THE ABC NEWSLETTER

ISSN 0169-0191 CWI, Amsterdam Issue 6, May 1988

CONTENTS

Some Notes

The Newsletter is intended to provide information about ABC and to provide a forum for discussions.

Write to us if you want to be added to our mailing list.

You are encouraged to submit any articles you see fit. Articles don't have to contain fully thought-out ideas, but may be yet undeveloped thoughts intended to stimulate discussion. The kinds of articles we have in mind are: interesting programs, either written or suggestions; unusual applications; letters, discussions on points of the language, proposed improvements, experience with the language, and so on.

If you are fortunate enough to be connected to a network with a gateway to UUCP net, you can submit articles and send mail to:

```
abc@cwi.nl
```

For older mailers, use:

```
mcvax!abc.uucp
```

Otherwise, articles and mail should be sent to

The ABC Newsletter
CWI/AA
POB 4079
1009 AB Amsterdam
The Netherlands

New Publications

There is only one new publication since the last newsletter.

An Alternative Simple Language and Environment for PCs
Steven Pemberton, 9 pages.

This article describes the background to ABC, gives an overview of the language, and presents the ABC editor and the programming environment the language is embedded in.
Published in IEEE Software, Vol 4, No 1, January 1987, pp. 56-64. A reprint appears in this newsletter.

Progress towards ABC

Steven Pemberton

CWI

Since the last newsletter, the ABC group has been busy revising ABC, updating the implementations, making implementations for new machines, and rewriting and updating the documentation. Here is an overview of the latest developments.

THE LANGUAGE

As described in the last newsletter by Lambert Meertens, several parts of the language have been revised on the basis of our experience with *B*. Here is a summary of the changes:

Representations

● Names may now include points, such as `number.of.occurrences`.
● The exponent part of a number is now written with a lower-case `e` rather than upper-case: `1e-10`. Furthermore, numbers with exponent parts are exact and not approximate numbers.

How-to's

● The keyword `HOW'TO` is now two keywords `HOW TO`. Furthermore, the keywords `YIELD` and `TEST` have been replaced by `HOW TO RETURN` and `HOW TO REPORT` respectively.
● All keywords up to the first parameter of a command are now significant, rather than just the first keyword.
● Parameter passing has changed for command how-to's: any value that the parameter has is copied into the template parameter when the command is invoked, and if the value has changed at the end, it is copied back.
● Template parameters to commands may now be multiple names, such as

```
    HOW TO MOVE x, y TO a, b:
```

Predefined commands

● The keyword `SET'RANDOM` has become two keywords `SET RANDOM`.
● `CHOOSE` has been replaced by a function `choice`. Similarly, `DRAW` has been replaced by `random`.
● There is a new command `PASS` that does nothing: it is used as a place holder, for instance in a `SELECT` alternative where you want nothing to happen:

```
    SELECT:
        too.big: SHORTEN
        too.small: LENGTHEN
        just.right: PASS
```

A command-suite must now contain at least one command.

Expressions

● Range displays for lists may now contain more than one range, such as `{"a".."z"; "0".."9"}`.
● In a range-display, if the lower-bound is greater than the upper-bound, the range is empty, rather than giving an error.
● The priorities of some operators have been changed. In particular, `@` and `|` have been integrated into the priorities of operators.
● The expressions `"word"|8` and `"word"@-8` both return `"word"` now, rather than failing.

Predefined functions

● There are four new functions on texts: `lower` which converts all upper-case letters in a text to lower-case, `upper` which does the complement conversion, `stripped` which strips leading and trailing spaces from a text, and `split` which splits a text into its space-separated parts. For instance:

```
    lower "aBc" = "abc"
    upper "aBc" = "ABC"
    stripped "  aBc  " = "aBc"
    split " The new ABC  " =
      {[1]:"The";[2]:"new";[3]:"ABC"}
```

● There are three new functions on numbers: `exactly x` which returns the number `x` as an exact number, `angle (x, y)` which returns the angle between the x axis and a line joining the origin and the point `(x, y)` and `radius (x, y)` which returns the length of the same line.
● The numeric function `atan` has been renamed `arctan`. Dyadic versions of `sin`, `cos`, `tan`, and `arctan` have been added. For instance `360 sin x` returns the sine of `x` in degrees.
● The function `n th'of t` has been renamed and changed to `t item n`.

Tests

● The test `1 = ~1` now succeeds. A new predefined predicate `exact` has been introduced to test if a number is exact or approximate.
● The predicate `not'in` is now spelt `not.in`.
● `PARSING` has been completely removed.

Terminology

Several terms have been renamed. A unit is now called a *how-to*, a target is now called an *address*. Formal parameters are now called *template parameters*. Texts, lists and tables are collectively called *trains*. A tag is now a *name*. An identifier is now a *naming*.

THE IMPLEMENTATIONS

Apart from the implementations for Unix and the IBM PC (and compatibles), there are now also implementations for the Apple Macintosh, and the Atari ST.
Good news is that the system is much faster in many places. For instance, exact arithmetic is much faster than in B, and the editor responds much faster.
Also good news is that workspaces are now properly supported: you can create new workspaces, change to another workspace, and copy data and how-to's between workspaces, within ABC.
On all machines, insofar as it is possible, the systems have the same user-interface, so there are no differences between different machines. However, there is one additional facility on the Macintosh, namely that you can use the mouse when editing (this is not yet supported on the Atari ST, though it is planned).
The ABC system is written in C. This means that the speed of ABC depends on how good the C compiler is. To give you an idea of the to-be-expected relative speeds of ABC, here is a list of the *dhrystone* speeds of the different compilers we use. Dhrystone is a benchmark program that measures the pure speed of a given processor-compiler combination. It uses no floating point, and does no input or output. For the IBM PC, the Macintosh and the Atari ST with the compilers that we used, we have the following figures:

IBM PC, 4MHz	400 dhrystones
IBM PC/AT, 8MHz	1500 dhrystones
Macintosh Plus	600 dhrystones
Atari ST	1000 dhrystones

The last two figures especially show the role that the compiler plays: the Macintosh and the Atari ST use the same processor, but the compiler on the ST produces code that is more than 60% faster.

THE ABC PROGRAMMER'S HANDBOOK

The Programmer's Handbook has been completely overhauled. Every chapter has been rewritten, and re-organised to make it more accessible and in particular an extra chapter of examples has been added, showing how you define common data-types like stacks and queues in ABC, and giving lots of example programs, both serious, and less serious.

WHEN

We are currently putting the last polish to the system, and discussing with publishers about publishing the book. We hope to have the system available in the Summer, but it really depends on the publishers. Anyway, when it is ready, it will be announced to all subscribers to the newsletter.

A Chess Program in *B*

R.J. van der Moolen

Vrije Universiteit, Amsterdam

1. Introduction

In order to finish my study of computer science at the university I had to make a comparison of some higher languages. The languages involved were SETL, Prolog, Smalltalk, *B* and Icon. This comparison was done by writing a chess problem solver in each of the languages. In this paper I will describe my experiences with *B*.

2. The Chess Program

The program is a chess problem solver which gets as input a chess problem of the form "mate in N moves" and gives the first move as output. The program consists of several modules. The unit which controls the work is the dyadic function "evaluate". This work consists of four steps:

1. find all the possible moves
2. choose one move from them and perform that move
3. check legality of new board-position; if not legal, try next move (go to 2)
4. evaluate (recursively) all the moves that may follow (go to 1); if the move appears not to be the right one, try next (go to 2).

What does a move look like? Every move is a compound consisting of source, target and extra. Source and target are just fields on the board, represented by a compound: (column, row). Extra contains extra information for special kinds of move, e.g. castling, *en passant*, and promotion. For a normal move, extra just contains the empty string.

Now what the program does is just try every move it can find, one by one, until it has found one that leads to mate within the specified number of plies, or there are no moves left to try. In the first case the move is printed, in the latter there is no solution. Every time the program tries a move the situation on the board changes. This is indicated by the state the program keeps track of. A state is a compound, which consists of a board, the player who is to move, the rook-fields which may no longer be involved in castling because the corresponding rook is moved (subset of {a1; h1; a8; h8}), the king-fields not usable for castling (subset of {e1, e8}), and the field from which a pawn may be captured *en passant* (if present; if not, it is just (0, 0)). The board is a table with the fields as (compound) keys and compounds (piece, color) as associates. Only occupied fields are present in the table.

The YIELD-unit "evaluate" looks like this:

```
(1)  YIELD state evaluate level:
(2)       PUT to'move state IN tm
(3)       IF level = 0 AND tm= "white":
(4)            RETURN no'move
(5)       FOR move IN all'moves state:
(6)            PUT state do'move move IN new'state
(7)            IF legal new'state:
(8)                 IF new'state evaluate dim = no'move:
(9)                      RETURN move
(10)      SELECT:
(11)           tm = "black" AND level = 1 AND ok state:
(12)                RETURN ((0, 0), (0, 0), "stale-mate")
(13)           ELSE: RETURN no'move
(14) no'move:
(15)      RETURN ((0, 0), (0, 0), "no move")
(16) dim:
(17)      RETURN {["white"]: level; ["black"]: level - 1}[tm]
```

Step 1 (see above) is done in line (5), step 2 in line (6) , step 3 in line (7) and step 4 in line (8). In this way eventually the right move is found (if possible of course).

