
<tr>
<td>[ ]</td>
<td></td>
<td>mod</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L</td>
<td>shift-d</td>
<td>min</td>
</tr>
<tr>
<td>F</td>
<td>shift-s</td>
<td>max</td>
</tr>
<tr>
<td>*</td>
<td>shift-p</td>
<td>pwr</td>
</tr>
<tr>
<td>⊗</td>
<td>shift-o</td>
<td>circle funct</td>
</tr>
<tr>
<td>o</td>
<td></td>
<td>pi times</td>
</tr>
</tbody>
</table>

Page 20
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Shift Key</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>!</code></td>
<td><code>shift-a</code></td>
<td>combinatorial factorial</td>
</tr>
<tr>
<td><code>∧</code></td>
<td><code>shift-v</code></td>
<td>and</td>
</tr>
<tr>
<td><code>∨</code></td>
<td></td>
<td>or</td>
</tr>
<tr>
<td><code>∧</code></td>
<td><code>option-a</code></td>
<td>nand</td>
</tr>
<tr>
<td><code>∀</code></td>
<td><code>option-v</code></td>
<td>nor</td>
</tr>
<tr>
<td><code>&lt;</code></td>
<td></td>
<td>lt</td>
</tr>
<tr>
<td><code>&gt;</code></td>
<td></td>
<td>gt</td>
</tr>
<tr>
<td><code>≤</code></td>
<td><code>shift-4</code></td>
<td>le</td>
</tr>
<tr>
<td><code>≥</code></td>
<td><code>shift-7</code></td>
<td>ge</td>
</tr>
<tr>
<td><code>=</code></td>
<td></td>
<td>eq</td>
</tr>
<tr>
<td><code>±</code></td>
<td><code>shift-3</code></td>
<td>ne</td>
</tr>
<tr>
<td><code>~</code></td>
<td></td>
<td>not</td>
</tr>
<tr>
<td><code>?</code></td>
<td></td>
<td>deal random number</td>
</tr>
<tr>
<td><code>ρ</code></td>
<td><code>shift-r</code></td>
<td>rho</td>
</tr>
<tr>
<td><code>ι</code></td>
<td><code>shift-i</code></td>
<td>iota</td>
</tr>
<tr>
<td><code>ε</code></td>
<td><code>shift-e</code></td>
<td>epsilon</td>
</tr>
<tr>
<td><code>Τ</code></td>
<td><code>shift-n</code></td>
<td>encode</td>
</tr>
<tr>
<td><code>⊥</code></td>
<td><code>shift-b</code></td>
<td>decode</td>
</tr>
<tr>
<td><code>⊗</code></td>
<td><code>shift-6</code></td>
<td>transpose transpose</td>
</tr>
<tr>
<td><code>,</code></td>
<td></td>
<td>catenate ravel</td>
</tr>
<tr>
<td><code>↑</code></td>
<td><code>shift-y</code></td>
<td>take</td>
</tr>
<tr>
<td><code>↓</code></td>
<td><code>shift-u</code></td>
<td>drop</td>
</tr>
<tr>
<td><code>&lt;</code></td>
<td><code>shift-[</code></td>
<td>assign</td>
</tr>
<tr>
<td><code>&gt;</code></td>
<td><code>shift-]</code></td>
<td>goto</td>
</tr>
<tr>
<td><code>Ι</code></td>
<td><code>shift-'</code></td>
<td>i-beam</td>
</tr>
</tbody>
</table>
Appendix A - the MacAPL keyboard

This keyboard shows the characters generated when the shift key is depressed. To produce A and V, use option-a and option-v.
Appendix B - some example functions

A - a chessboard - this function can be read from the read menu or the )read system command. Then, typing cb will produce a simple chessboard in about half a minute. As can be seen from the listing below, APL is certainly very powerful! (this listing can be generated by using the )list system command).

b←cb;temp
b←' *[1+(2|L(18)+.2)^2|L(7126)+3]
b[1;]+b[18;]+b[1;]+b[26;]←'o'
temp←beep 1

B - a simple histogram - by giving this function a list (array) of numbers, a simple histogram will be drawn. This kind of APL one-liner is very popular and this is only a simple one!

z←hist vect
z←(' ',•')[1+(41/vect)^.≤vect]

C decimal to hex - the following function will convert its argument (a hex string) to its decimal equivalent.

r ← dtogh
r ← '0123456789abcdef'[1+(5p16)Th]
D - a graphing function - this function takes two arguments - the first is an array of x values and the second is a string representing a function.

\[ z \leftarrow x \text{ graph } y ; a \]
\[ a \leftarrow (\phi x)^o \cdot 1 \text{ y} \]
\[ z \leftarrow (2 \times px) \cdot 0 \text{ \( \cdot \) \( \cdot \)} [a + 1 + 2 \times 0 = (\phi x)^o \cdot xx] \]

There are a large number of functions other than these, such as while and for (structured programming), htod and dtoh (base conversion between hex and decimal) and area (for calculating numerical integrations).

Apart from these functions, there are two workspaces which will give a demonstration of some of these functions. They are called graphics and ws. After loading either of these, they will take over, using \( \Box \)l.
Part Three

Technical Manual

MacAPL
MacAPL Technical Manual

Ed Wilkinson

1986
1 - Introduction

This technical manual is a brief guide to the inner workings of MacAPL. It is directed at anyone who is attempting to modify MacAPL. Its existence relates to the fact that, with little documentation, I had great trouble in understanding the program initially. To help navigate the sources, Appendix A includes a cross reference of function definitions.

The working disk I used contained about 36 source files, various header files and a few oddments, such as a make batch program. I have also included two help files which were part of the earlier APL\11 on the Vax. These may be of some use.

Thus manual does not document how to use MacAPL - for that see the User manual. If anybody would like a copy of either MacAPL (as a user) or the complete work disk (as a developer) the address to write to is:

Department of Computer Science,
University of Canterbury,
Private Bag,
Christchurch,
New Zealand.

Please include a disk and enough money to cover postage.
2 - Program structure

The source file which is "in charge" is mo.c - this contains all routines which are specifically related to the Mac. Initially, I was going to have several of these but instead just tagged everything onto the end of mo.c. This should explain its strange structure and contents.

As with most Mac applications, MacAPL does some initialising in main() before going into an (infinite) event-driven loop. Quitting is done by calling exittoshell() at some stage.

I/O is done via drawchar() and not TextEdit. This was an early decision and possibly a wrong one. Keystrokes are stored in a buffer prline indexed by col. Normally when a return character is entered, processline() (in a0.c) is called. This in turn calls compile() (in lex.c) to compile the input line into APL code. If the input compiles without errors then it is executed by execute() in a1.c. This completes a cycle and MacAPL returns to the main loop again.
3 - Data structures

There are three main data structures in MacAPL - nlist, item and si. Information about these can be found in a0.c and apl.h but I will give a brief overview here.

3.1 item

struct item {
  char rank;
  char type;
  int size;
  int index;
  double *datap;
  int dim[MRANK];
}

This is the data structure which contains APL data. Usually its type (item.type) is either DA (for numbers) or CH (for character data). A list of all different types is given in apl.h. The operand stack is an array of pointers to items.
3.2 nlist

struct nlist {
    char  use;
    char  type;
    struct *itemp;
    char  *namep;
    int   label;
}

An nlist is used to store variables, functions and file descriptors. An array of these
is used a symbol table which also stores the associated data via nlist.itemp.

