A SHORT INTRODUCTION TO THE ABC LANGUAGE

While the full documentation about ABC is in the ABC Programmer's Handbook, this article should give you just enough information to get going.

THE LANGUAGE

ABC is an imperative language originally designed as a replacement for BASIC: interactive, very easy to learn, but structured, high-level, and easy to use. ABC has been designed iteratively, and the present version is the 4th iteration. The previous versions were called B (not to be confused with the predecessor of C).

It is suitable for general everyday programming, the sort of programming that you would use BASIC, Pascal, or AWK for. It is not a systems-programming language. It is an excellent teaching language, and because it is interactive, excellent for prototyping. It is much faster than 'bc' for doing quick calculations.

ABC programs are typically very compact, around a quarter to a fifth the size of the equivalent Pascal or C program. However, this is not at the cost of readability, on the contrary in fact (see the examples below).

ABC is simple to learn due to the small number of types in the language (five). If you already know Pascal or something similar you can learn the whole language in an hour or so. It is easy to use because the data-types are very high-level.

The five types are:

- numbers: unbounded length, with exact arithmetic the rule
- texts (strings): also unbounded length
- compounds: records without field names
- lists: sorted collections of any one type of items (bags or multi-sets)
- tables: generalised arrays with any one type of keys, any one type of items (finite mappings).

THE ENVIRONMENT

The implementation includes a programming environment that makes producing programs very much easier, since it knows a lot about the language, and can therefore do much of the work for you. For instance, if you type a W, the system suggests a command completion for you:

```
W?RITE ?
```

If that is what you want, you press [tab], and carry on typing the expression; if you wanted WHILE, you type an H, and the system changes the suggestion to match:

```
WH?ILE ?:
```

This mechanism works for commands you define yourself too. Similarly, if you type an open bracket or quote, you get the closing bracket or quote for free. You can ignore the suggestions if you want, and just type the commands full out.

There is support for workspaces for developing different programs. Within each workspace variables are persistent, so that if you stop using ABC and come back later, your variables are still there as you left them. This obviates the need for file-handling facilities: there is no conceptual difference between a variable and a file in ABC.

The language is strongly-typed, but without declarations. Types are determined from context.

IMPLEMENTATIONS

The sources for the Unix version are being posted to comp.sources.unix; the binaries to comp.binaries.{mac, ibm.pc, atari.st}. They should also be available from some servers (for instance by anonymous ftp from mcsun.eu.net [-n 192.16.202.1]).

There is an irregular newsletter available from us, and a book "The ABC Programmer's Handbook" by L. Geurts, L. Meertens and S. Pemberton, will be published by Prentice-Hall this year (ISBN 0-13-000027-2).

ADDRESS

ABC Implementations
CWI/AA
Kruislaan 413
1098 SJ AMSTERDAM
The Netherlands

Email: abc@cwi.nl

EXAMPLES

The (second) best way to appreciate the power of ABC is to see some examples (the first is to use it). In what follows, >>> is the prompt from ABC:

NUMBERS

```
>>> WRITE 2**1000
10715086071862673209484250490600018105614048117055336074437503883703510511249361224931983788156958581275946729175531468251871452856923140435984577574698574803934567774824230985421074605062371141877954182153046474983581941267398767559165543946077062914571196477686542167660429831652624386837205668069376
>>> PUT 1/(2**1000) IN x
>>> WRITE 1 + 1/x
10715086071862673209484250490600018105614048117055336074437503883703510511249361224931983788156958581275946729175531468251871452856923140435984577574698574803934567774824230985421074605062371141877954182153046474983581941267398767559165543946077062914571196477686542167660429831652624386837205668069377
```

TEXTS

```
>>> PUT ("ha " ^^ 3) ^ ("ho " ^^ 3) IN laugh
>>> WRITE laugh
ha ha ha ho ho ho
>>> WRITE #laugh
18
>>> PUT "Hello! "^^1000 IN greeting
>>> WRITE #greeting
7000
```

