The *B* Programmer's Handbook

Leo Geurts Lambert Meertens Steven Pemberton

```
PUT expr IN target
WRITE expr
READ target EG expr
READ text RAW

INSERT expr IN list
REMOVE expr FROM list
DELETE target

DRAW number
CHOOSE target FROM tlt
SET'RANDOM expr

CHECK test
IF test: COMMANDS
SELECT:
    test: COMMANDS
    ...
    ELSE: COMMANDS
```

```
WHILE test: COMMANDS
FOR tag, ... IN tlt: COMMANDS

HOW'TO KEYWORD ...: COMMANDS
YIELD function operand: COMMANDS
TEST predicate operand: COMMANDS

QUIT
RETURN expr
REPORT test
SUCCEED
FAIL

SHARE tag, ...

REFINEMENT: COMMANDS
refinement: COMMANDS
```



```
1.23E-45
{expr; ...}
{expr..expr}
{[expr]: expr; ...}
"text `expr` ..."
table[expr]
text@number | number
```

```
~ + - * / ** mod abs
sign round floor ceiling
*/ /* root e exp log
pi sin cos tan atan

^ ^^ << >> ><
min max # th'of keys

= <> < <= > >= in not'in
```

```
AND OR NOT
```

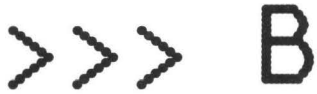
```
SOME/EACH/NO tag, ... IN expr HAS test
SOME/EACH/NO tag, ... PARSING text HAS test
```

For using *B* on:

- Unix systems
- IBM PC's and compatibles under DOS

The *B* Programmer's Handbook

Leo Geurts Lambert Meertens Steven Pemberton



For using *B* on:
- Unix systems
- IBM PC's and compatibles under DOS

Centrum voor Wiskunde en Informatica
(Centre for Mathematics and Computer Science)
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

ISBN 90 6196 295 1

CONTENTS

Preface

B is a simple but powerful new programming language designed at the CWI for use in personal programming.

This book is principally designed to accompany the implementation of *B*, and it consists of revised and updated versions of earlier documents. While it is not meant as a course, it should be useful to people who already have some programming experience and want to learn *B*.

Note that the name "*B*" is only a temporary working title, and the language bears no relationship to the predecessor of the language "C". *B* is currently being revised and when the revision process is complete the language will get its final name "ABC".

The other members of the *B* group have played a large role in making all this possible; they are Frank van Dijk, Timo Krijnen and Guido van Rossum.

A QUICK LOOK AT B

CONTENTS

1. INTRODUCTION

B is a powerful, easy-to-learn and easy-to-use interactive programming language, intended for personal computing and designed as a modern alternative to BASIC and an easy alternative to Pascal. It supports data structuring and structured programming. What started as a beginners' programming language turns out to be a fascinating tool for novices and experts alike. This chapter gives an informal overview of the language, meant for those who are already familiar with one or two other programming languages.

B is not simply a language, it is a language embedded in a dedicated system containing a syntax-directed screen editor, 'file' maintenance functions, etc. Programming in *B* has a strongly interactive feel. Rather than writing one long program, one develops a set of procedures, functions and data which are kept by the system until they are intentionally deleted.

2. AN EXAMPLE

```
YIELD gcd(a, b):
   CHECK (a, b) <> (0, 0)
   PUT abs a, abs b IN a, b
   WHILE b > 0:
      PUT b, a mod b IN a, b
      \mod gives the remainder of division
   RETURN a
```

Once this piece of *B* (the definition of a function delivering the greatest common divisor of two numbers) has been entered, the command

```
WRITE gcd(21, 28)
```

writes 7 on the screen. The body of this YIELD-unit is straightforward:

- PUT has an expression to the left of IN, and a *target* to the right. As we shall see later, this multiple PUT in fact puts one *compound* expression in one *compound* target. PUTting IN an operand of a function is allowed but does not affect the environment. It is impossible for a function (or any other expression) to alter its environment: it can only deliver a value.
- A CHECK keyword is followed by a test. If the test fails, an appropriate message is given and execution halts. In this case, the compound a, b should not be equal to 0, 0.
- A control command, such as WHILE, controls the indented commands following the corresponding colon. Thus, the user need not bother about pairs of BEGIN and END or other delimiters for grouping commands. For the same reason, the whole body of the YIELD has been indented. If there is only one controlled command, the whole construction may be displayed on a single line:

```
IF a < 0: PUT -a IN a
```

- The line starting with \ is a comment. It could have been appended to the

previous line.

In the *B* system a friendly editor takes care of the tedious aspects of typing the program, such as indentation and other layout matters, capitals, keywords, etc.

3. TYPES

The power of *B* is largely due to its carefully designed system of data types and associated operations. There are two basic types — *numbers* and *texts* —, and three structures creating new types from existing ones — *compounds*, *lists* and *tables*.

3.1. Numbers

Integers in *B* are conventional, except that there is no restriction on length. Furthermore, integers are just a special case of *exact*, i.e. rational, numbers (integral fractions). For example, `1.25` is an exact number. The monadic operators `*/` and `/*` yield the numerator and the denominator of an exact number. For this purpose, fractions are automatically reduced to lowest terms; so `1.25 = 125/100` is reduced to `5/4`. The following one-liner for the greatest common divisor of two exact positive numbers uses this property:

```
YIELD gcd(a, b): RETURN a / */(a/b)
```

In addition to exact numbers, there are *approximate* (floating point) numbers. The operator `~` is used for conversion to an approximate number: `~22/7` does not yield an exact, but an approximate number.

Operations:

plain arithmetic	`+x, x+y, -x, x-y, x*y, x/y`
to the power	`x**y`
round upwards	`ceiling x`
round downwards	`floor x`
round to nearest integer	`round x`
round to `n` digits after decimal point	`n round x`
sign	`sign x`
absolute value	`abs x`
square root	`root x`
`n`th root	`n root x`
remainder of division	`a mod n`
numerator of exact number	`*/ x`
denominator of exact number	`/* x`
natural logarithm	`log x`
logarithm to base `b`	`b log x`
exponential function	`exp x`
trigonometric functions	`sin x, cos x, tan x, atan x`

Mixed arithmetic is allowed, as in `2*sin(x-1)`.

3.2. Texts

`"Merry Christmas"` is an example of a text (*text* being a less esoteric term for *string*).

Operations:

join	`"now"^"here" = "nowhere"`
repeat	`"-"^^5 = "-----"`
tail	`"nowhere"@3 = "where"`
head	`"nowhere"|2 = "no"`
number of characters	`#"nowhere" = 7`
number of occurrences	`"e"#"nowhere" = 2`
selection	`4 th'of "nowhere" = "h"`
scan the characters	`FOR c IN "nowhere":` `  PUT freq[c]+1 IN freq[c]`

The trim operations `@` and `|` may be combined to yield any subtext. For example, to obtain the subtext starting at position 3 and having length 4, we may write:

```
"nowhere"@3|4 = "wher"
```

Such subtexts may also be used as targets. For instance, after

```
PUT "nowhere" IN word
PUT "bless" IN word@3|4
```

the target `word` contains `"noblesse"`.

3.3. Compounds

A compound is a sequence of values of possibly different types. It is like a record in other languages, but it has no field names. Compounds are useful for multiple `PUT`s, for parameters and operands (as in the first `gcd` example), for multi-dimensional tables (`t[i, j]`), but also as values to be put in a table (instead of separate values put in separate tables under the same key). Examples:

```
PUT a, b, c IN b, c, a \cyclic permutation
PUT "May", 22 IN anniversary
PUT 1813, anniversary IN date
PUT date IN year, (month, day)
```

3.4. Lists

A list is a sorted sequence of values with multiple occurrences allowed. All entries of a list must have the same type. For example,

```
WRITE {"u"; "e"; "a"; "y"; "o"; "i"}
```

gives

```
{"a"; "e"; "i"; "o"; "u"; "y"}
```

and

```
FOR i IN {"ay"; "i"; "eye"; "aye"}: WRITE #i
```

gives

```
2 3 3 1
```

The entries of a list may be of any type. This only works because an order is associated with each type, e.g.:

```
2.99 < 3.00
"az" < "b"
("b", "c") < ("z", "a")
{} < {"a"; "z"} < {"b"}
```

Instead of `{1; 2; 3; 4; 5; 6; 7}` the notation `{1..7}` may be used; similarly, `{"b"..."d"}` stands for `{"b"; "c"; "d"}`.

Operations:

initialize	`PUT {"wart"; "eye"; "mouth"} IN face`
add entry to list	`INSERT "nose" IN face`
	`INSERT "eye" IN face`
remove *one* instance	`REMOVE "wart" FROM face`
number of entries	`#face = 4`
number of occurrences	`"eye"#face = 2`
scan	`FOR f IN face: INSERT f IN f2`
smallest (= first) element	`min {3; 2} = 2`
largest (= last) element	`max {3; 2} = 3`
selection	`2 th'of {1; 9; 8; 5} = 5`

Because lists are kept sorted there is no need for sorting programs. If a different order is required for the elements of a list, it is often useful to create a list of compounds. Suppose the texts are wanted in length order:

```
PUT {} IN llist
FOR text IN list: INSERT #text, text IN llist
```

To output the texts in the desired order:

```
FOR l, text IN llist: WRITE text /
```

(Note the use of a compound target following FOR.)

3.5. Tables

Tables are like arrays or tables in other languages, but the keys (= indices) may be of any type, and need not be consecutive:

```
PUT {} IN tel
PUT 364775 IN tel["Mary"]
PUT 655852 IN tel["John"]
WRITE tel
```

gives as output on the screen:

```
{["John"]: 364775; ["Mary"]: 655852}
```

All keys of a table must have the same type. All associates (the stored values) must also have the same type, but, of course, the types of the keys and the associates need not be the same.

Operations:

initialize	`PUT {} IN count`
add entries	`FOR f IN face: PUT 0 IN count[f]`
modify entries	`FOR f IN face:` `  PUT count[f]+1 IN count[f]`
delete entry	`DELETE count["nose"]`
length	`#count = 3`
the list of keys	`keys count = {"eye"; "nose"}`
number of occurrences in the associates	`2#count = 1`
number of occurrences in the keys	`"nose" # keys count = 1`
scan the associates	`FOR i IN count: PUT s+i IN s`
scan the keys	`FOR f IN keys count:` `  WRITE f, count[f] /`
smallest associate	`min count = 1`
smallest key	`min keys count = "eye"`
largest associate	`max count = 2`
largest key	`max keys count = "nose"`

The `keys` function plays an important role in nearly all applications of tables.

4. USER-DEFINED COMMANDS AND FUNCTIONS

Just as a YIELD defines a new function, a HOW'TO unit defines a new command:

```
HOW'TO EMPTY s: PUT {} IN s
HOW'TO PUSH v ON s: PUT v IN s[#s+1]
YIELD top s: RETURN s[#s]
HOW'TO POP s: DELETE s[#s]
```

Once these unit definitions have been given, the commands

```
EMPTY numstack
PUSH 3 ON numstack
```

make numstack equal to {[1]: 3}. So a HOW'TO unit is to *B* what a procedure or sub-routine is to other languages. The general appearance of user-defined commands is the same as that of predefined commands such as PUT ... IN ... . (However, no control commands of the type WHILE ... : can be defined by the user.)

In contrast to YIELDs, the execution of a command defined by a HOW'TO does change the environment when something is PUT IN a parameter: that is the very purpose of commands. Any changes made by a YIELD are made to a scratch-pad copy of the calling environment, which is thrown away upon termination of the YIELD. When an operand is given to a YIELD, only its value is passed, whereas in the case of a HOW'TO the parameter itself is passed.

Parameters in both HOW'TOs and YIELDs require no parentheses (except to indicate priorities). The top function is called thus:

```
PUT top opstack IN op
```

though, of course, you may write top(opstack) as well.

There are also dyadic functions (with two operands), such as the predefined function mod, and zeroadic functions (without operands), such as pi.

Functions with more than two operands have to be modeled as monadic (with one parameter) or dyadic ones with the aid of compound operands, e.g.,

```
YIELD (x, y, z) rotated (pitch, roll, yaw): ...
```

5. TARGETS

If we use a command such as

```
PUT {2; 3; 5; 7} IN primes
```

as an immediate command (as opposed to inside a HOW'TO or a YIELD unit), the target primes stays in existence until the command DELETE primes is given. Targets used inside a unit are local to that unit, and are alive only as

long as the execution of a call of the unit lasts. One way of introducing permanent targets in a unit is using parameters, but there is also another way. If, in the example of the previous section, there is only a single stack for EMPTY, PUSH, etc. to work on, it is a nuisance to have to pass it as a parameter each time. The definitions may then be changed to:

```
HOW'TO EMPTY:
    SHARE stack
    PUT {} IN stack

YIELD top:
    SHARE stack
    RETURN stack[#stack]
etc.
```

SHARE **stack** indicates that the identifier **stack** in this unit will not introduce or use a target local to the unit, but will refer to a permanent target. Other units wishing to refer to the same **stack**, should now also SHARE **stack**, since otherwise their use of the name **stack** would introduce a local target at that point.

Targets should be used consistently for values of one type. Local targets may be used for different types of values during the execution of different calls, but during one call a local target may contain only values of the same type. So, if we have defined this SWAP command:

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

we may use it at one time to swap two numbers, at another time to swap two texts, but never to swap a number and a text.

The system checks this partly statically, i.e. at the time the units are typed in, partly dynamically, i.e. during execution time. Checking is done on a basis of consistency, not on a basis of conformity to declarations, which do not exist in *B*.

6. CONTROL COMMANDS

There are two selection commands. In addition to

```
IF a < 0: ...
```

(no ELSE allowed), *B* also has

```
SELECT:
    test1: ...
    test2: ...
    test3: ...
```

The first test to succeed determines the alternative to be executed. At least

one test must succeed. Instead of the last test, the keyword ELSE may be used, which always succeeds:

```
SELECT:
   e < 0: INSERT e IN pos
   e > 0: INSERT e IN neg
   ELSE: PUT nzero+1 IN nzero
```

For loops, apart from FOR ... IN ..., there is also

```
WHILE test: ...
```

with the usual meaning.

For leaving a HOW'TO before its end, the command

```
QUIT
```

is used. It may occur anywhere inside a HOW'TO unit.

