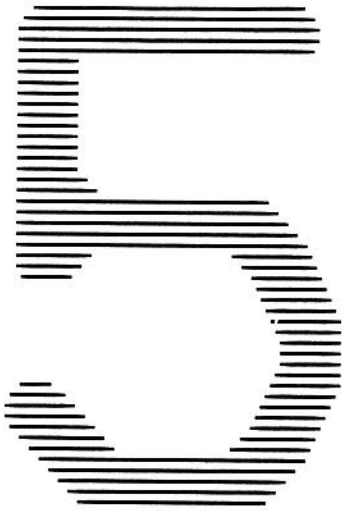
THE ABC NEWSLETTER

ISSN 0169-0191 CWI, Amsterdam Issue 5, October 1986

CONTENTS

Some Notes
New Publications
>>> From *B* to ABC
Letter to the editor
The Cleaning Person Algorithm
Backtracking in *B*: the Budd Challenge
Primality Testing in *B*
Order Forms



Some Notes

The newsletter is intended to provide information about *B* 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:

```
timo@mcvax.UUCP
```

Otherwise, articles and mail should be sent to

The a*B*c Newsletter
Informatics - AA
CWI
POB 4079
1009 AB Amsterdam
The Netherlands

New Publications

There is only one new publication since the last *B* newsletter. An abstract of it can be found in the corresponding article in this newsletter.

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. 3.90.

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. 7.50 to cover bank charges.

THE ABC NEWSLETTER

From *B* to ABC: the Salient Changes

Lambert Meertens

CWI

The past year, we have been working hard on the revision from *B* to ABC, making an inventory of proposed changes (using the experience collected over several years of use) and debating their pros and cons. There are two problems in that kind of undertaking. One is that new proposals may have unexpected drawbacks, and so you must be careful in adopting changes, more so than for a first 'trial' version of a language. This danger can be minimised by only considering minor changes, and in fact we felt no urge to do otherwise, since we were already unreasonably happy with unrevised *B* and only wanted to give a final high polish. The second problem is that two or more nice suggestions, fine enough by themselves to be voted in by acclamation, may be incompatible if taken together. Here, it is a good thing to 'revise by repeated approximation': take preliminary decisions on the proposals, and let the result simmer for some time to see where conflicts arise, revise the revision again etc. until stability.

We think we have now reached a point of stability, more or less. Not fully, since (a) we are still open to further input, especially if it does not entail a complete upheaval of everything that *B* stands for, and (b) we decided to categorise the changes in three classes: 'Hurry', 'Nice', and 'Wait'. This has to do with the fact that we want to bring out a next release of our system, as soon as possible, that understands (revised) ABC.

Now, some—in fact, not a few—of the proposed changes are so minor that it is hard to explain the difference with the current situation. These may have to do with behaviour in marginal situations for which user-oriented semantic descriptions, like in The *B* Programmer's Handbook, are naturally ambiguously phrased. To give one example, in order to allay the reader's now piqued curiosity: The Handbook does not specify where the extra space character should go if a text of even length is centred in a field of odd length, as in `"at"><7`. This is defined in the much more precise Draft Proposal, the Holy Writ for implementers, and there is now a proposal to make a change there. We do not feel compelled to implement such minor changes in the first ABC'd release of the system. Therefore, this change has status: 'Wait'—that is, no 'Hurry'. Some other proposed changes are not that minor, but are strictly 'upwards compatible': they do not invalidate any existing *B* code, but impart a meaning to hitherto undefined things. This refers both to certain new syntactic forms, as to semantic extensions of existing constructs. Here, again, we felt free to confer the status 'Wait' to the proposals.

Other proposed changes are very visible and so got the label 'Hurry'. Here, existing *B* code is invalidated, either because the syntax has changed, or because the syntax is still the same but the semantics is different. Of course, we have tried to minimise such changes, especially in the group of 'same form, different content'. If we adopted one of those, we felt the merits to be substantial enough to outweigh the inconvenience to the existing user community (and don't forget that we are big-time users of *B* ourselves). To minimise the pain, the next release will in some cases still accept the old syntax style next to the new one, similar to the behaviour of the (then only) C compiler when the syntax of, among others, initialisations was changed. Possibly, this release will also come with some 'off-line' conversion tool. (The first *B* implementation pulled another stunt in this area. It was finished earlier than the Draft Proposal, and was updated along with the final polishing of that document. If it recognised a piece of syntax that got changed recently, it not only accepted the obsolete form, but recast it into current syntax and stored that back. Very convenient, that; we shall probably not go as far out of our way this time.)

The proposals that we categorised as 'Wait' are generally of minor import to the user, true trifles. Some changes, however, among those of the strictly-upwards-compatible kind, are such improvements that we would really like to bring them out in the very first next release. These constitute the 'Nice' category. Whether we succeed in incorporating all of them, we doubt, but most will be in there. (There are degrees of 'niceness' that will determine our priorities.)

This long, but perhaps interesting by itself, digression was needed to explain what I meant with 'stability'. There is another categorisation: proposals can be in the (currently) 'Accepted' or 'Rejected' sets, but also in the (as of yet) 'Undecided' set, which is where they all were, initially.

The point where we are now, is that all proposals still in the Undecided category have the label 'Wait': whatever the final decision, it will not influence the next release of the system, nor the wording in user-oriented documentation. In most of these cases, the proposal is still undecided because it appears to have possible ramifications that we could not quite fathom—safer, then, to let it stay in limbo. If one of these Waiters turns out, on further examination, to bite a Nice or even Hurried change, it will be summarily rejected.

Some statistics. We started out with a collection of some 69 proposals. (Sometimes it is a matter of taste if you count something as one single proposal or as two closely related proposals.) Of these, 36 are now Accepted and 15 Rejected, leaving 18 Undecided. Of the thirty-six chosen ones, 22 are Hurried, 7 are Nice, and another 7 lie in Wait.

Below, I only describe the more important changes. (Some of the Accepted—Hurry ones are too petty to mention here. A full account is available on request.)

- The most prominent change, one that will in fact influence every existing *B* program, is that `HOW'TO` will henceforth be spelled `HOW TO`.
- Not less dramatic is the following change: instead of `YIELD` we have now `HOW TO RETURN`, and instead of `TEST` similarly `HOW TO REPORT`. So all units will start with `HOW TO`. It is no longer appropriate to call command definitions 'how-to-units' then, and we shall rename them 'command-definitions'. Instead of 'yield-units' we have then 'function-definitions', and 'test-units' become 'predicate-definitions'.
- The signs `'/'` and `'#'` are allowed in names (tags) and keywords, and spaces are not. In current *B*, the `'/'`-sign provides a way to introduce pseudo-spaces, as in `'last'time'seen'`. In the future, a `'.'`-sign can be used for the same purpose: `'last.time.seen'`. This is an addition: the use of `'/'` and `'#'` stays allowed since it is handy for such locutions as `'x'`. For a long time, we hesitated between the signs `'.'`, `'_'` and `'~'`. The decisive advantage of the `'.'` is that it is a lower-case character on all keyboards (except the Struldbrugg-YUC models).
- Moreover, the name of a user-defined command will no longer consist of its first keyword, but of all keywords together preceding the first parameter, if any (and otherwise simply all keywords). So then users can define and use commands like `PUT OFF task`. The predefined command `SET'RANDOM` is also changed into `SET RANDOM`. A further proposal is to consider the whole 'keyword skeleton' as determining the command name, allowing `PUT a OVER b` next to `PUT a IN b`. This last business is Undecided: we fear adverse effects on the friendliness of the editor.
- Comparison between exact and approximate numbers will be changed in accordance with the proposal in issue 4 of The *B* Newsletter. Henceforth, `root 4 = 2` shall succeed.
- The parameter mechanism for user-defined commands, currently call-by-name as in ALGOL 60, will be replaced by copy-restore. Side effects of evaluating or locating a parameter (which, because of the scratchpad-copy semantics of expressions, are already restricted to run-time errors, I/O, or changing the state of the pseudo-random-number generator) are thus incurred only once. The main change, therefore, is in the reduction of aliasing. Moreover, putting different values in actually identical output parameters will be a (dynamically) signalled error, exactly as happens already in

```
PUT 1, 2 IN i, i.
```

It will remain an error to put a value in a formal parameter that does not correspond to an actual *target* parameter.

