The Programming Language BPL

M. H. Williams
Department of Computer Science, Heriot-Watt University, 79 Grassmarket, Edinburgh EH1 2HJ, UK

BPL is a programming language which was developed from the two languages BASIC and Pascal. The chief aim in designing BPL was to create a language as simple to use as BASIC with the additional data types and control structures which make Pascal attractive to use. It is intended in the first instance as a language to teach programming but it can be used for any application where BASIC might have been used, and in many such cases it is simpler to use and the result more readable than if BASIC had been used. For instance, the record facilities make the language more amenable to commercial applications, and the pointer facilities to manipulating complex linked data structures.

1. INTRODUCTION

With the abundance of programming languages already available, it is with great trepidation that one dares to announce yet another one. Nevertheless the language described here, BPL, has a number of merits and for this reason the reader is exhorted to read on and judge it for himself.

The language originates from a search for a suitable language to teach to two different types of undergraduate student—a first language for undergraduate Computer Science students and a language for other undergraduate science (or non-science) students. Two languages which suggested themselves for this purpose are:

(a) BASIC. This is simple to teach and simple to learn. The BASIC system is geared to helping the user. After a very limited period of time the student who has been taught BASIC has a better understanding of (or, if you prefer it, is less confused about) the process of computing than a student who has learnt a conventional batch language. Input/output is particularly simple and the notion of numeric and string variables is ideal for the beginner; file handling and matrix manipulation facilities enable more sophisticated applications to be handled very simply.

(b) Pascal. This is a well-structured language (i.e. one which lends itself to the construction of well-structured programs) and is thus a better language from the point of view of teaching good programming habits and style. In addition Pascal has a rich collection of data types which lend themselves to programming a wide range of applications. In particular, pointer, record and definable scalar types and more flexible array subscripts enable one to write more readable programs.

Both languages have very appealing features and either language could be used to teach both groups of students, although in general BASIC would probably be preferred for the non-Computer Science student while Pascal might be more attractive for teaching the Computer Science student. However, it would be nice to have a language which has the advantages of both BASIC and Pascal. It was for this reason that BPL was developed.

BPL (which stands for BASIC-Pascal-Liaison or Better Programming Language or simply a conversational programming language which was designed after APL) is designed as a two-tier language which combines the advantages of BASIC and Pascal. The lower level, a self-sufficient subset of BPL, is very similar to BASIC and provides a language suitable for teaching to service courses. It has all the advantages of BASIC together with those of the structured control structures of Pascal; in addition the facilities of the full language can be called upon if necessary. The full language has the advantage of the simplicity of BASIC combined with the flexibility of Pascal.

Since BASIC and Pascal are two rather different languages, BPL was designed to be as compatible as possible with BASIC but with the concepts of Pascal superimposed onto it. As both BASIC and Pascal have a very simple syntactic structure with unique keywords at the start of most statements and declarations, BPL has been constructed likewise. This simplifies matters both for the compiler-writer and for the user. Furthermore, BPL is designed to be compiled or interpreted as desired.

BPL is not intended to be just a teaching language. Any programmer familiar with BASIC should find BPL very similar to BASIC and in many cases easier to use because of the additional facilities. In particular the notions of records, pointers and definable scalar types can make programming simpler and programs more readable.

2. DEFINITION OF THE LANGUAGE

2.1 Statements and commands

Like BASIC, BPL is intended as a conversational programming language. The user interacts with the system and may give (a) commands, (b) indirect statements, and (c) direct statements.

A command is a directive to the system, concerned with the filing, retrieving, listing, tracing, running, etc., of programs.

An indirect statement is a program statement which is preceded by a statement number (an integer). When an indirect statement is encountered, it is stored away for execution at a later stage.

1. ⟨indirect statement⟩ : : = ⟨statement no⟩ [2]
⟨statement part⟩ [73]

CCC-0010-4620/82/0025-0289 $9.00

2. ⟨statement no⟩::=⟨non-zero digit⟩ $\{\langle digit\rangle\}_0^\infty$ [4,3]
3. ⟨digit⟩::=0|⟨non-zero digit⟩ [4]
4. ⟨non-zero digit⟩::=1|2|3|4|5|6|7|8|9

The extended form of BNF described by Lee and others is used here.[1] Any portion of a production enclosed in braces $\{\ \}_i^j$ indicates repetition of the contents at least i and not more than j times. Production numbers for nonterminals occurring on the right hand sides of productions are given in square brackets at the end of each line.

A direct statement is a program statement which is not preceded by a statement number. Such a program statement is executed when it is encountered.

5. ⟨direct statement⟩::=⟨stm⟩ [74]

In BASIC indirect statements may be entered in any order and the system puts them into ascending order of statement number; likewise in BPL the order of entering indirect statements is unimportant except for the restrictions discussed in the section on Procedures and Functions (Section 2.9).

A statement number is any unsigned integer in the range 1 to *maxstmno*, where *maxstmno* is an implementation dependent limit.

One important difference between BPL and BASIC as regards format is that in BASIC spacing in a program statement or command is generally unimportant whereas in BPL keywords, numbers and identifiers must be separated by at least one space or some nonalphanumeric symbol. Whereas in BASIC a program statement may not overflow over the end of a line, in BPL a continuation indicator (&) at the end of a line indicates that the statement continues on the next line. The number of continuation lines permitted will in general be system dependent but a minimum of 500 characters should be permitted.

2.2 Variables, types and identifiers

Each data item of a program may be stored in a 'cell' which may be either (a) a simple variable, (b) a field of a record or (c) an element of an array.

A simple variable is a simple cell capable of containing a single data item of a particular type; a record is a composite cell consisting of a collection of one or more simple cells not necessarily of the same type; an array is a collection of cells (simple or composite), all of the same type.

The type of a data item may be any of the following:

(a) Simple data types—REAL, INTEGER, BOOLEAN;

(b) Strings—these are divided into *fixed length strings* which are declared as STRING(n) where n is an integer denoting the number of characters in the string, and *variable length strings* which are declared as STRING (⟨=n) where n is an integer denoting the maximum number of characters in the string;

6. ⟨string type⟩::=STRING ($\{<=\}_0^1$⟨integer⟩) [14]

(c) Definable scalar type—this is a data type which can only assume values from some ordered set of identifiers specified in the type declaration;

(d) Subrange type—this is a special type of integer or scalar type which can only take on values in a limited range;

(e) Pointer type—this represents an address of a data item and is used mainly with dynamic storage allocation.

In addition there is a record type which is a collection of one or more named components or *fields*; the type of each field may be any of the above five types or may be a record type.

In what follows, one must distinguish between an identifier used as the name of a variable, record or array, and an identifier used as the name of a new data type. There are two types of identifiers used in the language—these are plain identifiers

7. ⟨plain identifier⟩::=⟨letter⟩ $\{\langle letter\ or\ digit\rangle\}_0^\infty$ [10] [8]
8. ⟨letter or digit⟩::=⟨letter⟩|⟨digit⟩ [10, 3]

and string identifiers

9. ⟨string identifier⟩::=⟨letter⟩ $\{\langle letter\ or\ digit\rangle\}_0^\infty$\$ [10] [8]
10. ⟨letter⟩::=A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z
11. ⟨identifier⟩::=⟨plain identifier⟩|⟨string identifier⟩ [7] [9]

Plain identifiers are used for all identifiers referring to types and to non-string variables and arrays; string identifiers refer to string variables and arrays. These identifier forms are compatible with names of variables and arrays in BASIC while at the same time permitting longer and more meaningful names too. As the Ent in *The Lord of the Rings*[2] so wisely commented, 'Real names tell you the story of the things they belong to in my language'.

One important point to note is that simple variables may either be declared or may be allocated automatically on encountering the first reference to the identifier (as in BASIC). In this language there are three modes of operation: (a) No declaration by default—in this mode all identifiers must be declared before they are used.

(b) Declaration by default with check—in this mode any identifier not declared will be allocated automatically as a variable of appropriate type on encountering the first reference to it. At the end of the segment, the system lists all automatically allocated identifiers and the lines where they are encountered.

(c) Declaration by default without check—same as (b) but without obtaining a listing of automatically allocated variables for each segment.

The type of an automatically allocated variable is determined by the type of the identifier—if it is a plain identifier, the variable is assumed to be real; if it is a string identifier, the variable is assumed to be a variable length string whose maximum length is some implementation dependent constant, *defaultstrlen* (this should be at least 64 characters).

This same default rule for determining the type of automatically allocated simple variables also applies to other situations. For example, arrays, fields and formal parameters must all be declared but if no type is specified the type is assumed to be either real or variable length string according to whether the identifier is plain or string.

2.3 Constants

A numeric constant is a signed or unsigned integer or real number.

12. ⟨numeric constant⟩::={+|−}$_0^1$⟨number⟩ [13]
13. ⟨number⟩::=⟨integer⟩|⟨real number⟩ [14, 15]
14. ⟨integer⟩::={⟨digit⟩}$_1^\infty$ [3]
15. ⟨real number⟩::={⟨digit⟩}$_0^\infty$.{⟨digit⟩}$_1^\infty$ [3, 3]
{E⟨exponent⟩}$_0^1$|{⟨digit⟩}$_1^\infty$E⟨exponent⟩ [16, 3, 16]
16. ⟨exponent⟩::={+|−}$_0^1${⟨digit⟩}$_1^\infty$ [3]

A string constant is a string of characters enclosed in double quotation marks. If a double quotation mark itself forms part of the string, it is written twice.

17. ⟨string constant⟩::=
{"{⟨non-quote character⟩}$_0^\infty$"}$_1^\infty$

where ⟨non-quote character⟩ is any character other than a quotation mark.

A Boolean constant is one of the values TRUE or FALSE.

18. ⟨Bool constant⟩::=TRUE|FALSE

A scalar constant is a plain identifier.

19. ⟨scalar constant⟩::=⟨plain identifier⟩ [7]

The only pointer constant is NIL:

20. ⟨pointer constant⟩::=NIL

2.4 Type definitions

A data type represents a class of values which a data item can have. The class of values associated with the standard data types REAL, INTEGER and BOOLEAN are fairly obvious. On the other hand, definable scalar types, record types, etc., have no obvious class of values associated with them and these must be spelled out in type definitions. Thus a type definition defines an identifier to be associated with a class of values.

21. ⟨type definition⟩::=TYPE⟨type identifier⟩= [22]
⟨defined type⟩ [23]
22. ⟨type identifier⟩::=⟨plain identifier⟩ [7]
23. ⟨defined type⟩::=⟨scalar type⟩ [24]
|⟨subrange type⟩|⟨pointer type⟩| [25, 35]
⟨record type⟩|⟨array type⟩ [26, 36]

A scalar type defines an ordered set of scalar constant values.

24. ⟨scalar type⟩::=(⟨scalar constant⟩ [19]
{, ⟨scalar constant⟩}$_1^\infty$) [19]

For example

```
10 TYPE SEX = (MALE, FEMALE)
20 TYPE DAY = (MON, TUES, WED, THURS, FRI,
                                  SAT, SUN)
```

A subrange type defines a subset of the set of integers or a subset of a set of scalars

25. ⟨subrange type⟩::=⟨integer⟩..⟨integer⟩| [14, 14]
⟨scalar constant⟩..⟨scalar constant⟩ [19, 19]

For example

```
10 TYPE DIGIT = 0..9
20 TYPE WEEKDAY = MON..FRI
```

A record type defines a composite type consisting of a number of fields, each with a type and an identifier associated with it.

A record definition may include one or more variant parts. A variant part consists of a set of different subrecords corresponding to different values of a particular field known as the tag field. The value of the tag field determines the subrecord assumed at any instant. A tag field must be of type INTEGER, a defined scalar type or subrange.

