APL\11

An APL Interpreter for UNIX[1]

Michael Cain

November 12, 1992

1.0 Introduction

This manual describes APL\11, an APL interpreter that runs under the UNIX operating system. The interpreter is written in C and the source code is freely available. I currently check a single set of source code on all of

- SunOS 4.1 on a Sun3 using the bundled cc,
- SunOS 4.1 on a SPARC using the bundled cc,
- SunOS 4.1 on a SPARC using GNU gcc 1.40, and
- LINUX on a 386sx using GNU gcc 2.2.2.

The code is reasonably portable, subject to several restrictions including interchangeable pointers and ints. Portability is discussed in more detail in another section.

APL\11 has a relatively long history. The original program was written by Ken Thompson at Bell Laboratories, apparently in the days before Version 6 UNIX. This version of the interpreter was extensively modified at Purdue University.

Other versions have been included on the Berkeley distribution tapes. I got the Purdue version in 1981 while I was working at Bell Labs, then carried it with me to Bellcore and U S WEST.

Early in '92, I acquired a 386sx, a copy of Linus Torvalds' linux, and a copy of Bellcore's MGR windowing software. And a little voice in the back of my head that said "This multitasking virtual-memory windowed system is nice, and if it had APL it would be just about perfect!" There's no accounting for taste. After asking USENET, it was apparent that no other source-code available UNIX-based APL interpreters had been written. Which eventually led to the current situation. AT&T has given permission to distribute the source code, and I've cleaned it up, fixed several errors, written some new documentation and formatted a lot of the old Purdue documentation, added a little additional functionality. It's available for general use. Subject to the various lawyer-type restrictions in the next section.

1 UNIX is a registered trademark.

It's important to acknowledge the contributions of others: Ken Thompson, for starting it; John Bruner, the graduate student at Purdue from whom I got my copy and, based on the comments in the code, did a lot of work on making the interpreter more robust; Anthony Reeves, then of the EE faculty at Purdue, whose name appears on much of the documentation; Marty Glopta at Bell Labs who did a lot of the initial legwork involved in getting approval to distribute the source; and Lee Dickey for allowing me to use watserv1 as a distribution medium.

I plan to provide support for the interpreter as my personal time and interests permit. The normal distribution channel will be the ftp archive at watserv1.waterloo.edu. If you are actively using APL\11, drop me a line. Mail should be sent to mcain@advtech.uswest.com. Bug reports are welcome, although I can't promise how soon they'll get fixed.

Patches that fix bugs are even more welcome. I would like to extend the interpreter to include more modern APL features like nested arrays.

Enjoy!

2.0 Legal Stuff

1. Definitions: "Licensor" is U S WEST Advanced Technologies, Inc., a Colorado corporation, whose principal place of business is located at 4001 Discovery Drive, Boulder, Colorado, 80303. "Software" means the source code computer programs known as APL\11 and all related materials, documentation, and information received by Licensee from Licensor.
2. Rights in Software: Licensor has obtained license rights under a quit claim license from AT&T which allows the free distribution of the Software to third persons.
3. License Grant: In accordance with the terms herein, Licensor grants to Licensee and Licensee accepts from Licensor a royalty free, perpetual, non-exclusive and non-transferable license to use, copy, modify and sublicense the Software to third persons.
4. Warranty Disclaimer: THIS SOFTWARE IS LICENSED "AS IS," AND LICENSOR DISCLAIMS ANY AND ALL WARRANTIES, WHETHER EXPRESS OR IMPLIED, INCLUDING (WITHOUT LIMITATION) ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.
5. Indemnity: Licensee hereby indemnifies Licensor and holds Licensor harmless from and against any and all claims, including claims of infringement, which arise out of Licensee's use or sublicense of the Software as licensed hereunder.
6. Limitation of Liability: In no event shall Licensor be liable for any indirect, incidental, consequential, special, or exemplary damages or lost profits, even if Licensor has been advised of the possibility of such damages. SOME STATES DO NOT ALLOW THE LIMITATION OR EXCLUSION OF LIABILITY FOR INCIDENTAL OR CONSEQUENTIAL DAMAGES, SO THE ABOVE LIMITATION OR EXCLUSION MAY NOT APPLY TO YOU.
7. General Provisions:
   A. This License shall be governed by the laws of the State of Colorado.
   B. Licensee also agrees not to use any trade name, service mark, or trademark of Licensor or refer to Licensor in any promotional activity or material without first obtaining the prior written consent of Licensor.
   C. No action, regardless of form, arising out of the License may be brought by Licensee more than one year after the cause of action has arisen.
   D. If any provision of this License is invalid under any applicable statute or rule of law, it is to that extent to be deemed omitted.
   E. The waiver or failure of Licensor to exercise in any respect any right provided for herein shall not be deemed a waiver of any further right hereunder.
   F. Each party acknowledges that it has read this License, it understands it, and agrees to be bound by its terms, and further agrees that this is the complete and exclusive statement of the License betweeen the two parties, which supersedes all prior proposals, understandings, whether oral or written, between the parties relating to this License. This License may not be modified or amended except by written instrument duly executed by both parties.