- Also, multiple (collateral) names will be allowed as formal parameters in a command definition, e.g. `HOW TO SWAP a, b`, so that the parameter mechanisms of commands and of functions and predicates are almost completely unified—differences are that the parameters of functions and predicates are *always* copied in, and that there is no 'restore' (which would be invisible anyway because of the scratchpad copying), so that the formal parameter can be used as a local target, even if the actual parameter is a formula. If one of the fields of a multiple formal parameter is assigned to, but not all fields, then either the actual must be multiple as well, or it may be single but must then be already initialised at the time of the call. This prevents the creation of compound locations not all of whose fields have a value.
- A new command `PASS` will serve to specify the dummy (null) action. Empty command-suites will no longer do: each command-suite must contain at least one command. Next to improving clarity, this also does away with the problem that an indentation error, as in

```
IF x > 1:
PUT x-1 IN x
```

gets uncaught; this will cause a syntax error in the future. (In fact, the editor will leave a hole for the missing command-suite.)
- The `PARSING` construct will be replaced by a

somewhat less general, but in other respects more powerful and in any case more user-friendly construct, although probably not in the next release. Even then, PARSING will be gone (at least from the documentation; it may temporarily remain in the system to alleviate switch-over problems). The big problem with PARSING was that it was so excruciatingly slow that no-one in their right minds used it in the first place. This was not a matter of a more efficient implementation; its inefficiency is inherent to its semantics. The new form, then, will be a FITS test of the form

tex FITS *inf*

in which *tex* is a textual-expression and *inf* is a new animal, dubbed an 'input-format'. May some examples suffice instead of a formal definition:

```
"$ 123.45" FITS "$`x EG 0`"
```

will succeed and put the value (number) 123.45 in x. This x is a bound tag, of course: its scope is the part that is reachable only by dint of the success of the test.

```
"f 123.45" FITS "`cur RAW``x EG 0`"
```

succeeds as well, and also puts the text "f " in cur.

```
line FITS "`k RAW`: `i RAW`"
```

has the same meaning as the test in current *B*:

```
SOME k, c, i PARSING line HAS c = ": "
```

except, of course, that the dummy target c has gone. Thus, the first occurrence of ": " is used to split the text contained in line. The FITS construct fails if no parsing will make the expression 'fit' the input-format; moreover, the parsing strategy is 'greedy', just as with the present PARSING construct.

If the body of the input-format consists of a single 'input-conversion', the enclosing quotes may be dropped. So, to convert a text to a pair of numbers, the following test suffices:

```
t FITS x, y EG 0, 0
```

There are some murky matters not yet completely cleared up (like whether

```
"12321" FITS "`x EG 0`2`y EG 0"
```

is in error or puts 1, 321 in x, y, or perhaps 123, 1), which is one of the reasons why this might not yet show up in the next release.

- Input-formats are also allowed following READ. In fact, the existing forms READ x EG y and READ x RAW become then special cases of this. The execution of a command like

```
READ "Article: `a RAW` (Code: `c EG 0`)"
```

will prompt the user with

```
Article: ? (Code: ?)
```

Again, this is not expected for the next release.

- The command CHOOSE will be replaced by a monadic function choice, making it possible to do something like

```
TAKE choice exits
```

in a single command. Similarly, DRAW is replaced by a zeroadic function random, allowing the use of expressions like

```
(log(1-random))*sin(2*pi*random)
```

(which has a normal Gaussian distribution).

- The following kind of iteration may be added:

```
FOR [k]: i IN table: ...
```

to allow an easy way to traverse the keys and associated items of a table together. Although rather nice, this change is still on the waiting list of undecided proposals.

- The E in the floating-point number format is going to be replaced by a lower-case e. The primary reason for this change is the improved legibility; a secondary reason is the disappearance of an anomaly in the editor to cope with the case of commands with a keyword E. Furthermore, on input, numbers in this format will be handled as *exact* numbers; thus, 1e9 evaluates to the same number as 1*10**9. If an approximate number is intended, the user has to indicate this explicitly, as in ~1e9.

- We have noticed that errors of the form exemplified by

```
SELECT:
    line|8 = "CONTINUE": GO ON
    line|3 = "BYE": QUIT
    ELSE: UNKNOWN COMMAND
```

are rather more common than we had anticipated. The problem is that if line contains the text "BYE", then the evaluation of line|8 causes a run-time error. In ABC, t|n will be equivalent to t|min{n; #t} in current *B*, so that the code fragment above works as expected. Similarly, t@n will be equivalent to t@max{n; 1}. If n < 0, the evaluation of t|n will still signal an error, and similarly for n > #t+1 and t@n.

- The evaluation of a list-display like {1..n} will no longer signal an error in ABC if n < 0; its value is simply the empty list. So, unlike in current

B, the following is a safe way to compute the intersection of two non-empty ranges of integers `r` and `s`:

```
{max{min r; min s}..min{max r; max s}}
```

This change, as well as the following, is only 'Nice'.

- The following will all be valid ABC:

```
{1..i-1; i+1..n}
{-a..a; -b..b}
{1; 2; 4..9; 5; 11}
```

- The priorities of the operators will be slightly changed, allowing, among others, the currently not permitted

```
2 round x >> 7
```

but ruling out the currently allowed

```
t|2^^n
```

(and some ambiguities permitted by the Draft Proposal as well).

- The name of the function `atan` will be changed to `arctan`. Possibly, also `arcsin` and `arccos` will be added. Instead of dyadic `x atan y`, there will be `angle(x, y)`, and then a Nice addition is its natural companion `radius(x, y)`, computing the same as `root(x**2+y**2)`, but offering for many applications a far more convenient diction. In particular, if coordinates `(x, y)` are stored as table items, say, you can use expressions like `angle t[i]` without having to unpack the compound first. The value of `angle(0, 0)`, unlike that of `0 atan 0` in unrevised *B*, is not undefined, but zero.
- Another proposal in the Nice category: to facilitate computation with degrees instead of with radians, dyadic versions of all trigonometric and cyclometric functions are added in ABC. The first parameter is the value of a full circular arc expressed in the units desired. For example, since a full circle is 360°, use `360 sin 15` to compute the sine of 15°.
- Some new functions will simplify text handling, especially, but not only, user-supplied interactive input text. To convert to lower and upper case, we have in ABC:

```
lower "E.E. Cummings"
upper "L19: cvtbl *-12(fp),r0"
```

returning respectively:

```
"e.e. cummings"
"L19: CVTBL *-12(FP),R0"
```

The function `stripped` will return its argument with all leading and trailing blanks removed. To see if the reply to a question is any of *"yes"*, *"Y"*, *" yep!!"*, etc., it suffices to test if

```
(stripped lower reply)|1 = "y"
```

Nice would be a function `words`, or `split`, that takes a text and splits it (on the blanks) into a sequence of words.

- The formula `n th'of s`, with its attendant abominations like `2 th'of s` and the frequent mistake `n'th of s`, will be traded in for `s item n`. The order of the parameters is more in line with other selections like `s[n]` and `s@n|1`.

We have also designed several extensions to ABC, for example for matrix/vector functions (see issue 4 of The *B* Newsletter) and for graphics. These will not be in the next release. In fact, if they will be offered in the future, they will be possibly be separate additions with their own manuals that people may or may not acquire.

Going beyond this, we are also considering extensions that are primarily intended for applications designers. Among these are a facility for dealing with data entry (a generalisation of the `FITS` test), and also a possibility for adding new, user- (or in fact designer-) defined, types (that are atomic to the end-user).