26. ⟨record type⟩::=RECORD⟨field def⟩ [27]
{; ⟨field def⟩}$_0^\infty$END [27]
27. ⟨field def⟩::=⟨field name⟩{:⟨type⟩}$_0^1$| [33, 31]
⟨variant part⟩ [28]
28. ⟨variant part⟩::=
ON⟨tag field name⟩:⟨variant type⟩ [29, 30]
{CASE⟨constant⟩{, ⟨constant⟩}$_0^\infty$: [34, 34]
⟨field def⟩{;⟨field def⟩}$_0^\infty$}$_1^\infty$ [27, 27]
{DEFAULT:⟨field def⟩{;⟨field def⟩}$_0^\infty$}$_0^1$ [27, 27]
ENDON
29. ⟨tag field name⟩::=⟨field name⟩ [33]
30. ⟨variant type⟩::=INTEGER|⟨type identifier⟩ [22]
31. ⟨type⟩::={POINTER TO|↑}$_0^1$⟨simple type⟩ [32]
32. ⟨simple type⟩::=REAL|INTEGER|BOOLEAN|
⟨string type⟩|⟨type identifier⟩ [6, 22]
33. ⟨field name⟩::=⟨identifier⟩ [11]
34. ⟨constant⟩::=⟨numeric constant⟩| [12]
⟨Bool constant⟩| [18]
⟨scalar constant⟩|⟨pointer constant⟩| [19, 20]
⟨string constant⟩ [17]

ON, CASE, DEFAULT, ENDON, etc., may each appear at the start of a new continuation line. The space allocated for a record with variant parts will be determined by the maximum size of each variant part.

If a field is of type string, the field name must be a string identifier, otherwise it must be a plain identifier. If the ⟨type⟩ part is omitted from a field definition, the type of the field is assumed according to the previously mentioned rule: if the identifier is plain, the type of the field is assumed to be REAL, if it is string, the type is assumed to be variable length string with the default maximum length. For example,

```
10 TYPE SEXTYPE = (MALE, FEMALE)
20 TYPE MARTYPE = (SINGLE, MARRIED,
                    DIVORCED, SEPARATED)
30 TYPE EMPLREC = RECORD EMPLNO:
                              INTEGER;  &
     NAME$:STRING(< = 20);
                    ADDRESS$;WAGE;      &
     ON SEX:SEXTYPE                     &
     CASE MALE:ARMYNO$:STRING(15)       &
     CASE FEMALE:MAIDENNAME$:
                         STRING(< = 20) &
     ENDON;                             &
     ON MARSTAT:MARTYPE                 &
     CASE SINGLE:NEXTOFKIN$:
                         STRING(< = 20) &
     DEFAULT:NOOFCHILDREN:INTEGER       &
     ENDON                              &
END
```

A diagram of this record is shown in Fig. 1.

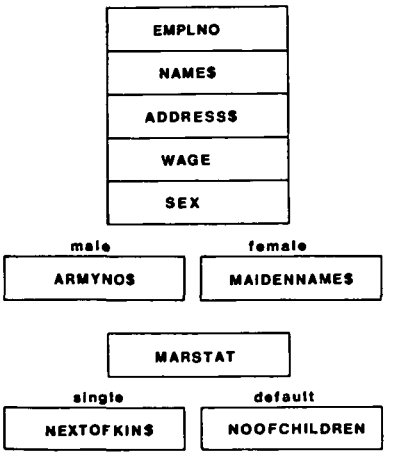


Figure 1. Diagram of record with variant parts.

A pointer type defines a pointer to a data item of a particular type.

35. ⟨pointer type⟩::=POINTER TO ⟨simple type⟩|↑⟨simple type⟩ [32, 32]

There is no semantic difference between these two forms. An example of pointer definitions is

```
100 TYPE INTPTR = POINTER TO INTEGER
110 TYPE REC = RECORD LLINK:POINTER TO
  & REC;INFO;RLINK:POINTER TO REC END
```

An array type defines an ordered collection of items of the same type.

36. ⟨array type⟩::=ARRAY[⟨bound⟩ {,⟨bound⟩}$_0^\infty$]OF⟨type⟩ [37] [37, 31]
37. ⟨bound⟩::={⟨integer⟩..}$_0^1$⟨integer⟩| ⟨type identifier⟩ [14, 14] [22]

If a bound is given as a single integer, this represents the upper bound, the lower bound being zero; if it is given as a pair of integers (subrange), the first represents the lower bound, the second the upper bound. If the bound is a type identifier, it may only be a scalar or subrange type. For example

```
20 TYPE ARR1 = ARRAY [DAY] OF INTEGER
30 TYPE ARR2 = ARRAY [1..10, 1..20]
                  OF STRING (< = 20)
```

All type identifiers and field names must be unique identifiers.

2.5 Variable and array declarations

Simple variables need not necessarily be declared if their types are REAL or variable length string (with maximum length equal to the default) as such variables will be allocated automatically on encountering a reference to the variable. However, if one does wish to declare such variables, or if a variable has type other than these two default types, variable declarations are used.

38. ⟨variable declaration⟩::=VAR⟨identifier⟩ {,⟨identifier⟩}$_0^\infty$:⟨type⟩ [11] [11, 31]

For example

```
200 VAR I,J,K:INTEGER
210 VAR REC1:EMPLREC
220 VAR A$,B$:STRING(< = 30)
```

Static arrays are declared using a TYPE and VAR statement as outlined; for example

```
10 TYPE ARRTYP = ARRAY[1..10]OF INTEGER
21 VAR SUMARR:ARRTYP
```

However, in BPL one may also use dynamic arrays, i.e. arrays whose bounds can be determined at run-time. In order to declare a dynamic array one uses a DIM statement which has the form

39. ⟨dim stm⟩::=DIM⟨array part⟩ {;⟨array part⟩}$_0^\infty$ [40] [40]
40. ⟨array part⟩::=⟨id list⟩ (⟨dbound⟩{,⟨dbound⟩}$_0^\infty$){OF⟨type⟩}$_0^1$ [41] [42, 42, 31]
41. ⟨id list⟩::=⟨array name⟩ {,⟨array name⟩}$_0^\infty$ [43] [43, 43]
42. ⟨dbound⟩::={⟨numeric exp⟩..}$_0^1$⟨numeric exp⟩ [56] [56, 56]
43. ⟨array name⟩::=⟨identifier⟩ [11]

Each array name must be a string identifier if the elements of the array are strings otherwise it must be a plain identifier. Once again if an 'OF⟨type⟩' clause is omitted, the types of the arrays are determined by default—REAL arrays if the identifiers are plain, variable length string arrays if the identifiers are string identifiers. Whether the 'OF⟨type⟩' clause is present or not, all array names in an identifier list must be of the same type. For example

```
10 DIM MATRIX,IJ(A,B) OF
                  INTEGER;NAME$(1..B)
```

declares three arrays, two two-dimensional integer arrays MATRIX and IJ whose row subscript runs from 0 to *A* and column subscript from 0 to *B*, and one one-dimensional variable length string array NAME$ whose subscript ranges from 1 to *B*.

A DIM statement may occur in any segment of a program. When it is executed, the appropriate arrays will be set up. On exit from the segment to the calling segment the arrays are lost so that on re-entry to the segment one must execute a DIM statement once again before the arrays will be set up. The values of the bounds of the array may be different on each entry to the segment; however, within a segment the bounds of an array will remain fixed from the time that the DIM statement is executed to the time that one exits from the segment. The scope and uniqueness of variable and array names is discussed in Section 2.9.

2.6 Reference to data items

A data item may be a simple variable in which case it may be referred to directly by using its identifier. If the data item is a pointer variable then using its identifier will yield the pointer value contained in the variable, while if the pointer variable identifier is followed by the symbol '↑', the value of the data item pointed to by this variable will be used, e.g.

```
110 VAR IPTR:POINTER TO INTEGER
```

reference to IPTR yields the pointer value,
reference to IPTR↑ yields the integer value pointed to.

If the data item is a record variable (say X), then reference to X refers to the whole record; if one wishes to access a field Y of record X, one writes 'X.Y'. Likewise a data item may be an array element in which case it is referred to by a subscripted variable and if the array is of type pointer or record, this may in turn be followed by a sequence of ↑s or .⟨field name⟩s. For example,

```
TYPE AREC = RECORD FIELD1;FIELD2:
                   POINTER TO AREC END
DIM ARECARR (10) OF POINTER TO AREC
ARECARR(I)↑.FIELD2↑.FIELD1
```

In the case of a string data item, a substring may be accessed by specifying the first and last character positions of the substring in square parentheses after the string identifier, e.g.

```
STR$ [2,I + 1]
```

Thus a variable is defined by the following BNF definition:

44. ⟨variable⟩::=⟨array name⟩(⟨expression⟩ [43, 48]
{,⟨expression⟩}$_0^\infty$)⟨rest var⟩| [48, 45]
⟨identifier⟩⟨rest var⟩ [11, 45]
45. ⟨rest var⟩::={⟨field or pointer part⟩}$_0^\infty$ [46]
{⟨substring part⟩}$_0^1$ [47]
46. ⟨field or pointer part⟩::=.⟨field name⟩|↑ [33]
47. ⟨substring part⟩::=[⟨numeric exp⟩, [56]
⟨numeric exp⟩] [56]

2.7 Expressions

An expression consists of variables, constants and function calls separated by operators and parentheses. A BNF definition of an expression is as follows:

48. ⟨expression⟩::=⟨simple exp⟩|⟨Bool exp⟩ [55, 49]
49. ⟨Bool exp⟩::=⟨Bool term⟩ [50]
{OR⟨Bool term⟩}$_0^\infty$ [50]
50. ⟨Bool term⟩::=⟨Bool factor⟩{AND [51]
⟨Bool factor⟩}$_0^\infty$ [51]
51. ⟨Bool factor⟩::={NOT}$_0^1$⟨Bool primary⟩ [52]
52. ⟨Bool primary⟩::=⟨variable⟩| [44]
⟨Bool constant⟩|⟨fn designator⟩ [18, 62]
|(⟨Bool exp⟩)|⟨relation⟩ [49, 53]
53. ⟨relation⟩::=⟨simple exp⟩ [55]
⟨relational op⟩⟨simple exp⟩ [54, 55]
54. ⟨relational op⟩::=< =|<|> =|>|= |<>
55. ⟨simple exp⟩::=⟨numeric exp⟩ [56]
|⟨scalar exp⟩| [63]
⟨pointer exp⟩|⟨string exp⟩ [65, 64]
56. ⟨numeric exp⟩::={⟨adding op⟩}$_0^1$⟨term⟩ [61, 57]
{⟨adding op⟩⟨term⟩}$_0^\infty$ [61, 57]
57. ⟨term⟩::=⟨factor⟩{⟨multiplying op⟩ [58, 60]
⟨factor⟩}$_0^\infty$ [58]
58. ⟨factor⟩::=⟨primary⟩{**⟨primary⟩}$_0^\infty$ [59, 59]
59. ⟨primary⟩::=⟨numeric var⟩|⟨number⟩| [71, 13]
⟨fn designator⟩|(⟨numeric exp⟩) [62, 56]
60. ⟨multiplying op⟩::=*|/
61. ⟨adding op⟩::=+|−
62. ⟨fn designator⟩::=⟨fn name⟩ [72]
({⟨actual par⟩{,⟨actual par⟩}$_0^\infty$}$_0^1$) [67, 67]
63. ⟨scalar exp⟩::=⟨scalar var⟩ [68]
|⟨scalar constant⟩|⟨fn designator⟩ [19, 62]
64. ⟨string exp⟩::=⟨string primary⟩ [66]
{+⟨string primary⟩}$_0^\infty$ [66]
65. ⟨pointer exp⟩::=⟨pointer var⟩| [69]
⟨pointer constant⟩|CREATE(⟨type⟩)| [20, 31]
ADDRESSOF(⟨variable⟩)| [44]
⟨fn designator⟩ [62]
66. ⟨string primary⟩::=⟨string var⟩ [70]
|⟨string constant⟩|⟨fn designator⟩ [17, 62]
67. ⟨actual par⟩::=⟨expression⟩ [48]
68. ⟨scalar var⟩::=⟨variable⟩ [44]
69. ⟨pointer var⟩::=⟨variable⟩ [44]
70. ⟨string var⟩::=⟨variable⟩ [44]
71. ⟨numeric var⟩::=⟨variable⟩ [44]
72. ⟨fn name⟩::=⟨identifier⟩ [11]

A summary of the operators is given in Table 1.