What often comes up in programming is that a lot of extra program text is required for exceptions. A chess-game contains many exceptions (see above), so one can imagine that they caused my program to grow fast. These extra things were necessary in finding all the possible moves, and in performing those moves (steps (1) and (2)). To avoid making this article too long I will only show the part that performs the moves.

```
YIELD state do'move move:
    \ Perform the move and return the new state.
    PUT move IN source, target, extra
    PUT target IN column, row
    PUT state IN board, to'move, rooks'moved, kings'moved, en'passant
    PUT board[source] IN piece, color
    PUT board moved (source, target) IN new'board
    PUT rooks'moved, kings'moved IN new'rmoved, new'kmoved
    PUT (0, 0) IN new'ep
    \
    \ up to here everything was normal
    \ the rest handles special moves
    \
    \ to prevent using the rook in castling:
    IF piece = "rook" AND source in corner:
        INSERT source IN new'rmoved
    IF piece = "king" AND source = start'field to'move:
        \ to prevent any castling moves in the rest of the game
        INSERT source IN new'kmoved
        \ for castling rook has to be moved too.
        IF column = 3 OR column = 7:
            PUT (new'board, to'move) do'rook column IN new'board
    IF piece = "pawn":
        IF row = endrow: \ promotion
            PUT (extra, to'move) IN new'board[target]
        IF extra = "double": PUT target IN new'ep
        IF extra = "ep": \ for en-passant the other pawn has to be taken
            DELETE new'board[target plus (0, -direction to'move)]
    \ return the new state
    RETURN new'board, reverse to'move, new'rmoved, new'kmoved, new'ep
endrow:
    RETURN {["white"]: 8; ["black"]: 1}[to'move]
corner:
    RETURN {(1, 1); (8, 1); (1, 8); (8, 8)}

YIELD (board, to'move) do'rook column:
    \ perform rook-move for castling
    PUT first'row IN first
    SELECT:
        column = 3: RETURN board moved ((1, first), (4, first))
        ELSE: RETURN board moved ((8, first), (6, first))
first'row:
    RETURN {["white"]: 1; ["black"]: 8}[to'move]

YIELD board moved (from, to):
    \ move a piece
    PUT board[from] IN board[to]
    DELETE board[from]
    RETURN board
```

```
YIELD start'field color:
    \ Returns the startfield of the king.
    RETURN {["white"]: (5, 1); ["black"]: (5, 8)}[color]

YIELD direction color:
    \ returns the direction in which the pawn may move
    RETURN {["white"]: 1; ["black"]: -1}[color]
```

3. Results of the testing

The designers of *B* always have said that *B* is easy to learn and easy to use. Of course everyone is proud of what he creates, so I did not accept this without seeing it for myself. Well, it appeared to be right this time. *B* really is easy to learn and easy to use. The time needed to learn *B* is negligible. It is easy to express thoughts in *B*, in a very natural way. My program was ready in less than a week. The other languages needed much more time, varying from 2.5 weeks to a month.

Alas, this time is not the only time of importance. While I was waiting for the program to finish execution, I had the opportunity to do a lot of other things, including going home and returning the next day. What I want to say is, that the execution-time of a large program in *B* is much too long. This is the weakest point of *B*. In fact, when this is improved someday, *B* will be a really nice language to work with.

Because of my slow program in *B*, I don't recommend using *B* for very large programs. That does not mean, however, that *B* is useless in my opinion. *B* is perfectly suitable for prototyping purposes. In addition, if someone is able to make *B* faster, this could speed up the programs in such a way that they become useful for more than just prototyping. I don't know if they are planning such a thing, I even don't know if it is possible at all, but it is worth thinking about.

Another *B*-feature is the environment. This environment is something one has to get used to. This causes no serious problems, but experienced programmers in other languages need to get familiar with the way suggestions are made by *B*. I guess beginners don't have this trouble, because they are not used to other languages without syntax-directed editors. However, learning goes fast enough when using the editor.

In the overall-judgment I gave in my report about the mentioned five languages *B* appeared to be the best. That was not due to its speed; that was as bad as the speed of all the other languages. *B* wins with its ease of learning and ease of programming. It is possible to express thoughts in a very natural way in *B*. This is not just helpful to beginners, it is also very pleasant for experienced programmers. Because of this pleasant use it's a pity the programs are that slow.

Approximate Numbers

Steven Pemberton

CWI

There are two kinds of numbers in ABC: exact and approximate. Exact numbers are the result of computations where the result can be computed exactly: addition, subtraction, multiplication, division, and some exponentiations.
However, some operations, such as taking the square root, can't deliver an exact result in general, and deliver an approximate number instead.
Of course, some exact numbers (like 4), have exact square roots, but if you want exact results in these cases you have to write your own function to calculate them, something like:

```
HOW TO RETURN xroot x:
   IF exact x:
      PUT root */x, root /*x IN a, b
      PUT (round a)/(round b) IN r
      IF r*r = x: RETURN r
   RETURN root x
```

For approximate numbers, the ABC implementations use the hardware representation of floating-point, and this is the only place where ABC programs can deliver different results on different machines (apart from the workings of the random generator): some machines have greater floating-point accuracy than others.
Machines hold floating-point numbers as an encoding of a pair of numbers *(f, e)*, where *f*, the fraction, has a fixed number of digits to some base *b*. Such a pair then represents the number $f \times b^e$.
To find out how much accuracy your machine has, you can use the following command (adapted from a routine by Malcolm (reference 1)) to find the base and number of significant digits used on your machine:

```
HOW TO PRINT ACCURACY:
    PRINT BASE
    PRINT FRACTION
PRINT BASE:
    PUT ~2, ~2, ~0 IN a, b, base
    WHILE a+1-a-1 = ~0: PUT a+a IN a
    WHILE base = ~0: PUT a+b-a, b+b IN base, b
    WRITE "Base =", base /
PRINT FRACTION:
    PUT 1, base IN sig, b
    WHILE b+1-b-1 = ~0: PUT sig+1, b*base IN sig, b
    WRITE "Significant digits =", sig /
    WRITE "This is about `decimal` decimal digits" /
decimal:
    RETURN 1 round (sig/(base log 10))
```

It works by first looking for a whole number that doesn't produce a different number when you add 1 to it. For instance, suppose the representation we are examining has 4 digits to base 10. Then you can add 1 to all 4 digit numbers, but not to 5 digit numbers, so it will try in turn 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, and 8192. Finally it will try 16384, which is only representable as 1638×10^1, which is 16380.
The routine then searches for a number that can be added to it which does produce a different number. Exactly which number it finds depends on whether the machine rounds the last digit of a calculation, or just chops excess digits off. So it tries to add 2, 4, and 8. If the machine rounds then 16380 + 8 = 16388, which rounded to 4 places is 16390. However, if the machine chops, then 16388 chopped is 16380, so then it tries to add 16, which gives 16396, and chopped this gives 16390, so in either case we get the same result from the two types of machine.
Now subtracting these two values, 16390 − 16380, gives the base used, in this case 10.

Having found the base, the routine then goes on to find the number of significant digits by repeatedly multiplying by the base until again 1 cannot be added. In our case, it will try 1, 10, 100, 1000, and finally 10000 to which 1 cannot be added. From the number of times that the multiplication is done, the number of significant digits can be deduced.
Here is an example of its output, for a VAX:

```
>>> PRINT ACCURACY
Base = 2
Significant digits = 56
This is about 16.9 decimal digits
```

The Consequences of Approximate Numbers
The advantage of using approximate numbers in ABC is that for large numbers they are generally faster to calculate than the equivalent exact numbers. The big disadvantage is of course that your results are only approximately correct.
To take an example, consider the value $\frac{1}{\sqrt{x}-\sqrt{x-1}}$. This is the same as $\sqrt{x}+\sqrt{x-1}$ (you just multiply all through by $\frac{\sqrt{x}+\sqrt{x-1}}{\sqrt{x}+\sqrt{x-1}}$).
However, if you calculate these using approximate numbers, you will get different numbers out. For instance:

```
HOW TO COMPARE VALUES FOR x:
    PUT root x, root (x-1) IN r, r1
    PUT 1/(r-r1), r+r1 IN u, v
    WRITE "1/((root x)-(root (x-1)))=", u /
    WRITE "(root x)+(root (x-1))=    ", v /
    WRITE "Difference =", u-v /

>>> COMPARE VALUES FOR 10**12
1/((root x)-(root (x-1)))= 2000042.97959778
(root x)+(root (x-1))=     1999999.9999995
Difference = 42.9795982764044
>>> COMPARE VALUES FOR 10**15
1/((root x)-(root (x-1)))= 63161283.7647059
(root x)+(root (x-1))=     63245553.2033676
Difference = -84269.4386616889
```

The explanation for this is that if you subtract two very close numbers, you lose a lot of accuracy. Try doing it for $x=256$ with 4 digits of accuracy: root 256 = 16.00, and root 255 = 15.97. Subtracting these two gives 0.03, leaving only one digit of accuracy where before you had four. The inverse of this is 33.33, while the sum of the two is 31.97, a very different value (and as close as you can get to the real value with 4 digits).
So the question arises, in a given program, how can you tell how close a result is to the *real* answer. The rest of this article presents a package to help answer this question.

