AB43.4.1 MABEL: A Beginner's Programming Language.

P.R. King, G. Cormack, G. Dueck, R. Jung, G. Kusner, J. Melnyk

(Department of Computer Science, University of Manitoba,
Winnipeg, Manitoba R3T 2N2, Canada)

ABSTRACT

This paper presents a preliminary version of an introductory programming language. The design of MABEL is far from frozen, and many of the decisions taken are, at best, tentative. Our hope in presenting the language at this stage is to obtain input from a wider source. Hence we earnestly solicit constructive cricitism, and ask readers to accept the current document in this spirit.

1. INTRODUCTION

MABEL (MAnitoba BEginner's Language) is a programming language for people who have never programmed before. It is a simple, general-purpose language. Hopefully, this does not imply that with MABEL one can only do simple things. Rather, MABEL is intended to provide a simple introduction to the art of programming by assisting the newcomer in the design of sequential algorithms. MABEL is designed to be simple to teach and to use.

The designers received suggestions from a variety of sources, both within the University of Manitoba, from students and instructors alike, and from a number of high school instructors within the Winnipeg School System who were asked to identify areas of difficulty encountered both by themselves and by their students. Each member of the group had a "pet" language (PASCAL, COBOL, ALGOL W, ALGOL 68, PL/1 and SNOBOL), features of which were advanced by its proponent and avidly attacked by others. We also listened (though frequently pretending otherwise) to the comments of colleagues outside the group as features on which we sought opinion were surreptitiously leaked.

Existing beginning languages, such as B_0 and the Toronto SP/k system, were given careful attention.

From this diversity of advice, sometimes helpful but often impossible or derisory, the criteria of §2 were established. This list became our bible, sacrosanct and inviolable, by virtue of which all design decisions were taken and to which all disputes were referred.

The majority of the time spent in actual design was spent in taking three basic decisions, namely the primitive types, the data structuring facilities and the parameter mechanism. These decisions and their rationale will be discussed in §3. Once they had been taken, most of the remainder of the design followed relatively rapidly and easily. There was some hectic infighting over the form of the repetitive construct, but the bloodshed was minimal compared to that occasioned by discussions over the primitive types, for example. The remainder of MABEL, after the three basic decisions, will be discussed in §4.

Personal prejudice began to rear its ugly head when deciding upon the concrete syntax, and the current proposals may suffer in that regard as a result of the occasional compromise decision. Some sample programs appear as an appendix, and a MABEL syntax chart, à la Watt, Sintzoff and Peck, is appended.

2. DESIGN CRITERIA FOR MABEL

The objective of MABEL is to provide as smooth an introduction as possible to the esoteric art of programming. Whether or not the beginner will graduate from his lowly state, and what happens when he does, is not deemed of any great relevance in determining how to effect such an introduction. If MABEL is a good introduction to more complex languages we would regard this as a bonus rather than a result of design.

Although nine criteria are explicitly discussed, several points not given in the list were considered, but ultimately excluded from the design. These included whether MABEL should provide an introduction to machine architecture, whether MABEL should be extensible and whether MABEL should define such (non-elementary but potentially simple) features as program modules and linkage to other languages. Such features are now being considered in the context of a systems implementation language being designed as a MABEL superset.

We first consider five "positive" criteria: those which were of major importance in the design of MABEL.

(i) Simplicity. The beginner must not be confused by a large number of unorthogonal features. There must be no discrepancies between the meaning of constructs when they appear in different situations or in the ways in which they may be used. Thus, the distinction between <statement> and <simple statement> in ALGOL W is not simple; the ALGOL 68 iterative statement is far from simple both because one requires so much information before it can be used even for simple applications, and because two applications, such as

```
FOR i TO n DO  read(a[i]) OD
```

and

```
WHILE REAL x;  read(x); x>0 DO SKIP OD
```

have vastly different forms and purposes; the use of pointers and associated dereferencing and aliasing is very far from simple.

(ii) Readability. The reader should find MABEL programs relatively self-documenting and self-verifying. These requirements impinge on design at both the abstract and concrete levels.

MABEL must be surprise free, conform wherever possible to accepted mathematical meaning (5/3 is the same as 5.0/3.0) and adhere to the precepts of structured programming.

(iii) Teachability. No feature was added to MABEL until one had demonstrated a simple means of teaching it to beginners. The language should be teachable in a "continuous" fashion, by incorporating features which can be exemplified and assimilated in small "upwards-compatible" stages, rather than features which require a lot of detailed information before they can be put to simple use.

(iv) Introduction to design of algorithms. The beginning programmer is habitually faced with two problems; firstly, to design a sequential algorithm for the problem at hand, which is rarely in sequential form, and secondly, to cast this algorithm into the form required by the particular programming language being used. MABEL has been designed to assist the user in the first of these: all else is of subsidiary importance, and one will observe that MABEL lacks certain "standard" language features since they do not contribute directly to this end.

(v) Versatility. Many students have a low opinion of their introductory language as a direct result of disappointment in the applicative examples with which the "power" of the language

was illustrated. One wonders to what extent such samples are chosen simply because the language in question is just so restricted. MABEL is a general-purpose language and, as the examples in the appendix show, has the power to be used for "real" problems. We hope that MABEL will cater, to some extent at least, to the ex-beginner who nonetheless wishes to continue using MABEL because he likes it.

Two further criteria were deemed of somewhat less importance:

(vi) Small compiler. It is quite probable that a common environment for a language like MABEL will be mini-computers. Thus the MABEL compiler must be of modest size, and this should be reflected in the language design.

(vii) Simple compilation. It is highly desirable that MABEL be 1-pass. Equally, it must be easy to associate clear, meaningful diagnostics with both compile-time and run-time errors. Our experience is that these latter questions are as much matters of language design as of compiler design.

