ma
the
ma
tisch

cen
trum

AFDELING INFORMATICA IW 46/75 DECEMBER

L.J.M. GEURTS & L.G.L.T. MEERTENS

DESIGNING A BEGINNERS' PROGRAMMING LANGUAGE

Prepublication

amsterdam 1976

stichting
mathematisch
centrum

MC

AFDELING INFORMATICA IW 46/75 DECEMBER

L.J.M. GEURTS & L.G.L.T. MEERTENS

DESIGNING A BEGINNERS' PROGRAMMING LANGUAGE

Prepublication

2e boerhaavestraat 49 amsterdam

Printed at the Mathematical Centre, 49, 2e Boerhaavestraat, Amsterdam.

AMS(MOS) subject classification scheme (1970): 68A30

ACM-Computing Reviews-categories: 4.22

Designing a beginners' programming language *)

by

L.J.M. Geurts & L.G.L.T. Meertens

ABSTRACT

FORTRAN : ALGOL 60 = PL/I : ALGOL 68 = BASIC : ?

KEY WORDS & PHRASES: *programming language, structured programming, programming language design, simplicity*

*) This paper is not for review; it is meant for publication elsewhere

0. INTRODUCTION

Among programming languages there is a family of languages which are characterized by their syntactic and semantic simplicity and their suitability for conversational use. Perhaps the most familiar of these is BASIC, some others being FOCAL, JOSS and TELCOMP. The typical user of such languages is not a professional programmer, nor does he dwell in an academic computer science environment. He does not have the time nor the ambition to learn a complicated language for the occasional program he writes.

Now, these languages were mostly designed before the current ideas on "structured" programming became generally accepted. They lack most of the tools that a programming language can provide for taking a structured approach to programming, presumably because about the same effects could be obtained with "simpler" means. Unfortunate as this is by itself, the situation is particularly bad since, for example, BASIC - maybe the worst villain in this respect - is quite commonly used to teach high school students introductory courses in computer science. There exists little material on the effect on a programmer of the first programming language he is exposed to, but from personal experience we have the strong impression that in many cases it deeply influences his thinking habits for a long time to come.

It is out of concern with this situation that we have looked at the problem of designing a language which would have an appropriate arsenal of structured-programming tools (thus eliminating their harmful equivalents) and yet be very simple. To facilitate discussion, we have called the hypothetical language fulfilling these criteria "B". Thusfar, the results are incomplete and in some respects unsatisfactory. If we nevertheless present our attempts here, it is in the hope that others be stimulated to examine our approach critically, to suggest simplifications or other improvements, or to contribute otherwise to the solution of the problems that have to be overcome.

It is a legitimate question if it is necessary to develop a new language and if not one of the existing languages could take the place of the question mark in the abstract. We did review many of these, but, without naming any specific language, we found none meeting all of our three criteria, as given in section 2. If one of these criteria is disregarded, however, it is easy enough to find a satisfactory language, but in each case we hoped to do significantly better with respect to the third criterion. In fact, since none of the existing programming languages appears to have been designed with this particular combination of objectives, this situation is not surprising. We also looked into the possibility of taking a subset of an existing programming language. Apart from the fact that the definition of such a subset constitutes in essence the design of a new language, this did not work out either. The problem, in general, was that either the expressive power of the subset was definitely insufficient, or the semantics were too complicated. It is interesting to note that the languages that appeared to give the best results were quite wide-spread and successful ones: ALGOL 60 and PASCAL.

1. DESIGNING A PROGRAMMING LANGUAGE

In designing a programming language - or, for that matter, any major system - one faces the task of bridging the gap between the design objectives and the basic material from which the system is to be built. It has been suggested that a good approach is the top-down one, starting from the objectives, and not commit-ting oneself to design decisions until further postponement becomes impossible. Now, this approach is feasible only if it is possible to separate the system into a number of subsystems which are relatively independent, i.e., design decisions in one subsystem interact hardly with design decisions in the other ones. If such a separation is possible at all in language design, we have been unable to find it. (An exception may be separation of the choice of basic data types, data structuring primitives and the corresponding operations. We have not yet given these issues much thought.)