Range Arithmetic
This package simulates approximate arithmetic, and keeps for each arithmetic value an upper and lower bound on its real value. For instance, using 4 digits, *pi* lies between 3.141 and 3.142.
Then for each operation, it calculates the upper and lower bounds of the result. For instance, when adding two values (l,u) and (l',u'), the result is $(rounded.down(l+l'), rounded.up(u+u'))$. For *pi* + *pi*, this would give (6.282, 6.284). In ABC:

```
HOW TO RETURN a plus b:
    PUT a, b IN (la, ua), (lb, ub)
    RETURN down (la + lb), up (ua + ub)
```

or shorter yet:

```
HOW TO RETURN (la, ua) plus (lb, ub)
     RETURN down (la + lb), up (ua + ub)
```

Since we want to simulate approximate arithmetic using a fixed number of digits of accuracy, the functions `up` and `down` take a number and remove excess digits, rounding either up or down. The contents of the shared location `accuracy` specify how many digits of accuracy we are interested in:

```
HOW TO RETURN up x:
     SHARE accuracy
     IF x=0: RETURN 0
     PUT 0, 10**accuracy IN shift, limit
     WHILE abs x < limit: PUT x*10, shift - 1 IN x, shift
     WHILE abs x > limit: PUT x/10, shift + 1 IN x, shift
     RETURN (ceiling x) * 10**shift
```

This takes a number like 3.14159265, and multiplies or divides it so that its integral part has `accuracy` digits, also calculating the number of digits it has been shifted. So for `accuracy = 4`, `x` ends up as `3141.59265`, with `shift = -3`. It then applies `ceiling`, which gives 3142, and multiplies this by 10^{-3} to shift it back to give 3.142.
The function to round down is the same, except it uses `floor` instead of `ceiling`.

```
>>> PUT 4 IN accuracy
>>> WRITE down pi, up pi
3.141 3.142
>>> WRITE down 123, up 123
123 123
>>> WRITE down 12345, up 12345
12340 12350
```

Because exact arithmetic is being used, any value for `accuracy` can be used. Calculating to 100 places will only take more time.
Subtraction is similar to addition:

```
HOW TO RETURN (la, ua) minus (lb, ub):
     RETURN down (la-ub), up (ua-lb)
```

Multiplication is slightly harder, mainly because of problems caused when multiplying values whose upper and lower bounds have different signs. Here we take the easy way out, and calculate all possible values and identify the maximum and minimum:

```
HOW TO RETURN (la, ua) times (lb, ub):
     RETURN minmax {la*lb; la*ub; ua*lb; ua*ub}
```

where `minmax` returns the minimum and maximum values from a list:

```
HOW TO RETURN minmax list: RETURN down min list, up max list
```

We can do the same for division, but additionally we have to check for division by zero:

```
HOW TO RETURN (la, ua) over (lb, ub):
     CHECK NOT (lb <= 0 <= ub) \Division by zero
     RETURN minmax {la/lb; la/ub; ua/lb; ua/ub}
```

To use the package, a method is needed for converting an ABC number like `pi` into a range:

```
HOW TO RETURN range v: RETURN down v, up v
```

Just to show it works:

```
>>> WRITE range pi
3.141 3.142
>>> WRITE range 123
123 123
```

```
>>> WRITE range 12345
12340 12350
```

Also useful is a way of printing ranges more visibly as ranges:

```
HOW TO RETURN bounds (l, u): RETURN "[`l`:`u`] "

>>> WRITE bounds range pi
[3.141:3.142]
>>> WRITE bounds range 123
[123:123]
```

Now finally to define a function for square root. This uses the Newton-Raphson method, which repeatedly calculates for argument a, $r = ½(\frac{a}{r} + r)$, until two consecutive r's are sufficiently close:

```
HOW TO RETURN sq.root (l, u):
    RETURN down sq.root' l, up sq.root' u

HOW TO RETURN sq.root' f:
    IF f=0: RETURN 0
    PUT f, (f+1)/2 IN r, r'
    WHILE converging: PUT r', (f/r' + r') / 2 IN r, r'
    RETURN r
converging:
    REPORT down r <> down r'
```

The definition of closeness is whether the two approximations differ only in digits outside the accuracy we are interested in.

```
>>> WRITE bounds sq.root range 256
[16:16.01]
>>> WRITE bounds sq.root range 2
[1.414:1.415]
```

(Actually `sq.root'` can be speeded up by adding at the beginning

```
SHARE accuracy
PUT accuracy+1 IN accuracy
```

and replacing `(f/r' + r') / 2` by `up((f/r' + r') / 2)`. In that way, intermediate calculations don't get calculated with too great an accuracy.)

Now, finally back to our original problem:

```
>>> PUT range 1 IN one
>>> PUT sq.root range 256 IN r256
>>> PUT sq.root range 255 IN r255
>>> WRITE bounds (r256 plus r255)
[31.96:31.98]
>>> WRITE bounds (one over (r256 minus r255))
[20:33.34]
```

This makes it quite clear which of the two results is more accurate.
This last result is both shocking and sobering. But if you work it out, root 256 is [16:16.01] and root 255 is [15.96:15.97]. The difference of these two is [0.03:0.05], so obviously the inverse is [20:33.34].
Even if we use `range 16` instead of `r256`, we still get a wide range:

```
>>> WRITE bounds (one over((range 16) minus r255))
[25:33.34]
```

For more information about range arithmetic, see reference 2.

References
1. M. A. Malcolm, *Algorithms to reveal properties of floating-point arithmetic,* Communications of the ACM, Volume 15, Number 11, November 1972.
2. D. E. Knuth, *The Art of Computer Programming,* Volume 2, Seminumerical Algorithms, Addison-Wesley, Reading, Mass., 1973.

Erratum

The article "Backtracking in *B*: the Budd Challenge" in issue 5 of the a*B*c Newsletter suffered from two misprints.

On page 18 there should have been ellipsis between `Solution 1` and `Solution 26` to indicate that the rest of the output was not printed.

Just above the bottom of page 18 one line disappeared. This line indicated the start of the second solution. Together with the announcing paragraph, the output should have been:

If you add the restriction that the same piece should not be moved twice in succession, you get two solutions:

```
Solution 1
Move (1, "a", "b")
Move (2, "a", "c")
Move (1, "b", "a")
Move (2, "c", "b")
Move (1, "a", "b")
Solution 2
Move (1, "a", "c")
Move (2, "a", "b")
Move (1, "c", "b")
```

My Experiences with ABC

Jurjen Bos
CWI

In about 1978, I heard from a friend of mine, who worked at the CWI, that they were making a new programming language called *B*. He told me some incredible things, like an editor that suggests commands, and checks the syntax while you type. There was no implementation yet, he told me, but they were working on it. It sounded all too beautiful to be true. For years, this was all I knew about the *B* language.

At the end of last year, I got a job at the CWI, and then I saw implementations of *B* for the first time. I requested a copy for my Mac at home. They sent me to Timo Krijnen, who told me that I could have a copy of the prerelease Mac implementation for the revised language ABC if I wanted, on the condition I would report all bugs I encountered. Of course, I said yes, and I got my copy of ABC for the Macintosh.

When I first used the language, I thought that it would be handy for beginners only; but as an experienced programmer, I find it easier and faster to use than any other language. The first program I wrote in ABC was written about twice as fast as its equivalent in Turbo Pascal (This language is widely known to have a very user-friendly implementation). Since then, I do all my programming in ABC.

The version I got was, as mentioned above, a prerelease version, still having some bugs. However, those bugs couldn't spoil the fun of using ABC. Actually the time spent in working around them was less then the time saved by using ABC.

I have written several programs in ABC already. Two small examples are appended to this article. I also have written a very sophisticated program that factors numbers using some of the best known algorithms.

My favorite features of the language are: the very fast programming, the easy debugging and editing of existing programs, and the clear error messages. To tell an honest story, I must tell too that I find it sometimes very slow.

