Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

L.G.L.T. Meertens, S. Pemberton

Description of B

Department of Computer Science Note CS-N8405 July

DESCRIPTION OF B

LAMBERT MEERTENS, STEVEN PEMBERTON
Centre for Mathematics and Computer Science, Amsterdam

B is a simple but powerful new programming language designed for use in personal computing. This report is intended as a reference book for the users of *B*, though it will also be useful for experienced programmers who want to learn *B*.

1980 MATHEMATICS SUBJECT CLASSIFICATION: 69D41
KEY WORDS & PHRASES: Programming Languages; B.

Note CS-N8405
Centre for Mathematics and Computer Science,
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

DESCRIPTION OF B

CONTENTS

0. INTRODUCTION

B is a simple but powerful new programming language, designed for use in personal computing. (Note: the name "*B*" is only a temporary working title, and the new language bears no relation to the predecessor of C.) The foremost aim in the design of *B* has been the ease of use for programmers who want to produce working programs without having to master a complex tool. An implementation of *B* is available from the *B* group at the CWI, currently only under UNIX* or its look-alikes, but soon (scheduled early 1985) also for the IBM-PC under MS-DOS. Remarks concerning the implementation appear in this description between double braces {{ and }}.

This description of *B* originated from a text, prepared by the first author, for use in teaching *B* during the Fall term of 1982 at New York University. The aim is to provide a reference book for the users of *B* that is more accessible than the somewhat formal "Draft Proposal" [2]. While it is not a text book, it should also be useful to people who already have ample programming experience and want to learn *B*. A text book for beginners is also available from the CWI [1].

In this description we have tried to remain close to the Draft Proposal in order to facilitate cross-referencing. To this end, all section numbers from section 4 onwards are the same as in the Draft Proposal. However there are some changes in terminology. Some minor differences are

Draft Proposal:	This description:
textual-display	text-display
textual-body	text-body
LIST-body	optional-list-body
TABLE-body	optional-table-filler-series

But the main difference is perhaps in the treatment of "collateral":

Draft Proposal:	This description:	Examples		
collateral-expression	expression	`a`	`(a, b)`	`a, b`
TYPE-expression	single-expression	`a`	`(a, b)`	
TYPE-TYPES-expression	multiple-expression			`a, b`

Whereas these are purely descriptional differences, there are also a few differences in content. Where the Draft Proposal has the keyword `ALLOW`, *B* now has the keyword `SHARE`; a command `READ` ... `RAW` has been added; and approximate-constants may no longer consist of just an exponent-part: `E-1` must now be written `1E-1`.

References

[1] *Computer Programming for Beginners, Introducing the B Language, Part 1*,
Leo Geurts, CWI, Amsterdam, 1984

[2] *Draft Proposal for the B Programming Language*,
Lambert Meertens, CWI, Amsterdam, 1981

* Unix is a trademark of Bell Laboratories.

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 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 on the lines 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 other 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:
●
```

```
optional-ANYTHING:
● empty
● ANYTHING
```

The "optional" rule is included to save many rules in the definition. For example, it stands for a rule

```
optional-comment:
● empty
● comment
```

and similar rules. (Empty produces an empty result.)

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 ″). A keyword 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 ″). A tag 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

```
unit:
● how-to-unit
● yield-unit
● test-unit
```

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

```
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

```
how-to-unit:
● HOW'TO formal-user-defined-command :
        command-suite
        optional-refinement-suite
```

```
formal-user-defined-command:
● keyword optional-formal-tail
```

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-tail:
● formal-parameter optional-formal-trailer
● formal-trailer
```

```
formal-trailer:
● keyword optional-formal-tail
```

```
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]
```

A how-to-unit defines the meaning of a new command (see “user-defined-commands”, section 5.1.16). The above unit defines a `PUSH ... ON ...` command. 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.

See also: quit-command (5.1.11), share-heading (4.5), user-defined-commands (5.1.16).

4.2. YIELD-UNITS

yield-unit:
- `YIELD` formal-formula :
  command-suite
  optional-refinement-suite

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

formal-zeroadic-formula:
- zeroadic-function

formal-monadic-formula:
- monadic-function formal-operand

formal-dyadic-formula:
- formal-operand dyadic-function formal-operand

Functions must not be "overloaded" (multiply defined), and a user-defined function must be represented by a tag. 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, and the restrictions also apply 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
```

A yield-unit defines the meaning of a new function (see "Formulas with user-defined functions", section 6.1.6). The example given above defines complex division. (Complex numbers are not a built-in type of *B*.)
Functions may be zeroadic (no operands), monadic (one trailing operand) or dyadic (two operands, one at the left and one at the right).

See also: return-commands (5.1.12), share-headings (4.5), formulas with user-defined functions (6.1.6).