Two final criteria were considered to be of rather minimal importance to the design of MABEL:

(viii) Introduction to programming languages. *"And visit the sins of the fathers upon the children unto the third and fourth generation."* This theology appears rampant in programming language design (and is, we claim, responsible for a multitude of disastrous design decisions). It is

not the philosophy of MABEL: we do not accept that transition from a simple to a more complex language is facilitiated by incorporating bad features from the complex language in the simple one.

(ix) Run time efficiency. Although the run time efficiency of both time and space requirements are of minor importance in the design of a beginner's language, they should not be entirely ignored if the language is to gain any degree of acceptance as a viable product.

3. THREE FUNDAMENTAL DESIGN DECISIONS

i) Simple types in MABEL

MABEL has a single simple type. The programmer may define and manipulate constants and variables of this simple type, and compose structured types from it. In this respect, MABEL resembles SNOBOL 4, and uses the same syntax for literals, representing them as character sequences enclosed within ", " or ',', pairs.* MABEL is not a string processing language. Naturally, the programmer will be aware that certain program variables are restricted to certain subdomains of this one type and that certain operators make sense only in certain subdomains, but MABEL considers such subdomains to be entirely the programmer's responsibility rather than a static, feature. (A clever compiler, however, might handle some of them statically.)

In retrospect one wonders why this decision took us so long to take; it now appears entirely natural and obvious. The reasons for

---

* Currently, the quotes are optional for integer constants. This is under review.

having multiple pre-defined types, in ALGOL for example, appear to be

- increased static security
- increased readability, and self-documentability
- increased run-time efficiency
- ability to use generic operators

The first of these is to a large extent an implementation concern and low on our score-card. The second is highly debatable. One can as easily read and comprehend

```
P + B * Q - 3      or
A AND C OR Q<3
```

without consulting the declarations or knowing the types of A, B, C, P or Q as with; any complete understanding requires detailed dictionary type descriptions in either case. Few beginner programs ever run in production mode; thus the third reason hardly applies. Finally, generic operators are especially confusing to the beginner; why should

```
"PQRS" < "XYZ"      or      3* "XYZ"
```

be meaningful? If one means "comes alphabetically before" or "replicate three times", then one should say so.

Further, multiple types add to the complexity of a language per se, by virtue of the diverse denotations required and, most of all, by virtue of the type conversions, both explicit and implicit. The beginner needs no assistance in accepting that

```
'3' + '4.7'
```

is perfectly sensible and yields '7.7', whereas

```
'3' + 'XYZ'
```

is not sensible and will produce rubbish.

It is to be admitted that typing permits certain errors to be caught earlier than can be done in a typeless language, but it is not clear that the class of such errors is sufficiently broad to

justify the complexity of multiple types. On the other hand the adoption of a single type added considerably to the ease of description of transput (c.f. §4 iv) and assisted greatly in defining the data structuring facility of MABEL, the second fundamental design decision.

(ii) Data Structures in MABEL

In order to satisfy the criterion of versatility, MABEL should provide the powers afforded by conventional data structures, including pointers, heap-storage management and flexible arrays. Conventional arrays as in FORTRAN and ALGOL would be quite unorthogonal with the single primitive type of MABEL. Restricting indexation to integers is inappropriate since integer is not a predefined type, and since strings have no inherent order, the concept of an array as an ordered set is equally unsuitable. It was decided to replace the conventional array by a facility such as the table concept of SNOBOL. SNOBOL tables are one-dimensional and each item is selected by a unique key; use of the same key accesses the same element while use of a new key creates a new element.

Next, the possibility of multi-dimensional tables was considered. To examine their usefulness in a beginner's language, illustrative examples currently used for high-school and first-year students were scrutinised. Most examples appeared highly contrived to make use of two dimensional arrays. A typical example is the construction of a table of student numbers and their grades according to course number. There are usually far more courses offered than are taken by a particular student so that the table will typically be sparse; the beginner is then forced to write code (to ignore the empty entries) which is not part of the processing algorithm. What is needed is a table keyed by student-name, with each entry a table of marks keyed by course-name.

From these considerations emerged the MABEL data structure as a table with multiple sub-keys, where a key may be any expression which yields a simple value. A multi-dimensional array would be represented by using the same number of keys at all times. A COBOL or PL/1 structure is achieved by restricting keys to constants. By making use of the full power of an arbitrary number of variable keys, any tree whatsoever may be represented as a MABEL STRUCTURE, without introducing any notion whatsoever of pointers. Some examples illustrating these remarks appear in the appendix; the reader might wish to consult these before continuing.

The following formal rules serve to describe the syntax and semantics of the MABEL STRUCTURE facility:

A A structure may have a simple value or a multiple value. Let S be an arbitrary structure (which might, of course, be a variable or constant or an expression or delivered by a function) and k, k_1, k_2, ... arbitrary keys.

B (i) If S has a multiple value, S may be qualified thus:

S.k

to yield the corresponding (sub-) structure.

(ii) If S has a simple value then S may not be qualified.

C (i) If S has a simple value then S may be explicitly coerced to yield that value thus:

S.

(ii) If S has a multiple value, S may not be so coerced.

Thus, a reference to a sub-structure of S is of one of the forms

S S.k ..... $S.k_1.k_2. \ldots .k_n$

while a reference to an element (simple value) in a structure is of one of the forms

S. S.k. ..... $S.k_1.k_2. \ldots .k_n.$

MABEL structures also permit heap-like memory management. Assuming that the MABEL prelude contains a function UNIQUE*, successive calls of which produce distinct, arbitrary simple values, then the following four groups of code contain equivalent phrases:

A. ALGOL 68:

```
MODE T = STRUCT (REF T link, INT i);
REF  T  p;
```

PL/1:

```
DECLARE 1  T BASED,
           2 LINK POINTER,
           2 I FIXED BINARY;
DECLARE P  POINTER;
```

MABEL:

```
STRUCTURE   T;
CONSTANT    LINK:"LINK";          #FIELD OF T. name
CONSTANT    I:"I";                #FIELD OF T. name
VARIABLE    P;                    #NAME WITHIN T
```

B. ALGOL 68:

```
p : = HEAP T : = (NIL, 17)
```

PL/1:

```
ALLOCATE  T  SET (P);
P → T.LINK = NULL;
P → T.I    = 17;
```

MABEL:

```
    P: = UNIQUE;
T.P : = (| LINK: NULL, I:17  |)
```

C.