3.0 Building the Interpreter – Simple Case

This section describes the steps needed to build the APL\11 interpreter from source, assuming that nothing goes wrong. Building the interpreter requires yacc and a K&R-compatible C compiler. A simple makefile is included with the source. The command

```
$ make
```

with no arguments should produce an executable file named "*apl*".

If make is not available, the interpreter can be built by hand almost as easily. The sequence of commands

```
$ yacc apl.y
$ cc -o apl *.c -lm
    [compiler diagnostics]
$
```

should produce the same executable file. A relatively large number of warnings will be issued by modern C compilers.

The reasons are discussed in more detail in the next section.

A quick test of the interpreter can be run with the command

```
$ ./apl <quick
```

which should get the following results

```
$ ./apl <quick
a p l  1 1
12 nov 1992

clear ws
         a{1 2 3
         b{3 4 5
         a+b
 4 6 8
         aXb
  3  8 15
         a%b
 .333333333 .500000000 .600000000
         aJ.*b
   1   1   1
   8  16  32
  27  81 243
         )off
$
```

If so, you can skip the next section. If not, getting APL running on your machine will be somewhat more complicated.

4.0 Building the Interpreter – Hard Case

Portability is a relative thing. Successfully moving a piece of code from one machine to another involves many different factors. This section discusses some of the assumptions made by the APL\11 source code. Moving the interpreter to a machine on which some of those assumptions are no longer true will probably be quite painful. This material is intended to help you decide if you are in that situation, or if there's still some hope of an "easy" fix.

The APL\11 source shows its age in several ways. First, it's K&R rather than ANSI C. Second, it predates the void data type, so is full of functions of type int that should be type void. Third, it goes back to the bad old days when pointers and ints were treated as though they were interchangeable. Because of these things, running the code through a modern C compiler results in a lot of warnings.

The assumption that pointers (all types of pointers) and ints are interchangeable is critical. I seriously doubt that the code could ever be ported to an environment where this is not true. Some of the MS-DOS compiler models do not satisfy this condition. The code also does a lot of pointer arithmetic and comparisons. If pointers can't be treated as ints, these are apt to be a source of problems. Again, situations like the MS-DOS segmented models should be avoided.

While the code doesn't use very many different types of structures, it does some fairly ugly things with the ones that are defined. I believe that the following is the worst case. If your compiler doesn't mess this up, it probably won't mess up the other structures.

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

Item structures are not directly allocated. Instead, when a new item is being created, the code computes the amount of storage actually needed to hold the item and information about it. This is space for the four ints rank, type, size and index, space for as many ints as the value in rank, and space for as many datas as the value in size. Data is usually defined as a double. Assuming that p points to the allocated space, `p->datap` is set to `&(p->dim[p->rank])`.

Clearly, inserting padding for alignment or some rearrangements of the elements of the structure will break the code.

Another assumption made by the code is that doubles (or floats in the case of a single-precision version) do not have any alignment requirements. Some machines, for example, require that a double begin at an address that is a multiple of eight. Compilation of constant numeric expressions into byte code will be broken if the machine has alignment requirements. Fortunately, starting a double at an arbitrary location is okay with most contemporary processors. However, some optimizing compilers may force alignment in order to improve execution speed.