There is a designed addition that we have not yet classified; maybe it should be incorporated in 'general' ABC and not be part of an extension. It is the addition of a new sequence type. Although it is possible to use tables to model sequences, some natural operations on sequence are difficult to handle; in particular, joining two sequences and extracting a (contiguous) subsequence. The proposed addition would introduce sequences as arbitrarily long, ordered but not sorted, collections of items of the same type. A possible syntax for sequence-displays is that of

```
{0;; 1;; 0;; 2;; 0;; 1;; 0;; 3},
```

that is, like a list-display except that the semicolons are doubled. Not an obvious candidate for a beauty contest, but we haven't been able to think of anything nicer that does not conflict with existing syntax. The notation `{}`, already doing double duty for empty lists *and* tables, would acquire a third meaning: that of an empty sequence. The displays `{a}` and `{a..b}`, currently only lists, would also get an extra meaning: they would also be acceptable sequence-displays. The static type-finding algorithm is easily amended to cope with this. The generic operations on texts, lists and tables (iterating with `FOR` or with

SOME/EACH/NO, the function item and the monadic and dyadic versions of #, min and max) would extend to sequences. Other operations on sequences would be: the join function ^ and the repeat function ^^, and the trims @ and | (now only defined on texts). A new monadic function items can be used to turn a text, list or table into a sequence; the effect of FOR i IN tlt: ... is the same as that of FOR i IN items tlt: .... Like keys, this function takes constant time, independent of the size of its parameter. (Using items on a sequence is also permitted; the result is the same as the parameter supplied.)

Although an obvious possibility to consider, it will probably not be the case that texts are a special case of sequences. There is a difference between "abc" and {"a";; "b";; "c"} since identifying these would imply that "a" is the same as {"a"}, which looks undesirable. But this needs further study; maybe there is a satisfactory and consistent solution to this.

Letter to the editor

Dr. Hanno Wupper

Rechenzentrum Ruhr-Universität Bochum

Dear Colleagues,

Let me quickly give you some remarks about the *B* system, which we have been using here now for some months.

First of all: the language, the editor and the handbook turned out to be even much better than I dared to hope in my secret dreams. I have used the system in a course for school teachers (sekundarstufe II), who are interested in computer science but have not had previous knowledge of programming languages. The course has been most interesting, and now some people have good reason to think that such an editor, such a well designed and well described language are the state of the art. (They will have some bad experiences when they investigate the machinery they find at their schools ...)

Most useful in teaching was the UNDO-key: whenever someone was lost in a hopeless situation, I could use it to slowly go back and reconstruct the mistakes in thinking that eventually lead into that situation ("Look, when you tried this, you must have thought that ... but ... "). This possibility of backtracking and reasoning about the decision tree is much more important than the simple fact that errors can be undone without too much typing. Another good thing is that, in contrast to text editors, program transformations are the more difficult the greater the change in semantics is, but, of course, you know that.

Believe me that I am really enthusiastic, and please do not misinterpret the following criticisms, which will come in the groups (A) Language, (B) Name, (C) Editor, (D) Efficiency, and (E) Bugs.

A. Language

Nearly everything seems to be perfect. Perhaps you should not revise the language too early but rather collect ideas for some time. At the moment, I see few reasons to ask for "more, more".

But: did you ever think about national variants? If German pupils (and German teachers) write their first programs, they will not have enough knowledge and interest in English to be able to design linguistically beautiful texts. If the keywords could be in German, they would more easily get the feeling that the inner beauty of good programs can be reflected by their representation.

While I thought about this all, an unorthogonality occurred to me: nearly all of the keywords fall into one of two classes: proper commands, which could easily be redefined, such as `CHECK`, `PUT`, `SET'RANDOM`, and particles that structure the language, such as `HOW'TO`, `QUIT`, `IF`, `SOME`. Only two things do not fit into this scheme: `READ` is the only thing where the first keyword is not sufficient to distinguish between two commands, and `WRITE` uses an entirely surprising and unique syntax.

B. Name

The existence of this implementation will propagate the name *B* to many people. Will they not be irritated by a change of name?

C. Editor

It is easy to put something into the copy buffer by mistake which novices will find very difficult to get out again. If, for example, one wants to move the (long and complicated) expression from

`IF` *expression* :

somewhere, you might by mistake copy the colon as well and then find that

expression :

will nowhere be accepted. There is no *obvious* easy way to, say, clear the whole buffer.

D. Efficiency

Some operations on lists and especially some arithmetic operations on machines without 8087 coprocessor are *very* slow. The slowness itself is not such a problem for beginners who do not know implementations of other languages, but often they find it difficult to decide whether anything is going on at all. Perhaps some flashing mark would be helpful, or "wait wait"?

The storage limitations and inefficiency of the present implementation would not be a problem, if units that have been developed and tested could be compiled. Has anyone ever attempted to write the machine independent parts of a *B* compiler? *B* would be an obvious language. Is something like that available?

E. Bugs

1. After `bio -i`, `bio -r` is necessary, otherwise all targets will be forgotten.
2. After you get the message that the actual unit is too large (and after you pressed LOOK and then deleted something) everything looks o.k. and you can beautifully type on. Only, typically, thereafter your large unit disappears without warning and without leaving anything behind.

One last question regarding the Handbook: why did you not attempt to explain as much as possible of the commands and operations in the form of a standard prelude?

Yours,

Hanno Wupper

The *B* Group Replies

The following is the substance of our reply to Dr. Wupper.

A1. Language revision

The revision of the language is the result of 5 years experience with the current version of the language, so we certainly don't regard it as premature! Furthermore, it is being done to improve the language: it is not a case of "more more", but rather "better".

A2. National Variants.

Yes, we have thought of national variants, and actually already have a French version, which proved very popular at a recent French educational software exhibition in Paris. We also plan a Dutch version, and would be very pleased to co-operate on the preparation of a German one.

With regards to `READ EG` and `READ RAW`, this problem is solved in the revision. With respect to `WRITE`, well, the only addition is the `/` character to specify new lines, which isn't so problematic; you may not redefine built-in commands anyway.

B. Name

The name has been a constant problem for us. The first version of our *B* was defined in 1975, and we have been using the name since then, although always saying that it was only a working title which would be changed when the language reached its final form. This was so that earlier versions of the language wouldn't taint the final version. Unfortunately, around the same time, another group designed a language that they called B, and due to the success of its successor (C) in the ensuing years, that B has become famous, in name at least. Thus we are constantly plagued by people telling us that there is already a language called B, or by people who ignore our *B*, thinking it is the older one. So really the name must be changed, and for us the name ABC appears to be the best candidate, suggesting as it does the simplicity that we have strived to achieve, (plus at least containing the old name, even if it doesn't start with it).

C. Editor

We are currently busy redesigning the editor. Probably the following version will be far more oriented towards the abstract parse-tree rather than the concrete tree, so that you will be able to focus on the expression of "`IF expression:`" far easier than on the expression plus the colon. Since the colon is always suggested by the editor, we expect that this will be more convenient.

It is true that there is no convenient way to clear the copy buffer, but on the other hand there is no reason why you should want to (except perhaps to get rid of the `[copy buffer]` message from the screen), since you can just copy something over the top of it.

D. Efficiency.

Agreed that the current version is very slow. The next release promises to be faster (it currently runs about twice as fast). Yes, perhaps it would be an idea to indicate that something is happening: the Macintosh watch is something along these lines.

We have lots of ideas about implementing parts of *B* in *B*, however, we have no current plans to produce a compiler, since many other things, such as a fuller environment, are higher on our list of priorities.

E. Bugs.

Thank you for the report about `bio -i`. It will be fixed.

The message you get when store is filling up is not that the unit is too big, but that store is nearly used up. You should then exit the editor as soon as possible. But the documentation and the error message should make this clearer.

We don't think that defining as much of *B* as possible using a standard prelude would help much. It is our feeling that a description in words, though less precise, is easier to understand. Furthermore, there are very few commands you can really treat like this; most would still have to be described with words.

The Cleaning Person Algorithm

Tim Budd

CWI

The *B* language is designed to be used by individuals having little or no previous programming experience, working on a personal computer. A primary aim of the language is to provide a simple and easy to understand tool that is nevertheless powerful enough to facilitate the solution of non-trivial problems. To this end, many of the details of the underlying machine that conventionally a programmer must be concerned with can be ignored in *B*. For example, rational numbers are maintained internally in an unbounded form; thus the user does not need to be aware of the machine "word size" or of exceeding the hardware precision of numerical values. In a similar manner, the user should not need to be aware of the limitations imposed by a finite memory on a computer; it should appear to the user as if the amount of memory available is essentially limitless.