7. TESTS

An order is associated with all types, so <, <=, =, >=, > and <> (unequal) are defined for all values having the same type (but not for values of different types). Multiple comparisons are allowed:

```
WHILE min {x; y} < z < max {x; y}: PUT f z IN z
```

To discover if a list (or text or table) contains a certain element (or character or associate), the in test may be used:

```
SELECT:
   i in keys count: PUT count[i]+1 IN count[i]
   ELSE: PUT 1 IN count[i]
```

Tests may be combined with AND, OR and NOT:

```
IF i in keys t AND t[i] > 2: ...
IF NOT (c = "a" OR c = "b"): ...
```

Such combined tests are evaluated from left to right. In an AND combination, evaluation stops as soon as a failing test is encountered; in an OR combination it stops at the first succeeding test.

B also has versions of the mathematical quantifiers SOME, EACH and NO:

```
IF SOME d IN {2..n-1} HAS n mod d = 0:
   WRITE n, "has factor", d
```

If the SOME test succeeds, it assigns to the target between SOME and IN the first value found to satisfy the test following HAS. Examples of NO and EACH are:

```
IF NO year, anniversary IN birthday HAS year = 1813:
   WRITE "list is incomplete"

SELECT:
   EACH name IN keys birthday HAS name|1 = "B":
      WRITE "B-composers abound"
   ELSE: WRITE name
```

By combining the use of quantifiers with the keyword PARSING instead of with IN, powerful text recognition functions can be obtained:

```
IF SOME p, q, r PARSING text HAS q = ",":
   PUT p^" "^r IN text
```

This SOME test succeeds if `text` can be split into subtexts `p`, `q` and `r` (i.e., `p^q^r = text`) such that the test following HAS succeeds. So, the IF command substitutes a space for the first comma, if any, in `text`. The following command eliminates all bracketed parts of a text (assuming there are no nested brackets):

```
WHILE SOME a, l, b, r, c PARSING text HAS l^r = "[]":
   PUT a^c IN text
```

Just as a new function may be defined with a YIELD unit, a new test may be defined with a TEST unit:

```
TEST a subset b: REPORT EACH x IN a HAS x in b
```

In a TEST, REPORT is used instead of RETURN. To avoid REPORT 1=1 or REPORT 0=1, the commands SUCCEED and FAIL are available.

As in YIELDs, operands are passed by value and the environment is not affected by changes made by a TEST.

8. INPUT/OUTPUT

Input from the user at the keyboard is read by a READ command, such as

```
READ size EG 0, 0
```

where the expression following EG specifies the type of the expression to be entered (here a compound with two numeric fields). This is the only spot in *B* where a declaration-like construction is needed to make full static type checking possible. The type of the expected expression may be given by e.g. 0 or "", or by any other expression. In the following example, the READ command demands the input to be of the same type as the elements of the list `legal'moves`:

```
READ move EG min legal'moves
```

READ prompts the user, who may then enter any expression of the desired

type, using permanent targets and calls of YIELDs.

READ requires texts to be quoted, just as in *B* itself. For situations where this is a nuisance, a special version of the READ command is available:

```
READ line RAW
```

which interprets the input line as a text and assigns it to the target line.

For output to the screen, a WRITE command is used. As shown above, WRITE can handle expressions of any type. Apart from the operations mentioned in 2.2, the following operations are useful for formatting purposes:

```
align left                (7*8)<<9 = "56        "
align right               (7*8)>>9 = "        56"
centre                    (7*8)><9 = "   56     "
insert value of expression "1K is `2**10`" = "1K is 1024"
```

9. REFINEMENTS

Hierarchical programming in *B* is aided by refinements. These are like lightweight procedures, but:

- refinements belong to a unit (i.e. HOW'TO, YIELD or TEST), and are written at the end of the unit;
- refinements have no parameters or operands;
- refinements have no local names: the meaning of a name used in a refinement is determined by the context at the point where the refinement is called.

Refinements come in three kinds, analogous to HOW'TO, YIELD and TEST units. Some examples are seen in this definition of a command to compute the average of a sequence of positive numbers:

```
HOW'TO AVERAGE:
   INIT
   GET'INPUT
   WHILE valid:
      TREAT
      GET'INPUT
   OUTPUT
INIT: PUT 0, 0 IN sum, n
GET'INPUT:
   WRITE "Number: "
   READ i EG 0
TREAT: PUT sum+i, n+1 IN sum, n
valid: REPORT i > 0
OUTPUT: WRITE "The average was:", average
average: RETURN sum/n
```

Of course, refinements become more useful as programs become longer.

10. RANDOM

The command

```
DRAW r
```

assigns an arbitrary approximate number to `r`, such that `0 <= r < 1`.

The command

```
CHOOSE cap FROM {"A".."Z"}
```

assigns an arbitrary capital letter to `cap`. Likewise,

```
CHOOSE c FROM "mississippi"
```

assigns an arbitrary letter from the text `"mississippi"` to `c`. In the same way, it is possible to CHOOSE an arbitrary associate FROM a table. The meaning of CHOOSE may be described in terms of DRAW in the following way:

```
HOW'TO CHOOSE item FROM collection:
    DRAW r
    PUT 1+floor(r*#collection) IN i
    PUT i th'of collection IN item
```

(This is another example of a unit which may be used with parameters of different types — text, list or table —, as long as they are consistent with the operations inside.)

CHOOSE and DRAW are based on a pseudo-random sequence. To make programs replicable, the sequence may be entered at a specified point by starting with a SET'RANDOM command, e.g. like this:

```
SET'RANDOM "Today is my birthday."
```

or with a call with any other expression of any type as a parameter.

11. SAMPLE PROGRAM 1: A TELEPHONE LIST

If we want to maintain a telephone list, we can use a *B* table such as:

```
phone = {["Al"]: "(0765)9768"; ["Mary"]: "(0697)6458"}
```

Looking up someone's number, or changing it or adding someone to the list is easy:

```
>>> WRITE phone["Mary"]
(0697)6458
>>> PUT "(0634)6348" IN phone["Al"]
>>> PUT "(0765)7754" IN phone["John"]
```

(>>> is the prompt for an immediate command.) For starting the list, it is convenient to have a piece of program to help:

```
HOW'TO FILL t:
    PUT {} IN t
    READ'NAME
    WHILE name > "":
        READ'NUMBER
        PUT number IN t[name]
        READ'NAME
READ'NAME:
    WRITE "Name: "
    READ name RAW
READ'NUMBER:
    WRITE "Number: "
    READ number RAW
```

Having given this definition we may fill a table `tel` by just typing the names and the numbers:

```
>>> FILL tel
Name: John
Number: (0765)7754
Name: Mary
Number: (0697)6458
Name: Al
Number: (0634)6348
Name:
>>>
```

If it happens that we want to know if there is somebody in `tel` with, say, the number `(0634)6348`, we can check in this way:

```
>>> IF SOME n IN keys tel HAS tel[n] = "(0634)6348":
        WRITE n
Al
```

If this happens very often, it may be useful to construct the inverse table, in which we can directly look up the name belonging to a telephone number. Assuming there are no people in the table sharing the same number, the inverse table `name` can be built this way:

```
>>> PUT {} IN name
>>> FOR n IN keys tel: PUT n IN name[tel[n]]
>>> WRITE name[″(0634)6348″]
Al
```

For printing the table in a neat way we can define this command PRINT'TEL:

```
HOW'TO PRINT'TEL t:
   FOR a IN keys t:
      WRITE a^(″.″^^20))|20, t[a] /
```

The new command can be used for printing the original table tel or for the inverse table name:

```
>>> PRINT'TEL tel
Al..................(0634)6348
John................(0765)7754
Mary................(0697)6458
>>> PRINT'TEL name
(0634)6348..........Al
(0697)6458..........Mary
(0765)7754..........John
```

12. SAMPLE PROGRAM 2: A GUESSING GAME

Here is a simple program that makes the computer guess which letter of the alphabet we have thought of. The program takes the form of a definition of a new command GUESS:

```
HOW'TO GUESS:
   PUT {"a".."z"} IN cands
   PICK
   WHILE answer <> "OK":
      IF cands = {}:
         WRITE "There are no letters left."
         QUIT
      PICK
PICK:
   PUT middle IN guess
   WRITE guess, " "
   READ answer RAW
   SELECT:
      answer = "+":
         PUT {guess..max cands} IN cands
         REMOVE guess FROM cands
      answer = "-":
         PUT {min cands..guess} IN cands
         REMOVE guess FROM cands
      ELSE:
         WRITE "I'm glad."
         PUT "OK" IN answer
middle: RETURN (round(#cands/2)) th'of cands
```

Once we have given this definition, we can activate it by simply typing the command GUESS. The computer will then show its guesses on the screen, each time waiting for our hint, which may be '-' if the letter we have thought of comes earlier in the alphabet, or '+' if it comes later, or anything else if the guess is right. A session might look like this:

```
>>> GUESS
m -
f +
i +
k -
j all right
I'm glad.
>>>
```

If we want to change roles with the computer, having it think of a letter and let us guess, we can define this command:

```
HOW'TO LET'ME'GUESS:
   WRITE "Guess my letter: "
   CHOOSE it FROM {"a".."z"}
   READ guess RAW
   PUT 1 IN count
   WHILE guess <> it:
      SELECT:
         guess < it: WRITE "Too low, try higher: "
         guess > it: WRITE "Too high, try lower: "
      READ guess RAW
      PUT count+1 IN count
   SELECT:
      count = 1: WRITE "Right the first time!"
      count <= 5: WRITE "You got it, very good!"
      ELSE: WRITE "OK, it took `count` tries."
```

A session where we do the guessing will look something like this:

```
>>> LET'ME'GUESS
Guess my letter: n
Too high, try lower: h
Too high, try lower: e
Too low, try higher: f
You got it, very good!
```

USING B

CONTENTS

Introduction

This chapter is an introduction to using the *B* system. Throughout the chapter, the notation [accept], for instance, is used to represent the operation *accept*. This isn't an actual key on the keyboard, but represents whatever key is used for the accept operation, since you can decide this yourself. The default key bindings, as they are called, are described on the Quick Reference card.

On smaller machines, where space is at a premium, there is no built in editor, and so there is no focus, no suggestions, and none of the editing operations described in this chapter are available.

Starting up

The first response you should get from the *B* system when you start it up is a prompt that looks like this:

```
>>> ?
```

The underlined question mark is the indication from the *B* system that it is expecting input from you. (In fact, it depends on the sort of screen you have whether it is underlined, displayed in reverse video, or what. In any case it is displayed in some special way, and we shall use underlining here.) When it follows the three arrows >>>, called the *command prompt*, it is expecting you to type in a command. The question mark is called a *hole* and indicates that something should be filled in; the underline is called the *focus* and shows where you are currently working.

Typing to *B*: suggestions

You can fill this hole by typing in a WRITE command for instance: you type a W (which you don't have to type in upper-case: the system knows that it may only be upper-case here), and you immediately see:

```
>>> W?RITE ?
```

This extra stuff to the right of the focus is a *suggestion*. Most times that you type a W as the first letter of a command, it is because you want a WRITE. Therefore the editor suggests this, with an additional hole for the expression that you want to write. If you do want a WRITE (as in this case) you may press [accept] to accept the suggestion — the editor then moves to the first unfilled hole in the command, which in this case is the only one, and you get:

```
>>> WRITE ?
```

You can now type an expression and press [newline]. The system evaluates the expression, prints the result, and then gives you a new command prompt. Here are a few examples of WRITE commands:

```
>>> WRITE 2+2
4
>>> WRITE root 2
1.414213562373095
>>> WRITE pi
3.141592653589793
>>> WRITE 10**100-1
99999999999999999999999999999999999999999999999999999999999999999999999999999999
>>> ?
```

Undoing mistakes

If you make a mistake while typing and spot it before you type [newline], an easy way to correct it is to use [undo]. Pressing [undo] takes you back to the situation before you typed the last key (*exactly* the situation, as you will see clearly after a little use). If you type [undo] twice, you will be taken back to the situation as it was two keys ago, and so forth. You can regard [undo] as a way of travelling back in time.

Thus, if you meant to type `WRITE pi`, but instead typed `WRITE po`, you will see this:

```
>>> WRITE po?
```

Now pressing [undo] will give you

```
>>> WRITE p?
```

Now you can type the `i` and the [newline]. Repeatedly using [undo] instead will give you the following sequence:

```
>>> WRITE p?
>>> WRITE ?
>>> W?RITE
>>> ?
```

Because of memory constraints, currently you can only go back a limited number of keystrokes, and in any case only as far as the command prompt (and thus not back to previous commands). You will see other ways to correct mistakes shortly.

If you should use [undo] once (or more times) too often, you can use [redo] to undo the effects of [undo].

Typing brackets and quotes

If you make a mistake so that the result is illegal *B*, but don't notice before you press [newline] you will get an error message:

```
>>> WRITE root 9+16
*** There's something I don't understand
    WRITE root 9+16
               ^
*** The problem is: priorities? use ( and ) to resolve
>>> ?
```

The problem here is that the system doesn't know if you want to apply `root` to `9` or `9+16` and you should use brackets to show which.

When you type an open bracket, the system automatically supplies the matching closing bracket for you:

```
>>> WRITE root(?)
```

You now type in the expression

```
>>> WRITE root(9+16?)
```

You may now type [newline] ([accept] will take you over the closing bracket, but it is not necessary to do this):

```
>>> WRITE root(9+16)
5
>>> ?
```

You can write any legal *B* value:

```
>>> WRITE {1..10}
{1; 2; 3; 4; 5; 6; 7; 8; 9; 10}
>>> WRITE "Hello! "^^3
Hello! Hello! Hello!
>>> ?
```

Just as with brackets, the system automatically supplies the closing brace }, and the closing quote ". In the latter case, where you want to type something after the closing quote you may either use [accept] or type the quote itself, in order to position after it.

Immediate commands

Commands typed as a response to the command prompt (like all those seen up to now) are called 'immediate' commands, since they are executed immediately. Another example is the `PUT` command. Just as with `WRITE`, when you type the first letter of the command, the system provides a suggestion:

```
>>> P?UT ? IN ?
```

Again, you use [accept] to go to the first hole:

```
>>> PUT ? IN ?
```

Here you type an expression,

```
>>> PUT root 2? IN ?
```

followed by another [accept] to take you to the second hole:

```
>>> PUT root 2 IN ?
```

where you can type a target, followed by [newline]:

```
>>> PUT root 2 IN a
>>> PUT root 3 IN b
>>> WRITE a
1.4142135623730 95
>>> WRITE b
1.732050807568877
>>> WRITE a, b
1.4142135623730 95 1.732050807568877
>>> WRITE a*a, b*b
2 3
>>> ?
```

Permanent targets

The targets that you create in this way, through immediate commands, are called 'permanent targets', because if you stop using the system and come back later and start using the system again you will find that the targets are still there, with the same values as before.