Instead, we tried to clarify to ourselves our design objectives, took some general, but rather committing, decisions on matters of principle, sometimes by just cutting the knot, and next proceeded to fill in the open spots straightforwardly, just to see what a programming language along these lines might look like. Especially in this latter part decisions are often quite arbitrary and completely open to revision; we felt it wiser to go to the bottom and then re-iterate the design process, than to remain pondering imponderable design decisions.

2. CLARIFICATION OF DESIGN OBJECTIVES

Our design objectives are
- simplicity
- suitability for conversational use
- inclusion of structured-programming tools.

2.1. *Simplicity*

Simplicity in a programming language has two aspects which may, but need not, be at a par: simplicity for the user, and simplicity for the implementer.

For the kind of user we have in mind, simplicity would mean:
1. he has only a small number of constructions to learn;
2. the concrete syntax of the constructions is suggestive of their meanings and therefore easy to remember;
3. the semantics for each construction is as straightforward as can be;
4. it must be possible to postpone learning more complicated concepts, if any, until the simpler ones are understood.

It should be understood that we are aiming at a far simpler language than FORTRAN, ALGOL 60 or PASCAL, that is, simpler in the above respects. This implies, of course, that a proficient programmer may feel hampered by the poorness of the language. But B is not intended to serve his purposes, and he would be well advised to use other languages. The typical program we envisage is small (100 lines, say) and relatively straightforward.

If anything, B should be no exploring ground for new concepts in programming. Elegance is of no concern, simple-mindedness is.

It follows from our aims that the whole enterprise will be a failure if B should not gain widespread use. Obviously, the quality of B alone, supposing the design efforts turn out successful, is no warrant for acceptance. A necessary condition is the availability of implementations, which can be furthered by

simplicity for the implementer. (Of course there are many more conditions, but these are not intrinsic to the language.) Simplicity implies here:
- a straightforward parsing scheme which (hopefully) needs only one pass;
- no need for optimization;
- simple memory management at run time;
- few run-time routines.

Typically, B should be implementable on small minicomputers.

Unless we head for an "interpretive" language (see 3.2), the one-pass goal suggests declaration before use, except maybe in those cases where the meaning of identifiers is clear from the context. Simple memory management would be served by having all programs satisfy the "most recent" property (if at some time more than one instance of a recursive procedure is active, the static chain of lexicographically enclosing blocks contains the most recently activated one). One way to achieve this is to forbid routines as parameters. Garbage collection is another source of difficulties; if a heap is necessary, it would be nice, for example, to deal memory in chunks of uniform size, to which only one pointer may refer and from which only one pointer emerges. These issues require further study.

We also strive for uniformity of implementations, meaning that a program which runs on one implementation will also run on another one with the same result (but for limitations of time or memory size). As far as B is concerned, this means we are restricted to a small set of generally available characters. It would also mean, for example, that real arithmetic would have to be specified down to the last bit. If properly done, this may be helpful to implementers on minicomputers.

A special consequence of this uniformity is that the semantics should be deterministic. This is also desirable for another reason. If non-determinism of the semantics would be reflected in (pseudo-)non-determinism of the implementations, this would be very unhelpful to the programmer who tries to find why his program fails. If, on the other hand, the implementations are deterministic where the language is not, experience shows that eventually, in the mind of the programmer, the semantics of the programming language is supplanted by the semantics of the implementation.

2.2. *Suitability for conversational use*

We require that B is suited for conversational use. The term "conversational" (or "interactive") has no clearly delineated meaning. In some cases the error messages of a compiler are already considered conversational if directed to a terminal; on other occasions the term is reserved for natural language oriented systems which display a sophisticated form of intelligence.

We choose to use the term for a system with the following aspects:
- it follows the "utterances" of the user closely, and reacts immediately when-

ever appropriate, rather than keeping its reaction till the final moment of analysis when the user is done;

- it displays one "face" to the user, rather than a variety of faces of subsystems on different levels, such as an editor, a file system, a compiler, each with its own conventions and reactions and hardly aware of each other's existence;
- it does not leave the user uncertain whose turn it is and prompts him whenever a reaction is required.

It must be possible to integrate B into such a system, and in fact we hope ultimately to define the complete B-system, rather than just the B-language. The B-editor should already perform the parsing and detect most syntactical errors. This means that the syntax of B must be such that the effect of syntactical errors is as local as possible, and it suggests that the language should be line oriented; that is, programs are considered as sequences of lines, not as mere sequences of symbols. If the editor knows the syntax, this also gives perspectives for simplifying editing commands.