As previously mentioned, exceptions are an area where it is hard to write portable code. I have attempted to make the source code portable in the compiler sense by `#ifdef`'ing each use of an exception name (or signal name, if you prefer). For example,

```
#ifdef SIGPIPE
    signal(SIGPIPE, panic);
#endif
```

This makes the compiler happy, but is not enough to guarantee that the interpreter will do the expected thing in all exception cases.

5.0 Running the Interpreter

The APL\11 interpreter is invoked with the simple command "*apl*". APL (from here on I'll generally use APL and APL\11 interchangeably; if I mean some other APL, it should be clear from context) responds with a header message. A workspace name may be provided as a command-line argument. If the named workspace can be found, it is loaded. If it is not found, APL looks for a workspace named "*continue*" and loads that instead. If a workspace is loaded and contains a latent expression, the latent expression is executed.

Workspaces are simply UNIX data files. The name of the workspace is the same as the file name. Workspaces can be saved in either native or portable form. Save and load in the native form is much quicker than the portable form. The portable format used by APL\11 does not, unfortunately, conform to the standard workspace interchange format. So while it allows APL11 workspaces to be moved from one machine running APL11 to another, workspaces can't be moved to other APLs.

Once entered, the main command loop prompts the user for input. The user types an APL system command or expression. APL attempts to execute the expression. The set of system commands is described in detail in a later section. Many, such as "*)off*" and "*)clear*" are much the same as the system commands of other APLs. Some, such as "*)editf*" and "*)read*", are unique (in my experience) to APL\11.

5.1 Character Set

APL uses the ASCII character set to represent APL characters. The particular mapping takes some getting used to.

The complete allowed character set is described in detail in Appendix A. A quick description is provided here.

- Names, such as used for variables and functions, use lower-case alphanumerics. A second character set, corresponding to the traditional understruck characters, is also available and is generated by overstriking the lower-case characters with an upper-case "*F*".
- Where a particular non-alphanumeric APL character has a look-alike in the ASCII set, the look-alike is used. This covers plus, minus, equal, square brackets, etc.
- Where there are no true look-alikes, one of two rules is followed. If there is an ASCII character that resembles the APL character, it is used. This leads to "*%*" for divide, "*X*" for multiplication, etc.
- The other way of handling characters is to use the upper-case ASCII character at the corresponding keyboard position. This rule yields "*A*" for alpha, "*R*" for rho, "*E*" for epsilon, "*Y*" for the up-arrow (take), "*U*" for the down-arrow (drop), etc.
- Overstruck characters are typed using backspace. The order in which the component characters are typed does not matter. One common problem occurs when backspace is also used as the erase character. In that case, the backspace must be "escaped" in some fashion so that it actually reaches APL. Or the user may find it more convenient to change the erase character to something else that is used by neither the interpreter or the tty driver (something like control-F).

The internal character set is such that it should be a straightforward matter to write input and output filters for interacting with the interpreter. These filters would be specific to particular display devices. Unfortunately, display editors will generally not work gracefully with such filters. At some point in the

future, I would like to add some simple function editing capabilities to the interpreter itself in order to get around that problem.

Literal character strings have been extended somewhat in order to be more UNIX-like. The backslash character is used to indicate some special characters. `\n` is newline, `\b` is backspace, `\t` is tab, `\r` is carriage return, and `\\` is a backslash. The backslash can also be used to specify three-digit octal values. Using an octal value, the ASCII ESC character would be `\033`. This extension makes it easier to handle things like escape sequences for terminal control.

By way of some summary, here is a transcript of a short APL session. I have always liked APL\11's) script capability for recording a session. Notice that since function editing is done by an external editor, none of the editing portion of the session appears in the record. Also, the plot function is my own, and isn't very good.

```
linux> apl
a p l \ 1 1
12 nov 1992

clear ws
        )script log
[new file]
        )load utility.ws
  18.52.14 08/25/92 utility.ws
        )fns
plot   and  vs  space
        )editf and
        a{100 1RI100     C a is one column, 1-100
        a{a-1  C subtract 1 from everything
        a{aX0.0628       C map to 0 to two-pi
        a{a,(10a),20a    C add sine and cosine columns
        20 60 plot a     C plot
      1.00 XXXXX     000000000  XXXX
 |    XX  00       00          XX
 |     XX0 0       X
 |     0XX  0     X
 |    00  X  00  X
       .50 |   00    X  00X
 |  00       X  00        X
 |  0         X  0       X
 | 0          XX  0     X
 |0 XX  0   X
       .00 0   X   0 XX  0
 |    XX 00         XX  0
 |     X   0        X  0
 |      X  00      X  00
 |       X  00    X  00
      -.50 |        X  00 XX  0
 |         X  0XX  0
 |X XX0 0
 | XX        XX 000       00
 |   XXXXXXXX     000000000
     -1.00 |         |         |         |         |         |         |
         .00       1.17      2.33      3.50      4.67      5.83      7.00

        )script off
```

