DRAFT PROPOSAL FOR THE *B* PROGRAMMING LANGUAGE

SEMI-FORMAL DEFINITION

LAMBERT MEERTENS

MATHEMATICAL CENTRE 1981

DRAFT PROPOSAL FOR THE *B* PROGRAMMING LANGUAGE

SEMI-FORMAL DEFINITION

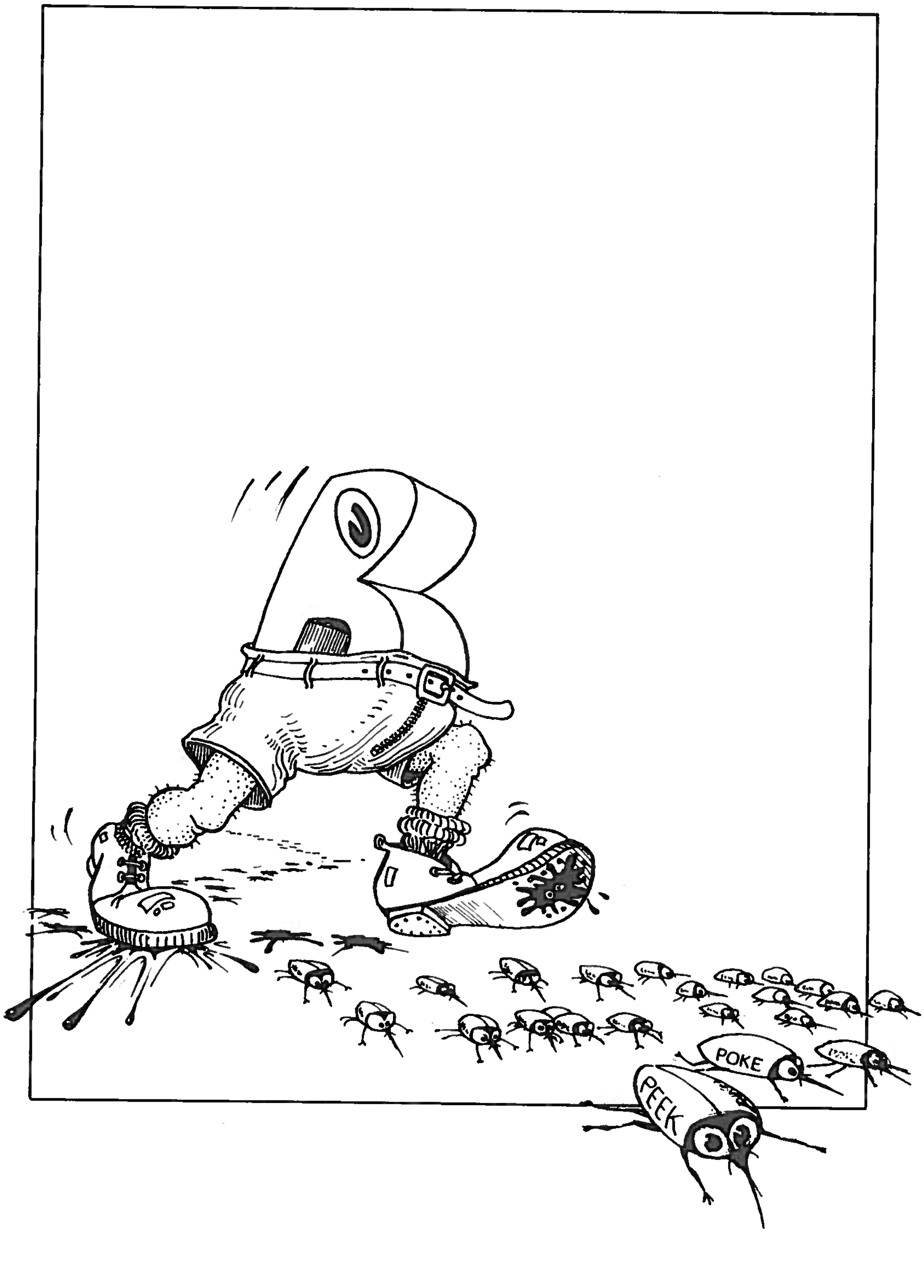
LAMBERT MEERTENS

MATHEMATISCH CENTRUM AMSTERDAM 1981

Printed at the Mathematical Centre, 413 Kruislaan, Amsterdam.

The Mathematical Centre, founded the 11th of February 1946, is a non-profit institution aiming at the promotion of pure mathematics and its applications. It is sponsored by the Netherlands Government through the Netherlands Organization for the Advancement of Pure Research (Z. W. O.).

ACM—Computing Reviews—category: 4.22
ISBN 90 6196 238 2


POKE
PEEK

Contents

QUICK REFERENCE

Numbers are exact or approximate. You get an exact number even if you use *3.14* or *22/7*. You get an approximate number if you use *E* for the ten power, or if you use the ~ function (pronounced “about”). For example, *~1000 = E3*, and *~0.005 = 0.5E−2*. You may also write *~(a+b)* etc.
Warning: an approximate number is *never* equal to an exact number. If you want to test if you may divide by *x*, and if you are not very sure that *x* is exact, it is not safe to use the test *x <> 0* (which is shorthand for *(x < 0 OR x > 0)*). You should use *~x <> ~0*.
If functions like +, −, *, / and ** work on exact numbers, the result is also computed exactly, except if the exponent *n* in x**n is a fraction. (A formula like a*x**2+b*x+c stands for what is usually written as ax^2+bx+c: your computer cannot stand dancing lines and requires that you write * whenever you mean multiplication, even in cases like 2*x.) Arithmetic on approximate numbers gives approximate results (which, for many purposes, are precise enough, and often are computed much faster). Functions like *root*, *sin* and *log* always give an approximate result. (So *root 4 <> 2* and *log 1 <> 0*). More details are given at the functions below.

Texts consist of characters and are written like *'Jack and Jill'* or *"Jack and Jill"*. (The characters meant are not Jack and Jill, but the “*J*”, “*a*”, etc. You may use any printing character and the space.) Which of the forms you use, the one with single quotes or the one with double quotes, makes no difference to your computer. Never confuse the number *747* with the text *'747'*. Whereas 747 = 3*249, *'747'* is quite another text than the text '3*249', and '3'*'249' is not even a text; to your computer it is meaningless. The number *747* can be used to do arithmetic; to your computer it does not consist of characters and it is written that way only because the dominant earthian species has twice five wriggly appendices sprouting from its upper tentacles and finds this clumsy notation convenient, and because you are (presumably) a member of that species and your computer tries to please you. The text *'747'*, on the other hand, cannot be used in arithmetic, and if you nevertheless try to do so, your computer will warn you. It really is three characters in a row. The so-called quotes on the outside do not really count. They only serve to make clear where the text begins and ends. If you say prayers, it does not mean that you say “prayers”. But if you say “prayers”, you don’t say the quotes, do you? You can find out the length of a text with the function #. For example, # *'toe' = 3*. If you use *'* before and after your text, you can only use it inside if you double it thus: *''*. Your computer knows that you really mean it only once: # *'p''q' = 3*. The rules for *"* are similar.
But if you use the other quote sign inside than the one you use on the outside, you should not double it. So write either: *'He said: "don''t!"'* or: *"He said: ""don't!"""*.

Inside texts, you can use weirdos (which are known as conversions) of the form \`e\`. Your computer computes the value and replaces the conversion by a suitable text. For example, if *i = 239* and *j = 4649*, then '\`i\` * \`j\` = \`i*j\`' = '239 * 4649 = 1111111'. Within the conversions the need to double the outside quotes inside has disappeared: '\`#'toe'\`' = '3'. (Don't look too long at it if you don't want to strain your eyes.) On the other hand, if you use a single \` as character in a text, you have to double it.
You can join two texts thus: *'now'^'here' = 'nowhere'*, and you can repeat a text as many times as you want: *'ox'^^3 = 'ox'^'ox'^'ox' = 'oxoxox'* (just like x**3 = x*x*x). You can take texts apart thus: *'lamplight'@4 = 'plight'* (since the "*p*" is the fourth character) and *'scarface'|5 = 'scarf'*.
You may combine @ and |: *'Benedictine'@4|5 = 'edictine'|5 = 'edict'*, and *'Benedictine'|8@4 = 'Benedict'@4 = 'edict'*.
Forms with @ and | may be used as targets:
if *t* has as content *'Benedictine'*, and you tell your computer to

```
PUT 'zedr' IN t@4|5
```

it puts *'Benzedrine'* in *t*; if *t* is *'participle'* and you tell your computer to

```
PUT '' IN t|8@7
```

it puts *'particle'* in *t*; and if *t* is *'creation'* and you tell your computer to

```
PUT 'm' IN t@4|0
```

or to

```
PUT 'm' IN t|3@4
```

it puts *'cremation'* in *t*.

Compounds are a bunch of values grouped together. For example, if you want to keep track of which books you have lent when to whom of your friends, you may tell your computer to

```
PUT 'N&P', 'Mote' IN book
PUT 84, 3, 17 IN date
INSERT book, date, 'bearded gnome' IN books'lent
```

and your computer inserts *(('N&P', 'Mote'), (84, 3, 17), 'bearded gnome')* in the list of lent books it keeps for you. (Better ask him his name next time, though.)
You can obtain the fields (as they are called) by putting the compound in a compound target. In the example, your computer would obey

```
PUT book IN author, title
```

by putting *'N&P'* in *author* and *'Mote'* in *title*.
The following is a neat trick to swap the contents of two targets:

```
PUT a, b IN b, a.
```

This tells the computer to make the compound *(a, b)* and to decompose it into *(b, a)*.

Lists are like lists you make to do shopping: if you and a friend of yours each make a list, and your list is

tooth paste
shampoo
cucumbers
yoghurt
muffins
birthday present for linda

and your friend has

birthday present for linda
shampoo
tooth paste
muffins
cucumbers
yoghurt

and you compare lists, you will exclaim: why, we have *exactly* the same list. Similarly, your computer considers {*t; s; c; y; m; b*} and {*b; s; t; m; c; y*} as the *same* list. In fact, it always sorts the entries in a list from low to high; if you tell your computer to

```
PUT {5; 7; 3; 2} IN a
INSERT 4 IN a
WRITE a
```

you will see *{2; 3; 4; 5; 7}* written. The same entry may occur several times in a list. If you tell your computer to

```
PUT {} IN letters
FOR c IN 'mississippi':
    INSERT c IN letters
WRITE letters
```

it writes back {*'i'; 'i'; 'i'; 'i'; 'm'; 'p'; 'p'; 's'; 's'; 's'; 's'*}.
You may insert all kinds of values in a list, but for each list they must all be the same type of value (all numbers, or all texts, etc.). You may use {*1..n*} as shorthand for {*1; 2; ... ; n−1; n*} and similarly {*'a'..'z'*}.