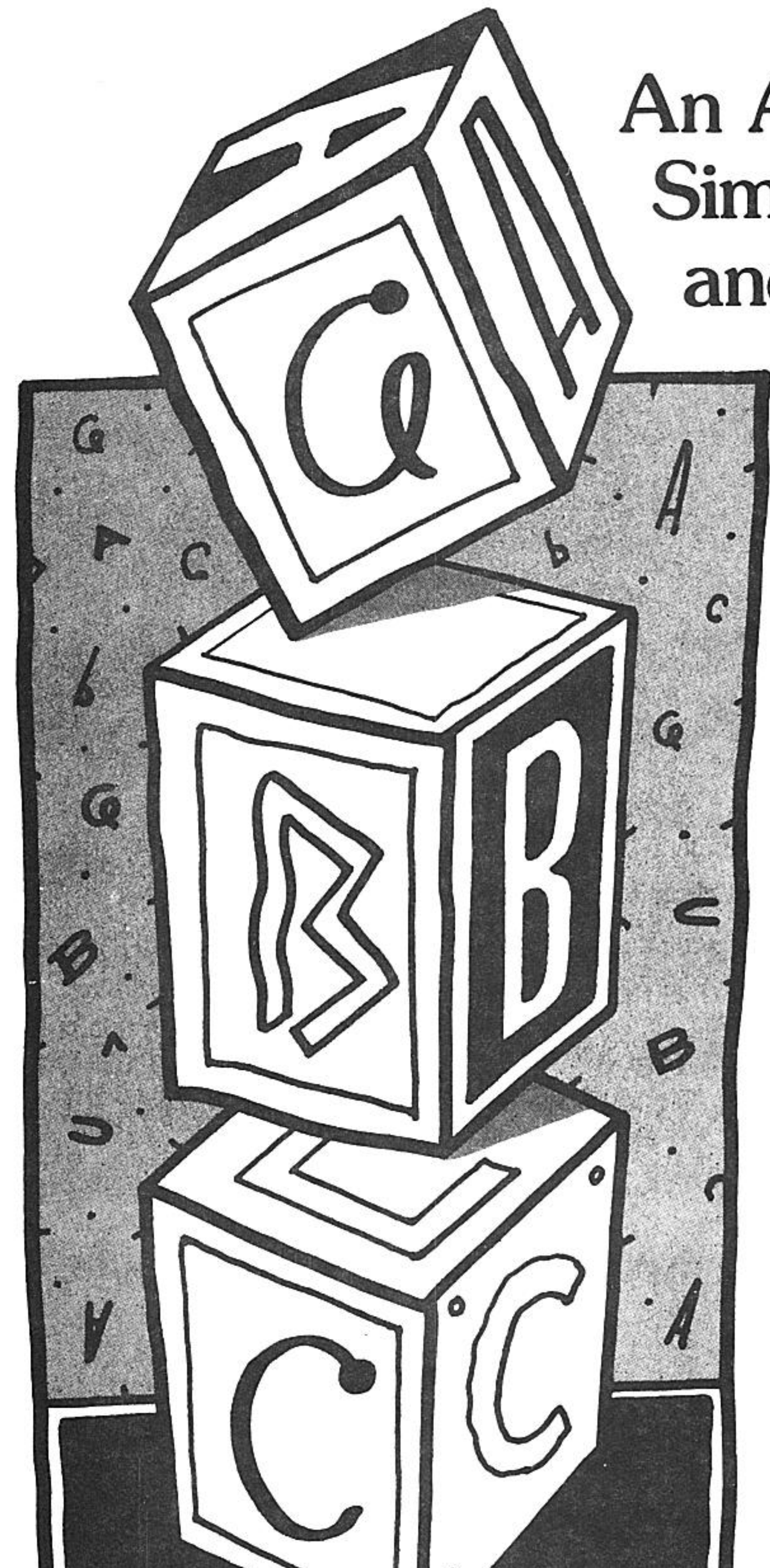
To conclude, I am happy that there finally exists a programming language that is made for people to use, and not for computers to run.

One of my first programs was a small how-to to compute the natural number `e`:

```
HOW TO COMPUTE E:
   WRITE "e=2."
   PUT 20/3, 6, 5/3, 4 IN r, f, d, n
   WHILE 1=1:
      WHILE f = floor (r+d):
         WRITE f<<1
         PUT 10*(r-f), 10*d IN r, d
         PUT floor r IN f
      PUT d/n IN d
      PUT r+d IN r
      PUT floor r IN f
      PUT n+1 IN n
```

Finally, on the next page is the listing (or output) of a self-reproducing program:

```
HOW TO SELFREPRODUCE:
   PUT "" IN dna
   PUT dna^"HOW TO SELFREPRODUCE:\n1PUT \q\q IN dna\d\n1PUT dna IN p\n1" IN dna
   PUT dna^"PUT {[\q0\q]: 0; [\q1\q]:3; [\q2\q]: 6; [\q3\q]: 9; [\q4\q]" IN dna
   PUT dna^": 12; [\q5\q]: 15} IN indent\n1WHILE p<>\q\q:\n2SELECT:\n3p" IN dna
   PUT dna^"|1<>\q\\\q:\n4WRITE p|1\n4PUT p@2 IN p\n3p|2=\q\\n\q:\n4WRI" IN dna
   PUT dna^"TE /\q \q^^indent[p@3|1]\n4PUT p@4 IN p\n3p|2=\q\\q\q:\n4WR" IN dna
   PUT dna^"ITE \q\q\q\q\n4PUT p@3 IN p\n3p|2=\q\\d\q:\n4PUT dna IN p2\" IN dna
   PUT dna^"n4WHILE p2<>\q\q:\n5WRITE /\q   PUT dna^\q\q\bp2|59\b\q\q I" IN dna
   PUT dna^"N dna\n5PUT p2@min{60; #p2+1} IN p2\n4PUT p@3 IN p\n3p|2=\q" IN dna
   PUT dna^"\\\\\q:\n4WRITE \q\\\q\n4PUT p@3 IN p\n3p|2=\q\\b\q:\n4WRIT" IN dna
   PUT dna^"E \q\b\b\q\n4PUT p@3 IN p" IN dna
   PUT dna IN p
   PUT {["0"]: 0; ["1"]: 3; ["2"]: 6; ["3"]: 9; ["4"]: 12; ["5"]: 15} IN indent
   WHILE p<>"":
      SELECT:
         p|1<>"\":
            WRITE p|1
            PUT p@2 IN p
         p|2="\n":
            WRITE /" "^^indent[p@3|1]
            PUT p@4 IN p
         p|2="\q":
            WRITE """"
            PUT p@3 IN p
         p|2="\d":
            PUT dna IN p2
            WHILE p2<>"":
               WRITE /"   PUT dna^""`p2|59`"" IN dna"
               PUT p2@min{60; #p2+1} IN p2
            PUT p@3 IN p
         p|2="\\":
            WRITE "\"
            PUT p@3 IN p
         p|2="\b":
            WRITE "``"
            PUT p@3 IN p
```



An Alternative Simple Language and Environment for PCs

ABC is a simple language for personal computing. Intended as an alternative to Basic, it has grown to be a powerful tool for expert users, too.

Steven Pemberton, CWI Amsterdam

ABC is a programming language being designed and implemented at CWI, the Centre for Mathematics and Computer Science, in Amsterdam. It began as an attempt to design a suitable alternative to Basic for beginner programmers — a language that was still easy to learn, still interactive, but was easier to use and offered program structure. It has developed into an interesting and pleasurable tool for beginners and experts alike. The box at right describes why a language like ABC is needed.

ABC is being designed and implemented with an integrated programming environment. Although the project's emphasis has shifted from beginners to personal computing since its beginnings in 1975, the main design objectives have remained the same:

- simplicity,
- suitability for interactive use, and
- availability of tools for structured programming.

The language has been designed iteratively. The version described here is the fourth iteration. The first two versions were the work of Lambert Meertens and Leo Geurts of the Centre for Mathematics and Computer Science (then called the Mathematical Centre) in 1975-76 and 1977-79. They were definitionally simple — easy to learn and easy to implement.

In the third iteration,[1,2] designed in 1979-81 with the help of Robert Dewar of New York University, it became conceptually simple. It is still easy to learn, by having few constructs — but it is also also easy to use because it has powerful constructs without the restrictions professional programmers are trained to put up with but a newcomer finds irritating, unreasonable, or silly.

Furthermore, this third version (which had a working title of B) was designed deliberately with the new generation of computers in mind by relegating machine efficiency to a lower priority than programmer efficiency.

It is surprising to realize how quickly recent developments in computer technology have caught up with us. Only a short time ago, we were telling people that we weren't considering implementing ABC on machines with less than 128K bytes of main storage, and getting surprised reactions that we were considering such huge machines. Today, many PCs start at 128K bytes, and 512K bytes is quite normal.

One reason that ABC needs a lot of storage is that it is not just a language but a complete programming environment. Traditional computer use for programming involves not only learning the programming language but also a whole host of subsystems and their commands, such as the operating system's command language, editor, and compilers, which are often completely separate and noncooperating.

ABC, on the other hand, shows a unified face to the user, so it is not necessary to learn anything outside the ABC system. This means that ABC must be able to perform many tasks normally delegated to the

New computers, old languages: ABC's background

It is a common observation that the latest personal computers are very powerful, certainly more powerful and more capacious than many of the previous generation of large computers.

Thus, it is likely that many PCs will spend most of their time idle — not from lack of use, but from underused capacity. And this is not because of delusions of grandeur on the purchaser's part: The central processor, the part that is responsible for much of the measure of speed in a computer, is but a tiny part of the cost of a modern computer. There is no economic (or other) advantage in using a slower processor.

It is therefore surprising that most programming on PCs is done with programming languages that are usually 15 to 20 years old and designed for computers of an earlier generation. Whatever PC you buy, you can be sure that the one language available is Basic, a language designed in the mid-1960s, which has been described as "an adaptation to early and very marginal computer technology" by Seymour A. Papert (in *Computers and Learning in the Computer Age*, M.L. Dertouzos, ed., MIT Press, Cambridge, Mass., 1979).

Thus you have the strange situation of people programming the computers of the eighties with a language of the 1960s, a language unable to take advantage of the increased capabilities of the newer machines.

Basic's two main advantages are that it is interactive and that it is simple. Interactiveness is the ability to type in and run a program immediately without going through any intermediate stages like translating the program into machine code. It is also the ability to correct a program and rerun it immediately.

Strictly speaking, this is a property of a language implementation and not of the language itself, because in principle it is possible to make an interactive implementation of any language or a noninteractive version of Basic. But that notwithstanding, a language usually has features that orient it more or less towards interactive implementation, and Basic is usually implemented interactively, unlike other languages.

Simplicity is a property often claimed for a programming language or system, although there are two senses to the word that in some ways conflict. You can have definitional simplicity, where there are only a few concepts, and you can have what might be called conceptual simplicity, where the concepts are closest to your needs.

For instance, consider the difference in simplicity of use between automatic and manual gears in cars and the extra complexity in the construction of automatic gears. As another example, all operations in Boolean algebra can be expressed using a single operator nand representing "not and." However, no one would consider the expression

```
(drowning nand (drowning nand drowning)) nand ((waving nand waving) nand (drowning
nand drowning))
```

as simpler than the equivalent

```
not (waving or drowning)
```

despite the larger number of concepts in the second.