There are several ways in which an executing *B* system can run out of memory. The most obvious fashion is by creating large objects. Just as there are no intrinsic limitations on the size of numbers, there should be no reason why a user could not create a table containing many thousands of entries, even if the size of the table exceeded available memory. However, memory can also be quickly consumed by a proliferation of small objects. This will be particularly true in the revised ABC system, since the values of all identifiers will be recorded at the end of each command, and these values remembered for some period of time. Saving the memory in this manner permits the user to back up execution or to make changes to previous commands. These facilities will not be discussed here; it is only necessary to note that this feature tends to consume a considerable amount of primary memory that is only infrequently accessed.

In short, a problem can arise where there are too many objects and too little memory. The Cleaning Person algorithm is intended to address this problem. As the *B* system is executing, a second process (*the cleaning person*), running in parallel, attempts to move objects from primary memory to secondary storage (such as a disk). Some special characteristics of *B* objects, notably that once created they are never modified, make this solution feasible. This paper presents only an informal overview of the cleaning person algorithm. A more detailed description of the algorithm can be found in [1].

A basic assumption of the cleaning person algorithm is that all of memory can be viewed as a (not necessarily binary) tree. In general, the idea is that the cleaning person attempts to move sections of this tree to disk. One constraint is that nodes that have been moved to disk can only point to other nodes that have also been moved to disk. A consequence of this is a node can be moved only after all its children have been moved.

When a node is moved to disk, a special mark (a single bit in the tag field) is set in the copy of the node that remains in memory. An entry is made in an object called the *disk mapping* table. This entry contains a pointer to both the location of the node in memory and its location on the disk. Basically, the disk mapping table provides the logical mapping between nodes that are in memory and those that are on disk.

When a pointer to an object that has been marked as being moved to disk is found, the pointer is changed to point instead to the disk mapping table. When the reference count on the original node goes to zero, all pointers to it have been found and changed to point to the disk mapping table, and the memory occupied by the node can be released. When the reference count on an entry in the disk mapping table goes to zero, it implies that all pointers to the associated node have themselves been moved to disk, and thus the disk mapping table entry can be removed.

In this manner the disk mapping table requires entries only for those nodes for which pointers still exist in memory. It is not necessary to maintain entries in the disk mapping table for those nodes for which all pointers have themselves been moved to disk. A hashing scheme, that can be triggered by either a disk address or a memory address, is used to reference entries in the disk mapping table.

The cleaning person is most useful on archival material, that is, portions of memory that are likely to be only infrequently used. In general it will be difficult to determine what nodes in memory satisfy this criterion. A trick using "recently used" bits is employed to insure that nodes that are moved to disk have not been

accessed in the recent past, under the assumption that they will therefore likely not be accessed in the near future.

Of course, even the most dusty archival material is sometimes accessed, and a technique must be provided for bringing nodes back from disk into memory. This involves once more setting up a disk mapping table entry, and copying the information back into a possible different location in memory. The address of the information on disk is retained, so that if again after a while it is found to be not recently referenced, pointers will slowly change from the address in memory to the address on disk.

One notable feature of the cleaning person algorithm is that it is independent of the rest of the *B* system, including the memory manager. Furthermore it can be structured in small discrete steps, each step leaving memory in a consistent state. Thus an attractive scheme would schedule the cleaning person when no other activities are pending, such as between characters during user input; or when memory is almost exceeded.

These, and many more, issues are addressed more fully in the technical report describing the algorithm:

[1] Tim Budd, "The Cleaning Person Algorithm", Report CS-R8610, Centrum voor Wiskunde en Informatica, February 1986. (12 pages).

Backtracking in *B*: the Budd Challenge

Steven Pemberton

CWI

In his article "(Extremely) Simple Logic Programming in *B*" in issue 4 of the *B* Newsletter, Tim Budd presented a *B* program for solving the "Farmer, Wolf, Goat and Cabbage in a boat trying to get to the other side of the river without eating each other" problem. The problem involves the four named trying to cross a river with a boat that can only take two of them at a time, with the added complication that only the farmer can row, and that the goat will eat the cabbage, and the wolf will eat the goat, if they are left without the farmer's supervision.

Tim's program uses backtracking, and in the way it is formulated takes advantage of the fact that functions in *B* have no side-effects, in effect doing the backtracking automatically for you. However, the program does have the 'side effect' of writing the results out as it goes along, rather than returning the result. When pointing this out, Tim says:

> *This is a two-edged sword, however, since in some cases one would like to modify the global environment and that then becomes more difficult.*

In other words, if the effect you want to have is more than just writing your answer out, bad luck.

The purpose of this article is to show that it is not at all difficult to write such a program, both using *B*'s automatic backtracking *and* returning a result.

Just for interest's sake, I shall also modify some other aspects of Tim's program, to compare how different approaches to data-structures affect the program, but this in no way affects the main point of the article, which is to demonstrate returning results in a backtracking program.

First of all, I'm going to alter how the positions of the participants are represented: Tim used a compound like `(1, 1, 1, 1)` to represent all four participants being on the north bank: each field was the position of one participant in the order: farmer, wolf, goat, cabbage; a zero represented the south bank.
Rather than represent north and south by 1 and 0, I shall use the texts `"N"` and `"S"` (I would have used `"north"` and `"south"`, but this makes some of the lines below wider than will fit on the page). Tim said that he chose the integers to make the conversion from one to the other easier. For the text representation, we have to alter the definition of the function `opposite`:

```
YIELD opposite position:
   RETURN {["N"]: "S"; ["S"]: "N"}[position]
```

Just to test it:

```
>>> WRITE opposite "N"
S
>>> WRITE opposite "S"
N
```

(Alternatively, a global target could contain the above table, and be SHARE d wherever needed.)

Now a more explicit data-structure can be used to represent the positions, such as a table:

```
{["cabbage"]: "N"; ["farmer"]: "N"; ["goat"]: "N"; ["wolf"]: "N"}.
```

Next we need an easy method of altering the set of positions when a boat-load goes to the opposite bank. If we represent a boat-load by a list of who is in the boat, for instance `{"farmer"; "goat"}` then we can use the following function to take a position and a list of travellers and return the new position:

```
YIELD position altered'for occupants:
   FOR occupant IN occupants:
      PUT opposite position[occupant] IN position[occupant]
   RETURN position
```

```
>>> PUT {} IN start
>>> FOR participant IN {"farmer"; "goat"; "wolf"; "cabbage"}:
        PUT "N" IN start[participant]
>>> WRITE start altered'for {"farmer"; "goat"}
{["cabbage"]: "N"; ["farmer"]: "S"; ["goat"]: "S"; ["wolf"]: "N"}
```

We also need a way to see if a given position is 'safe', that is to say that the goat is not left alone with the cabbage, and that the wolf is not left alone with the goat:

```
TEST safe pos:
   REPORT farmer'with'goat OR goat'out'of'mischief
farmer'with'goat:
   REPORT pos["farmer"] = pos["goat"]
goat'out'of'mischief:
   REPORT pos["cabbage"] <> pos["goat"] <> pos["wolf"]
```

(Notice the shorthand `c<>g<>w` instead of `c<>g AND g<>w`. Another way of writing the same is `g not'in {c; w}`.)

Now, the new version of the program to solve the river problem, instead of printing out its results as it goes along, will return the answer as a value. This result is a sequence of boat-loads necessary to solve the problem.

Sequences are typically represented in *B* as tables with the elements of the sequence as associates, and integers as keys. Elements are then added to the sequence with a command such as

```
PUT element IN sequence[#sequence]
```

so that the first element is stored at key `[0]`, the second at `[1]`, and so on.

