Algorithmic Languages, de Bakker/van Vliet (eds.)
© IFIP, North-Holland Publishing Company, 1981, 167–184

Issues in the Design of a Beginners' Programming Language

Lambert Meertens

Mathematical Centre, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands

Some problems are related that have been encountered in the design of a programming language for beginners. The solutions were sometimes unexpected, and required doing away with preconceptions. The use of systematic methods has been of some help.

1. Introduction

Of the commonly available algorithmic languages, some are definitely better suited to convey the algorithmic thoughts of the programmer than others. Whatever the preferred point of view, be it structured programming, provability of correctness or the expressibility of abstraction, some languages stand out for their excellence, some for their abomination.

The latter should not worry us for languages in disuse. It should, for languages used widely. The relatively abominable FORTRAN, though far from dead, seems on its way out. Reasonable alternatives for FORTRAN exist. That absolute champion, BASIC, however, is steadily marching on. Moreover, BASIC has it attractive points, from the viewpoint of the casual, non-professional user.

An attempt is under way to redress that situation, by issuing a rival language, provisionally referred to as 'B' (no relation to the precursor of C; the 'B' is only a language-name name referring to the yet unknown language name). For a language to beat a rival, more is involved than language issues. The example of FORTRAN more than goes to show this point. This paper will be restricted, however, to linguistic points. It is not intended as an introduction to B, but as an exposition of some of the choices and problems encountered in the process of designing an algorithmic language. The attempt has been to base the solutions, in a rational way, on the design objectives.

B is designed as the limit of a sequence: B_0, B_1, ... The most recent approximation, B_2, is the joint effort of Robert Dewar of the Courant Institute of Mathematical Sciences, New York University, Leo Geurts of the Mathematical Centre, and the author. Contributions have been made by Peter King of the University of Manitoba, Jack Schwartz of the Courant Institute, and Dick Grune and Paul Klint of the Mathematical Centre. The responsibility for the opinions expressed is solely that of the author.

2. The Design Objectives for B

The idea underlying the design objectives for B are: beat the enemy at its strong points. The same idea has governed the design of ELAN [5]. There is one important difference: ELAN aims primarily at the 'market' of (introductory) education in computer science, whereas B aims first of all at personal computing. The latter has not always been the case. The first approximation of B (see [3]) was designed when personal computing was in its infancy. Although the design objectives themselves have remained the same, their impact on the design has changed quite drastically.

The design objectives for B are:
– simplicity;
– suitability for conversational use;
– inclusion of structured-programming tools.

These objectives are elaborated upon in [3]. The change referred to above is mostly concerned with the objective of simplicity. In [3], this is interpreted as simplicity not only for the user, but also for the implementer. It is stated that "B should be implementable on small minicomputers".

The latter reflects our awareness, at the time, of the onset and future importance of personal computing. At the same time, it reveals a lack of perception of the torrent of hardware evolution. Tomorrow's hand-held computers are yesterday's main-frames. Designing a language to run smoothly on eight bit 8K machines is designing for the past. In designing B_2, it was decided to ignore implementation issues completely. Not that we do not care about implementation complexity; for the time being we have merely disregarded the feelings of prospective implementers and concentrated on the happiness of the user. Once sufficient implementation experience is available, it may be decided to revise features that pose undue implementation problems in exchange for little or no gain in language appeal. The impact of ALGOL 68R on the revision of ALGOL 68 reveals that this may even help to improve the language from the user's point of view.

3. The Types of B_2

In B_0 and B_1, the types were **INT, REAL, STRING** and '**RANGE**' types (similar to the scalar types of PASCAL), and **ARRAY**s of scalar elements indexed by a compound of **RANGE** values (but without the PASCAL restriction of compile-time fixed bounds). The type system had not really been given much thought, and was the first thing tackled again in the design of B_2.

The type system of B_2 has been designed in a new way that is, in itself, of interest. If a sufficiently powerful collection of types is available (where 'type' includes type constructors as 'array'), any desired type (e.g., deque, or ternary tree) can be 'simulated' or implemented by the user. The type could also be added as a 'standard' type to the language. This may increase the ease of use of the language. Not all types, however, are equally helpful in this respect. Moreover, the language is made more complex, and possibly much so. A type system is competitive only if it is better than each other type system in at least one respect (ease of use, simplicity).