LISTS

```
>>> WRITE {1..10}
{1; 2; 3; 4; 5; 6; 7; 8; 9; 10}
>>> PUT {1..10} IN l
>>> REMOVE 5 FROM l
>>> INSERT pi IN l
>>> WRITE l
{1; 2; 3; 3.141592653589793; 4; 6; 7; 8; 9; 10}
>>> PUT {} IN ll
>>> FOR i IN {1..3}:
        INSERT {1..i} IN ll
>>> WRITE ll
{{1}; {1; 2}; {1; 2; 3}}
>>> FOR l IN ll:
        WRITE l /
{1}
{1; 2}
{1; 2; 3}
>>> WRITE #ll
3
```

COMPOUNDS

```
>>> PUT ("Square root of 2", root 2) IN c
>>> WRITE c
("Square root of 2", 1.414213562373095)
>>> PUT c IN name, value
>>> WRITE name
Square root of 2
>>> WRITE value
1.414213562373095
```

A TELEPHONE LIST

This uses the table data-type. In use, tables resemble arrays:

```
>>> PUT {} IN tel
>>> PUT 4054 IN tel["Jennifer"]
>>> PUT 4098 IN tel["Timo"]
>>> PUT 4134 IN tel["Guido"]
>>> WRITE tel["Jennifer"]
4054
```

You can write all ABC values out. Tables are kept sorted on the keys:

```
>>> WRITE tel
{["Guido"]: 4134; ["Jennifer"]: 4054; ["Timo"]: 4098}
```

The keys function returns a list:

```
>>> WRITE keys tel
{"Guido"; "Jennifer"; "Timo"}
>>> FOR name IN keys tel:
        WRITE name, ":", tel[name] /
Guido: 4134
Jennifer: 4054
Timo: 4098
```

You can define your own commands:

```
HOW TO DISPLAY t:
  FOR name IN keys tel:
    WRITE name<<10, tel[name] /

>>> DISPLAY tel
Guido      4134
Jennifer   4054
Timo       4098
```

To find the user of a given number, you can use a quantifier:

```
>>> IF SOME name IN keys tel HAS tel[name] = 4054:
        WRITE name
Jennifer
```

Or create the inverse table:

```
>>> PUT {} IN subscriber
>>> FOR name IN keys tel:
        PUT name IN subscriber[tel[name]]

>>> WRITE subscriber[4054]
Jennifer
>>> WRITE subscriber
{[4054]: "Jennifer"; [4098]: "Timo"; [4134]: "Guido"}
```

Commands and functions are polymorphic:

```
>>> DISPLAY subscriber
4054       Jennifer
4098       Timo
4134       Guido
```

Functions may return any type. Note that indentation is significant -

there are no BEGIN-END's or { }'s:

```
HOW TO RETURN inverse t:
  PUT {} IN inv
  FOR k IN keys t:
    PUT k IN inv[t[k]]
  RETURN inv

>>> WRITE inverse tel
{[4054]: "Jennifer"; [4098]: "Timo"; [4134]: "Guido"}
>>> DISPLAY inverse inverse tel
Guido      4134
Jennifer   4054
Timo       4098
```

A CROSS-REFERENCE INDEXER

'Text files' are represented as tables of numbers to strings:

```
>>> DISPLAY poem
1        I've never seen a purple cow
2        I hope I never see one
3        But I can tell you anyhow
4        I'd rather see than be one
```

The following function takes such a document, and returns the cross-reference index of the document: a table from words to lists of line-numbers:

```
HOW TO RETURN index doc:
  PUT {} IN where
  FOR line.no IN keys doc:
    TREAT LINE
  RETURN where
TREAT LINE:
  FOR word IN split doc[line.no]:
    IF word not.in keys where:
PUT {} IN where[word]
    INSERT line.no IN where[word]
```

TREAT LINE here is a refinement, directly supporting stepwise-refinement. 'split' is a function that splits a string into its space-separated words:

```
>>> WRITE split "Hello world"
{[1]: "Hello"; [2]: "world"}
>>> DISPLAY index poem
But        {3}
I          {2; 2; 3}
I'd        {4}
I've       {1}
a          {1}
anyhow     {3}
be         {4}
can        {3}
cow        {1}
hope       {2}
never      {1; 2}
one        {2; 4}
purple     {1}
rather     {4}
see        {2; 4}
seen       {1}
tell       {3}
than       {4}
you        {3}
```