The simplicity of Basic is definitional: It is easy to implement and has few concepts to be learned, but once learned it remains easy to use only for very small programs. Beyond that, it is like cutting your lawn with a pair of scissors.

Basic is also widely taught in preuniversity education. Apart from the perceived simplicity of the language, machines large enough to run anything other than Basic were usually too expensive. This is changing quickly with the coming generation of cheap large computers, but the risk is that Basic will continue to be used from sheer momentum and perceived investment in the language (a common barrier to change outside education, too).

What risk? Basic has little to recommend it for educational use. Just as with one-finger typing, where learning to type properly means first getting rid of your old habits, Basic's paucity of structuring facilities means that much time has to be dedicated to learning ways of getting around its expressive poverty — and the student ends up learning bad habits that must only be unlearned to progress to other languages.

ABCs of ABC

To produce a simple but powerful language, we provided few but powerful constructs.

Data types. There are five, all without size limits:
- numbers
- text strings
- compound values
- lists
- tables

No declarations. Types are derived and checked from context.

Commands. There are four assignment and data-structure update commands:
- Put
- Insert
- Remove
- Delete

There are two input/output commands:
- Read
- Write

There is a randomizing command:
- Set Random

There are five flow-control commands:
- If
- While
- For
- Select
- Check

Definitions. To define your own commands and functions, with their associated exit commands:
- How To (command) with Quit
- How To Return (function) with Return
- How To Report (test) with Report, Succeed, and Fail
- Share (for importing global variables)

Refinements support stepwise programming.

Operators. Numeric operators include:
- + (addition)
- − (subtraction)
- * (multiplication)
- / (division)
- ** (exponentiation)
- random (random number)
- root *x* and *n* root *x* (*n*th root of *x*)

Text operators include:
- ^ (concatenation)
- ^^ (duplication)
- *t* | *n* (first *n* characters of *t*)
- *t* @ *n* (substring starting at position *n*)
- *e* < < *n* (pad expression *e* to the left within *n* spaces)
- *e* > < *n* (center *e* within *n* spaces)
- *e* > > *n* (pad *e* to right within *n* spaces)

General operators include:
- = (equality)
- < > (inequality)
- > (greater than)
- < (less than)
- in (membership test)
- # (value size)
- min (minimum value)
- max (maximum value)
- choice (random value from a text, list, or table)

operating system or other subsystems, such as editors.

We have now finished a final polishing of the language, based on five years' experience of using and teaching B, resulting in ABC.

Language overview

As an example of ABC's simplicity and power, consider the task of creating and maintaining a database of telephone numbers. (See the box at left for brief definitions of commands and operators.) You first create an empty telephone list:

```
PUT {} IN tel
```

and then add a few numbers:

```
PUT 4133 IN tel["Leo"]
PUT 4141 IN tel["Doug"]
PUT 4166 IN tel["Paul"]
```

Now individual numbers can be looked up (italics are used throughout for output produced by ABC):

```
WRITE tel["Leo"]
4133
```

or the whole list can be written out:

```
WRITE tel
{["Doug"]: 4141; ["Leo"]: 4133; ["Paul"]:
4166}
```

The names are kept sorted.

Of course, if the list becomes large, this sort of output becomes hard to read, so the list can be written more tidily with the following. (A / in a Write command causes a new line to be written).

```
FOR name IN keys tel:
   WRITE name, ":", tel[name] /
Doug: 4141
Leo: 4133
Paul: 4166
```

It is easy to find out which name belongs to a given number:

```
IF SOME name IN keys tel HAS tel[name]
      = 4133:
   WRITE name
Leo
```

But if this is done often, it is easier to produce the inverse table:

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

```
WRITE subscriber[4133]
Leo
WRITE subscriber
{[4133]: "Leo"; [4141]: "Doug"; [4166]:
"Paul"}
```

If you need to compute the inverse of a table often, it is easy to make a function to do it for you. Functions may return values of any type.

The telephone list is saved automatically, without any further action on your part. If you log out and come back later, it will still be there.

Types and values. ABC has just two basic data types: numbers and text. It has just three ways to combine values: compounds, lists, and tables.

Numbers. The seasoned computer user will be surprised by ABC's handling of numbers. First, following the maxim of no restrictions, numbers may be as large as you want (within the physical limits of the computer's memory). You may calculate 10^{200} as easily as 10^2. A Dutch newspaper recently dedicated a whole page to printing the value $2^{132049}-1$ (the largest prime then known), which had been calculated with the ABC program WRITE 2**132049 −1 that — although it took a while to run — produced the final answer of more than 39,000 digits.

Second, when possible, numbers are always kept exact — even fractional numbers. Thus, as long as you use exactness-preserving operations like addition, subtraction, multiplication, and even division, a number is calculated exactly. Operations like taking the square root cannot produce an exact result in general and so produce an approximate number, rounded to some length.

Along with the usual arithmetic operators, there is a set of mathematical functions. A nice feature is that several take two forms. For instance, root x returns the square root of x, while n root x returns its nth root; log x returns the natural logarithm, while b log x returns the logarithm to the base b.

Text. Text is handled as strings of printable characters. Unlike many other languages, ABC has a full range of operations on text strings such as joining them together, replicating them, and taking substrings. Just as with all types in ABC, there is no maximum size imposed on a text string, nor is the size declared in advance.

For example, consider the definition of a function to capitalize a word. It uses two predefined functions, Upper and Lower,

Numbers can be as large as you want: 10^{200} is calculated as easily as 10^2. Plus numbers are kept exact whenever possible.

that convert all letters in a text string into uppercase and lowercase, respectively. The operator | returns a substring of the given length, the operator @ gives the substring starting at the given position, and the operator ^ joins two text strings. Comments are preceded with \.

```
HOW TO RETURN capitalized word
    \ "word" is the parameter
  RETURN (upper word|1)^(lower
    word@2)

WRITE capitalized "amsterdam"
Amsterdam
WRITE capitalized "LONDON"
London
```

Compound values. Compounds are tuples (or records, as they are called in some languages), although they are hardly noticeable in ABC. For instance in

```
WRITE a, b
WRITE x[a, b]
```

both occurrences of *a, b* are an expression of type compound. An example of where they are noticeable is in

```
PUT "Winston Smith", 45 IN person
```

which packs the two values in the one location. The only other action you can do on a compound is unpack it:

```
PUT person IN name, age
WRITE name
Winston Smith
WRITE age
45
```

Compounds can be used to swap the values of locations:

```
PUT a, b IN b, a
```

List values. Lists are sorted collections of elements, again unrestricted in size. The elements of a list must all be the same type, but they may otherwise be any type. Thus, you may have lists of text strings, numbers, compounds, lists of other lists, and so on.

Elements may be duplicated. Among other things, you can insert elements, delete elements, find out if an element is present, and find the size of a list. As with other types, the Put command assigns a whole list to a location. Insert adds an extra element to an existing list:

```
PUT {1..10} IN l
WRITE l
{1; 2; 3; 4; 5; 6; 7; 8; 9; 10}
REMOVE 5 from l
INSERT pi IN l
WRITE l
{1; 2; 3; 3.141592653589793; 4; 6; 7; 8;
9; 10}
```

The following defines a command that uses a list of numbers and the sieve method to calculate primes:

```
HOW TO SIEVE TO n:
        \ name is SIEVE TO
   PUT {2..n} IN set
        \ set to be sieved
   WHILE set > {}:
        \ repeat indented part
      PUT min set IN p
        \ smallest member
      WRITE p
      FOR m IN {1..floor (n/p)}:
        \ remove multiples of prime
         IF m*p in set:
            REMOVE m*p FROM set

SIEVE TO 50
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47
```

Table values. A table (which was the data type used in the telephone list example at the beginning of this section) is a generalization of an array. It is a mapping from values of any one type onto values of any one other type.

Standard programming languages let you map only contiguous integers (and sometimes a few other similar types) onto other types. But ABC lets you use any type for the array indexes — whether you want mappings from text to lists, from tables to numbers, or from tables to other tables.

For example, consider a program that is repeatedly required to calculate a result for the same input values. One way to speed this up is to use a memo function that remembers past values and doesn't recalculate them when asked for them again.

This example uses a table to store pairs of points already calculated and the distances between them. The operator Keys returns as a list the set of indexes used so far in the table. In this example, this set contains all pairs of points stored in the table. The Share command causes a global variable to be used instead of a variable local to the command.

```
HOW TO DISTANCE a TO b IN r:
  SHARE memo
  IF {a; b} not.in keys memo:
    PUT a, b IN (x, y), (x', y')
    PUT root((x-x')**2 + (y-y')**2)
      IN memo[{a; b}]
  PUT memo[{a; b}] IN r
```

```
PUT {} IN memo
DISTANCE (0, 0) TO (3, 4) IN dist1
DISTANCE (5, 2) TO (20, 10) IN dist2
DISTANCE (3, 4) TO (0, 0) IN dist3
WRITE dist1, dist2, dist3
5 17 5
WRITE memo
{[{(0, 0); (3, 4)}]: 5; [{(5, 2); (20, 10)}]:
17}
```

The two points are stored in the table as a list {*a; b*} instead of as a compound (*a*, *b*). This means they will be sorted, so the order used in the parameters won't matter (as the first and third calls of Distance show).