ALGOL 68:		link OF p
PL/1	:	P → T.LINK
MABEL	:	T.P.LINK.

D.

```
ALGOL 68:  no explicit garbage collection
   PL/1:   DELETE P→T
  MABEL:   T.P.  := UNDEFINED*;
```

A typical declaration would be

```
VARIABLE   UNIQUEX:
FUNCTION   UNIQUE  RETURNS VALUE:
     UNIQUEX := UNIQUEX & 'Z';
RETURN UNIQUEX
```

---

* The MABEL prelude contains the declaration of a constant UNDEFINED whose value is "$$UNDEFINED". All MABEL simple variables are initialized to that value (including UNIQUEX used in the preceding footnote).

These examples, together with those in the appendix, illustrate how the MABEL structure facility provides all the power deemed necessary while maintaining its essential simplicity. It will be remarked how central the single simple type is to its formulation. We are grateful to Robert Dewar for pointing out the similarity between the STRUCTURE of MABEL and maps in the language SETL, although the MABEL feature was developed quite independently and with different goals.

(iii) The parameter mechanism in MABEL

It is essential that MABEL have a simple parameter mechanism. Further, the beginner should not be burdened with words like VALUE, RESULT, name, reference and their diverse and confusing effects. MABEL therefore has a single parameter transmission mechanism: all parameters are called by "constant", that is, by value without the "free" local variable. Thus no formal parameter can be assigned to, a natural and readily assimilated rule; to a mathematician, the notion of a function changing one of its arguments is quite foreign.

A mechanism is needed for returning one or several values. In MABEL this is achieved by a RETURN statement.

Notice that structures are passed in the same manner. (The specification of a function includes the specification of each parameter as a structure, function or simple value, the latter being the default, as well as the specification of the value(s) returned). Since copying of structured values is only necessary when the actual parameter is used non-locally in the procedure body and assigned to, and such instances can be easily detected statically, the mechanism is not inefficient. One can optimise further by only copying the entries in the structure which are changed (as is done with multiple values in the ALGOL 68S compiler.

A function or procedure is quite permissible as a parameter; we have endeavoured to make it clear from the syntax that it is the function which is passed and not the value yielded by a call.

4. OTHER MABEL CONSTRUCTS

i) Control structures.

MABEL is range-structured, a new range and scope being defined by either a block or a function (procedure) body.

MABEL has a single conditional construct which, following the philosophy alluded to in §2 (iii) may be introduced incrementally without confusing the beginner. A simple conditional would be

```
IF A
IS B THEN statement
```

which may be supplemented by an else part:

```
IF A
IS B THEN statement 1
ELSE statement 2
```

Both statement 1 and statement 2 may comprise a sequence of statements (in which case, each statement in the sequence will be indented; c.f. §4(v)).

The conditional may be further extended:

```
IF A
IS B THEN statement 1
IS C THEN statement 2
IS D|E THEN statement 3
ISNT F|G|H THEN statement 4
ELSE statement n+1
```

A is compared with B,C,D,E,F, etc. consecutively until a match is encountered; in the case of IS, the corresponding statement is executed, while in the case of ISNT attention is turned to the next comparison if there is one.

There appears to be no problem in teaching this construct. We are encouraged to believe that it is highly readable by virtue of supportive evidence from a series of experiments in which a sequence of examples was presented to a number of non-programmers, none of whom had any difficulty in describing the flow of control.

MABEL has two repetitive constructs. The form of the first is

```
FOR id INDEXING structure DO
    statement-list
```

where the statement-list will probably involve structure.id. This permits indexing over an entire structure, and is somewhat similar to its counterpart in B_0, although MABEL has no range concept.

This construct is useful but limited. For example, FOR cannot imply an order in which the elements of the structure are accessed. MABEL therefore provides a second, completely general repetition facility, which permits both counting loops and recursive loops. The simplest form is

```
REPEAT
```

The elaboration consists in replacing REPEAT by a copy of the block in which it occurs. (Notice that REPEAT is not equivalent to a GOTO.) At the head of the block a number of variables may be initialised and they may be updated by REPEAT

```
BEGIN WITH I; = 1
  .
  .
  .
IF I ISNT 10 THEN REPEAT WITH I+1
  .
  .
  .
END
```

Again, this powerful feature is easy to teach. One's first demonstration program is usually

```
BEGIN
VARIABLE X,Y,Z;
GET X,Y;
Z:= X+Y;
PUT Z, X, Y;
END
```

The bright student in the front row usually objects at this point that the program only handles one set of data, and will ask how one may "repeat the process". Upon seeing the program

```
BEGIN
VARIABLE X,Y,Z;
GET X,Y;
Z:= X+Y;
PUT Z,X,Y;
REPEAT
END
```

the same bright student may press his luck, object that the loop is infinite and wonder how one may "put a limit on the number of times it repeats". (If one does not have a bright student, plant an accomplice.) At this stage, one introduces a simple WITH and conditional. Later on one may examine the effect of instructions between REPEAT and END.

Perhaps one should re-emphasize that MABEL is principally designed to provide an introduction to the formulation of algorithms, a wide class of which are recursive. It thus seems entirely appropriate to include a recursive control structure.

These are currently the only loops in MABEL. A possible drawback is that loops analogous to

```
FOR i TO UPB a - 1  DO ... OD
```

must be written using REPEAT. From a number of examples the designers feel that this is not a serious drawback; we would not be

averse to including a further iterative construct but a satisfactory one has still to be found.

(ii) Subroutines, calls and formulae.