So we compiled a list of candidate types (including, e.g., bag, deque, enumerated types, map, multi-valued map, queue, sequence, set, stack and tree), constructed various schemes for implementing these types in terms of other types, and assigned numerical values for (relative) algorithmic importance and learning complexity of each type and for implementation complexity of each scheme. The values took into account, of course, that the user we have in mind is not a computer scientist. This made it possible, with the assistance of a program, to weed out the non-competitive type systems from the rather large powerset of the candidate types. The resulting list of competitive systems was quite small, and it was easy, using old-fashioned human taste, to settle on one for use in B_2.

If B_1 might be called ALGOL 60 in BASIC-like disguise (the abstract of [3] reads: "FORTRAN : ALGOL 60 = PL/I : ALGOL 68 = BASIC : ?"), B_2 came out like SETL [1] in sheep's clothing. The result is that the types of B_2 are 'number', 'text', 'compound', 'list' and 'table'.

Numbers come in two kinds, 'exact' (i.e., rational) and 'approximate' (i.e., floating point). The distinction is made at run time. This choice attempts to combine the following desiderata:

(a) The user must be allowed control over quantities that should not be subject to rounding errors. (The choice for rational numbers, rather than integers, is mainly a nicety. But there is some obvious advantage in having, e.g., **1.25**, represent an exact number.)

(b) The user should have no need to worry about the distinction if it is not important. (E.g., adding exact and approximate numbers is allowed.)

(c) The language has strong typing.

(d) Coercions, i.e., automatic implicit type conversions, are deemed undesirable.

(e) Approximateness propagates upwards in evaluating arithmetic expressions.

(This list is not really exhaustive. It implies, among others, the presupposition that there should be some built-in treatment of approximate numbers.)

The approach taken satisfies these five desiderata almost perfectly. Almost ...; in conformance with Murphy's Eighth Law, there is one ugly snag. If **x** is approximate, **x/x** does not equal **1**. For approximateness propagates, and the approximate number **x/x** cannot be equal to the exact number **1**. It is, presumably, equal to the approximate number **~1**.

In fact, no proper solution satisfying the desiderata (a) through (e) exists. As soon as one of these is lifted, a full solution becomes possible. The fact that **1** does not equal **~1** is a violation of (b): sometimes the user does have to worry. We choose this solution because we felt that the user should be careful anyway when comparing approximate numbers and has no business to expect exact answers. Moreover, it is still possible to define the comparison **1 = ~1** to succeed, even though the values are not 'identical'. The solution of allowing one coercion, from exact to approximate numbers (and coercions in its wake on composite values), is still under consideration.

Texts are quite ordinary strings. (The term 'text', instead of the esoteric 'string', was taken from [5].) No character values are provided; a text of length one will do. Two subtext operators are available. If the value of **t** is the sequence of characters $c_1, \ldots, c_n$, then the expression **t@p**, with $1 \leq \mathbf{p} \leq n+1$, stands for $c_{\mathbf{p}}, \ldots, c_n$ and the value of **t|q**, with $0 \leq \mathbf{q} \leq n$, is $c_1, \ldots, c_{\mathbf{q}}$. A common combination will be **t@p|q**. If **t|q^t@(q+1)** is defined ('^' is concatenation), its value is **t**.

These subtext operators may also be applied to text variables in target ('l.h.s.') positions. The replacing text need not have the same length as the text replaced.

Compounds (tuples) are like structured values ('records'), but without tags for selecting the fields. If, e.g., **u** and **v** are variables, then **u,v** may be used in a target position. This allows decomposition of compounds.

Lists exist for values of any type (e.g., list of list of text). A list is simply a multi-set, or bag. In an algorithmic context, given the choice between sets and multi-sets, the latter are more useful. Having both is unnecessarily complex, and even a potential source of confusion. Since we do not expect the user to be familiar with the concept of a multi-set, the semantics are explained in terms of ordered lists. A consequence is that a total ordering has to be defined on the values of any given type. This can be done in a reasonably natural way.

Tables are like SETL maps: generalized arrays whose domain is variable and not necessarily a range of consecutive values. In contrast to SETL, tables are a genuine type, not a syntactic sugaring for interpreting a set of pairs as a map. In particular, a table cannot be a 'multi-valued' map.