5.2 Saving and Loading

Workspaces are explicitly loaded by the command "`)load wsname`". The current workspace can be explicitly saved by the command "`)save wsname`". The workspace name must always be provided. Additional information about the `)load` and `)save` system commands, as well as the other system commands supported by APL\11, is provided in Appendix B.

A saved workspace includes function definitions and variables. If there are suspended functions, all of

the local variables for those functions are also saved.

However, the APL state indicator is not saved, so that those variables become global in scope when the saved workspace is loaded. Trying to save the state indicator is very hard in the current version of the interpreter because of the way the main program loop is implemented. Someone should fix this sometime.

`)save` and `)load` write and read native-mode workspaces. The commands `)psave` and `)pload` write and read workspaces in a more portable format. This format should be portable from one machine to another, so long as they both use the ASCII character set. When examined manually, the format can be seen to be somewhat human readable. Editing a saved workspace is not safe. Function definitions are terminated by a null character, which most editors will simply drop while reading. The image saved by the editor would not be acceptable when read by APL.

`)psave` and `)pload` are slower than their native-mode equivalents, especially if the workspace contains large amounts of data. Saving a workspace in portable format can result in some loss of precision for floating point values.This is a result of the portable workspace format, which represents numbers using decimal digits and ASCII characters (that is, `"1.23456e-78"`).

5.3 Function Editing

APL\11 does not include a built-in editing capability. All functions are edited by dumping the source code into a temporary file and then invoking the user's editor-of-choice on that temporary file. The temporary file is created in `/tmp`. Which editor is invoked is determined by the value of the shell variable `EDITOR`. If `EDITOR` is not defined, the name of the editor defaults to `"vile"`, a clone of Berkeley vi. I thought about defaulting to `"ed"`, but decided that only us old-timers remember how to use line-oriented non-display editing.

The new function definition is read back in when the editing process terminates. The source code is not compiled into byte codes until the "executable" is actually needed.

Syntax errors in the source code will not be detected until compilation occurs. As described previously, overstruck characters must have the form a-backspace-b. In order to use these characters in a function, the editor must be capable of putting real backspaces into the source file. Most editors are capable of displaying and inserting backspaces, but may vary in the details of how it's actually done.

APL will not allow the user to edit suspended functions. If the user attempts to do so, APL responds with the message `"si damage -- type ')reset'"`. There isn't any actual damage to the state indicator, although the message might be interpreted that way. Another reason for adding simple function editing capabilities to APL\11 itself (see also the preceding section on "Function Editing") is to allow the user to modify suspended functions when such changes do not create any problems.

5.4 Exceptions

Proper handling of exception conditions is one of the hardest areas to make truly portable. Different machines and operating systems allow different kinds of exceptions to occur. Some are pretty much the same on all machines; division by zero is an error on most computers[2]. Bad pointers, on the other hand, can result in a variety of different errors on different machines: segmentation violations, bus errors, etc.

2 One exception that I know about is old CDC hardware. There, division by zero results in a special bit pattern representing an indefinite or undefined value, but is not an error. However, attempting to use the indefinite value in a subsequent calculation is an error. The subsequent calculation always seemed to be in a different subroutine several pages away in the source listing. This made debugging more difficult.

I have attempted to make the source code portable in the compiler sense by `#ifdef`'ing each use of an exception name (or signal name, if you prefer). For example,

```
#ifdef SIGPIPE
    signal(SIGPIPE, panic);
#endif
```

This keeps the compiler happy because the code doesn't reference signals that don't exist. However, on any particular machine there may be signals which exist but are not properly trapped. When the corresponding exceptions occur, the system default action (often a core dump) will be taken rather than the interpreter's recovery actions.