2.3. *Inclusion of structured-programming tools*

The most important design goal is the inclusion of a set of structured-programming tools. Abstractly, we consider any language feature that aids in proving program correctness a structured-programming tool. (There are many other ways of looking at structured programming, all of which eventually seem to converge to the same set of tools.)

There are several ways in which a structured-programming tool can facilitate a correctness proof. We can at least discern the following three aspects.

a. The various constructions should have a clearly understandable meaning, i.e., a meaning which is easily expressible in terms of assertions, can be grasped intuitively, and does not require retention of the original definition. To illustrate this point, the while and the if-then-else construction are conceptually simple, but the meaning of the for-statement of ALGOL 60 can only be fully understood using the definition in the Revised Report on ALGOL 60. For example, the effect of the statement

```
for i:= 0 while false, i step 1 until 0, i + 1 step i - 1 until 10, i do
begin print (i); i:= i + 1 end
```

is not easily determined as printing

```
0   3   7  15.
```

b. A proof can usually be divided into relatively independent smaller proofs.

The language should permit this division to be reflected in the program texts. (Stepwise refinement is the corresponding method of program construction.) The usual tools are block structure and procedures. However, most programming languages hardly encourage a top-down approach. One either has to substitute the refinement literally in the program text, with the effect that the original structure is hidden, or must use procedures, thus incurring a great loss of efficiency, and moreover, the definition of the procedure must often be inserted preceding the application.

c. The level of abstraction at which it is possible to understand an algorithm and, therefore, the level at which it is convenient to prove the correctness, is invariably much higher than the level supported by the programming language, even for simple programs. Here, even more strongly than in the previous aspect, one wishes to be able to clearly separate the different levels of abstraction in the program text. One way to do this is to build several layers, each providing the primitives for the next one.

We hasten to say that we do not expect that the user of B will write down a correctness proof for his programs; the point is that a program that is easily proved correct is easily understood, and the hypothetical correctness proof is reflected in the construction of the program.

The distinction between these three aspects is not always clear-cut, and we used it only to guide our thinking. As to the first aspect, a schematic description of our design method may be given as follows. Take a feature F which is under consideration as a candidate for inclusion in B. (At this stage F is an imprecise concept.) Now try to find the algorithmic concepts which might be implemented using F, and examine if these specific concepts themselves merit being translated into new features to be included in B. This may very well lead to the conclusion that inclusion of F itself is undesirable. (Just as the goto-statement may be abolished in favour of a while and an if-then-else construction.) Otherwise, the semantics of F are chosen such that they do not give rise to surprises if used to model these notions. An example of such a surprise, if the passing of variables as parameters is implemented with the ALGOL-60 call-by-name semantics, is illustrated by

```
procedure swap (p, q); integer p, q;
begin integer h; h:= p; p:= q; q:= h end,
```

after which *swap (i, a[i])* and *swap (a[i], i)* have different meanings. Complications are the facts that the number of features to be included should remain limited, and that two conceptually different notions may be semantically so close as to be confusing.

3. GENERAL DECISIONS

3.1. *Abstract syntax*

One of the issues where a decision appeared necessary at an early stage is that of the general syntactic approach, which has implications both for control structures and for data structures. As to the abstract syntax, i.e. the basic method of program composition, it was not hard to decide that an expression language, where (in general) the elaboration of any construction yields a result, was out of the question: in such languages, e.g., ALGOL 68, constructions which have an effect, e.g., assignations or closed clauses, have side effects if used as expressions. If expressions do have side effects, there are no good program points to assign assertions to; in fact, the meaning of a program with such expressions may be difficult to grasp unless the side effects belong to a different level of abstraction. If expressions with side effect should be allowed at all, they should be the exception rather than the rule, and be confined to a clearly separate corner, such as function procedures.

The choice therefore fell on the more conventional approach of program composition as a sequence of statements, executed in turn.

3.2. *Concrete syntax*

For the concrete syntax, the possibilities seem almost unlimited. Yet, the following line of reasoning gave us some hold.

