# Contents

1 Introduction ................................................. 3

2 Features .................................................... 3

3 The APL Characters - Transliteration ......................... 4
   3.1 Keywords .................................................. 5
      3.1.1 Declarations .......................................... 5
      3.1.2 Flow Control ......................................... 6
   3.2 Functions .................................................. 6
   3.3 System Variables .......................................... 8
   3.4 System Functions ......................................... 9
      3.4.1 Numeric/Conversions .................................. 9
      3.4.2 Text .................................................. 9
      3.4.3 Workspace/Memory Information/Management .......... 9
      3.4.4 Environmental Interaction ......................... 10
      3.4.5 File Input/Output .................................. 11

4 Complex Numbers ............................................. 13
   4.1 (Ordinary) Complex Numbers ............................... 13
   4.2 Quaternions .............................................. 13
   4.3 Octonions .................................................. 15
   4.4 Alternate Multiplication Tables ......................... 16
      4.4.1 Alternate Quaternions ................................ 16
      4.4.2 Alternate Octonions ................................ 16
   4.5 Conversion Between Complex Representations ............ 17
      4.5.1 The Circular Function ............................... 17
      4.5.2 Array Conversion .................................... 18

5 Parenthetic Expressions and Arrays .......................... 20

6 Boxed Arrays .................................................. 21
   6.1 Structural Functions ...................................... 21
   6.2 Link ....................................................... 22
   6.3 Each ....................................................... 23
   6.4 Partition ................................................... 24
   6.5 Generalized Index ........................................ 26
7 Flow Control
7.1 If-Then-Else
7.2 For Loops
7.3 While Loop
7.4 Cond
7.5 Error Trapping

8 User Defined Programs
8.1 Function Definition
8.2 Direct Definition
8.3 Anonymous Functions
8.4 Operator Definition

9 Matlab Interface

10 Known Bugs

11 Known deficiencies

12 Variances from the ISO standard

13 Examples

14 Future Concepts/Ideas/Plans

15 Thanks

16 Questions

References
1 Introduction

This is a brief discussion of aplc, a program to translate APL into c, and hence (given a c compiler) to compile APL.

I won’t discuss the APL language here.

In 1988, T. Budd ([1]) designed an APL compiler, and wrote some experimental code. He released this code on the internet. While demonstrating compilation of APL, it was a bit buggy, and lacked many features required for solving real problems. This is really not a deficiency of the original code, as it was designed as an experiment, and wasn’t intended to be of practical use.

I came across the code, and wanting to learn about compilation, C, and APL compilation in particular, I started playing with the code. While playing, I added lots of the missing pieces, to the point where I and at least one other (J. B. W. Webber) use it for “real” work (whatever that is). Meanwhile I’ve hacked at just about every line of code, and added lots of stuff.

I’m not sure quite how far I’ll go with this. Budd himself did some newer work with a somewhat different architecture (ref...). I’ve thought a few times about starting over from scratch given what I’ve learned.

2 Features

Two major features of the compiler, and a major difference with standard APL, have to do with typing scoping, and the existence of declarations for these.

- Declarations are not necessary for compilation; the compiler will guess. This may not be what was desired.
- Different types (int and real) may be added etc. as in standard APL.
- Integer promotion to real does NOT occur automatically.
- Arrays may be entered in multiline form using ().
- Variables are either local to a function, or global to all.
- The semicolon notation of standard APL may be used to declare local variables (e.g. .dl r .is a F b;i;j)
- Simple one line direct definition.
- A variety of system functions for files.
- A variety of system functions for processes.
- All of the complex numbers (ordinary complex, quaternions, octonions).
- Some flow control constructs.
- Simple error trapping.
- User defined operators.
- Anonymous functions using {}.
- Built in operators and functions, and user defined operators and functions interact.
3 The APL Characters - Transliteration

Transliteration is necessary to represent the APL characters in ASCII. This is an interesting subject discussed elsewhere...

The best thing here to understand things is to look at \textit{apl.lex}. But here's a current list. Shown is the aplc transliteration, the APL character, and comments. In some cases multiple names are equivalent.

<table>
<thead>
<tr>
<th>APLC</th>
<th>APL</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>_</td>
<td>high minus; e.g. _1 for negative 1</td>
</tr>
<tr>
<td>@</td>
<td>@</td>
<td>comment</td>
</tr>
<tr>
<td>.diamond .dm</td>
<td>@</td>
<td>statement separator</td>
</tr>
<tr>
<td>.and,</td>
<td>^</td>
<td></td>
</tr>
<tr>
<td>.nand</td>
<td>^</td>
<td></td>
</tr>
<tr>
<td>.or</td>
<td>v</td>
<td></td>
</tr>
<tr>
<td>.lt, &lt;</td>
<td>&lt;</td>
<td></td>
</tr>
<tr>
<td>.le</td>
<td>&lt;=</td>
<td>less than or equal to</td>
</tr>
<tr>
<td>.eq, =</td>
<td>=</td>
<td></td>
</tr>
<tr>
<td>.ne</td>
<td>!=</td>
<td>not equal</td>
</tr>
<tr>
<td>.ge</td>
<td>&gt;=</td>
<td>greater than or equal to</td>
</tr>
<tr>
<td>.gt, &gt;</td>
<td>&gt;</td>
<td></td>
</tr>
<tr>
<td>.assign, .is</td>
<td>=</td>
<td></td>
</tr>
<tr>
<td>#, .quad .bx</td>
<td>,</td>
<td>(.bx still recognized, but may go away)</td>
</tr>
<tr>
<td>.de, .decode</td>
<td>\downarrow</td>
<td>base value, down-tack</td>
</tr>
<tr>
<td>.dl, .del</td>
<td>\downarrow</td>
<td></td>
</tr>
<tr>
<td>.dq, .domino</td>
<td>\downarrow</td>
<td></td>
</tr>
<tr>
<td>.da, .drop</td>
<td>\downarrow</td>
<td></td>
</tr>
<tr>
<td>.gwdrop</td>
<td>\downarrow</td>
<td></td>
</tr>
<tr>
<td>¦</td>
<td>\downarrow</td>
<td></td>
</tr>
<tr>
<td>.en, .encode</td>
<td>\top</td>
<td>representation, up-tack</td>
</tr>
<tr>
<td>.ep, .epsilon,</td>
<td>\in</td>
<td>member of or element of</td>
</tr>
<tr>
<td>.do, .execute, .xq</td>
<td>\downarrow</td>
<td>hydrant</td>
</tr>
<tr>
<td>.fm, .format</td>
<td>\top</td>
<td>thorn</td>
</tr>
<tr>
<td>.go, .goto</td>
<td>\rightarrow</td>
<td>right arrow</td>
</tr>
<tr>
<td>.ind, .index</td>
<td>{, }</td>
<td>generalized indexing</td>
</tr>
<tr>
<td>.io, .iota</td>
<td>-downarrow</td>
<td>stile</td>
</tr>
<tr>
<td>.lo, .circle</td>
<td>\uparrow</td>
<td>up-stile</td>
</tr>
<tr>
<td>.match</td>
<td>\equiv</td>
<td>match, identical</td>
</tr>
<tr>
<td>.pa, .part</td>
<td>\downarrow</td>
<td></td>
</tr>
<tr>
<td>%, .div, .divide</td>
<td>\downarrow</td>
<td></td>
</tr>
<tr>
<td>.exp, ^</td>
<td>*</td>
<td>bang</td>
</tr>
<tr>
<td>.fl, .floor</td>
<td>\downarrow</td>
<td>down-stile</td>
</tr>
<tr>
<td>.lg, .log</td>
<td>\log</td>
<td></td>
</tr>
<tr>
<td>. (apl2) partition</td>
<td>\downarrow</td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>#</td>
<td>tilde</td>
</tr>
</tbody>
</table>
3.1 Keywords