The equivalents of multidimensional tables can be achieved using compounds as keys:

```
PUT {} IN event
PUT "Xmas" IN event["Dec", 25]
PUT "New Year" IN event["Jan", 1]
WRITE event
{["Dec", 25]: "Xmas"; ["Jan", 1]: "New
Year"}
```

Items in a table can be deleted with the Delete command.

Apart from the operations that work only on text, lists, and tables, there are several generic operators that work on all three. They include Max, which will return the largest element of a text, list, or table, and the operator #, which returns the number of elements in a text, list, or table. Similarly, a For command will step through the elements of any of the three data types.

ABC supplies high-level tools that you can use for low-level operations, rather than the usual set of low-level tools you must construct high-level functions with.

Any command or function that you define yourself with only generic operators and commands can also be used generically.

High-level data types

As you can see, ABC has a small set of rather powerful data types. Most other languages supply you with low-level tools that you must then use to build your own high-level tools. ABC does it just the other way around: You get high-level tools that you can also use for low-level purposes. For instance:

- In other languages, if your numbers go higher than a certain limit, you must write your own numerical package. In ABC there is no maximum limit, so small and large numbers can be handled with the same ease.
- In traditional languages, if you want to use sparse arrays, you must write a package to implement them using other data types. In ABC, sparse arrays (tables) are the default, but you can use them nonsparsely without extra effort.
- Traditional languages sometimes supply a pointer type that you can use to create data space dynamically. In ABC, data space is automatically dynamic. Furthermore, pointers are frequently used in other languages for sorting and searching. ABC supplies these sorting and searching facilities as primitives. If you still need to use pointers, you can represent them with tables, but with additional advantages like being able to print them out.[3]

Structured programming tools

The example ABC programs show that the data types are somewhat unusual but that the commands, or statements, are rather familiar. There are the usual input and output commands, the assignment command, the If, While, and For commands, and so on.

Commands like If and While are well-known tools for structured programming. An unusual feature of ABC is program refinement. Refinements explicitly support the idea of stepwise refinement, a technique where you specify your program in a short, high-level form that gives a good overview of what the program does — in effect reducing it to several simpler, related programs.

This high-level form is then refined by writing these lower level programs in the same manner until the lowest level is reached that can be expressed. For instance, the top level of a game-playing program might look like this:

```
INITIALIZE
PLAY.ONE.GAME
WHILE more.wanted:
  PLAY.ONE.GAME
```

Then PLAY.ONE.GAME might look like this:

```
WHILE not.over:
  DISPLAY.BOARD
  GET.MOVE
  IF not.over:
    MAKE.MOVE
```

Further steps refine not.over, DISPLAY.BOARD, and so on.

Stepwise refinement is intended to make the process of writing a large program easier by splitting the task into several smaller, and therefore more manageable, subtasks. Surprisingly, although the technique has been around for more than a decade, very few programming languages explicitly support it.

Although subroutines can be used for stepwise refinement in other languages, they rarely are used because of the execution overheads associated with calling a subroutine. By supplying a facility without these overheads, ABC encourages the use of stepwise refinement.

Benefits and trade-offs

A good example of ABC's ease of use is that global variables are permanent in the sense that they remain not only while you work at the computer but even after switching it off and returning later. Thus, variables may be used instead of files in the traditional sense, so there is nc need for extra file-handling facilities in the language.

Because variables are dynamic, and unrestricted in size, using them instead of files causes no difficulties. Quite the reverse in fact, because you now can use the powerful data types, which give you random and even associative access to the contents, along with all their predefined operators.

Program length. As an example of these benefits, compare the following programs in ABC and Pascal to find the length of the longest line in a text file. In ABC:

```
PUT 0 IN longest
FOR line IN document:
   PUT max{longest; #line} IN longest
WRITE longest
```

In Pascal:

```
program count(document, output);
var document: text;
   c: char;
   length, longest: integer;
begin
   reset(document);
   longest := 0;
   while not eof(document)
   do begin
         length := 0;
         while not eoln(document)
         do begin
               read(document, c);
               length := length + 1
            end;
         readln(document);
         if length > longest
         then longest := length
      end;
   write(longest)
end.
```

These programs illustrate clearly how compact and readable ABC programs are. Often ABC programs are a quarter or a fifth of the length of their equivalent Pascal and C programs. Examples include a 1000-line C program that resulted in a 200-line ABC program, a 110-line Pascal cross-reference program that became a 24-line ABC program, and a 284-line Pascal program published in the November 1984 *Byte* that has an equivalent ABC program of only 24 lines.

The program-size ratio compared to Basic would be even greater in ABC's favor. This clearly has consequences for programmer efficiency, especially because programmer effort is proportional not to program length but to a power of program length. Brooks[4] reports that this power is around 1.5, implying that ABC is something like an order of magnitude easier to use than traditional languages.

A program you might expect to take a week of programming in a traditional language takes about an afternoon in ABC.

This seems to be borne out in practice: A program that you might expect to take a week of programming in a traditional language takes about an afternoon in ABC.

Execution speed. The other side of this coin is that, because of its higher level, ABC is no longer so straightforward to implement. And because it is interpreted, programs will not run as fast as equivalent programs in compiled languages.

However, the new generation of personal computers are so powerful that they spend a large proportion of their time idle. This trade-off of computer time against programmer time is more than reasonable in view of this excess computational capacity: Most people would far rather spend less time programming in exchange for a slower program.

Of course, the end user of a program cares about how fast a program runs — but not to the exclusion of all other factors, like the cost and the reliability of a program. Many people use programs written in higher level languages that run slower than if they had been written in assembler. This is one reason high-level languages have been more successful than assembly language.

Furthermore, there are other trade-offs involved when comparing noninteractive languages with interactive ones, such as the absence of a translation phase in an interactive language. For example, in an interactive language, a change in a single line can be tried immediately without having to wait for the whole program to be recompiled.

Comparing ABC with Basic on this score is another matter. Basic implementations tend to be slow anyway, yet many people are willing to accept this slowness in return for interactive access. For instance, Bentley[5] reports that Basic on an (apparently large) personal computer he used ran at 100 instructions per second — even slower than the first commercially produced computers of the 1950s, which ran at 700 instructions per second!

Higher level commands like ABC's take more time individually, but fewer need to be executed to do the same job, and more work is done at the faster system level than is done with a lower level language. The combined effect depends on the application and on the mix of operations in a program.

Simple programs, which take little time anyway, and programs that contain only simple numeric operations will generally run slower. A program to sum a thousand logarithms took one second in compiled Pascal, two seconds in interpreted Pascal, and nine seconds in ABC.

But more complicated tasks may well run faster in ABC than if they were coded in a lower level language. For instance, the above program to find the longest line in a 1000-line file took 31 seconds in interpreted Pascal, 13 seconds in compiled Pascal, and five seconds in ABC.

However, even if a program in ABC runs slower than acceptable (for instance a commercial application that must run as fast as possible on a slow microcomputer), the programmer efficiency of ABC still makes it a good choice for the prototyping phase of a project.

Teaching

While ABC was not specifically designed for educational use, it turns out to be well-suited for teaching. The availability of program-structuring and data-structuring facilities, including support for stepwise refinement, means that students are less likely to adopt bad habits.

More important, because of ABC's high level, a student can quickly become competent enough to produce useful programs rather than just trivial exercises.

ABC is being used in several European educational institutes of different levels and types, with enthusiastic responses. Teachers especially find the interactive elements of ABC useful because many elementary syntax errors cannot be made in ABC and because students are encouraged to try features for themselves rather than asking what to do. The teacher thus has more time to answer the less trivial questions.

Interaction

Just as with Basic, any ABC command typed at the terminal is executed immediately. Thus, you may use all the features of ABC as a sort of high-grade calculator:

```
WRITE root 2
1.41421356237
```

Furthermore, since user-written programs are called in exactly the same way as built-in ABC commands, much of the need for a separate command language often found on computers disappears. Variables serve as files, and since programs are just the equivalent of subroutines in other languages, parameters can be passed to programs using the same parameter-passing mechanism. Systems that allow parameter passing usually do so with a completely different mechanism.

ABC's interactiveness also means that declarations are not used. Basic users usually perceive this as an advantage because it means less typing, while users of other languages (such as Pascal) accept declarations on the grounds that they let type inconsistencies and other similar errors be detected before the program is run, reducing the time taken to get a program correct.

ABC supplies the advantages of both by inferring the types of variables from how they are used (for instance, if you write $a*2$, a must be a number) and by checking that all such uses are consistent. Furthermore, inconsistencies are checked by the editor, increasing the interactive feel of the language.[6]

One demand on an interactive language is that typing be minimized, since so much time is spent at the keyboard. One solution to this (used by many interactive systems) is to use abbreviated commands, but this

The editor knows about things like matching brackets and supplies them for you, so certain typing errors are not possible.

generally results in very cryptic-looking commands.

ABC solves this by having a dedicated editor that knows much about the syntax and semantics of ABC. As an example, consider the Write command. This is the second-most-used command in ABC (the first is Put), so when you type a "W" as first letter of a command, it is more than likely that you want a Write command. Thus, the moment you type a "W," the system immediately suggests the rest of the command to you and shows that it has one parameter:

```
W?RITE ?
```

If you want a Write, you press the tab key and the system positions the cursor so you can type in the expression you want to write:

```
WRITE ?
```

If you don't want a Write, but a While, you ignore the suggestion and type the next character, an "H." The system then changes the suggestion to match:

```
WH?ILE ?:
```

Suggestions also work for the commands you define yourself (such as the Sieve To defined earlier).