Originally, there were many restrictions in this type system. For example, the elements of a compound, list or table could only be numbers or texts. Table keys (indices) were numbers, texts or compounds. Especially the compounds had a special status. Although we thought we had good reasons for these restrictions (at the time the decision to ignore the ease of implementation had not been fully mentally digested), one by one better reasons appeared to relax these constraints. At first, the relaxations tended to make the complexity worse, until we took the step that, in hindsight, seems so obvious: the type system was made completely orthogonal: tables may be indexed with tables, and so on. (This decision nevertheless required reworking most of the provisional language definition.)

As the type system stands now, we are quite pleased by it. The types appear in some way to span together the space of needs, as was the purpose of the exercise. A carefully tamed 'free' type was at some time included, but abandoned later on.

4. Command Syntax

Commands (statements) in B are rather wordy. Each command begins with a keyword, and keywords are also used to separate the parameters of a command. For example, the following is an assignment command:

```
PUT a+1 IN a.
```

The philosophy behind this approach is given in [3]. An obvious drawback of verbose syntax is that the user has to key in so many symbols. However, as is already stated in [3], the language is embedded in a system that is dedicated to B. In particular, the editor knows the syntax of B. If this is combined with screen-editing facilities, it is possible to reduce the number of key strokes drastically. As soon as the editor knows (or maybe guesses) that a **PUT** command is intended, it may already display the **IN** and position the cursor at the first parameter.

In [4] it is remarked that the keyword approach makes it possible to have user-defined commands. This option has indeed been chosen for B_2. Such command definitions take the role of procedures. For example, the user may define

```
HOW'TO INCR x: PUT x+1 IN x
```

and next use this **INCR** command as though it had been part of the language all of the time.

Since programs are entered through a B-dedicated editor, it is realistic to consider program lay-out as an integral part of the syntax. In particular, indentation is used to indicate grouping of commands. Although this was already so in [3], it took us quite some time to disengage ourselves completely from the idea that programs are prepared on one system and parsed by a second one that need not trust its input. The fact that there is no distinction between editor and parser means that no special delimiters like **BEGIN** and **END** are needed. That **BEGIN** was superfluous, we had already realized; but this was true anyway. But for quite some time, we required **END** lines, as in

```
FOR p IN feasible:
    IF p in cand:
        REMOVE p FROM cand
        INSERT p IN chosen
    END IF
END FOR
RETURN chosen, cand.
```

But the lines with **END** are pure noise. Once one gets used to it, the following is much more legible:

```
FOR p IN feasible:
    IF p in cand:
        REMOVE p FROM cand
        INSERT p IN chosen
RETURN chosen, cand.
```

5. Strong Typing without Declarations

It has been clear from the beginning that B should have strong typing. Not for efficiency reasons, but to aid the user in spotting silly errors as soon as possible. It seemed to us that this calls for declarations revealing the type of identifiers. (The FORTRAN 57 solution of restricting the choice of identifiers for a given type is unacceptable, as is the addition of special symbols as in BASIC.)

One of the attractive features of BASIC is the lack of declarations. Therefore, without really believing in it, we have searched for a system that allows strong typing without declarations. (The advantage of declarations that they provide a redundancy protecting against typos can be taken over by checks against the use of uninitialized variables and warnings for assignment to dead variables.) In some languages with strong typing, it is essential that the type of identifiers is revealed through a declaration. For ALGOL 68, e.g., the value yielded by

```
(amode block = ("abc", "def");
 2 upb block
)
```

is 3 if **amode** is [,] **char**, but 1 if **amode** is [, ,] **char**. But this is clearly a peculiarity. In almost all cases one can reconstruct the types from the context in which identifiers are used.

This has led us to finding a system for B_2 in which it is always possible to reconstruct the type of identifiers from the context. This statement should be slightly weakened in two respects.

The first is that it may be possible to assign types to the identifiers consistently in more than one way. This happens, for example, in

```
PUT {} IN x
IF x = {}: WRITE 'yes'.
```

Here **x** could be an empty list of numbers, but it could equally well be an empty list of texts or anything else, or, in fact, an empty table (assuming **x** is not used otherwise). In such cases the net effect is always the same for each type assignment, so we do not care. It also happens in

```
PUT a IN a,
```

if no other assignments to **a** are made. But then **a** is not initialized, which is illegal by itself (and is checked statically).