Here is a list of keywords recognized by the compiler.

3.1.1 Declarations

For a declaration these must be preceded by “:decl”. From the perspective of the aplc parser, these names can be used for variable or function names, however note that the code generation can still cause conflicts with C keywords.

First the classes:

- #global – a global variable; note that this is somewhat like the “extern” qualifier of C in that it implies that the storage for this variable is given in some other place.
- #fun – a function
- #op – an operator
- #scalar – a scalar, with rank is 0
- #vector – a vector, with rank 1
- #var – a variable

Next the types, which may also be referenced outside :decl statements as integers:

- #char – character type (ASCII)
- #bool – an int with range [0,1] (#bit is currently equivalent)
- #boxed – an enclosed type
- #int – as C int
- #real – as C double
- #complex – a complex number made using 2 doubles
• #quat – a complex number made using 4 doubles
• #oct – a complex number made using 8 doubles

3.1.2 Flow Control
Finally some new keywords distinguished via colon, for flow control, including error trapping:

• :catch
• :if
• :then
• :elseif
• :then
• :endif
• :for
• :do
• :endfor
• :while
• : endwhile
• :cond

3.2 Functions
Here’s a list of current features, in alphabetical order. Some things were completely absent before I added them (sws). (*) means tested with (at least) the ISO standard examples.

• assign, reassign
• binomial/factorial/gamma (sws)*
• box (<) (sws)
• catenate
• catenate with axis (sws) including laminate
• ceiling
• compression
• deal
• dformat
• decode*
• diamond - not this changes line numbers
• divide quad (solution to linear equations, inverse - LAPACK QR) (sws)
• drop (sws)
• each (sws)
- encode (sws)*
- execute (sws) .xq string (only does conversion to numeric)
- expansion*
- factorial (sws)*
- floor
- format (sws)*
- gradeup
- gradedown
- gwdrop - the Guibas and Wyatt version, no overdrop
- gwtake - the Guibas and Wyatt version, no overtake
- identity for floor, ceiling (largest and smallest numbers) (sws)
- index *
- inner product
- iota (monadic)
- iota (dyadic)
- laminate (sws)
- lamp (comment)
- link create boxed list (sws)
- match check for object equality (sws)
- member of *
- outer product
- quad
- quote quad (sws) *
- reduction
- residue
- reverse *
- roll
- rotate *
- rho
- scalar functions (circle, +, -, *, %, ^ , !)
- scan (ceiling, floor)
- subscripted assign (sws)
- take (sws)
- transpose*
- unbox (> ) (sws)
3.3 System Variables

- #args - call line arguments (argv) as a matrix
- #io - index origin. Note that this currently can’t be localized, and should be set once at the start of the file if changed from the default (1).
- #outmap - changes output and error streams, a vector of length 2, which refers to the direction of (output, error); for each component:
  
  0 means stdout
  1 means stderr
  
  The default is 0 1.
- #pp - printing precision
- #prng - set the pseudo-random number generator
  
  0 use system rand/srand.
  1 use George Marsaglia’s mother of all random number generators.
- #rl - random link; the seed for the pseudo-random number generator.
- #tcbel ascii bel character, for terminal control
- #tcbs bs backspace
- #tccr cr carriage return
- #tcdel del
- #tcfcf ff
- #tcht ht tab
- #tcff lf linefeed
- #tcnul nul
- #ts timestamp (year, month, day, hour, min, sec, ms)
- #jts: Julian time in seconds
3.4 System Functions

3.4.1 Numeric/Conversions

- #za: (Ambivalent) convert a real or integer array to a complex (quaternion or octonion)
  - Monadic: the last axis of the array must be of dimension 2,4, or 8.
    - #za 10 20 < - > 10i20
  - Dyadic: last axis is sorted according to left
    - left size must be first right shape
    - 1 2 #za 10 20 < - > 10i20
    - 2 1 #za 10 20 < - > 20i10

- #az: (Ambivalent) convert a complex (quaternion or octonion) array to a real array
  - Monadic: the real array has an extra last dimension of 2, 4, or 8.
    - #az 10i20 < - > 10 20
  - Dyadic: left selects complex components
    - 1 2 #az 10i20 < - > 10 20
    - 1 2 2 #az 10i20 < - > 10 20 20

- #fi: (Monadic) string to number conversion, quite liberal. Includes integers, floats (decimal or e/E or d/D) and sign represented as (-, ng, +).

- #vi: (Monadic) string to number conversion validation. Shows which results of #fi are valid numbers.