In this program, the result is a sequence of boat-loads and so it will be represented as a table with integers as indexes, and boat-loads as associates. The only difference is that the boat-loads are produced by the program in reverse order to how they will be printed out, and so the elements will be added at the *head* of the sequence. This can be done with the command `PUT element IN path[-#path]`, which we can make into a function:

```
YIELD path with element:
   PUT element IN path[-#path]
   RETURN path
```

An example of a sequence of boat-loads is

```
{[-2]: {"farmer"; "goat"}; [-1]: {"farmer"}; [0]: {"wolf"; "farmer"}}
```

representing the farmer taking the goat across the river, returning alone, and then taking the wolf.

The program either succeeds, and produces such a path, or the (sub-)problem has no solution, and so it should fail. To represent these two cases, the program returns a compound, consisting of a text, either `"success"` or `"failure"` indicating whether it has succeeded or not, and then the path, empty for failure, and the solution for success*. To make life easier, let's have a command to print out the result (this is just a simple version, it could be made prettier):

```
HOW'TO PRINT result'and'path:
   PUT result'and'path IN result, path
   SELECT:
      result = "failure":
         WRITE "No solution" /
      ELSE:
         FOR move IN path:
            WRITE "Move", move /
```

*At first I had the program only return a path, and used an 'impossible' path, such as `{[1]: {}}` to represent failure. I quickly saw this as a typical programmer's kludge, and replaced it with the cleaner solution here.

The original program took as parameter the target position of the four, and then searched *backwards* to see if there was a way that that position could be reached from the start position (which was 'hard-wired' in the program). It was necessary to search backwards, because, like the new version, it produced its answers backwards, and so the two backwardses gave the right result. Another change I have made to the original program is to make both the start and the target positions parameters, and to make the program search from the start position for a path to the target position. As I have already said, it produces its answers backwards, but stores them in the result backwards, so that they still come out the right way round.

The final change is that now 'allowable' boat-loads are not hard-wired in the program, but represented as a list: a maximum of two can fit in the boat (it's a very large cabbage), and furthermore only the farmer can row:

```
>>> PUT {} IN allowable
>>> INSERT {"farmer"} IN allowable
>>> FOR other IN {"cabbage"; "goat"; "wolf"}:
        INSERT {"farmer"; other} IN allowable
```

We can now write a function that returns which boat-loads are possible from the current position:

```
YIELD possible'from position:
   SHARE allowable
   PUT {} IN result
   FOR boat'load IN allowable:
      IF EACH occupant IN boat'load HAS on'same'side:
         INSERT boat'load IN result
   RETURN result
on'same'side:
   REPORT position[occupant] = position[min boat'load]
```

So now, after that introduction to the data-types, we can see the program proper. You'll notice comparing it with Tim's original, that the main body is more or less the same, except that results are being returned rather than tests being reported. The main difference is in the refinement to try the next move. Tim's program tried each of four refinements until one succeeded. The new version tries each of the allowable boat-loads until one succeeds. If none succeeds, it fails. A marginal issue is how to treat the case where the aim is reachable, but not safe in itself. Tim's version accepted it, and printed the result; this version by reversing the order of the first two IF commands, does not accept it.

```
YIELD position path'to aim:
   SHARE looking
   IF position in looking OR NOT safe position: RETURN failure
   IF position = aim: RETURN success
   INSERT position IN looking
   RETURN next'move
next'move:
   FOR boat'load IN possible'from position:
      PUT new'position path'to aim IN result, path'
      IF result <> "failure": RETURN result, path' with boat'load
   RETURN failure
new'position: RETURN position altered'for boat'load
success: RETURN "success", {}
failure: RETURN "failure", {}
```

(Remember that `looking` is used to prevent the program searching for a solution to a position it is already trying to solve — otherwise you can get an infinite loop.) Now to try it out:

```
>>> PUT {}, {} IN start, aim
>>> FOR occupant IN {"farmer"; "goat"; "wolf"; "cabbage"}:
       PUT "N", "S" IN start[occupant], aim[occupant]
>>> PRINT start path'to aim
```

```
Move {"farmer"; "goat"}
Move {"farmer"}
Move {"cabbage"; "farmer"}
Move {"farmer"; "goat"}
Move {"farmer"; "wolf"}
Move {"farmer"}
Move {"farmer"; "goat"}
```

Now to try it with a different aim: the farmer has to get the cabbage to the other side (to sell it to the hungry computer programmer who lives there):

```
>>> PUT start IN aim
>>> PUT "S" IN aim["cabbage"]
>>> PRINT start path'to aim
Move {"farmer"; "goat"}
Move {"farmer"}
Move {"cabbage"; "farmer"}
Move {"farmer"; "goat"}
```

Now, finally, the farmer is fed up with goats, cabbages and wolves, and wants to be alone on the other side:

```
>>> PUT start IN aim
>>> PUT "S" IN aim["farmer"]
>>> PRINT start path'to aim
No solution
```

Alas, poor farmer.

Producing the result forwards

The question arises as to *why* the program produces the result backwards. The answer for Tim's version of the program is that you don't know until you've reached the goal whether the current position is on a successful path, and so you can't write anything until then. With the new version you do have the option of producing the results forwards. To do it you have to pass to `path'to` not only the current position, but how you got to it (as a path). Just to show you how it would look, here is `path'to` altered in that way. The function `with` would also have to be altered to append elements to the path, rather than prepend them.

```
YIELD (route, position) path'to aim:
   SHARE looking
   IF position in looking OR NOT safe position: RETURN failure
   IF position = aim: RETURN success
   INSERT position IN looking
   RETURN next'move
next'move:
   FOR boat'load IN possible'from position:
      PUT (new'route, new'position) path'to aim IN result, path'
      IF result <> "failure": RETURN result, path'
   RETURN failure
new'route: RETURN route with boat'load
new'position: RETURN position altered'for boat'load
success: RETURN "success", route
failure: RETURN "failure", {}
```

You would then have to call it as

```
PRINT ({}, start) path'to aim
```

However, you may notice that `route` here contains more or less the complementary information to what `looking` contains. If you represent a path as the sequence of positions, instead of the sequence of moves, you can do away with `looking` altogether, with the choice of `SHARE`ing `route` or passing it as a

parameter:

```
YIELD (route, position) path'to aim:
   IF position in route OR NOT safe position: RETURN failure
   PUT route with position IN route
   IF position = aim: RETURN success
   RETURN next'move
next'move:
   FOR boat'load IN possible'from position:
      PUT (route, new'position) path'to aim IN result, path'
      IF result <> "failure": RETURN result, path'
   RETURN failure
new'position: RETURN position altered'for boat'load
success: RETURN "success", route
failure: RETURN "failure", {}
```

This means that you get as output the states rather than the moves you have to make:

```
>>> PRINT ({}, start) path'to aim
{["cabbage"]: "N"; ["farmer"]: "N"; ["goat"]: "N"; ["wolf"]: "N"}
{["cabbage"]: "N"; ["farmer"]: "S"; ["goat"]: "S"; ["wolf"]: "N"}
{["cabbage"]: "N"; ["farmer"]: "N"; ["goat"]: "S"; ["wolf"]: "N"}
{["cabbage"]: "S"; ["farmer"]: "S"; ["goat"]: "S"; ["wolf"]: "N"}
{["cabbage"]: "S"; ["farmer"]: "N"; ["goat"]: "N"; ["wolf"]: "N"}
{["cabbage"]: "S"; ["farmer"]: "S"; ["goat"]: "N"; ["wolf"]: "S"}
{["cabbage"]: "S"; ["farmer"]: "N"; ["goat"]: "N"; ["wolf"]: "S"}
{["cabbage"]: "S"; ["farmer"]: "S"; ["goat"]: "S"; ["wolf"]: "S"}
```

However, for the rest of this article I will stick with the reverse method.

Another possible representation

You may remark in `path'to` that there is no dependency within the unit on exactly how a position is represented. What would we have to change if we represented it as, for instance, `{["N"]: {"farmer"; "goat"}; ["S"]: {"cabbage"; "wolf"}}`, a representation that more closely represents the real state of affairs?