The return statement of a function (the last statement in the function body) may return several values:

```
RETURN  I, J,  (I+J),  ARRAY.I.J.
```

Coupled with this, MABEL permits parallel assignations as in

```
I, J := 3, 4;

A, B := B, A;

X,Y,Z:= A,B MULT C,D;
```

The third of these may not be entirely clear. Many languages distinguish between operators and functions. The language provides both, but the programmer (usually) may only define functions. This implies that formulae involving user-defined operations must be written in Polish notation, which is confusing to both the programmer and reader.

MABEL makes no distinction between an operator and a function. A function in MABEL has an arbitrary number of left and right parameters and returns an arbitrary number of results. A function may have no parameters or zero left parameters, but we feel this latter will occur less frequently than might be presupposed. Not only does this afford a natural way to write functions and calls, but is an excellent aide-memoire. One can more easily remember the specifications of the substring function, for example, if one writes

```
A  SUBSTR  I,J
```

rather than

```
SUBSTR (A,I,J)
```

Thus, MABEL function calls are simply an extension of the familiar notation

```
X + Y
```

The possible syntactic ambiguity in, for example

```
.... := X   PLUS  Y,Z
```

is easily resolved by parenthesising calls in a list, as in

```
A  SUBSTR (I + J), (MAX  K,J)
```

The introduction of user-defined infix operators raises the question of how operator priorities shall be handled. The possibilities appear to be

- no priorities, which would require that formulae would have to be completely parenthesized
- integer priorities, as in ALGOL 68
- a small number (say 4) of priority levels, the priority of a new operator being defined by something like

  ```
  PRIORITY   ADD   LIKE +
  ```

- left-to-right (or similar) evaluation, optionally combined with any of the first three.

Of these possibilities, the second appears the least satisfactory; programmers in general and beginners especially remember relative rather than absolute priorities. The third has attractions, but requires a new construct, requires that a user assign a priority even if he does not wish to for a particular operation, and implies the introduction of somewhat arbitrary decisions; one could argue for days about the relative priorities of things like SUBSTR and & (concatenate), for example. To avoid such arbitrariness MABEL currently uses the first alternative and ignores the fourth, but this is somewhat tentative.

As remarked previously, a function (or procedure) specification involves specification of the parameters and values returned; this is naturally true for function parameters, which are specified using a "model" as in

```
FUNCTION    FUNCTION   F  SIMPSON   A,B  RETURNS VALUE:
   MODEL    F X   RETURNS VALUE;
   VARIABLE S, H, N;
            .
            .
            .
   RETURN (H * ((( F A ) + (F B) ) +S))  /3
```

If F were to have a function or procedure parameter, it too would have a model.

(iii) Declarations and constants

All variables must be declared. All declarations must appear at the head of a block on a function (procedure) body. There is no initialisation of variables within declarations.

These restrictions, if indeed one considers them restrictions, are made for pedagogical reasons, although they also assist in one-pass compilation. Consider the examples:

```
BEGIN                                   BEGIN
VARIABLE C;                             VARIABLE C;
C:=  7 ;                                C: = 8;
   .                                       .
   .                                       .
   .                                       .
    BEGIN                                   BEGIN
    VARIABLE D:=C;                          C: = 7;
         .                                       .
         .                                       .
         .                                       .
    VARIABLE C:= 5;                         VARIABLE C;
         .                                       .
         .                                       .
         .                                       .
```

These are both grossly unreadable and will cause intolerable surprises to the newcomer (if not the expert too!).

Compile time constants are permitted as in

```
CONSTANT  PI:'3·14159', PRIMES: (| '1':2, '2': 3, '3': 5, '4': 7|)
```

but the following is not permitted

```
CONSTANT NEWPI:  '4'*(ARCTAN 1);
```

It might be hard to explain to a beginner why NEWPI should be a "constant" and would be hard to prohibit examples like

```
CONSTANT  A:B,   B:A;
```

in a consistent manner.

iv) Transput

MABEL provides two sets of transput primitives. The first is intended for use by rank beginners, and is an extremely simple stream transput facility. The beginner will, at a very early stage, appreciate the meaning of

```
X, Y, Z  : = '1', '2', '3';
```

and shortly thereafter will learn that

```
GET  X, Y, Z
```

where the data contains the list of literals

```
'1', '2', '3'     (or '1'  '2'  '3')
```

means precisely the same thing. The corresponding output construct is typified by

```
PUT  (X + '1'),(Y + '1'),(Z + '1');
```

which produces

```
'2'  '3'  '4'
```

on the printed page. The remaining primitive he may use is

```
NEWLINE
```

This elementary format-free transput is easy to learn but is insufficient for all but the most basic purposes. MABEL also provides two simple record transput primitives:

```
READ   var,  var,  var, ..., var;
WRITE  exprn,  exprn, ..., exprn;
```

which may optionally specify a file-name:

```
READ  A,B,C  FROM STANDIN;
WRITE (A+B), (C+D) TO STANDBACK;
```

Each variable is read from and expression written to a new record in the appropriate file, which is STANDIN or STANDOUT (which are also accessed by GET and PUT) if no file name is specified. MABEL takes the view that the beginning programmer should be made aware that transput operations are essentially string transfers; hence, it is the programmer's responsibility to manipulate the corresponding strings as he wishes (although MABEL will provide various functions to assist him).

It will be observed that all these transput operations involve constructs rather than function calls. We consider the additional seven reserved words introduced (for a total of 32) far preferable to introducing "pseudo" functions with a variable number of parameters, as is the case in ALGOL W.

(v) Operations and other oddments.

The MABEL "system" comprises three components: the kernel, the prelude and libraries.

The kernel incorporates all the MABEL constructs, including a set of "primitives" which will rarely be used by programmers but which are complete in that all operations may be defined in terms of them as described in the next paragraph. The primitives currently used are

SPLIT char FROM string

APPEND string TO string

The kernel includes some global constraints, such as file names but does not include any function or procedure definitions.