The functions CREATE and ADDRESSOF are two pointer-valued functions whose effect is as follows:

(a) CREATE is equivalent to the 'new' function in Pascal. It creates an instance of the data type specified (using the concept of a 'heap' as in Pascal) and returns a pointer to this item;

Table 1. Summary of operators in BPL expressions

Operator	Operation	No. of operands	Type of operands	Type of result
NOT	Boolean NOT	1	Boolean	Boolean
AND	Boolean AND	2	Boolean	Boolean
OR	Boolean OR	2	Boolean	Boolean
= <>	comparison	2	numeric, scalar, string, pointer — both of same type	Boolean
< <= > >=	comparison	2	numeric, scalar, string — both of same type	Boolean
+	addition	2	numeric	integer unless either or both of operands is real
−	subtraction	2	numeric	
*	multiplication	2	numeric	
/	division	2	numeric	real
**	exponentiation	2	numeric	real
+	unary plus	1	numeric	real if operand is real
−	sign inversion	1	numeric	otherwise integer
+	concatenation	2	string	string

(b) ADDRESSOF takes as argument the name of a simple variable, array element or record and returns the address of this item.

In each case the type of object pointed to must match the type expected (e.g. in an assignment statement, a relation or a procedure call) and if it does not match, an error must be flagged.

The standard functions available include all the standard BASIC functions (SQR, LOG, EXP, INT, LEN, MOD etc.) as well as SUCC(scalar exp)—yields the value succeeding the value of the scalar expression; PRED(scalar exp)—yields the value preceding the value of the scalar expression.

2.8 Statements

The statements available in the language are as follows:

73. ⟨statement part⟩::=⟨stm⟩|⟨proc or fn stm⟩| [74, 131]
⟨type definition⟩|⟨variable declaration⟩ [21, 38]
74. ⟨stm⟩::=⟨assign stm⟩|⟨input–output stm⟩| [75, 87]
⟨dim stm⟩|⟨control stm⟩|⟨rem stm⟩ [39, 104, 128]

Assignment statement. The assignment statement (⟨assign stm⟩) enables a numeric, Boolean, string, scalar or pointer expression to be assigned to the corresponding destination. It also permits the copying of a record variable to another variable of the same type or the performance of certain array manipulations and the assignment of the result to an array. It has the form

75. ⟨assign stm⟩::={LET}$_0^1$⟨numeric dest⟩ [76]
=⟨numeric exp⟩| [56]
{LET}$_0^1$⟨Bool dest⟩=⟨Bool exp⟩| [77, 49]
{LET}$_0^1$⟨string dest⟩=⟨string exp⟩| [78, 64]
{LET}$_0^1$⟨scalar dest⟩=⟨scalar exp⟩| [79, 63]
{LET}$_0^1$⟨pointer dest⟩=⟨pointer exp⟩| [80, 65]
{LET}$_0^1$⟨record dest⟩=⟨record var⟩| [81, 85]
{LET}$_0^1$⟨array name⟩ [43]
={⟨array exp⟩|⟨numeric exp⟩} [82, 56]
76. ⟨numeric dest⟩::=⟨numeric var⟩|⟨fn name⟩ [71, 72]
77. ⟨Bool dest⟩::=⟨Bool var⟩|⟨fn name⟩ [86, 72]
78. ⟨string dest⟩::=⟨string var⟩|⟨fn name⟩ [70, 72]
79. ⟨scalar dest⟩::=⟨scalar var⟩|⟨fn name⟩ [68, 72]
80. ⟨pointer dest⟩::=⟨pointer var⟩|⟨fn name⟩ [69, 72]
81. ⟨record dest⟩::=⟨record var⟩ [85]
82. ⟨array exp⟩::=⟨array term⟩ [83]
{+⟨array term⟩}$_0^\infty$ [83]
83. ⟨array term⟩::=⟨array factor⟩ [84]
{*⟨array factor⟩}$_0^\infty$ [84]
84. ⟨array factor⟩::=⟨array name⟩ [43]
|TRANSPOSE(⟨array name⟩) [43]
85. ⟨record var⟩::=⟨variable⟩ [44]
86. ⟨Bool var⟩::=⟨variable⟩ [44]

If the right hand side is a single array name, the contents of the array are copied to the array on the left hand side. If the right hand side is a numeric expression its value is copied to every element of the array. If addition or multiplication or TRANSPOSE is specified, array addition/multiplication/transpose is performed. If a real expression is assigned to an integer or subrange variable, the value is rounded to the nearest integer before assignment. Any value assigned to a subrange variable is also checked to ensure that it lies within the permissible limits for the subrange type.

Input–output statements. Input–output statements are as follows:

87. ⟨input–output stm⟩::=⟨data statement⟩ [88]
|⟨read statement⟩|⟨print statement⟩| [90, 94]
⟨file statement⟩ [101]
|⟨reset statement⟩ [103]

As in BASIC, items of data may be stored in a DATA statement which has the form

88. ⟨data statement⟩::=DATA⟨data constant⟩ [89]
{,⟨data constant⟩}$_0^\infty$ [89]
89. ⟨data constant⟩::=⟨numeric constant⟩| [12]
⟨string constant⟩|⟨scalar constant⟩| [17, 19]
⟨Bool constant⟩ [18]

There is only one input statement which has the form

90. ⟨read statement⟩::=READ
{#⟨input channel⟩:}$_0^1$ [91]
{USING BINARY:}$_0^1$⟨destination list⟩ [92]
91. ⟨input channel⟩::=DATA|CONS|⟨numeric exp⟩ [56]

If DATA is specified, input is taken from DATA statements (i.e. standard BASIC READ), if CONS, input is taken from the console (equivalent to BASIC INPUT) and if an expression with *n* as value, input is taken from file *n*. If the channel specification is omitted, #DATA: is assumed.

Data is read from the specified source and unpacked into the destinations specified in the destination list—these may be simple variables, subscripted variables, fields, arrays (in which case data items are read into each element of the array) or records (in which case data items are read into each field of the record) provided that all destinations have the same type.

92. ⟨destination list⟩::=⟨dest⟩{,⟨dest⟩}$_0^\infty$ [93, 93]
93. ⟨dest⟩::=⟨numeric var⟩|⟨Bool var⟩| [71, 86]
⟨string var⟩|⟨scalar var⟩| [70, 68]
⟨record var⟩|⟨array name⟩ [85, 43]

The USING option will be described under the PRINT statement below. Note that one may not read a value into a pointer variable nor into a record containing a field of type pointer nor into an array of pointers.

The only output statement has the form

94. ⟨print statement⟩::=PRINT
{#⟨output channel⟩:}$_0^1$ [98]
{USING BINARY:}$_0^1$⟨print list⟩ [95]
95. ⟨print list⟩::={{⟨print item⟩}$_0^1$ [96]
⟨print separator⟩}$_0^\infty${⟨print item⟩}$_0^1$ [97, 96]
96. ⟨print item⟩::=⟨print exp⟩⟨print format⟩ [100, 99]
|TAB(⟨numeric exp⟩) [56]
|NL(⟨numeric exp⟩)|⟨record var⟩ [56, 85]
|⟨array name⟩⟨print format⟩ [43, 99]
97. ⟨print separator⟩::=,|;
98. ⟨output channel⟩::=CONS|LP|⟨numeric exp⟩ [56]
99. ⟨print format⟩::={:⟨numeric exp⟩ [56]
{:⟨numeric exp⟩}$_0^1$}$_0^1$ [56]

100. ⟨print exp⟩::=⟨numeric exp⟩|⟨scalar exp⟩ [56, 63]
|⟨string exp⟩|⟨Bool exp⟩ [64, 49]

If CONS is specified, output is sent to the user's console (i.e. standard BASIC PRINT), if LP, output is sent to the line printer, and if an expression with *n* as value, output is sent to file *n*. If the channel specification is omitted, #CONS: is assumed. When a PRINT statement is executed the contents of the print list are sent to the output device. As in BASIC the output medium is divided into zones of length *j* characters (*j* is implementation dependent—typically 15). Whenever a print separator is encountered in the print list, it is treated as follows:

if it is a semicolon, it has no effect
if it is a comma, skip to the next zone boundary

The function TAB(X) causes the print position to move forward to position Y where Y = X modulo the number of character positions per line of the output medium. If the print position is already beyond this position, the TAB function has no effect. The function NL(X) causes the output to move X lines vertically. If the print list ends on a separator, the line will not be printed and the next PRINT statement will continue output on the same line; otherwise the line will be printed and the output medium moved to a new line. If any item in a print list overflows over the end of a line, the line is printed and the remainder of the print list is continued on the following line.

For example

```
10 PRINT TAB(10);"RESULTS";NL(1);TAB(10);
                          "-------",NL(2);TAB(10);
20 PRINT "NO","NAME"
```

will cause

```
VVVVVVVVVRESULTS
VVVVVVVVVV-------
VVVVVVVVVNOVVVNAME
```

to be output (where each occurrence of the symbol V is used to denote a space in the final output). If an expression or array name is used as a print item without a print format, the system decides on the best format to use to print the value(s)[3]. If an expression (or array name) is followed by '*:n*' where *n* is a numeric expression, then the value of the expression (values in the array) is printed so that it occupies *n* print positions (right justified). If it is too long to fit into *n* positions the full number is printed despite the number of positions required. If a numeric expression *e* is followed by '*:n:m*' where *n* and *m* are numeric expressions, then the value of *e* is printed so that it occupies *n* print positions and has *m* digits to the right of the decimal point.

Note that just as pointer values cannot be read, likewise they cannot be printed. Thus it is not permissible to print the contents of a pointer variable, or a record containing a field of type pointer, or an array of pointer values.

The clause USING BINARY may be used with files. If this clause is used in a PRINT statement, numeric items are transmitted in binary instead of being converted to decimal, scalar items are transmitted as binary numbers representing their position in the list of scalar constants, etc. If the clause is used in a READ statement the system will expect binary values.

The file statement establishes a relationship between a particular channel number and a file. It has the form

101. ⟨file statement⟩::=FILE#⟨numeric exp⟩: [56]
⟨string exp⟩{,⟨mode⟩}$_0^1$ [64, 102]
102. ⟨mode⟩::=READ|WRITE|READWRITE

When a file statement is executed the numeric expression is evaluated and rounded to the nearest integer to give the channel number. If a file is already assigned to that channel, this file is closed. The string expression is evaluated to give the new file name. This file is opened and assigned to that channel. The mode specifies the mode in which the file must be opened—it may be omitted if the file is to be opened for reading.

The reset statement restores the data pointer to the beginning of the first DATA block or to the beginning of a file (corresponds to rewinding a magnetic tape file). It has the form

103. ⟨reset statement⟩::=RESET{⟨numeric exp⟩}$_0^1$ [56]

Without the numeric expression this statement resets the DATA block pointer, with it it resets the file pointer for the appropriate file.

The Boolean function EOF(X) indicates whether or not the file on channel X is currently positioned at the end of the file.