Well, the main change would come in `altered'for`, which would look like this:

```
YIELD positions altered'for occupants:
   FOR occupant IN occupants:
      SELECT:
         SOME place IN keys positions HAS occupant in positions[place]:
            REMOVE occupant FROM positions[place]
            INSERT occupant IN positions[opposite place]
   RETURN positions

>>> PUT {["S"]: {}} IN banks
>>> PUT {"farmer"; "goat"; "cabbage"; "wolf"} IN banks["N"]
>>> WRITE banks altered'for {"goat"; "cabbage"}
{["N"]: {"farmer"; "wolf"}; ["S"]: {"cabbage"; "goat"}}
```

A corresponding change also has to be made to `possible'from`.
The only other thing that must be changed is the test `safe`, but it can be expressed even more simply now, since we can use a list of illegal positions:

```
>>> PUT {} IN illegal
>>> INSERT {"goat"; "cabbage"} IN illegal
>>> INSERT {"goat"; "wolf"} IN illegal
>>> INSERT {"goat"; "cabbage"; "wolf"} IN illegal
```

and then use this test:

```
TEST safe position:
   SHARE illegal
   REPORT NO place IN position HAS place in illegal
```

Multiple paths

An interesting change to consider is returning not just one solution, but *all* solutions to the problem. Clearly then, `path'to` must return not just one path, but all paths solving the problem, so we'll call it `paths'to`. In this case we don't need to return the 'success' or 'failure' indication: if it fails, it just returns the empty list: no solutions found. If the list is non-empty, then it succeeded. So `PRINT` will look like this*:

```
HOW'TO PRINT result:
   PUT result IN paths
   SELECT:
      paths = {}:
         WRITE "No solution" /
      ELSE:
         FOR i IN {1..#paths}:
            WRITE "Solution", i /
            FOR move IN i th'of paths:
               WRITE "Move", move /
```

The main part of `paths'to` is again hardly different from its predecessor. If the current position is where we wanted to go, then the result is the single empty path `{{}}`. If the routine fails, it returns no paths, i.e. `{}`. Otherwise, for each possible new position from the current position, it gets all paths from the new position to the aim. For each of these paths it adds the boat-load that created the new position to the front of the path. Obviously, if no paths are possible then it will not add the boat-load to any path.

```
YIELD position paths'to aim:
   SHARE looking
   IF position in looking OR NOT safe position: RETURN failure
   IF position = aim: RETURN success
   INSERT position IN looking
   RETURN all'paths
all'paths:
   PUT {} IN results
   FOR boat'load IN possible'from position:
      PUT new'position paths'to aim IN paths
      FOR path IN paths:
         INSERT path with boat'load IN results
   RETURN results
new'position: RETURN position altered'for boat'load
success: RETURN {{}}
failure: RETURN {}
```

And now to show it works:

```
>>> WRITE start
{["N"]: {"cabbage"; "farmer"; "goat"; "wolf"}; ["S"]: {}}
>>> WRITE aim
{["N"]: {}; ["S"]: {"cabbage"; "farmer"; "goat"; "wolf"}}
>>> PRINT start paths'to aim
```

*The assignment of `result` to `paths` is necessary because of the way parameters get passed to HOW'TOs. Without this assignment `paths'to` would get called three times. There is a proposal to change this in the revised version of *B*.

```
Solution 1
Move {"farmer"; "goat"}
Move {"farmer"}
Move {"cabbage"; "farmer"}
Move {"farmer"; "goat"}
Move {"farmer"; "wolf"}
Move {"farmer"}
Move {"farmer"; "goat"}
Solution 2
Move {"farmer"; "goat"}
Move {"farmer"}
Move {"farmer"; "wolf"}
Move {"farmer"; "goat"}
Move {"cabbage"; "farmer"}
Move {"farmer"}
Move {"farmer"; "goat"}
```

Solving other problems

Now that the program takes most of its information from data-structures, it's interesting to try and solve a different problem with the same program.

Consider this one: you have two jugs of different capacities, say j litres and k litres. You are allowed to empty either jug, fill either jug, or pour one into the other until the one is empty, or the other is full. The problem is, what series of actions are necessary to end up with the jugs containing m and n litres?

Right, let's call the two jugs A and B, and represent the state of the two jugs as a table of their contents. For instance, `{["A"]: 8; ["B"]: 5}` shows that A contains 8 litres and B 5. We can represent the set of allowable actions (the target `allowable`) as a list of actions, where each action is a compound giving the type of action, and the name of the jug. For instance `("Fill", "A")`.

```
>>> PUT {} IN allowable
>>> FOR action IN {"Fill"; "Empty"; "Pour"}:
        FOR jug IN "AB":
            INSERT (action, jug) IN allowable
```

(The action `("Pour", "A")` means pour A into B.)

Now, although strictly speaking, the actions possible from a given situation depend on whether the jugs are empty or full, and so on, it actually doesn't matter, because if you fill an already full jug, or empty an already empty one, you get the same situation, and the program ensures that situations don't get repeated, so we don't have to worry. Therefore, `possible'from` is very simple:

```
YIELD possible'from position:
   SHARE allowable
   RETURN allowable
```

There are also no illegal states:

```
>>> PUT {} IN illegal
```

Finally, we have to provide an `altered'for` for the new representations. To be able to carry out the actions like `Fill` we have to know the capacities of the jugs:

```
>>> PUT {} IN full
>>> PUT 8 IN full["A"]
>>> PUT 5 IN full["B"]
```

The only unobvious case is pouring: the amount that you pour is either the contents of the whole jug, or as

much as will fit, whichever is less:

```
YIELD amount altered'for actions:
   SHARE full
   FOR action, jug IN actions:
      SELECT:
         action = "Empty": PUT 0 IN amount[jug]
         action = "Fill": PUT full[jug] IN amount[jug]
         action = "Pour":
            PUT min {amount[jug]; full[other]-amount[other]} IN x
            PUT amount[other] + x IN amount[other]
            PUT amount[jug] - x IN amount[jug]
   RETURN amount
other:
   RETURN opposite jug
```

Obviously, `opposite "A"` gives `"B"`, and vice-versa. Now to try it (the output format of PRINT has also been changed):

```
>>> PUT {}, {} IN start, aim
>>> PUT 0, 0 IN start["A"], start["B"]
>>> PUT 4, 0 IN aim["A"], aim["B]
>>> PRINT start paths'to aim
Solution 1
Fill A, Pour A, Empty B, Pour A, Fill A, Pour A, Empty B, Pour A,
Empty B, Pour A, Fill A, Pour A, Fill A, Empty A, Pour B, Fill B,
Pour B, Empty A, Pour B, Fill B, Pour B, Fill B, Pour B, Empty A,
Pour B

Solution 26
Fill B, Pour B, Fill B, Pour B, Empty A, Pour B, Fill B, Pour B,
Fill B, Pour B, Empty A, Pour B
```

Exercise

Finally, as an exercise, consider what needs to be changed in order to solve the Towers of Hanoi problem. If you just use the restriction that a disk may not be placed on a smaller disk, it gives 12 solutions for moving just two disks from one pile to another! Here's the longest (the first line means "move piece 1 from rod a to rod c"):

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

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")
Move (1, "a", "c")
```

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

Conclusion

Experience with other programming languages can mislead one into thinking that because functions in *B* can have no side-effects certain practices are impossible. This article has attempted to show that in fact this is not so, largely because of the ease of returning values of any type in *B*.

Primality Testing in *B*

Evangelos Kranakis
Steven Pemberton

Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

Simple *B* programs for two types of primality tests are presented. The first type consists of *deterministic primality tests* and the second of *probabilistic primality tests.* These programs confirm one of the main strengths of *B*, i.e. the ability to write in simple and understandable fashion what would be much longer programs in many other currently used programming languages.

1. Introduction

Primality testing is the problem of determining if a given integer is prime (i.e. whose only divisors are 1 and the integer itself) or composite. In many respects this problem is similar to the factoring problem, i.e. given an integer n to find its nontrivial factors (if any). Since if you know the factorization of n it is easy to test if n is prime or composite, the former problem seems to be easier than the latter. Nevertheless, to this date no polynomial time deterministic algorithm is known for testing primality. The problem of primality testing is not merely of academic nature. Recent developments in public key cryptography indicate several intrinsic interconnections between the difficulty of decrypting encrypted messages in certain public key cryptosystems and the factoring problem. The programs given in this and the next sections are the *B* versions of well-known algorithms. The clarifications necessary for a theoretical understanding of the primality tests presented can be found in [3]. The reader can get acquainted with the basic aspects of *B* in [2].

The first program concerns modular exponentiation and will be very useful in the sequel; given positive integers x,n,m it outputs $x^n \bmod m$. The method used is widely known in the literature as *modular exponentiation by repeated squaring and multiplication.**

```
YIELD (x, n) powermod m:
    PUT 1 IN z
    WHILE n <> 0:
        WHILE n mod 2 = 0:
            PUT n/2, (x*x) mod m IN n, x
        PUT n-1, (z*x) mod m IN n, z
    RETURN z
```

For any integer $a \geqslant 2$ the next program returns e_{a-2}, where e_{a-2} is defined inductively as follows: $e_0 = 4$ and $e_{k+1} = e_k^2 - 2$.