Let the term "constructor" stand loosely for those symbols or combinations of symbols which form, so to say, the skeleton of a construction. In ALGOL 68 we find, among others, constructors *+*, *:=*, *@*, *¢¢* and *if then else fi*. These symbols are, in general, either rather special characters or word delimiters. Since, for B, we are confined to a small character set, only a few special characters are available, fewer than the number of constructors we need. It is, of course, possible to combine several characters into one symbol (*:=*, or even *:/=:*), but this is mnemonically bad, since such symbols are hardly suggestive of the associated meaning. Another possibility is overloading of constructors, but that seems even worse - just consider the parsing problems in ALGOL 68, for automata and human beings alike, arising from the overloading of *:* and *(*.

This leads us straight to the use of word delimiters. Now, we think it a bad idea, if only for competitive reasons, to require a form of stropping. On the other hand, it appears unwise to have reserved words, not so much because we want to encourage choosing such words as identifiers, but because of the havoc such an accidental choice may work in the parsing of the program. (Also, this approach would not allow the beginner to have only a partial knowledge of the set of reserved words.) The only way out is to have such a syntactic structure that it is always clear to the parser whether a word has to be interpreted as an identifier or as a "keyword", as in PL/I. A simple way to achieve this would be to alternate keywords and identifiers as in FOR I FROM M TO N DO S OD. This would imply that a construction beginning with a keyword cannot take the place of such an identifier. Therefore we make the following distinction. Let the term "construend" refer to the constructions "held together" by the keywords of the constructors (the same way as operands are held together by operators). We then have for construends "statements", which always start with keywords, and "expressions", which never do. We take care that the first keyword of the constructor of any construction is unique, i.e., distinct from the remaining keywords of that constructor and from all keywords of other constructions. This precludes having both IF THEN and IF THEN ELSE, or both FOR WHILE DO OD and WHILE DO OD, or PR PR. Under these conditions it is possible to construct with great liberty almost any combination of keywords, expressions and statements, as long as the keywords of the constructor up to a potential construend tell us whether to expect next a statement or an expression, or nothing at all.

3.2. *Compilation vs. interpretation*

Another quite general issue is the choice between orientation towards compilation vs. interpretation. (These terms are not the most felicitous, since they refer to properties of implementations, not of languages, but we hope their meaning is nevertheless clear.) The orientation towards interpretation is in many aspects very attractive. In general, the syntax and semantics may be greatly simplified by having a run-time determined type. The implementation effort may then in general be greatly reduced. Another perspective is the integration of program statements and system commands. This is done in some of the interactive languages; for example, PRINT (I × I) FOR I = 1, 10 may be used as a statement, but also as a command, so that the system contains in fact a glorified desk calculator. The idea is very appealing; after ample consideration, however, we have decided not to pursue it, since we fear it might encourage an attitude towards programming that we would rather discourage: programming should be done in the mind or on paper, not at a terminal. Moreover, we did not quite see how the interpretive orientation could be reconciled with (a) the locality of scope of identifiers required for factoring correctness proofs, or (b) easy use of assertion oriented proofs, since the assertions would have to be extended with clauses like "if this variable has the proper type, if that variable has the proper type,...". For these reasons, we closed our ears to the chant of the sirens and decided on compilation orientation.

4. DEFINITION OF B_0

Presented below is the "definition" of a language B_0, an order 0 approximation of B, interspersed with some justification for particular choices made. As mentioned before, the choices made are often quite arbitrary; in general, if no justification is mentioned for a choice which may be thought questionable, the reason is probably that no such justification exists. No attempt is made for any formality or rigidity, as this would be completely pointless at this stage. Also, no attempt has been made for clarity or completeness of description; we heavily rely on the reader's knowledge of programming language concepts and his intuitive understanding.

B_0 is neither more nor less than the result of the very first term of an iterative design process, such as is normally not disclosed for the world to behold.

We have freely incorporated any feature found in existing languages where this seemed desirable. In such cases, in general, no reference or credit is given.