Control statements. The control statements available include the following

104. ⟨control stm⟩::=⟨conditional statement⟩| [105]
⟨loop statement⟩|⟨multi-way branch⟩| [110, 119]
⟨exit statement⟩|⟨ stop statement⟩ [126, 125]

A conditional statement may be part of an IF–THEN construct or part of an IF–THEN–ELSE construct. There are two forms of IF–THEN construct, a single line IF–THEN statement having the format

IF ⟨Bool exp⟩ THEN ⟨simple statement⟩ ENDIF

and a multi-line construct of form

IF ⟨Bool exp⟩ THEN {⟨simple statement⟩}$_0^1$
{⟨stm⟩}$_0^\infty$
ENDIF

where the IF part, the ENDIF part and each intervening statement besides the simple statement appears on a separate line with a separate statement number (the IF part having the lowest statement number of the sequence and the ENDIF part the highest). An IF–THEN–ELSE construct also has two forms, a single line ELSE part:

IF ⟨Bool exp⟩ THEN {⟨simple statement⟩}$_0^1$
{⟨stm⟩}$_0^\infty$
ELSE ⟨simple statement⟩ ENDIF

and a multi-line ELSE part:

IF ⟨Bool exp⟩ THEN {⟨simple statement⟩}$_0^1$
{⟨stm⟩}$_0^\infty$
ELSE {⟨simple statement⟩}$_0^1$
{⟨stm⟩}$_0^\infty$
ENDIF

where the IF part, ELSE part, ENDIF part and each intervening statement besides the two simple statements appears on a separate line with a separate statement number.

105. ⟨conditional statement⟩::=⟨if stm⟩| [106]
⟨else stm⟩|⟨endif stm⟩ [107, 108]
106. ⟨if stm⟩::=IF⟨Bool exp⟩THEN [49]
$\{$⟨simple statement⟩$\{$ENDIF$\}_0^1\}_0^1$ [109]
107. ⟨else stm⟩::=ELSE$\{$⟨simple statement⟩
$\{$ENDIF$\}_0^1\}_0^1$ [110]
108. ⟨endif stm⟩::=ENDIF
109. ⟨simple statement⟩::=⟨assign stm⟩| [75]
⟨read statement⟩|⟨print statement⟩| [90, 94]
⟨file statement⟩|⟨reset statement⟩| [101, 103]
⟨return statement⟩|⟨call statement⟩| [137, 135]
⟨stop statement⟩|⟨exit statement⟩|⟨dim stm⟩| [125, 126, 39]
⟨if stm⟩|⟨on statement⟩|⟨while statement⟩| [106, 120, 111]
⟨repeat statement⟩|⟨for statement⟩ [113, 115]

If the Boolean expression (⟨Bool exp⟩) is true, the sequence of statements following the symbol THEN is executed; if it is false then either the statement sequence following the symbol ELSE is executed (if there is an ELSE part), or control is passed to the first statement after the ENDIF symbol.

There are three formats for loops: a while loop, a repeat loop and a for–next loop. The while loop has the form

WHILE ⟨Bool exp⟩ DO ⟨loop id⟩
$\{$⟨stm⟩$\}_0^\infty$
ENDWHILE ⟨loop id⟩

where the WHILE part, the ENDWHILE part and each intervening statement is placed on a separate line with its own statement number. The effect of this statement is to repeatedly execute the sequence of statements until the Boolean expression is found to be false. If its value is false initially, the statement sequence is not executed at all. The loop identifier may be any plain identifier or the null string and serves mainly to identify the two parts of the loop. If a plain identifier is used, an inner loop may not use as its loop identifier the same identifier as is used by an enclosing loop.

The repeat loop has the form

REPEAT ⟨loop id⟩
$\{$⟨stm⟩$\}_0^\infty$
UNTIL ⟨Bool exp⟩ FOR ⟨loop id⟩

where the REPEAT part, the UNTIL part and each intervening statement occurs on a separate line with its own statement number. In this form of loop the statement sequence forming the body of the loop is executed at least once. On reaching the UNTIL part, the Boolean expression is evaluated and if it is false, the body of the loop is executed once more. This pattern is repeated until on testing the Boolean expression it is found to be true. As for the while loop, the loop identifier may be null, in which case the keyword FOR may be omitted.

The for–next loop causes a set of statements to be executed a number of times while a sequence of values is assigned to a variable (known as the *control variable*). It has two forms. A numeric for–next loop has the format

FOR ⟨numeric var⟩=⟨numeric exp⟩
⟨to-downto⟩⟨numeric exp⟩$\{$STEP⟨increment⟩$\}_0^1$
$\{$⟨stm⟩$\}_0^\infty$
NEXT⟨numeric var⟩

where the FOR part, the NEXT part and the intervening statements all occur on separate lines with their own statement numbers. A loop of form

FOR v = exp_1 TO exp_2 STEP exp_3
S
NEXT v

will be interpreted as

LET v = exp_1
LET *tempfin* = exp_2
LET *tempstep* = exp_3
WHILE (v − *tempfin*)**tempstep* ⟨0 DO *LOOP1*
S
LET v = v + *tempstep*
ENDWHILE *LOOP1*

Since the representation of decimal fractions in binary machines is necessarily approximate, the loop will operate within the limits of such arithmetic approximations in the case of a REAL controlled variable. If the STEP clause is omitted, an increment of 1 is assumed if TO is used, and −1 if DOWNTO is used.

A scalar for–next loop has the form

FOR⟨scalar var⟩=⟨scalar exp⟩⟨to-downto⟩⟨scalar exp⟩
$\{$⟨stm⟩$\}_0^\infty$
NEXT⟨scalar var⟩

The loop

FOR v = sexp_1 TO sexp_2
S
NEXT v

will be interpreted as

LET v = sexp_1
LET *temp* = sexp_2
WHILE v < = *temp* DO *LOOP1*
S
LET v = SUCC(v)
ENDWHILE *LOOP1*

Similarly

FOR v = sexp_1 DOWNTO sexp_2
S
NEXT v

will be interpreted as

LET v = sexp_1
LET *temp* = sexp_2
WHILE v > = *temp* DO *LOOP1*
S
LET v = PRED(v)
ENDWHILE *LOOP1*

Loops of all three types may be nested provided that the loop id (if there is one) or controlled variable used in an inner loop is different from any used in enclosing loops.

110. ⟨loop statement⟩::=⟨while statement⟩| [111]
⟨endwhile statement⟩|⟨repeat statement⟩| [112, 113]

⟨until statement⟩|⟨for statement⟩|
⟨next statement⟩ [114, 115, 116]

111. ⟨while statement⟩::=WHILE⟨Bool exp⟩ [49]
DO{⟨loop id⟩}$_0^1$ [117]

112. ⟨endwhile statement⟩::=ENDWHILE
{⟨loop id⟩}$_0^1$ [117]

113. ⟨repeat statement⟩::=REPEAT{⟨loop id⟩}$_0^1$ [117]

114. ⟨until statement⟩::=UNTIL⟨Bool exp⟩ [49]
{FOR⟨loop id⟩}$_0^1$ [117]

115. ⟨for statement⟩::=
FOR⟨numeric var⟩=⟨numeric exp⟩
⟨to-downto⟩ [56, 118]
⟨numeric exp⟩{STEP⟨numeric exp⟩}$_0^1$| [56, 56]
FOR⟨scalar var⟩=⟨scalar exp⟩ [63]
⟨to-downto⟩⟨scalar exp⟩ [108, 63]

116. ⟨next statement⟩::=NEXT⟨numeric var⟩| [71]
NEXT⟨scalar var⟩ [68]

117. ⟨loop id⟩::=⟨plain identifier⟩ [7]

118. ⟨to-downto⟩::= TO|DOWNTO

The multi-way branch has the form

$$\begin{array}{l} \text{ON}\langle\text{simple var}\rangle \\ \left\{\begin{array}{l} \text{CASE}\langle\text{expression}\rangle \\ \quad \{..\langle\text{expression}\rangle\}_0^1\{,\langle\text{expression}\rangle \\ \quad \{..\langle\text{expression}\rangle\}_0^1\}_0^\infty:\langle\text{simple statement}\rangle \\ \{\langle\text{stm}\rangle\}_0^\infty \end{array}\right\}_1^\infty \\ \left\{\begin{array}{l} \text{DEFAULT}:\langle\text{simple statement}\rangle \\ \{\langle\text{stm}\rangle\}_0^\infty \end{array}\right\}_0^1 \\ \text{ENDON}\langle\text{simple var}\rangle \end{array}$$

where the ON Part, each CASE part, the DEFAULT part (if one is present), the ENDON part and each intervening statement besides the simple statements shown occur on separate lines, each with its own statement number. The simple var may be of type REAL, INTEGER, string, subrange or scalar. If it is a numeric variable (REAL, INTEGER or subrange), the expressions must all be numeric, if it is a string variable, the expressions must all be strings and if it is a scalar variable, the expressions must be scalar constants.

The effect of the multi-way branch is to evaluate each case expression (⟨expression⟩) and each case range (⟨expression⟩..⟨expression⟩) in turn until it finds a case expression whose value matches the value of the ⟨simple var⟩, or a case range such that the value of the ⟨simple var⟩ lies within the range denoted by the two expressions. If such a case is found, control is passed to the sequence of statements immediately following the case expressions; if the value of the variable does not match any of the cases, control is passed to the default section if one is present or otherwise to the statement after the ENDON. If control is passed to one of the case alternatives, execution continues until the next CASE, DEFAULT or ENDON is encountered, at which point control is transferred to the statement after the ENDON.

As with the loops the simple var in the ENDON statement must match that in the ON statement.

119. ⟨multi-way branch⟩::=⟨on statement⟩| [120]
⟨case statement⟩|⟨default statement⟩|[121, 122]
⟨endon statement⟩ [123]

120. ⟨on statement⟩::=ON⟨simple var⟩ [124]

121. ⟨case statement⟩::=CASE⟨expression⟩ [48]
{..⟨expression⟩}$_0^1${,⟨expression⟩ [48, 48]
{..⟨expression⟩}$_0^1$}$_0^\infty$:⟨simple statement⟩ [48, 109]

122. ⟨default statement⟩::=DEFAULT:
⟨simple statement⟩ [109]

123. ⟨endon statement⟩::=ENDON⟨simple var⟩ [124]

124. ⟨simple var⟩::=⟨identifier⟩ [11]

The STOP statement causes execution of a program to cease.

125. ⟨stop statement⟩::=STOP

There are two forms of exit statement, viz.

126. ⟨exit statement⟩::=EXITL{⟨loop id⟩}$_0^1$| [117]
EXITP{⟨proc or fn name⟩}$_0^1$ [127]

127. ⟨proc or fn name⟩::=⟨proc name⟩|⟨fn name⟩ [132, 72]

The first form (EXITL) causes an exit from the statically enclosing loop specified to the statement immediately following that loop. If the loop id is omitted, the innermost enclosing loop with a null loop id is exited. The second form (EXITP) is used to return to the most recent invocation of the procedure or function specified, unstacking the details of any procedures or functions encountered in the interim. This statement assumes that the procedure or function concerned has been entered but not yet returned from at the time—if not an error will be flagged on execution of the statement. If no procedure or function name is specified the system will return to the main program.

REM statement. As in BASIC the REM statement is defined as

128. ⟨rem stm⟩::=REM{⟨character⟩}$_0^\infty$ [129]
129. ⟨character⟩::=⟨non-quote character⟩|"

It is used to insert comments into a program.