Tables are somewhat like dictionaries. A short English-Dutch dictionary (not sufficient to maintain a conversation) might be

aardvark:	aardvarken
apartheid:	apartheid
furlough:	verlof
of:	van
or:	of
van:	bestelwagen
yacht:	jacht

Table entries, like entries in a dictionary, consist of two parts. The first part is called the *key*, and the second the *associate*. All keys must be the same type of value, and similarly for associates. A table may be written thus: {*['I']: 1; ['V']: 5; ['X']: 10*}.
If this table has been put in a target *roman*, then *roman['X'] = 10.*
Your computer keeps the tables sorted by key. If you next tell your computer to

```
PUT 100 IN roman['C']
```

then *roman* will contain {*['C']: 100; ['I']: 1; ['V']: 5; ['X']: 10*}. You can find out what the keys are with the function *keys*; in the example, *keys roman* = {*'C'; 'I'; 'V'; 'X'*}.

PREDEFINED COMMANDS

HOW'TO c: commands
tells your computer how to execute *your* command c. It must not be used inside other commands.

YIELD f: commands
tells your computer what value it must yield for *your* formula f when it is computed. It must not be used inside other commands.

TEST p: commands
tells your computer whether *your* proposition p should succeed or fail when it is tested. It must not be used inside other commands.

CHECK test
checks if the test succeeds, in which case nothing happens, but aborts if the test fails.

WRITE e
writes the value of e on the screen. It gives new lines for any /-signs before and after e.

READ t *EG* e
asks an expression from you to put in t. The e tells your computer what type of expression to ask for (number, text, etc.).

PUT e *IN* t
puts the value of e in t.

DRAW t
draws a random number (from ~0 up to ~1) and puts it in t.

CHOOSE t *FROM* l
chooses at random an element from the text, list or table l and puts it in t. (The element is not removed from l.)

SET'RANDOM e
sets the random generator, using the value of e.

REMOVE e *FROM* l
removes the value of e from the list held in l. The value must occur in that list. It is removed only once.

INSERT e *IN* l
inserts the value of e in the list held in l.

DELETE t
deletes the target t. This is used mostly to delete entries from tables or to kill permanent targets.

QUIT
quits from a *HOW'TO* or refinement.

RETURN e
returns the value of e from a *YIELD* or refinement for further computation.

REPORT test
reports from a *TEST* or refinement whether the test succeeds or fails.

SUCCEED
reports success from a *TEST* or refinement.

FAIL
reports failure from a *TEST* or refinement.

IF test*:* commands
executes the commands if the test succeeds.

```
SELECT:
     test: commands
     .
     .
     .
     test: commands
```
selects the first test to succeed and executes the commands after that test. At least one test must succeed. To make sure, the last test may be *ELSE*, which catches if all other tests fail.

WHILE test*:* commands
executes the commands if the test succeeds, and keeps repeating this while the test keeps succeeding. If it fails the very first time around, the commands are not executed at all.

FOR t *IN* e*:* commands
executes the commands for t ranging over the successive characters of e if e is a text, entries of e if e is a list, and associates of e if e is a table.

ALLOW t
allows the use of the permanent t inside a *HOW'TO-*, *YIELD-* or *TEST*-body. It must occur there at the head.

PREDEFINED FUNCTIONS AND PREDICATES

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 (i.e., *~y <> ~0*). 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 (i.e., *~x < ~0*), *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*.

abs x
returns the absolute value of *x*. The result is exact if the operand is exact.

sign x
returns an exact number from {*−1..+1*} with the same sign as *x* (where, e.g., *sign ~0 = sign −~0 = 0*).

floor x
returns the largest integer not exceeding *x* in arithmetic magnitude (so, even if perhaps *3 > ~3*, *floor ~3* still returns *3*).

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)*. (Both operands may be approximate, and *n* may be negative, but not zero.)

*\/*x*
returns the smallest positive integer *q* such that *q*x* is an integer. The value of *x must* be an exact number.

**/x*
returns 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*.

cos x
returns an approximate number by applying the cosine function to *x*.

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)*.

(There should also be a collection of simple matrix functions.)

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 that is not negative.

x<<n
converts *x* to a text (see 5.1.2.2.b) 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 likely effect is that your beautiful lay out is spoiled. The value of *n must* be an integer.

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.

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.

Functions and predicates 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 is equal to the first operand (which *must* have 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 is equal to the first operand (which *must* have the same type as the associates in the table.) For example, *3 # {[1]: 3; [2]: 4; [3]: 3} = 2*.

e in t
accepts texts, lists and tables for the right operand. It succeeds if *e#t > 0* succeeds.

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

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 'syrupy' = 'p'*, *min {1; 3; 3; 4} = 1*, and *min {[1]: 3; [2]: 4; [3]: 3} = 3*. The text, list or table *must not* be empty.

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* have 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* have 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*.

0. Introduction

B is a programming language designed to be used on personal computers. Its primary aim is ease of use for the programmer who wants to produce working programs without having to master a complex tool. *B* is a simple but powerful language, suitable for applications as developing your own games, bookkeeping in and around the house (not only financial), simple engineering computations, solving puzzles, or learning how to program. It is not suited to the development of huge production software, such as operating systems or compilers. (Writing an interpreter in *B* for a mildly complex language is, however, quite feasible.)

At first sight this definition of *B* may give the impression that *B* is not simple at all. Although *B* has grown beyond the original intention, we still feel that the language has a pervading basic simplicity that is hidden by this definition. This may largely be caused by the fact that the simplicity we had in mind when designing *B* was not definitional simplicity. In fact, it turns out that efforts to prevent "surprises" (i.e., the definition prescribes an effect that the innocent user would not expect) do not help to keep the definition short. Also, some of the more powerful concepts of *B* are easier to grasp than to describe formally.

One of the strong points of *B* (as we see it) is that the user can start using *B* productively before she knows all concepts of *B*. The more advanced concepts (e.g., parsing) are best explained in terms of the basic concepts.

This (provisional) definition of *B* is called semi-formal because the present description, in spite of the use of a VW-grammar, does not capture most of the "static semantics" in the syntactic description. Instead, these requirements are stated informally. An (unsuccessful) attempt has been made to make the definition complete, in the sense that anything producible from the Syntax is either assigned a unique meaning, or outlawed. Nevertheless, the informality is such that the finer points may depend on the benevolent imagination of the reader.

The following may clarify some of the treatment of obscure issues. In the design process, we started with an informal, simple conception of the meaning of various constructions. In formalizing the description, it then turns out, from time to time, that there are dark corners in which ambiguities, or worse, are lurking. So we have a choice: restrict the access to the corner, or shed light on it by defining a meaning, thus stretching the original conception. The question we asked is: what does the user mean? If there is one obvious meaning she must have had in mind, we tried to define the Semantics accordingly. If there are several acceptable potential meanings, we tried to fence the corner off. For assessing the proposed language, most of this is not particularly relevant. With other solutions, the language would be equally useful.

This document is rather putting-off in style. It is meant to be digested by computer science gurus. Other interested parties are referred to the informal definition of *B* (which is, alas, not available yet).

In a full definition of *B* also editing and operating procedures should be described, since *B* is a full system, not a mere language. In the present document, only the language itself is defined. See, however, the Appendix.

B (which, by the way, is only a temporary name) is by no means frozen. All suggestions for improvements are welcomed. Please keep in mind that we want to keep *B* simple. Some non-technical background on the *B* project is given in GEURTS & MEERTENS[3] and MEERTENS[5].

In issuing this draft proposal, we express our feeling that the development of *B* has reached a state where pilot implementations are in order. Those interested in the development of *B* are urged to contact us, by writing to

B group
Mathematical Centre
Computer Science Department
P.O.B. 4079
1009 AB Amsterdam
The Netherlands

We are especially interested to hear from prospective implementers. We feel that implementers should, at this stage of affairs, not feel compelled to follow this definition to the last letter. Instead, if they see improvements keeping to the spirit of the language, they are urged to adopt these. But please, let us know. The experience gained will help us to improve *B* for the "official" release.

0.1. Some technical background

Although *B* has no declarations for variables, it is a strongly typed language. The type system is similar to that of LCF, for which a theory has been given in MILNER[6]. The high level of *B* should help to reduce the overhead in interpretive implementations. (A static type check and other static checks may be combined with an interpretive implementation.) Compilation of *B* should be possible by using techniques as described, e.g., by GEHANI[2]. For pilot implementations aiming at obtaining experience in the use of *B*, this does not seem worth the trouble. Also, any not grossly inefficient technique for handling lists and tables will do for such implementations. Obvious candidates for optimization are scratch-pad copying, parsing and the use of quantifications as the test of a while-command. Application of the monadic functions # and *keys* (the latter if no copy is taken) should have a negligeable cost, or at least not increasing linearly with the size of the objects involved. Similarly, the use of a list-display of the form $\{a..b\}$ should not require, if no copy is taken, that the list be created in its full length.

0.2. VW-grammars

The syntax of *B* in this document uses a VW-grammar, as in the ALGOL 68 Revised Report (VAN WIJNGAARDEN[7]).

VW-grammars can be used to define any language, however weird. Such is the power of the mechanism. In this definition, only a very limited use is made of that power. The main use of VW-grammar here is to enforce type agreement. If the type system is orthogonal (and it is so), this can be very naturally expressed in a VW-grammar. The agreement between "definitions" and "applications" (enforced in [7] by the "NEST") is not captured in the present syntax.

The following, terse, exposition on VW-grammars is purely informal. It is included only for the sake of making this document (to some extent) self-contained. An entertaining complete exposition of VW-grammars is given in CLEAVELAND & UZGALIS[1].

A VW-grammar is like a conventional BNF-grammar, except for two things. The first difference is rather superficial. The left-hand side of a production rule is separated by a colon from the right-hand side, alternative productions are separated by semicolons, and the members of a production are separated by commas. Moreover, the squiggles representing terminal symbols are not written as such in the production rules. Rather, they are defined in a separate section, called "Representations". For example, a typical BNF production rule as

```
<factor>::= <primary> | ( <expression> )
```

is written in VW-style as

```
factor:
    primary;
    open-sign, expression, close-sign.
```

(The use of hyphens in these rules is a local deviation in this document from the "official" VW-style.)

The other difference is much deeper. A VW-grammar has two levels. In the production rules words occur, spelled in capital letters (e.g., "TYPE"). These are known as "metanotions". A separate level of production rules, with two colons instead of one, (the "metalevel") defines "terminal metaproductions" for these metanotions (for "TYPE", among others, "numeric" and "textual"). The production rules containing metanotions serve as templates for spelled-out rules. For example, rule 6.1.2.1.a,

> TYPE-target-content: basic-TYPE-identifier.

serves as a template for rules as:

> numeric-target-content: basic-numeric-identifier.

and

> textual-target-content: basic-textual-identifier.

and, in fact, many more rules. In this way, the grammar partially enforces type agreement. If there is no need for agreement, this can be indicated by appending digits to the metanotions: metanotions with *different* digits need not agree. For example, rule 6.1.4.1.a,

```
TYPE2-table-selection:
    tight-table-with-TYPE1-keys-TYPE2-associates-expression,
        TYPE1-key-selector.
```

is a template for, among others,

```
numeric-table-selection:
    tight-table-with-textual-keys-numeric-associates-expression,
        textual-key-selector.
```

and

```
textual-table-selection:
    tight-table-with-numeric-keys-textual-associates-expression,
        numeric-key-selector.
```

"TYPE1" and "TYPE2" may also stand for the same type, as in

```
numeric-table-selection:
    tight-table-with-numeric-keys-numeric-associates-expression,
        numeric-key-selector.
```

A guard of the form “where ... is ...” may appear in front of an alternative. This means that the alternative applies only if the two “...” parts agree. This depends on the substitutions for the metanotions. A guard of the form “unless ... is ...” works the other way around.