3.4.2 Text

- #ss: (Dyadic) Stringsearch, as in STSC's APL
  
  i .is text #ss string

  Here is an example:

  #.is a .is 'a to t o btog'
  a to t o btog
  j .is a #ss 'to'
  a, .fm ((.rho a),1) .rho j
  a 0
  0
  t 1
  o 0
  0
  t 0
  0
  o 0
  0
  b 0
  0
  t 1
  o 0
  g 0
  0

3.4.3 Workspace/Memory Information/Management

- #free: (Monadic) free's the space allocated to variable A
  
  #free A

- #type: (Monadic) gives the type of variable A (doesn’t work for numbers yet). For example the type of an integer variable is #int etc
  
  #type A
3.4.4 Environmental Interaction

- **#pipe**: (Dyadic) Pass data to a set of shell commands, and then back to APL.

  ```apl
  r .is cmd #pipe data
  ```

  Dyadic pipe offers a convenient and easy way of passing character data from Apl through a pipeline of shell commands, and back into Apl. Something to watch out for, is that the data flow in Apl is right to left, but in a pipeline of shell commands it is left to right:

  ```apl
  Sorted .is 'sort | pr' #pipe Unsorted
  ```

  Pipe may also be used to pass data out through a pipeline of commands, to file or the line printer, for example, (when it returns an empty vector).

  ```apl
  'sort | pr | lpr' #pipe Unsorted
  ```

  Pipe may be used to input file or other data through a pipeline of commands and into Apl:

  ```apl
  Text .is 'ls | wc' #pipe ''
  ```

- **#spawn**: (Ambivalent) spawn a shell command.

  ```apl
  Monadic  fd .is #spawn 'shellcmd'
  Dyadic    fd .is fdc #spawn 'shellcmd'
  ```

  Spawn a unix command, connect to stdin, stdout, etc. of spawned process. The right argument is the command pipeline. The left argument controls which file descriptors (fds) of the command are connected to as input or output; sets blocking/non-blocking I/O. Spawn has two modes, character vector or numeric scalar, for each fd.

  1. Character vector.

  ```apl
  {NN}{i|o}{b|n}
  ```

  is interpreted as

  ```apl
  {fd of command},{input|output},{blocking|non-blocking}.
  ```

  2. Numeric scalar. In this case we get a set of fds. If the argument is $K$, we get fds $0,...,K − 1$ from the predefined table.

      | N | equivalent char fdc |
      |---|---------------------|
      | 0 | 00ib                |
      | 1 | 01ob                |
      | 2 | 02ob                |
      | 3 | 03ib                |
      | 4 | 04ob                |
      | 5 | 05ib                |
      | 6 | 06ob                |
      | 7 | 07ib                |
      | 8 | 08ob                |
      | 9 | 09ib                |
      |10 | 10ob                |
The spawn returns integer pipe file descriptors for the current process. Here are some examples:

- Fds .is '00in02on' #spawn Cmd @ connects to stdin, stderr of Cmd.
- Fds .is 3 #spawn Cmd @ connects to stdin, stdout, stderr.

- #system: executes system commands
  r .is #system cvec

3.4.5 File Input/Output

- #append: append data to a file; will create file if it does not exist already
  'name' #append cvec, or " for stdio (sws)
  (fd, count, type, size, sign, startbyte, skip) #append

- #close: close a file;
  #close fd

- #fcntl: change status of existing FileDescriptor(s):
  Res .is Modes #fcntl FileDescriptor(s)
  (char) Modes
  (int) FileDescriptor
  change status of existing FileDescriptor(s):
  Modes :
  a - append
  n - non-blocking
  + - turn on :
  - - turn off : one at a time only

- #lseek: go to a location in a file. Returns the resulting location in the file, bytes, or -1 for error.
  Whence #lseek (FileDescriptor, Offset)
  char Whence : 'b' - seek from file beginning
  : 'c' - seek from current position
  : 'e' - seek from file end

- #open: open a file
  fd .is #open 'filename'
  fd .is 'modes' #open 'filename' @ to open with modes
  fd .is n #open 'filename' @ to open with modes
  fd .is modes #open fd
  fd .is n #open fd

  modes meaning
  r read
  c create (+ write, but don’t overwrite)
  w write (overwrite if exists, create otherwise)
  a append to end of existing file, create otherwise
  cw c
  ca a
  n non-blocking read/write (ttys, pipes),
  affects all subsequent operations.

  default mode is read only. Default file permissions (for create) is 0644.

- #read: read components from a file or file descriptor
  Monadic: read characters from stream
  c .is #read stream
c .is #read 'name' @ read from file
  c .is #read " @ read from stdio
  c .is #read fd @ read from file descriptor
  c .is #read fd,count,type,size,sign,[startbyte,skip] @ binary read
Dyadic: read various binary types from a stream
  c .is (count,type,size,sign,[startbyte,skip]) #read fd @ binary read
    char read
      c .is count #read stream @ to read count components
      c .is (count,9) #read stream @ to read count components
    bool read
      c .is (count, 1) #read stream @ reads count bits into a boolean
    int read
      c .is (count, 4, size,sign #read stream @ reads count ints
        size is one of:
        0 short
        1 int
        2 long
        sign is one of:
        0 unsigned
        1 signed
    float read
      c .is (count, 5, size) #read stream @ read count reals
        size is one of:
        0 float
        1 double

• #write: write (chars) to a file
  'name' #write cvec
  " #write cvec @ writes to stdio
  fd #write cvec
  (fd, count, type, size, sign, startbyte, skip) #write x
4 Complex Numbers

There are actually three generalizations of numbers that can be based on square roots of $-1$, the ordinary complex numbers, quaternions, and octonions.

4.1 (Ordinary) Complex Numbers

Here I’ve followed the standard implementations, except that I’ve used “i” instead of “j”. The reason is partly that I prefer “i,” but mostly because due to quaternions, we need both. Note that “j” will still work just like “i” except that you’re really using a quaternion then.

Complex numbers are entered via “i” between the real and imaginary parts, much as “e” is used between the mantissa and exponent:

\[ a \text{.is } 0i1 \]

Complex number components are of double type. Most of the usual scalar operations apply as well to complex numbers, except for the logicals and the relations that require an ordering (e.g., no <). For complex numbers, the circular functions have been extended to the range $[-12, 12]$ in the usual way.

Dyadic format works simply for the complex types - the two format numbers specify the total width and the precision for each component.

Domino has not been generalized for the complex types yet.