4.1. *Layout*

Typographical display features, such as space or new line, play a role in the syntax of B_0. They are required to separate lexical units which otherwise might be taken for one unit, e.g. a keyword and a tag. They are not allowed within lexical units. At any given position a transition to a new line may be forbidden, optional or obligatory, depending on the particular construction in which it occurs. Each statement may start at a new line, and, moreover, unless the statement is the last part of another statement, the new line is always obligatory, and so constitutes a sequencing operator, just like the semi-colon in ALGOL. Some constructions have other obligatory line transitions. The syntax of B_0 is such that a new line where forbidden or no new line (but a space) where obligatory, never changes a valid program into another valid program. As a consequence, a B_0 editor that is aware of the syntax and automatically indents at each new line, may also automatically increase the indentation level at each new line which is not obligatory, thus indicating continuation of the running statement. Similarly, at the end of each statement the editor can restore the old indentation level. As a result, B_0 programs always have a reasonable layout.

4.2. *Control structures*

The traditional if-then-else construction seems to perform two conceptually different functions: to prescribe an action in a specific, typically rare, event, as in

$$\underline{\mathit{if}}\ x > max\ \underline{\mathit{then}}\ max := x\ \underline{\mathit{fi}}$$

or to select an action according to the appropriate case, as in

```
if   x < 0 then sign:= -1
elif x > 0 then sign:= +1
else sign:= 0 fi.
```

It is only by coincidence if there are exactly two cases from which to select in the latter case. Therefore B_0 has two different constructions:

```
IF condition statement
```

and

```
CASE condition₁ statement₁
  .
  .
  .
CASE conditionₙ statementₙ
ELSE statement.
```

(Note that the keyword ELSE is, technically speaking, superfluous.) Of course, the conditions are (a special case of) expressions. They are tested sequentially; each alternative starts on a new line. The above examples would become, assuming some format for assignations:

```
IF X > MAX PUT X IN MAX
```

and

```
CASE X < 0 PUT -1 IN SIGN
CASE X > 0 PUT +1 IN SIGN
ELSE PUT 0 IN SIGN.
```

This form of the case-statement was suggested by the structured presentation of the semantics in the Revised ALGOL 68 Report.

We have some doubt whether the presence of the if statement beside the case statement is really desirable: the semantics are so close that the uninitiated programmer may get confused and use the former construct where the latter is more appropriate. The reverse situation is not so bad; it it simply means that the programmer has to consider the action to be taken if the (exceptional) condition is not met.

For the case-statement, we would have preferred semantics where the ordering of cases is immaterial. The only possible way we see, is to test all conditions and to require that at most one succeeds. This means however, introduction of run-time errors for cases which, abstractly viewed, are perfectly valid, such as

```
CASE problem-can-be-solved-by-method-a APPLY method-a
CASE problem-can-be-solved-by-method-b APPLY method-b
ELSE reach-for-higher-means,
```

where either method might apply to a particular problem.

In cases were no action is required in the ELSE part, the PASS statement is used:

```
CASE A > 0 PUT P + 1 IN P
CASE A < 0 PUT P + 1 IN P
ELSE PASS.
```

For repetition the obvious choice seems to be:

```
WHILE condition statement.
```

Although this is our choice indeed, we have also given attention to possibilities as

WHILE condition$_1$ statement$_1$
.
.
.
WHILE condition$_n$ statement$_n$
DONE,

but the advantage of having this multi-conditional form available does not seem to outweigh the disadvantage of the extra DONE in the much more frequent uni-conditional case. Should it be decided, however, to discard the if-statement, then this form becomes the most attractive one.

No provisions are given for escape from a while statement. All "solutions" known to us are rather ad-hoc and violate the principle that upon completion the condition is known to fail. Also, no repeat-until construction is provided, since it is an open invitation for the common beginner's programming error of overlooking the possibility that a loop may be "empty", as in

```
REPEAT PUT A // 10 IN A
UNTIL A < 10,
```

which yields the first digit of A, unless A happened to be a one-digit number.

A condition may take the form

simple-condition$_1$, ..., simple-condition$_n$

with the meaning that the simple-conditions are evaluated from left to right, until one of them fails (in which case the whole condition fails) or all are found to succeed. (Note that the order of evaluation is immaterial if the expressions have no side effects, unless the evaluation may yield an error.)
Example:

```
        1 <= I, I <= N.
```

No connectives are provided in B_0 for disjunction or negation. It may turn out in practice that this is untenable, especially for the while-statement as it stands now.