```
YIELD sq a:
    PUT 4 IN e
    FOR i IN {1..a-2}: PUT e*e -2 IN e
    RETURN e
```

On input a the following program outputs a random b in the interval $[1,a-1]$ such that $\gcd(a,b) = 1$. The expected number of iterations of the WHILE loop is $\phi(a)$, where ϕ is Euler's totient function.

```
YIELD random a:
    PUT a IN b
    WHILE gcd (b, a) > 1:
        CHOOSE b FROM {1..a-1}
```

* One of the reasons for using it is that in some primality tests the numbers involved are so large that you would simply run out of computational capacity if you were to use usual exponentiation.
This program was written after a discussion with L. Meertens.

```
    RETURN b
```

The next sequence of programs culminate in a program that computes the *Legendre-Jacobi* symbol of two integers. For any given integer a the following program outputs the largest exponent e such that 2^e divides a.

```
YIELD ex a:
    PUT 0 IN e
    WHILE (a/(2**e)) mod 2 = 0:
        PUT e+1 IN e
    RETURN e
```

The following program on input a positive integer c returns its parity, i.e. 1 if c is even, and -1 otherwise.

```
YIELD parity c:
    SELECT:
        c mod 2 = 1: RETURN -1
        c mod 2 = 0: RETURN 1
```

On input an odd integer a the following program computes $(-1)^{(a^2-1)/8}$. The next one, on input odd integers a,b computes $(-1)^{(a-1)(b-1)/4}$.

```
YIELD j1 a:
    CHECK a mod 2 = 1
    RETURN parity ((a*a - 1)/8)

YIELD j2(a, b):
    CHECK b mod 2 = 1 AND a mod 2 = 1
    RETURN parity ((a-1)*(b-1)/4)
```

And finally, this last program computes the *Legendre-Jacobi* symbol of two integers a,b.

```
YIELD j(a, b):
    CHECK gcd (a, b) = 1
    PUT b/(2**(ex b)), 1 IN b, c
    WHILE a > 1 AND b > 1:
        PUT ex a IN e
        PUT a/(2**e) IN a
        PUT (j2(a, b))*((j1 b)**e)*c IN c
        PUT b mod a, a/(2**(ex a)) IN a, b
    RETURN c
```

2. Deterministic Tests

A primality test is called deterministic if the answer given by the test on input an arbitrary integer n is always correct. Although such a test would be very desirable no polynomial time (in the length of n) test is known. Thus, the fastest such test known today (the Rumeley-Adleman test) runs in time $O((\log n)^{c\log\log\log n})$. The difficulty of the problem is also indicated by the fact that nothing better is known on the set of binary representations of prime numbers than that it belongs to the class $NP \cap Co-NP$.

The oldest test known is the sieve of Eratosthenes. It is inefficient, but it is still the only way to get a list of all the primes $\leq$ a given integer.*

```
YIELD sieve n:
    PUT {2..n}, {} IN set, primes
    WHILE set > {}:
        PUT min set IN p
```

* A similar *B* program for the sieve appeared in [1].

```
        INSERT p IN primes
        WHILE SOME s IN set HAS s mod p = 0:
            REMOVE s FROM set
    RETURN primes
```

Thus, the *B* command

```
WRITE sieve 50
```

yields the set of primes less than or equal to 50

```
{2; 3; 5; 11; 13; 17; 19; 23; 29; 31; 37; 41; 43; 47}.
```

For special types of numbers primality testing can be done *efficiently*. The first such efficient test to be considered is known as the *Lucas-Lehmer* test and is used to test the primality of Mersenne numbers, i.e. numbers of the form $2^a - 1$, for $a > 2$.* In fact, $2^a - 1$ is prime if and only if $e_{a-2} \equiv 0 \bmod (2^a - 1)$.

```
HOW'TO LL a:
    CHECK a > 2
    SELECT:
        (sq a) mod ((2**a)-1) = 0:
            WRITE "2**`a` - 1 is prime" /
        ELSE:
            WRITE "2**`a` - 1 is composite" /
```

Proth's test is used to test numbers of the form $k\,2^a + 1$, where $k \leqslant 2^a + 1$, $a > 1$, and 3 does not divide k. It uses the fact that $k\,2^a + 1$ is prime if and only if $3^{k\,2^{a-1}} \equiv -1 \bmod (k\,2^a + 1)$.

```
HOW'TO PROTH ka:
    PUT ka IN k, a
    CHECK k <= 2**a + 1 AND a > 1 AND k mod 3 <> 0
    SELECT:
        (3, k*(2**(a-1))) powermod (k*(2**a) + 1) = k*(2**a):
            WRITE "`k`*(2**`a`) + 1 is prime" /
        ELSE:
            WRITE "`k`*(2**`a`) + 1 is composite" /
```

Pepin's test is a special case of Proth's test and is used to test the primality of Fermat numbers, i.e. numbers of the form $F_a = 2^{2^a} + 1$. Again, it uses the fact F_a is prime if and only if $3^{(F_a - 1)/2} \equiv -1 \bmod F_a$.

```
HOW'TO PEPIN a:
    SELECT:
        (3, 2**((2**a)-1)) powermod ((2**(2**a))+1) = 2**(2**a):
            WRITE "2**(2**`a`) + 1 is prime" /
        ELSE:
            WRITE "2**(2**`a`) + 1 is composite" /
```

3. Probabilistic Tests

In contrast to deterministic primality tests, probabilistic primality tests may not always give the right answer. In fact there is a positive real $\epsilon < 1$, independent of n, such that for all integers n if the answer of the test is *COMPOSITE* then n is indeed composite, but if the answer of the test is *PRIME* then n may not be prime; however, the probability of error is $\leqslant \epsilon$. The basic ingredient of a probabilistic primality test is the specification of an efficient test T. In the course of an execution of the algorithm it is required to draw a

* The 29th Mersenne prime is $2^{132049} - 1$. The 30th is not yet known, although $2^{216091} - 1$ is the largest known. It consists of 65050 digits and the testing was done on a Cray X-MP computer. The chances are that you will become the worst enemy of your system administrator if you use the test given here to check the primality of such big Mersenne numbers.

random integer satisfying certain conditions. Given such a test T the primality algorithm A_T corresponding to T is defined as follows.

Input: $n > 1$.
Step 1: Choose random $0 < b < n$ such that $\gcd(b,n) = 1$.
Step 2: Check if $T(b,n)$.
Output:

$$A_T(n) = \begin{cases} PRIME & \text{if } T(b,n) \text{ is true} \\ COMPOSITE & \text{if } T(b,n) \text{ is false.} \end{cases}$$

Naturally, one can significantly reduce the probability of error by making independent random choices of b in successive runs of the algorithm.

The first test to be described is the so called Rabin's test. The disjunction immediately following the SELECT command provides the test T for the algorithm A_T presented above.

```
HOW'TO RABIN a:
    CHECK a mod 2 = 1
    PUT random a IN b
    SELECT:
        (SOME i IN {1..ex (a-1)} HAS pmi = a-1) OR pmex = 1:
            WRITE "`a` is probably prime" /
        ELSE:
            WRITE "`a` is composite" /
pmi:
    RETURN (b, (a-1)/(2**i)) powermod a
pmex:
    RETURN (b, (a-1)/(2**(ex (a-1)))) powermod a
```

This second version of Rabin's test is equivalent to the previous one. However, the main improvement is that if the conjuction immediately following the SELECT command is true then the program returns the factorization of the integer tested. Thus, with a bit of luck one can factor the tested integer.