The second is that commands defined with **HOW'TO** may be truly generic. The definition

```
HOW'TO SWAP a AND b: PUT b, a IN a, b
```

will work for any type, as long as the two parameters have the same type. So no type can be assigned to **a** and **b**. Instead, the requirement is that if **HOW'TO**s are expanded as macros to an arbitrary depth, consistent type assignment remains possible. This raises some hard questions, and undecidability is lurking around the corner [2, 6]. Nevertheless, for B_2 this appears to be decidable without undue restrictions. Only after the last sentence was written down, did the author become aware of the work on type polymorphism by Milner [10]. Although this is described for an applicative language, it appears equally applicable for a language as B. In fact, the situation is simpler there, since the items carrying a polymorphic type are not treated as values in B.

There is one point where an unconventional step had to be taken to uphold the system. If a value comes into being through an operation on other values, it is sufficient if the result type is only dependent on the operand types, which is the case in B_2. We may thus concentrate on the spots where values appear directly. This can happen in two ways.

One is through a constant denotation (literal). This is no problem, since constants in B_2 immediately reveal their types, with one exception: for empty lists or tables. This case has been treated above.

The other case is when a value is obtained through interactive input. There is no a priori way to determine the type. Therefore, it is required that the **READ** command reveal the type of the (expected) input. A first attempt required the presence of a 'type specifier', where the size of the syntax for specifiers turned out not unsubstantial. This was not very satisfying; it meant the user had to learn a lot of (relatively weird) syntax for this one purpose. Luckily, we found another solution, made possible by the fact that for each value an explicit notation can be given. The type is now specified by providing a 'sample': an expression of the same type. So one has to write, e.g.,

```
READ n, v EG 0, {''}
```

if **n** is a number variable and **v** is a list of texts. (The constant {} will not do in this case.)

6. Formulas

Just like 'procedure calls' and 'commands' are unified in B_2, so are 'function calls' and 'formulas'. A new operator or function is introduced by a **YIELD** unit:

```
YIELD fac n:
     PUT 1 IN f
     FOR i IN {1..n}: PUT f*i IN f
     RETURN f.
```

The compound mechanism gives a natural way to introduce more parameters:

```
YIELD abs (x, y): RETURN sqrt(x*x+y*y).
```

The parentheses are only required since the formal parameter is an explicit compound; the definition might also have run:

```
YIELD abs z:
     PUT z IN x, y
     RETURN sqrt(x*x+y*y).
```

These two definitions are functionally completely equivalent.

For some reason or other, the priorities of operators are a trouble spot in algorithmic languages. An extreme solution as in APL is not attractive; the more so since B_2 is not really expression-oriented. Anyway, it is unacceptable if **2*n+1** really means **2*(n+1)** (although it certainly helps in making the users feel they belong to an esoteric cult). The MABEL solution of requiring parentheses as soon as several operators are involved [7], combines the virtues of simplicity and error resistance. Still, it seems a bit harsh to require parenthesizing of **2*m*n**.

The solution that has been adopted for B_2 is to require parenthesizing whenever the priorities are not established by standing convention and might matter. This is achieved by not assigning simple priorities to operators, but a priority *interval* instead. This interval represents a 'fuzzy' priority. If the precedence decision is independent of the choice of priorities from the intervals, the expression is acceptable. Otherwise, parentheses must be inserted. User-defined operators are always assigned the maximal interval.

Acceptable expressions are, e.g., **m*n/d+c+1**, **a−b+1** and **2*sqrt x**. Unacceptable are **a/2*b**, **a/2/b** and **sqrt 2*x**, to give just a few examples. Of course, the editor warns the user on the spot that parentheses must resolve the ambiguity.

It was a bit surprising that such a simple device as priority intervals could be tuned to give such reasonable results.

7. Generators

Lists are only useful if there is some easy way to step through them. Originally, there were two ways for stepping through a list, one (**OVER alist**) in the normal, and one (**REVO alist**) in reversed order (word play intended). The second form followed an idea from [9], and was connected to the scalar type requirement for table domains in B_0. Once this requirement is relaxed, the convenience of the additional form no longer justifies the extra complexity.

The keyword **OVER** was changed to **IN** for B_2. For example, the command

```
FOR i IN a: INSERT i IN b
```

merges list **a** into **b**. This was done after it had already been decided to allow quantified tests: the test

```
SOME i IN a HAS i<0
```

succeeds if **a** contains a negative element (and sets **i** to stand for the value of the first such element, if any). Instead of **SOME**, also **EACH** and **NO** are allowed.