2.9 Procedures and functions

In BPL one may define procedures and functions. Statements related to procedures and functions are:

130. ⟨proc or fn stm⟩::=⟨call statement⟩| [135]
⟨return statement⟩|⟨procedure statement⟩| [137, 131]
⟨function statement⟩|⟨endproc statement⟩| [138, 134]
⟨endfn statement⟩ [139]

A procedure definition has the form

⟨procedure statement⟩
{⟨variable declaration⟩}$_0^\infty$
{⟨stm⟩}$_0^\infty$
⟨endproc statement⟩

where

131. ⟨procedure statement⟩::=PROCEDURE
⟨proc name⟩ [132]
{(⟨formal par⟩{:⟨type⟩}$_0^1$ [133, 31]
{,⟨formal par⟩{:⟨type⟩}$_0^1$}$_0^\infty$)}$_0^1$ [133, 31]

132. ⟨proc name⟩::=⟨plain identifier⟩ [7]

133. ⟨formal par⟩::=⟨identifier⟩ [11]
134. ⟨endproc statement⟩::=ENDPROC

and the procedure heading, the intervening variable declarations and statements, and the endproc statement all occur on separate lines with their own statement numbers.

A procedure is called by a statement of form

135. ⟨call statement⟩::=⟨proc name⟩ [132]
{(⟨actual par⟩{,⟨actual par⟩}$_0^\infty$)}$_0^1$ [136, 136]
136. ⟨actual par⟩::=⟨expression⟩ [48]

A formal parameter may be a plain or string identifier, depending on its type. If the type part is omitted, the parameter is assumed to be variable length string if it has a string identifier, or real if it has a plain identifier. When the procedure is called, the actual parameters must correspond in number, type and order to the formal parameters in the procedure definition. If an actual parameter is a single variable or an array name, the address of this variable is passed across to the formal parameter; otherwise the actual parameter is taken as an expression, is evaluated and a pointer to this value associated with the formal parameter. After assigning actual parameters to formal parameters, the statements in the body of the procedure are executed until a return statement of form

137. ⟨return statement⟩::=RETURN

or the end of the procedure (ENDPROC) is reached. In either case control is returned to the statement after the call in the calling segment (A segment being taken to be a procedure, a function or the main body of the program).

A function definition is very similar and has the form

⟨function statement⟩
{⟨variable declaration⟩}$_0^\infty$
{⟨stm⟩}$_0^\infty$
⟨endfn statement⟩

where

138. ⟨function statement⟩::=FUNCTION
⟨fn name⟩({⟨formal par⟩{:⟨type⟩}$_0^1$ [72, 133, 31]
{,⟨formal par⟩{:⟨type⟩}$_0^1$}$_0^\infty$}$_0^1$){OF ⟨type⟩}$_0^1$ [133, 31, 31]
139. ⟨endfn statement⟩::=ENDFN

Within the body of the function definition, the function identifier may be used (without parentheses following it) as a destination, but any other access to the function identifier implies a recursive call of the function and must be so written. Within the function body there must be at least one statement which assigns a value to the function identifier, and at least one such statement must be executed at every entry of the function. The final value assigned to the function identifier is the value returned by the function. A function name may be a plain or string identifier depending on its type. As with formal parameters, if the 'OF⟨type⟩' clause is omitted, the function is taken to be real or variable length string depending on the function name. A function call has the form

⟨fn name⟩({⟨actual par⟩{,⟨actual par⟩}$_0^\infty$}$_0^1$)

(see production 62) and may be used in an expression in the usual way. As with a procedure, control is returned to the calling segment on encountering a RETURN statement or the ENDFN statement.

Variables and arrays may be classified into two types: local or global. Any variable or array which is declared in a variable declaration or DIM statement in a procedure or function is local to that procedure or function and is accessible only within that procedure or function segment. The name of such a variable or array must be different from that of any other variable or array within that segment. Any other variables, i.e. those declared in variable declarations in the main body of the program and those allocated automatically anywhere in a program are taken to be global variables and are accessible throughout the main body and in any procedure or function in which the same identifier is not used as a local variable. Once again the names of such variables or arrays must be different from those of any other global variables or arrays or type identifiers.

Procedures and functions may be called recursively. Each procedure and function has its own set of statement numbers which are independent of statement numbers occurring in any other procedure or function or in the main body of the program. In order to handle input and editing of procedures, two modes of operation are used: (a) normal mode—in which all indirect statements are stored as part of the main program; (b) procedure mode—in which all indirect statements are stored as part of the current procedure or function.

Initially the system is in normal mode. If a PROCEDURE or FUNCTION statement is encountered then, provided that the system is not already in procedure mode (for if it is, an error will be flagged), the mode is switched to procedure mode. In procedure mode indirect statements of the procedure body are entered. These may be entered in any order provided that the PROCEDURE or FUNCTION statement has the lowest line number and the ENDPROC or ENDFN statement has the highest. One remains in procedure mode until the ENDPROC or ENDFN statement is encountered whereafter the mode reverts to normal mode.

If one wishes to return from normal mode to a procedure which has already been entered in order to alter it, this is done by means of the command:

140. ⟨editfn command⟩::=EDITFN⟨proc or fn name⟩ [127]

This causes the system to be placed in procedure mode and the appropriate procedure is once again accessible. One may extract oneself from procedure mode once again by giving the appropriate ENDPROC or ENDFN statement, this time either as a direct statement or as an indirect one.

Note that DATA statements may not occur in a procedure or function body—they may only occur in the main body.

2.10 Commands

The standard commands include

141. ⟨command⟩::=⟨editfn command⟩| [140]
⟨new command⟩|⟨old command⟩| [142, 144]
⟨save command⟩|⟨replace command⟩| [146, 147]
⟨run command⟩| [148]
⟨bye command⟩|⟨resequence command⟩| [149, 150]

⟨list command⟩| [155]
⟨trace command⟩|⟨break command⟩| [156, 158]
⟨vtrace command⟩| [166]
⟨untrace command⟩|⟨unbreak command⟩| [157, 159]
⟨unvtrace command⟩| [167]
⟨compile command⟩|⟨continue command⟩| [170, 163]
⟨mode command⟩| [171]
⟨last command⟩|⟨unlast command⟩ [164, 165]

142. ⟨new command⟩::=NEW⟨prog name⟩ [143]
143. ⟨prog name⟩::=⟨plain identifier⟩ [7]

NEW causes the current program to be erased from main memory in preparation for input of a new program.

144. ⟨old command⟩::=OLD⟨prog name⟩ [143]
$\{,\langle\text{file name}\rangle\}_0^1$ [145]
145. ⟨file name⟩::=⟨plain identifier⟩ [7]

OLD erases the current program from main memory and fetches the program specified by ⟨prog name⟩ from the file ⟨file name⟩ (or if no file name is given then the file currently open is used).

146. ⟨save command⟩::=SAVE $\{\langle\text{file name}\rangle\}_0^1$ [145]

SAVE stores the current program in a subfile of a file. The name of the subfile into which the program is stored is that given in the most recent NEW or OLD command. The file used is either the one specified in the SAVE command, or if none is specified, the file currently open.

147. ⟨replace command⟩::=REPLACE $\{\langle\text{file name}\rangle\}_0^1$ [145]

REPLACE causes the current program to replace the contents of an existing subfile. The name of the subfile is that given in the latest NEW or OLD command; the file used is the one specified (or the one currently open).

148. ⟨run command⟩::=RUN

RUN causes the current program to be interpreted.

149. ⟨bye command⟩::=BYE

BYE causes an exit from the BPL system.

150. ⟨resequence command⟩::=
RESEQUENCE$\{\langle\text{proc fn or all}\rangle\}_0^1$ [151]
$\{\langle\text{old first stm no}\rangle\{,\langle\text{new first stm no}\rangle$ [152, 153]
$\{,\langle\text{stm step}\rangle\}_0^1\}_0^1\}_0^1$ [154]
151. ⟨proc fn or all⟩::=⟨proc or fn name⟩|ALL [127]
152. ⟨old first stm no⟩::=⟨statement no⟩ [2]
153. ⟨new first stm no⟩::=⟨statement no⟩ [2]
154. ⟨stm step⟩::=⟨integer⟩ [14]

RESEQUENCE changes the value of some or all of the statement numbers of a program segment. If a procedure or function name is specified the statement numbers of that segment are resequenced; if ALL is specified all segments are resequenced; if neither a procedure or function name nor ALL is specified the statement numbers of the main program are resequenced. Resequencing starts at the old first statement number or if no such statement number exists in the specified segment, the next highest statement number is taken as the starting point. If omitted, the old first statement number is taken as one.

The new first statement number specifies the new value of the statement number of the first line to be resequenced (if omitted this defaults to 10). The statement step specifies the increment in value between successive new statement numbers (if omitted this defaults to 10). Resequencing continues to the end of the segment.

155. ⟨list command⟩::=LIST⟨arg⟩ [160]
156. ⟨trace command⟩::=TRACE⟨arg⟩ [160]
157. ⟨untrace command⟩::=UNTRACE⟨arg⟩ [160]
158. ⟨break command⟩::=BREAK⟨arg⟩ [160]
159. ⟨unbreak command⟩::=UNBREAK⟨arg⟩ [160]
160. ⟨arg⟩::=$\{\langle\text{proc fn or all}\rangle\}_0^1$ [151]
$\{\langle\text{first line no}\rangle\{-\langle\text{last line no}\rangle\}_0^1\}_0^1$ [161, 162]
161. ⟨first line no⟩::=⟨statement no⟩ [2]
162. ⟨last line no⟩::=⟨statement no⟩ [2]

For each of the above commands, if a procedure or function name is specified, the command applies only to that segment; if ALL is specified, it applies to all segments; if neither, it applies only to the main program body. If a single statement number is specified, the command applies only to the program statement with this statement number in the appropriate segment(s); if a pair of statement numbers is specified, all statements with statement numbers in the inclusive range of this pair in the appropriate segment(s) are concerned; if no statement number is specified, the command applies to all statements of the appropriate segment(s).

LIST causes the specified statements to be listed, TRACE sets the trace bits on the appropriate statements to enable tracing,[4] UNTRACE unsets the trace bits. BREAK sets the break bits on the appropriate statements causing execution to stop whenever it reaches such a statement,[4] UNBREAK unsets the break bits. If the program has stopped at a break point, execution may be continued by typing the command CONTINUE, defined as

163. ⟨continue command⟩::=CONTINUE

Another debugging command which is used in conjunction with TRACE is

164. ⟨last command⟩::=LAST⟨integer⟩ [14]

Under normal operation the TRACE facility will give a continuous trace of the execution of the program. However, when the command LAST *n* is given, the trace lines are not printed out but stored on a circular list with capacity for *n* entries. On encountering an error or the end of the program, the most recent *n* entries of this TRACE are printed out. Similarly

165. ⟨unlast command⟩::=UNLAST

is used to switch off this facility.

166. ⟨vtrace command⟩::=VTRACE⟨varlist⟩ [168]
167. ⟨unvtrace command⟩::=UNVTRACE $\{\langle\text{varlist}\rangle\}_0^1$ [168]
168. ⟨varlist⟩::=⟨trace var⟩$\{,\langle\text{trace var}\rangle\}_0^\infty$ [169, 169]
169. ⟨trace var⟩::=⟨simple var⟩|⟨array name⟩ [124, 43]

VTRACE sets the trace bits on the specified variables and arrays. UNVTRACE unsets the trace bits; if no variable list is specified, it unsets the trace bits on all variables and arrays. If on execution a value is assigned to a variable or an element of an array (in an assignment statement, READ, FOR or NEXT) for which the trace

bit is set, the statement number, the name of the variable and its new value are printed out.

170. ⟨compile command⟩::=COMPILE ⟨prog name⟩{,⟨file name⟩}$_0^1$ [143, 145]

COMPILE causes the current program to be compiled and stored in the subfile specified under ⟨prog name⟩ in the specified file (or if no file is specified, the current file).

The form of the MODE command is given by:

171. ⟨mode command⟩::=MODE⟨declaration mode⟩ [172]

172. ⟨declaration mode⟩::=*DEF*|*LIST*|*NOLIST*

This command sets the mode (as described in Section 2.2) according to whether all variables must be declared, or variable declarations may be omitted and the variables automatically allocated and listed at the end of each segment, or variables are allocated automatically without such a listing.