6.0 Internals

Someday, this section will describe the internal workings of the APL\11 interpreter. It is intended as a starting point for a programmer attempting to modify the interpreter itself. It will contain a lot of information that I wish I had had before I started working on the code. However, that day has not yet arrived, and these couple of paragraphs are placeholders only.

For the brave of heart who jump in anyway, some housekeeping hints. I edit C language source code in a fairly big text window (40 rows of 90 columns) with tab stops set at every fourth column. I've been through the entire body of code and brought it pretty much into alignment with my own personal preferences for things like indenting, grouping, and brace placement. Looking at the code using other arrangements such as eight-column tab stops or a "normal" 24 by 80 screen will probably make things look uglier than they actually are.

6.1 Principle Data Structures

6.2 Main Program Loop

6.3 Compiling

6.4 Execution

Appendix A

The APL\11 Character Set

This section provides a quick reference for the character set used by the APL\11 interpreter. Note that both upper- and lower-case characters are used. <*bs*> denotes backspace.

```
a-z             letter
0-9             digit
`               negative sign
'--'            string
C <bs> J  C     comment
( )
[ ; ]           indexing
L               quad
L <bs> '        quote quad

                dyadic          monadic
                ----------------------
+               add             plus
-               sub             negate
X               mult            sign
%               div             reciprical
|               mod             absolute value
D               min             floor
S               max             ceiling
*               pwr             exp
O <bs> *        log             ln (log base e)
O               circle funct    pi times
' <bs> .        combinatorial   factorial
!               combinatorial   factorial
^               and
V               or
^ <bs>          nand
V <bs>          nor
<               lt
>               gt
$               le
&               ge
=               eq
#               ne
                not
?               deal            random number
R               rho             rho
I               iota            iota
E               epsilon
N               encode
B               decode
\ <bs> 0        transpose       transpose
,               catenate        ravel *
Y               take
U               drop
{               assign
}               goto
B <bs> N        i-beam          i-beam
L <bs> %        matrix div      matrix inverse
| <bs> H        grade up.*
G <bs> |        grade down *
B <bs> J        execute
N <bs> J        format
/               compress *
/ <bs> -        compress
/ <bs> -        compress
\               expand *
\ <bs> -        expand
O <bs> |        rotatereverse *
O <bs> -        rotatereverse
op /            reduce *
```

```
op / <bs> - reduce
J.op        outer product
op.op       inner product
```

* may be subscripted with a scalar

Appendix B

System Commands

The following is a complete list of APL system commands.

`)clear`

This command is used to completely initialize an APL workspace. Usually when APL is started, it will print: `"clear ws"`. This means that no internal variables or functions are defined. 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 output the message `"clear ws"`.

`)erase list`

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.

`)save xxx`

This command causes APL to write its internal workspace into a UNIX file. This allows the current session to be resumed at a later time. If the save is successful, APL will output the date and time.

`)vsave xxx`

This command allows parts of a workspace to be saved. The functions and variables which are specified will be saved in a UNIX file in the same format as produced by `")save"`. APL will prompt for the names of the functions and variables to be saved. When you have entered the last name, type a blank line to end the save operation. The workspace you have created with `")vsave"` may be loaded with `")load"` at some later time. `")vsave"` does not affect variables in the internal workspace.

`)load xxx`

This command is used to tell APL to load the UNIX file `"xxx"` into APL 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.

`)copy xxx`

This command instructs APL to locate the UNIX file `"xxx"` and load it into its internal workspace, similar to the `")load"` command. 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.

`)digits n`

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 answer with the number of digits it was using.

`)origin n`

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, APL\11 allows any integer value. APL will answer with the origin it was using.

`)width n`

This command tells APL to print n characters per line. This is useful to keep output from being printed outside of the physical terminal width. Lines longer than this length will be "wrapped-around". APL answers this command with the previous terminal width.

`)off`

This command does the same thing that a control-D does - it terminates the APL session.

`)continue`

`")continue"` is a combination of the `")save"` and `")off"` commands. The internal workspace is saved in a file named `"continue"`, and then APL is terminated. Since APL will by default look for the file "continue" in the current directory when it is next run, this provides a convenient method of suspending and resuming an APL session.