You can find out which targets exist by typing two equals signs after the prompt:

```
>>> ==
a b
>>> PUT "hello", {1..10} IN message, list
>>> ==
a b list message
>>> WRITE list
{1; 2; 3; 4; 5; 6; 7; 8; 9; 10}
>>> WRITE message
hello
>>> ?
```

Deleting targets

To get rid of targets you no longer want, use the DELETE command:

```
>>> DELETE a, b
>>> ==
list message
>>> WRITE a
*** Can't cope with problem in your command
    WRITE a
*** The problem is: a has not yet received a value
>>> ?
```

As you can see, after the DELETE command both a and b have ceased to exist.

Other immediate commands

In fact, almost any *B* command can be used as an immediate command; the only exceptions are the commands used to terminate TESTs and YIELDs, namely RETURN, REPORT, SUCCEED and FAIL (as well as SHARE but that isn't strictly a command).

```
>>> WRITE list
{1; 2; 3; 4; 5; 6; 7; 8; 9; 10}
>>> INSERT 5 IN list
>>> REMOVE 6 FROM list
>>> WRITE list
{1; 2; 3; 4; 5; 5; 7; 8; 9; 10}
>>> CHOOSE number FROM list
>>> WRITE number
9
>>> CHECK 5 in list
>>> CHECK 6 in list
*** Your check failed in your command
    CHECK 6 in list
>>> FOR i IN list: WRITE 10*i
10 20 30 40 50 50 70 80 90 100
>>> ==
list message number
>>> ?
```

Note that a CHECK command that succeeds doesn't print any message, and that the target i used in the FOR command doesn't exist afterwards.

Indentation

When you want to type a WHILE command, and you type the initial W, the suggestion you get is of course W?RITE ?. The system always matches the suggestion to what you have typed, so if you type an H here the system then suggests

```
>>> WH?ILE ?:
```

Now you can press [accept] to go to the hole, type the test, and press [newline]:

```
>>> WHILE list <> {}:
        ?
```

Because the *B* system knows that the commands of a WHILE must be indented, it indents for you automatically. You may now type in the commands to be part of the WHILE, and each time the system indents you to the right place:

```
>>> WHILE list <> {}:
        CHOOSE number FROM list
        REMOVE number FROM list
        WRITE number
        ?
```

After the last command you just need to type an extra [newline], and the system undoes the indentation one level, and executes the WHILE:

```
>>> WHILE list <> {}:
        CHOOSE number FROM list
        REMOVE number FROM list
        WRITE number
2 8 4 7 9 5 3 1 5 10
>>> WRITE list
{}
>>> ?
```

If there is only one simple command to be repeated, it may be on the same line as the WHILE, but doesn't have to be:

```
>>> WHILE number in list: REMOVE number FROM list
```

Finishing a *B* session

The one command that has a different meaning when you use it as an immediate command is QUIT, which just terminates the *B* session. When you type the Q you will get a suggestion as usual.

```
>>> Q?UIT
```

Here there are no holes, but you must still press [accept] to accept the suggestion, before pressing [newline].

Making your own commands

You can create your own commands by typing in a how-to unit that defines what your command means. Suppose you type in response to the *B* prompt:

```
>>> HOW'TO GREET:
        WRITE "Hello"
```

The system gives a suggestion for HOW'TO, and supplies indentation for you, just as with WHILE, and you finish by pressing [exit] or by repeatedly pressing [newline] until you get the command prompt again (an exit can't be undone). You execute your own commands just as built in commands by typing its name after the prompt. You will notice that after typing the G you will get a suggestion for it:

```
>>> G?REET
```

Just as with QUIT, there are no holes, but you must [accept] the suggestion.

```
>>> GREET?
```

Now press [newline] and your command gets executed, and you get the prompt again:

```
>>> GREET
Hello
>>> ?
```

You may use your own commands just like any in-built command:

```
>>> FOR i IN {1..10}: GREET
HelloHelloHelloHelloHelloHelloHelloHelloHelloHello
>>> ?
```

If the command you define has parameters, you get holes in the suggestion, just as with normal commands. For instance, suppose you prefer LET a BE 10 to *B*'s PUT 10 IN a. Well, then you can define the following unit:

```
>>> HOW'TO LET a BE b:
        PUT b IN a
```

Now typing an L, you get the following:

```
>>> L?ET ? BE ?
```

You can then use [accept] in the usual way.

Correcting errors: the focus

Apart from [undo], another way of correcting errors is to correct a whole line. Here you use the ability to move the focus about. Earlier the focus was a single character, just the hole. However, the focus may be more than one character: it may be several characters, a whole command, or even several commands.

One way of moving the focus is using [upline] and [downline]. [Upline] moves the focus up to the previous line so that it includes the whole line. [Downline] does the same but in the other direction. So if you have the following situation:

```
>>> FOR i IN {1..3}:
        WRITE /
        GREET
        WRITE /?
```

then typing [upline] gives you

```
>>> FOR i IN {1..3}:
        WRITE /
        GREET
        WRITE /
```

Here the hole in the last line has disappeared (because the line is legal *B*) and the focus has moved up to the whole of the preceding line. You may press [upline] several time to go up several lines. So, pressing [upline] again gives:

```
>>> FOR i IN {1..3}:
        WRITE /
        GREET
        WRITE /
```

Now the point of all this is, that if you have a line in a command or unit that you want to change, you can move the focus to it, and press [delete] to get rid of it. This leaves a hole in its place so that you can type a replacement line:

```
>>> FOR i IN {1..3}:
        ?
        GREET
        WRITE /
```

If you don't want to replace the line, but completely delete it, then pressing [delete] again deletes the hole too:

```
>>> FOR i IN {1..3}:
        GREET
        WRITE /
```

Remember that [undo] works with *any* operation, so if you accidentally delete the wrong piece of text, [undo] will bring it back again.

Changing existing units

Just as you can type two equals signs to find out what permanent targets you have, you can type two colons to find out what units you have. This gives you a list of the first line of each unit:

```
>>> ::
HOW'TO GREET:
>>> ?
```

If you want to change any of these units, you can type a colon followed by the name of the unit you want:

```
>>> :GREET
```

(If the unit you want to change is the last unit you typed in or changed in this session, or the last unit that you got an error message about, then you don't even need to type its name: the system remembers the name of the unit, so all you need to do is type a single colon.)

What happens now is that the whole unit is displayed (or as much as will fit on the screen if it is big) with the focus on the line you were last at in the unit. You can now move the focus to the lines you want to change, and change them. When you have finished, you use [exit].

Errors in units

If when you type in or change a unit, the result has an error in it, you will get an error message from the *B* system. This may happen when you press [exit], or when you run the unit, depending on the sort of error it is. For instance, in this unit, the parameter is `x`, but `n` is used instead:

```
>>> YIELD square x:
        RETURN n*n
>>> WRITE square 4
*** Can't cope with problem in line 2 of square
    RETURN n*n
*** The problem is: n has not yet received a value
>>> ?
```

When you get such a message, it is very easy to make the necessary correction: since the unit you want to change is the last one that you got an error message for, you only need to type a colon after the prompt:

```
>>> :
YIELD square x:
   RETURN n*n
```

As you see, you are positioned at the line that gave the error message. Now you can either press [delete] and retype the whole line, or move the focus to the part that is in error, and deal only with that.

Other focus moves

Apart from [upline] and [downline] there are four other groups of focus operations:

[first] and [last] to make the focus smaller,
[widen] and [extend] to enlarge it,
[previous] and [next] to move it sideways, and
[up], [down], [left] and [right] for moving a single hole focus about.

Remember that [undo] works with *any* operation, so if you accidentally use the wrong focus move, [undo] will move it back again.

Making the focus smaller

In the case of the YIELD above we want to make the focus smaller, by going to the last part of the line. The operation [last] works by narrowing the focus to the last part of what is enclosed. Thus, pressing [last] we see:

```
YIELD square x:
   RETURN n*n
```

Pressing [delete] deletes the part in the focus, leaving only a hole:

```
YIELD square x:
   RETURN ?
```

Now we can type the correct expression and then press [exit]:

```
YIELD square x:
   RETURN x*x
>>> ?
```

To give you more of an idea what [last] does, here is what happens with a sequence of them on the following:

```
FOR i IN {1..10}: WRITE i*i
FOR i IN {1..10}: WRITE i*i
FOR i IN {1..10}: WRITE i*i
FOR i IN {1..10}: WRITE i*i
FOR i IN {1..10}: WRITE i*i?
```

The operation [first] narrows the focus to the first enclosed part of the focus. Consider the following unit:

```
>>> YIELD all'chars:
        WRITE {" "..”~”}
>>> WRITE all'chars
*** There's something I can't resolve in all'chars
    WRITE {" "..”~”}
*** The problem is: YIELD-unit returns no value
>>> ?
```

The problem is WRITE has been used, when it should have been RETURN, since this is a YIELD, not a HOW'TO. So, we type a colon after the prompt:

```
>>> :
YIELD all'chars:
   WRITE {" "..”~”}
```

Pressing [first] narrows the focus to just the keyword:

```
YIELD all'chars:
   WRITE {" "..”~”}
```

Pressing [delete] deletes the focus:

```
YIELD all'chars:
   ? {" "..”~”}
```

and now we can type RETURN followed by [exit]:

```
YIELD all'chars:
   RETURN {" "..”~”}
>>> ?
```

To give you more of an idea what [first] does, here is what happens with a sequence of them:

```
FOR i IN {1..10}: WRITE i*i
FOR i IN {1..10}: WRITE i*i
FOR i IN {1..10}: WRITE i*i
FOR i IN {1..10}: WRITE i*i
?FOR i IN {1..10}: WRITE i*i
```

Making the focus larger The operation [widen] enlarges the focus to the next larger object. For example, if you have this:

```
>>> PUT {"bread"; "butter"} IN shopping
```

[widen] gives you

```
>>> PUT {"bread"; "butter"} IN shopping
```

and a [delete] removes both entries:

```
>>> PUT {?} IN shopping
```

Here is a sequence of [widen]:

```
FOR i IN {?1..10}: WRITE i*i
FOR i IN {1..10}: WRITE i*i
FOR i IN {1..10}: WRITE i*i
FOR i IN {1..10}: WRITE i*i
FOR i IN {1..10}: WRITE i*i
FOR i IN {1..10}: WRITE i*i
```

The operation [extend] extends the focus to the right if possible, and otherwise to the left. For instance, in the following situation

```
>>> PUT {"bread"; "butter"} IN shopping
```

[extend] gives you

```
>>> PUT {"bread"; "butter"} IN shopping
```

and [delete] then leaves the first entry:

```
>>> PUT {"bread"?} IN shopping
```

Here is a sequence of extends:

```
FOR i IN {?1..10}: WRITE i*i
FOR i IN {1..10}: WRITE i*i
FOR i IN {1..10}: WRITE i*i
FOR i IN {1..10}: WRITE i*i
FOR i IN {1..10}: WRITE i*i
FOR i IN {1..10}: WRITE i*i
FOR i IN {1..10}: WRITE i*i
FOR i IN {1..10}: WRITE i*i
FOR i IN {1..10}: WRITE i*i
FOR i IN {1..10}: WRITE i*i
FOR i IN {1..10}: WRITE i*i
FOR i IN {1..10}: WRITE i*i
```

As you can see, [extend] usually enlarges the focus at a slower rate than [widen].

Moving the focus sideways

The operation [next] moves the focus to the next object to the right. For instance, to focus on the expression of a PUT command:

```
PUT a*a IN b
```

press [first]:

```
PUT a*a IN b
```

and then [next]:

```
PUT a*a IN b
```

Here are some examples of [next] and [first] in action:

```
WHILE list <> {}:
   CHOOSE number FROM list
   WRITE number
   REMOVE number FROM list
```

[next]:

```
WHILE list <> {}:
   CHOOSE number FROM list
   WRITE number
   REMOVE number FROM list
```

[first]:

```
WHILE list <> {}:
   CHOOSE number FROM list
   WRITE number
   REMOVE number FROM list
```

[next]:

```
WHILE list <> {}:
   CHOOSE number FROM list
   WRITE number
   REMOVE number FROM list
```

[first]:

```
WHILE list <> {}:
   CHOOSE number FROM list
   WRITE number
   REMOVE number FROM list
```

[next]:

```
WHILE list <> {}:
   CHOOSE number FROM list
   WRITE number
   REMOVE number FROM list
```

[first]:

```
WHILE list <> {}:
   CHOOSE number FROM list
   WRITE number
   REMOVE number FROM list
```

[next]:

```
WHILE list <> {}:
   CHOOSE number FROM list
   WRITE number
   REMOVE number FROM list
```

[first]:

```
WHILE list <> {}:
   CHOOSE number FROM list
   WRITE n?umber
   REMOVE number FROM list
```

[next]:

```
WHILE list <> {}:
   CHOOSE number FROM list
   WRITE nu?mber
   REMOVE number FROM list
```

The operation [previous] works exactly the same as [first] but in the opposite direction.

Moving a single hole

The final way of moving the focus is to use the operations [up], [down], [left] and [right]. These operations make a hole before the focus, position on it, and then move it one line up or down, or one character left or right. For example, here is a sequence of [left]:

```
WHILE list <> {}:
WHILE ?list <> {}:
WHILE? list <> {}:
WHIL?E list <> {}:
WHI?LE list <> {}:
WH?ILE list <> {}:
W?HILE list <> {}:
?WHILE list <> {}:
```

Here is a sequence of [down]:

```
WHILE list <> {}:
   CHOOSE number FROM list
   REMOVE number FROM list
   WRITE number

WHILE list <> {}:
   CHO?OSE number FROM list
   REMOVE number FROM list
   WRITE number

WHILE list <> {}:
   CHOOSE number FROM list
   REM?OVE number FROM list
   WRITE number

WHILE list <> {}:
   CHOOSE number FROM list
   REMOVE number FROM list
   WRI?TE number
```

The operations [right] and [up] work exactly the same, but in the other directions.

Copying

It is often the case that you want to duplicate a piece of program. If the focus is on something other than a hole, [copy] copies whatever is in the focus to what is called the *copy buffer* and lets you know that there is something in the copy buffer by displaying the words `[Copy buffer]` at the bottom of the screen, along with an indication of what is stored.

If, however, the focus is on a hole, [copy] copies the contents of the buffer into that hole. The `[Copy buffer]` message then disappears, though actually the contents of the buffer remain, so you can continue to use it. However, if the contents of the buffer may not be copied there, you just get a bleep; for instance you can't copy a `WRITE` command to a place where an expression must be.

You can use [copy] for moving things too: focus on what you want, press [copy], press [delete] twice to delete it and the hole that gets left after the first delete, move to where you want, make a hole, and press [copy] again.

For example, suppose you use the following unit:

```
HOW'TO AVERAGE tl:
   PUT 0 IN sum
   FOR x IN tl: PUT sum+x IN sum
   WRITE sum/#tl

>>> AVERAGE {1...3}
2
>>> AVERAGE {1}
1
>>> AVERAGE {}
*** Can't cope with problem in of your unit AVERAGE
    WRITE sum/#tl
*** The problem is: in i/j, j is zero
>>> ?
```

The WRITE should only be done if tl is not empty, so it must be put in an IF command. So we display the unit:

```
>>> :
HOW'TO AVERAGE tl:
   PUT 0 IN sum
   FOR x IN tl: PUT sum+x IN sum
   WRITE sum/#tl
```

Press [copy]:

```
HOW'TO AVERAGE tl:
   PUT 0 IN sum
   FOR x IN tl: PUT sum+x IN sum
   WRITE sum/#tl
[Copy buffer: WRITE ...]
```

Press [delete]:

```
HOW'TO AVERAGE tl:
   PUT 0 IN sum
   FOR x IN tl: PUT sum+x IN sum
   ?
[Copy buffer: WRITE ...]
```

Type IF #tl>0: and [accept]:

```
HOW'TO AVERAGE tl:
   PUT 0 IN sum
   FOR x IN tl: PUT sum+x IN sum
   IF #tl>0: ?
[Copy buffer: WRITE ...]
```

and then [copy] again:

```
HOW'TO AVERAGE tl:
   PUT 0 IN sum
   FOR x IN tl: PUT sum+x IN sum
   IF #tl>0: WRITE sum/#tl?
```

If you need to copy to a hole, making a hole is straightforward: for instance, a [newline] always makes a hole on a new blank line after the line that the focus was positioned on, [accept] always takes you to the first hole on a line, or if there is no hole on the line, then it makes one at the end of the line, and [left] or [right] makes a hole to the left or right of the first character of the focus.

You can use [copy] for copying between different units too. Even if you end the *B* session, and come back later, and start using *B* again, the copy buffer is kept.

Additionally, if the `[Copy buffer]` message isn't being displayed, that is, if the copy buffer is empty, and you [delete] something, whatever you delete is put into the copy buffer; similarly if the copy buffer is empty, each immediate command is stored in the copy buffer. Thus if you mis-type an immediate command, you can use [copy] to bring it back, and edit it.

Remember that [undo] works with *any* operation, so if you accidently copy the wrong piece of text, [undo] will uncopy it.

Dealing with brackets and quotes

The *B* system always ensures that you have matching brackets and quotes. This means that you can never insert or delete just one of a pair, but must handle both at once. Inserting is easy: you just focus on what you want to have in brackets:

```
PUT root x + 1 IN a
```

and type an open bracket:

```
PUT (?root x)+ 1 IN a
```

Deleting, you have to copy the contents, delete the whole, and copy the contents back:

```
PUT 1+(root x) IN a
```

[copy] and [widen]

```
PUT 1+(root x) IN a
```

[delete]

```
PUT 1+? IN a
```

[copy]

```
PUT 1+root x? IN a
```

Capital letters

As you will have remarked, you hardly ever have to type a capital letter: the *B* system usually knows where a letter must be a capital, and so supplies it for you, even if you don't use the shift key.

There are a small number of places where *B* can't tell, and where you do have to use the shift key:
— In tests, for SOME, EACH, NO, NOT, AND, and OR
— For the E in numbers, like 3.14E100
— In the " : " command when you want to display or make a change to a HOW'TO
— For the name of a command-refinement when you define it
— For the second and later keywords of commands not yet defined.
In all these cases, you only have to use the shift key for the *first* letter: *B* then knows that the rest of the word must be in upper case.

Renaming and deleting units

If you change the name of a unit, or the number of operands of a YIELD or TEST, by changing its heading, then you get the new unit *and* the old one. So, renaming GREET into HELLO, and then giving a :: command would give:

```
>>> ::
HOW'TO GREET:
HOW'TO HELLO:
HOW'TO AVERAGE tl:
YIELD square x:
>>> ?
```

If you delete the *whole* of a unit (by pressing [widen] until the focus is on the whole unit, and then pressing [delete]) the unit disappears. Thus deleting GREET like this will give you:

```
>>> ::
HOW'TO HELLO:
HOW'TO AVERAGE tl:
YIELD square x:
>>> ?
```

Changing targets

You may also use the editor for changing permanent targets. Just as you use a single " : "for editing units, a single " = " followed by the name of a target will display the contents of the target, and let you make changes in the usual way. In fact, you may replace the contents by any *expression*. When you press [exit], the expression is evaluated, and if all is ok the value is put in the target.

Workspaces

A workspace is a collection of units plus a permanent environment. You can have several workspaces: to create a new one you just start *B* up in a new directory in the filestore.

To move parts of units and targets between workspaces, just use [copy]: start *B* up in the one workspace, save whatever you want to move in the copy buffer, exit *B*, move to the destination workspace, re-enter *B*, and copy the buffer back.

If you want to move a whole unit from one workspace to another, you copy that unit in the copy buffer, move to the other workspace, type the first line of the unit, press [return] once, press [widen] until the focus is on the whole first line, press [delete], and then [copy].

To move a whole permanent target from one workspace to another, you copy its value in the first workspace, then move to the other workspace, create a target of the desired name and type (with a PUT command), edit the new target, and copy the value from the copy buffer.

Record and play

Sometimes you need to repeat a sequence of keystrokes several times. For instance, if you want to rename a target in a unit, you have to do it once for each occurrence of the target. An easy way to do this is to *record* a sequence of keystrokes. If you press [record], the message [recording] appears at the bottom of the screen, and any keys that you type thereafter are processed normally and recorded at the same time, until you press [record] again. Then pressing [play] plays those recorded keystrokes back.

So, for instance, focus on the target you want to rename, press [record], press [delete], type the new name, and press [record] again. Then focus on the next occurrence of the target, and press [play].

Keystrokes that cause an error during recording are not recorded.

Very large units

If a unit gets so large that it doesn't all fit on the screen, then the *B* system displays as much of it as possible and displays a bar on the bottom line of the screen to indicate which part of it with relation to the whole unit is visible. For instance, this would show you that you are looking at roughly the middle third of the unit:

```
-----------############-----------
```

Redisplaying the screen

Sometimes the screen can get messed up (for instance, if it gets accidentally unplugged). If this happens, or you don't believe what you see on the screen, you can always get confirmation by pressing [look]. This causes the screen to be redisplayed.

Interrupting a running command

If a command is executing, and you want to stop it, pressing [interrupt] aborts the command and gives you a prompt again.

Incomplete units

If, when typing in or correcting a command or unit, you press [exit] and there are still unfilled holes, the system tells you so, and you must fill or delete them. If you want to exit leaving the holes, to fill them later, use [interrupt]. The system won't let you run an incomplete unit.

Getting help

Pressing [help] gives you a quick summary of all key bindings. Refer to the "Quick Reference" card for a brief reminder of the features of the *B* language, and the "Description of *B*" for more detailed explanations.

Running *B* non-interactively

You can also run *B* non-interactively. In this case *B* commands are read from the standard input, and executed. You may only use normal *B* commands in this case: no focus moves, or editing, and no `::` or `==` commands.

Summary of editing operations

[up] [down] [left] [right]: Make the focus a hole, and go one line up or down, or one character left or right.

[upline] [downline]: Move the focus to the whole line above or below.

[previous] [next]: Move the focus to the preceding or following object.

[widen]: Enlarge the focus to the next enclosing object.

[extend]: Enlarge the focus to the right, or if this is not possible, to the left.

[first] [last]: Reduce the focus to the first or last enclosed object.

[delete]: Delete the focus. If the copy buffer is empty, silently save the contents of the focus there.

[copy]: If the focus is more than a hole, copy its contents to the copy buffer. If the focus is a hole, copy the contents of the copy buffer into the hole.

[accept]: Move the focus to the first hole on the first line that the focus is in, or to after the first enclosing closing bracket, brace or quote, whichever comes first, or otherwise to the end of the line.

[return]: Go to a new line, or if on an empty line, decrease the indentation one level, or otherwise do an [exit].

[undo]: Undo the effect of the last key stroke, whatever it was (except [exit] or a [return] that causes an exit).

[redo]: Undo the effect of the last [undo].

[record]: Start recording all keystrokes typed. If recording already in progress, stop recording.

[play]: Play recorded keystrokes back.

[look]: Redisplay screen.

[help]: Display current key-bindings.

[exit]: Exit current unit, or command (can't be undone).

[interrupt]: Interrupt current command executing.

An example

Suppose we want to change the AVERAGE unit given earlier so that it uses a separate YIELD that returns the sum of the elements of a list or table. So we start off with

```
HOW'TO AVERAGE tl:
   PUT 0 IN sum
   FOR x IN tl: PUT sum+x IN sum
   IF #tl>0: WRITE sum/#tl
```

and want to finish with

```
HOW'TO AVERAGE tl:
   IF #tl>0: WRITE (sum tl)/#tl

YIELD sum tl:
   PUT 0 IN sum
   FOR x IN tl: PUT sum+x IN sum
   RETURN sum
```

First we have to alter AVERAGE so that it uses the new yield:

```
>>> :AVERAGE
HOW'TO AVERAGE tl:
   PUT 0 IN sum
   FOR x IN tl: PUT sum+x IN sum
   IF #tl>0: WRITE sum/#tl
```

Press [up] twice:

```
HOW'TO AVERAGE tl:
   PUT 0 IN sum
   FOR x IN tl: PUT sum+x IN sum
   IF #tl>0: WRITE sum/#tl
```

press [extend]

```
HOW'TO AVERAGE tl:
   PUT 0 IN sum
   FOR x IN tl: PUT sum+x IN sum
   IF #tl>0: WRITE sum/#tl
```

press [copy] (we can use the two commands for the body of `sum`) and then [delete]:

```
HOW'TO AVERAGE tl:
   ?
   IF #tl>0: WRITE sum/#tl
[Copy buffer: PUT ... IN ... ...]
```

Press [delete] again:

```
HOW'TO AVERAGE tl:
   IF #tl>0: WRITE sum/#tl
[Copy buffer: PUT ... IN ... ...]
```

Now we have to alter the `WRITE` to call the new `YIELD`: press [last]

```
HOW'TO AVERAGE tl:
   IF #tl>0: WRITE sum/#tl
[Copy buffer: PUT ... IN ... ...]
```

[last]

```
HOW'TO AVERAGE tl:
   IF #tl>0: WRITE sum/#tl
[Copy buffer: PUT ... IN ... ...]
```

first

```
HOW'TO AVERAGE tl:
   IF #tl>0: WRITE sum/#tl
[Copy buffer: PUT ... IN ... ...]
```

Type an open bracket:

```
HOW'TO AVERAGE tl:
   IF #tl>0: WRITE (?sum)/#tl
[Copy buffer: PUT ... IN ... ...]
```

Type right three times:

```
HOW'TO AVERAGE tl:
   IF #tl>0: WRITE (sum?)/#tl
[Copy buffer: PUT ... IN ... ...]
```

and type a space and `tl`:

```
HOW'TO AVERAGE tl:
   IF #tl>0: WRITE (sum tl?)/#tl
[Copy buffer: PUT ... IN ... ...]
```

Now exit. Now we want to use the copied text to make the new `YIELD`: Type `y`:

```
>>> Y?IELD ?:
[Copy buffer: PUT ... IN ... ...]
```

accept:

```
>>> YIELD ?:
[Copy buffer: PUT ... IN ... ...]
```

Type `sum tl` followed by return:

```
YIELD sum tl:
   ?
[Copy buffer: PUT ... IN ... ...]
```

copy:

```
YIELD sum tl:
   PUT 0 IN sum
   FOR x IN tl: PUT sum+x IN sum?
```

and then [return]:

```
YIELD sum tl:
   PUT 0 IN sum
   FOR x IN tl: PUT sum+x IN sum
   ?
```

Type `r`:

```
YIELD sum tl:
   PUT 0 IN sum
   FOR x IN tl: PUT sum+x IN sum
   R?ETURN ?
```

[accept]:

```
YIELD sum tl:
   PUT 0 IN sum
   FOR x IN tl: PUT sum+x IN sum
   RETURN ?
```

Type `sum` and then [exit].

```
YIELD sum tl:
   PUT 0 IN sum
   FOR x IN tl: PUT sum+x IN sum
   RETURN sum
>>> ?
```

CONTENTS

DESCRIPTION OF B

1. VALUES IN B

B has two basic types of values: numbers and texts, and three ways of making new types of values from existing ones: compounds, lists and tables. The built-in functions for operating on these values are described in section 6.1.6 entitled "Formulas with predefined functions".

Numbers

Numbers come in two kinds: exact and approximate. Exact numbers are rational numbers. For example, `1.25 = 5/4`, and `(1/3)*3 = 1`. There is no restriction on the size of numerator and denominator. Approximate numbers are implemented by whatever the hardware or software has to offer for fast but approximate arithmetic (floating point).
The arithmetic operations and many other functions give an exact result when their operands are exact, and an approximate result otherwise, but the function `sin`, for example, always returns an approximate number.
An exact number can be made approximate with the ~ function (e.g. `~1.25`); the functions `round`, `floor` and `ceiling` can be used to convert an approximate number to an exact one. Exact and approximate numbers may be mixed in arithmetic, as in `4 * atan 1`.

Texts

Texts (strings) are composed of printable ASCII characters. They are variable length, and are ordered in the usual lexicographic way: `"a" < "aa" < "b"`.
There is no type "character": a text of length one will do.
The printable characters are the 95 characters represented below, where the blank space preceding ' ! ' stands for the (otherwise invisible) space character:

```
 !"#$%&'()*+,-./0123456789:;<=>?
@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_
`abcdefghijklmnopqrstuvwxyz{|}~
```

The ordering on the characters is the ASCII collating order, which is the order in which the characters are displayed above.

Compounds

A compound consists of a sequence of values, its "fields". For example, the number `3` and the text `"xyz"` may be combined to give the compound `3, "xyz"`. Compounds are also ordered lexicographically.
For example, `(3, "xyz") < (3, "yz") < (pi, "aaa")`. For this to be meaningful, the compounds that are compared must be of the same type. This means that they have the same number of fields, and that corresponding fields are of the same type.
The only way to obtain the individual fields of a compound is to put it in a multiple-target with the right number of components, as in

```
PUT name IN last'name, first'name, middle'name.
```

Lists

A list is a *sorted* sequence of values, its "entries". All entries of a list must be of the same type, and this determines the type of the list. The length of a list may vary without influencing its type. When an entry is inserted in a list (with an `INSERT` command), it is automatically inserted in the list in the proper position in the sorting order. A list may contain duplicates of the same entry. Entries may be removed with the `REMOVE` command. Again, lists themselves are ordered lexicographically.

Tables

A table consists of a (sorted) sequence of "table entries". Each table entry is a pair of two values: a *key* and an *associate*. All keys of a table must be of the same type; similarly, all associates must also be of the same type (but that type may be different to that of the keys). A table may not contain duplicate keys. If `k` is a key of the table `t`, then `t[k]` gives the associate corresponding to `k`. New entries can be made, or existing entries modified, by putting the associate value in the table after selecting with the key value, as in `PUT a IN t[k]`. Entries can be deleted with the `DELETE` command, as in `DELETE t[k]`. The ordering is again lexicographic.

2. SYNTAX DESCRIPTION METHOD

The syntax of *B* is given in the following form: each rule starts with the name of the thing being defined followed by a colon; following this are one or more alternatives, each marked with a • in front. Each alternative is composed of symbols that stand for themselves, or the names of other rules. These other rules are then defined elsewhere in the grammar, or possibly in the same rule. As an example, here is a simple grammar for a small part of English:

```
sentence:
• declarative
• declarative , connective sentence
```

```
declarative:
• collective-noun verb collective-noun
• collective-noun do not verb collective-noun
```

```
collective-noun:
• cats
• dogs
• people
• the police
```

verb:
- love
- hate
- eat
- hassle

connective:
- and
- but
- although
- because
- yet

This produces sentences like:

```
dogs do not love the police
the police hassle dogs
cats do not hate cats , but cats hate dogs ,
     because dogs hate cats
people eat dogs , yet dogs love people
```

You will notice that the names of rules are in a different typeface to words that stand for themselves. In the grammar of *B* that follows, furthermore, rule names are all in lower-case letters, while words that stand for themselves are all in upper-case letters, so they are easily distinguished.

It often happens that a part of an alternative is optional. There is a special rule for this:

empty:
-

(Empty produces nothing.)

optional-ANYTHING:
- empty
- ANYTHING

The "optional" rule is included to save many rules in the definition, and stands for rules like:

optional-comment:
- empty
- comment

3. REPRESENTATIONS

new-line:
- optional-comment new-line-proper indent

A *B* program consists of indented lines. A new-line-proper marks a transition to a new line. An indent stands for the left margin blank offset. Initially, the left margin has zero width. The indentation is increased by an increase-indentation and decreased again by a decrease-indentation. These always come in pairs and serve for grouping, just as BEGIN-END pairs do in other programming languages. An increase-indentation is always preceded by a line ending with a colon (possibly followed by comment).

comment:
- optional-new-line-proper optional-spaces \ comment-body optional-further-comment

further-comment:
- new-line-proper optional-spaces \ comment-body optional-further-comment

spaces:
- space optional-spaces

Comments may be placed at the end of a line or may stand alone on a line. No comment may precede the first line of a unit (see section 4).
A comment-body may be any sequence of printable characters.

Example comment:

```
\modified 6/4/84 to reject passwords of length < 6
```

Keywords are composed of capital letters (`A` to `Z`), digits, and quotes (`'` and `"`), and must start with a letter. For example, `A3'B"` is a keyword.

Tags are composed of lower-case letters (`a` to `z`), digits, and quotes (`'` and `"`), and must start with a letter. For example, `a3'b"` is a tag.

Some other signs are composite: `..`, `**`, `*/`, `/*`, `^^`, `<<`, `><`, `>>`, `<=`, `<>` and `>=`. Spaces are freely allowed between symbols, but not within keywords, tags, numeric-constants and composite signs. Sometimes spaces are required

to separate keywords and tags from following symbols. For example, `cos y` is not the same as `cosy`: the latter is taken to be one tag.

4. UNITS

Units are the building blocks of a *B* "program". You can define new commands, functions and predicates by writing a unit. These units reside in a work-space.

unit:
- how-to-unit
- yield-unit
- test-unit

unit-body:
- optional-share-heading
  command-suite
  optional-refinement-suite
  optional-comment

share-heading:
- new-line SHARE identifier optional-share-heading

Tags used as targets (variables) in a unit (except those that are formal-parameters) are by default local to the unit. If a target should be shared between several units, this can be indicated by listing the tag in a share-heading at the start of the unit body. It stands then for a global target of the work-space. The global targets together with their contents are also called the "permanent environment", because they survive on logging out.

The execution of a yield- or test-unit cannot alter the values of shared targets or delete or create new shared targets so that the changes survive after the execution of the unit (in other words such units cannot have "side-effects"). If the unit appears to modify a shared target, it effectively modifies a local "scratch-pad" *copy* of that target, and the change is invisible after the execution of the unit.

Example share-heading

```
SHARE name'list, abbreviation'table
```

refinement-suite:
- new-line refinement optional-refinement-suite

When writing a unit, the specification of some parts (commands, expressions and tests) may be deferred by using a "refinement". These refinements are then specified at the end of the unit.

4.1. HOW-TO-UNITS

A how-to-unit defines the meaning of a new command (see "user-defined-commands", section 5.1.16, for how to execute them). Once the command has been defined, it may be used in the same way as the built-in commands. Other user-defined commands may be used in the body of a unit even if they have not yet been defined, though they must be defined by the time the unit is invoked.

how-to-unit:
- `HOW'TO` formal-user-defined-command :
  unit-body

formal-user-defined-command:
- keyword optional-formal-parameter optional-formal-trailer

The first keyword of a formal-user-defined-command must be unique, i.e., different from the first keywords of all predefined and other user-defined commands. So it is impossible to redefine the built-in commands of *B*. It may also not be `HOW'TO`, `YIELD`, `TEST`, `SHARE` or `ELSE`. Otherwise, it may be chosen freely. There are no restrictions on the second and further keywords.

formal-trailer:
- keyword optional-formal-parameter optional-formal-trailer

formal-parameter:
- tag

Note that, although actual-parameters (section 5.1.16) and formal-operands (section 4.2) may be composite, formal-parameters must be simple tags.

Example how-to-unit:

```
HOW'TO PUSH value ON stack:
   PUT value IN stack[#stack+1]
```

See also: quit-command (5.1.11), user-defined-commands (5.1.16).

4.2. YIELD-UNITS

A yield-unit defines the meaning of a new function (see "Formulas with user-defined functions", section 6.1.6, for how to execute them).
Functions may be zeroadic (no operands), monadic (one trailing operand) or dyadic (two operands, one at the left and one at the right). They return a value with the RETURN command (section 5.1.12).

yield-unit:
- YIELD formal-formula :
  unit-body

formal-formula:
- formal-zeroadic-formula
- formal-monadic-formula
- formal-dyadic-formula

formal-zeroadic-formula:
- tag

formal-monadic-formula:
- tag formal-operand

formal-dyadic-formula:
- formal-operand tag formal-operand

Functions must not be "overloaded" (multiply defined). However, a given tag may be used at the same time for a dyadic function and either a zeroadic or a monadic function or predicate. (In other words, you may not have a function that is both monadic and zeroadic, for otherwise it would be impossible to decide what was meant in cases such as `f + 1`, where `f` could be either zeroadic or monadic; this restriction also applies to combinations of functions and predicates.)

formal-operand:
- single-identifier

Example yield-unit:

```
YIELD (a, b) over (c, d):
   PUT c*c+d*d IN rr
   RETURN (a*c+b*d)/rr, (-a*d+b*c)/rr
```

See also: return-commands (5.1.12), formulas with user-defined functions

(6.1.6).

4.3. TEST-UNITS

A test-unit defines the meaning of a new predicate (see "Propositions with user-defined predicates", section 6.3.2, for how to execute them). Like functions, predicates may be zeroadic, monadic or dyadic.
Tests do not return a value, but succeed or fail via the REPORT, SUCCEED and FAIL commands.

test-unit:
- TEST formal-proposition :
  unit-body

formal-proposition:
- formal-zeroadic-proposition
- formal-monadic-proposition
- formal-dyadic-proposition

formal-zeroadic-proposition:
- tag

formal-monadic-proposition:
- tag formal-operand

formal-dyadic-proposition:
- formal-operand tag formal-operand

Like functions, predicates must not be "overloaded", though a given tag may be used at the same time for a dyadic predicate and either a zeroadic or a monadic function or predicate.

Example test-unit:

```
TEST a subset b:
   REPORT EACH x IN a HAS x in b
```

See also: report-commands (5.1.13), succeed-command (5.1.14), fail-command (5.1.15), propositions with user-defined predicates (6.3.2).

4.4. REFINEMENTS

Refinements support the method of "top-down" programming, also known as programming by "stepwise refinement". The body of a unit may be written using refined-commands, -expressions and -tests that reflect the appropriate coarse-grained level of the algorithm. In subsequent refinements, these may be refined to the necessary detail, possibly in several steps. As with units, there are three kinds of refinements. The differences with units are:
— refinements are bound to a unit and may not be invoked from other units;
— the tags known inside the unit are also known inside the refinement;
— no parameters or operands can be passed when the refinement is invoked.

refinement:
- command-refinement
- expression-refinement
- test-refinement

command-refinement:
- keyword : command-suite

The keyword of a command-refinement must be different from the first keywords of all predefined commands, and it may also not be `HOW'TO`, `YIELD`, `TEST`, `SHARE` or `ELSE`. It may, however, be the same as the first keyword of a user-defined-command.

Example command-refinement:

```
SELECT'TASK:
   PUT min tasks IN task
   REMOVE task FROM tasks
```

expression-refinement:
- tag : command-suite

Example expression-refinement:

```
stack'pointer:
   IF stack = {}: RETURN 0
   RETURN max keys stack
```

test-refinement:
- tag : command-suite

Example test-refinement:

```
special'case:
   REPORT position+d = line'length
```

See also: refined-commands (5.1.17), refined-expressions (6.1.7), refined-tests (6.3.3).

4.5. COMMAND-SUITES

Command-suites form the bodies of units, refinements, and control-commands.

command-suite:
- simple-command
- increase-indentation optional-command-sequence decrease-indentation

A command-suite may only follow the preceding colon on the same line if it is a simple-command. Otherwise, it starts on a new line, with all lines of the command-suite indented.

command-sequence:
- new-line command optional-command-sequence

The execution of the command-suite of a yield-unit or expression-refinement must end in a return-command, and return-commands may only occur within such command-suites.

The execution of the command-suite of a test-unit or test-refinement must end in a report-, succeed- or fail-command, and these may only occur within such command-suites.

Example command-suite:

```
IF name in keys abbreviation'table:
   PUT abbreviation'table[name] IN name
IF name not'in name'list:
   INSERT name IN name'list
```

The commands of a command-suite are executed one by one, until the last one has been executed or until a terminating command (see section 5.1) is executed.

5. COMMANDS

Commands may be given as "immediate commands", interactively from the keyboard, or may be part of a unit. If commands are given as immediate commands, they are obeyed directly: any targets in the command are then interpreted as global targets from the permanent environment. Within a unit, targets are local, unless they have been listed in a share-heading (see section 4). If the interrupt key is pressed while a command is executing, execution is aborted, and the user is prompted for another immediate command.

command:
- simple-command
- control-command

5.1. SIMPLE-COMMANDS

simple-command:
- check-command
- write-command
- read-command
- put-command
- draw-command
- choose-command
- set-random-command
- remove-command
- insert-command
- delete-command
- terminating-command
- user-defined-command
- refined-command

terminating-command:
- quit-command
- return-command
- report-command
- succeed-command
- fail-command

5.1.1. CHECK-COMMANDS

Check-commands are used to check that a condition is true. They may be used, for example, to check the requirements of parameters or operands on entry to a unit. The liberal use of check-commands helps to get programs correct quickly.

check-command:
- CHECK test

Example check-command:

```
CHECK i >= 0 AND j >= 0 AND i+j <= n
```

When a check-command is executed, its test is tested. If the test fails, an error is reported and execution halts. Otherwise, no message is given and execution continues.

5.1.2. WRITE-COMMANDS

Write-commands are used to write values on the screen. All values in *B* may be written.

write-command:
- WRITE new-liners
- WRITE optional-new-liners expression optional-new-liners

new-liners:
- / optional-new-liners

Example write-commands:

```
WRITE //
WRITE // "Give a value in the range 1 through `n`: "
```

The expression is converted to a text and written on the screen. Each / gives a transition to a new line. Note that you write no comma before or after the /s.

With the exception of adjacent texts, values that are adjacent are written separated by a space. Compounds within other values (within lists, tables or other compounds) are written with commas between their fields, and where necessary, the whole surrounded by brackets. Similarly, inner texts are written enclosed by quotes. Compounds and texts not within other values are output without commas, brackets and quotes. Thus,

```
WRITE 0, 1, ",", 2, "!", "!", 3, {1; 2} /
WRITE {["a","b"]:("b","a"); ["b","a"]:("a","b")} /
```

gives

```
0 1 , 2 !! 3 {1; 2}
{["a", "b"]: ("b", "a"); ["b", "a"]: ("a", "b")}
```

For formatting purposes, see the operators >>, <<, and >< in section 6.1.6, "Functions on Texts", and the conversions in text-displays in section 6.1.5, "Displays".

5.1.3. READ-COMMANDS

Read-commands are used to read input from the user. Values of any type can be read.

read-command:
- READ target EG expression
- READ target RAW

Example read-commands:

```
READ n, s EG 0, ""
READ line RAW
```

The execution of a read-command prompts the user to supply one input line.
If an EG part is present, the input is interpreted as an *expression* of the same type as the expression following EG. (Usually, the example expression will consist of constants, but other expressions are also allowed.) The input expression is evaluated in the permanent environment (so local tags of units cannot be used) and put in the target. To input a text-display (literal), text quotes are required.
If RAW is specified, the target must be a text target. The input line is put in the target literally. No text quotes are needed.
If the user presses the interrupt key instead of supplying a value, the read-command, and in fact the whole program, is aborted. This is useful for entering a sequence of data of unspecified length.

5.1.4. PUT-COMMANDS

Put-commands are the assignment commands of *B*.

put-command:
- PUT expression IN target

Example put-command:

```
PUT a+1, ({}, {1..a}) IN a, b
```

The value of the expression is put in the target. This means that the value will be held in a location for the target, until a different value is put in the target, or the target is deleted. If no such location exists already, it is created on the spot. Here, as in other cases, the types must agree. (In the current implementation this is not checked in general.)

See also: targets (6.2).

5.1.5. DRAW-COMMANDS

Draw-commands are used to obtain a random approximate number.

draw-command:
- DRAW target

Example draw-command:

```
DRAW r
```

A random approximate number from ~0 up to, but not including, ~1 is drawn and put in the target.

5.1.6. CHOOSE-COMMANDS

Choose-commands are used to obtain a random element from a text, list or table.

choose-command:
- CHOOSE target FROM expression

Example choose-commands:

```
CHOOSE guess FROM {0..99}
CHOOSE answer FROM {"yes"; "no"}
CHOOSE exit FROM exits[current'room]
```

The expression must have a text, list or table as value. This value must not be empty. An item is drawn at random from the value (characters from a text, entries from a list and associates from a table) and put in the target. The item is not removed from the value.

5.1.7. SET-RANDOM-COMMANDS

Set-random-commands are used to start or re-start the random sequence used for draw- and choose-commands.

set-random-command:
- SET'RANDOM expression

Example set-random-command:

```
SET'RANDOM "Monte Carlo", run
```

The (pseudo-)random sequence used for draw- and choose-commands is reset to a point, depending on the value of the expression (how this is done is not further specified). Each *B*-session starts this sequence at some random point.

5.1.8. REMOVE-COMMANDS

Remove-commands are used to remove an entry from a list.

remove-command:
- REMOVE expression FROM target

Example remove-command:

```
REMOVE task FROM tasks
```

The target must hold a list, and the value of the expression must be an entry of that list. The entry is removed. If it was present more than once, only one instance is removed.

5.1.9. INSERT-COMMANDS

Insert-commands are used to insert an element in a list.

insert-command:
- INSERT expression IN target

Example insert-command:

```
INSERT new'task IN tasks
```

The target must hold a list. The value of the expression is inserted as a list entry. If that entry was already present, one more instance will be present.

5.1.10. DELETE-COMMANDS

Delete-commands are used to delete table entries and other (non-trimmed) targets. They can be used as immediate commands to delete unwanted permanent targets.

delete-command:
- DELETE target

Example delete-command:

```
DELETE t[i], u[i, j]
```

The location for the target ceases to exist. If a multiple-target is given, all its single-targets are deleted. If a table-selection-target is given, the table must contain the key that is used as selector. The table entry with that key is then deleted from the table. It is an error to delete a trimmed-text-target (e.g., `t@2`).

Note that the meaning of `DELETE t[i], t[j]` is well defined, even if `i` and `j` have the same value.

5.1.11. QUIT-COMMAND

A quit-command is used for exit from how-to-units or command-refinements, or to terminate a *B* session.

quit-command:
- QUIT

A quit-command may only occur in the command-suite of a how-to-unit or command-refinement, or as an immediate command.

Example quit-command:

```
QUIT
```

The execution of a quit-command causes the termination of the execution of the how-to-unit or command-refinement in whose command-suite it occurs. If it occurs in a command-refinement, the execution of the invoking refined-

command is thereby terminated and the further execution continues as if the refined-command had terminated normally. Otherwise, the execution of the invoking user-defined-command is terminated and the further execution continues similarly.
Given as an immediate command, QUIT terminates the current session. All units and targets in the permanent environment survive and can be used again at the next session.

5.1.12. RETURN-COMMANDS

Return-commands are used to terminate a yield-unit or expression-refinement, and return a value.

return-command:
• RETURN expression

Example return-command:

```
RETURN (a*c+b*d)/rr, (-a*d+b*c)/rr
```

The execution of a return-command causes the termination of the execution of the yield-unit or expression-refinement in whose command-suite it occurs. The value of the expression is returned as the value of the invoking user-defined function or refined-expression. Return-commands may only occur within the command-suite of a yield-unit or expression-refinement.

5.1.13. REPORT-COMMANDS

Report-commands are used to terminate a test-unit or test-refinement, reporting success or failure.

report-command:
• REPORT test

Example report-command:

```
REPORT i in keys t
```

The execution of a report-command causes the termination of the execution of the test-unit or test-refinement in whose command-suite it occurs. The invoking user-defined predicate or refined-test succeeds/fails if the test of the report-command succeeds/fails. If the invoker is a test-refinement, any bound tags set by a for-command (see section 5.2.4) or a quantification (section 6.3.7) will temporarily survive, as described under REFINED-TESTS (section 6.3.3).
Report-commands may only occur within the command-suite of a test-unit or test-refinement.
The command "REPORT test" is equivalent to

```
SELECT:
    test : SUCCEED
    ELSE: FAIL
```

5.1.14. SUCCEED-COMMAND

A succeed-command is used to terminate a test-unit or test-refinement, reporting success.

succeed-command:
- SUCCEED

Example succeed-command:

```
SUCCEED
```

The execution of a succeed-command causes the termination of the execution of the test-unit or test-refinement in whose command-suite it occurs. The invoking user-defined predicate or refined-test succeeds. As with report-commands, bound tags temporarily survive.
Succeed-commands may only occur within the command-suite of a test-unit or test-refinement.
The command SUCCEED is equivalent to REPORT 0 = 0.

5.1.15. FAIL-COMMAND

A fail-commands is used to terminate a test-unit or test-refinement, reporting failure.

fail-command:
- FAIL

Example fail-command:

```
FAIL
```

The execution of a fail-command causes the termination of the execution of the test-unit or test-refinement in whose command-suite it occurs. The invoking user-defined predicate or refined-test fails. As with report-commands, bound tags temporarily survive.
Fail-commands may only occur within the command-suite of a test-unit or test-refinement.
The command FAIL is equivalent to REPORT 0 = 1.

5.1.16. USER-DEFINED-COMMANDS

These are used to execute commands defined by how-to-units.

user-defined-command:
- keyword optional-actual-parameter optional-trailer

trailer:
- keyword optional-actual-parameter optional-trailer

actual-parameter:
- identifier
- target
- expression

The keywords and actual-parameters must correspond one to one to those of the formal-user-defined-command of one unique how-to-unit.

Example user-defined-commands:

```
CLEAN'UP
DRINK me
TURN a UPSIDE DOWN
PUSH v ON operand'stack
```

A user-defined-command is executed in the following steps:
1. Any local tags in the how-to-unit that might clash with tags currently in use are systematically replaced by other tags that do not cause conflict.
2. Each actual-parameter is placed between parentheses (and) and then substituted throughout the unit for the corresponding formal-parameter.
3. The command-suite of the unit, thus modified, is executed.

The execution of the user-defined-command is complete when the execution of this command-suite terminates (normally, or because of the execution of a quit-command). After the execution is complete, the local tags of the unit are no longer accessible.

Note that while the *values* of operands are passed to yield- and test-units, the parameters themselves are passed to how-to-units, and evaluated inside the unit, possibly several times, and that values put in targets that are parameters are passed back to the place where the user-defined-command is executed.

See also: how-to-units (4.1), quit-command (5.1.11).

5.1.17. REFINED-COMMANDS

These are used to execute commands defined in command-refinements.

refined-command:
- keyword

The keyword of a refined-command must occur as the keyword of one command-refinement in the unit in which it occurs. That command-refinement specifies the meaning of the refined-command.

Example refined-command:

```
REMOVE'MULTIPLES
```

A refined-command is executed by executing the command-suite of the corresponding command-refinement. The execution of the refined-command is complete when the execution of this command-suite terminates (normally, or because of the execution of a quit-command).

See also: command-refinement (4.4), quit-command (5.1.11).

5.2. CONTROL-COMMANDS

control-command:
- if-command
- select-command
- while-command
- for-command

5.2.1. IF-COMMANDS

If-commands are used to conditionally execute a command-suite depending on the success of a test. If something should be executed on failure too, or there are more alternatives, a select-command should be used.

if-command:
- `IF` test `:` command-suite

Example if-command:

```
IF i < 0: PUT -i, -j IN i, j
```

The test is tested. If it succeeds, the command-suite is executed; if it fails, the command-suite is not executed.
The command "`IF` test `:` command-suite" is equivalent to:

```
SELECT:
    test : command-suite
    ELSE: \do nothing.
```

See also: select-commands (5.2.2).

5.2.2. SELECT-COMMANDS

Select-commands are used to conditionally execute one of a series of command-suites depending on the success of associated tests.

select-command:
- SELECT : alternative-suite

alternative-suite:
- increase-indentation alternative-sequence decrease-indentation

alternative-sequence:
- newline single-alternative optional-alternative-sequence
- newline else-alternative

single-alternative:
- test : command-suite

else-alternative:
- ELSE : command-suite

Example select-commands:

```
SELECT                          SELECT:
    a < 0: RETURN -a                a < 0: RETURN -a
    a >= 0: RETURN a                ELSE: RETURN a
```

The tests of the alternatives are tested one by one, starting with the first and proceeding downwards, until one is found that succeeds. The corresponding command-suite is then executed. ELSE may be used in the final alternative as a test that always succeeds. If all the tests fail, an error is reported.

5.2.3. WHILE-COMMANDS

While-commands are used to repeatedly execute a command-suite depending on the success of a test.

while-command:
- WHILE test : command-suite

Example while-command:

```
WHILE x > 1: PUT x/10, c+1 IN x, c
```

If the test succeeds, the command-suite is executed. If the execution of the command-suite terminates normally, the test is tested a second time, and if it succeeds, the command-suite is executed again, and so on, until the test fails,

or until an escape is forced by a terminating command. If the test fails the very first time, the command-suite is not executed at all.

5.2.4. FOR-COMMANDS

For-commands are used to repeat a command-suite once for each element of a text, list or table.

for-command:
- FOR in-ranger : command-suite

in-ranger:
- identifier IN expression

Example for-commands:

```
FOR i IN {1..10}: WRITE i, i**2 /
FOR k IN keys t: WRITE k, ":", t[k] /
FOR i, j IN keys t: PUT t[i, j] IN t'[j, i]
```

The value of the expression must be a text, list or table. One by one, each item of that value (characters for a text, list entries for a list and associates for a table) is put in the identifier, and each time the command-suite is then executed. For example,

```
FOR c IN "ABC": WRITE "letter is ", c /
```

is equivalent to

```
WRITE "letter is ", "A" /
WRITE "letter is ", "B" /
WRITE "letter is ", "C" /
```

If `t` is a table, then "`FOR a IN t: TREAT a`" treats the associates of `t` in the same way as

```
FOR k IN keys t:
   PUT t[k] IN a
   TREAT a
```

Note that the expression of a for-command is evaluated once. Altering the value of the expression within the command-suite does not alter how often the command-suite is executed.

The tags of the identifier of a for-command may not be used as targets or target-contents outside such a for-command. They are "bound tags", and lose their meaning outside the for-command. There is one exception to this rule: if a for-command is used in a test-refinement, and within the for-command a report-, succeed- or fail-command is executed, the currently bound tags will

temporarily survive as described under REFINED-TESTS (section 6.3.3).

See also: quantifications (6.3.7).

6. EXPRESSIONS, TARGETS AND TESTS

6.1. EXPRESSIONS

In *B*, the evaluation of an expression cannot alter the values of targets that currently exist, nor can it create new targets that survive the expression. If an expression appears to alter a target, it effectively modifies a local "scratch-pad" *copy* of that target, and the change is invisible outside the expression.

expression:
- single-expression
- multiple-expression

single-expression:
- basic-expression
- (expression)

basic-expression:
- simple-expression
- formula

simple-expression:
- numeric-constant
- target-content
- trimmed-text
- table-selection
- display
- refined-expression

tight-expression:
- simple-expression
- zeroadic-formula
- (expression)

right-expression:
- tight-expression
- monadic-formula

Example basic-: simple-: tight-: right-expressions:

```
a       a       a         a
-a                        -a
a+b
                  (a+b)   (a+b)
```

The various kinds of expressions that are distinguished here serve to define the syntax in such a way that no parentheses are needed where the meaning is sufficiently clear.

multiple-expression:
- single-expression , single-expression
- single-expression , multiple-expression

Example multiple-expressions:

```
1, "abc"
(1, 0), (0, 1), (-1, 0), (0, -1)
```

The value of a multiple-expression composed of single-expressions separated by commas is the compound whose fields are the values of the successive single-expressions.

6.1.1. NUMERIC-CONSTANTS

numeric-constant:
- exact-constant
- approximate-constant

exact-constant:
- integral-part optional-fractional-part
- integral-part .
- fractional-part

integral-part:
- digit optional-integral-part

digit:
- 0
- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9

fractional-part:
- . digit
- fractional-part digit

approximate-constant:
- exact-constant exponent-part

exponent-part:
- E optional-plusminus integral-part

plusminus:
- +
- -

Example exact-constants: approximate-constants:

```
666                2.99793E8
666.               2.99793E+8
3.14               1E-9
```

The value of an exact-constant is an exact number. For example, `1.25` stands for the exact number `5/4`. The value of an approximate-constant is an approximate number. The exponent-part gives the power of ten in floating-point notation. For example, `1.2345E2` and `~123.45` are (approximately) the same.

6.1.2. TARGET-CONTENTS

target-content:
- tag

The value of a target-content is the value last put in the target whose name is the given tag.