Some metanotions are used, but not defined (“FIRST”, “NAMED”, “NEXT”, “NOTION”, “TAG”). It is not particularly important what their exact definition is, as long as there is a sufficient supply of terminal metaproductions. In particular, “NOTION” has anything as terminal metaproduction.

0.3. General constructions

0.3.1. Syntax

```
A) ITEM::
        identifier;
        target;
        expression.

a) collateral-TYPE-ITEM:
        TYPE-ITEM;
        where TYPE is compound-with-TYPE1-TYPES-fields,
            TYPE1-TYPES-ITEM.
```

(For TYPE and TYPES, see 1.1.)

```
b) TYPE-ITEM:
        basic-TYPE-ITEM;
        open-sign, collateral-TYPE-ITEM, close-sign.
```

(For basic-identifiers, -targets and -expressions, see 6.2.1.1.a, 6.2.0.1.a and 6.1.0.1.a.)

```
c) TYPE-TYPES-ITEM:
        TYPE-ITEM, comma-sign, TYPES-ITEM.
```

d) All tags contained in an identifier must be different.

```
e) optional-NOTION:
        ;
        NOTION.
```

Examples:

a) collateral-TYPE-identifier:
a
(a)
(a, b, (c, d))
a, b, (c, d)

b) TYPE-identifier:
a
(a)
(a, b, (c, d))

(For the Semantics of TYPE-TYPES-identifiers, collateral-targets and -expressions, see 6.2.1.2.b, 6.2.0.2.a and 6.1.0.2.a.)

0.4. Symbols and representations

0.4.1. Syntax

a) *symbol:
NAMED-keyword;
NAMED-tag;
NAMED-sign.

b) *typographical-display-feature:
space;
new-line;
increase-indentation;
decrease-indentation.

c) new-line:
optional-comment, new-line-proper, indent.

d) comment:
optional-new-line-proper, optional-spaces, comment-sign,
comment-body, optional-further-comment.

e) further-comment:
new-line-proper, optional-spaces, comment-sign, comment-body,
optional-further-comment.

f) spaces:
space, optional-spaces.

0.4.2. Representations

a) The representation of a NAMED-keyword is obtained by representing NAMED in the "keyword alphabet", which consists of 38 marks. The representations establish a one-one-correspondence between the terminal metaproductions of "NAMED" and the sequences of marks whose first mark is a letter.

In this document, the following marks are used for the keyword alphabet:

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

0 1 2 3 4 5 6 7 8 9 ' "

where the first line gives the letters.

Although the rules for representing a NAMED-keyword are not further specified here, for the keywords named in the Syntax a rather obvious correspondence should be applied, in which, e.g., a poiuyt-keyword is represented by *POIUYT*.

b) The representation of a NAMED-tag is obtained by representing NAMED in the "tag alphabet", which consists of 38 marks. There is a one-one-correspondence between the terminal metaproductions of "NAMED" and the sequences of marks whose first mark is a letter.

In this document, the following marks are used for the tag alphabet:

a b c d e f g h i j k l m n o p q r s t u v w x y z

0 1 2 3 4 5 6 7 8 9 ' "

where the first line gives the letters. (For some examples of tags, see 6.2.1.I.a.)

(If the keyword and tag alphabets are indistinguishable, extra parentheses may be needed to disambiguate some combinations.)

c) The representation of the NAMED-signs is as follows:

sign	**representation**	**sign**	**representation**
colon-sign	*:*	one-sign	*1*
comment-sign	\	two-sign	*2*
new-line-sign	*/*	three-sign	*3*
open-sign	*(*	four-sign	*4*
close-sign	*)*	five-sign	*5*
comma-sign	*,*	six-sign	*6*
zero-sign	*0*	seven-sign	*7*

sign	**representation**	**sign**	**representation**
eight-sign	8	over-sign	/
nine-sign	9	to-the-power-sign	**
point-sign	.	numerator-sign	*/
apostrophe-sign	'	denominator-sign	/*
quote-sign	"	join-sign	^
convert-sign	`	repeat-text-sign	^^
curly-open-sign	{	adjust-left-sign	<<
curly-close-sign	}	center-sign	><
enumeration-sign	;	adjust-right-sign	>>
through-sign	..	number-sign	#
sub-sign	[	less-than-sign	<
bus-sign	]	at-most-sign	<=
behead-sign	@	equals-sign	=
curtail-sign	|	unequal-sign	<>
about-sign	~	at-least-sign	>=
plus-sign	+	greater-than-sign	>
times-sign	*		

(The following signs occur only as the sign of a textual-item.)

sign	**representation**	**sign**	**representation**
space-sign		capital-o-sign	O
exclamation-sign	!	capital-p-sign	P
dollar-sign	$	capital-q-sign	Q
percent-sign	%	capital-r-sign	R
ampersand-sign	&	capital-s-sign	S
question-sign	?	capital-t-sign	T
capital-a-sign	A	capital-u-sign	U
capital-b-sign	B	capital-v-sign	V
capital-c-sign	C	capital-w-sign	W
capital-d-sign	D	capital-x-sign	X
capital-e-sign	E	capital-y-sign	Y
capital-f-sign	F	capital-z-sign	Z
capital-g-sign	G	underscore-sign	_
capital-h-sign	H	a-sign	a
capital-i-sign	I	b-sign	b
capital-j-sign	J	c-sign	c
capital-k-sign	K	d-sign	d
capital-l-sign	L	e-sign	e
capital-m-sign	M	f-sign	f
capital-n-sign	N	g-sign	g

sign	representation	sign	representation
h-sign	*h*	r-sign	*r*
i-sign	*i*	s-sign	*s*
j-sign	*j*	t-sign	*t*
k-sign	*k*	u-sign	*u*
l-sign	*l*	v-sign	*v*
m-sign	*m*	w-sign	*w*
n-sign	*n*	x-sign	*x*
o-sign	*o*	y-sign	*y*
p-sign	*p*	z-sign	*z*
q-sign	*q*		

d) With the exception of a comment, the user has no control over the way typographical-display-features are inserted in her units and how they are displayed (see Appendix). With the current syntax, comments are not allowed preceding or following a unit. This should be relaxed.

e) (The following is tentative.) In the "canonic representation" a space is automatically inserted before and after keywords to separate these from adjacent other symbols, before and after ORDER-signs, predicates and non-empty optional-new-liners of a write-command, after a tag, textual-display or numeric-constant if followed by another tag, textual-display or numeric-constant, after a comma, colon and enumeration-sign, after a dyadic-function if followed by a (monadic) function, and before a comment-sign if preceded by another symbol on the same line. Moreover, if spaces have been inserted in a construction, that construction is separated by spaces from adjacent symbols, and if a space is inserted between a dyadic-function and one of its operands, then a space is also inserted between the function and the other operand. However, at most one space is inserted, except before a comment-sign, and no space is inserted within a symbol, a constant or a textual-display, except possibly within conversions contained therein, after any of the opening or before any of the closing parentheses, nor before a comma, colon or enumeration-sign. The examples in this document follow the—tentative—canonic representation. Most situations mentioned occur in:

```
IF a < b AND 0 in min q: \example of space insertion
    WRITE / {a; b}, 1+3.14E-2, 1 + sin 3.14, min 'aA' /
```

f) On displaying a command- or alternative-suite, an increase-indentation causes an incrementation of the indentation (left margin setting). A new-line-proper causes a transition to the next line, and an indent positions at the current indentation position. A decrease-indentation resets the indentation to the previous position.

1. Types and values

a) A "type" is a semantic attribute of a *B* value. It is a terminal metaproduction of "TYPE" not containing "poly".

b) A "polytype" is a syntactic (static semantic) attribute of a *B* expression. It is any terminal metaproduction of "TYPE". The polytype may be "refined" to the type of the values that may result from evaluating the expression by consistently substituting some type for all occurrences of the form "poly-TAG" in the polytype. For example,

```
table-with
          -compound-with
                        -poly-a
                        -poly-b
                   -fields
     -keys
          -poly-a
     -associates
```

may be refined to

```
table-with
          -compound-with
                        -textual
                        -numeric
                   -fields
     -keys
          -textual
     -associates
```

(where the lay out is intended to suggest the structure of the (poly)type.)

c) A self-contained collection of *B* units contains implicit definitions for the polytypes of all its expressions, in particular for its identifiers. At the time of application, the types to which these polytypes should be refined are known from the invocation.

1.1. Metaproduction rules

A) TYPE::
    numeric;
    textual;
    COMPOUND;
    LIST;
    TABLE;
    poly-TAG.

B) COMPOUND::
    compound-with-TYPE-TYPES-fields.

C) TYPES::
    TYPE;
    TYPE-TYPES.

D) LIST::
    list-of-TYPE.

E) TABLE::
    table-with-TYPE1-keys-TYPE2-associates.

1.2. Values

a) Each value has a type. The set of values of a given type *T* is called the *T* "domain". If T_1 and T_2 are different types, the T_1 and T_2 domains are disjoint. For example, each list-of-*T* domain contains its own empty list. Moreover, there exists a set of "characters" (see 1.2.2.a), the character domain. (Characters are not values. See, however, 1.2.2.c.)

b) A total ordering is defined on each domain (but not across domains). If the elements of the domain are sequences of other values or of characters, then that ordering is the lexicographic (dictionary) ordering, using the ordering already defined on the values or characters from which the sequences are composed. For example, the test

(1, 'aa') < (1, 'z') < (1.001, '')

succeeds.

1.2.1. Numbers

a) A "number" is either an "exact number" or an "approximate number" and has the type "numeric". The "exact numbers" are the rational numbers; arithmetic on exact numbers is supposed to be exact and to imply no constraints (other than memory exhaustion) on the size of the integers involved. The "approximate numbers" are a subset of the real numbers; in general, arithmetic on approximate numbers is not exact and yields, unless some specific functions are involved, again an approximate number. In an implementation, these numbers may be modelled using a floating point representation. In contrast to mathematical practice, the sets of exact and approximate numbers are disjoint. This has, in particular, the consequence that, if the value of x is an approximate number, the test $x/x = 1$ does not succeed. So, as a consequence of 1.2.1.b, one of $x/x < 1$ and $x/x > 1$ must succeed. (The test $x/x = \sim 1$, however, might have more success.) For a limited number of operations, a run-time check is used to ensure that a number is an exact number or even an integer. (An example is the requirement that the values of i and j in $\{i..j\}$ be integers. This is preferred to other solutions (e.g., automatic rounding) for reasons of security.)

b) The numbers are ordered according to their arithmetic magnitude, with some tie breaking rule (not further specified, but consistent) for exact and approximate numbers with the same magnitude.