4.2 Quaternions

Hamilton’s quaternions are extremely useful for describing rotations in 3 dimensions (actually unit quaternions are used). They are extensively used in 3d graphics, spacecraft dynamics and control, and in crystallography. They are similar to complex numbers, except that now there are 3 square roots of $-1$, the units i,j,k, perhaps familiar from vector mechanics.

\[ -1 = i \times i = j \times j = k \times k, \]

Note that these units are not comutative:

\[
\begin{align*}
  k &= i \times j \\
  i &= j \times k \\
  j &= k \times i \\
  -k &= j \times i
\end{align*}
\]

Quaternions behave in ways very similar to ordinary complex numbers, yet are obviously not comutative. They are entered exactly analogously to the ordinary complex numbers, and not all components are needed:

\[ a \text{.is } 1 0i1 0j1 0k1 1i2k3 1j2k4 \]

Here’s the quaternion multiplication table:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>0i1</th>
<th>0j1</th>
<th>0k1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0i1</td>
<td>0j1</td>
<td>0k1</td>
</tr>
<tr>
<td>0i1</td>
<td>0i1</td>
<td>-1</td>
<td>0k1</td>
<td>0j -1</td>
</tr>
<tr>
<td>0j1</td>
<td>0j1</td>
<td>0k -1</td>
<td>-1</td>
<td>0i1</td>
</tr>
<tr>
<td>0k1</td>
<td>0k1</td>
<td>0j1</td>
<td>0i -1</td>
<td>-1</td>
</tr>
</tbody>
</table>

Note that any of the quaternionic imaginary units could be used as a complex unit instead of $i$. The results are the same:
This is a pleasing design since some prefer $j$ as the imaginary unit.
4.3 Octonions

Cayley discovered these last generalized complex numbers. They consist of seven square roots of \(-1\). In addition to being non commutative, they are also non associative.

There are many choices of multiplication table for octonions. I've chosen a fairly standard one that includes the quaternions above. The 7 units used are \(i, j, k, U, I, J, K\). Note that \(E\) is usually used where I've substituted \(U\), for obvious reasons.

Here’s the octonion multiplication table:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>0i1</th>
<th>0j1</th>
<th>0k1</th>
<th>0U1</th>
<th>0I1</th>
<th>0J1</th>
<th>0K1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0i1</td>
<td>0j1</td>
<td>0k1</td>
<td>0U1</td>
<td>0I1</td>
<td>0J1</td>
<td>0K1</td>
</tr>
<tr>
<td>0i1</td>
<td>0i1</td>
<td>−1</td>
<td>0k1</td>
<td>0j−1</td>
<td>0I1</td>
<td>0U−1</td>
<td>0K−1</td>
<td>0J1</td>
</tr>
<tr>
<td>0j1</td>
<td>0j1</td>
<td>0k−1</td>
<td>−1</td>
<td>0i1</td>
<td>0J1</td>
<td>0K1</td>
<td>0U−1</td>
<td>0I−1</td>
</tr>
<tr>
<td>0k1</td>
<td>0k1</td>
<td>0j1</td>
<td>0i−1</td>
<td>−1</td>
<td>0K1</td>
<td>0J−1</td>
<td>0I1</td>
<td>0U−1</td>
</tr>
<tr>
<td>0U1</td>
<td>0U1</td>
<td>0I−1</td>
<td>0J−1</td>
<td>0K−1</td>
<td>−1</td>
<td>0i1</td>
<td>0j1</td>
<td>0k1</td>
</tr>
<tr>
<td>0I1</td>
<td>0I1</td>
<td>0U1</td>
<td>0K−1</td>
<td>0J1</td>
<td>0i−1</td>
<td>−1</td>
<td>0k−1</td>
<td>0j1</td>
</tr>
<tr>
<td>0J1</td>
<td>0J1</td>
<td>0K1</td>
<td>0U1</td>
<td>0I−1</td>
<td>0j−1</td>
<td>0k1</td>
<td>−1</td>
<td>0i−1</td>
</tr>
<tr>
<td>0K1</td>
<td>0K1</td>
<td>0J−1</td>
<td>0I1</td>
<td>0U1</td>
<td>0k−1</td>
<td>0j−1</td>
<td>0i1</td>
<td>−1</td>
</tr>
</tbody>
</table>

Note that any of the octonionic imaginary units could be used as a complex unit instead of \(i\). The results are the same. Hence one can choose any of the common choices for a complex imaginary unit \((i, j, I, J)\). Of course internally they are somewhat different.

Note that the set \((i, j, k)\) are the ordinary quaternions, however the following groups behave the same way:

\((i,U,I)\) \((i,K,J)\)
\((j,U,J)\) \((j,I,K)\)
\((k,U,K)\) \((k,J,I)\)

\(112j3k4%413j2k1\)

0.6666667i0.3333333k0.6666667

\(1i2U14I413U211\)

0.6666667i0.3333333I0.6666667

While some subsets of the octonions behave as quaternions, octonions in general are not associative. Here are some examples, which demonstrate that octonions are not associative:

\[a \cdot b \cdot c\]
\[-1 \quad 0U-1\]
\[a \cdot (b \cdot c)\]
\[-1 \quad 0U-1\]
\[(a \cdot b) \cdot c\]
\[-1 \quad 0U1\]
4.4 Alternate Multiplication Tables

For quaternions and octonions, since they are not commutative, the order of multiplication matters. As the default I’ve chosen the standard order. One could however interpret “i*j” in an APL order as right-to-left, equivalent to normal mathematical notation “j*i.” I’ve experimented with this ordering, and it is available via #define RLMORDER 1 in runfun_res.c.

4.4.1 Alternate Quaternions

Using an APL-ish right-to-left convention, we have:

\[-1 = i \times i = j \times j = k \times k,\]

\[k = j \times i\]
\[i = k \times j\]
\[j = i \times k\]
\[−k = i \times j\]

Here’s the alternate quaternion multiplication table:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>0i1</th>
<th>0j1</th>
<th>0k1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0i1</td>
<td>0j1</td>
<td>0k1</td>
</tr>
<tr>
<td>0i1</td>
<td>0i1</td>
<td>−1</td>
<td>0k1</td>
<td>0j1</td>
</tr>
<tr>
<td>0j1</td>
<td>0j1</td>
<td>0k1</td>
<td>−1</td>
<td>0i1</td>
</tr>
<tr>
<td>0k1</td>
<td>0k1</td>
<td>0j −1</td>
<td>0i1</td>
<td>−1</td>
</tr>
</tbody>
</table>