Another form of repetition is given by

```
        FOR id OVER range-id statement
```

or

```
        FOR id REVO range-id statement.
```

As in ALGOL 68, the identifier is bound to the statement and cannot be assigned to. Range-identifiers correspond to the type of index values and are used in array-declarations. The keyword REVO reverses the order so that the range is traversed from upper to lower bound.
Example:

```
        FOR I OVER ROW
            FOR J OVER COL
                PUT 0 IN A(I, J)
```

For grouping a sequence of statements into one, we have the block

```
        BEGIN
        statement_1
            .
            .
            .
        statement_n
        END.
```

We are not too pleased with the keywords BEGIN and END: they have an imperative connotation rather than a parenthetical one.

Declarations may be interspersed between the statements. As a rule, declaration must precede application.

4.3. *Procedures*

Conceptually, we can divide the use of procedures into

- refinement;
- process specification where the iterative structures are insufficeint or cum - bersome;
- new functions or operations.

For the first type of use, parameters and recursion are not needed (and even unwanted). In this case access to non-local entities is standard. The other two types, which are not clearly distinct, need some kind of parameters. In these cases, we consider access (other than through parameters) of non-local entities which belong to the realm where the procedure is applied, undesirable and unnecessary.

For refinement, one can use a statement

```
DO ref-id
```

and then define the refinement by

```
ref-id: statement.
```

The effect is as though the statement were textually substituted for the piece of text DO ref-id. In order to avoid confusion it is required that all identifiers used in the statement are "visible" from the position where the refinement is defined.

B_0 does not have other types of procedures. The reason that this obvious structured-programming tool is not included, is simply that we have not (yet) found a satisfactory approach to the parameter mechanism. The call-by-name mechanism of ALGOL 60 and the call-by-value mechanism of ALGOL 68 are both quite simple, but each has aspects making it unattractive for B:

a. call by name:
- may not be what is needed in the program;
- in certain cases a rather intricate systematic change of identifiers is needed;
- there exists a discrepancy between the abstract replacement one imagines while programming, and the concrete replacement by textual substitution (cf.2.3.a);
- implementation problems.

b. call by value:
- requires a generalized notion of "value" (for example, procedures as values);
- problems in specifying the type of the parameters;
- either addresses are values, or unacceptable inefficiencies are incurred if, for example, arrays are transmitted.

The conclusion seems to be that B_1 will have about the parameter mechanism of PASCAL.

The philosophy of the keywords opens the possibility of user-defined statements, as in

```
          DEF INCR X
              PUT X + 1 IN X
              ENDDEF,
```

after which a new INCR statement is defined. This also must await a choice for the parameter mechanism.

4.4. *Data structures*

The predefined basic types are INT, REAL and STRING. The usual arithmetic operations +, -, *, / and ** are available, where for exponentiation the exponent must be an unsigned INT constant. If the operands are of mixed type, automatic widening from INT to REAL takes place. Division always yields a REAL value. Integral division is written with the operator // . Here, in contrast to ALGOL 60/68, (-7) // 3 = -3, so that (A + B) // B = A // B + 1 always holds. The priorities of the operators are the conventional ones (and -2**2 = -4). For comparison, we have <, <=, =, <>, >= and >. Special functions available are SQRT, LN, EXP, SIN, COS, ATAN, SIGN, ABS and ENTIER, just as in ALGOL 60 and with the same type conventions. However, the result of ABS has the same type as its argument, and ATAN takes two arguments; in the sense of numerical analysis, if

```
          PHI = ATAN (X, Y)    and     R = SQRT (X**2 + Y**2),
```

then

```
          X = R * COS (PHI)    and    Y = R * SIN (PHI).
```

For strings, the basic operations are + (concatenation) and the comparison operators. The function HEAD yields a string consisting of the first character of its argument; the function TAIL yields a string consisting of its argument minus its first character; if the argument of HEAD or TAIL is the empty string, then so is the result.

New basic types may be created by a range-definition:

RANGE range-id FROM int-expression$_1$ TO int-expression$_2$

or

RANGE range-id HAS tag$_1$, ... , tag$_n$.

The obligation to PASCAL should be obvious. A type BOOL could be defined as

```
          RANGE BOOL HAS TRUE, FALSE.
```

That this type is not predefined is not without reason. In most cases, clarity is served by an explicit indication of what the alternatives in a two-valued type stand for, as in

```
RANGE PASSAGE HAS OPEN, CLOSED
```

so that one may ask

```
IF AHEAD = OPEN ....
```

The functions LWB and UPB yield the lower and upper bound of a range (not of an array!).