1.2.2. Texts

a) A "text" is a (possibly empty) sequence of characters and has the type "textual". Acceptable characters are the printing ASCII characters, including the space, but not tab, backspace and new line or carriage return. They are the 95 characters represented on the lines below, where the blank space " " preceding "*!*" stands for the (otherwise invisible) space character:

```
 ! " # $ % & ' ( ) * + , - . / 0 1 2 3 4 5 6 7 8 9 : ; < = > ?

@ A B C D E F G H I J K L M N O P Q R S T U V W X Y Z [ \ ] ^ _

` a b c d e f g h i j k l m n o p q r s t u v w x y z { | } ~
```

b) The ordering on the characters is the ASCII collating order. (This is the order in which the characters are displayed above.)

c) In the sequel, a text consisting of one character will be identified with that character (so the statement "the text *T* is a character" is equivalent to "the length of the text *T* is one").

1.2.3. Compounds

a) A "compound" is a sequence (tuple) of at least two "fields" and has a type of the form "COMPOUND". Each "field" is a value of some type. If the type of the compound is compound-with-T_1-...-T_n-fields, where each T_i, $1 \leqslant i \leqslant n$, is a type, then the compound has n fields, and the type of the i-th field is T_i.

1.2.4. Lists

a) A "list" is a (possibly empty) *sorted* sequence of "list entries" and has a type of the form "LIST". Each "list entry" is a value. If the type of the list is list-of-T, then each list entry has the type T. A list entry may occur more than once in a list. For example, the list {*1; 3; 3; 4*} contains two occurrences of the number 3.

1.2.5. Tables

a) A "table" is a (possibly empty) *sorted* sequence of "table entries" and has a type of the form "TABLE". Each "table entry" is a sequence of two values, the "key" and the "associate" of the table entry. If the type of the table is table-of-T_1-keys-T_2-associates, where T_1 and T_2 are types, then each table entry has a key of type T_1 and an associate of type T_2. Different table entries have different keys. A given table entry occurs at most once in a table (contrary to list entries in lists).

2. Locations and environments

2.1. Locations

a) A "location" is either a "simple location", a "trimmed-text location", a "compound location" or a "table-selection location". Each location has a type. (Locations are not values.)

b) A value of some type may be "put in" a location of that same type, whereupon the "content" of the location is that value. If, subsequently, another value is put in the location, the content of the location is thereupon that other value.

c) In some cases a location may be "emptied", whereupon it no longer has a content, until again a value is put in the location.

2.1.1. Simple locations

a) A new "simple location" may be created for some given type. (If that type is some COMPOUND, the thus created location is, nevertheless, not a compound location.)

b) Until a value is put in a newly created simple location, it has no content.

c) A simple location may be emptied.

2.1.2. Trimmed-text locations

a) A "trimmed-text location" is a location for textual. It is composed of another location for textual, the "root location" (which is a simple, table-selection or again a trimmed-text location), and two integers, the "behead" and the "curtail".

b) A trimmed-text location T composed of a root location R, a behead B and a curtail C has no content if R has no content, if $B < 0$, if $C < 0$ or if $B+C > n$, where n is the length of the content of R. Otherwise, let the content of R be the text $r_1, \ldots, r_n$. The content of T is then $r_{B+1}, r_{B+2}, \ldots, r_{n-C-1}, r_{n-C}$.

c) A text consisting of m characters $c_1, \ldots, c_m$ is put in a trimmed-text location T composed of a root location R, a behead B and a curtail C as follows:
Let the content of R be $r_1, \ldots, r_n$.
The text $r_1, \ldots, r_B, c_1, \ldots, c_m, r_{n-C+1}, \ldots, r_n$ is put in R.

(For example,

```
PUT 'impious' IN t
PUT 'sh' IN t@5
```

has the effect of putting *'impish'* in *t*, and

```
PUT 'lush' IN t
PUT 'b' IN t|0
```

puts *'blush'* in *t*. Trims may be stacked:

```
PUT 'hereby' IN t
PUT 's' IN t@5|1
```

altogether puts *'heresy'* in *t*.)

d) A trimmed-text location must not be emptied. If the Semantics, during the execution of a command, prescribes that a trimmed-text location be emptied, an error is signalled.

2.1.3. Compound locations

a) A "compound location" is a location for some COMPOUND. It is a sequence of locations (not all of which are necessarily different).

b) A compound location C composed of n locations $L_1, \dots, L_n$ has no content if, for some i, $1 \leqslant i \leqslant n$, L_i has no content. Otherwise, let, for $1 \leqslant i \leqslant n$, V_i be the content of L_i. The content of C is then the compound whose fields are $V_1, \dots, V_n$.

c) A compound consisting of n fields $F_1, \dots, F_n$ is put in a compound location C composed of n locations $L_1, \dots, L_n$ as follows:

```
For $i = 1, \dots, n$:
    $F_i$ is put in $L_i$.
If the content of $C$ is not $F_1, \dots, F_n$ (e.g., because for some $i$ and $j$,
        $1 \leqslant i, j \leqslant n$, $L_i$ and $L_j$ are the same location and $F_i$ is different
        from $F_j$):
    An error is signalled.
```

(The order in which the fields are put in the component locations should be immaterial. The current semantic model for trimmed-text locations appears to have a defect in this respect; moreover, it sometimes results in surprises. It seems possible to give a semantic model for trimmed-text locations for which the intended results are obtained, but only at the cost of considerable complications. The essence of the idea is to adorn the interstices between the characters with unique labels. Note that *PUT t[j], t[i] IN t[i], t[j]* is well defined, regardless of whether i and j have the same value or not, as long as these values are keys of the table t.)

d) A compound location is emptied by emptying the locations of which it is composed.

2.1.4. Table-selection locations

a) A "table-selection location" is composed of a location for some TABLE, the "parent location" (which is either a simple or again a table-selection location), and a key.

b) A table-selection location L composed of a parent location P and a key K has no content if P has no content or if the content of P contains no table entry whose key is *K*. Otherwise, let the content of *P* be the table *T*. The content of *L* is then the associate of the table entry of *T* whose key is *K*.

c) A value *V* is put in a table-selection location *L* composed of a parent location *P* and a key *K* as follows:

```
Let T be the content of P.
Let T' be the table whose table entries are those of T, with the omission of
            any table entry with key K, but with a new table entry composed
            of the key K and the associate V.
T' is put in P.
```

d) A table-selection location *L* composed of a parent location *P* and a key *K* is emptied as follows:

```
If L has no content:
    The emptying is a dummy action.
Otherwise:
    Let T be the content of P.
    Let T' be the table whose table entries are those of T, with the omis-
                sion of the table entry whose key is K.
    T' is put in P.
```

2.2. Environments

a) An "environment" is a (partial) map from tags to simple locations. The simple location to which a tag *T* is mapped by an environment *E* is said to be the location "accessed" by *T* in *E*. There is one distinguished environment (for each work-space), the "permanent environment". (Environments are not values.)

b) An environment may be created, its domain may be augmented with tags, and tags may be discarded from the domain of an environment. The tags with which an environment domain is augmented, may be "bound". (This is the case if the augmentation is caused by locating the collateral- or TEXTUALS-identifier of an in- or parsing-ranger, or if they are taken from another environment in which they are bound.)

c) A tag may be made to access a simple location in an environment.

d) A "transparent copy" *E'* of an environment *E* is a newly created environment with the same domain as *E*, in which each tag accessing a location in *E* is made to access that same location.

(Transparent copies serve to allow side-effects caused by commands whilst precluding the export of tags local to the command. This model should be equivalent to the more usual model where locales are chained in an environ(ment); the present model has no need for searching that chain. It is suspected that the present Semantics would work as well, with the same effects, if no transparent copies were made. Because we are not sure, they are prescribed for robustness' sake. Moreover, implementers are thereby suggested a slightly less unrealistic model for environments in case they model the shielding mechanism (4.0.2.c) with conventional techniques.)

e) A "scratch-pad copy" E' of an environment E is a newly created environment with the same domain as E, in which each tag accessing a location L in E is made to access a newly created simple location L' for the same type, in which location L' the content of L, if any, is put.

(Scratch-pad copies serve to warrant that formulas and tests cannot have side-effects. However, currently, the following are still allowed:
- read and write;
- the setting and drawing of random numbers (5.1.5.2.b);

and, of course, the signalling of errors.

A simpler, but more restrictive, way to obtain a similar result would be to outlaw any putting to "non-local" locations in the elaboration of formulas and tests. If putting to the operands is also forbidden and the above-mentioned side-effects are also outlawed, it is even possible to unify the parameter-passing mechanism for how-to- and other units, at the cost of requiring basic-identifiers as formal-operands.)

3. Elaboration

a) The computer is able to "elaborate" certain "constructions", as producible from the Syntax: it is able to "execute", among others, "commands", to "evaluate", among others, "expressions" (6.1.a), to "locate", among others, "targets" (6.2.a) and to "test", among others, "tests" (6.3.a). The sections headed "Semantics" specify the elaboration of constructions.

b) Each elaboration is performed in an environment, descending, ultimately, from the permanent environment (2.2.a).

c) In general, the elaboration of a construction requires the elaboration of some or all of its component constructions. Unless the Semantics prescribes otherwise, these component constructions are elaborated in the same environment as the original construction.

d) If the Semantics requires the elaboration of some construction, but that elaboration is not specified by the Semantics, then that construction contains a largest component construction for which the Semantics does specify its elaboration, and that elaboration is the required elaboration for the original construction. (For example, the value of the key-selector *[i, j]* is that of its collateral-expression *i, j*.)

4. Units

a) Units are the building blocks from which a *B* "program" is composed. They are not executed, but "invoked" by commands. In an interactive *B* system, these units will reside, conceptually, in a work-space.

b) For non-interactive pilot implementations of *B*, the following syntax is suggested:

```
program:
    optional-unit-suite, immediate-command.

unit-suite:
    unit, new-line, optional-unit-suite.

immediate-command:
    command, optional-refinement-suite.
```

A program is then executed by executing the immediate-command. An immediate-command is executed as though it were the body of an invoked how-to-unit, except that it is executed in the permanent environment (2.2.a). The request to execute an immediate-command is refused, however, if it does not pass the (strict) "static checks" (7.a).

c) For a properly interactive implementation, the following approximation is suggested:

```
session:
    user-action, optional-session;

user-action:
    unit;
    immediate-command.
```

After each execution of an immediate-command, those tags are deleted from the domain of the permanent environment that access a location with no content (because of a delete-command). The following is therefore allowed

```
PUT 0 IN x
DELETE x
PUT 'a' IN x
```

when given at the outer level, but not inside a command-suite.

4.0.1. Syntax

a) *unit:
   how-to-unit;
   yield-unit;
   test-unit.

b) refinement-suite:
   new-line, refinement, optional-refinement-suite.

c) A unit must pass the (relaxed) static checks (7.a). (In particular, the static type check ensures that the polytypes of identifiers, including those of formal-parameters and -operands, are consistent throughout the unit.)

d) A tag "local" (4.0.2.a) in a unit must not occur in that unit in an allow-heading.

e) A tag "global" (4.0.2.b) in a unit not occurring there in an allow-heading, may only occur as the tag of a function or predicate.

f) The keyword of a command-refinement contained in a unit may occur otherwise in that unit as the first keyword of a command only if that command is a refined-command.

g) The tag of the basic-identifier of an expression-refinement contained in a unit may occur otherwise in that unit only as the tag of a refined-expression.

h) The tag of the basic-identifier of a test-refinement contained in a unit may occur otherwise in that unit only as the tag of a refined-test.