In B_0, the domain of a table had to be defined as a **RANGE** type in order to create the table. With a dynamic domain, this no longer applies. But there should be some way for the user to go through a table domain. As a first attempt, a domain operator was introduced: **keys t** (during some time written **[]t**) gives the list of keys **i** such that **t[i]** is defined. So we could write:

FOR i IN keys t: . . . .

Switching to a seemingly unrelated topic, we wanted some simple but powerful mechanism for text parsing. A first attempt was a '**FITS** test' of the form

e FITS $\mathbf{v}_1$: $\mathbf{t}_1, \ldots, \mathbf{v}_n$: $\mathbf{t}_n$,

with **e** a text expression, $\mathbf{v}_i$ variables and $\mathbf{t}_i$ tests. (The keyword **FITS** keeps appearing and disappearing in the design of B, each time with a different meaning.) The whole test succeeds if an assignment of texts to $\mathbf{v}_1, \ldots, \mathbf{v}_n$ is possible, such that $\mathbf{e} = \mathbf{v}_1\hat{}\ldots\hat{}\mathbf{v}_n$ and all of the tests $\mathbf{t}_i$ succeed. If several successful assignments were possible, the lexicographically first one would be returned.

Now this would have filled an appreciable part of the syntax for one specialized capability. Moreover, it was unlike anything else in the language. Then we realized that we almost had the capability already there, right under our hands. For the semantics were exactly those of

SOME $\mathbf{v}_1, \ldots, \mathbf{v}_n$ **IN ??? HAS** $(\mathbf{t}_1$ **AND** $\cdots$ **AND** $\mathbf{t}_n)$,

provided some suitable expression for the ??? could be substituted. This expression should be a list of all compounds $\mathbf{s}_1, \ldots, \mathbf{s}_n$ such that $\mathbf{e} = \mathbf{s}_1\hat{}\cdots\hat{}\mathbf{s}_n$. A provisional notation for this list was **e/n** (**e** divided in **n** parts). This raises the problem that the type of **e/n** is dynamically dependent on **n**, which is incompatible with strong typing. If the form were only allowed in this context, the problem would disappear; in fact, the **n** is then redundant, since there are exactly **n** bound variables.

This triggered the solution adopted now. It is illustrated by the following example:

```
WHILE SOME h, s, t PARSING sent HAS s = ',':
        INSERT h IN words
        PUT t IN sent.
```

If **sent** contains a comma, the parsing will be found that positions **s** at the first comma (so **h** will not contain a comma). If **sent** does not contain a comma, the test fails. If **sent** originally held the text **'hickory,dickory,dock'**, the effect is that of

```
INSERT 'hickory' IN words
INSERT 'dickory' IN words
PUT 'dock' IN sent.
```

This is the most complicated feature in B_2; it is, however, quite powerful. Its semantics can be explained in already familiar terms. At the same time, it takes away the nagging problem that a simple command as

```
PUT 'memory is becoming cheap'/24 IN m
```

threatens to blow up even gigabyte systems.

When **OVER** and **REVO** were originally introduced, and when they were replaced by **IN**, we did not think of the construction as a generator. With **PARSING**, we clearly have a generator. It is quite natural then to have a generator **INDEXING** to go through all keys of a table. For example,

```
PUT 0 IN s
FOR i INDEXING t: PUT s+t[i] IN s
```

sums the elements of **t**.

Such a decision may seem simple. But it has many ramifications. One is that the function **keys** should be abolished. Inspection of programs shows that in practice it is never used in a command like

```
PUT keys t IN kt.
```

But the function is used in other ways, such as

```
PUT min keys t IN mt,
```

which finds the smallest key in the domain of **t**. The meaningful test

```
i in keys t
```

would also have to be replaced by some new notation. Instead, it was decide to leave **keys** alone, not to introduce **INDEXING**, but to generalize **FOR ... IN ...** to iterate also over the characters of a text and the elements of a table. Summing the elements of a table may thus be written:

```
PUT 0 IN s
FOR e IN t: PUT s+e IN t.
```

The same generalization applies, of course, to **SOME . . . IN . . .**, but also to all functions and tests previously only defined on lists (such as **min** and **in**).