`)fns`

This command causes APL to list the names of all of the functions which are defined in its internal workspace.

`)vars`

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

`)lib`

This command is similar to the UNIX `"ls"`. It causes APL to list the names of all of the UNIX files in the current directory. Long file names tend to result in ugly output; I modified the existing code to use more contemporary (read file system independent) methods to get the names, but didn't change it in any other ways. Names are truncated to 14 characters and don't always line up nicely in columns.

`)editf xxx`

This command is used to create and edit functions. If the function named `"xxx"` exists in the workspace, APL will write it into a temporary file in `/tmp` and then will invoke the editor defined by the shell variable `EDITOR` to edit that file. If `EDITOR` is not defined, the default value `"vile"` is used. When you have finished editing the file, and you exit the editor, APL will come back and will read the function in from the temporary file.

`)edit xxx`

This command is similar to `")editf"` except that `"xxx"` is a UNIX filename. APL will execute the editor to edit the file named `"xxx"`, and when the editing is complete, APL will read that file into the workspace. The difference between `")edit"` and `")editf"` is that `")editf"` essentially edits functions directly from the workspace, while `")edit"` gets the

functions from a named file.

)read xxx

At times it is desirable to read a function which is stored in an ASCII file into the internal workspace without editing it. The *")read"* command causes APL to read the file named *"xxx"* into the workspace as a function. Note that *")read"* and *")load"* are **not** the same thing. *")load"* reads a complete new workspace into APL from a workspace-format file, while *")read"* reads a function from an ASCII file and adds it to the current workspace.

)write xxx

This command is the complement of *")read"*. It takes the function *"xxx"* from the current workspace and writes it to an ASCII file named *"xxx"*. This is useful for writing functions which will be *")read"* later into other workspaces or which will be printed on the line-printer. Note *")write"* and *")vsave"* are not the same thing, for *")write"* is used to write a function into an ASCII file and *")vsave"* saves a selected subset of the internal workspace in a workspace-format file.

)drop list

This command performs the same function as UNIX *"rm"*. The names of the files to be deleted should be separated by spaces or tabs. The files may be APL workspaces, ASCII files, or any other type of file.

)script xxx

This command places APL into a "protocol" mode. Following this command, APL will copy all input from the terminal and output to the terminal to the file *"xxx"*. Thus, *"xxx"* is a complete transcript of the APL session. To turn off the script file, type *")script off"*. The protocol file which is produced will contain all of the output produced by APL itself, but will, unfortunately, not contain any output produced by another process (such as the editor).

)trace

This command turns on APL's internal "trace" mode. When tracing is turned on, APL will report the function name and line number of each line in every function executed. Thus, the flow of execution from the start to the end of a run can be followed.

)untrace

This command turns off "trace" mode.

)si

This command is useful when something goes wrong. When an error occurs, the function that was executing is "suspended". The *")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.

)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.

)shell

This command is useful when it is desired to escape from the APL environment temporarily without having to save the current internal workspace, exit APL, and later re-enter APL and

reload the workspace. "*)shell*" invokes the UNIX shell indicated by the *SHELL* environment variable. If *SHELL* is not defined, the shell defaults to "*/bin/sh*". When you exit the shell, you return to APL.

)list xxx

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

)prws

This command causes APL to print the contents of the workspace in a readable format. Non-scalar variables are displayed along with their dimensions; functions are displayed as via the "*)list*" system command. While a workspace listing can be placed into a file by use of a script file, a more efficient means of generating a workspace listing is to use the program "*prws*".

)debug

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.

)code fn

This command causes APL to print the compiled internal code for the function "*fn*". This is also intended for APL system development and not for general use.

)memory

The result of this command is a report on current dynamic memory usage. The basic form is "n bytes in m blocks". These values represent memory in use rather than memory available. Available memory is not terribly meaningful when the process size can be increased on request until some system limit is reached.

Appendix C

APL\11 Quad Functions

The following quad functions are defined under APL\11:

`⎕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.

`⎕run 'unix command'`

The argument passed to `"⎕run"` is executed as a UNIX shell command.

`⎕fork xx`

`"xx"` is a dummy argument. A "fork" system call is performed. This quad function should be used by experienced UNIX users only and probably will be followed by a `"⎕exec"` quad function. The process-id of the child is returned to the parent; a zero is returned to the child.