4.0.2. Locality and shielding

a) A tag is "local" in a unit *U* if it occurs in *U*
- in a formal-parameter or -operand,
- in the basic-identifier of a basic-target but not in an allow-heading in *U*,
- in the collateral-identifier of a for-command or ranger,

or ● in the basic-identifier of an expression- or test-refinement.

b) A tag is "global" in a unit *U* if it occurs in *U* but is not local in *U*.

c) A copy of a unit *U* "shielded for" an environment *E* is a unit that is a copy of *U* in which, for the tags local in *U*, new tags have been consistently substituted that are (arbitrarily chosen but) different from each other, from the tags global in *U* and from the tags in the domain of *E*.

(Shielding serves to prevent interference between the tags imported by actual-/formal-parameter substitution and the local tags.)

4.1. How-to-units

4.1.1. Syntax

a) how-to-unit:
how-to-keyword, formal-user-defined-command, colon-sign,
command-suite,
optional-refinement-suite.

b) formal-user-defined-command:
FIRST-keyword, optional-formal-tail.

c) formal-tail:
formal-parameter, optional-formal-trailer;
formal-trailer.

d) formal-trailer:
NEXT-keyword, optional-formal-tail.

e) formal-parameter:
basic-TYPE-identifier.

f) The FIRST-keyword of a formal-user-defined-command must be unique, i.e., different from all FIRST-keywords of formal-user-defined-commands of other how-to-units. Otherwise, it may be chosen freely (given the way of composing keywords from the keyword alphabet), except that it must not be a "predefined" first keyword (*ALLOW*, *CHECK*, *CHOOSE*, *DELETE*, *DRAW*, *FAIL*, *FOR*, *HOW'TO*, *IF*, *INSERT*, *PUT*, *QUIT*, *READ*, *REMOVE*, *REPORT*, *RETURN*, *SELECT*, *SET'RANDOM*, *SUCCEED*, *TEST*, *WHILE*, *WRITE*, *YIELD*).

g) All tags contained in the formal-user-defined-command must be different.

Example:
a) how-to-unit:
```
HOW'TO PUSH val ON stack:
     PUT val IN stack[#stack]
```

4.2. Yield-units

4.2.1. Syntax

A) ADIC::
  zeroadic;
  monadic;
  dyadic.

a) yield-unit:
  yield-keyword, formal-TYPE-ADIC-formula, colon-sign,
    command-suite,
    optional-refinement-suite.

b) formal-TYPE-zeroadic-formula:
  zeroadic-function.

c) formal-TYPE-monadic-formula:
  monadic-function, formal-operand.

d) formal-TYPE-dyadic-formula:
  formal-operand, dyadic-function, formal-operand.

e) formal-operand:
  TYPE-identifier.

f) Functions must not be "overloaded". However, a given tag may be used, at the same time, (*i*) for a zeroadic- or monadic-function and (*ii*) for a dyadic-function.

g) All tags contained in the formal-formula must be different.

Example:
a) 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
```

4.3. Test-units

4.3.1. Syntax

a) test-unit:
  test-keyword, formal-ADIC-proposition, colon-sign,
    command-suite,
    optional-refinement-suite.

b) formal-zeroadic-proposition:
  zeroadic-predicate.

c) formal-monadic-proposition:
  monadic-predicate, formal-operand.

d) formal-dyadic-proposition:
  formal-operand, dyadic-predicate, formal-operand.

e) Predicates must not be "overloaded". However, a given tag may be used, at the same time, (*i*) for a zeroadic- or monadic-predicate and (*ii*) for a dyadic-predicate.

f) All tags contained in the formal-proposition must be different.

Example:
a) test-unit:

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

4.4. Refinements

4.4.1. Syntax

a) refinement:
  command-refinement;
  expression-refinement;
  test-refinement.

b) command-refinement:
  FIRST-keyword, colon-sign, command-suite.

c) expression-refinement:
  basic-TYPE-identifier, colon-sign, command-suite.

d) test-refinement:
NAMED-tag, colon-sign, command-suite.

Examples:

b) command-refinement:
```
SELECT'TASK:
    PUT min tasks IN task
    REMOVE task FROM tasks
```

c) expression-refinement:
```
stack'ptr:
    IF stack = {}: RETURN 0
    RETURN max keys stack
```

d) test-refinement:
```
special'case:
    REPORT pos+d = line'length
```

4.5. Command-suites

4.5.1. Syntax

a) command-suite:
simple-command;
increase-indentation, optional-allow-heading,
optional-command-sequence, decrease-indentation.

b) allow-heading:
new-line, allow-keyword, collateral-TYPE-identifier,
optional-allow-heading.

c) command-sequence:
new-line, command, optional-command-sequence.

d) An allow-heading may only occur in the command-suite of a unit (and not of a refinement).

e) All tags occurring in an allow-heading must be different from each other and from all tags occurring in the formal-command, -formula or -proposition of the unit in which it is contained.

f) Each execution path (7.d) 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 a command-suite.

g) Each execution path (7.d) 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 a command-suite.

4.5.2. Semantics

a) A command-suite *C* is executed as follows:
The simple-command or the command-sequence, if any, of the optional-command-sequence is executed.

b) A command-sequence *C* is executed as follows:
The command is executed.
If this execution results in the execution of a terminating-command contained in *C*:
    The execution of the largest command-suite in which *C* is contained is terminated, and a value or outcome, if appropriate, is returned or reported to the invoker of that command-suite.
Otherwise:
    The command-sequence, if any, of the optional-command-sequence is executed.

(Note that allow-headings are not executed.)

5. Commands

5.0.1. Syntax

a) command:
    simple-command;
    control-command.

5.1. Simple-commands

5.1.0.1. Syntax

a) simple-command:
    check-command;
    write-command;
    read-command;
    put-command;
    choose-command;
    draw-command;
    set-random-command;
    remove-command;
    insert-command;

delete-command;
terminating-command;
user-defined-command;
refined-command.

b) terminating-command:
quit-command;
return-command;
report-command;
succeed-command;
fail-command.

5.1.1. Check-commands

5.1.1.1. Syntax

a) check-command:
check-keyword, test.

Example:
a) check-command:

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

5.1.1.2. Semantics

a) A check-command is executed as follows:
The test is tested.
If the test succeeds:
the execution of the check-command is complete (and the environment offered is discarded).
Otherwise:
An error is signalled.

(Although the effect of run-time errors is not further specified here, the idea is that the execution is terminated and that helpful information on the nature of the error is displayed. In a check-command, *extra* care should be exercised to provide such information; e.g., in a form as

```
Your check in YIELD upper m failed:
    CHECK i >= 0 AND j >= 0 AND i+j <= n
                                    ↑↑
    (i = 0, j = 5 and n = 4).)
```

5.1.2. Write-commands

5.1.2.1. Syntax

a) write-command:
  write-keyword, new-liners;
  write-keyword, optional-new-liners, collateral-TYPE-expression,
    optional-new-liners.

b) new-liners:
  new-line-sign, optional-new-liners.

Example:
a) write-command:
WRITE //
WRITE // 'Give a value in the range 1 through `n`: '

5.1.2.2. Semantics

a) A write-command is executed as follows:
If the write-command does not contain an expression:
  $n-1$ blank lines are displayed on the interactive output device, where the write-command contains n new-line-signs, followed by a positioning at the beginning of the next line.
Otherwise:
  The collateral-expression is evaluated, giving a value V.
  If the collateral-expression is preceded by n new-line-signs, $n > 0$:
    $n-1$ blank lines are displayed on the interactive output device, followed by a positioning at the beginning of the next line.
  V is "converted" to a text T.
  T is displayed on the interactive output device (in a typographical lay out that is not further defined here), with at least one space preceding T if not at the beginning of the line.
  If the collateral-expression is followed by n new-line-signs, $n > 0$:
    $n-1$ blank lines are displayed on the interactive output device, followed by a positioning at the beginning of the next line.

b) (The following "definition" is provisional.) A value V is "converted" to a text T as follows:
Let S be the shortest expression not containing keywords other than "*E*" (which occurs in approximate-constants), functions other than monadic "$-$", tags, predicates or conversions, and in which the fillers in list- and table-displays are sorted according to the usual

ordering, such that *S* (whose meaning is environment-independent) would evaluate to a value "close to" *V*.
T is the text whose characters correspond, one to one, to the signs and spaces of *S*, except that the initial and final QUOTE-signs of textual-displays are omitted, and CHARACTER-images are replaced by single CHARACTER-signs (so that a text is converted to itself).

c) A value *V′* is "close to" a value *V* if ... (This should determine, e.g., to how many digits a value as 22/7 is converted.)

5.1.3. Read-commands

5.1.3.1. Syntax

a) read-command:
read-keyword, collateral-TYPE-target, eg-keyword,
collateral-TYPE-expression.

Example:
a) read-command:
READ n, s EG 0, ''

5.1.3.2. Semantics

a) A read-command is executed as follows:
The target is located, giving a location *L*.
The user is prompted to supply a collateral-TYPE-expression.
The user-supplied collateral-expression is evaluated in the permanent environment (2.2.a), giving a value *V*.
V is put in *L*.

(Note that the collateral-expression of the read-command itself is not evaluated.)

(There should also be some way to read "raw" lines supplied by the user into textual locations.)

5.1.4. Put-commands

5.1.4.1. Syntax

a) put-command:
put-keyword, collateral-TYPE-expression, in-keyword,
collateral-TYPE-target.

Example:
a) put-command:
PUT a+1, ({}, {1..a}) IN a, b

5.1.4.2. Semantics

a) A put-command is executed as follows:
The collateral-expression is evaluated and the collateral-target is located, giving a value *V* and a location *L*.
V is put in *L*.

5.1.5. Draw-commands

5.1.5.1. Syntax

a) draw-command:
draw-keyword, numeric-target.

Example:
a) draw-command:
DRAW r

5.1.5.2. Semantics

a) A draw-command is executed as follows:
The target is located, giving a location *L*.
A number x is "drawn".
The number x is put in *L*.

b) A number x is "drawn" by taking the next number from a pseudo-random sequence of approximate numbers, uniformly distributed on [0, 1). (So $\sim 0 \leqslant x < \sim 1$.)

5.1.6. Choose-commands

5.1.6.1. Syntax

a) choose-command:
where TYPE2 lodges TYPE1,
choose-keyword, collateral-TYPE1-target, from-keyword, TYPE2-expression.

b) where TYPE2 lodges TYPE1:
    where TYPE2 is textual and TYPE1 is textual;
    where TYPE2 is list-of-TYPE1;
    where TYPE2 is table-with-TYPE0-keys-TYPE1-associates.

Example:
a) choose-command:
   CHOOSE e FROM exits[cur'room]

5.1.6.2. Semantics

a) A choose-command is executed as follows:
The collateral-target is located and the expression is evaluated, giving a location *L* and a value (text, list or table) *V*.
If *V* is an empty text, list or table:
    An error is signalled.
Otherwise:
    A number *x* is drawn (5.1.5.2.b).
    Let *n* be the number of characters or list or table entries of *V*.
    The *i*-th character, list entry or associate of *V* is put in *L*, where $i = 1 + \text{floor}(n*x)$.

5.1.7. Set-random-commands

5.1.7.1. Syntax

a) set-random-command:
    set-random-keyword, collateral-TYPE-expression.

Example:
a) set-random-command:
   SET'RANDOM 'monte carlo', run

5.1.7.2. Semantics

a) A set-random-command is executed as follows:
The collateral-expression is evaluated, giving a value *V*.
The sequence from which numbers are drawn (5.1.5.2.b) is set to a pseudo-random sequence depending only on *V*.

5.1.8. Remove-commands

5.1.8.1. Syntax

a) remove-command:
remove-keyword, collateral-TYPE-expression, from-keyword,
list-of-TYPE-target.

Example:
a) remove-command:
REMOVE task FROM tasks

5.1.8.2. Semantics

a) A remove-command is executed as follows:

```
The collateral-expression is evaluated and the target is located, giving a value
        V and a location L (for some LIST).
If L has no content:
    An error is signalled.
Otherwise:
    Let W be the content of L.
    If W does not contain V as a list entry:
        An error is signalled.
    Otherwise:
        Let W' be the list whose list entries are those of W, except that V
                occurs one time less in W'.
        W' is put in W.
```

5.1.9. Insert-commands

5.1.9.1. Syntax

a) insert-command:
insert-keyword, collateral-TYPE-expression, in-keyword,
list-of-TYPE-target.

Example:
a) insert-command:
INSERT new'task IN tasks

5.1.9.2. Semantics

a) An insert-command is executed as follows:
The collateral-expression is evaluated and the target is located, giving a value *V* and a location *L*.
If *L* has no content:
    An error is signalled.
Otherwise:
    Let *W* be the content of *L*. Let *W′* be the list whose list entries are those of *W*, except that *V* occurs one time more in *W′*.
    W′ is put in *L*.

5.1.10. Delete-commands

5.1.10.1. Syntax

a) delete-command:
    delete-keyword, collateral-TYPE-target.

Example:
a) delete-command:
    DELETE t[i], u[i, j]

5.1.10.2. Semantics

a) A delete-command is executed as follows:
The collateral-target is located, giving a location *L*.
If *L* has no content:
    An error is signalled.
Otherwise:
    L is emptied.

5.1.11. Quit-command

5.1.11.1. Syntax

a) quit-command:
    quit-keyword.

b) A quit-command may only occur in the command-suite of a how-to-unit or refined-command.

Example:
a) quit-command:
QUIT

5.1.11.2. Semantics

a) A quit-command is executed as follows:
The execution of the largest command-suite containing the quit-command is terminated, whereby the execution of the user-defined-command or refined-command that invoked that command-suite is complete.

5.1.12. Return-commands

5.1.12.1. Syntax

a) return-command:
return-keyword, collateral-TYPE-expression.

b) The TYPE must be that of the formal-TYPE-formula or basic-TYPE-identifier of the yield-unit or expression-refinement in which the return-command occurs.

Example:
a) return-command:
*RETURN (a*c+b*d)/rr, (−a*d+b*c)/rr*

5.1.12.2. Semantics

a) A return-command *R* is executed as follows:
The collateral-expression is evaluated, giving a value *V*.
The execution of the largest command-suite in which *R* is contained is terminated, and *V* is returned to the formula or expression-refinement that invoked that command-suite.

5.1.13. Report-commands

5.1.13.1. Syntax

a) report-command:
report-keyword, test.

Example:
a) report-command:
REPORT i in keys t

5.1.13.2. Semantics

a) A report-command *R* is executed as follows:
The test is tested.
The execution of the largest command-suite in which *R* is contained is terminated, and the outcome of the test is reported to the proposition or test-refinement that invoked that command-suite.

5.1.14. Succeed-command

5.1.14.1. Syntax

a) succeed-command:
    succeed-keyword.

Example:
a) succeed-command:
  SUCCEED

5.1.14.2. Semantics

a) A succeed-command *S* is executed, in an environment *E*, as follows:
The execution of the largest command-suite in which *S* is contained is terminated, and success, offering *E*, is reported to the proposition or test-refinement that invoked that command-suite.

5.1.15. Fail-command

5.1.15.1. Syntax

a) fail-command:
    fail-keyword.

Example:
a) fail-command:
  FAIL

5.1.15.2. Semantics

a) A fail-command *F* is executed, in an environment *E*, as follows:
The execution of the largest command-suite in which *F* is contained is terminated, and failure, offering *E*, is reported to the proposition or test-refinement that invoked that command-suite.

5.1.16. User-defined-commands

5.1.16.1. Syntax

a) user-defined-command:
FIRST-keyword, optional-tail.

b) tail:
actual-parameter, optional-trailer;
trailer.

c) actual-parameter:
collateral-TYPE-ITEM.

d) trailer:
NEXT-keyword, optional-tail.

e) The FIRST- and subsequent NEXT-keywords must correspond one to one to those of the formal-user-defined-command of one unique how-to-unit.

Examples:
a) user-defined-command:
CLEAN'UP
DRINK me
TURN a UPSIDE'DOWN
PUSH v ON operand'stack

5.1.16.2. Semantics

a) A user-defined-command *U* is executed, in an environment *E*, as follows:
Let *H* be the how-to-unit whose formal-user-defined-command *F* has a FIRST-keyword that is the same as that of *U* (so all keywords of *U* and *F* agree).
Let *H'* be a copy of *H*, shielded (4.0.2.c) for *E*.
Let *H''* be a modified version of *H'*, obtained by consistently substituting, in *H'*, for each occurrence of a tag occurring in a formal-parameter in *H'*, the corresponding parameter of *U*, after enclosing the latter between an open- and a close-sign.
The command-suite of *H''* is executed in a transparent copy (2.2.d) of *E*.

(For example, given

```
HOW'TO TALLY x IN t:
    SELECT:
        x in keys t: PUT t[x]+1 IN t[x]
        ELSE: PUT 1 IN t[x]
```

the user-defined-command *TALLY a, b IN q* is executed by executing

```
SELECT:
    (a, b) in keys (q): PUT (q)[(a, b)]+1 IN (q)[(a, b)]
    ELSE: PUT 1 IN (q)[(a, b)]
```

Moreover, given

```
HOW'TO ADD:
    ALLOW stack
    PUT #stack IN t
    PUT stack[t]+stack[t-1] IN stack[t-1]
    DELETE stack[t]
```

and the following invocation context:

```
FOR t IN addends:
    PUSH t ON stack
    ADD
```

the user-defined-command *ADD* is executed by executing a version in which *t* has been disambiguated (by shielding), such as

```
PUT #stack IN t'
PUT stack[t']+stack[t'-1] IN stack[t'-1]
DELETE stack[t']
```

5.1.17. Refined-commands

5.1.17.1. Syntax

a) refined-command:
  FIRST-keyword.

b) The keyword of a refined-command must occur as the keyword of one command-refinement in the unit (or immediate-command; see 4.b) in which it occurs.

Example:
a) refined-command:
REMOVE'MULTIPLES

5.1.17.2. Semantics

a) A refined-command *R* is executed as follows:
Let *C* be the command-suite of the command-refinement, contained in the same unit as *R*, whose FIRST-keyword is the same as that of the refined-command.
C is executed (in the same environment as *R*).

(Note that neither shielding, nor scratch-pad or even transparent copying occurs. Identifiers are not local to the command-suite of a refinement.)

5.2. Control-commands

5.2.0.1. Syntax

a) control-command:
if-command;
select-command;
while-command;
for-command.

5.2.1. If-commands

5.2.1.1. Syntax

a) if-command:
if-keyword, test, colon-sign, command-suite.

Example:
a) if-command:
IF i < 0: PUT −i, −j IN i, j

5.2.1.2. Semantics

a) An if-command is executed as follows:
The test is tested, offering an environment *E*.
If the test succeeds:
The command-suite is executed in *E*.
Otherwise:
The execution of the if-command is complete.

5.2.2. Select-commands

5.2.2.1. Syntax

a) select-command:
    select-keyword, colon-sign, alternative-suite.

b) alternative-suite:
    increase-indentation, new-line, alternative-sequence,
        decrease-indentation.

c) alternative-sequence:
    single-alternative;
    else-alternative;
    single-alternative, new-line, alternative-sequence.

d) single-alternative:
    test, colon-sign, command-suite.

e) else-alternative:
    else-keyword, colon-sign, command-suite.

Examples:
a) select-command:

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

5.2.2.2. Semantics

a) An alternative-sequence is executed, in an environment *E*, as follows:
Case A: it has a single-alternative:
    The test of the single-alternative is tested, offering an environment *E'*.
    If the test succeeds:
        The command-suite of the single-alternative is executed in *E'*.
    Otherwise:
        If it has a (different) alternative-sequence:
            That alternative-sequence is executed in *E'*.
        Otherwise:
            An error is signalled.
Case B: it has an else-alternative:
    The command-suite of the else-alternative is executed in *E*.

5.2.3. While-commands

5.2.3.1. Syntax

a) while-command:
while-keyword, test, colon-sign, command-suite.

Example:
a) while-command:
WHILE x > 1: PUT x/10, c+1 IN x, c

5.2.3.2. Semantics

a) A while-command is executed, in an environment *E*, as follows:
The test is tested, offering an environment *E'*.
If the test succeeds:
The command-suite is executed in *E'*.
The while-command is executed (again) in *E*.
Otherwise:
The execution of the while-command is complete.

5.2.4. For-commands

5.2.4.1. Syntax

a) for-command:
for-keyword, in-ranger, colon-sign, command-suite.

b) in-ranger:
where TYPE2 lodges TYPE1,
collateral-TYPE1-identifier, in-keyword, TYPE2-expression.

(For "where ... lodges ...", see 5.1.6.1.b.)

Example:
a) for-command:
FOR i, j IN keys t: PUT t[i, j] IN t'[j, i]

5.2.4.2. Semantics

a) A for-command is executed, in an environment *E*, as follows:
The expression of the in-ranger is evaluated, giving a value (text, list or table) *V*.
Let *E'* be a transparent copy of *E*.

The collateral-identifier *I* of the in-ranger is located in *E'*, giving a new location *L*, and the tags with which the domain of *E'* is thereby augmented, are bound.
For each value *X* "generated" (b, c, d) by *V* "for" *I*, in turn:
    X is put in *L*.
    The command-suite is executed in *E'*.

(Note that the expression is evaluated only once, and that the location receives each time the first value not yet treated. So, for example,

```
PUT {1..10} IN a
FOR i IN a:
    WRITE i
    REMOVE max a FROM a
    PUT i+1 IN i
    WRITE i
```

has the same effect as