3.3 si

struct si {
    int    suspended;
    struct si *sip;
    struct nlist *np;
    int    funlc;
    struct item **oldsp;
    char   *oldpcp;
    jmp_buf env;
};

The struct si is used to implement the state indicator, i.e. recursive functions calls.
When a function is executed, a new struct si is generated and linked to any previous
ones, thus providing a run-time stack for APL. However, MacAPL contains a bug which means that if an error occurs during the execution of a function, it does not suspend, but continues executing instead! This is a nuisance and makes debugging very difficult! However, it should be fixable.

Many other data structures are declared in apl.h, including chrstrct and uci. This first is used to allow multiple indirections to be represented in a single expression (see ai.c in funcomp()), while the second is used during workspace load and save. The struct thread is used to store various workspace values, e.g. the current value of origin digits and width.

3 - Compiling

The parser for MacAPL was originally developed using Yacc and Lex on Unix. These are tools for automatically generating language parsers and save a great deal of time-consuming and error-prone work.

To compile an input line, Yacc is given the first token found by lex and does the appropriate shifts and/or reduces. Whenever it needs a new token, it calls lex to return the next input token. The APL parser accepts an LALR(1) grammar. This roughly means that by looking 1 token ahead, it can decide what kind of input it is looking at, e.g. a monadic or dyadic function. For anybody who intends to modify the APL language, the Yacc source and output are included in the appendices.

As the input line is being compiled, a line of object code is being built. This will include interpreter op-codes (defined in apl.h) such as IMMED, QUOT along with pointers to related data. If the entire line compiles correctly then the resulting line is passed back ready for execution.
5 - Execution of APL object code

The object code to be executed is arranged in postfix form, e.g. CONST CONST ADD CONST MULT PRINT EOL. This allows it to be executed in sequential order using an operand stack. Before an operator is executed, the corresponding argument(s) are fetched (a0.c) from the symbol table and pushed onto the stack. Then the operator performs its function upon these stack-bound values. After popping them off it may optionally push a result onto the stack.

Errors are handled using the setjump/longjump calls in C. Each time throughout the main loop a setjump is performed (a macro called setexit() is used, actually). This saves the stack environment at that point. Then, whenever an error occurs, a longjump is performed to return to the main loop again. This is efficient and usually works well.

6 - Functions

User-defined functions are accessed either via the function read and write menu items or using )read and )write. Source code for functions in the workspace is stored in a file called aplws. This file exists only when MacAPL is running. When a function is read into the workspace, the source is appended to aplws and the offset is recorded in the function's nlist.label field. This allows easy access to the source. If the function is changed and read in again, the new source is still appended to aplws and the function's nlist.label is updated to point to the new source.

Compiling functions takes three passes and produces code for each line along with startup and shutdown code. This deals with local variable management and the (optional) returning of a function result. Unfortunately, there is a bug somewhere which means that labels cannot be used in functions. I think that the problem is related to
csize() and catcode() in a0.c. Also, function tracing doesn't work and therefore has been commented out. Yet another problem exists with respect to functions - when an error occurs, instead of suspending the current function, MacAPL just continues going! This is not much use for debugging and means that )si and )reset are redundant. The problem here lies with either error() (in a2.c) or ex_fun() (in ai.c).

To allow the use of any editor, functions are simply stored in TEXT files.

7 - workspaces file structure

Routines for saving and loading workspaces live in aj.c. These all use the Megamax Unix style file I/O, mainly because it works! For best results, however, they should be replaced by the Mac file routines. This would allow access to files on different disks - currently, this is not possible.

As mentioned before, the struct uci is used for transferring data. Initially, the struct thread is written to the workspace file. Then, for each entry in the symbol table, the corresponding variable value or function source is written. This is done by firstly writing the length of data to come followed by the data. The file creator and type are always MAPL and WKSP for two reasons. Firstly, to allow a distinctive icon and secondly, to allow double-clicking of a workspace from the finder.

8 - Resources

APL.r contains definitions for such things as FREF and BNDL resources, along with the about... alert. I have also added (using the resource editor) the APL font (font number 2 - same as New York) and icons so that MacAPL can be copied o.k.
Appendix A - a cross-reference