In the prelude are defined a host of MABEL functions. These include arithmetic operations such as

+, -, X, /, **, <, > etc., DIV, MOD, FLOOR etc.

string operations such as

& { concatenate}, SUBSTR { "ABCD" SUBSTR 0,2 yields "AB"},

CB, CA { comes alphabetically before and after}, REPLACE,

CONTAINS, REVERSE

and the (non-McCarthy) logical operations

AND, OR, NOT, XOR

The prelude may be written entirely in terms of the kernel, and this will be incorporated in the definition of MABEL. Hopefully this definition of the prelude will be correct and an aid to portability; it should be directly usable by an implementer with possible loss of efficiency being the only penalty.

A number of standard libraries will be included in the MABEL definition. Others may be added at installations.

There are two ways to include comments in MABEL programs.

- All text from # to the end of the current input record is treated as comment.
- Comments may be included within the brackets (*, *).

In the latter case the brackets may be nested and will be matched by the compiler, thus permitting sections of program including comments to be "commented out" for testing purposes.

MABEL currently uses a 56 character set consisting of

- letters A-Z
- digits 0 - 9
- operators + - * / ¬ = < >
- punctuation ( ) ' " ; : . , | # space

ASCII has currently been adopted as the collating sequence. Identifiers are (arbitrarily long) sequences of letters and digits starting with a letter, while function symbols are identifiers or sequences of operators. (Special symbols are never sequences of operators.) We emphasize that generic operators are not permitted; operator

(function) identification follows precisely the same rules as for identifiers.

MABEL encourages good program layout. When a group of statements is to form a single compound statement and there are no explicit delimiters, these statements must be indented,at the same level. This applies in three situations: following THEN, following ELSE and following DO. (All other compound statements have delimiters such as BEGIN...END, PROCEDURE....FINISH and FUNCTION....RETURN.) We feel that good program layout should be mandatory rather than optional; indentation is a powerful, all too frequently ignored control structure.

5. MABEL Implementations

A compiler for the current version of MABEL has been written at the University of Manitoba. It could be made available to anybody willing to experiment with the language. Its brief specifications are as follows:

Computer:	IBM 370 under OS or VS
Source Language:	PL/1
Compiler Size:	200 K
Space Requirements:	Compiler: 256 K Run time: 4K + Object Code + Memory area (run-time parameter)
Parser:	LR(1) with local error correction
Object Code:	370 Object Code Object code is combined with 4K of run-time routines. Run time includes a garbage collector and error traceback.

ACKNOWLEDGEMENT

The first-named author wishes to thank his grade 2 daughter for pointing out that 10-3+2 is equal to 5.

APPENDIX: ILLUSTRATIVE EXAMPLES

Three examples are given, all of which have run successfully under the current MABEL compiler. The first is a simple prime seive program; the second evaluates simple arithmetic expressions while the third, a family tree program, is intended to illustrate the power and potential of the MABEL STRUCTURE facility. Two sets of output appear for the third program; the second set illustrates the run time dump produced in the event of a run time error (here activated by execution of the statement STOP).

OPTIONS SPECIFIED LMARGIN=0,,RMARGIN=8,PAGES=5,CODEGEN=0,LINES=60

OPTIONS IN EFFECT LINES=60,PAGES=005,CODEGEN=0,LMARGIN=00,RMARGIN=08,OBJECT=OBJECT

```
003A     1 (* THIS PROGRAM USES THE SEIVE METHOD TO CALCULATE THE PRIME
           2    NUMBERS BETWEEN 1 AND MAX    *)
           3
           4   BEGIN
           5      CONSTANT FALSE : 'FALSE', TRUE : 'TRUE';
0088       6      CONSTANT ZERO : 0, ONE : 1;
0080       7
           8      STRUCTURE SEIVE;
0086       9      VARIABLE UNDEFINED;
008A      10      CONSTANT MAX : 100;
00D0      11
          12      BEGIN WITH J := 2;
0106      13         IF SEIVE.J. IS UNDEFINED THEN
0120      14                        WRITE J;
0126      15                        BEGIN WITH K := J + J;
016E      16                           IF MAX >= K IS TRUE THEN
01A2      17                                          SEIVE.K. := FALSE;
01AC      18                                          REPEAT WITH K + J;
01DA      19                        END
020E      20         IF MAX > J IS TRUE THEN REPEAT WITH J + ONE;
027C      21      END
0280      22   END
```