4.4.2 Alternate Octonions

Here’s the octonion multiplication table, using right-to-left order:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>0i1</th>
<th>0j1</th>
<th>0k1</th>
<th>0U1</th>
<th>0I1</th>
<th>0J1</th>
<th>0K1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0i1</td>
<td>0j1</td>
<td>0k1</td>
<td>0U1</td>
<td>0I1</td>
<td>0J1</td>
<td>0K1</td>
</tr>
<tr>
<td>0i1</td>
<td>0i1</td>
<td>−1</td>
<td>0k −1</td>
<td>0j1</td>
<td>0I −1</td>
<td>0U1</td>
<td>0K1</td>
<td>0J −1</td>
</tr>
<tr>
<td>0j1</td>
<td>0j1</td>
<td>−1</td>
<td>0i −1</td>
<td>0J −1</td>
<td>0K −1</td>
<td>0U1</td>
<td>0I1</td>
<td>0j1</td>
</tr>
<tr>
<td>0k1</td>
<td>0k1</td>
<td>0j −1</td>
<td>0i1</td>
<td>0k −1</td>
<td>0i −1</td>
<td>0j −1</td>
<td>0k −1</td>
<td>0i1</td>
</tr>
<tr>
<td>0U1</td>
<td>0U1</td>
<td>0I1</td>
<td>0J1</td>
<td>0K1</td>
<td>−1</td>
<td>0i −1</td>
<td>0j −1</td>
<td>0k −1</td>
</tr>
<tr>
<td>0I1</td>
<td>0I1</td>
<td>0U −1</td>
<td>0K1</td>
<td>0J −1</td>
<td>0i −1</td>
<td>−1</td>
<td>0k1</td>
<td>0j −1</td>
</tr>
<tr>
<td>0J1</td>
<td>0J1</td>
<td>0K −1</td>
<td>0U −1</td>
<td>0I1</td>
<td>0j1</td>
<td>0k −1</td>
<td>−1</td>
<td>0i1</td>
</tr>
<tr>
<td>0K1</td>
<td>0K1</td>
<td>0J1</td>
<td>0I −1</td>
<td>0U −1</td>
<td>0k1</td>
<td>0j1</td>
<td>0i −1</td>
<td>−1</td>
</tr>
</tbody>
</table>

16
4.5 Conversion Between Complex Representations

It is of course a necessity to convert between various representations of complex numbers. The circular function has traditionally been used for this.

4.5.1 The Circular Function

The usual extensions to the circular functions, with domain $[-7, 7]$ for reals, and $[-12, 12]$ for the complex types.

\[
\begin{align*}
-12 & \quad \exp(i \times r) \\
-11 & \quad i \times r \\
-10 & \quad +r \\
-9 & \quad r \\
-8 & \quad \sqrt{1 - r^2} \\
-7 & \quad \arctanh(r) \\
-6 & \quad \text{arcosh}(r) \\
-5 & \quad \text{arcsinh}(r) \\
-4 & \quad \sqrt{1 + r^2} \\
-3 & \quad \arctan(r) \\
-2 & \quad \text{arccos}(r) \\
-1 & \quad \text{arcsin}(r) \\
0 & \quad \sqrt{1 - r^2} \\
1 & \quad \sin(r) \\
2 & \quad \cos(r) \\
3 & \quad \tan(r) \\
4 & \quad \sqrt{1 + r^2} \\
5 & \quad \sinh(r) \\
6 & \quad \cosh(r) \\
7 & \quad \tanh(r) \\
8 & \quad \sqrt{1 - r^2} \\
9 & \quad \text{real}(r) \\
10 & \quad |r| \\
11 & \quad \text{imag}(r) \\
12 & \quad \text{arc}(r)
\end{align*}
\]

Conversion between real, imaginary parts (rectangular) can be done using left arguments (9,11), as shown in the following:

```plaintext
#.is ez .is (1  2i4  3.1i9
     -4i1.1  5i2  6.25i3)
  1 2i4  3.1i9
4i1.1  5i2  6.25i3
#.is ezR .is 9 .lo ez
   1 2  3.1
  4 5  6.25
#.is ezI .is 11 .lo ez
   0 4  9
1.1 2  3
```

\text{ezR,[0.5]} ezI

\begin{align*}
1 & 2  3.1 \\
4 & 5  6.25 \\
0 & 4  9 \\
1.1 & 2  3
\end{align*}
Note that having a separate left number for each component isn't so practical for quaternions and octonions. I would have to add 6 more numerical arguments, and their negatives for the reverse.

### 4.5.2 Array Conversion

It seems more natural to allow an array representation of the complex types, so I've added this to aplc. One can represent a complex number as 1i2, or the array 12. This is more convenient for quaternions and octonions. The system functions #az, #za facilitate this conversion.

```apl
#za 1 2
1i2
#az 1i2
1 2
#.is zv .is #za (1 0
  2 1
  3 2)
1 2i1 3i2
#az zv
1 0
2 1
3 2
ez
  1 2i4  3.1i9
4i1.1 5i2 6.25i3
#az ez
  1 0
  2 4
3.1 9
  4 1.1
  5 2
6.25 3

ez - #za #az ez
0 0 0
0 0 0
```

This formulation is also practical for the quaternions and octonions:

```apl
eo
  1 2i4  3.1j9
4i1.1 5i2U6 6.25j3K5
#.is ea .is #az eo
  1 0 0 0 0 0 0 0
  2 4 0 0 0 0 0 0
3.1 0 9 0 0 0 0 0
  4 0 0 0 1.1 0 0
  5 2 0 0 6 0 0 0
```
The dyadic forms allow pulling out or adding particular complex axes as real slices.

1 2 #az 10i20
10 20
2 #az 10i20
20
1 2 #za 10 20
10i20
2 1 #za 10 20
20i10
eq
  1 2i4 3.1i4j9
  4i1.1 5k2 6.25i3j2k10
2 #az eq
  0
  4
  4

1.1
  0
  3
1 3 #az eq
  1 0
  2 0
  3.1 9
  4 0
  5 0
6.25 2
## 5 Parenthetic Expressions and Arrays

I've extended parenthetic expressions to include direct input of arrays, for example

```plaintext
b .is (1 2 3
4 5 6)
```

produces an array of shape 2 1 3 directly, without using reshape. This can also be done with expressions, for example

```plaintext
bb .is (1 .link 'a'
'x' .link 3
1000 .link 'xxx')
```

```
+-----+---+
| 1   |a |
+-----+---+
|x    | 3 |
+-----+---+
|1000|xxx|
+-----+---+
```

or

```plaintext
I .is 2 2 .rho 3 .take 1
cc .is ((0*I), I
(-I*3), 8*I)
```

```
0 0 1 0
0 0 0 1
-3 0 8 0
0 -3 0 8
```
6 Boxed Arrays