access() a0.c ex_ddom() a8.c
alpha(s) lex.c ex_ddyad() a1.c
asym(arg) gamma.c ex_deal() af.c
badfnsv() ai.c ex_dfmt() al.c
bidx() a0.c ex_dibm() a9.c
cat0() a5.c ex_diot() ag.c
cat1() a5.c ex_div() a0.c
catchsigs() a0.c ex_drho() a3.c
catcode() a0.c ex_drop() ab.c
checksp() a0.c ex_dscal() a1.c
clear() aj.c ex_dtrn() ab.c
collapse() a0.c ex_dup() az.c
com1() ad.c ex_elid() a4.c
comk() ad.c ex_eps() ag.c
compile() lex.c ex_eq() a0.c
concat() a0.c ex_exd() ad.c
convrt() aj.c ex_exd0() ad.c
copy() a0.c ex_exdk() ad.c
csize() a0.c ex_exec() az.c
dealloc() a0.c ex_exit() az.c
digit(s) lex.c ex_exp() a0.c
dstack() a2.c ex_fac() a0.c
dump() ao.c ex_fdef() ai.c
<table>
<thead>
<tr>
<th>Function</th>
<th>File</th>
<th>Function</th>
<th>File</th>
</tr>
</thead>
<tbody>
<tr>
<td>dupdat()</td>
<td>a0.c</td>
<td>ex_float()</td>
<td>az.c</td>
</tr>
<tr>
<td>epr0()</td>
<td>a2.c</td>
<td>ex_floor()</td>
<td>ao.c</td>
</tr>
<tr>
<td>epr1()</td>
<td>a2.c</td>
<td>ex_fork()</td>
<td>az.c</td>
</tr>
<tr>
<td>epr2()</td>
<td>a2.c</td>
<td>ex_fun()</td>
<td>ai.c</td>
</tr>
<tr>
<td>equal()</td>
<td>lex.c</td>
<td>ex_gdd()</td>
<td>aa.c</td>
</tr>
<tr>
<td>erase()</td>
<td>a0.c</td>
<td>ex_gddk()</td>
<td>aa.c</td>
</tr>
<tr>
<td>error()</td>
<td>a2.c</td>
<td>ex_gdu()</td>
<td>aa.c</td>
</tr>
<tr>
<td>evLix()</td>
<td>ah.c</td>
<td>ex_gduk()</td>
<td>aa.c</td>
</tr>
<tr>
<td>ex_abs()</td>
<td>ao.c</td>
<td>ex_ge()</td>
<td>ao.c</td>
</tr>
<tr>
<td>ex_add()</td>
<td>ao.c</td>
<td>ex_gt()</td>
<td>ao.c</td>
</tr>
<tr>
<td>ex_and()</td>
<td>ao.c</td>
<td>ex_hprint()</td>
<td>a2.c</td>
</tr>
<tr>
<td>ex_ap()</td>
<td>a5.c</td>
<td>ex_ibr()</td>
<td>ai.c</td>
</tr>
<tr>
<td>ex_arg1()</td>
<td>a5.c</td>
<td>ex_ibr0()</td>
<td>ai.c</td>
</tr>
<tr>
<td>ex_arg2()</td>
<td>a5.c</td>
<td>ex_immed()</td>
<td>ah.c</td>
</tr>
<tr>
<td>ex_asgn()</td>
<td>a4.c</td>
<td>ex_index()</td>
<td>a4.c</td>
</tr>
<tr>
<td>ex_auto()</td>
<td>a4.c</td>
<td>ex_iprod()</td>
<td>a7.c</td>
</tr>
<tr>
<td>ex_base()</td>
<td>a4.c</td>
<td>ex_kill()</td>
<td>az.c</td>
</tr>
<tr>
<td>ex_botch()</td>
<td>a4.c</td>
<td>ex_label()</td>
<td>a4.c</td>
</tr>
<tr>
<td>ex_br()</td>
<td>a4.c</td>
<td>ex_le()</td>
<td>ao.c</td>
</tr>
<tr>
<td>ex_br0()</td>
<td>a4.c</td>
<td>ex_list()</td>
<td>al.c</td>
</tr>
<tr>
<td>ex_cat()</td>
<td>a5.c</td>
<td>ex_log()</td>
<td>a0.c</td>
</tr>
<tr>
<td>ex_catk()</td>
<td>a5.c</td>
<td>ex_loge()</td>
<td>a0.c</td>
</tr>
<tr>
<td>ex_cdyad()</td>
<td>a1.c</td>
<td>ex_lit()</td>
<td>a0.c</td>
</tr>
<tr>
<td>ex ceil()</td>
<td>a0.c</td>
<td>ex_max()</td>
<td>a0.c</td>
</tr>
<tr>
<td>ex_chdir()</td>
<td>az.c</td>
<td>ex_mdom()</td>
<td>a8.c</td>
</tr>
<tr>
<td>ex_cir()</td>
<td>a0.c</td>
<td>ex_menc()</td>
<td>al.c</td>
</tr>
<tr>
<td>ex_close()</td>
<td>az.c</td>
<td>ex_meps()</td>
<td>al.c</td>
</tr>
<tr>
<td>ex_com()</td>
<td>ad.c</td>
<td>ex_mfmt()</td>
<td>al.c</td>
</tr>
<tr>
<td>Function</td>
<td>Source File</td>
<td>Function</td>
<td>Source File</td>
</tr>
<tr>
<td>------------</td>
<td>-------------</td>
<td>------------</td>
<td>-------------</td>
</tr>
<tr>
<td>ex_com0()</td>
<td>ad.c</td>
<td>ex_mibm()</td>
<td>a9.c</td>
</tr>
<tr>
<td>ex_comb()</td>
<td>a0.c</td>
<td>ex_min()</td>
<td>a0.c</td>
</tr>
<tr>
<td>ex_comk()</td>
<td>ad.c</td>
<td>ex_minus()</td>
<td>ao.c</td>
</tr>
<tr>
<td>ex_creat()</td>
<td>az.c</td>
<td>ex_miot()</td>
<td>a3.c</td>
</tr>
<tr>
<td>ex_crp()</td>
<td>al.c</td>
<td>ex_mod()</td>
<td>a0.c</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ex_mrho()</td>
<td>a3.c</td>
</tr>
<tr>
<td>fetch()</td>
<td>a0.c</td>
<td>ex_mtrn()</td>
<td>ab.c</td>
</tr>
<tr>
<td>fetch1()</td>
<td>a0.c</td>
<td>ex_mul()</td>
<td>a0.c</td>
</tr>
<tr>
<td>fetch2()</td>
<td>a0.c</td>
<td>ex_nand()</td>
<td>ao.c</td>
</tr>
<tr>
<td>fix()</td>
<td>a0.c</td>
<td>ex_nc()</td>
<td>al.c</td>
</tr>
<tr>
<td>flop()</td>
<td>lex.c</td>
<td>ex_ne()</td>
<td>ao.c</td>
</tr>
<tr>
<td>forloop()</td>
<td>a0.c</td>
<td>ex_nilret()</td>
<td>ai.c</td>
</tr>
<tr>
<td>funcomp()</td>
<td>ai.c</td>
<td>ex_nl()</td>
<td>al.c</td>
</tr>
<tr>
<td>fundef()</td>
<td>ai.c</td>
<td>ex_nor()</td>
<td>ao.c</td>
</tr>
<tr>
<td>fundedit()</td>
<td>ai.c</td>
<td>ex_not()</td>
<td>ao.c</td>
</tr>
<tr>
<td>funread()</td>
<td>ai.c</td>
<td>ex_open()</td>
<td>az.c</td>
</tr>
<tr>
<td>funwrite()</td>
<td>ai.c</td>
<td>ex_oprod()</td>
<td>a7.c</td>
</tr>
<tr>
<td>fuzz()</td>
<td>a0.c</td>
<td>ex_or()</td>
<td>ao.c</td>
</tr>
<tr>
<td>gamma(arg)</td>
<td>gamma.c</td>
<td>ex_pi()</td>
<td>ao.c</td>
</tr>
<tr>
<td>gd0()</td>
<td>aa.c</td>
<td>ex_pipe()</td>
<td>az.c</td>
</tr>
<tr>
<td>gd1()</td>
<td>aa.c</td>
<td>ex_plot()</td>
<td>ak.c</td>
</tr>
<tr>
<td>gdd()</td>
<td>aa.c</td>
<td>ex_plus()</td>
<td>ao.c</td>
</tr>
<tr>
<td>gdu()</td>
<td>aa.c</td>