The editor also knows about things like matching brackets and supplies them for you, so certain typical sorts of typing errors are not possible. These suggestions are just suggestions — you can still type letter for letter, ignoring the suggestions, and get the same result.

ABC uses indentation to indicate command nesting, so there is no need to bracket commands with Begin and End or similar command pairs. The editor knows about indentation and supplies it automatically. For instance after typing the first line of a For command, you get

```
FOR i IN {1..10}:
    ?
```

To leave one level of indentation, you just type an extra return.

(With the suggestion mechanism, you could have typed the above For command as

```
f tab i tab {1..}
```

followed by a return.)

Instead of a single-character cursor that most text editors have, the ABC system has a multicharacter focus, in the style of more modern text editors. However, the ABC focus is based on the syntax of ABC, and there are editor commands to move it according to the program structure. For instance if you are focused on the following command:

```
PUT a - b IN a
```

pressing the First key focuses on the first (variable) part of the command:

```
PUT a - b IN a
```

Typing an open parenthesis here encloses the focus in parentheses (because of the suggestion mechanism):

```
PUT (a - b) IN a
```

There are Next, Widen, and Last keys to perform similar focus selection. Thanks to this focus, the editor doesn't need lots of commands to delete characters, words, and lines and to copy characters, words, and lines — just a few to move the focus and a few to specify the action on the focus, such as copying or deleting it.

Environment

The ABC editor is a central element of the ABC programming environment. When in ABC, you are always using the editor, even when typing data for Read commands. The ABC system is organized so the editor is used instead of many functions that would normally be performed by a separate command language, like deleting, copying, and renaming files and directories, switching to other directories, and deleting jobs.

The ABC system consists of workspaces, each containing any number of documents. These documents are of several different types, such as programs, global variables, and text documents. It is possible to edit any document. Index documents list the program units, variables, and so on in a workspace. A global index lists the workspaces. The indexes are editable, too: If you delete an entry in the list, the corresponding object disappears. Similarly, you can use the editor to copy or rename any entry.

There is also a document for each workspace, called the session record, where you can issue commands and run the programs in the workspace.

A feature of ABC's what-you-see-is-what-you-get philosophy is that you may edit the commands you have entered and executed in the session record. This causes the changed commands to be reexecuted as if you had typed the commands that way in the first place. For instance, if you had typed in the following commands

```
PUT 2 IN a
PUT root a IN b
WRITE b
1.41421356237
```

and then went back to the first command and changed the "2" to "10," the system would display

```
PUT 10 IN a
PUT root a IN b
WRITE b
3.16227766017
```

This is similar to how spreadsheet programs update their displays after changes.

The system also has an advanced undo mechanism. Any operation can be undone, and by repeatedly pressing the undo key, more and more can be undone, and redone again if you undo too much. In principle, you can go back as far as you want, just as in principle lists can be as long as you want. In practice, of course, it depends on your resources. Not only is this exceptionally useful when you delete the wrong section of a document but also when you delete the wrong variable or program.

Furthermore, it can be used in place of an interrupt when running a program, since the return that started a command running can be undone, returning you to the state before it started running — thus stopping the command.

But how does ABC compare with other programming environments? When considering such a question, you have to take into account the aims and purposes of the language and whether it was designed with its environment. Many classical environments, such as C's Unix,[7] bear the marks of being designed for interactive use but lack unity in their components.

The ABC editor is the central element of the ABC programming environment: You are always using the editor, even when typing data.

For instance, with C, the standard editors know nothing about the language and don't interact with the error messages from the compiler, so they can't take you automatically to the lines in error. Other environments are built around languages not designed for interactive use, such as Pascal. While these make the language much nicer for the programmer to use, they can't take advantage of the unity of language and environment.

Smalltalk is a good example of language and environment designed together. Unlike most languages (and like ABC), it exists only in interactive implementations. It has many features desirable in an interactive system, such as a unified debugger — although the editor is very simple and knows nothing of the language, and the language is harder to learn than ABC.

Implementation

Part of our effort is to create ABC implementations. A pilot implementation ran from 1981 to 1984, and a new portable implementation for Unix machines has been distributed to several dozen sites (with more sites expected). A first implementation for the IBM PC and compatibles under MS-DOS is now available, and plans for other personal computers, such as the Macintosh and the Atari ST series, are well advanced.

The original implementation was written in 1981. It was explicitly designed as a pilot system to explore the language rather than to produce a production system, so the priority was on implementation speed rather than on execution speed. As a result, it was produced by one person in two months. It was slower than desirable, but still usable.

The current versions of the system[8] are aimed at wider use, and therefore speed and portability have become an issue. The system has also become more functional in the rewrite. Like the pilot system, they were written in C. They were produced by first modularizing the pilot system and then systematically replacing modules so we had a running ABC system at all times. They were produced in a year by a group of four.

Several interesting implementation techniques have been used to speed up typical ABC programs. As an example, ABC values are implemented with pointers and reference counts so the cost of assigning a value to a variable is independent of the value's size: It is as cheap to copy a large list as it is to copy a number.

This means that there is a value size above which this method becomes cheaper than ordinary copying. This critical size is rather small, and, since ABC values easily become large, it is advantageous. (The size depends on the architecture of the machine it is running on and on the dynamic pattern of the running program. One report showed that for a DEC PDP 11, it is around four words on average.[9] We have not experimented with our implementation to see what it is there.)

Furthermore, Put commands are typically the most executed command in programs, and so it makes sense to choose a method that favors them.

We have just finished the last polishing of the language and have cleaned up a few odd corners. We are now adapting the implementation to this revision. When that is complete, the language will be formally released with the *ABC Programmer's Handbook*,[10] which describes the language and its use.

The implementations run on larger machines running Unix and MS-DOS. While not all the facilities of the environment described here are implemented in these releases (in particular, editing the session record), most are implemented and the rest will be later. The MS-DOS implementation will be included free with the programming handbook, while the Unix implementation is available at cost by writing to the author.

After the formal release, we will focus our work on the environment — for instance, to do for graphics and data entry what we have so far done for programming. □

References

1. Leo Geurts, "An Overview of the B Programming Language," *SIGPlan Notices*, Vol. 17, No. 12, Dec. 1982.
2. Lambert Meertens and Steven Pemberton, "Description of B," *SIGPlan Notices*, Vol. 20, No. 2, Feb. 1985.
3. Steven Pemberton, "Examples of B," *B Newsletter*, No. 2, CWI, Amsterdam, June 1984.
4. Fred P. Brooks, *The Mythical Man Month*, Addison-Wesley, Reading, Mass., 1975.
5. Jon Bentley, "Programming Pearls," *Comm. ACM*, Vol. 27, No. 3, Mar. 1984.
6. Lambert Meertens, "Incremental Polymorphic Type-Checking in B," *Proc. 10th ACM Symp. Princ. Programming Languages*, ACM Press, New York, 1983, pp. 265-275.
7. Brian W. Kernighan and Rob Pike, *The Unix Programming Environment*, Prentice-Hall, Englewood Cliffs, N.J., 1984.
8. Lambert Meertens and Steven Pemberton, "An Implementation of the B Programming Language," Tech. Report CS-N8406, CWI, Amsterdam, 1984.
9. Peter G. Hibbard, Paul Kneuven, and Bruce W. Leverett, "A Stackless Runtime Implementation Scheme," *Proc. Fourth Int'l Conf. Design and Implementation of Algorithmic Languages*, Courant Institute, New York University, New York, 1976.
10. Leo Geurts, Lambert Meertens, and Steven Pemberton, *The ABC Programmer's Handbook*, To appear.

Steven Pemberton works on the ABC project at CWI (Centre for Mathematics and Computer Science) in Amsterdam. His experience includes work in Pascal and Algol 68. His research interests include programming language design and implementation and programming methodology.

Pemberton has been a lecturer at Brighton Polytechnic in England and a member of the research groups at Manchester University and Sussex University in England.

The author can be reached at Informatics AA, CWI, Postbox 4079, 1009 AB Amsterdam, The Netherlands.

Publications about *B* and ABC

Many people have asked for an overview of the currently available publications about *B* and ABC. They are gathered here under topic. Within each topic the order is chronological.

Language definition

Designing a Beginners' Programming Language,
Leo Geurts and Lambert Meertens, 18 pages.

This was the first article on *B*. It clarifies some of the design objectives and describes the result of the first iteration in the defining process. Published in New Directions in Algorithmic Languages 1975, ed. S.A. Schuman, IRIA, Rocquencourt (1976). Available from CWI as report IW 46, price Dfl. 4.00.

Program Text and Program Structure,
Lambert Meertens, 11 pages.

Proposes the method of stepwise refinement as a means to make the structure of program development explicit in the program text. Published in Constructing Quality Software, ed. S.A. Schuman, North-Holland Publ. Co. (1978). Available from CWI as report IW 78, price Dfl. 4.00.

Keyword Grammars,
Leo Geurts and Lambert Meertens, 12 pages.

This is a rather technical derivation of a simple condition for the "keyword skeletons" in a language that guarantees the existence of a simple no-backup parser. It influenced the keyword stucture of *B*. Published in Implementation and Design of Algorithmic Languages, eds. J. André and J.-P. Banâtre, IRIA, Rocquencourt (1978). Available from CWI as report IW 86, price Dfl. 4.00.