I've added a start at boxed arrays. Boxing creates a new type. One can create them using the "less-than" symbol.

```
<1 2
+-----+
| 1 2 |
+-----+
#.is a .is <'x'
+++
|x|     
+++
<a   
+++++
|++++|
||x||   
|++++|
+++++
```

The opposite symbol, "greater-than" reverses the process, returning the original. It has no effect on a normal array.

```
>a
x
>>a
x
```

6.1 Structural Functions

Catenate works on the boxed type as usual, allowing the creation of heterogeneous structures.

```
a,<2 3 4
+-------+
|x| 2 3 4|
+-------+
```

Unboxing structures requires bringing them to

```
#.is c .is (<10),(<2 3 4 5),<3 1 .rho 1I3
+-----------------------+
| 10| 2 3 4 5| 1I3|
|   | 1I3| 1I3|
|   | 1I3| 1I3|
+-----------------------+
```

```
>c
10 0 0 0
0 0 0 0
0 0 0 0
2 3 4 5
0 0 0 0
0 0 0 0
1I3 0 0 0
1I3 0 0 0
1I3 0 0 0
```
Other structural primitives also work as expected.

\[ c[1] \]
\[ +---+ \]
\[ | 10| \]
\[ +---+ \]

### 6.2 Link

\[
1 \text{ .link 2 3 }
\]
\[ +-----------+ \]
\[ | 1| 2 3| \]
\[ +-----------+ \]

\[
1 \text{ .link 2 3 .link 5 5 5 }
\]
\[ +-----------+ \]
\[ | 1| 2 3| 5 5 5| \]
\[ +-----------+ \]

\[
1 \text{ .link 2 3 .link (5 5 5} \]
\[ 5 5 5) \]
\[ +-----------+ \]
\[ | 1| 2 3| 5 5 5| \]
\[ | | | 5 5 5| \]
\[ +-----------+ \]
6.3 Each

Each (') works to distribute functions over them. Note that currently it only works with user defined functions.

```plaintext
.dl r .is fn x
  r .is 2*x
.dl

.dl r .is x gn y
  r .is x*y
.dl

  a .is (<2 3),<100
  a
  +--------+
  | 2 3| 100|
  +--------+

  fn 1111
  2222
  fn " 1111
  +------+
  | 2222|
  +------+

  fn " a
  +--------+
  | 4 6| 200|
  +--------+

  a gn" <1000
  +----------------+-
  | 2000 3000| 100000|
  +----------------+-

  (<<2000) gn" a
  +----------------+-
  | 4000 6000| 200000|
  +----------------+-

  a gn" a
  +--------+
  | 4 9| 10000|
  +--------+
```
6.4 Partition

Partition works much like the APL2 function (not the same as cut).

```
1 1 2 .part 'abc'
+--+-+
|ab|c|
+--+-+
#.is a .is 'abc'
abc
1 1 2 .part a
+--+-+
|ab|c|
+--+-+
#.is saves .is ' a stitch in time'
 a stitch in time
(saves .ne ' ') .part saves
+-----------------------+
|a|stitch|in|time|
+-----------------------+
#.is m .is .fm 3 3 .rho 1 10 3.142 2 100 6.283 3 1000 9.425
 1 10 3.142
 2 100 6.283
 3 1000 9.425
#.is mp .is (~.and/[1]m = ') .part m
+--------+
|1| 10|3.142|
+--------+
|2| 100|6.283|
+--------+
|3|1000|9.425|
+--------+
#.is n .is 4 3 .rho .iota 12
 1 2 3
 4 5 6
 7 8 9
10 11 12
#.is np .is 1 0 1 .part n
+--------+
| 1 | 2 | 3 |
+--------+
| 4 | 6 | 8 |
+--------+
| 7 | 9 |
+--------+
|10|12|
+--------+
#.is np .is 1 0 1 1 .part[1] n
+-----------------------+
| 1 | 2 | 3 |
+-----------------------+
| 7 10| 8 11| 9 12|
+-----------------------+
#.is np .is 1 1 0 .part[2] n
+--------+
```
6.5 Generalized Index

a .is 10+1 2 3 4
4 .ind a
14
(2 1 .rho 1 3) .ind a
11 13
#.is b .is 2 3 .rho .lo .iota 6
3.141593 6.283185 9.424778
12.56637 15.70796 18.84956
2 .ind b
15.70796
(3 2 .rho 1 1 2 2 2 3) .ind b
3.141593 15.70796 18.84956
(1 1
2 2
1 2
2 3) .ind b
3.141593 15.70796
6.283185 18.84956
#.is c .is 2 3 4 .rho 1.1*.iota 24
1.1 2.2 3.3 4.4
5.5 6.6 7.7 8.8
9.9 11 12.1 13.2

14.3 15.4 16.5 17.6
18.7 19.8 20.9 22
23.1 24.2 25.3 26.4
(2 2 3 .rho 1 1 2 2 2 2 3 4 1 1 1) .ind c
2.2 19.8
26.4 1.1
.rho (.iota 0) .ind 110
0
1 .ind 110
110
1 1 1 .ind 110
110 110 110
7 Flow Control

I've implemented some simple constructs in addition to the traditional goto. The current keywords are distinguished by leading colons. Note that the first three of these (if, for, while) are currently implemented using the preprocessor to generate the usual gotos.

7.1 If-Then-Else

The keywords :if, :then, :elseif, :endif work as might be expected. This is implemented using the preprocessor. Note that other APLs use line separation or diamond instead of :then. Here are two examples

```
dl r .is tst a
  r .is 0
:if a>0 :then
  r .is a
:endif
:if a<0 :then r .is -1 :endif
.dl
.dl r .is tst a
  r .is 0
:if a>0 :then
  :if a>5 :then
    r .is a*a
  :else
    r .is a
  :endif
:elseif a<0 :then
  :if a<_5 :then
    r .is 25 - a*a
  :else
    r .is -1
:endif
:endif
.dl
```

7.2 For Loops

The :for keyword requires :do and :endfor, but may be on a single line or spread out as in the example. Note that other APLs use line separation or diamond instead of :do. For loops may be nested. This keyword is implemented using the preprocessor.

```
:for aa .is 1 2 3 :do
  b .is aa
  b
:endfor
```

7.3 While Loop

The :while keyword requires :do and :endwhile, but may be on a single line or spread out as in the example. Note that other APLs use line separation or diamond instead of :do. While loops may be nested. This keyword is implemented using the preprocessor.
:while i<r :do
  b .is b+i
  i,b
  i .is i+1
:endwhile

:while i<r :do b .is b+i .diamond i .is i+1 :endwhile

7.4 Cond

This works like Iverson’s direct definition conditional. There is one keyword, and 3 parts:

(case0) :cond (test) (case1)

If the test is 0, case0 is executed, else case1 is executed. Here’s an example similar to the Dictionary.

```
fn: .omega :cond (1=.alpha) ~.omega
  0 fn 0 1
  1 fn 0 1
  1 0
```

Cond syntax is like a function, initially grabbing the immediate expression to it’s left (before evaluation), hence parenthesis are advised for the left expression as in the following user-defined ambivalent exponential.

```
fnx: r .is ^.omega :cond (0<#type .alpha) .alpha^.omega

2 fnx 3
  2980.958
  ^2^3
  2980.958
```

```
fn: r .is (^ .omega) :cond (0<#type .alpha) .alpha^ .omega

2 fn 3
  8
```

7.5 Error Trapping

A simple error trapping mechanism has been added using the :catch keyword. If there’s an error in it’s left argument, :catch evaluates it’s right argument. Currently only aplc errors are caught. Here’s a simple example

```
.dl x .is test n
@ x .is .iota n
(x .is .iota n) :catch x .is 0
.dl

test 10
  1 2 3 4 5 6 7 8 9 10

test 10.1
  0
```
8 User Defined Programs

This includes functions, which may take variables as arguments, and operators, which may take functions and variables as arguments.

8.1 Function Definition

User defined functions are defined using .dl, or .dl. The definition is enclosed in the .dl’s, and includes possible local variable declarations.

```
.dl
  (header)
  (declaration statements)
  (program statements)
  .dl
```

A function header looks like one of the following, depending if it is monadic (only a right argument) or dyadic (arguments on both sides) and whether or not it returns a value.

```
  r .is a fun b
  r .is fun b
  r .is fun
  a fun b
  fun b
  fun
```

In addition, variables localized to the program may be listed using semicolons.

```
.dl r .is a fun b;c;d;e
...
.dl
```

Statements are usual APL expressions, and may be labeled. User-defined functions may be the arguments of built-in or user-defined operators.

8.2 Direct Definition

Direct definition is another way for the user to define functions. A function is defined on a single line. The left argument is .alpha, the right .omega. Note that direct definitions must appear in a global context (not inside functions as they have the same form as labels). Other variables present are global. Here’s a simple example

```
  glak: .alpha + 2*.omega
  1 glak 2
  5
```

Only this simple form of direct definition is available, however using :cond, the examples from Dictionary APL may be coded with trivial modification.

8.3 Anonymous Functions

These are apparently functions as values, without names. Of course the C code has a name created. These are created as expressions similar to direct definition, but enclosed in curly brackets ({}). Here is a simple example.
Note .apply is quite simple currently, only working with scalar functions.

8.4 Operator Definition

User defined operators are also defined using del, or .dl, and may have similar declaration statements. An operator header looks like one of the following, depending if it is monadic (only a right argument) or dyadic (arguments on both sides) for both variable arguments and function arguments, and whether or not it returns a value.

\[ r \ .is \ a \ (f \ uop \ g) \ b \quad a \ (f \ uop \ g) \ b \]
\[ r \ .is \ a \ (f \ uop ) \ b \quad a \ (f \ uop ) \ b \]
\[ r \ .is \ (f \ uop \ g) \ b \quad (f \ uop \ g) \ b \]
\[ r \ .is \ (f \ uop ) \ b \quad (f \ uop ) \ b \]

The “inner” operands \((f,g)\) are functions by default. However this maybe changed using exterior declarations of the operator. Here is a simulation of a monadic rank operator. The valence is declared as “#valence21,” indicating the form is inner dyadic/outer monadic \((f \ rankm g) x\), and the types of the (inner) operands are declared as “#operand21,” indicating that the left is a function and the right a value.

@ dyadic operator, monadic overall
@ inner operands FOV
:decl #op #valence21 #operand21 rankm

@ dyadic/monadic, acting on function,variable
@ simple rank for monadic fns
.dl d .is (fL rankm k) b
k3 .is .rv 3 .rho .rv ,k
k .is k3[1]
c .is k box b
@ apply fL to each cell
d .is > fL"c
.dl

This way the rank operator may be used as usual
.iota rankm(0) 1 2 3
 1 0 0
 1 2 0
 1 2 3

(Complete code for this and ambivalent rank is given in test_anon).

Operators are C functions, with either function or trs pointers as arguments. The compiler automatically
creates anonymous functions to handle situations such as

```c
+ uop a
```

The compiler attempts to re-use these, so if an anonymous “+” is used later as well only 1 anonymous function
is defined. A subtle issue is that operators may be chained on the left, causing multiple anonymous functions
e.g.

```c
fn uop1 uop2 a
```

the argument to `uop2` is the derived function `{fn uop1}`. The compiler creates such anonymous functions
automatically.

## 9 Matlab Interface

The Matlab interface (sws) This allows one to compile an APL function (dyadic or monadic) into a mex file
for dynamic linking with Matlab.

## 10 Known Bugs

The file must end with a newline for the parser or else it will complain.

## 11 Known deficiencies

- various system variables and functions could be added. For example #pw is not implemented (doesn’t
  seem to be much need)
- execute is restricted to numeric conversion. This is likely to stay, as an interpreter is necessary to really
  implement execute.
- bit type; While there is a distinction between int and bool, all are implemented as integers, wasting
  space.
- interprocudural analysis (inter)
- integer promotion - in APL, integers become reals if they get too big to be represented as integers.
  That doesn’t happen here.
- floats that are near integers may not be treated as such...

## 12 Variances from the ISO standard

- Input of integers in floating point format doesn’t work. E.g.

```apl
b .is 1e7
```

will make b a real (this cannot be overruled by declaration).
• Data changes due to computations - for example \( i.is i+1 \) in a loop should result in an integer becoming a real, but just overflows. Data changes due to (sub) assignments are handled.

• Base value (decode) does not allow character arguments. There doesn’t seem to be much point in this, but the standard allows

\[
\begin{align*}
\langle \, \rangle & . \text{de} \ 3 \\
0 & \\
\langle a \rangle & . \text{de} \ . \text{iota} \ 10 \\
0 &
\end{align*}
\]

• Deal - the left and right arguments to deal must be integers.