4.3. TEST-UNITS

test-unit:
- `TEST` formal-proposition :
  command-suite
  optional-refinement-suite

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

formal-zeroadic-proposition:
- zeroadic-predicate

formal-monadic-proposition:
- monadic-predicate formal-operand

formal-dyadic-proposition:
- formal-operand dyadic-predicate 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
```

A test-unit defines the meaning of a new predicate (see "Propositions with user-defined predicates", section 6.3.2). 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.

See also: report-commands (5.1.13), succeed-command (5.1.14), fail-command (5.1.15), share-headings (4.5), propositions with user-defined predicates (6.3.2).

4.4. REFINEMENTS

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
```

Refinements support the method of "top-down" programming, also known as programming by "step-wise refinement". The body of a unit may be written using refined-commands, -expressions and -tests that reflect the appropriate, coarse-grained, level of abstraction for expressing the algorithmic intention. 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;
— all tags known inside the unit are also known inside the refinement;
— no parameters or operands can be passed when the refinement is invoked.

{{Currently, refinements may only occur within unit bodies, and not in "immediate commands".}}

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

4.5. COMMAND-SUITES

command-suite:
● simple-command
● increase-indentation optional-share-heading 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.

Example command-suite

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

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.
A share-heading may only occur in the command-suite of a unit (and not of a refinement or compound-command).

Example share-heading

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

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-sequence

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

5. COMMANDS

Commands may be given as "immediate commands", directly from the terminal, 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 above).
If the user presses the interrupt key 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-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. Check-commands 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.

5.1.2. WRITE-COMMANDS

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

new-liners:
- / optional-new-liners

Examples of write-commands:

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

The expression is converted to a text and written to 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
WRITE {1; 2}, {['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-command:
- READ target EG expression
- READ target RAW

Examples of 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-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. {{This is currently not checked in general.}} See also the sections on various kinds of targets below (section 6.2).

5.1.5. DRAW-COMMANDS

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-command:
● CHOOSE target FROM expression

Example choose-command:

```
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-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.

5.1.8. REMOVE-COMMANDS

```
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-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-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`).

5.1.11. QUIT-COMMAND

```
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-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-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

```
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

```
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

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.

Examples of 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.

5.1.17. REFINED-COMMANDS

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).

5.2. CONTROL-COMMANDS

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

5.2.1. IF-COMMANDS

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.
(If something should be executed on failure too, or there are more alternatives, you should use a select-command instead.)
The command " IF test : command-suite" is equivalent to:

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

5.2.2. SELECT-COMMANDS

select-command:
- SELECT: alternative-suite

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

alternative-sequence:
- single-alternative
- else-alternative
- single-alternative new-line alternative-sequence

single-alternative:
- test : command-suite

else-alternative:
- ELSE: command-suite

Examples of 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-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, and the while-command is repeated, 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-command:
● FOR in-ranger : command-suite

in-ranger:
● identifier IN expression

Example for-command:

```
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 the command-suite 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
```

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

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

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

Examples of 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

Examples of 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
- integral-part digit

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:
- +
- -

Examples of 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, because `123.45 = 1.2345*10**2`.

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

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

Examples of 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-selection:
- tight-expression [expression]

Example table-selection:

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

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

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 will 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, the back-quote ` must be written twice too, giving ``. Otherwise, it signals a conversion.

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.

Examples of 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'.
```

The quotes and conversion-signs that had to be written twice according to the above 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".

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

Examples of 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

Examples of 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

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`, in spite of what other programming languages might lead you to expect.
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

Examples of 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 will 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. 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 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.

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).

`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 `3.1415926535`...
`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 `2.7182818284`...
`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.
`x<<n`	converts `x` to a text and adds space characters 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`	converts `x` to a text and adds space characters 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`	converts `x` to a text and adds space characters 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) 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`, use `min keys t`.
`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 will 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

Examples of multiple-targets:

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

If a value is put in a multiple-target, 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. 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`), an error is reported {{but this is currently not checked}}.
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

Examples of identifiers:	single-identifiers:
`a`	`a`
`(a)`	`(a)`
`(a, b, (c, d))`	`(a, b, (c, d))`
`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
```

Examples of 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. 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. 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 useful 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".)

Examples of 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.

Note carefully that an approximate number is *never* equal to an exact number, so, for instance, if you want to compare an approximate number `a` for equality with an exact number `e`, you should write

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

This also allows you 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 will 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.

Propositions with predefined predicates

`e in t`	accepts texts, lists and tables for the right operand. It succeeds if `e#t > 0` succeeds, in other words, 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 will 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`. The line after (indicated with three dots), however, can be reached if the divisibility test fails. So there `d` has ceased to exist.

See also: test-refinements (4.4).

6.3.4. CONJUNCTIONS

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

Examples of 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

Examples of 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

quantification:
- quantifier ranger HAS right-test

quantifier:
- SOME
- EACH
- NO

ranger:
- in-ranger
- parsing-ranger

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).

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

Examples of 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 quantifier may not be used as targets or target-contents outside such a quantifier. They are "bound tags", and lose their meaning outside the quantifier, 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). The line after (indicated with three dots), however, 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 palindrome
    WHILE SOME hd, x, tl PARSING palindrome HAS x'non'letter:
        PUT hd^tl IN palindrome
    WRITE palindrome /
x'non'letter: REPORT #x = 1 AND x not'in {'a'..'z'}
```

will successively find and remove all non-letters from the text in palindrome 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 palindrome.
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).

INDEX