`⎕exec matrix`

Takes a two-dimensional character matrix and formats it into a UNIX "exec" system call. The matrix passed as an argument must be two-dimensional, the rows must be ***zero*** terminated. This quad function should be used by experienced UNIX users only.

`⎕wait xx`

This quad function is used in conjunction with `"⎕fork"` - it returns a 3-element vector of information about a child process which has terminated. The first element is either the PID of a completed child process or -1 (no children). The second is the status of the dead PID and the last is the completion code. This quad function should be used by experienced UNIX users only.

`⎕exit code`

This quad function is used to terminate the execution of the current process, with the completion code `"code"`. It should be used to terminate child processes and can be used to terminate an APL session; however, it is recommended that `"⎕exit"` be used by experienced UNIX users only.

`⎕pipe xx`

This quad function can be used to set up a pipe (used for interprocess communication) It returns a 2-element vector containing the two "pipe" file descriptors.

`⎕chdir 'directory'`

This quad function can be used to change APL to another directory. Normally, APL runs in the directory that you were in when it was started by the `"apl"` command. This function changes APL to another directory. The argument is a character vector specifying the new directory (there is NO way to default this). A 0 is returned if the "chdir" was successful; a -1 is returned if it failed.

`mode ⎕open 'file'`

This function is dyadic. It opens a UNIX file for use by an APL function with calls via

"⎕*read*" and "⎕*write*". The first argument is the mode for the open (0=read, 1=write, 2=read/write). The second argument is a character vector containing the file name. The file descriptor of the opened file is returned (-1 for error).

⎕*close fd*

This function complements "⎕*open*". The argument is the file descriptor of a UNIX file to be closed. This function returns 0 for success or -1 for failure.

mode ⎕*creat 'file'*

This function creates a UNIX file. If the file already exists, it is truncated to zero length. The creation mode is specified as the first argument (see CHMOD(I) for mode description). The filename is specified in a character vector as the second argument. The file descriptor of the created file (or -1 for error) is returned.

fd ⎕*read nbytes*

This function reads a specified number of bytes from a designated file. The first argument is the file descriptor; the second is the number of bytes to be read. The data which is read is returned. Note that the returned vector is always character data - to convert to numeric format see the function "⎕*float*".

fd ⎕*write data*

This function writes data to a specified file. The first argument is the file descriptor; the second is the data to be written. The number of bytes written is returned as the count. Any type of data (character or numeric) may be written in this manner.

⎕*seek (fd,pos,mode)*

This function executes the "seek" system call on a UNIX file. The argument to "⎕*seek*" is a three-element vector containing the file descriptor, seek offset, and mode (see SEEK (II)). A 0 is returned for a successful seek; -1 is returned if an error is detected.

pid ⎕*kill signal*

This function executes the "kill" system call. The first argument specifies what process is to be signalled. The second argument specifies what signal is to be sent. A 0 is returned for a successful "kill"; -1 is returned if the specified process could not be found or is not owned by the current user. For more information on signals, see KILL (II).

⎕*rd fd*

This function reads one line from the file descriptor specified. If the line is completely blank, a null string is returned. An end-of-file will also return a null string. Otherwise, the returned value is the character string which was read.

⎕*rm 'filename'*

The specified file will be removed, equivalent to)*drop filename*. A 0 is returned for a successful remove; -1 is returned if the file could not be removed or does not exist.

⎕*dup fd*

This function executes the "dup" system call. It returns an integer number which may be used as a file descriptor on later I/O calls. The new file descriptor is a duplicate of the argument "*fd*". If the argument file descriptor could not be duplicated, -1 is returned.

fd ⎕*ap 'string'*

This quad function is used to append a character string onto the end of a UNIX file. The first

argument specifies the file descriptor of the file (which should have been opened earlier). The second argument is a character array which is to be appended. A carriage return is automatically appended to the end of each row of the character array when it is appended to the end of the file.

⎕rline fd

This quad function is identical to "*⎕rd*", described above.