3. EXAMPLES

Three examples of simple BPL programs are given in Figs 2–4. Figure 2 contains a simple program for playing a simulated game of craps. This is based on the example of Dwyer.[5] The function RND is used to generate random numbers—RND(−1) initializes the random number generator and generates a starting value (which should be different each time) and RND(0) returns the next random number (in the range 0 to 1, excluding 1). Figure 3 illustrates a simple file application—updating a master file with amendments contained in an amendment file, while Fig. 4 shows how a monkey puzzle sort may be implemented.

```
1    REM ***************************************************
2    REM **  SIMULATED CRAPS GAME AS DESCRIBED IN         **
3    REM **  CREATIVE COMPUTING,VOL.3,NO.6,PP.83-84(1977)**
4    REM ***************************************************
10   TYPE WINSTATUS=(WIN,LOSE,UNCERTAIN)
20   TYPE ROLLTYPE=2..12
30   VAR RESULT:WINSTATUS
40   VAR ROLL1,ROLL2:ROLLTYPE

10   FUNCTION ROLL( ) OF ROLLTYPE
20   LET ROLL=INT(6*RND(0)+1)+INT(6*RND(0)+1)
30   ENDFN

100  LET X=RND(-1)
110  PRINT"SIMULATED CRAPS GAME - YOU START WITH $10"
120  LET YOURMONEY=10
130  REPEAT
140     PRINT"HOW MUCH DO YOU WANT TO BET?"
150     READ#CONS:BET
160     LET ROLL1=ROLL( )
170     PRINT"ROLL IS";ROLL1
180     ON ROLL1
190        CASE 4,5,6,8,9,10:PRINT "YOUR POINT IS";ROLL1
200           LET RESULT=UNCERTAIN
210           WHILE RESULT=UNCERTAIN DO
220              LET ROLL2=ROLL( )
230              PRINT"NEXT ROLL IS";ROLL2
240              IF ROLL1=ROLL2 THEN LET RESULT=WIN
250              ELSE IF ROLL2=7 THEN LET RESULT=LOSE ENDIF
260              ENDIF
270           ENDWHILE
280        CASE 2,3,12:LET RESULT=LOSE
290        CASE 7,11: LET RESULT=WIN
300     ENDON ROLL1
310     IF RESULT=WIN THEN LET YOURMONEY=YOURMONEY+BET
320        PRINT"YOU WIN! YOU NOW HAVE $";YOURMONEY
330     ELSE LET YOURMONEY=YOURMONEY-BET
340        PRINT "TOUGH..YOU LOSE. YOU NOW HAVE $";YOURMONEY
350     ENDIF
360     PRINT"WANT TO PLAY AGAIN?"
370     READ#CONS:ANSWER$
380  UNTIL ANSWER$ <> "YES"
390  PRINT"YOU ENDED UP WITH $";YOURMONEY
400  END
```

Figure 2. BPL program for simulating the game of craps.

```
1    REM ***********************************************
2    REM ** PROGRAM TO READ IN FILE OF AMENDMENTS TO **
3    REM ** WAGES OR OVERTIME HOURS AND UPDATE A     **
4    REM ** MASTER FILE CONTAINING EMPLOYEE RECORDS  **
5    REM ***********************************************
10   TYPE AMENDT=(WAGE,OVERTIME)
20   TYPE EMPREC=RECORD EMPNO:INTEGER;NAME$:STRING(20);  &
                        OTIME;BASICPAY END
30   TYPE AMENDREC=RECORD NO:INTEGER;                    &
                  ON AMTYPE:AMENDT                       &
                  CASE WAGE:NEWPAY                       &
                  CASE OVERTIME:NEWOT                    &
                  ENDON                                  &
                  END
40   VAR EMP:EMPREC
50   VAR AMEND:AMENDREC
60   VAR NOERRYET:BOOLEAN
70   VAR AMENDFILE,MASTER,NEWMASTER:INTEGER
120  NOERRYET=TRUE
130  FILE#1:"AMENDFILE"
140  FILE#2:"MASTERFILE"
150  FILE#3:"NEWMASTER",WRITE
160  AMENDFILE=1
170  MASTER=2
180  NEWMASTER=3
190  READ#MASTER:USING BINARY:EMP
200  WHILE NOT EOF(AMENDFILE) AND NOERRYET DO LOOP1
210     READ#AMENDFILE:AMEND
220     WHILE EMP.EMPNO<>AMEND.NO AND NOT EOF(MASTER)DO
230        PRINT#NEWMASTER:USING BINARY:EMP
240        READ#MASTER:USING BINARY:EMP
250     ENDWHILE
260     IF EOF(MASTER) THEN
270        PRINT"INVALID EMPLOYEE NO";AMEND.NO
280        NOERRYET=FALSE
290     ELSE ON AMEND.AMTYPE
300          CASE WAGE: EMP.BASICPAY=AMEND.NEWPAY
310          CASE OVERTIME: EMP.OTIME=AMEND.NEWOT
320        ENDON
330     ENDIF
340  ENDWHILE LOOP1
350  REM IF STILL MASTER LEFT,COPY TO END OF FILE
360  WHILE NOT EOF(MASTER)DO
370     PRINT#NEWMASTER:USING BINARY:EMP
380     READ#MASTER:USING BINARY:EMP
390  ENDWHILE
400  REM WRITE LAST RECORD
410  PRINT#NEWMASTER:USING BINARY:EMP
420  END
```

Figure 3. Program for updating a master file with amendments contained in an amendment file.

```
10   REM ***********************************************
20   REM *  THIS PROGRAM READS IN N NUMBERS AND        *
30   REM *  SORTS THEM INTO ASCENDING ORDER USING A    *
40   REM *  MONKEY PUZZLE SORT                         *
50   REM ***********************************************
60
70   TYPE TREEREC=RECORD LLINK:POINTER TO TREEREC;
           INFO;  RLINK:POINTER TO TREEREC END
80   TYPE TREEPTR=POINTER TO TREEREC
90   VAR P,ROOT:TREEPTR

10   PROCEDURE TRAVERSE(Q:  TREEPTR)
20   REM THIS TRAVERSES A BINARY TREE IN POSTORDER
30   IF Q↑.LLINK<> NIL THEN TRAVERSE(Q↑.LLINK) ENDIF
40   PRINT Q↑.INFO
50   IF Q↑.RLINK <> NIL THEN TRAVERSE(Q↑.RLINK) ENDIF
60   ENDPROC

10   PROCEDURE INSERT(Q:TREEPTR,X)
20   REM THIS INSERTS AN ITEM X INTO A BINARY TREE
30   IF Q = NIL THEN P:=CREATE(TREEREC)
40   P↑.LLINK = NIL
50   P↑.RLINK = NIL
60   P↑.INFO = X
70   Q = P
80   ELSE IF X<Q↑.INFO THEN INSERT(Q↑.LLINK,X)
90   ELSE INSERT(Q↑.RLINK,X) ENDIF
100  ENDIF
110  ENDPROC

100  ROOT = NIL
110  READ N
120  FOR I = 1 TO N
130  READ NUMBER
140  INSERT(ROOT,NUMBER)
150  NEXT I
160  TRAVERSE(ROOT)
170  END
```

Figure 4. BPL program for performing a monkey puzzle sort.

An interpreter for BPL is currently being implemented in Pascal for an ICL 1900 computer. Work is also being done on a compiler for the language (also in Pascal).

Since the initial submission of this paper several other languages have appeared which have similar characteristics, e.g. COMAL. These languages have varying degrees of complexity, and slightly differing objectives. For example, the chief objective in the design of BPL is to provide a language which can be taught at two levels—at the lower level, a subset of BPL is very similar to a structured form of BASIC, at the higher level, the

complete language is an interactive language which is probably more powerful than Pascal.

In view of this common interest it might be useful to establish a working group to study developments in this area.

Acknowledgements

The author wishes to thank the referee for his helpful suggestions, and Marianne Collett, Peter Clayton, Gavin Hall and Grant Pote for their respective parts in implementing the language. The work was done at Rhodes University, Grahamstown, South Africa.

REFERENCES

1. J. A. N. Lee, The formal definition of the BASIC language. *The Computer Journal* **15**, 37–41 (1972).
2. J. R. R. Tolkien, *The Lord of the Rings, Part II: The Two Towers.* Methuen, Toronto (1971).
3. G. M. Bull, W. Freeman and S. J. Garland, *Specification for Standard BASIC.* NCC Publications, Manchester (1973).
4. G. M. Bull, Dynamic debugging in BASIC. *The Computer Journal* **15** (No. 1), 21–24 (1972).
5. T. A. Dwyer, The 8-hour wonder. *Creative Computing* **3** (No. 6), 78–85 (1977).

Received November 1978

APPENDIX 1

List of reserved words for BPL statements

AND	ELSE	FILE	OF	REPEAT	TRANSPOSE
ARRAY	END	FOR	ON	REM	TRUE
BINARY	ENDFN	FUNCTION	OR	RESET	TYPE
BOOLEAN	ENDIF	IF	POINTER	RETURN	UNTIL
CASE	ENDON	INTEGER	PRED	STEP	USING
CONS	ENDPROC	LET	PRINT	STOP	VAR
DATA	ENDWHILE	LP	PROCEDURE	STRING	WHILE
DEFAULT	EOF	NEXT	REAL	SUCC	WRITE
DIM	EXITL	NIL	READ	TAB	
DO	EXITP	NL	READWRITE	THEN	
DOWNTO	FALSE	NOT	RECORD	TO	

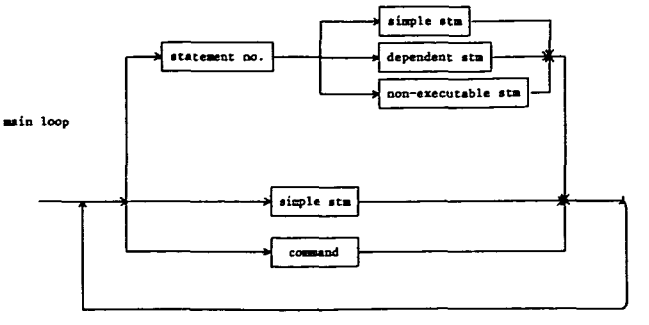
List of reserved words for commands

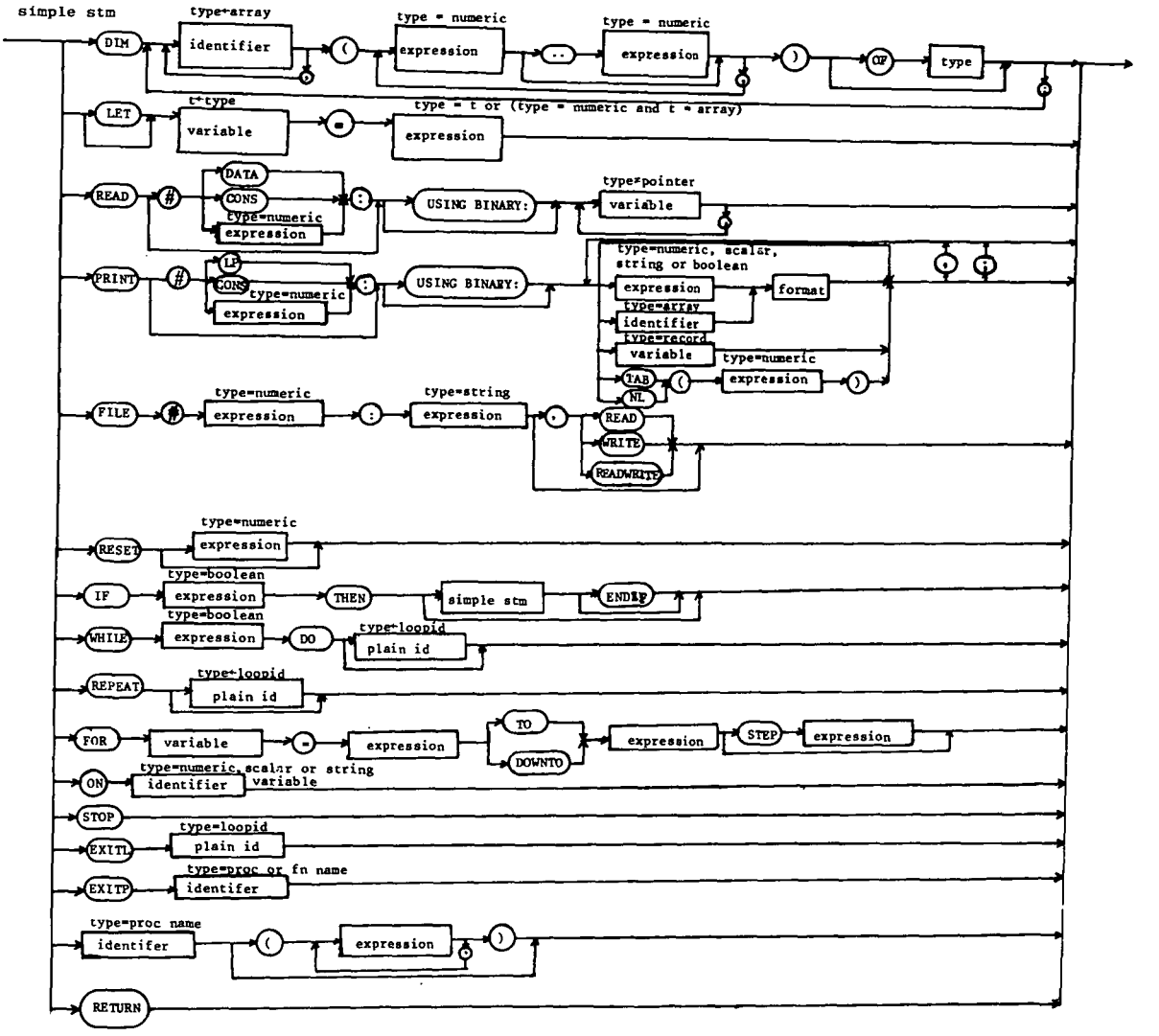
ALL	CONTINUE	LIST	OLD	SAVE	UNTRACE
BREAK	DEF	MODE	REPLACE	TRACE	UNVTRACE
BYE	EDITFN	NEW	RESEQUENCE	UNBREAK	VTRACE
COMPILE	LAST	NOLIST	RUN	UNLAST	

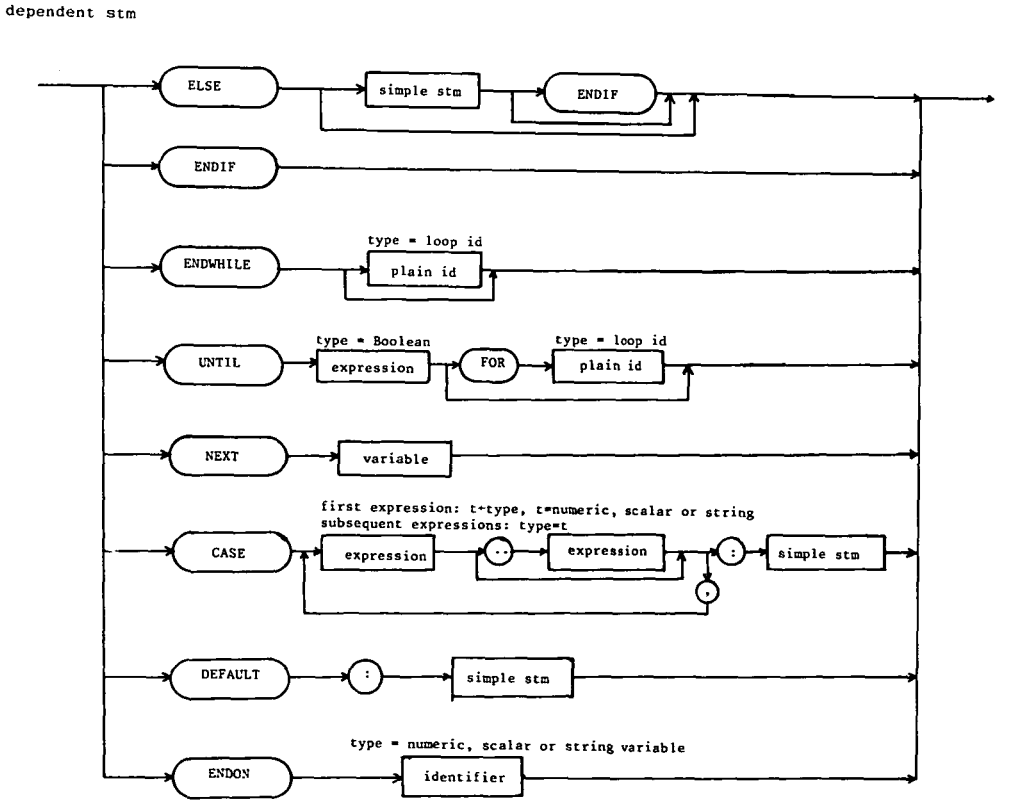
APPENDIX 2

Syntax diagrams for BPL

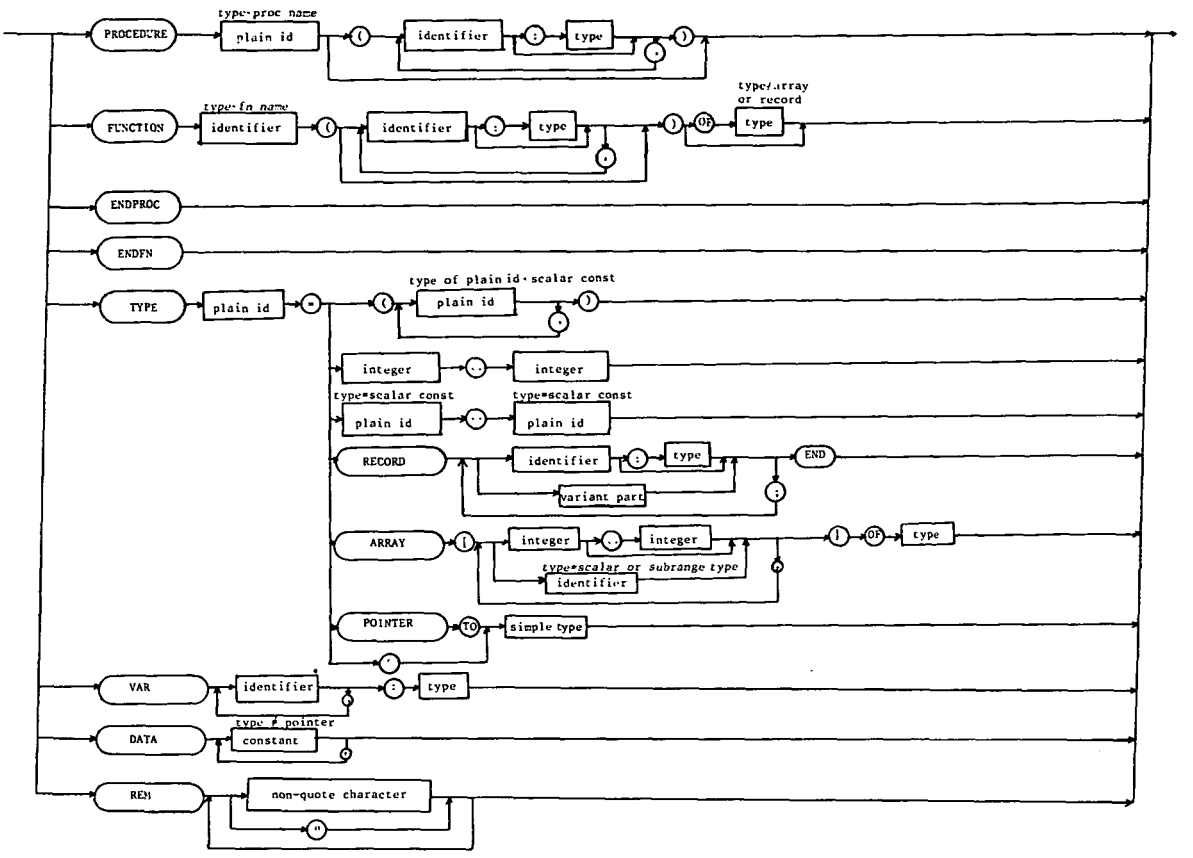
The comments regarding assignment of and checks on type which occur above some of the rectangular boxes are not exhaustive but do give some additional information which may assist the reader.