<td>ex_print()</td>
<td>a2.c</td>
</tr>
<tr>
<td>genlab()</td>
<td>lex.c</td>
<td>ex_prws()</td>
<td>al.c</td>
</tr>
<tr>
<td>getchar()</td>
<td>a2.c</td>
<td>ex_pwr()</td>
<td>a0.c</td>
</tr>
<tr>
<td>getdat()</td>
<td>a0.c</td>
<td>ex_rand()</td>
<td>a1.c</td>
</tr>
<tr>
<td>getnam()</td>
<td>lex.c</td>
<td>ex_rav()</td>
<td>a5.c</td>
</tr>
<tr>
<td>getnm()</td>
<td>aj.c</td>
<td>ex_ravk()</td>
<td>a5.c</td>
</tr>
<tr>
<td>Function</td>
<td>Source File</td>
<td>Library Function</td>
<td>Source File</td>
</tr>
<tr>
<td>-------------------</td>
<td>-------------</td>
<td>------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>getnum()</td>
<td>lex.c</td>
<td>ex_rd()</td>
<td>az.c</td>
</tr>
<tr>
<td>getquad()</td>
<td>lex.c</td>
<td>ex_read()</td>
<td>az.c</td>
</tr>
<tr>
<td>index1()</td>
<td>a4.c</td>
<td>ex_recip()</td>
<td>ao.c</td>
</tr>
<tr>
<td>intrprws()</td>
<td>a0.c</td>
<td>ex_red()</td>
<td>a6.c</td>
</tr>
<tr>
<td>intr()</td>
<td>a0.c</td>
<td>ex_red0()</td>
<td>a6.c</td>
</tr>
<tr>
<td>invert()</td>
<td>lex.c</td>
<td>ex_redk()</td>
<td>a6.c</td>
</tr>
<tr>
<td>iodone()</td>
<td>az.c</td>
<td>ex_rep()</td>
<td>ae.c</td>
</tr>
<tr>
<td>iofname()</td>
<td>az.c</td>
<td>ex_rest()</td>
<td>ai.c</td>
</tr>
<tr>
<td>ipr1()</td>
<td>a7.c</td>
<td>ex_rev()</td>
<td>ab.c</td>
</tr>
<tr>
<td>isize()</td>
<td>aj.c</td>
<td>ex_rev0()</td>
<td>ab.c</td>
</tr>
<tr>
<td>line(x,y)</td>
<td>ak.c</td>
<td>ex_revk()</td>
<td>ab.c</td>
</tr>
<tr>
<td>listdir()</td>
<td>aj.c</td>
<td>ex_rot()</td>
<td>ac.c</td>
</tr>
<tr>
<td>lsize()</td>
<td>aj.c</td>
<td>ex_rot0()</td>
<td>ac.c</td>
</tr>
<tr>
<td>lsq()</td>
<td>a8.c</td>
<td>ex_rotk()</td>
<td>ac.c</td>
</tr>
<tr>
<td>mainloop()</td>
<td>a0.c</td>
<td>ex_run()</td>
<td>al.c</td>
</tr>
<tr>
<td>map()</td>
<td>ab.c</td>
<td>ex_scan()</td>
<td>ak.c</td>
</tr>
<tr>
<td>map1()</td>
<td>ab.c</td>
<td>ex_scn0()</td>
<td>ak.c</td>
</tr>
<tr>
<td>menc1()</td>
<td>al.c</td>
<td>ex_scnk()</td>
<td>ak.c</td>
</tr>
<tr>
<td>move(x,y)</td>
<td>ak.c</td>
<td>ex_seek()</td>
<td>az.c</td>
</tr>
<tr>
<td>name()</td>
<td>lex.c</td>
<td>ex_sgn()</td>
<td>ao.c</td>
</tr>
<tr>
<td>neg(arg)</td>
<td>gamma.c</td>
<td>ex_shell()</td>
<td>ai.c</td>
</tr>
<tr>
<td>newdat()</td>
<td>a0.c</td>
<td>ex_signl()</td>
<td>az.c</td>
</tr>
<tr>
<td>nlook()</td>
<td>a0.c</td>
<td>ex_sub()</td>
<td>a0.c</td>
</tr>
<tr>
<td>nsave()</td>
<td>aj.c</td>
<td>ex_take()</td>
<td>ab.c</td>
</tr>
<tr>
<td>opn()</td>
<td>a0.c</td>
<td>ex_unlink()</td>
<td>az.c</td>
</tr>
<tr>
<td>panic()</td>
<td>a0.c</td>
<td>ex_wait()</td>
<td>az.c</td>
</tr>
<tr>
<td>pline()</td>
<td>a0.c</td>
<td>ex_write()</td>
<td>az.c</td>
</tr>
<tr>
<td>pop()</td>
<td>a0.c</td>
<td>exd1()</td>
<td>ad.c</td>
</tr>
<tr>
<td>Function</td>
<td>File</td>
<td>Function</td>
<td>File</td>
</tr>
<tr>
<td>------------------</td>
<td>--------</td>
<td>------------------</td>
<td>--------</td>
</tr>
<tr>
<td>pos(arg)</td>
<td>gamma.c</td>
<td>exdk()</td>
<td>ad.c</td>
</tr>
<tr>
<td>pr2d()</td>
<td>aj.c</td>
<td>execute()</td>
<td>a1.c</td>
</tr>
<tr>
<td>printf()</td>
<td>a2.c</td>
<td>extend()</td>
<td>ae.c</td>
</tr>
<tr>
<td>psiskp()</td>
<td>a1.c</td>
<td>fappend()</td>
<td>az.c</td>
</tr>
<tr>
<td>pstr()</td>
<td>a0.c</td>
<td>fdat()</td>
<td>aj.c</td>
</tr>
<tr>
<td>putchar()</td>
<td>a2.c</td>
<td>putdat()</td>
<td>a0.c</td>
</tr>
<tr>
<td>putf()</td>
<td>a2.c</td>
<td>putn(n)</td>
<td>a2.c</td>
</tr>
<tr>
<td>puto()</td>
<td>a2.c</td>
<td>putto()</td>
<td>a0.c</td>
</tr>
<tr>
<td>rav0(k)</td>
<td>a5.c</td>
<td>rav1()</td>
<td>a5.c</td>
</tr>
<tr>
<td>red0(k)</td>
<td>a6.c</td>
<td>red1()</td>
<td>a6.c</td>
</tr>
<tr>
<td>reverse()</td>
<td>ai.c</td>
<td>revk()</td>
<td>ab.c</td>
</tr>
<tr>
<td>rlcmp()</td>
<td>a0.c</td>
<td>rline()</td>
<td>a0.c</td>
</tr>
<tr>
<td>rot1()</td>
<td>ac.c</td>
<td>rotk(k)</td>
<td>ac.c</td>
</tr>
<tr>
<td>run()</td>
<td>al.c</td>
<td>s2vect()</td>
<td>a0.c</td>
</tr>
<tr>
<td>scalar()</td>
<td>ae.c</td>
<td>scan0()</td>
<td>ak.c</td>
</tr>
<tr>
<td>scan1()</td>
<td>ak.c</td>
<td>sichk()</td>
<td>ai.c</td>
</tr>
<tr>
<td>size()</td>
<td>a0.c</td>
<td>solve()</td>
<td>a8.c</td>
</tr>
<tr>
<td>strlen()</td>
<td>al.c</td>
<td>takezr()</td>
<td>ab.c</td>
</tr>
<tr>
<td>tback()</td>
<td>ai.c</td>
<td>td1()</td>
<td>ab.c</td>
</tr>
<tr>
<td>term()</td>
<td>a0.c</td>
<td>topfix()</td>
<td>a0.c</td>
</tr>
<tr>
<td>trn0()</td>
<td>ab.c</td>
<td>vfname()</td>
<td>ah.c</td>
</tr>
<tr>
<td>vsave()</td>
<td>aj.c</td>
<td>wsload()</td>
<td>aj.c</td>
</tr>
<tr>
<td>wssave()</td>
<td>aj.c</td>
<td>yyerror()</td>
<td>lex.c</td>
</tr>
<tr>
<td>yylex()</td>
<td>lex.c</td>
<td>yyparse()</td>
<td>a0.c</td>
</tr>
<tr>
<td>yyparse()</td>
<td>yacc.c</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
state 0
\$accept : _line $end