```
HOW'TO RABIN1 a:
    CHECK a mod 2 = 1
    PUT random a IN b
    SELECT:
        SOME i IN {0..ex (a-1)} HAS factor > 1:
            WRITE "`a`=`factor`*`a/factor`" /
        pm <> 1:
            WRITE "`a` is composite," /
            WRITE "   since `b`**`a-1` mod `a` = `pm` <> 1" /
        ELSE:
            WRITE "`a` is probably prime" /
factor:
    RETURN gcd (a, ((b, (a-1)/(2**i)) powermod a) - 1)
pm:
    RETURN (b, a-1) powermod a
```

Thus, the *B* command

```
FOR i IN {1..5}: RABIN1 2**(2**5) + 1
```

uses Rabin's test to check five times the primality of the 5th Fermat number*

* The compositeness of the 5th Fermat number was first proved by Euler in 1731, thus disproving Fermat's ill-fated conjecture (made almost a century earlier) that all the F_a's are prime.

$$F_5 = 2^{2^5}+1 = 4294967297.$$

There are five lines of *B* output.

```
4294967297=641*6700417
4294967297 is composite,
   since 4083994560**4294967296 mod 4294967297 = 2308631200 <> 1
4294967297 is composite,
   since 2224497408**4294967296 mod 4294967297 = 626184398 <> 1
4294967297 is composite,
   since 35176256**4294967296 mod 4294967297 = 2762233453 <> 1
4294967297 is composite,
   since 3441948800**4294967296 mod 4294967297 = 1324609083 <> 1.*
```

The last probabilistic primality test is the so called *Solovay-Strassen* test.

```
HOW'TO SS a:
    CHECK a mod 2 = 1
    PUT random a IN b
    PUT j (b, a), (b, (a-1)/2) powermod a IN jba, baa
    SELECT:
        jba = 1 AND baa = 1: WRITE "`a` is probably prime" /
        jba = 1 AND baa <> 1: WRITE "`a` is composite" /
        jba = -1 AND baa = a-1: WRITE "`a` is probably prime" /
        jba = -1 AND baa <> a-1: WRITE "`a` is composite" /
```

REFERENCES

[1] Geurts, L., *A Short Introduction to the B Language, The B Newsletter*, Centre for Mathematics and Computer Science, Amsterdam, Issue 1, August 1983.

[2] Geurts, L., Meertens, L. and Pemberton, S., *The B Programmer's Handbook*, Centre for Mathematics and Computer Science, Amsterdam, 1985.

[3] Kranakis, E., *Primality and Cryptography*, Wiley-Teubner, 1986.

* This is a probabilistic primality test, so you should not be surprised if the *B* output you obtain is not exactly the same as the one above.

HOW'TO ORDER *B* for UNIX:

To order the Mark 1 implementation of *B*, running on UNIX† systems, you should fill out the order form below, and two signed copies of the SOFTWARE AGREEMENT on the next page. Send it to:

a*B*c Group, Unix distribution
Informatics - AA
CWI
POB 4079
1009 AB Amsterdam
The Netherlands

You will then receive:

- a tape with the sources (including an installation guide)
- the following documentation:
  - Description of *B*
  - A Users Guide to the *B* System
  - *B* Quick Reference Card
  - Manual Pages

Also, one of the two copies of the SOFTWARE AGREEMENT will be returned to you signed.

† UNIX is a Trademark of AT&T Bell Laboratories

--

ORDER FORM

Please send us the Mark 1 implementation of *B* for UNIX systems for the price of Dfl 100 (US $ 35) (to cover materials, postage and bank charges) for which we will be invoiced.

Name: .....

Firm/Institute: .....

Address: .....

.....

Country: .....

Telephone: .....

Internet network address: .....

Machine type: ☐ Vax ☐ Sun ☐ PDP ☐ other:

Operating System: ☐ 4.2 BSD ☐ Version 7 ☐ System V ☐ other:

Check required tape parameters:

density	☐ 800 bpi	☐ 1600 bpi
format	☐ Tar, blocksize 1	☐ Ansi D format
	☐ Tar, blocksize 20	☐ Ansi F format

(For other media and formats, please inquire.)

We include **two** copies, both **signed**, of the SOFTWARE AGREEMENT.

Signature and Date:

Please, fill out **both** copies below, and **sign** them.

SOFTWARE AGREEMENT

Effective as of ..... 198.., Stichting Mathematisch Centrum (SMC), having an office at 413 Kruislaan, 1098 SJ Amsterdam, and

(LICENSEE), having an office at

agree as follows:

SMC grants fee-free to LICENSEE a personal, non-transferable and non-exclusive right to use the computer programs and documentation relating to the Mark 1 implementation of *B* (LICENSED SOFTWARE).

LICENSEE agrees not to sell, lease or otherwise transfer or dispose of the LICENSED SOFTWARE in whole or in part.

SMC makes no warranties, express or implied. SMC shall not be held to any liability with respect to any claim by LICENSEE or a third party on account of, or arising from, the use or the inability to use LICENSED SOFTWARE.

Neither this agreement nor any rights hereunder, in whole or in part, shall be assignable or otherwise transferable.

signed by

For LICENSEE	For Stichting Mathematisch Centrum
Name:	Name: .....
Title:	Title: Director of Software Licensing
Signature and Date:	Signature and Date:

--

SOFTWARE AGREEMENT

Effective as of ..... 198.., Stichting Mathematisch Centrum (SMC), having an office at 413 Kruislaan, 1098 SJ Amsterdam, and

(LICENSEE), having an office at

agree as follows:

SMC grants fee-free to LICENSEE a personal, non-transferable and non-exclusive right to use the computer programs and documentation relating to the Mark 1 implementation of *B* (LICENSED SOFTWARE).

LICENSEE agrees not to sell, lease or otherwise transfer or dispose of the LICENSED SOFTWARE in whole or in part.

SMC makes no warranties, express or implied. SMC shall not be held to any liability with respect to any claim by LICENSEE or a third party on account of, or arising from, the use or the inability to use LICENSED SOFTWARE.

Neither this agreement nor any rights hereunder, in whole or in part, shall be assignable or otherwise transferable.

signed by

For LICENSEE	For Stichting Mathematisch Centrum
Name:	Name: .....
Title:	Title: Director of Software Licensing
Signature and Date:	Signature and Date:

HOW TO ORDER *B* for the IBM PC:

To order the prototype of the implementation of *B* for the IBM PC or compatibles, running under MS-DOS versions 2.0 (or higher), you should fill in the order form below, and send it to:

a*B*c Group, PC distribution
Informatics - AA
CWI
POB 4079
1009 AB Amsterdam
The Netherlands

To cover materials and handling, you should either enclose a cheque or money order, payable to Stichting Mathematisch Centrum - Amsterdam, or (if you live in The Netherlands) transfer to the postgiro account below.

You will then receive:

- a floppy with the binary;
- The *B* Programmer's Handbook;
- a *B* Quick Reference Card.

ORDER FORM

Please send me a copy of the prototype *B* system for the IBM PC, including documentation.

☐ I enclose a cheque or international money order, payable to Stichting Mathematisch Centrum - Amsterdam, for Dfl 100 (or US $ 35).

☐ I have transferred Dfl 100 to postgiro account 462890, Stichting Mathematisch Centrum - Amsterdam, mentioning "*B* voor de IBM PC".

Name: .....

Firm/Institute: .....

Address: .....

.....

Country: .....

Telephone: .....

Network address: .....

Machine(s): ☐ IBM PC ☐ IBM XT ☐ Olivetti M24
☐ Apricot Portable ☐ Apricot F1 ☐ other: .....

Required media:

☐ 5¼" double sided, double density floppy disk
☐ 3½" double sided floppy disk

Required version:

☐ full implementation (at least 384 K bytes)
☐ small version (only 256 K)

Signature and Date: