PRELIMINARY

BON USER'S MANUAL

February 1, 1969

K. L. Thompson
as told to
M. D. McIlroy
R. Morris

1. Introduction

Bon is an interactive language. It uses concepts from several other languages, but it has a distinctive flavor of its own. Because elaborate computations can be performed with a small set of elementary constructs, Bon is a pleasant and quite interesting language to use. It is a new language, so comments will be welcomed.

2. Statements

2.1 Statement sequencing

Bon executes statements one at a time, either immediately from a typewriter or from a stored program, whose statements are numbered sequentially. A statement counter, denoted ".", remains at zero during execution of immediate statements. During execution of the stored program, the statement counter counts upward after each statement has been executed. Transfer of control may be effected by assigning to ".". Normal statement sequencing is interrupted by calling or returning from a function or when an execution fault is encountered. For example, the Bon statement

```
. := 0
```

will always place Bon in the immediate mode, and

```
. := .
```

will cause an infinite loop if it appears in the stored program but has no effect if immediate. (":=" is the assignment operator.)

2.2 Statement forms

Every Bon statement occupies one line. The simplest kind of statement consists of a single expression:

```
x + 3
a := 5
b := (a := a+1) + a
print(b + print(a))
```

The first statement, although legal, has no effect, since the value of an expression occurring as a statement is discarded. The second example is an ordinary assignment. The third example shows that an assignment is a legitimate component of an expression. The value of an assignment is the value of the expression on its right. The builtin function "print(...)" prints the value of its argument as a line of output and returns as its value the string value of its argument. The fourth example thus prints two lines of output, "6" and "12".

A statement may be prefixed with any number of identifiers, each follwed by colon:

```
joe: x := x + 3
```

Control may be transferred to the statement labeled by "joe" by the assignment

```
. := joe
```

Any statement may be suffixed with one of these clauses:

```
; while ...
; until ...
; if ...
; unless ...
; init ...
```

In each case "..." stands for an expression. A suffixed statement may have a suffix; thus any number of suffixes are permitted. "While" and "until" specify conditional repetition of the suffixed statement; "if" and "unless" specify conditional execution. Thus

```
i := 0
print(i := i+1); while i<10
```

prints the integers 1 to 10. The termination test is made before each repetition, including the first. The same thing could have been written in other ways.

```
i := 0
print(i := i+1); until i=10

print(i := i+1); while i<10; init i:=0
```

Suffix "init" specifies that the following expression is to be executed before execution of the suffixed statement. This facility makes possible quite elaborate computations in one line of code executed immediately. "Init" also makes it posssible to supply initial values after the body of a loop has been written.

3. Data

3.1 Data types

Quantities are manipulated in Bon (as in most languages) by referring to their names. Names commonly appear in programs as identifiers or as subscripted identifiers. It is also possible for the value of an expression to be a name. A value may be assigned to a name whether that name is given an identifier or is computed. Identifiers consist of upper case letters, lower case letters, digits, underlines, and overstruck characters in these four classes; an identifier may not begin with a digit.

The type of any variable is determined dynamically rather than by declaration. A variable has the type of the quantity that was last assigned to it. There are 7 types of data values:

```
arithmetic
pointer
label
string
boolean
null
builtin
```

Every assignment has the form

```
name := value
```

If the expression on the right is actually a name, then its current value is obtained before the assignment is made. A name, such as the expression

```
x
```

appear anywhere within an expression, but subexpressions

```
x := x
```

h has the same value, can only appear where a value is
. Moreover the former expression might make sense even
re a value had been assigned to x, while the latter two

would not.

Arithmetic type
Arithmetic variables are either fixed or floating numbers, but the distinction should seldom be of concern. The type of an arithmetic value can be forced by one of the built-in functions "fixed(...)" or "float(...)".

float w/a 3/15/69

Arithmetic constants, look like Fortran constants, except that integer-valued constants without decimal points are always fixed, and "+" is not allowed in exponents. Examples

1968	fixed
3.14159	float
10e3	fixed
10e-2	float

not true 3/15/69

[The variable "pi" starts out with a suitable initial value.

Pointer type
The value of a pointer variable is a name. Indirect addressing, i.e. extracting the value of "a" quantity named by a pointer is done by the unary operator "*". The inverse of this operator is the unary "&", which yields a value that is the name of its argument. Thus

```
a := &b
*a := 100
print(b)
```

prints "100". Indeed "*&a=a" is always true in Bon.

Label type
Labels refer to statements, and unlike statement numbers, they remain firmly attached even though other statements are inserted or deleted from programs. A variable may receive a label value by appearing as the prefix of a statement. The assignment is made at the time the statement is typed in. The quantity "." is always of type label. For example

```
      s := 0
       i := 1
loop: s := s + a[i]*b[i]
      i := i+1
      . := loop
```

computes a dot product, though rather clumsily. (Square brackets indicate subscripts.) "Loop" here is not a statement label in the traditional usage of other languages; rather it is an ordinary variable that happens to have a label for its present value. It would be perfectly reasonable later on to assign data of some other type to "loop". (Incidentally the loop would break out to call the typewriter when finally a nonexistent element was accessed.)

Labels also serve to denote the entry points of functions. Thus the expression

```
process(x)
```

would invoke the function whose first line of code was labeled by "process". The expressions

```
O(a,b,c)
f[i](x)
```

are equally valid. The first invokes a function, whose first line is to be read from the typewriter, with argument "a,b,c". The second picks the ith element from an array and invokes it as a function with argument "x".

<u>String type</u>
A string is an arbitrary sequence of ascii characters. String constants may be written by enclosing ascii text that does not contain newline characters in single quotes, or in double quotes:

```
print("Theta's more neatly typed 'θ'")
```

The identifiers "bs" and "nl" have string initial values, the ascii backspace and newline characters.

<u>Boolean type</u>
There are only two Boolean values. Initially Bon assigns one to the name "true" and the other to "false".

<u>Null type</u>
There is a single null value denoted "()". Its principal use is to supply empty arguments to function calls.

<u>Builtin type</u>
Builtin functions are defined by code internal to Bon. The names listed in Section 8 have builtin initial values.

<u>List type</u>
A list is a pair of values, or a pair of names. An element of a list may also be a list, so general binary tree structures may be constructed. A list is an expression, thus

```
(x := 3),(y := 5)
i,. := i+1,loop
```

are valid statements. ("," is the list operator)

3.2 <u>Arrays</u>

Arrays in Bon have a completely dynamic interpretation. The notation name[value]

stands for a distinct name, distinguished by a unique combination of governing name and subscript value (compared in the sense of "==", defined in 4.1). Thus "A[1]", "A[1,2]", "A[1][2]" are all distinguishable names in Bon, as are "table['cat']", "table['1']", "table[1]".

3.3 Type conversions

The following table defines all legal conversions between various types. Conversions may be caused by requirements on arguments of builtin functions or by requirements on operands of operators.

	fixed	float	label	string	boolean
fixed		(1)	(2)	(3)	(4)
float	(5)		(6)	(3)	(4)
pointer				(7)	
label	(8)	(6)		(9)	(6)
string	(10)	(10)	(6)		(6)
boolean	(11)	(6)	(6)	(12)	
null				(13)	
builtin					
list					

(1) Floating number of equal value.
(2) Label of line so numbered.
(3) Convert to equivalent constant, prefixed by "-" if necessary.
(4) Zero converts to false, all other values to true.
(5) Greatest integer not exceeding floating value.
(6) Convert via fixed.
(7) "&" prefixed either to the name of the quantity pointed to, or to the name of the array of which that quantity is an element.
(8) The integer line number of the labeled statement.
(9) The line, as originally typed, to which the label is attached.
(10) If the string is an arithmetic constant, perhaps prefixed by "-", then the result is the corresponding arithmetic value.
(11) "1 if true, 0 if false.
(12) "true" if true, "false" if false.
(13) "()"

4. Expressions

Expressions are made up of constants, of names, of unary and binary operators, and of two kinds of parentheses, "()" and "[]". Parentheses serve their usual role of grouping. Square parentheses cause the value of the enclosed expression to be taken. In addition, round parentheses surround the arguments of functions, and square parentheses surround the subscripts of arrays. The hierarchy of operators, in increasing order of binding strength is

:=	assignment
,	list formation
¦	or
&	and
= ^= ==	equality comparison
> >= ^> < <= ^<	order comparison
¦¦	concatenation
%	mod
+ -	addition and subtraction
/	division
*	multiplication
!	exponentiation
+ - * $ & ^	unary plus, unary minus, pointer indirection, string indirection, pointer creation, not

All binary operators associate from the left. Thus the following pairs are equivalent

```
a+b-c          ((a+b)-c)
a/b/c          ((a/b)/c)
a:=b:=c        ((a:=b):=c)
```

The last pair is illegal, however, because in it a value appears on the left of an assignment instead of a name.

Unary operators associate from the right, for example

```
-*x      means     -(*x)
```

All operators expect values for their operands except
":=" expects its left operand to be a name
"&" expects a name
"," accepts both operands as names, in which case the result is a name list. Otherwise the values of the operands are taken and the result is a value list.

4.1 Operations

<u>Assignment</u>
The operator ":=" interacts with the whole language, so its effect is described throughout this manual. In particular its syntactic behavior is described in section 2, its distribution over lists in section 4.2.

Assignment to "." during the execution of any statement suppresses the incrementation of "." that would ordinarily occur at the end of the statement.

<u>Arithmetic operations</u>
Binary arithmetic operators, "+", "-", "*", "/", and "!" all work in a mathematically reasonable way. Each (except "!")

demands operands of identical type, fixed or floating, and converts both to floating if they differ. The result is of the same type as the forced operands, except in the case of "/", which always yields a floating result. The unary operators "+" and "-" demand arithmetic arguments. The binary operator "!" accepts operands of mixed type; its result is fixed only if the first argument is fixed and the second argument is a non-negative fixed or floating integer.

The mod operation "%" is defined by

a % b   means   a - fixed(a/b)*b

where "fixed(...)" is the greatest integer function.

<u>Logical operations</u>
The operations "&", "|", and "^" convert operands to Boolean. They behave in the usual way.

<u>String operation</u>
Concatenation "||" converts both arguments to strings and gives the usual result.

<u>Equality comparisons</u>
The operators "=" and "^=" (not equal) work between pairs of values. If the values are of different type, both are converted to arithmetic type. The result of comparison is the expected Boolean value.

The operator "==" (identically equal) tests both operands for identity in type as well as value. It is unique in not distributing over "," (see <u>Lists</u> below).

<u>Order comparisons</u>
The operators ">", ">=" (greater than or equal to), "^>" (not greater than), "<", "<=", "^<" convert both arguments to arithmetic unless both are strings. Arithmetic comparison is done algebraically, string comparison lexicographically. The result is the expected Boolean.

<u>Indirection operators</u>
The operator "*" expects a pointer argument and returns that pointer as a name.

The operation "$" expects a string operand. The string is interpreted as an identifier, and the result is the corresponding name.

<u>Pointer creation operation</u>
The operator "&" expects a name as argument. It returns a pointer designating that name.

<u>List creation</u>

The operator "," between two expressions makes a list. If either operand is a value, both are made values, and the result is a value list. Otherwise the result is a name list. Comma like all other operators is left associative. That is

```
a,b,c       means       ((a,b),c)
```

4.2 Distributive laws

All unary operators and many builtin functions of one argument, when applied to a list, yield a list of results of elementwise applications of the operator:

```
-(1,2,3)       means       -1,-2,-3
```

Subscripts distribute similarly over a list of names:

```
(a,b)[1,2]       means       a[1,2],b[1,2]
```

If any binary operator (other than "==") appears between two lists, then the result is the same as a list of results of the operator appearing pairwise between elements of the two lists. As a consequence of the convention of left association, pairing in incompletely parenthesized lists is done from right to left. If one list is shorter than the other, its leftmost element is repeated enough to make up the difference. With one exception for ":=", this rule applies even in the degenerate case where one operand is not a list at all:

```
2*(a,b,c)       means       2*a,2*b,2*c
(1,2)-(a,b,c)       means       1-a,1-b,2-c
```

The operator "==" compares its operands for identity of type, dimension (number of elements), and value. It is the only operator that never distributes over ",".

The operator ":=" with a list on the right and a nonlist on the left causes the list to be assigned as a whole to the name on the left. In other respects the distribution rules for ":=" are regular.

All the elements of a list are evaluated before the list is used as a list of values, so that in particular

```
a,b := b,a
```

performs an interchange of two quantities. Even more exotic, the code

```
A[i],i,j := j,A[i],i; while i^=0; init i,j := 1,0
```

reverses the sense of a chain of links threading the vector A from A[1] until a zero link is encountered.

5. Creation of a Bon program

Program input is accepted in two modes, immediate and stored. Bon starts in the immediate mode in which every statement is executed as soon as it has been typed. The stored program is created by calling the builtin function

```
append(label)
```

This function causes following lines from the typewriter to be stored as program, just after the specified line. If the label is null, the the stored code is appended to the end of already stored text. The end of stored input mode is signaled by the appearance of a line consisting of "." alone.

For example, the following sequence starting in immediate mode stores a two line program and then transfers to it twice.

```
append()
start: print('no' || A || A)
. := 0
.
A,. := 'nse',start
A,. := ' no',start
```

The output of this fragment, which appears interleaved with the last two input lines, is

```
nonsense
no no no
```

Stored program may be modified by deletion as well as appending. For this purpose there is a function

```
delete(label)
```

One could use this function to wipe out the whole stored program this way:

```
delete(1); while true
```

To print a stored statement, use the form

```
print(label)
```

This works because label-to-string conversion yields the statement so labeled. Remember, though, that

```
print(2)
```

will print "2" To print line number 2, one must force a label argument. The builtin "label" does the trick:

```
print(label(2))
```

Bon can be terminated by typing

```
quit()
```

5.1 Entering Bon

In Multics Bon is entered by typing "bon" at command level. Bon acknowledges before actual typing of Bon statements may begin. Bon may be put to work on a prepared program by typing

```
bon pathname
```

The specified file behaves as if it were the typewriter. Once it reads beyond the end of the file, Bon reverts to the typwriter for further input.

5.2 Typing conventions

Standard erase and kill conventions hold:

(erase) causes the immediately preceding character position, or blank string to be deleted
@ (kill) causes all preceding characters on the line to be deleted.

On 1050 terminals, with the CTSS and Multics standard typewriter ball (963), all syntactic characters except "[]" are available. Escape conventions are

¢<	for	[
¢>	for	]

No escapes are required on model 37 teletypes. By convention, the acute accent serves as a single quote.

Use of "@" and "#" and escapes ("\" or "¢") in Bon text depends on system escape conventions, see Multics Console User's Guide.

5.1 Debugging

All errors encountered during execution are reported to the typewriter. After reporting, Bon gives control to the typewriter by invoking statement 0 as a function.

Syntax errors are discovered while statements are being input. All lines of input typed after a syntactically bad line but before the error report are ignored. If "append()" is in effect at the offending line, it remains in effect after the report.

Most diagnostics are reasonably self-explanatory. A few are worth noting:

- "Major cycle loop" means 1000 statements have been executed without returning to the typewriter. This guards against interminable loops. To go on for another 1000 statements, type "return()"

- "Minor cycle loop" means 1000 operations have been done within one statement. The interrupted statement can't be continued.

- "Push" and "Pop" occur when Bon's bookkeeping goes awry. These are temporary glitches, usually arising after other errors. If Bon will continue at all, you can be confident that no data has been messed up.

- "Label not current" means that control has gone to a line that does not exist. A retyped line, even with the same identifier, has a new label.

<u>Dumping</u>
The namelist, i.e. all the current identifiers, can be printed by executing

```
dump()
```

If the "full" option (see next paragraph) is on at the time of dumping, then the following information will be printed for each variable in addition to its name.

```
type
value, if any
"subscripted", if subscripted versions of this name exist
"associated", if the current name is associated
```

To dump selected variables, say "a,b,c", in full option execute dump with an argument:

```
dump(a,b,c)
```

<u>Execution modes</u>
There are several diagnostic modes, which can be switched on and off in alternation by executing

```
option(mode)
```

where "mode" is a string that names one of these modes:

space	cause "print" to use space instead of comma between elements of printed lists
full	cause full information about each identifier to be printed by "dump()"

The normal setting of all options is off.

6. Functions

Besides builtin functions, Bon allows functions defined by program. Labels mark the starting point of function executions, and "return(...)" marks the end. The extent of a function is thus determined by flow of control rather than syntactically as in Fortran or Algol.

Function invocation is indicated in the form

```
label(...)
```

where "..." is an expression or null. A previous example could have been written functionally this way

```
append()
start: print('no' || A || A), return()
.
(A := 'nse'), start()
(A := ' no'), start()
```

As before the interleaved output would be

```
nonsense
no no no
```

A function may return a value, which is specified as the argument of "return(...)". Functions may have parameters. Parameters (but not only parameters) are said to be <u>associated</u> with each level of function invocation. New associated names can be created by use of the builtin function

```
assoc(...)
```

where "..." is a name or a list of names. "Assoc(...)" pushes previous syntactically identical names down out of the way and creates new names. The function value of "assoc(...)" is the new name, or list of new names.

Upon return from a function level, its associated names disappear and the previous ones are restored.

The entire list of parameters can be accessed by means of

```
                param(0)
```

("Param(i)" for i>0 fetches the parameter, often a list, for the ith preceding function level.)

"Assoc(...)" is a convenient way to get working storage within a function. For example, a function which expects a parameter with three elements, could split the items out into three temporary variables named "a", "b", "c" by the simple line of code

```
                assoc(a,b,c) := param(0)
```

This works because the value of "assoc(a,b,c)" is a list of names, and the value of "param(0)" is a list. The trick is so useful that it has been defined as a builtin function

```
                entry(...)
```

Here is an entire program to transform polar to rectangular coordinates:

```
                rect: entry(r, θ)
                      return(r*cos(θ), r*sin(θ))
```

If a function needs temporary storage for quantities other than parameters, "assoc(...)" comes in handy. Here is a complete program to compute sin(x) by the Taylor's series

```
        sin(x) = x*(1 - (x*x/2*3)*(1 - (x*x/4*5)*(1 - ...
```

At first blush forbidding, such things turn out to be easy to write:

```
        sin: entry(x)
             assoc(s,t,n) := x,1
             s:=s+(t:=-t*x!2/(n+1)*(n:=n+2)); until s+t=s
             return([s])
```

Several points are worth noting:
- The expression "assoc(s,t,n):=x,1", which obtains and initializes working storage, means just the same as "assoc(s,t,n):=x,x,1".

- Partial sums are computed in "s", individual terms in "t". The repetition condition "until s+t=s" stops things when t gets so small that it no longer makes any difference. This formulation depends on floating point arithmetic, and is not equivalent to the nonsensical condition "until t=0".

- The unusual argument in "return([s])" is intended to force a value. This is necessary, because "return(...)" is quite happy to pass a name through unchanged. Since "s" is an associated variable, it is discarded upon return; to return its name would be to return a nonentity.

Most builtin functions of one argument, of which "assoc(...)" and "entry(...)" are examples, distribute over lists like unary operators. The following complex multiplication function makes significant use of this convention.

```
     cmult: entry((x,y),(u,v)), return(x*u-y*v,y*u+x*v)
```

Interesting things happen when functions return names. This example returns the name of the larger of a list of two elements:

```
                 max: assoc(x,y) := &param(0)
                      return(*x); if *x >= *y
                      return(*y)
```

The next statement uses "max(...)" to increment the larger of "a" and "b".

```
                 max(a,b) := max(a,b) + 1
```

Recursion is legitimate. The following function finds the maximum value in a tree structure. It depends on the builtin function "dimen(...)", which returns the number of elements in its parameter.

```
                 max: entry(x,y)
                      x := max(x); if dimen([x])>1
                      y := max(y); if dimen([y])>1
                      return([x]); if x>=y
                      return([y])
```

The curious arguments of return are once again caused by the necessity of forcing a value. A clever way to do both returns in one line, if the arguments are arithmetic, is

```
                 return((x>y)*x + (x<=y)*y)
```

The former formulation is slightly more general, however, because it works equally well on strings.

6.1 Useful programs

The function change(a,b) causes the stored lines labeled a through b to be deleted and accepts succeeding lines of input to be stored in their place. As usual, input is terminated by a line consisting of ".". "change(a)" causes a single line to be replaced.

```
                 change: assoc(x,a,b) := fixed(param(0))
                         delete(a),(x:=x+1); while x<=b
```

```
        append(a-1)
        return()
```

The following line creates abbreviations "p(...)", "a(...)", "d(...)" for three very common builtins

```
    p,a,d := print,append,delete
```

7. Official syntax

The following BNF syntax has been written as briefly as possible. Other syntactic matters are covered in the succeeding informal description.

```
statement ::= expression ¦ identifier : statement ¦
              statement ; suffix

suffix ::= while svalue ¦ until svalue ¦ if svalue ¦
           unless svalue ¦ init expression

expression ::= name ¦ value

name ::= . ¦ identifier ¦ name[value]¦ function ¦
           name , name ¦ $ value ¦ * value ¦ ( name )

value ::= constant ¦ name ¦ function ¦ value binop value ¦
           unop value ¦ & name ¦ ( value ) ¦
           name := value ¦ [ expression ]

function ::= svalue ( expression ) ¦ svalue ( )

constant ::= fixcon ¦ floatcon ¦ stringcon ¦ ( )

fixcon ::= integer ¦ integer e integer

floatcon ::= real ¦ real e integer ¦ real e - integer ¦
          integer e - integer

real ::= integer . ¦ integer . integer ¦ . integer

unop ::= + ¦ - ¦ ^

binop ::= λ ¦ & ¦ or ¦ = ¦ ^= ¦ == ¦ > ¦ >= ¦ ^> ¦ < ¦ <= ¦
           < ¦ cat ¦ % ¦ + ¦ - ¦ / ¦ * ¦ !
```

identifier is a nonempty string of possibly overstruck upper and lower case letters, digit and underline not beginning with a digit.

svalue is a value that is not of type list

while, until, if , unless, init, or, cat, e are the symbols "while", "until", "if", "unless", "init", "¦", "¦¦", "e"

integer is a nonempty string of digits

stringcon is an arbitrary string of characters excluding newline bounded left and right by single or double quotes. Both quotes must be the same and may not appear within the string.

Use of blanks An arbitrary number of blanks may appear adjacent to any syntactic category in the BNF, except that
- blanks may not appear within a fixcon or a floatcon,
- blanks must intervene between a suffix keyword (while, until, if, unless, init) and a following identifier or digit,
- blanks may intervene between "(and an immediately following ")".

8. Builtin functions

Each builtin demands an argument of the kind specified in the description. Arguments of other kinds are converted appropriately. "Result is argument" refers to converted value of argument.

8.1 Unary functions

All of these functions generalize to list arguments in the same way that unary operators do.

append(label) Store following packet of lines up to "." after designated Store at beginning of program if argument is 0. Store at end of program if argument is null.

assoc(name) Create an associated variable for the name. Result is new name.

boolean(boolean) Result is argument.

builtin(string) Result is builtin function of specifed name.

cos(float) Result is cosine of argument.

delete(label) Causes designated statement to be deleted from stored program. Null argument causes last line to be deleted. Result is null.

dump(name) Causes "full" dump of argument. If argument is not a name, causes dump of entire namelist. Result is argument.

entry(name) Same as expression "assoc(name) := param(0)".

exp(float) Result is exponential function of argument.

fixed(fixed) Result is argument.

float(float) Result is argument.

index(string,string) searches for the second string within the first string. Result is the character number of the beginning of the second string or 0 if the search fails.

label(label) Result is argument.

length(string) Result is fixed, number of characters in argument.

log(float) Result is natural log of argument.

pointer(pointer) Result is argument.

size(string) Result is width of printed representation of string.

option(string) Causes the sense of the named option (see 5.) to be reversed. Result is boolean denoting new setting, true for on, false for off.

sin(float) Result is sine of argument.

string(string) Result is argument.

type(expression) Result is string denoting type of argument. Values are "boolean", "builtin", "fixed", "float", "label", "null", "list", "pointer", "string", "undefined".

8.2 Binary builtins

atan(float,float) Result is arctan of two arguments. In first and fourth quadrants, atan(x,y) is the same as the customary arctan(y/x).

char(string,fixed) Result is a one-character string, consisting of the designated character, counting from 1 at the left, in the given string.

repeat(string,fixed) Result is string, n-fold concatenation of first argument.

8.3 Condensation builtins

These builtins create simple values as functions of structures of values.

all(boolean) Result is boolean and of all values in the structure.

any(boolean) Result is boolean or of all values in the structure.

cat(string) Result is concatenation of all values in the structure.

max(value) Result is maximum of all values in the structure.

min(value) Result is minimum of all values in the structure.

product(arithmetic) Result is product of all values in structure.

sum(arithmetic) Result is sum of all values in the structure

8.4 Special builtins

These builtins treat their single arguments as an entity for special actions. In general a null argument may appear wherever an expression is required.

dimen(expression) Null argument yields 0. Otherwise result is one plus the number of commas at the outer level of a minimally parenthesized representation of the argument. Does not take value of an argument that is a name or list of names.

print(expression) Result is same as that of "string(expression)". Prints the result on one line with commas and parentheses to denote its list structure.

quit(expression) Causes Bon to terminate execution.

random(expression) Result is float pseudorandom number uniformly distributed on the range zero-one.

return(expression) Causes current function level to be terminated together with its associated variables. Expression appears as function value in invoking level.

read(expression) Expression is ignored and result is a string which is the next line from the typewriter.

unique(expression) Result is a fixed integer, greater than all preceding results of "unique(...)". This integer is roughly proportional to running time up to now.