lex0 shift 2
lex1 shift 3
lex2 shift 4
lex3 shift 5
lex4 shift 6
lex5 shift 7
   error
line goto 1

state 1
\$accept : _line $end

\$end accept
   error

state 2
line : lex0_stat
line : lex0_bcommand command eol
line : lex0_tr_command eol
line : lex0_tr_expr eol

lpar shift 28
rpar shift 13
eol shift 11
Quad shift 29
comnt shift 14
transhift 10
string shift 26
nam shift 25
numb shift 33
nfun shift 24
mfun shift 22
dscal shift 35
mdscal shift 34
m shift 30
md shift 37
msub shift 31
mdsub shift 33
   error
stat goto 8
statement goto 12
expr goto 15
e1 goto 17
e2 goto 19
number goto 36
monadic goto 18
hprint goto 16
monad goto 20
smonad goto 21
monday goto 32
scalar goto 23
beomand goto 9
vector goto 27

state 3
line : lex1_stat

lpar shift 26
eol shift 14
Quad shift 29
count shift 14
string shift 26
nam shift 25
numb shift 38
nfun shift 24
mfun shift 22
dscal shift 35
mdscal shift 34
* shift 30
*md shift 37
*sub shift 31
*mdsub shift 33
* error

stat goto 39
statement goto 12
expr goto 15
e1 goto 17
e2 goto 19
number goto 36
monadic goto 18
hprint goto 16
monad goto 20
smonad goto 21
monody goto 32
scalar goto 23
vector goto 27

state 4
line : lex2_func

nam shift 43
nfun shift 44
mfun shift 45
dfun shift 46
* error

func goto 46
header goto 42
args goto 47
*anyname goto 41

state 5
line : lex3_func
nam  shift 43
nfun  shift 44
mfun  shift 45
dfun  shift 46
  error

func  goto 48
header goto 42
args  goto 47
anynam  goto 41

state 6
line : lex4_func

nam  shift 43
nfun  shift 44
mfun  shift 45
dfun  shift 46
  error

func  goto 49
header goto 42
args  goto 47
anynam  goto 41

state 7
line : lex5_fstat

lpar  shift 28
eol  shift 11
Quad  shift 29
c pract  shift 14
tran  shift 55
string  shift 26
nam  shift 57
numb  shift 38
nfun  shift 58
mfun  shift 59
dfun  shift 46
dscal  shift 35
mdscal  shift 34
w  shift 30
wd  shift 37
msub  shift 31
mddsub  shift 33
  error

labels  goto 51
label  goto 53
fstat0  goto 52
stat  goto 54
statement  goto 12
expr  goto 15
e1  goto 17
e2  goto 19
number  goto 36
monadic goto 18
anynmae goto 56
bprint goto 16
monad goto 20
smonad goto 21
monyda goto 32
scalar goto 23
fstat goto 50
vector goto 27

state 6
line : lex0 stat_ (1)
    reduce 1

state 7
line : lex0 bcomand_comand eol
    comexpr shift 61
comnam shift 62
comnull shift 64
comalist shift 63
error
comand goto 60

state 10
line : lex0 tran_eol
line : lex0 tran_expr eol
    lpar shift 28
eol shift 65
Quad shift 29
strng shift 26
nam shift 25
numb shift 38
nfun shift 24
mfun shift 22
dscal shift 35
dscal shift 34
m shift 30
md shift 37
masub shift 31
mdsub shift 33
error
    expr goto 66
e1 goto 17
e2 goto 19
number goto 36
monadic goto 18
monad goto 20
smonad goto 21
monyda goto 32
scalar goto 23
vector goto 27

state 11
stat : eol_ (33)
  . reduce 33

state 12
stat : statement_eol
 eol shift 67
  . error

state 13
bcomand : rpar_ (18)
  . reduce 18

state 14
statement : comnt_ (35)
  . reduce 35

state 15
statement : expr_ (36)
 hprint : expr_hsem output
  sem i shift 69
  . reduce 36
     hsem i goto 68

state 16
statement : hprint_ (37)
  . reduce 37

state 17
expr : el_ (42)
expr : el_d yadic expr
  com shift 81
  com0 shift 79
  asg shift 80
  null shift 74
  dfun shift 73
  dscal shift 77
  mdscal shift 34
  d shift 76
  md shift 37
  mdsub shift 83
state 18

expr : monadic_expr

1par shift 28
@par shift 29
string shift 26
name shift 25
number shift 38
function shift 24
mfun shift 22
dscal shift 35
mdscal shift 34
@ shift 30
md shift 37
msub shift 31
mdsub shift 33
.error

expr goto 84
e1 goto 17
e2 goto 19
number goto 36
monadic goto 18
monad goto 20
smonad goto 21
monady goto 32
scalar goto 23
vector goto 27

state 19

e1 : e2_ (45)
e1 : e2_ lsub subs rbkt

1bkt shift 86
.error reduce 45

1sub goto 85

state 20

monadic : monad_ (61)
.error reduce 61

state 21

monadic : smonad_subr
1bkt shift 86
\[\text{error}\]
subr goto 87

state 22
\[\text{monadic : mfun}_- \quad (53)\]
\[\text{reduce 53}\]

state 23
\[\text{monadic : scalar_comp}\]
\[\text{monadic : scalar_com subr}\]
\[\text{com shift 90}\]
\[\text{com shift 91}\]
\[\text{error}\]
\[\text{comp goto 89}\]

state 24
\[\text{e2 : nfun}_- \quad (47)\]
\[\text{reduce 47}\]

state 25
\[\text{e2 : nam}_- \quad (48)\]
\[\text{reduce 48}\]

state 26
\[\text{e2 : strng}_- \quad (49)\]
\[\text{reduce 49}\]

state 27
\[\text{e2 : vector}_- \quad (50)\]
\[\text{reduce 50}\]

state 28
\[\text{e2 : lpar_expr rpar}\]
\[\text{lpar shift 28}\]
\[\text{Quad shift 29}\]
\[\text{strng shift 26}\]
\[\text{nam shift 25}\]
\[\text{numb shift 38}\]
\[\text{nfun shift 24}\]
\[\text{mfun shift 22}\]
\[\text{dscal shift 35}\]
mdscal shift 34
m shift 30
md shift 37
msub shift 31
mdsub shift 33
, error
expr goto 32
e1 goto 17
e2 goto 19
number goto 36
monadic goto 18
monad goto 20
smonad goto 21
mondya goto 32
scalar goto 23
vector goto 27

state 29
	 e2 : quad_   (52)
	 . reduce 52

state 30
	 monad : m_   (66)
	 . reduce 66

state 31
	 monad : msub_   (67)
	 smonad : msub_   (69)
	 lbkt reduce 69
	 . reduce 67

state 32
	 monad : mondyaw_   (68)
	 . reduce 68

state 33
	 smonad : mdsusb_   (70)
	 mondyaw : mdsusb_   (90)
	 lbkt reduce 70
	 . reduce 90

state 34
	 mondyaw : mdscal_   (88)
	 scalar : mdscal_   (91)
com  reduce 91
com0 reduce 91
det reduce 91
  reduce 88

state 35
  scalar : dscal_  (92)
  reduce 92

state 36
  vector : number_vector
  vector : number_  (54)
  numb shift 38
  reduce 54
  number goto 36
  vector goto 35

state 37
  mondy : md_  (89)
  reduce 89

state 38
  number : numb_  (55)
  reduce 55

state 39
  line : lex1 stat_  (5)
  reduce 5

state 40
  line : lex2 func_  (6)
  reduce 6

state 41
  func : anyname_asg header
  args : anyname_asg anyname anyname
  args : anyname_asg anyname
  args : anyname_  (15)
  asg shift 94
  nam shift 43
  nfun shift 44
  nfun shift 45
dfun shift 46
  . reduce 15