8. The final composition

As has been clear from the exposition, composing a language is not merely a matter of putting ingredients together and stirring till the result is a smooth paste. It would be helpful to language designers, if some top-down design method existed for algorithmic languages. If such a method exists, it has escaped our attention. The requirement for applying a method as 'separation of concerns' is that the relevant concerns be separable. The whole experience of language design points in a different direction: apparently innocent minor decisions may quite unexpectedly work major havoc in seemingly unrelated corners. A well-composed language is one in which the 'features', although orthogonal, lend themselves to easy combination in many natural modes of expressing algorithmic thought. This means that the whole language is a tightly knit fabric, threatened by loose ends.

The best aid to systematic language design, until now, is the paradigm of orthogonality, that derives its name from the title of Van Wijngaarden's [14], but whose essence can already be found in his [13]. Experience shows that its application requires skill, if not expertise. It is interesting to see that the evolution of B has been in the direction of more orthogonality, mainly by virtue of the quest for simplicity.

For part of the work in designing B_2, a new systematic approach has been used: the method described in Section 3 to select the type system. This method is more widely applicable; it can be used, e.g., to find a proper system of string operations from a large set of candidates. Work is in progress to apply another systematic method for the final polishing of the whole language.

The idea has been used before by the author in a composition exercise of a different nature: composing a string quartet with traditional harmony [8]. The same idea is applicable here. In its bare essence, it boils down to considering all combinations of all alternatives for the microscopic design decisions. For each combination, a check list is inspected of potential unacceptable or undesirable consequences. For each transgression, a fine is imposed. The combination that collects the minimal total fine, comes out as the winner.

The method is, of course, NP-complete. In practice, however, it is expected to be feasible with the aid of some heuristics, since many design decisions form relatively independent small clusters. Still, this computational complexity is indicative of how hard it is to design a language. The example of the five reasonable desiderata for the numbers, only four of which could be satisfied simultaneously, is just one example of the problems a language designer may run across.

It would be misleading to call such methods 'language design by computer'. The real skill goes into identifying the decisions, weighing the importance and merits of various approaches, and identifying harmful combinations. Only a dumb, but hard, part of the work is left to brute force. It is expected that the first-time 'winner' will mainly serve to show deficiencies in the input to the program, and that several iterations will be needed to come up with a nice product. Indeed, the exercise may point out directions we have overlooked. If anything, the method requires that human prejudice is made explicit. The algorithm itself is, like Justice, blind-folded.

References

[1] R.B.K. Dewar, The SETL programming language, Courant Institute of Mathematical Sciences, New York University (1980).

[2] N. Gehani, Generic procedures: an implementation and an undecidability result, Comput. Languages 5 (1980) 155–161.

[3] L.J.M. Geurts and L.G.L.T. Meertens, Designing a beginners' programming language, in: S.A. Schuman (Ed.), New Directions in Programming Languages 1975 (IRIA, Roquencourt, 1976) pp. 1–18.

[4] L.J.M. Geurts and L.G.L.T. Meertens, Keyword grammars, in: J. André and J.-P. Banâtre (Eds.), Implementation and Design of Algorithmic Languages (IRIA, Rocquencourt, 1978) pp. 1–12.

[5] G. Hommel, J. Jäckel, S. Jähnichen, K. Kleine, W. Koch and K. Koster, ELAN – Sprachbeschreibung (Akademische Verlagsgesellschaft, Wiesbaden, 1979).

[6] H. Langmaack, On correct procedure parameter transmission in higher programming languages, Acta Inform. 2 (1973) 110–142.

[7] P.R. King, MABEL manual, University of Manitoba (1978).
[8] L.G.L.T. Meertens, The imitation of musical styles by a computer, in: Information Processing 68, Proc. of IFIP Congress 1968, Vol. 1 (North-Holland Publ. Co., Amsterdam, 1968) pp. xxv–xxvi.
[9] L.G.L.T. Mode and meaning, in: S.A. Schuman (Ed.), New Directions in Programming Languages 1975 (IRIA, Roquencourt, 1976) pp. 125–138.
[10] L.G.L.T. Meertens, Preliminary draft proposal for the B programming language, Mathematical Centre, Amsterdam (May 1981).
[11] R. Milner, A theory of type polymorphism in programming, J. Comput. System Sci. 17 (1978) 348–375.
[12] K. Tracton, 57 Practical Programs and Games in Basic (Tab Books, Blue Ridge Summit, 1978).
[13] A. van Wijngaarden, Generalized ALGOL, in: Symbolic Languages in Data Processing, Proc. of an ICC Symp. (Gordon and Breach, New York, 1962) pp. 409–419; also in: R. Goodman (Ed.), Annual Review in Automatic Programming, Vol. 3 (Pergamon Press, Oxford, 1963) pp. 17–26.
[14] A. van Wijngaarden, Orthogonal design and description of a formal language, Report MR 76, Mathematical Centre, Amsterdam (1965).