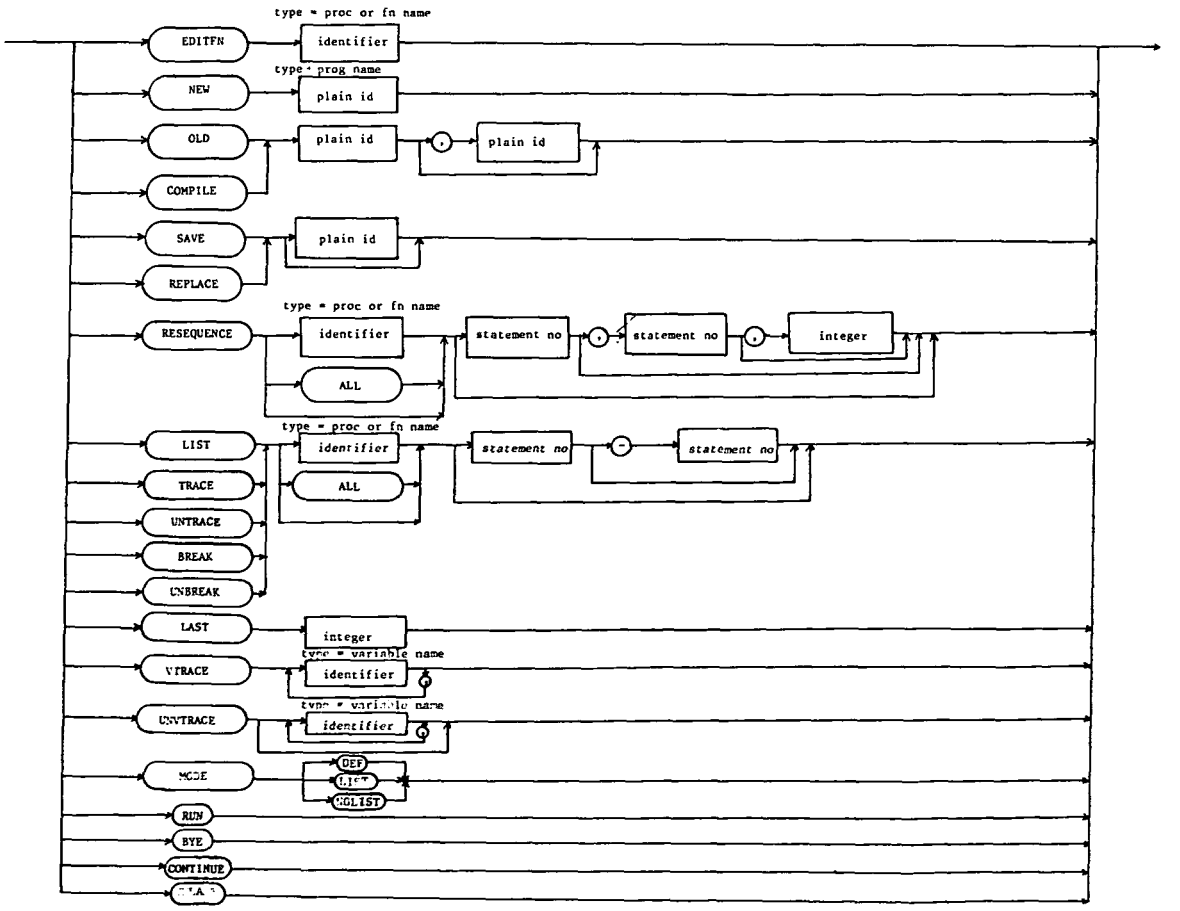


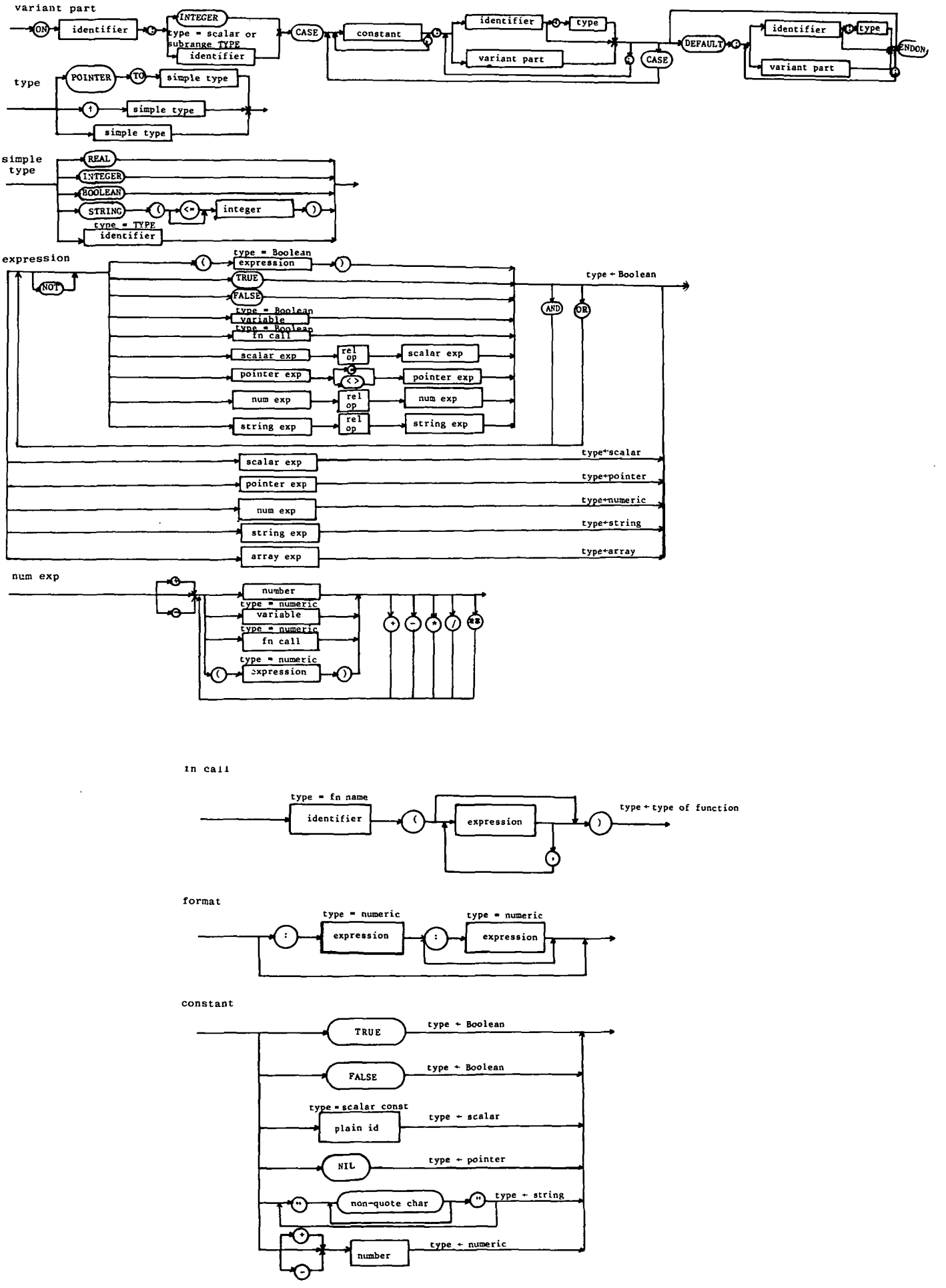


non-executable stm

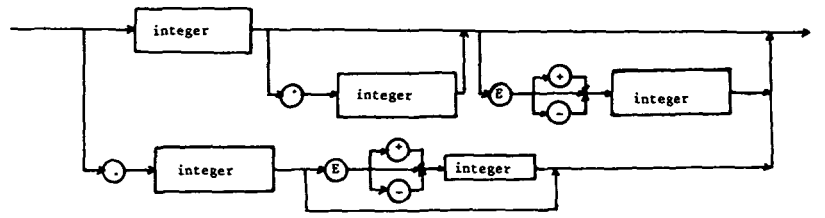


command

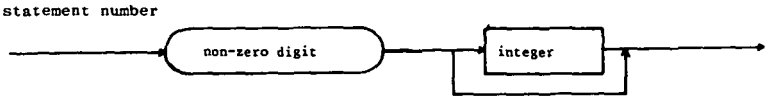




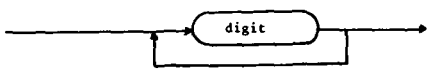
number



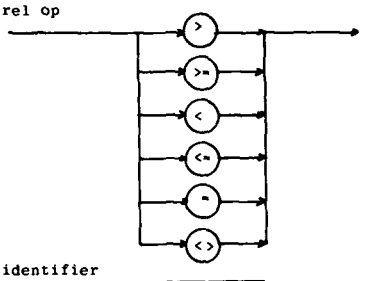
statement number



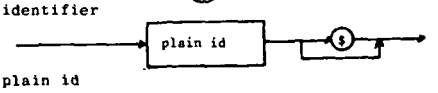
integer



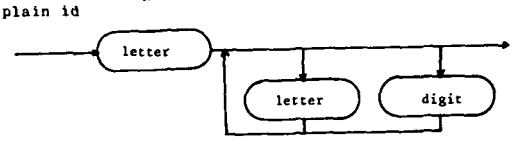
rel op



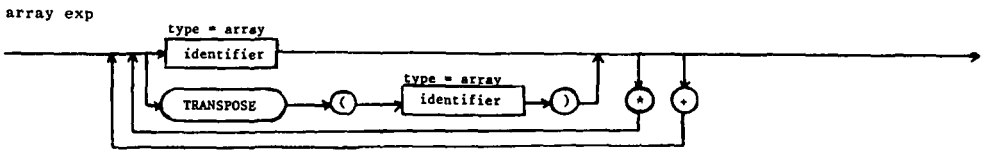
identifier



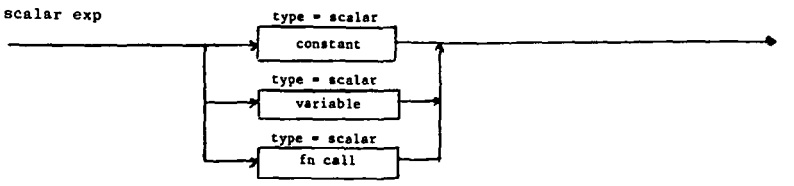
plain id



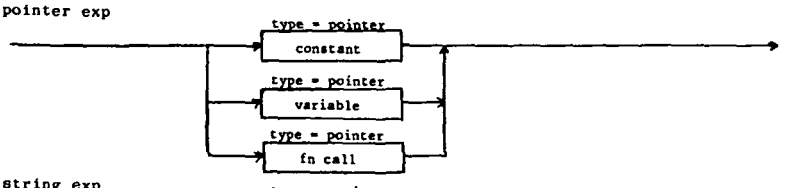
array exp



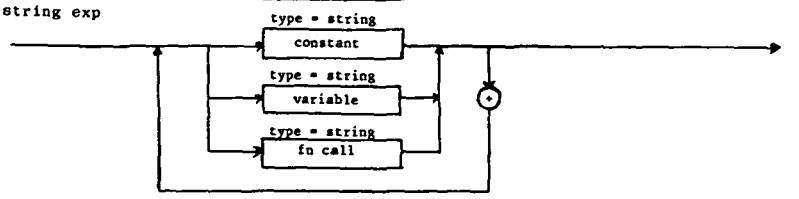
scalar exp

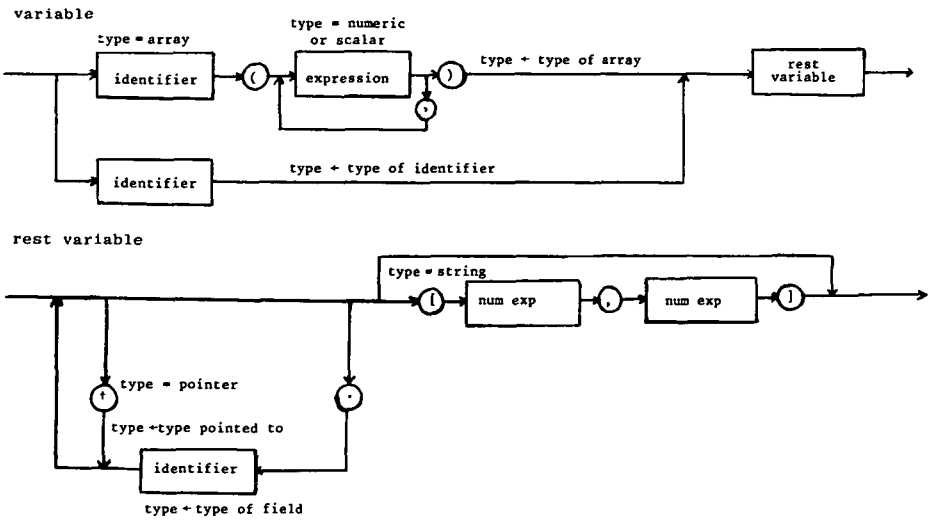


pointer exp



string exp





AN INVENTORY OF SOFTWARE PACKAGES FOR INFORMATION WORK

In early Fall 1981 an announcement was made that COSTI (National Center of Scientific and Technological Information) has contracted with UNESCO, for the preparation of an inventory of software packages written for information work. Developers and vendors of such packages were invited to register with COSTI, in order to be put on the mailing list to receive a detailed questionnaire.

The response to our announcement has been overwhelming. More than 150 packages written for mainframes, minicomputers and microprocessors are already on file and more letters are coming in daily. The questionnaire, authorized by an international panel set up by UNESCO, was sent to all respondents early in 1982.

Some questions naturally arise during this initial data collection phase. The inventory intends to describe all software packages which serve or may serve in textual and alphanumeric information work, library and documentation systems, SDI and online searches, information and fact storage, retrieval and distribution, text processing and publication, etc. In order to keep the inventory within reasonable limits, only systems which will be installed and/or be operational by December 1982 will be incorporated. Systems which are produced by non-commercial institutions are actively sought out. There is a special interest in packages which are potentially available and transferable to developing countries.

The inventory is scheduled for publication by the end of 1982. Considerable work is going into the preparation of a format to maximize its usefulness. There will be a number of tables which will enable the user to identify the packages which are closest to his specific requirements. There will be a concise summary for each package detailing its principal features, availability, costs, etc.

Any software producer or vendor who has not yet registered with COSTI is kindly invited to do so, always considering that this inventory will be a very useful tool in advancing the state-of-the-art of information work.

For more details, please contact the National Center of Scientific and Technological Information (COSTI), P.O. Box 20125, Tel-Aviv 61201, Israel, phone number 03–297781/292766, telex number 3–2332 CSTI IL.

In your initial reply, please refer to *UNESCO Inventory of Software Packages* and indicate name(s) of packages(s) to be submitted, with full mailing details.