anyname goto 95

state 42
  func : header_   (11)
  . reduce 11

state 43
  anyname : nam_   (93)
  . reduce 93

state 44
  anyname : dfun_   (94)
  . reduce 94

state 45
  anyname : dfun_   (95)
  . reduce 95

state 46
  anyname : dfun_   (96)
  . reduce 96

state 47
  header : args_autos
  eol shift 98
  semi shift 97
  . error
  autos goto 96

state 48
  line : lex3 func_   (7)
  . reduce 7

state 49
  line : lex4 func_   (8)
  . reduce 8
state 50
line: lex5 fstat_ (9)
   reduce 9

state 51
fstat: labels_fstat0
labels: labels_label

1par shift 28
eeol shift 11
Quad shift 29
count shift 14
tran shift 55
string shift 26
nam shift 57
numb shift 58
nfun shift 59
mfun shift 46
decal shift 35
miscal shift 34
w shift 30
md shift 37
msub shift 31
mdsub shift 33
   error

label goto 100
fstat0 goto 99
stat goto 54
statement goto 12
expr goto 15
e1 goto 17
e2 goto 19
number goto 36
monadic goto 18
anyname goto 56
hprint goto 16
monad goto 20
smonad goto 21
mondya goto 32
scalar goto 23
vector goto 27

state 52
fstat: fstat0_ (26)
   reduce 26

state 53
labels: label_ (27)
   reduce 27
state 54
  fstat0 : stat_  (30)
    reduce 30

state 55
  fstat0 : tran_eol
  fstat0 : tran_expr_eol

    lpar  shift 28
    eol  shift 101
    Quad  shift 29
    string  shift 26
    nam  shift 25
    numb  shift 38
    nfun  shift 24
    mfun  shift 22
    dscal  shift 35
    mdscal  shift 34
    m  shift 30
    md  shift 37
    msub  shift 31
    mdsub  shift 33
    error

    expr  goto 102
    e1  goto 17
    e2  goto 19
    number  goto 36
    monadic  goto 18
    monad  goto 20
    smonad  goto 21
    mondy  goto 32
    scalar  goto 23
    vector  goto 27

state 56
  label : anyname_cln

  cln  shift 103
  error

state 57
  e2 : nam_  (48)
  anyname : nam_  (93)

  cln  reduce 93
    reduce 48

state 58
  e2 : nfun_  (47)
anyname : nfun_ (34)
  cln reduce 94
    reduce 47

state 59
  monadic : mfun_ (63)
  anyname : mfun_ (35)
  cln reduce 95
    reduce 63

state 60
  line : lex0 bcommand comand_eol
    eol shift 104
      . error

state 61
  comand : comexpr_expr
    lpar shift 28
    quad shift 29
    string shift 26
    nam shift 25
    numb shift 38
    nfun shift 24
    mfun shift 22
dscal shift 35
mdscal shift 34
  m shift 30
  md shift 37
  msub shift 31
mdsub shift 33
    . error
expr goto 105
e1 goto 17
e2 goto 19
number goto 36
monadic goto 18
monad goto 20
smonad goto 21
smodya goto 32
scalar goto 23
vector goto 27

state 62
  comand : comnam_anyname
    nam shift 43
    nfun shift 44
    mfun shift 45
dfun shift 46
   . error
anyname goto 106

state 63
  comand : comlist_anynlist
  
  nam shift 43
  nfun shift 44
  mfun shift 45
  dfun shift 46
  . error
anyname goto 108
anylist goto 107

state 64
  comand : commull_' (22)
  . reduce 22

state 65
  line : lex0 tran eol_ (3)
  . reduce 3

state 66
  line : lex0 tran expr_eol
  
  eol shift 109
  . error

state 67
  stat : statement eol_ (34)
  . reduce 34

state 68
  hprint : expr hsemi_output
  
  lpar shift 28
  Quad shift 29
  string shift 26
  nam shift 25
  numb shift 38
  nfun shift 24
  mfun shift 22
  dscal shift 35
  mdscal shift 34
  w shift 30
  md shift 37
msub shift 31
mdsub shift 33
. error

output goto 110
expr goto 111
e1 goto 17
e2 goto 19
number goto 36
monadic goto 18
hprint goto 112
monad goto 20
smonad goto 21
mendya goto 32
scalar goto 23
vector goto 27

state 69
hsemi : semi_ (41)
. reduce 41

state 70
expr : el dyadic_expr
ipar shift 28
quad shift 29
string shift 26
num shift 25
numb shift 38
nfun shift 24
mfun shift 22
dscal shift 35
mdscal shift 34
m shift 30
md shift 37
msub shift 31
mdsub shift 33
. error

expr goto 113
e1 goto 17
e2 goto 19
number goto 36
monadic goto 18
monad goto 20
smonad goto 21
mendya goto 32
scalar goto 23
vector goto 27

state 71
dyadic : dyad_ (71)
. reduce 71
state 72
dyadic : sd'yad_subr

1bkt shift 89
. error
subr goto 114

state 73
dyadic : dfun_ (73)
. reduce 73

state 74
dyadic : null_dot scalar

dot shift 115
. error

state 75
dyadic : scalar_dot scalar

dot shift 116
. error

state 76
dyad : mondy_ (80)
. reduce 80

state 77
dyad : dscal_ (81)
scalar : dscal_ (92)

dot reduce 92
. reduce 81

state 78
dyad : d_ (82)
. reduce 82

state 79
dyad : com0_ (83)
. reduce 83
state 80
  dyad : asg_ (84)
     reduce 84

state 81
  dyad : com_ (85)
  mdcem : com_ (87)
  lbkt reduce 87
     reduce 85

state 82
  sdyad : mdcem_ (76)
     reduce 76

state 83
  mdcem : mdsup_ (86)
  mondy : mdsup_ (90)
  lbkt reduce 86
     reduce 90

state 84
  expr : monadic expr_ (43)
     reduce 43

state 85
  ei : e2 isup_sub subs rbkt
  sub : _ (50)
  lpar shift 28
  Quad shift 29
  string shift 26
  name shift 25
  numb shift 38
  nfun shift 24
  mfun shift 22
  dscal shift 35
  mdscal shift 34
  m shift 30
  md shift 37
  mdsup shift 31
  mdsup shift 33
     reduce 60
  expr goto 119
  e1 goto 17
  e2 goto 19
number  goto 36
subs  goto 117
sub  goto 118
monadic  goto 18
monad  goto 20
smonad  goto 21
mondya  goto 32
scalar  goto 23
vector  goto 27

state 86
  sub :  lbkt_.   (56)
  .  reduce 56

state 87
  monadic :  smonad subr_.   (62)
  .  reduce 62

state 88
  subr :  lbkt_.expr rbkt

  lpar  shift  28
  Quad  shift  29
  string  shift  26
  nam  shift  25
  numb  shift  38
  nfun  shift  24
  mfun  shift  22
  dscal  shift  35
  mdscal  shift  34
  m  shift  30
  md  shift  37
  msub  shift  31
  mdsub  shift  33
  .  error