Expressions of any range type may be used as arithmetic expressions, with automatic conversion to INT. The other way around is possible by a special simple-condition

```
int-expression FITS range-type-variable
```

which succeeds only if the value of the int-expression is within the range associated with the range-type-variable, whereupon that value is assigned to the variable. Thus, a loop such as

```
FOR I OVER H PUT 0 IN A(I)
```

is short for

```
BEGIN
VAR AUX TYPE INT
PUT LWB (H) IN AUX
VAR I TYPE H
WHILE AUX FITS I
    BEGIN
    PUT 0 IN A(I)
    PUT AUX + 1 IN AUX
    END
END.
```

For any of the basic types, simple variables may be declared in a declaration of the form

VAR id_1 TYPE $type\text{-}id_1$, ... , id_n TYPE $type\text{-}id_n$,

where the type-id is either INT, REAL or STRING, or a range-id.

The sequence

id_1 TYPE type-id, id_2 TYPE type-id

may be shortened to

```
id₁, id₂ TYPE type-id.
```

Constants may be declared by

CONST id_1 IS $expression_1$, ... , id_n IS $expression_n$.

There is no need to indicate a type here, as automatic conversion will take care whenever necessary.

Arrays of variables are declared by

ARRAY ($range\text{-}id_1$, ... , $range\text{-}id_d$) id_1 TYPE $type\text{-}id_1$, ... , id_n TYPE $type\text{-}id_n$,

with the same abbreviation as for simple variables.
Subscripting is only possible with subscripts of corresponding range-type. This implies that subscripts are either an identifier (constant or simple variable) or a subscripted variable. Practice only can teach us whether this restriction will be acceptable.

Assignment is only possible to variables; there are no such things as array expressions. The general form of an assignation is

PUT $expression_1$, ... , $expression_n$ IN $variable_1$, ... , $variable_n$,

where the types of the expressions must conform to those of the variables. The expressions are evaluated before the assignment takes place, so

```
PUT B, A IN A, B
```

will swap the contents of the variables A and B. This form of assignation has been chosen for didactic reasons, to emphasize the algorithmic notion of variables, instead of, e.g.,

```
SET X TO 1
```

or, even worse,

```
LET X BE 1
```

which is suggestive of algebraic rather than algorithmic variables. We have chosen not to incorporate structured variables. The reason for this is that structured variables make sense mainly to define abstract data types. The problem, therefore, is to find first a clear and simple way for introducing abstract data types with associated operations (which leads also to the problem of the parameter mechanism).

We have not found a satisfactory solution to the problem of uninitialized variables. Roughly, we can distinguish four approaches (apart from "who cares"):

1. Check at run-time. Disadvantage: yet another run-time error.
2. Default initialization. Disadvantage: if the intended initialization (to another than the default value) is accidentally omitted, this may pass by unnoticed; worse than alternative 1.
2. Initialization as part of the declaration. Disadvantage: duplication of the semantics of assignment; moreover, there are problems for arrays (unless all elements are initialized to one same value).
4. Static check whether all possible computation paths initialize a variable before it is used, with a suitable definition of "possible path". Disadvantage: the check is not very simple, and the correctness conditions may be unclear to the simple-minded user.

At the moment, we tend to favour the last approach, provided that it turns out not too complicated.

4.5. *Transput*

At the moment we envisage three transput statements:

PRINT expression$_1$, ... , expression$_n$,

NEWLINE

and

READ variable$_1$, ... , variable$_n$.

An alternative to the read statement would be to have an expression READ. This would be, however, an unnecessary introduction of an expression with side effects. It is intended that in

```
RANGE ANSWER HAS YES, NO
VAR GOON TYPE ANSWER
PRINT "DO YOU WISH TO CONTINUE?"
READ GOON
```

YES (or NO) would be valid input.
PRINT should output in a simple, standard format, the idea being that a programmer who wishes a special effect should take the trouble of constructing the necessary strings himself. An open problem is how to detect on input the end of a string.

5. EXAMPLE OF A B_0 PROGRAM

```
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
```