6.1.3. TRIMMED-TEXTS

These produce a part of a text.

trimmed-text:
- tight-expression `@` right-expression
- tight-expression `|` right-expression

Example trimmed-texts:

```
t@p
t|1
t|q@p
t@p|(q-p+1)
```

The value of the tight-expression must be a text T, and that of the right-expression must be an integer N.
If the sign between the expressions is `@`, then the value of the trimmed-text is that of T after removing the first N−1 characters. For example, `"lamplight"@4 = "plight"`. N must be at least 1 and at most one more than the length of T.
If the sign between the expressions is `|`, then the value of the trimmed-text is the text consisting of the first N characters of T. For example, `"scarface"|5 = "scarf"`. N must be at least 0 and at most equal to the length of T.
Note that the tight-expression itself may be a trimmed-text again. For example, `"department"|6@3 = "depart"@3 = "part"`.

6.1.4. TABLE-SELECTIONS

Table-selections are used to obtain an element from a table.

table-selection:
- tight-expression `[` expression `]`

Example table-selection:

```
t[i, j]
{["yes"]: 1; ["no"]: 0}[answer]
```

The value of the tight-expression must be a table T, and the value of the expression between the square brackets must be a key K of T. The value of the table-selection is then the associate of the table entry in T whose key is K.

6.1.5. DISPLAYS

Displays are used to express values for texts, lists and tables.

display:
- text-display
- list-display
- table-display

text-display:
- ' optional-text-body '
- " optional-text-body "

The text-displays '' and "" stand for the empty text. A text-body may be any sequence of printable characters (see section 1 under 'Texts') and conversions (see below). However, in a text-display in the ' ... ' style, any single quote ' in the text must be written twice to give ''. Otherwise, it would signal the end of the text-display. Similarly, in a text-display in the " ... " style, any double quote " in the text must be written twice to give "". Finally, in either style of text-display, the back-quote ` must also be written twice, giving ``. Otherwise, it signals a conversion.

The quotes and conversion-signs that have to be written twice according to these rules correspond to one character of the resulting text. For example, the number of characters in 'x''y""z' is 6, because it consists of one x, *one* ' character, one y, *two* " characters, and finally one z. Another way to specify the same text is "x'y""""z".

conversion:
- ` expression `

The requirement that some signs be written twice does not hold *inside* a conversion. For example, '`t['a']`' is proper, whereas '`t[''a'']`' is not.

Example text-displays:

```
''
'He said: "Don''t!"'
"He said: ""Don't!"""
'altitude is `a/1E3` km'
```

The value of a text-display is the text composed of the characters given between the enclosing text quotes. If the text-display contains conversions, the expressions of these conversions are evaluated first and converted to a text in the same way as for a write-command. For example, since

```
WRITE 239*4649
```

causes the text `1111111` to be written, the text-display

```
"239 times 4649 gives `239*4649`"
```

is equivalent to

```
"239 times 4649 gives 1111111".
```

list-display:
- { optional-list-body }

list-body:
- list-filler-series
- single-expression .. single-expression

The ambiguity in, e.g., `{1...9}`, is resolved by parsing it as `{1. .. 9}`.

list-filler-series:
- list-filler
- list-filler ; list-filler-series

list-filler:
- single-expression

Example list-displays:

```
{}
{x1; x2; x3}
{1..n-1}
{"a".."z"}
```

The value of `{}` is an empty list. (It may also be an empty table; see below.)
The value of a list-display containing list-fillers is the list whose entries are the values of those list-fillers. If values occur multiply, they give rise to multiple entries in the list.
For a list-display of the form `{p..q}`, p and q must both be integers, or both be characters (texts of length one). The resulting value is then the list of all integers or characters x such that $p \leqslant x \leqslant q$. For example, `{1..4}` = `{1; 2; 3; 4}` and `{"a".."c"}` = `{"a"; "b"; "c"}`.
If $p > q$, the list is empty, but this is only allowed if p and q are adjacent. If there is an intervening integer or character x (such that $p > x > q$), an error is reported.

table-display:
- { optional-table-filler-series }

table-filler-series:
- table-filler
- table-filler ; table-filler-series

table-filler:
- [expression] : single-expression

Example table-displays:

```
{}
{[i, j]: 0}
{[0]: {}; [1]: {0}}
{[name]: (month, day, year)}
```

The table-display {} stands for an empty table. Otherwise, each table-filler gives a table entry with key K and associate A, where K is the value of the expression between square brackets, and A is the value of the single-expression following the colon. The result is then the table containing these table entries. If there are *different* table entries with the same key, an error is reported. Multiple occurrences of the *same* table entry, however, are allowed. The extra occurrences are then simply discarded.

6.1.6. FORMULAS

formula:
- zeroadic-formula
- monadic-formula
- dyadic-formula

zeroadic-formula:
- zeroadic-function

monadic-formula:
- monadic-function actual-operand

dyadic-formula:
- actual-operand dyadic-function actual-operand

The parsing ambiguities introduced by these rules are resolved by priority rules, as follows:

1. If there is no parsing ambiguity (as in `1 + sin x`), no parentheses are needed.
2. If the order makes no difference (as in `a*b*c`, `a*b/c` or `a^b^c`), no parentheses are needed.
3. The five arithmetic functions `**`, `*`, `/`, `+` and `-` have their traditional priority rules:
   `**` comes before `*`, `/`, `+` and `-`;
   `*` and `/` come before `+` and `-`;
   combinations of `+` and `-` are computed from left to right.
   Note, however, that `a**b**c`, `a/b*c` and `a/b/c` are wrong.
4. The function `#` has a high priority, higher than the five arithmetic functions, and the function `~` has a higher priority than all other functions.
5. All other functions, in particular `^`, `^^`, `<<`, `><`, `>>` and all tags (like `sin` or `floor`) have no established priority and may be used
   — having formulas as operands only if these operands are parenthesized (except as in, e.g., `exp -x`, because of point 1 above, or as in `~1>>20` because of point 4);
   — in operands of other formulas only if these operands are parenthesized (except as above).

None of `a/b/c`, `a/b*c` and `sin x+y` is a correct formula. Each of these can be made correct by inserting parentheses, depending on the intention: either `(a/b)/c` or `a/(b/c)`, either `(a/b)*c` or `a/(b*c)`, and either `(sin x) + y` or `sin(x+y)`. Note that because of point 5 above `sin(x)+1` is just as wrong as `sin x + 1`.
The function `#` has been given a high priority since expressions like `#t+1` are so common, that it would be a nuisance to have to parenthesize these, and more so since `#(t+1)` is meaningless anyway. The reason for the high priority of the function `~` is to make `~0`, for example, for all practical purposes behave as a constant.

zeroadic-function:
- tag

monadic-function:
- ~
- +
- -
- */
- /*
- #
- tag

dyadic-function:
- +
- -
- *
- /
- **
- ^
- ^^
- <<
- ><
- >>
- #
- tag

actual-operand:
- single-expression

Example zeroadic-formula: monadic-formula: dyadic-formula:

```
pi          atan(y/x)      x atan y
```

Formulas with user-defined functions

A formula whose function is defined by a yield-unit, is evaluated in the following steps:

1. A copy is made of the current environment (the value of all targets), and all computations during the evaluation of the formula take place in this "scratch-pad copy".
2. Any local tags in the yield-unit that might clash with tags currently in use are systematically replaced by other tags that do not cause conflict.
3. The value of each actual-operand is put in the corresponding formal-operand, used as a (new) target.
4. The command-suite of the unit, thus modified, is executed.

The evaluation of the formula is complete when the execution of this command-suite terminates because of the execution of a return-command; the

value of the formula is the value returned.

Note than in contrast to how-to-units where parameters are evaluated *inside* the unit, the operands to yield-units are evaluated first, and then passed to the unit.

Formulas with predefined functions

A. Functions on numbers

`~x`	returns an approximate number, as close as possible in arithmetic magnitude to `x`.
`x+y`	returns the sum of `x` and `y`. The result is exact if both operands are exact.
`+x`	returns the value of `x`.
`x-y`	returns the difference of `x` and `y`. The result is exact if both operands are exact.
`-x`	returns minus the value of `x`. The result is exact if the operand is exact.
`x*y`	returns the product of `x` and `y`. The result is exact if both operands are exact.
`x/y`	returns the quotient of `x` and `y`. The value of `y` must not be zero. The result is exact if both operands are exact.
`x**y`	returns `x` to the power `y`. The result is exact if `x` is exact and `y` is an integer. If `x` is negative, `y` must be an integer or an exact number with an odd denominator. If `x` is zero, `y` must not be negative. If `y` is zero, the result is one (exact or approximate depending on `x`).
`n root x`	returns the same as `x**(1/n)`.
`root x`	returns the same as `2 root x`, the square root of `x`.
`abs x`	returns the absolute value of `x`. The result is exact if the operand is exact.
`sign x`	returns -1 if `x` is negative, 0 if `x` is zero, and 1 otherwise.
`floor x`	returns the largest integer not exceeding `x` in arithmetic magnitude.
`ceiling x`	returns the same as `- floor -x`.

`n round x` returns the same as `(10**-n)*floor(x*10**n+.5)`. For example, `4 round pi = 3.1416`. The value of `n` must be an integer. It may be negative: `(-2) round 666 = 700`.

`round x` returns the same as `0 round x`.

`a mod n` returns the same as `a-n*floor(a/n)`, that is, the remainder after dividing `a` by `n`. (Both operands may be approximate, and `n` may be negative, but not zero.)

`/*x` returns the "denominator" of `x`, that is, regarding `x` as the fraction `p/q`, the smallest positive integer `q` such that `q*x` is an integer. The value of `x` must be an exact number.

`*/x` returns the corresponding "numerator" with the same sign as `x`, the same integer as `(/*x)*x`. So, if `x` is exact, `x = (*/x)/(/*x)`.

`pi` returns approximately `3.14159265358979...`

`sin x` returns an approximate number by applying the sine function to `x`, with `x` in radians.

`cos x` returns an approximate number by applying the cosine function to `x`, with `x` in radians.

`tan x` returns the same as `(sin x) / (cos x)`.

`x atan y` returns an approximate number `phi`, in the range from (about) `-pi` to `+pi`, such that `x` is approximated by `r * cos phi` and `y` by `r * sin phi`, where `r = root(x*x+y*y)`. The operands must not both be zero.

`atan x` returns the same as `1 atan x`.

`e` returns approximately `2.718281828459...`

`exp x` returns approximately the same as `e**x`.

`log x` returns an approximate number by applying the natural logarithm function (with base `e`) to `x`. The value of `x` must be positive.

`b log x` returns the same as `(log x) / (log b)`, that is, the logarithm with base `b` of `x`.

B. Functions on texts

`t^u` returns the text consisting of `t` and `u` joined. For example, `"now"^"here" = "nowhere"`.

`t^^n` returns the text consisting of `n` copies of `t` joined together. For example, `"Fi! "^^3 = "Fi! Fi! Fi! "`. The value of `n` must be an integer and not negative; if it is zero, the result is the empty text.

`x<<n` returns `x` converted to a text with space characters added to the right until the length is `n`. For example, `123<<6 = "123 "`. In no case is the text truncated; if `n` is too small, the resulting text is as long as necessary. The value of `n` must be an integer, but `x` may be of any type. See write-commands, section 5.1.2, for details about converting values to texts.

`x><n` returns `x` converted to a text with space characters added to the right and to the left, in turn, until the length is `n`. For example, `123><6 = " 123 "`. In no case is the text truncated. The value of `n` must be an integer, but `x` may be of any type.

`x>>n` returns `x` converted to a text with space characters added to the left until the length is `n`. For example, `123>>6 = " 123"`. In no case is the text truncated. The value of `n` must be an integer, but `x` may be of any type.

C. Functions on texts, lists and tables

`keys t` requires a table as operand, and returns a list of all keys in the table. For example, `keys {[1]: 1; [4]: 2; [9]: 3} = {1; 4; 9}`.

`#t` accepts texts, lists and tables. For a text operand, its length is returned, and for a list or table operand, the number of entries is returned (where duplicates in lists are counted).

`e#t` accepts texts, lists and tables for the right operand.
For a text operand, the first operand must be a character, and the number of times the character occurs in the text is returned. For example, `"i"#"mississippi" = 4`.
For a list operand, the number of entries is returned that are equal to the first operand (which must be of the same type as the list entries.) For example, `3#{1; 3; 3; 4} = 2`.
For a table operand, the number of *associates* is returned that are equal to the first operand (which must be of the same type as the associates in the table.) For example, `3#{[1]: 3; [2]: 4; [3]: 3} = 2`.

`min t` accepts texts, lists and tables. For a text operand, its smallest (in the ASCII order, see section 1) character is returned, for a list operand, its smallest entry is returned, and for a table operand, its smallest *associate* is returned. For example, `min "uscule" = "c"`, `min{1; 3; 3; 4} = 1`, and `min{[1]: 3; [2]: 4; [3]: 3} = 3`. The text, list or table must not be empty.
To get the smallest *key* of a table `t`, `min keys t` is used.

`e min t` accepts texts, lists and tables for the right operand.
For a text operand, the first operand must be a character, and the smallest character in the text *exceeding* that character is returned. For example, `"i" min "mississippi" = "m"`.
For a list operand, the smallest entry is returned exceeding the first operand (which must be of the same type as the list entries.) For example, `3 min {1; 3; 3; 4} = 4`.
For a table operand, the smallest associate is returned exceeding the first operand (which must be of the same type as the associates in the table.) For example, `3 min {[1]: 3; [2]: 4; [3]: 3} = 4`.
There must be a character, list entry or table associate exceeding the first operand.

`max t` and `e max t` are like `min`, except that they return the largest element, and in the dyadic case the largest element that is less than the first operand. For example, `"m" max "mississippi" = "i"`.

`n th'of t` requires an integer in `{1..#t}` for the left operand, and accepts texts, lists and tables for the right operand. It returns the `n`'th character, list entry or associate.
In fact, `n th'of t`, for a text `t`, is written as easily `t@n|1`. For a table, it is the same as `t[n th'of keys t]`, which is something different from `t[n]`, unless, of course, `keys t = {1..#t}`. For a list, `1 th'of t` is `min t`.

6.1.7. REFINED-EXPRESSIONS

refined-expression:
- tag

The tag of a refined-expression must occur as the tag of one expression-refinement in the unit in which it occurs.

Example refined-expression:

```
stack'pointer
```

A refined-expression is evaluated in the following steps:
1. A copy is made of the current environment (the value of all targets), and all computations during the evaluation of the expression take place in this "scratch-pad copy".
2. The command-suite of the corresponding expression-refinement is executed.

The evaluation of the refined-expression is complete when the execution of this command-suite terminates because of the execution of a return-command; the value of the refined-expression is the value returned.

See also: expression-refinements (4.4).

6.2. TARGETS

target:
- single-target
- multiple-target

single-target:
- basic-target
- (target)

basic-target:
- tag
- trimmed-text-target
- table-selection-target

For the use of a tag as target, see below under IDENTIFIERS. For other kinds of targets, see below under the appropriate heading.

multiple-target:
- single-target , single-target
- single-target , multiple-target

Example multiple-targets:

```
nn, t2
(x0, y0), (x1, y1), (x2, y2), (x3, y3)
```

If a value is put in a multiple-target (including by commands like CHOOSE and READ), the value must be a compound with as many fields as there are single-targets separated by commas in the multiple-target. The successive fields are then put in the successive single-targets. It is an error if it makes a difference in what order the fields are put in the single-targets (as in `PUT 1, 2 IN x, x` where the final value of `x` might be either `1` or `2`).

Note that the meaning of `PUT a, b IN b, a` is well defined (provided that `a` and `b` are defined and have values of the same type): first the value of the expression `a, b` is determined, and that value is next put in `b, a`. Note also that the meaning of `PUT t[i], t[j] IN t[j], t[i]` is well defined, even if `i` and `j` have the same value. For although in this case a value is put twice in the same target, that value is the same each time, so the order does not matter.

6.2.1. IDENTIFIERS

identifier:
- single-identifier
- multiple-identifier

single-identifier:
- tag
- (identifier)

multiple-identifier:
- single-identifier , single-identifier
- single-identifier , multiple-identifier

Example identifiers:

```
a
(a)
(a, b, (c, d))
a, b, (c, d)
```

single-identifiers:

```
a
(a)
(a, b, (c, d))
```

All identifiers can be used as targets, but the converse is not true. For example,

```
FOR a[1] IN {1..3}: WRITE a
```

is wrong, because `a[1]`, although a target, is not an identifier. If something is put in a target that is a tag, and no location for that tag exists already, it is created first. If the location is created locally (the tag did not occur in an immediate command and was not listed in a share-heading), the location will cease to exist when the current unit is exited. For putting in multiple-identifiers, see multiple-targets in section 6.2.

6.2.2. TRIMMED-TEXT-TARGETS

trimmed-text-target:
- target @ right-expression
- target | right-expression

Example trimmed-text-targets:

```
t@p
t|1
t|q@p
t@p|(q-p+1)
```

The target must hold a text T, and the value of the right-expression must be an integer N.
If the sign used is `@`, then the trimmed-text-target indicates a location consisting of the positions of T starting with the N'th position. N must be at least 1 and at most one more than the length of T. A new text put in the trimmed-text-target replaces the part beginning at this position. For example, after

```
PUT "computer" IN tt
PUT "ass" IN tt@5
```

`tt` will contain the text `"compass"`.
If the sign used is `|`, then the trimmed-text-target indicates a location consisting of the first N characters of T. N must be at least 0 and at most equal to the length of T. A new text put in the trimmed-text-target replaces these characters. For example, after

```
PUT "computer" IN tt
PUT "ne" IN tt|4
```

`tt` will contain the text `"neuter"`.

Note that the target itself may be a trimmed-text-target again. For example, after

```
PUT "computer" IN tt
PUT "m" IN tt@4|1
```

`tt` will contain the text `"commuter"`.
Some special cases: `PUT "" IN t|1` removes the first character of the text in `t`; `PUT "." IN t@(#t+1)` appends a period to the text in `t`.

6.2.3. TABLE-SELECTION-TARGETS

table-selection-target:
- target [expression]

Example table-selection-target:

```
t[i, j]
```

The target must contain a table. The value of the expression is a key K, to be used as selector. For each key in the table, there is a location for the corresponding associate. If K is an existing key of the table, the location for the table-selection-target is that of the associate corresponding to K. If a value A is then put in the table-selection-target, the original associate held in that location is superseded by A. If K is not an existing key and a value A is to be put in the table-selection-target, a new location is created, and the (original) table is made to contain a new table entry consisting of K and A. K must be of the same type as the other keys of the table, and A of the same type as the other associates.

6.3. TESTS

Tests do not return a value, but succeed or fail when tested.
In *B*, the testing of a test cannot alter the values of targets that currently exist, nor can it create new targets that survive the test, with the exception of the temporary survival of bound tags as described under QUANTIFICATIONS (section 6.3.7) and REFINED-TESTS (section 6.3.3). If a test appears to alter an existing target, it effectively modifies a local, "scratch-pad" *copy* of that target, and the change is invisible outside the test.

test:
- tight-test
- conjunction
- disjunction
- negation
- quantification

tight-test:
- (test)
- order-test
- proposition
- refined-test

right-test:
- tight-test
- negation
- quantification

The various kinds of tests that are distinguished here serve to define the syntax in such a way that no parentheses are needed where the meaning is sufficiently clear.

6.3.1. ORDER-TESTS

order-test:
- single-expression order-sign single-expression
- order-test order-sign single-expression

order-sign:
- <
- <=
- =
- <>
- >=
- >

(The order-sign <> stands for "not equals".)

Example order-tests:

```
(i', j') > (i, j)
"0" <= d <= "9"
fa <= f(x) >= fb
```

The single-expressions are evaluated one by one, from left to right, and each adjacent pair is compared. As soon as a comparison does not comply with the given order-sign, the whole order-test fails and no further single-expressions are evaluated. The order-test succeeds if all comparisons comply with the specified order-signs.

An approximate number is *never* equal to an exact number. To compare an approximate number `a` for equality with an exact number `e`, you should write

```
IF a =~ e:
```

This can be used to test if a number is exact or not:

```
SELECT:
    x =~ x: WRITE "Approximate"
    ELSE: WRITE "Exact"
```

6.3.2. PROPOSITIONS

zeroadic-proposition:
- zeroadic-predicate

monadic-proposition:
- monadic-predicate actual-operand

dyadic-proposition:
- actual-operand dyadic-predicate actual-operand

zeroadic-predicate:
- tag

monadic-predicate:
- tag

dyadic-predicate:
- tag

Propositions with user-defined predicates

A proposition whose predicate is defined by a test-unit, is tested in the following steps:

1. A copy is made of the current environment (the value of all targets), and all computations during the testing of the proposition take place in this "scratch-pad copy".
2. Any local tags in the test-unit that might clash with tags currently in use are systematically replaced by other tags that do not cause conflict.
3. Each actual-operand is evaluated and put in the corresponding formal-operand, used as a (new) target.
4. The command-suite of the unit, thus modified, is executed.

The testing of the proposition is complete when the execution of this command-suite terminates because of the execution of a report-, succeed- or fail-command; the proposition succeeds or fails accordingly.

Note that like yield-units, and unlike how-to-units, the operands are evaluated before being passed to the unit.

Propositions with predefined predicates

`e in t` accepts texts, lists and tables for the right operand. It succeeds if `e#t > 0` succeeds, that is, if the value `e` occurs in `t`.

`e not'in t` is the same as `(NOT e in t)`.

6.3.3. REFINED-TESTS

refined-test:
- tag

Example refined-test:

```
special'case
```

A refined-test is tested in the following steps:
1. A copy is made of the current environment (the value of all targets), and all computations during the testing of the test take place in this "scratch-pad copy".
2. The command-suite of the corresponding test-refinement is executed.

The testing of the refined-test is complete when the execution of this command-suite terminates because of the execution of a report-, succeed- or fail-command, and the refined-test succeeds or fails accordingly.
Any bound tags set by a for-command or a quantification (see 6.3.7) at that time will temporarily survive for those parts that are reachable only by virtue of the outcome of the test. This is so that you can turn any test into a refined-test with the same effect.
For example, in

```
        IF divisible AND n > d**2: WRITE d
        ...
    divisible:
        REPORT SOME d IN {2..n-1} HAS n mod d = 0,
```

the bound tag `d` is set to a divisor of `n` if the refined-test succeeds, and since the part `n > d**2` is only reached after success, `d` may be used there. The same is true for the write-command using `d`. However, the line after (indicated with three dots) can be reached if the divisibility test fails, so there `d` has ceased to exist and may not be used.

See also: test-refinements (4.4).

6.3.4. CONJUNCTIONS

conjunction:
- tight-test AND right-test
- tight-test AND conjunction

Example conjunctions:

```
a > 0 AND b > 0
i in keys t AND t[i] in keys u AND u[t[i]] <> "dummy"
```

The tests of the conjunction, separated by AND, are tested one by one, from left to right. As soon as one of these tests fails, the whole conjunction fails and no further parts are tested. The conjunction succeeds if all its tests succeed.

6.3.5. DISJUNCTIONS

disjunction:
- tight-test OR right-test
- tight-test OR disjunction

Example disjunctions:

```
a <= 0 OR b <= 0
n = 0 OR s[1] = s[n] OR t[1] = t[n]
```

The tests of the disjunction, separated by OR, are tested one by one, from left to right. As soon as one of these tests succeeds, the whole disjunction succeeds and no further parts are tested. The disjunction fails if all its tests fail.

6.3.6. NEGATIONS

negation:
- NOT right-test

Example negation:

```
NOT a subset b
```

A negation succeeds if its right-test fails, and fails if that test succeeds.

6.3.7. QUANTIFICATIONS

Quantifications are easy ways of finding out if a test is true for no elements, or at least one, or every element of a text, list or table.

quantification:
- quantifier ranger HAS right-test

quantifier:
- SOME
- EACH
- NO

ranger:
- in-ranger
- parsing-ranger

(For in-rangers, see for-commands, section 5.2.4.)

parsing-ranger:
• multiple-identifier PARSING expression

Note that the identifier of a parsing-ranger must be a multiple-identifier (like p, q, r): it may not be a single-identifier (like pqr). Moreover, each of the single-identifiers (like p) must be plain tags. The reason is that this determines the number of parts which the value of the expression must be split into (see below).

Example quantifications:

```
SOME p, q, r PARSING line HAS q in {". "; "? "; "! "}
EACH i, j IN keys t HAS t[i, j] = t[j, i]
NO d IN {2..n-1} HAS n mod d = 0
```

The tags of the identifier of a quantification are "bound tags", and so may not be used outside the quantification except as described below.

The meaning of quantifications will first be described for the case of SOME ... IN ... .
The value of the expression must be a text, list or table. The items (characters, list entries or associates) of that value are assigned one by one to the identifier, and the right-test is tested each time. The quantification succeeds as soon as the right-test succeeds once. It fails only if the text, list or table is exhausted and the right-test has failed each time.
If the quantification succeeds, the bound tags set at that moment will temporarily survive and may be used in those parts that are reachable only by virtue of the outcome of the test.
For example, in

```
IF (SOME d IN {2..n-1} HAS n mod d = 0) AND n > d**2:
   WRITE d
...
```

the bound tag d is set to a divisor of n if the quantification succeeds, and since the part n > d**2 is only reached after success, d may be used there. The same is true for the write-command using d. So, if n has the value 77, 7 will be written, since the test n mod d = 0 succeeds the first time when d is set to 7 (and 77 > 7**2). However, the line after (indicated with three dots) can be reached if the divisibility test fails, so there d has ceased to exist and may not be used.

The meaning of a quantification SOME id IN tlt HAS prop can also be described as the meaning of the refined-test test'if'some, given a test-

refinement

```
test'if'some:
   FOR id IN tlt:
       IF prop: SUCCEED
   FAIL
```

The meaning of `EACH id IN tlt HAS prop` is the same as that of `NOT SOME id IN tlt HAS NOT prop`. In other words, an `EACH` quantification succeeds only if its right-test succeeds each time.
The meaning of `NO id IN tlt HAS prop` is the same as that of `NOT SOME id IN tlt HAS prop`. In other words, a `NO` quantification succeeds only if its right-test fails each time.
The rules for temporary survival are the same as for `SOME`. So an `EACH` or `NO` quantification will only have set its bound tags on failure. Thus, in the following, the bound tag `d` survives into the `ELSE`:

```
SELECT:
   NO d IN {2..n-1} HAS n mod d = 0: WRITE "prime"
   ELSE: WRITE "divisible by `d`"
```

If `PARSING` is specified, all *parsings* of the value of the given expression are tried, instead of its items. The value of the expression must be a text. A "parsing" of a text is a way of splitting it in parts. The text is split in all possible ways in as many parts as there are tags in the multiple-identifier, and each split is put in that identifier, whereupon the right-test is tested. For example,

```
SOME p, q, r PARSING "abracadabra" HAS (p=r AND #p>3)
```

will succeed with `p` and `r` set to `"abra"` and `q` set to `"cad"`. If the test `#p>3` is omitted, the quantification will succeed with the uninteresting result that `p` and `r` are set to `""` and `q` to `"abracadabra"`.
To give another example,

```
   PUT "a man, a plan, a canal: panama!" IN palin
   WHILE SOME hd, x, tl PARSING palin HAS x'non'let:
       PUT hd^tl IN palin
   WRITE palin /
x'non'let: REPORT #x = 1 AND x not'in {"a".."z"}
```

will successively find and remove all non-letters from the text in `palin`, finally leaving the text `"amanaplanacanalpanama"`. (This is not a recommended way, because it will be very slow. There are equally simple and much faster ways to achieve the same effect. The example is only chosen to illustrate the possibilities of `PARSING`.) Note that the test `#x = 1` here is essential. If it is omitted, the program will go into an endless loop "removing" empty texts

`x` from `palin`.

The meaning of `SOME p, q, ... PARSING whole HAS prop` may more precisely be described as follows. Let `parsings` stand for a list, containing *all* compounds with the same number of fields as the multiple-identifier `p, q, ...`, such that those fields (which are texts) joined together give the text `whole`. For example, in

```
SOME p, q, r PARSING "abracadabra" HAS (p=r AND #p>3)
```

the list `parsings` will begin with

```
{("", "", "abracadabra"); ("", "a", "bracadabra"); ... ,
```

contain somewhere in the middle

```
... ; ("abra", "cad", "abra"); ... ,
```

and end with

```
... ; ("abracadabr", "a", ""); ("abracadabra", "", "")}.
```

The effect of the quantification is then the same as that of

```
SOME p, q, ... IN parsings HAS prop.
```

The meaning of `EACH` or `NO` is accordingly defined.

See also: for-commands (5.2.4).