  expr  goto 120
  e1  goto  17
  e2  goto  19
  number  goto 36
  monadic  goto 18
  monad  goto 20
  smonad  goto 21
  mondya  goto 32
  scalar  goto 23
  vector  goto 27

state 89
  monadic :  scalar comp_.   (64)
  .  reduce 64
state 90
    monadic : scalar com_subr
    comp : _com_ (78)
    lbrkt shift 93
      reduce 78
    subr goto 121

state 91
    comp : _com_ (79)
      reduce 79

state 92
    e2 : 1par expr_rpar
    rpar shift 122
      error

state 93
    vector : number vector_ (53)
      reduce 53

state 94
    func : anyname asg_header
    nam shift 43
    nfun shift 44
    wfun shift 45
    dfun shift 46
      error
    header goto 123
    args goto 47
    anyname goto 124

state 95
    args : anyname anyname anyname anyname
    args : anyname anyname_ (14)
    nam shift 43
    nfun shift 44
    wfun shift 45
    dfun shift 46
      reduce 14
    anyname goto 125

state 96
    header : args autos_ (12)
reduce 12

state 97
    autos : semi nam autos
    nam shift 126
    . error

state 98
    autos : eol_ (17)
    . reduce 17

state 99
    fstat : labels fstat0_ (25)
    . reduce 25

state 100
    labels : labels label_ (28)
    . reduce 28

state 101
    fstat0 : tran eol_ (31)
    . reduce 31

state 102
    fstat0 : tran expr_eol
    eol shift 127
    . error

state 103
    label : acronym cln_ (29)
    . reduce 29

state 104
    line : lex0 bcomand comand eol_ (2)
    . reduce 2

state 105
    comand : comexpr expr_ (19)
state 106
command : comnam anyname_  (20)
  . reduce 20

state 107
command : comlist anylist_  (21)
  anylist : anylist_anyname
nam shift 43
nfun shift 44
mfun shift 45
dfun shift 46
  . reduce 21
  anyname goto 128

state 108
  anylist : anyname_  (24)
  . reduce 24

state 109
  line : lex0 tran expr eol_  (4)
  . reduce 4

state 110
  hprint : expr hsemi output_  (38)
  . reduce 38

state 111
  hprint : expr_hsemi output
  output : expr_  (39)
  semi shift 69
  . reduce 39
  hsemi goto 68

state 112
  output : hprint_  (40)
  . reduce 40

state 113
expr * ei dyadic expr_  (44)  
    . reduce 44

state 114
    dyadic * adyad subr_  (72)  
    . reduce 72

state 115
    dyadic * null dot_scalar
    dscal shift 35
    mdscal shift 130
    . error
    scalar goto 129

state 116
    dyadic = scalar dot_scalar
    dscal shift 35
    mdscal shift 130
    . error
    scalar goto 131

state 117
    ei = e2 lsub subs_rbkt
    subs = subs_semi sub
    rbkt shift 132
    semi shift 133
    . error

state 118
    subs = sub_  (57)
    . reduce 57

state 119
    sub = expr_  (59)
    . reduce 59

state 120
    subr = lbk expr_rbkt
    rbkt shift 134
    . error
state 121
   monadic : scalar com subr_   (65)
      . reduce 65

state 122
   e2 : 1par expr rpar_   (51)
      . reduce 51

state 123
   func : anyname asg header_   (10)
      . reduce 10

state 124
   args : anyname_anyname anyname
   args : anyname_anyname
   args : anyname_   (15)
   nam shift 43
   nfun shift 44
   mfun shift 45
   dfun shift 46
   . reduce 15
   anyname goto 95

state 125
   args : anyname anyname anyname_   (13)
      . reduce 13

state 126
   autos : semi nam_autos
   eol shift 98
   semi shift 97
      . error
   autos goto 135

state 127
   fstat0 : tran expr eol_   (32)
      . reduce 32

state 128
   anylist : anylist anyname_   (23)
state 128
  dyadic : nulldot scalar_ (74)
  . reduce 74

state 130
  scalar : mdscale_ (91)
  . reduce 91

state 131
  dyadic : scalar dot scalar_ (75)
  . reduce 75

state 132
  ai : a2 lsub subs rbkt_ (46)
  . reduce 46

state 133
  subs : subs semi_sub
  sub : _ (60)

  lpar shift 28
  Quad shift 29
  strng shift 26
  num shift 25
  numb shift 30
  nfun shift 24
  mfun shift 22
  dscal shift 35
  mdscale shift 34
  m shift 30
  md shift 37
  msub shift 31
  mdsub shift 33
  . reduce 60

  expr goto 119
  e1 goto 17
  e2 goto 19
  number goto 36
  sub goto 136
  monadic goto 18
  monad goto 20
  smonad goto 21
  mendya goto 32
  scalar goto 23
vector goto 27

state 134
    subr : lbrkt expr rbrkt_ (77)
    . reduce 77

state 135
    autos : semi nam autos_ (16)
    . reduce 16

state 136
    subs : subs semi sub_ (58)
    . reduce 58

62/127 terminals, 37/300 nonterminals
97/600 grammar rules, 137/750 states
0 shift/reduce, 0 reduce/reduce conflicts reported
48/390 working sets used
memory: states, etc. 690/12000, parser 299/12000
61/600 distinct lookahead sets
23 extra closures
288 shift entries, 12 exceptions
73 goto entries
136 entries saved by goto default
Optimizer space used: input 687/12000, output 278/12000
278 table entries, 64 zero
maximum spread: 236, maximum offset: 290
%{
%
union {
    char  *charp;
    char  charval;
}
%term  lex0, lex1, lex2, lex3, lex4, lex5, lex6
%term  lpar, rpar, lbrkt, rbrkt, eol, unk
%term  (charval)  com, com0, Quad, asg
%term  null, dot, cin, semi, comm, tran
%term  (charptr)  strntag, num, nfun, mfun, dfun
%term  (charval)  cocompr, cocomn, cocoml, cocoms
%term  (charval)  dscal, mdscal
%term  (charval)  m, d, md, msub, mdsub
%type  (charptr)  func, header, args, autos, labels, label
%type  (charptr)  fstat0, stat, statement, output, expr
%type  (charptr)  el, e2, number, subs, sub, monadic
%type  (charptr)  dyadic, subr, anyname, hprint
%type  (charval)  comand, lsub, monad, smonad, edyad
%type  (charval)  comp, dyad, mdcom, monadya, scalar
%
%
#include "apl.h"

int  vcount;
int  scount;
int  litflag;
int  lexsym;
int  context;
char  *iline;
char  *ccharp, *ccharp2;
data  lnumb;  /* current label number */
char  *labapp;  /* label prologue */
char  *labeppe;  /* label epilogue */
int  immedcmd;  /* immediate command number */
%
%
/*
* line-at-a-time APL compiler.
* first lexical character gives context.
*/
line:

/*
* immediate.
*/
lex0  stat =
{
integ = ccharp[-1];
if(integ != ASGN &
teg != PRINT
    & integ != COMMT)
    *ccharp++ = PRINT;
    *ccharp++ = EOL;
}
lex0 becomand comand eol =
{            *ccharp++ = IMMED;
            *ccharp++ = $3;
} !