• Declarations - allowed to make compilation more efficient; not required class: #global, #fun, #var (unknown local variable) type: #bool (#bit), #char, #int, #real, #complex, #quat, #oct rank: #scalar, #vector examples -

\[
\begin{align*}
: \text{decl} & \ #\text{global} \ i,j \\
: \text{decl} & \ #\text{scalar} \ #\text{int} \ k \\
: \text{decl} & \ #\text{var} \ x \\
: \text{decl} & \ #\text{fun} \ \text{round}
\end{align*}
\]

• Diamond - increments line number for successive statements

• gwdrop, gwtake are the original versions of drop and take coded by T. Budd. They don’t handle the overtake or overdrop cases, where the left argument is larger than the right shape.

• Execute - only does conversion of character strings to numbers

• Format; there are a number of small differences in format.
  – C’s printf is used. This is probably not standardized with regards to rounding.
  – C’s printing of negative numbers has been left alone (as a convenience for use in other programs) so \( -1 \) prints as \( -1 \).
  – exponential format – the exponent is right justified, and padded with 0’s, and may have a + sign.
  – dyadic fixed format includes leading 0 before numbers less than 1
  – columns are not guaranteed uniform in style for monadic format

• Goto should only be used to point to line labels

• Lazy (demand driven) evaluation allows things such as

\[
0 \ 1/2 \ 3 \ #0 \ 4
\]

to work (0%0 need not be computed since it’s not asked for).

• Order of execution may vary from strict right-to-left, for example in reshape, the left argument may be evaluated first. also reduction for commutative functions.

• Roll takes only integer arguments (not near integers).

• Scoping rules are just global or local to a single function.

• Workspaces are not implemented.

• Quad input only allows numeric constants - no evaluation, escapes, it also will stop immediately given a NL (doesn’t keep asking).
- \#free is like \#ex, but takes as argument the identifier directly, rather than a character string, e.g. \#ex 'A' becomes \#free A
- \#type is like \#nc, but also takes the identifier directly \#nc 'A' becomes \#type A. Note that this can be used to implement ambivalent functions
- reduction using the comparison functions (e.g. \=/) may compile, but really only works on boolean arguments.
- Comparisons in .match don’t use the tolerance.
13 Examples

See the examples directory. There you should find a number of examples, and hopefully some useful code. Some of these output codes that use the xterm emulation of Textronix 4010 graphics (I first started apl on IBM selectics and Textronix 4013s).
<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>aa2aplc.apl</td>
<td>Convert aplascii notation to aplc</td>
</tr>
<tr>
<td>ashell.apl</td>
<td>Simple shell</td>
</tr>
<tr>
<td>convert.apl</td>
<td>Converts notation from Budd’s original to current</td>
</tr>
<tr>
<td>dd.apl</td>
<td>Direct definition, dyadic</td>
</tr>
<tr>
<td>ddm.apl</td>
<td>Direct definition, monadic</td>
</tr>
<tr>
<td>dec2hex.apl</td>
<td>Decimal to hex number</td>
</tr>
<tr>
<td>eis_bal.apl</td>
<td>Part of EISPACK - eigenvalues, eigenvectors of a general matrix</td>
</tr>
<tr>
<td>eis_orth.apl</td>
<td>Part of EISPACK - eigenvalues, eigenvectors of a general matrix (orthogonal path)</td>
</tr>
<tr>
<td>epslon.apl</td>
<td>Compute machine epsilon</td>
</tr>
<tr>
<td>fib.apl</td>
<td>fibonachi test</td>
</tr>
<tr>
<td>gcd.apl</td>
<td>Some simple gcd (greatest common divisor) functions</td>
</tr>
<tr>
<td>gutil.apl</td>
<td>Generalized APL utilities</td>
</tr>
<tr>
<td>gutilt.apl</td>
<td>Test cases</td>
</tr>
<tr>
<td>lifet.al</td>
<td>Test of Conway’s life</td>
</tr>
<tr>
<td>pipetest.apl</td>
<td>Test of #pipe</td>
</tr>
<tr>
<td>polyk.apl</td>
<td>Create the 3 regular polytopes in n-space</td>
</tr>
<tr>
<td>primes.apl</td>
<td>Compute primes</td>
</tr>
<tr>
<td>prompt.apl</td>
<td>Quote quad output test</td>
</tr>
<tr>
<td>rank.apl</td>
<td>Simulation of rank operator</td>
</tr>
<tr>
<td>rep.apl</td>
<td>Simulation of replicate operator</td>
</tr>
<tr>
<td>split.apl</td>
<td>Split vector into boxed components</td>
</tr>
<tr>
<td>tv.apl</td>
<td>Generate an html link list of a directory tree</td>
</tr>
<tr>
<td>ulam.apl</td>
<td>S. Ulam’s spiral of primes</td>
</tr>
<tr>
<td>vtm.apl</td>
<td>Convert vector to matrix</td>
</tr>
<tr>
<td>gamma.apl</td>
<td>Compute and draw (in an xterm) a measure polytope (hypercube) in the input dimension space</td>
</tr>
<tr>
<td>lifex.al</td>
<td>Test of Conway’s life</td>
</tr>
<tr>
<td>polyn.apl</td>
<td>Create/draw the 3 regular polytopes in n-space</td>
</tr>
</tbody>
</table>
14 Future Concepts/Ideas/Plans

1. allow memory declaration
2. reduce use of mallocs
3. remove switch machinery if not used
4. more functions for nested/boxed arrays
5. get everything working and tested [pretty close, ongoing]
6. looping construct (a la FHD van Batenburg, APL91)?
7. flow control [mostly working]
8. boxed/nested arrays [working]
9. make the output code simpler, faster. It seems that this requires re-doing the basic assignment method, which binds late and so always requires re-mallocing. Once space is allocated for a variable, it should not need to be re-malloced unless it changes size.
10. allow declaration of variable sizes – for pre-allocation, speed.
11. improved operator support
12. generalized indexing
13. real bit types
14. sparse arrays
15. long double on sparc/amd?
16. arbitrary precision (gnu mp?)
17. embedded c code?
18. linear algebra
19. translate to/from aplascii/latex
20. lexer/parser – now the parser is mostly hand written (seems hard to do with yacc/bison).
21. apl interpreter
22. full execute
15 Thanks

Many thanks go to T. Budd, for releasing his code on the internet. Lots of new code and porting is due to J. B. W. Webber (J.B.W.Webber@ukc.ac.uk)

16 Questions

Contact me at swirlin@earthlink.net sam@kalessin.jpl.nasa.gov
See my home page at http://home.earthlink.net/ swsirlin/

References