Issues in the Design of a Beginners' Programming Language,
Lambert Meertens, 18 pages.

This article describes some unexpected solutions that were found for problems in the third iteration of designing *B*. It reinterpretes the original design objectives in the light of some rejected preconceptions. Published in Algorithmic Languages, ed. J.C. van Vliet, North-Holland Publ. Co. (1981). Available from CWI as report IW 161, price Dfl 4.00.

Draft Proposal for the B *Programming Language,*
Lambert Meertens, 88 pages.

This book is a specification of the whole *B* language, as it arose from the third design iteration. It is, however, rather technical for the casual reader. In addition it contains some thoughts on a *B* system. Published by and available from CWI as ISBN 90 6196 238 2, price Dfl. 14.10. A part of it, the *Quick Reference* also appeared in the *Algol Bulletin* number 48, August 1982.

Taal zonder naam,
Leo Geurts, 2 pagina's.

Een korte schets van een nederlandstalige versie van *B* als aanzet voor een ideale programmeertaal. Gepubliceerd in de HCC Nieuwsbrief, jaargang 5, nummer 6, blz 20-21, juli 1982.

Language and Environment

Ontwerp van een Programmeeromgeving voor een Personal Computer,
Leo Geurts, 13 pagina's.

Dit artikel bespreekt de consequenties van de doelstellingen die bij het ontwerpen van de taal *B* zijn gebruikt bij toepassing op de programmeeromgeving. Benadrukt wordt het belang van een geïntegreerd systeem dat voor verschillende funkties dezelfde communicatiemethode met de gebruiker hanteert. Gepubliceerd in Colloquium Programmeeromgevingen, CWI, Syllabus 30, Amsterdam (1983), prijs Dfl. 22.80.

On the Design of an Editor for the B *Programming Language,*
Aad Nienhuis, 16 pages.

Gives an overview of the design of the first approximation of the *B* dedicated editor. Published by CWI, report IW 248/83, price Dfl. 4.00.

The B *Programming Language and Environment*
Steven Pemberton, 12 pages.

Gives a description of *B* along with some background to it, and some justification for its existence, arguments about simplicity, interactiveness, programmer productivity, and talks about its suitability for use in schools. Published in CWI Newsletter, Vol. **1**, No. 3 (June 1984). Available free from CWI.

Towards a Specification of the B *Programming Environment,*
Jeroen van de Graaf, 23 pages.

This report contains an informal description and a tentative specification of the environment for *B*. Published by CWI, report CS-R8408, price Dfl. 4.00.

Taalprimitiva in B voor Grafisch Editen - een Verkenning,
M. Andreoli, 26 pagina's.

In dit rapport worden mogelijkheden onderzocht commando's aan een taal als *B* toe te voegen om "grafisch editen" (interactieve gegevensinvoer via een grafische interface) op een eenvoudige manier mogelijk te maken. Gepubliceerd door CWI, rapport CS-N8509, prijs Dfl. 4.00.

De programmeertaal B,
Leo Geurts, 3 pagina's.

Korte eenvoudige uitleg van taal en omgeving. Gepubliceerd in de HCC Nieuwsbrief, jaargang 8, nummer 5, mei 1985.

An Alternative Simple Language and Environment for PCs
Steven Pemberton, 9 pages.

This article is the first place to go if you want to know more about ABC. It describes the background to ABC, gives an overview of the language, and presents the ABC editor and the programming environment the language is embedded in. Published in IEEE Software, Vol. **4**, No. 1, January 1987, pp. 56-64, and ABC Newsletter 6.

Programmers Handbook

All articles referenced here were eventually gathered in *The* B *Programmers Handbook* mentioned at the end. A new version of this handbook for ABC should be available this year.

An Overview of the B *Programming Language, or* B *without Tears,*
Leo Geurts, 11 pages.

In informal introduction to the language suitable for people that are already proficient with some other high-level language like Pascal or C. It was published in SIGPLAN Notices Vol. **17**, No. 12, December 1982, or is available from the CWI, report IW 208/82, price Dfl. 4.00.

Description of B,
Lambert Meertens and Steven Pemberton,
38 pages.

Informal definition of *B*, which can be used as a reference book, and as an introduction for people with ample programming experience. Published in SIGPLAN Notices, Vol. **20**, No. 2, February 1985, pages 58 - 76. Available from CWI, note CS-N8405, price Dfl. 6.40.

A User's Guide to the B *System,*
Steven Pemberton, 10 pages.

A brief introduction to using the *B* implementation, including the *B*-dedicated editor. Published by CWI, note CS-N8404, price Dfl. 4.00.

B *Quick Reference Card,*

A single card including all the features of the language, the editor, and the implementation, for quick reference when using *B*. Available from CWI.

The B *Programmer's Handbook,*
Leo Geurts, Lambert Meertens, and Steven Pemberton, 80 pages.

A handbook containing a quick look at *B*, a guide to using the current implementations of *B*, and a description of *B*, thoroughly revised and updated. Published by CWI, ISBN 90.6196.295.1, price Dfl. 12.70.

The ABC Programmer's Handbook,
Leo Geurts, Lambert Meertens, Steven Pemberton, 185 pages.

A rewritten version of *The* B *Programmer's Handbook*, now with a chapter of example programs. To appear.

Programmers Tutorial

Part 2 of the publication that comprises this topic never made it. Together with some additional material the first part was used in several courses. Based on that experience it was decided to write a completely new book, that will soon conquer the world.

Computer Programming for Beginners — Introducing the B *Language — Part 1,*
Leo Geurts, 85 pages.

This is a text-book on programming for people who know nothing about computers or programming. It is self-contained and may be used in courses or for self-study. The focus is on designing and writing programs, and not on entering them in the computer, and so on. It introduces the language, and how to write small programs. Published by CWI, note CS-N8402, price Dfl. 12.70.

Cursus programmeren voor beginners — Een kennismaking met de programmeertaal B, *Deel 1,*
Leo Geurts, 85 bladzijden.

This is a translation of *Computer Programming for Beginners — Part 1*. Dit rapport bevat een beginnerscursus programmeren, gebaseerd op de nieuwe programeertaal *B*. De meeste elementaire programmeertechnieken en de meeste eigenschappen van *B* komen aan bod. De tekst vereist geen voorkennis, en is zowel voor cursussen als voor zelfstudie geschikt. Gepubliceerd door het CWI, Notitie CS-N8407, Dfl. 12.70.

Implementation

Incremental Polymorphic Type-Checking in B
Lambert Meertens, 11 pages.

B allows you to use variables without having to declare them, and yet gives you all the safety that declarations would supply. This paper describes how this is achieved, but is *very* technical. Definitely not for the faint-hearted. Published in the conference record of the 10th ACM Principles of Programming Languages, pages 265-275, 1983, and also by CWI, report IW 214/82, price Dfl. 4.00.

Making B Trees Work for B,
Timo Krijnen and Lambert Meertens, 13 pages.

This describes a method of implementing the values of *B*. It is rather technical. Published by CWI, report IW 219/83, price Dfl. 4.00.

On the Implementation of an Editor for the B *Programming Language,*
Frank van Harmelen, 18 pages.

Gives details of a pilot implementation of the *B* dedicated editor. Published by CWI, report IW 220/83, price Dfl. 4.00.

An Implementation of the B *Programming Language,*
Lambert Meertens and Steven Pemberton,
8 pages.

This gives an overview of the implementation and some of the techniques used in it.
Published in USENIX Washington Conference Proceedings (January 1984). Available from CWI, note CS-N8406, price Dfl. 4.00.

The Cleaning Person Algorithm,
Tim Budd, 12 pages.

This paper describes an algorithm that permits values to migrate easily between primary and secondary memory (or disk), permitting the *B* system to act as if the amount of memory was essentially limitless. Published by CWI as report CS-R8610, price Dfl. 4.00.

Newsletter

All issues of the newsletter are still available free from CWI. Here you find the contents of all issues, including this one.

Issue 1, August 1983.
- Available Publications about *B*
- A Short Introduction to the *B* Language
- A Glimpse at the *B* Environment
- Implementation Plans for *B*

Issue 2, June 1984.
- The Mark 1 Implementation
- Plans for the Near Future
- Examples of *B*
- The Highlights of *B*
- A Comparison of Basic and *B*
- A Comparison of Pascal and *B*

Issue 3, January 1985
- IBM PC Progress
- What is in the name of *B*?
- A File-Maintenance Program in *B*
- A Proposal for Matrix/Vector Functions in *B*
- Speeding up the *B* Implementation

Issue 4, September 1985
- New Unix Release
- *B* for the IBM PC
- Eh? *B* be 'ABC', see?
- A Program Example: Polynomials
- *(Extremely)* Simple Logic Programming in *B*
- `~1 <> 1`, A Nice Distinction?

Issue 5, October 1986
- >>> From *B* to ABC
- Letter to the Editor
- The Cleaning Person Algorithm
- Backtracking in *B*: the Budd Challenge
- Primality Testing in *B*

Issue 6, May 1988
- Progress towards ABC
- A Chess Program in *B*
- Approximate Numbers
- My Experiences with ABC
- An Alternative Simple Language and Environment for PCs
- Publications about *B* and ABC

Ordering

CWI publications can be ordered from

Publications Department
CWI
POB 4079
1009 AB Amsterdam
The Netherlands

You will be invoiced. The prices quoted exclude postage and packing, and for foreign orders there is an additional charge of Dfl. 8.50 to cover bank charges.