Appendix A: a B_0 and a B_2 Program for the Sieve of Eratosthenes

The following B_0 program is copied from [3].

```
BEGIN
CONST n IS 1999
RANGE sievesize FROM 2 TO n
RANGE primality HAS prime, nonprime
ARRAY (sievesize) a TYPE primality
FOR i OVER sievesize PUT prime IN a(i)
VAR k TYPE int, kmult TYPE sievesize
PUT 2 IN k
WHILE k*k FITS kmult
  BEGIN
  VAR k1 TYPE sievesize
  IF k FITS k1, a(k1)=prime DO sieve
  PUT k+1 IN k
  END
sieve:
  BEGIN
  PUT nonprime IN a(kmult)
```

```
        WHILE kmult + k FITS kmult PUT nonprime IN a(kmult)
        END
    FOR i OVER sievesize
    IF a(i) = prime
        BEGIN
        NEWLINE
        PRINT i
        END
    END
```

This problem was certainly not selected in [3] to show the cluminess of B_0. The algorithmic thought is captured more easily, though, in B_2:

```
HOW'TO SIEVE n:
   PUT {2..n}, 2 IN primes, k
   WHILE k*k<=n:
      PUT k*k IN kmult
      WHILE kmult<=n:
         IF kmult in primes: REMOVE kmult FROM primes
         PUT kmult+k IN kmult
      PUT k min primes IN k
   WRITE primes
SIEVE 1999
```

Note that this program is algorithmically slightly different from the B_0 program given above. The formula **k min primes** yields the smallest element of the list **primes** exceeding **k**.

Appendix B: a BASIC and a B_2 Program for Tabulating a Recurrent Sequence

The following program is copied from [12]. It has been selected because for this problem none of the 'strong' points of B_2, such as manipulation of lists, apply. For purposes of fair comparison, non-keywords have been rendered in lower case.

```
10 REM This program computes a table of Fibonacci
   numbers
20 PRINT 'Enter first term'
```

```
 30 INPUT a
 40 PRINT 'Enter second term'
 50 INPUT b
 60 PRINT 'Maximum number of terms ='
 70 INPUT n
 80 PRINT
 90 PRINT 'Table of Fibonacci numbers'
100 PRINT 'Term no.','Fibonacci number'
110 LET k=1
120 PRINT k,a
130 LET k=2
140 PRINT k,b
150 LET k=k+1
160 LET q=a+b
170 PRINT k,q
180 LET a=b
190 LET b=q
200 IF k>= n THEN 220
210 GOTO 150
220 PRINT 'Maximum numbers of terms reached'
230 PRINT
240 PRINT 'Type 1 to continue, 0 to stop'
250 INPUT l
260 IF l=1 THEN 280
270 STOP
280 PRINT
290 GOTO 20
300 END
```

The following B2 program is not an exact transliteration; it contains an obvious improvement that might also be applied to the BASIC version. As to the question if this is fair in making a comparison, it should be considered that part of the thesis motivating the development of B is that BASIC invites clumsy programming.

```
HOW'TO TABULATE'FIBONACCI'NUMBERS:
  PUT 'yes' IN cont
  WHILE cont|1='y':
    WRITE / 'Enter first term: '
```

```
READ a EG 0
WRITE / 'Enter second term: '
READ b EG 0
WRITE / 'Maximum number of terms='
READ n EG 0
WRITE // 'Table of Fibonacci numbers'
WRITE / 'Term no.  Fibonacci number'
FOR k IN {1..n}:
  WRITE / k>>5, a>>15
  PUT k+1, b, a+b IN k, a, b
WRITE / 'Maximum number of terms reached'
WRITE / 'Do you want another table? '
READ cont EG ''
```

This program shows some 'formatting': the formula **x>>n** yields a text of length **n** representing the value of **x**, right adjusted (left-padded with blanks).