NO SYNTAX ERRORS
NO SEMANTIC ERRORS

2
3
5
7
11
13
17
19
23
29
31
37
41
43
47
53
59
61
67
71
73
79
83
89
97

OPTIONS SPECIFIED LMARGIN=0,,RMARGIN=8,PAGES=5,CODEGEN=0,LINES=60

OPTIONS IN EFFECT LINES=60,PAGES=005,CODEGEN=0,LMARGIN=00,RMARGIN=08,OBJECT=OBJECT

```
003A     1        (*
         2             THIS PROGRAM READS EXPRESSION STRINGS AND EVALUATES THEM.
         3             STRINGS ARE OF THE FORM (3+(4*5)). THE OPERATORS ALLOWED
         4             ARE +,-,*,/. BLANKS ARE IGNORED.
         5             EXPRESSIONS MUST BE FULLY PARENTHESIZED.
         6
         7        BEGIN                                                    *)
         8           CONSTANT NULL : '';
006C     9           CONSTANT BLANK : ' ';
0080    10           FUNCTION A DIV B RETURNS VALUE:
00AA    11              VARIABLE Q,R;
00B2    12              Q,R := A DIVREM B;
00E4    13              RETURN Q;
0118    14           FUNCTION +-*/ L,OP,R RETURNS VALUE:
014C    15              VARIABLE RESULT;
0150    16              IF OP
        17              IS '+' THEN
017C    18                 RESULT := L + R;
019E    19              IS '-' THEN
01D6    20                 RESULT := L - R;
01F8    21              IS '*' THEN
0230    22                 RESULT := L * R;
0252    23              IS '/' THEN
028A    24                 RESULT := L DIV R;
02AC    25              IS NULL THEN
02D0    26                 RESULT := L;
02D6    27              RETURN RESULT;
0304    28           FUNCTION EVAL STRING RETURNS VALUE:
0324    29              VARIABLE L,OP,R,VAL,CH;
0338    30              L,OP,R,VAL := NULL,NULL,NULL,STRING;
0350    31              BEGIN
        32                 SPLIT CH FROM VAL;
0374    33                 IF CH
        34                 IS '*'|'-'|'/'|'+' THEN
0412    35                    OP := CH;
0418    36                    REPEAT
        37                 IS '(' THEN
0450    38                    VAL := EVAL VAL;
046C    39                    REPEAT
        40                 IS ')' THEN
04AA    41                    VAL := (+-*/ L,OP,R) & VAL;
04EA    42                 IS NULL THEN
050E    43                    VAL := ')';
0522    44                    REPEAT
        45                 IS BLANK THEN
054C    46                    REPEAT
        47                 ELSE
0558    48                    IF OP IS NULL THEN
0570    49                          L := L & CH;
0592    50                    ELSE
059E    51                       R := R & CH;
```

```
05C0    52                  REPEAT
05C6    53              END
05EE    54            RETURN VAL;
0622    55
        56
        57          BEGIN                       # MAINLINE
        58              VARIABLE INPUT;
0642    59              READ INPUT;
0648    60              IF INPUT ISNT 'END' THEN
0670    61                          WRITE INPUT & ('=' & (EVAL INPUT));
06CA    62                          REPEAT
        63              END
070A    64          END

NO SYNTAX ERRORS
NO SEMANTIC ERRORS



1+2=3
1+(2*3)=7
((1 + 2)*(4 + 7))/7=4
     0.04 SECONDS EXECUTION TIME
        0 STORAGE REGENERATIONS
```