/* immediate mode state indicator stuff */
lex0 tran eol =
{            *ccharp++ = SICLR0;
} !
lex0 tran expr eol =
{            *ccharp++ = SICLR;
} !

/* quad input */
lexi stat !

/* function definition */
lex2 func !

/* function prolog */
lex3 func !

/* function epilog */
lex4 func !

/* function body */
lex5 fstat ;

/* function header */
func:
    anyname asg header =
    {     switch(context) {
        case lex3:
            name($$, AUTO);
    }
/*
 * see comments in a1.c/funcomp()
 * concerning
 * label processing.
 */
*ccharp++ = ELID;
break;

case lex4:
    ccharp2 = ccharp;
    *ccharp++ = EOL;
    name($$, RVAL);
    name($$, REST);
invert($$, ccharp2);
}

header =
{
    if(context == lex3)
        *ccharp++ = ELID;
    if(context == lex4){
/* pop previous result */
    *ccharp++ = EOL;
/* return empty result */
    *ccharp++ = NILRET;
}
};

header;

args autos =
{
    if(context == lex4)
        invert($$, $2);
};

args:

 anyone anyname anyname anyname =
{
    $$ = ccharp;
    switch(context) {
        case lex2:
            name($2, BF);
            break;
        case lex3:
            name($3, ARG2);
            name($1, ARG1);
            break;
        case lex4:
            name($1, REST);
            name($3, REST);
    }
};

 anyone anyname =
{
    $$ = ccharp;
    switch(context) {

case lex2:
    name($1, NF);
    break;

case lex3:
    name($2, ARG1);
    break;

case lex4:
    name($2, REST);
}

anyname =
{
    if(context == lex2)
        name($$, NF);
    $$ = ccharp;
}

autos:
    semi nam autos =
    {
        $$ = $$;
        switch(context) {
            case lex3:
                name($2, AUTO);
                break;
            case lex4:
                ccharp2 = name($2, REST);
                invert($$, ccharp2);
        }
    }

eol =
    {
        $$ = ccharp;
    }

/*
 * system commands
 */
beomand:
    rpar =
    {
        litflag = -1;
    }
comand:
    commexpr expr 1
    commnam anyname =
    {
        name($2, NAME);
    } 1
    comalist anylist 1
    comnull ;
anylist:
  anylist anyname =
  {
    *ccharp++ = IMMED;
    *ccharp++ = immedcmd;
    name($2, NAME);
  };
  anyname =
  {
    name($1, NAME);
  };

/*
 * statement:
 * comments
 * expressions
 * heterogeneous output
 * transfers (in functions)
 */

fstat:
  labels fstat0 | fstat0;

labels:
  label | labels label;

label:
  anyname cln =
  {
    if(labgen)
      genlab($1);
  };

fstat0:
  stat =
  {
    integ = ccharp[-1];
    if(integ != ASGN && integ != PRINT
      && integ != COMMENT)
      *ccharp++ = PRINT;
  };
  tran eol =
  {
    $quire = ccharp;
    *ccharp++ = BRAN0;
  };
  tran expr eol =
  {
    $quire = "$2":
    *ccharp++ = BRAN;
  };

stat:
  eol =
  {
    $quire = ccharp;
    *ccharp++ = COMMENT;
  };

statement e01;
statement:
    count =
    {
        litflag = 1;
        $$ = ccharp;
        ccharp++ = COMNT;
    }
expr | hprint;
hprint:
    expr hsemi output ;
output:
    expr =
    {
        ccharp++ = PRINT;
    }
hprint ;
hsemi:
    semi =
    {
        ccharp++ = HPRINT;
    }
expr:
    e1 | monadic expr =
    {
        invert($$, $2);
    }
    e1 dyadic expr =
    {
        invert($$, $3);
    }
e1:
e2 | e2 lsub subs rbkt =
    {
        invert($$, $3);
        ccharp++ = INDEX;
        ccharp++ = scount;
        scount = $2;
    }
e2:
nfun =
    {
        $$ = name($$, FUN);
    }
nam =
    {
        $$ = name($$, NAME);
    }
string =
    {
        $$ = ccharp;
        ccharp += 2;
        integ = iline-13;
vecoun = 0;
for(;;)
{
    if(*iline == '\n') {
        nlexsym = unc;
        break;
    }
    if(*iline == int eq) {
        iline++;
        if(*iline != int eq)
            break;
    }
    *cchar++ = *iline++;
    vecoun++;
}
((struct charstc *)$$)->c[0] = QUOT;
((struct charstc *)$$)->c[1] = vecoun;
}

vector =
{
    *cchar++ = CONST;
    *cchar++ = vecoun;
    invert($$, cchar-2);
}

1par expr rpar =
{
    $$ = $2;
}

Quad =
{
    $$ = cchar;
    *cchar++ = $i;
}

vector:

number vector =
{
    vcount++;
}

number =
{
    vcount = 1;
}

number:

num =
{
    $$ = cchar;
    cchar += copy(DA, &datum, cchar, 1);
}

/*
 * indexing subscripts
 * optional expressions separated by semi
 */

1sub:

1bkt =
{
    $$ = scount;

scount = 1;
}

sub:
    sub 1
    sub semi sub =
    {
        invert($$, $$3);
        scount++;
    }

sub:
    expr 1
    =
    {
        $$ = ccharp;
        *ccharp++ = ELID;
    }

/*
 * return a string of a monadic operator.
 */
monadic:
    monad =
    {
        $$ = ccharp;
        *ccharp++ = $1;
    }
    smonad subr =
    {
        $$ = $2;
        *ccharp++ = $1+1;
    }
    sfun =
    {
        $$ = name($$, FUN);
    }
    scalar comp =
    {
        $$ = ccharp;
        *ccharp++ = $2+1;
        *ccharp++ = $1;
    }
    scalar com subr =
    {
        $$ = $3;
        *ccharp++ = $2+3;
        *ccharp++ = $1;
    }
    monad:
    m 1
    msub 1
    mondy a =
    {
        $$++;
    }
    smonad:
    msub 1
mdsub =
{
    ** = 2;
}

/*
* return a string of a dyadic operator.
*/
dyadic:
dyad =
{
    ** = ccharp;
    *ccharp++ = $1;
}
sdyad subr =
{
    ** = $2;
    *ccharp++ = $1;
}
dfun =
{
    ** = name(**, FUN);
}
null dot scalar =
{
    ** = ccharp;
    *ccharp++ = OPRDD;
    *ccharp++ = $3;
}
scalar dot scalar =
{
    ** = ccharp;
    *ccharp++ = IPRDD;
    *ccharp++ = $1;
    *ccharp++ = $3;
}
sdyad:

dcom =
{
    ** = 2;
}

/*
* single expression subscript
* as found on operators to select
* a dimension.
*/
subr*
  lbrk expr rbk =
  {
    ** = $2;
  }

/*
* various combinations
*/
comp:
  com | com0 ;
dyad:
  mndya | dscal | d | com0 | asg | com ;
cmdcom:
  cmdsub | com ;
mndya:
  mndscal | md | cmdsub ;
scalar:
  madscal | dscal ;
anyname:
  nae | nfun | nfun | dfun ;
%
#include "tab.c"
#include "iex.c"