⎕nc 'arg'

This function can be used to determine what type of variable an APL symbol is. The apl symbol must be specified inside quote marks, as shown. The returned value will be:

- 0 -- symbol is undefined
- 2 -- symbol is a label or variable
- 3 -- symbol is a function name
- 4 -- unknown type (should not occur)

⎕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.

signal ⎕sig action

This quad function allows signal processing to be turned on and off under APL function control. The first argument is the signal whose processing is to be changed. The second argument specifies how the signal will be processed - if zero, the signal will cause termination of APL and a possible core dump; if non-zero, the signal will be ignored. Note that the special way in which interrupts and other signals are processed by APL is turned off by a call to "*⎕sig*" and cannot be turned back on. A -1 is returned on error, a positive number or zero for success.

⎕float charvect

This quad function is useful in conjunction with "*⎕write*" and "*⎕read*". While any type of data may be written to a UNIX file with "*⎕write*", when it is read with "*⎕read*" it will be interpreted as character data. This function will convert a character array into numeric form. The array must be a multiple of 4 elements long for *apl2* and 8 for *apl*. The converted array is the returned value.

Appendix D

I-Beams

The following monadic i-beam functions are available:

20 This i-beam returns the time-of-day as the total number of 1/60 seconds which have elapsed since midnight. Division by 60 gives the number of seconds since midnight, etc.

21 This i-beam returns the total amount of CPU time used by the current APL session in 1/60 seconds. This includes the amount of time spent by the system performing I/O (sys time) and computational time ("user" time).

22 This i-beam returns the number of 8-bit bytes which are left in the workspace. However, this value is not really meaningful since the in-core workspace will be expanded, if possible, when full.

24 This i-beam returns the time-of-day (in 1/60 seconds) when the current APL session was begun.

25 This i-beam returns the current date as a 6-digit number of the form fImmddyyfR. Thus, February 23, 1978 would be 022378.

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 in pending functions (functions which called the current function and are waiting for its completion).

28 This i-beam returns the date (as a 6-digit number, fImmddyyfR) when the current APL session began.

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.

32 This i-beam returns the number of workspace bytes which are being used. It is the complement of i-beam 22, which tells how many bytes are unused. Thus, the maximum workspace size (in bytes) can be calculated by adding i-beams 22 and 32.

36 This i-beam returns the second element of the vector

returned by i-beam 27 -- that is, it returns the line number of the function which called the current function.

40 This i-beam returns the total amount of CPU time spent by any child processes of APL. Children of APL include the editor, the shell if ")shell" is used, anything run by "Lrun", and any processes executed using "Lfork".

41 This i-beam returns the total amount of "user" time spent by APL and all of its children.

42 This i-beam returns the total "system" time spent by APL and all of its children.

43 This i-beam returns the total amount of "user" time (computational time) spent by APL.

44 This i-beam returns the total amount of "sys" (I/O and other system calls) time spent by 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-looking display.

97 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.

98 This i-beam function turns off the trace of all memory allocations and deallocations which i-beam 99 turns on. It returns a 1 if the trace was on, and a 0 if it was off already.

99 This i-beam turns on the alloc/free trace mentioned above, which i-beam 98 turns off. It also returns a 1 if the trace was already on, or a 0 if it was off.

The following dyadic i-beam functions were implemented to compensate for the lack of sufficient quad variables. 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.

34 This i-beam implements the system "nice" function. The "nice" of the APL process will be set to the value specified by the left argument. A zero will be returned for success, a -1 is returned for failure. This is intended for background processing, not interactive use of APL.

35 This i-beam implements the system "sleep" function. APL will suspend itself for the number of seconds specified (by the left argument). The value returned is the value of the left argument.

63 This i-beam implements the system "empty" function. The left argument specifies a file descriptor of a pipe. If the pipe is empty, a 1 will be returned, if not-empty, a 0 will be returned. A -1 will be returned if the file descriptor is illegal or is not a pipe.

90 Normally, APL will not exit if it reads an end-of-file from a terminal. This safety feature may be disabled by using i-beam 90 with a zero left argument. A non-zero right argument restores the requirement that the user exit with ")off" or ")continue". [Note: this feature cannot be enabled if the input device is not a terminal.]

99 This i-beam causes the buffered version of APL to flush the buffer associated with the file descriptor specified as the left argument. A 0 is returned for success, -1 for failure. In unbuffered APL, 0 is always returned and no action is taken (since the command is meaningless). Note that flushing a pipe used for input may cause information to be lost.