OPTIONS SPECIFIED LMARGIN=0,,RMARGIN=8,PAGES=5,CODEGEN=0,LINES=60

OPTIONS IN EFFECT LINES=60,PAGES=005,CODEGEN=0,LMARGIN=00,RMARGIN=08,OBJECT=OBJECT

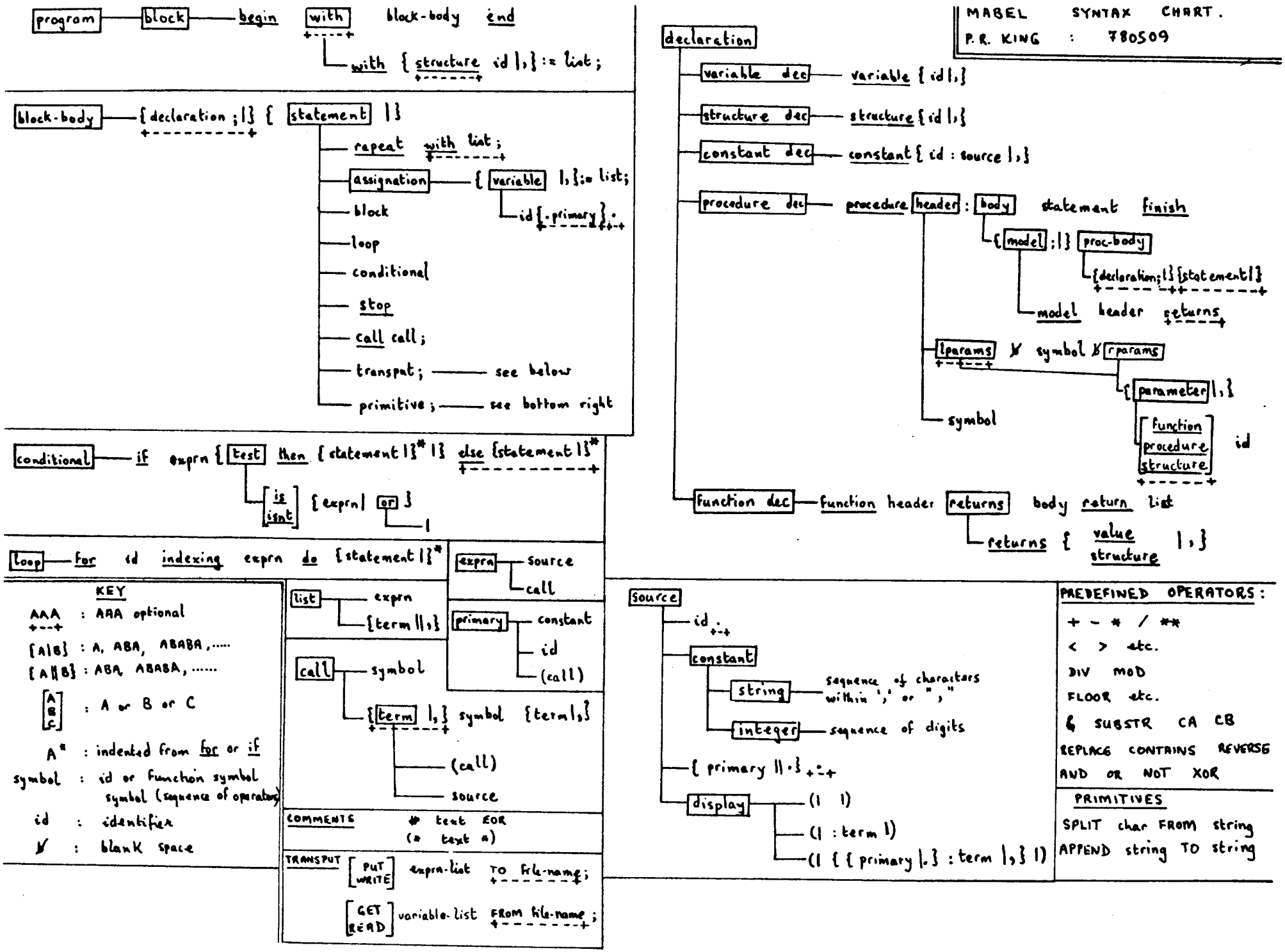
```
003A     1 (*****************************************************************
         2
         3    FAMILY TREE PROGRAM
         4
         5  *****************************************************************)
         6
         7 BEGIN
         8 VARIABLE UNDEFINED;
005A     9 CONSTANT NULL: '';
0070    10 STRUCTURE FAMILYTREE;
0076    11    CONSTANT MOTHER: "MOTHER";
008E    12    CONSTANT FATHER: "FATHER";
00A6    13    CONSTANT CHILD:  "CHILD";
00BE    14    CONSTANT SIBLING: "SIB";
00D4    15       # PARENTS NAME FURTHER QUALIFIES SIBLING;
        16 PROCEDURE BIRTH  NAME,NEWFATHER,NEWMOTHER:
0108    17    VARIABLE MIDDLENAME;
010C    18    IF FAMILYTREE.NAME.FATHER.
        19    IS UNDEFINED THEN
0128    20       FAMILYTREE.NAME :=
        21          (| MOTHER:  NEWMOTHER,
0140    22             FATHER:  NEWFATHER,
014A    23             SIBLING: (| NEWMOTHER: FAMILYTREE.NEWMOTHER.CHILD.,
0166    24                         NEWFATHER: FAMILYTREE.NEWFATHER.CHILD.|),
017E    25             CHILD: NULL |);
0192    26       FAMILYTREE.NEWFATHER.CHILD. := NAME;
019E    27       FAMILYTREE.NEWMOTHER.CHILD. := NAME;
01AA    28    ELSE
01BC    29       WRITE "NAME " & (NAME & "IS NOT UNIQUE; SPECIFY A MIDDLE NAME");
0234    30       GET MIDDLENAME;
023A    31       PUT MIDDLENAME; NEWLINE;
0244    32       CALL BIRTH (NAME & MIDDLENAME),NEWFATHER,NEWMOTHER;
0274    33    FINISH;
02A2    34 PROCEDURE ANCESTORS  NAME,PRTLINE:
02CC    35   WRITE PRTLINE & NAME;
02EE    36   IF FAMILYTREE.NAME.FATHER.
        37   IS NULL THEN
030A    38      WRITE (PRTLINE & "     ") & "FATHER UNKNOWN";
036C    39   ELSE
037E    40      CALL ANCESTORS FAMILYTREE.NAME.FATHER.,(PRTLINE & "     ");
03C0    41   IF FAMILYTREE.NAME.MOTHER.
03C6    42   IS NULL THEN
03E2    43      WRITE (PRTLINE & "     ") & "MOTHER UNKNOWN";
0444    44   ELSE
0456    45      CALL ANCESTORS FAMILYTREE.NAME.MOTHER.,(PRTLINE & "     ");
0498    46   FINISH;
04BA    47 PROCEDURE OFFSPRING  NAME,PRTLINE:
04E4    48    WRITE PRTLINE & NAME;
0506    49    BEGIN WITH NAME,PARENT,PRTLINE :=
        50                  FAMILYTREE.NAME.CHILD., NAME, (PRTLINE & "     ");
057E    51       IF NAME
```

```
        52          ISNT NULL THEN
0590    53             CALL OFFSPRING NAME,PRTLINE;
05A2    54             REPEAT WITH FAMILYTREE.NAME.SIBLING.PARENT., PARENT, PRTLINE;
05C2    55          END
05FC    56       FINISH;
0618    57    PROCEDURE SIBLINGS NAME:
0638    58       FOR PARENT INDEXING FAMILYTREE.NAME.SIBLING DO
065E    59          WRITE "SIBLINGS WITH PARENT: " & PARENT;
06A0    60          BEGIN WITH NAME,PARENT := FAMILYTREE.PARENT.CHILD., PARENT;
06E4    61             IF NAME
        62             ISNT NULL THEN
06F6    63                WRITE NAME;
06FC    64                REPEAT WITH FAMILYTREE.NAME.SIBLING.PARENT., PARENT;
0716    65             END
074A    66       FINISH;
076C    67    VARIABLE COMMAND,P1,P2,P3,P4,P5;
0784    68    FAMILYTREE := (| NULL: (|MOTHER:NULL,FATHER:NULL,CHILD:NULL|) |);
07D0    69    BEGIN
        70       GET COMMAND;
07F2    71       PUT 'COMMAND:',COMMAND;
0812    72       IF COMMAND
        73       IS "END" | UNDEFINED THEN PUT "GOOD BYE.....";
0872    74       IS "BIRTH" THEN
08AE    75          GET P1,P2,P3;
08C0    76          PUT P1,P2,P3; NEWLINE;
08D6    77          CALL BIRTH P1,P2,P3;
08EE    78          REPEAT
        79       IS "ANCESTORS" THEN
092E    80          GET P1;
0934    81          PUT P1; NEWLINE;
093E    82          CALL ANCESTORS P1,NULL;
0950    83          REPEAT
        84       IS "OFFSPRING" THEN
0990    85          GET P1;
0996    86          PUT P1; NEWLINE;
09A0    87          CALL OFFSPRING P1,NULL;
09B2    88          REPEAT
        89       IS "BROTHERS" | "SISTERS" THEN
0A1C    90          GET P1;
0A22    91          PUT P1; NEWLINE;
0A2C    92          CALL SIBLINGS P1;
0A38    93          REPEAT
        94       IS "STOP" THEN STOP
0A78    95       ELSE
0A7E    96          PUT "INVALID COMMAND, RE ENTER:";
0AAA    97          REPEAT
        98       END
0ADE    99    END

NO SYNTAX ERRORS
NO SEMANTIC ERRORS
```

```
COMMAND:   BIRTH      GOD
COMMAND:   BIRTH      ABEL       GOD        GOD
COMMAND:   BIRTH      CAIN       GOD        GOD
COMMAND:   HOWDY      INVALID COMMAND, RE ENTER:     COMMAND:  BIRTH      ADAM       ABEL       CAIN
COMMAND:   BIRTH      EVE        ADAM       CAIN
COMMAND:   ANCESTORS EVE
EVE
      ADAM
            ABEL
                  GOD
                        FATHER UNKNOWN
                        MOTHER UNKNOWN
                  GOD
                        FATHER UNKNOWN
                        MOTHER UNKNOWN
            CAIN
                  GOD
                        FATHER UNKNOWN
                        MOTHER UNKNOWN
                  GOD
                        FATHER UNKNOWN
                        MOTHER UNKNOWN
      CAIN
            GOD
                  FATHER UNKNOWN
                  MOTHER UNKNOWN
            GOD
                  FATHER UNKNOWN
                  MOTHER UNKNOWN
COMMAND:   OFFSPRING GOD
GOD
      CAIN
            EVE
            ADAM
                  EVE
      ABEL
            ADAM
                  EVE
COMMAND:   BROTHERS  EVE
SIBLINGS WITH PARENT: CAIN
EVE
ADAM
SIBLINGS WITH PARENT: ADAM
EVE
COMMAND:   END        GOOD BYE.....
      0.06 SECONDS EXECUTION TIME
         0 STORAGE REGENERATIONS
```

```
COMMAND:  BIRTH     GOD
COMMAND:  BIRTH     ABEL      GOD       GOD
COMMAND:  BIRTH     CAIN      GOD       GOD
COMMAND:  HOWDY     INVALID COMMAND. RE ENTER:    COMMAND:  BIRTH     ADAM      ABEL      CAIN
COMMAND:  BIRTH     EVE       ADAM      CAIN
COMMAND:  STOP
*** ERROR *** USER REQUESTED STOP
    PROGRAM WAS EXECUTING AT OFFSET 0A74 IN PROCEDURE MABEL
*** VARIABLES ACTIVE AT TERMINATION ***
VARIABLE UNDEFINED : "$$UNDEFINED"
CONSTANT VALUE NULL : ""
STRUCTURE FAMILYTREE : (|"":(|"MOTHER":"","FATHER":"","CHILD":"GOD"|),
                        "GOD":(|"MOTHER":"","FATHER":"",
                                "SIB":(|"":""|),"CHILD":"CAIN"|),
                        "ABEL":(|"MOTHER":"GOD","FATHER":"GOD",
                                 "SIB":(|"GOD":""|),"CHILD":"ADAM"|),
                        "CAIN":(|"MOTHER":"GOD","FATHER":"GOD",
                                 "SIB":(|"GOD":"ABEL"|),"CHILD":"EVE"|),
                        "ADAM":(|"MOTHER":"CAIN","FATHER":"ABEL",
                                 "SIB":(|"CAIN":"","ABEL":""|),"CHILD":"EVE"|),
                        "EVE":(|"MOTHER":"CAIN","FATHER":"ADAM",
                                "SIB":(|"CAIN":"ADAM","ADAM":""|),"CHILD":""|)|)
CONSTANT VALUE MOTHER : "MOTHER"
CONSTANT VALUE FATHER : "FATHER"
CONSTANT VALUE CHILD : "CHILD"
CONSTANT VALUE SIBLING : "SIB"
VARIABLE COMMAND : "STOP"
VARIABLE P1 : "EVE"
VARIABLE P2 : "ADAM"
VARIABLE P3 : "CAIN"
VARIABLE P4 : "$$UNDEFINED"
VARIABLE P5 : "$$UNDEFINED"
     0.10 SECONDS EXECUTION TIME
        0 STORAGE REGENERATIONS
```


MABEL SYNTAX CHART.
P.R. KING : 780509
program — block — begin — with — block-body — end
with — with { structure id |,} := list ;
block-body — { declaration ; |} { statement |}
statement — repeat with list ;
assignation — { variable |,} := list ;
variable — id { . primary }
block
loop
conditional
stop
call call ;
transput ; — see below
primitive ; — see bottom right
declaration
variable dec — variable { id |,}
structure dec — structure { id |,}
constant dec — constant { id : source |,}
procedure dec — procedure header : body statement finish
header — lparams symbol rparams
lparams — { parameter |,}
parameter — function / procedure / structure — id
symbol
body — { model ; |} proc-body
proc-body — { declaration ; |} { statement |}
model — model header returns
returns
function dec — function header returns body return list
returns — returns { value / structure |,}
conditional — if exprn { test then { statement |}* |} else { statement |}*
test — [is / isnt] { exprn | or |}
loop — for id indexing exprn do { statement |}*
KEY
AAA : AAA optional
{A|B} : A, ABA, ABABA, ...
{A||B} : ABA, ABABA, ...
[A / B / C] : A or B or C
A* : indented from for or if
symbol : id or function symbol
symbol (sequence of operators)
id : identifier
␣ : blank space
list — exprn — { term ||,}
call — symbol — { term |,} symbol { term |,}
term — (call) — source
exprn — source — call
primary — constant — id — (call)
COMMENTS # text EOR
(* text *)
TRANSPUT [PUT / WRITE] expr-list TO file-name ;
[GET / READ] variable-list FROM file-name ;
source — id
constant — string — sequence of characters within ' ' or " "
integer — sequence of digits
{ primary |.}
display — (| |)
(| : term |)
(| { { primary |.} : term |,} |)
PREDEFINED OPERATORS :
+ - * / **
< > etc.
DIV MOD
FLOOR etc.
& SUBSTR CA CB
REPLACE CONTAINS REVERSE
AND OR NOT XOR
PRIMITIVES
SPLIT char FROM string
APPEND string TO string