```
FOR i IN {1..10}:
    WRITE i
    WRITE i+1
PUT {} IN a.)
```

b) The values "generated by" a text *T*, "for" some (irrelevant) collateral-identifier, are the characters of *T*, taken in the order in which they occur in *T*.

c) The values "generated by" a list *L*, "for" some (irrelevant) collateral-identifier, are the list entries of *L*, taken in the order in which they occur in *L*.

d) The values "generated by" a table *T*, "for" some (irrelevant) collateral-identifier, are the associates of the table entries of *T*, taken in the order in which these occur in *T*.

6. Expressions, targets and tests

6.1. Expressions

a) An expression may be "evaluated", giving "its" value. The evaluation of an expression does not have side-effects on the environment in which it is evaluated.

6.1.0.1. Syntax

a) basic-TYPE-expression:
  simple-TYPE-expression;
  TYPE-ADIC-formula.

b) simple-TYPE-expression:
  TYPE-constant;
  TYPE-target-content;
  TYPE-trimmed-text;
  TYPE-table-selection;
  TYPE-display;
  TYPE-refined-expression.

c) tight-TYPE-expression:
  simple-TYPE-expression;
  TYPE-zeroadic-formula;
  open-sign, collateral-TYPE-expression, close-sign.

d) right-TYPE-expression:
  tight-TYPE-expression;
  TYPE-monadic-formula.

Examples:

a) basic-:	b) simple-:	c) tight-:	d) right-expression:
a	*a*	*a*	*a*
$-a$			$-a$
$a+b$			
		$(a+b)$	$(a+b)$

6.1.0.2. Semantics

a) A collateral-expression is evaluated as follows:
The constituent basic-expressions are evaluated, giving values $V_1, \ldots, V_n$.
Case A: $n = 1$:
  Its value is V_1.
Case B: $n > 1$:
  Its value is the compound whose fields are $V_1, \ldots, V_n$.

(The evaluation of basic-expressions is described in the sections below.)

6.1.1. Constants

6.1.1.1. Syntax

A) DIGIT::
    zero; one; two; three; four;
    five; six; seven; eight; nine.

a) numeric-constant:
    exact-constant;
    approximate-constant.

b) exact-constant:
    integral-part, optional-fractional-part;
    integral-part, point-sign;
    fractional-part.

c) integral-part:
    DIGIT-sign;
    integral-part, DIGIT-sign.

d) fractional-part:
    point-sign, DIGIT-sign;
    fractional-part, DIGIT-sign.

e) approximate-constant:
    optional-exact-constant, exponent-part.

f) exponent-part:
    e-keyword, optional-plusminus, integral-part.

g) plusminus:
    plus-sign;
    minus-sign.

Examples:

b) exact-constant:
666
666.
3.14

e) approximate-constant:
2.99793E8
2.99793E+8
E−9

6.1.1.2. Semantics

a) The value of an exact constant *C* is the exact number of which *C* is a conventional decimal representation.

b) The value of an approximate constant *C* is an approximate number that is as close as possible to the value of which *C* is a floating point representation.

6.1.2. Target-contents

6.1.2.1. Syntax

```
a) TYPE-target-content:
        basic-TYPE-identifier.
```

6.1.2.2. Semantics

```
a) A target-content is evaluated as follows:
The basic-identifier is located, giving a location L.
If L has no content:
    An error is signalled.
Otherwise:
    The value of the target-content is the content of L.
```

6.1.3. Trimmed-texts

6.1.3.1. Syntax

```
A) TRIM::
        behead;
        curtail.

a) textual-trimmed-text:
        tight-textual-expression, TRIM-sign, right-numeric-expression.
```

Examples:

```
a) textual-trimmed-text:
   t@p
   t|1
   t|q@p
   t@p|(q-p+1)
```

6.1.3.2. Semantics

```
a) A trimmed-text is evaluated as follows:
The tight-textual-expression and right-numeric-expression are evaluated, giv-
        ing values $T$ and $N$.
If $N$ is not an integer:
    An error is signalled.
Otherwise:
    Let $T$ be the sequence of characters $c_1, \dots, c_n$.
    Let $(B, C)$ be $(N-1, 0)$ if the TRIM-sign is a behead-sign, and $(0, n-N)$
            if it is a curtail-sign.
    If $B < 0$ or $C < 0$ or $B+C > n$:
        An error is signalled.
    Otherwise:
        The value of the trimmed-text is the text
                $c_{B+1}, c_{B+2}, \dots, c_{n-C-1}, c_{n-C}$.
```

(For example, the value of *'nowhere'|3* is *'now'*, and that of *'nowhere'@4* is *'here'*. Note that the user always counts from the left, whereas the Semantics, for curtailing, counts from the right.)

6.1.4. Table-selections

6.1.4.1. Syntax

a) TYPE2-table-selection:
tight-table-with-TYPE1-keys-TYPE2-associates-expression,
TYPE1-key-selector.

b) TYPE1-key-selector:
sub-sign, collateral-TYPE1-expression, bus-sign.

Examples:

a) table-selection:	b) key-selector:
t[i, j]	*[i, j]*

6.1.4.2. Semantics

```
a) A table-selection is evaluated as follows:
The tight-TABLE-expression and key-selector are evaluated, giving values $T$
        and $K$.
If $T$ contains a table entry whose key is $K$:
    The value of the table-selection is the associate of that table entry.
Otherwise:
    An error is signalled.
```

6.1.5. Displays

6.1.5.1. Syntax

A) QUOTE::
  apostrophe;
  quote.

B) CHARACTER::
  space; exclamation; quote; number; dollar; percent; ampersand; apostrophe; open; close; times; plus; comma; minus; point; over; DIGIT; colon; enumeration; less-than; equals; greater-than; question; behead; capital-LETTER; sub; comment; bus; join; underscore; convert; LETTER; curly-open; curtail; curly-close; about.

C) LETTER::
  a; b; c; d; e; f; g; h; i; j; k; l; m;
  n; o; p; q; r; s; t; u; v; w; x; y; z.

a) textual-display:
  QUOTE-sign, optional-style-QUOTE-textual-body, QUOTE-sign.

b) style-QUOTE-textual-body:
  style-QUOTE-textual-item, optional-style-QUOTE-textual-body.

c) style-QUOTE-textual-item:
  unless CHARACTER is QUOTE or CHARACTER is convert,
    CHARACTER-sign;
  where CHARACTER is QUOTE or CHARACTER is convert,
    CHARACTER-image;
  conversion.

d) CHARACTER-image:
  CHARACTER-sign, CHARACTER-sign.

e) conversion:
  convert-sign, collateral-TYPE-expression, convert-sign.

f) LIST-display:
  curly-open-sign, LIST-body, curly-close-sign.

g) LIST-body:
  optional-enumerated-LIST-filler;
  where LIST is list-of-TYPE, where TYPE is numeric or TYPE is textual,
    TYPE-bound, through-sign, TYPE-bound.

(The ambiguity in, e.g., {*1...9*}, is resolved by parsing this as {*1. .. 9*}.)

h) enumerated-NOTION:
  NOTION;
  NOTION, enumeration-sign, enumerated-NOTION.

i) list-of-TYPE-filler:
  TYPE-expression.

j) TYPE-bound:
  TYPE-expression.

k) TABLE-display:
  curly-open-sign, TABLE-body, curly-close-sign.

l) TABLE-body:
  optional-enumerated-TABLE-filler.

m) table-with-TYPE1-keys-TYPE2-associates-filler:
  TYPE1-key-selector, colon-sign, TYPE2-expression.

Examples:

a) textual-display:
  ''
  'He said: "don''t!"'
  "He said: ""don't!"""
  'altitude is `a/1E3` km'

f) LIST-display:
  {}
  {*x1; x2; x3*}
  {*1..n−1*}
  {*'a'..'z'*}

k) TABLE-display:
  {}
  {*[i, j]: 0*}
  {*[0]: {}; [1]: {0}*}
  {*[name]: (m, d, y)*}

6.1.5.2. Semantics

a) A textual-display is evaluated as follows:
The expression of each constituent conversion is evaluated and that value is "converted" (5.1.2.2.b) to a text.
The value of the textual-display is the text obtained by concatening the texts of the constituent textual-items, taken in order, where the text of a CHARACTER-sign or -image consists of the (single) corresponding character, and the text of a conversion is the text into which its value has been converted.

b) A LIST-display is evaluated as follows:
Case A: the body is empty:
Its value is an empty list.
Case B: it contains a constituent enumerated-filler:
All the constituent fillers are evaluated, giving a bag of values.
Its value is then the (sorted) list whose list entries are the values in the bag (so {*1; 1; 2*} and {*1; 2; 1*} give the same list, which is different from {*1; 2*}).
Case C: it contains a constituent through-sign:
The bounds are evaluated, giving values L and U.
If L or U is not an integer or a character:
An error is signalled.
Otherwise, if $U < X < L$, for some integer or character X:
An error is signalled.
Otherwise:
Its value is the list whose list entries are all integers or characters between L and U, including L and U (but if $L = U$, the list has one list entry, and if $U < L$, the list is empty).

c) A TABLE-display is evaluated as follows:
The constituent fillers are evaluated, giving a set of table entries, duplicates, if any, being discarded. (So {*[i]: j; [j]: i*} is lawful, even if *i* and *j* have the same value.)
If two different table entries have the same key:
An error is signalled.
Otherwise:
Its value is the table with those table entries.

d) A TABLE-filler is evaluated as follows:
The key-selector and expression are evaluated, giving values K and T.
Its value is then the "table entry" (1.2.5.a) composed of a key K and an associate T.

6.1.6. Formulas

6.1.6.1. Syntax

a) TYPE-zeroadic-formula:
zeroadic-function.

b) TYPE-monadic-formula:
monadic-function, actual-operand.

c) TYPE-dyadic-formula:
actual-operand, dyadic-function, actual-operand.

d) The parsing ambiguities introduced by rules b and c are resolved by priority rules, as follows:

Let a “priority interval” be a pair of integers (L, H), $L \leqslant H$.
The priority interval of an actual-operand that is a monadic- or dyadic-formula, is the priority interval of the function of that formula;
The priority interval of other actual-operands is (9, 9).
The priority interval of a monadic- or dyadic-function that is a tag is (1, 8).
The priority interval of the other monadic- and dyadic-functions is given by the following tables:

monadic-function	**priority interval**	**dyadic-function**	**priority interval**
~	(8, 8)	+	(2, 2)
+	(8, 8)	−	(2, 2)
−	(5, 5)	*	(4, 4)
*/	(1, 8)	/	(3, 4)
/*	(1, 8)	**	(6, 7)
#	(7, 7)	^	(2, 2)
		^^	(1, 8)
		<<	(1, 8)
		><	(1, 8)
		>>	(1, 8)
		#	(7, 8)

Let the priority interval of the formula be (L_f, H_f).
The left operand, if any, must then have a priority interval (L_o, H_o), such that $L_o \geqslant H_f$.
The right operand must then be a monadic-formula, or have a priority interval (L_o, H_o), such that $L_o > H_f$.

(‘Formulas’ for which this requirement cannot be satisfied, e.g., _a/b/c_, _a/b*c_ and _sin x+y_, are syntactically incorrect. They can be made correct by inserting parentheses, thus: _(a/b)/c_ or _a/(b/c)_, _(a/b)*c_ or _a/(b*c)_, and _(sin x) + y_ or _sin (x+y)_. The function ~ has been given a high priority, so that, e.g., _~0_ behaves as a constant. The function # is 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. Note that, in spite of what ALGOL-habits could make one think, _sin(x)+1_ is as unlawful as _sin x + 1_.)

e) zeroadic-function:
NAMED-tag.

f) monadic-function:
about-sign;
plus-sign;
minus-sign;
numerator-sign;
denominator-sign;
number-sign;
NAMED-tag.

g) dyadic-function:
plus-sign;
minus-sign;
times-sign;
over-sign;
to-the-power-sign;
join-sign;
repeat-text-sign;
adjust-left-sign;
center-sign;
adjust-right-sign;
number-sign;
NAMED-tag.

h) actual-operand:
TYPE-expression.

Examples:

a) zeroadic-formula:	b) monadic-formula:	c) dyadic-formula:
pi	*atan(y/x)*	*x atan y*

6.1.6.2. Semantics

a) A formula F is evaluated, in an environment E, as follows:
The actual-operands are evaluated, giving zero, one or two values $V_1, \ldots, V_n$.
Let U be the yield-unit whose formal-formula contains the same ADIC-function as F.
Let U' be a copy of U, shielded (4.0.2.c) for E.
Let E' be a scratch-pad copy (2.2.e) of E.
The formal-operands of the formal-formula of F' are located in E', giving locations $L_1, \ldots, L_n$.
$V_1, \ldots, V_n$ are put in $L_1, \ldots, L_n$.

The command-suite of U' is executed in E'; the value returned is the value of the formula.

6.1.7. Refined-expressions

6.1.7.1. Syntax

a) TYPE-refined-expression:
  basic-TYPE-identifier.

b) The basic-identifier of a refined-expression must occur as the basic-identifier of one expression-refinement in the unit (or immediate-command; see 4.b) in which it occurs.

Example:
a) refined-expression:
  stack'ptr

6.1.7.2. Semantics

a) A refined-expression is evaluated, in an environment E, as follows:
Let R be the expression-refinement whose basic-identifier is the same as that of the refined-expression.
Let E' be a scratch-pad copy (2.2.e) of E.
The command-suite of R is executed in E'; the value returned is the value of the refined-expression.

6.2. Targets

a) A target may be "located", giving "its" location. Locating a target does not have side-effects on the environment in which it is located.

6.2.0.1. Syntax

a) basic-TYPE-target:
  TYPE-identifier;
  TYPE-trimmed-text-target;
  TYPE-table-selection-target.

6.2.0.2. Semantics

a) A collateral-target is located as follows:
The constituent basic-targets are located, giving locations $L_1, \ldots, L_n$.

Case A: $n = 1$:
    Its location is L_1.
Case B: $n > 1$:
    Its location is the compound location composed of $L_1, \dots, L_n$.

(The locating of basic-targets is described in the sections below.)

6.2.1. Identifiers

6.2.1.1. Syntax

a) basic-TYPE-identifier:
        NAMED-tag.

b) A given tag must not stand, in the same unit (or immediate-command), for a basic-target and/or refined-expression and/or refined-test and/or function and/or predicate.

Examples:
a) basic-identifier:
  i
  supercalifragilisticexpialadocious
  t'
  t"
  man'of'war
  r2d2

6.2.1.2. Semantics

6.2.2. Trimmed-text-targets

6.2.2.1. Syntax

a) textual-trimmed-text-target:
        textual-target, TRIM-sign, right-numeric-expression.

(For "TRIM", see 6.1.3.1.A.)

Examples:
a) trimmed-text-target:
   t@p
   t|1
   t|q@p
   t@p|(q−p+1)

6.2.2.2. Semantics

a) A trimmed-text-target is located as follows:

```
The textual-target is located and the right-numeric-expression is evaluated,
        giving a location $P$ and a value $N$.
If $P$ has no content or $N$ is not an integer:
    An error is signalled.
Otherwise:
    Let the length of (the text which is) the content of $P$ be $n$.
    Let $(B, C)$ be $(N-1, 0)$ if the TRIM-sign is a behead-sign, and $(0, n-N)$
            if it is a curtail-sign.
    If $B < 0$ or $C < 0$ or $B+C > n$:
        An error is signalled.
    Otherwise:
        The location of the trimmed-text-target is the trimmed-text loca-
                tion composed of a parent location $P$, a behead $B$ and
                a curtail $C$.
```

6.2.3. Table-selection-targets

6.2.3.1. Syntax

a) TYPE2-table-selection-target:
        table-with-TYPE1-keys-TYPE2-associates-target, TYPE1-key-selector.

Example:
a) table-selection-target:
   t[i, j]

6.2.3.2. Semantics

a) A table-selection-target is located as follows:
The TABLE-target is located, and the key-selector is evaluated, giving a location *P* and a value *K*;
If *P* has no content:
    An error is signalled.
Otherwise:
    The location of the table-selection-target is the table-selection location composed of a parent location *P* and a key *K*.

6.3. Tests

a) A test may be "tested", whereupon it "succeeds" or "fails". Moreover, it "offers" an environment (in which a command-suite selected by the outcome of the test will be executed). Testing a test does not have side-effects on the environment in which it is tested.

b) If the Semantics defines the outcome of some test as the outcome of some other test, the environment offered is that offered by the other test. Otherwise, if the Semantics does not explicitly state which environment is offered, it is the environment in which the test is tested. (In general, it is a copy of the original environment, augmented by any tags for which a value is given by the testing, as in

WHILE SOME i IN digits HAS i > 9: REMOVE i FROM digits

in which the environment is temporarily augmented with *i*.)

6.3.0.1. Syntax

a) test:
    tight-test;
    conjunction;
    disjunction;
    negation;
    quantification.

b) tight-test:
    open-sign, test, close-sign;
    order-TYPE-test;
    ADIC-proposition;
    refined-test.

c) right-test:
   tight-test;
   negation;
   quantification.

(The testing of tests is described in the sections below.)

6.3.1. Order-tests

6.3.1.1. Syntax

A) ORDER::
   less-than;
   at-most;
   equals;
   unequal;
   at-least;
   greater-than.

a) order-TYPE-test:
   TYPE-expression, ORDER-sign, TYPE-expression;
   order-TYPE-test, ORDER-sign, TYPE-expression.

Examples:
a) order-test:
   (i', j') > (i, j)
   '0' <= d <= '9'
   fa <= f(x) >= fb

6.3.1.2. Semantics

a) An order-test *O* is tested as follows:
Case A: it has two expressions:
   The expressions are evaluated, giving values *V* and *W*;
   Its outcome is the outcome of the comparison of *V* against *W*, using the ORDER-sign.
Case B: it has a (descendant) order-test *O'*:
   O' is tested.
   If *O'* succeeds:
      Let *V* be the value of the (textually) last expression of *O'* (which has been evaluated in the testing of *O'*).
      The expression of *O* is evaluated, giving a value *W*.
      The outcome of *O* is the outcome of the comparison of *V* against *W*, using the ORDER-sign.

Otherwise:
    O fails.

b) The comparison of a value *V* against a value *W*, using an ORDER-sign, succeeds in the following cases:

ORDER-sign	**success if**
<	$V < W$
<=	$V \leqslant W$
=	$V = W$
<>	$V \neq W$
>=	$V \geqslant W$
>	$V > W$

Otherwise, the comparison fails.

6.3.2. Propositions

6.3.2.1. Syntax

a) zeroadic-proposition:
    zeroadic-predicate.

b) monadic-proposition:
    monadic-predicate, actual-operand.

c) dyadic-proposition:
    actual-operand, dyadic-predicate, actual-operand.

d) ADIC-predicate:
    NAMED-tag.

6.3.2.2. Semantics

a) A proposition *P* is tested, in an environment *E*, as follows:
The actual-operands are evaluated, giving zero, one or two values $V_1, \ldots, V_n$.
Let *U* be the test-unit whose formal-proposition contains the same ADIC-predicate as *P*.
Let *U′* be a copy of *U*, shielded (4.0.2.c) for *E*.
Let *E′* be a scratch-pad copy (2.2.e) of *E*.
The formal-operands of the formal-proposition of *U′* are located in *E′*, giving locations $L_1, \ldots, L_n$.
$V_1, \ldots, V_n$ are put in $L_1, \ldots, L_n$.

The command-suite of *U'* is executed in *E'*.
If success is reported, *P* succeeds, and if failure is reported, *P* fails, and the environment offered is *E*.

6.3.3. Refined-tests

6.3.3.1. Syntax

a) refined-test:
NAMED-tag.

b) The tag of a refined-test must occur as the tag of one test-refinement in the unit (or immediate-command; see 4.b) in which it occurs.

Example:
a) refined-test:
special'case

6.3.3.2. Semantics

a) A refined-test is tested, in an environment *E*, as follows:
Let *R* be the test-refinement whose tag is the same as that of the refined-test.
Let *E'* be a scratch-pad copy (2.2.e) of *E*.
The command-suite of *R* is executed in *E'*.
Let the environment offered by the outcome reported be *F*.
Let *E''* be a transparent copy of *E*.
The domain of *E''* is augmented with the bound tags in the domain of *F* not occurring in the domain of *E*, and these tags are made to access the locations they access in *F*.
The outcome of the refined-test is the outcome reported, but offering *E''*.

(This complicated operation on the environments is needed to ensure that the effect of

```
    IF rt: ...
    ...
rt: REPORT some'test
```

is the same as that of

```
    IF some'test: ...
    ...
```

as is the case with substituting the refining expression for the refined-expression if the refinement has only one return-command. For example, the following is allowed:

```
WHILE exception: NORMALIZE
exception: REPORT SOME i IN digits HAS i > 9
NORMALIZE: REMOVE i FROM digits
```

in which the bound *i* temporarily survives the test-refinement to be used in the command-suite governed by the refined-test.)

6.3.4. Conjunctions

6.3.4.1. Syntax

a) conjunction:
        tight-test, and-keyword, conjunct.

b) conjunct:
        right-test;
        conjunction.

Examples:
a) conjunction:

```
a > 0 AND b > 0
i in keys t AND t[i] in keys u AND u[t[i]] <> 'dummy'
```

6.3.4.2. Semantics

a) A conjunction *C* is tested in an environment *E* as follows:
The tight-test is tested, offering an environment *E'*.
If the tight-test fails:
    C fails and offers *E'*.
Otherwise:
    The conjunct is tested in *E'* and its outcome is the outcome of the testing of *C*.

(See the remark at the end of 6.3.5.2.a.)

6.3.5. Disjunctions

6.3.5.1. Syntax

a) disjunction:
        tight-test, or-keyword, disjunct.

b) disjunct:
        right-test;
        disjunction.

Examples:
a) disjunction:
   a <= 0 OR b <= 0
   n = 0 OR s[1] = s[n] OR t[1] = t[n]

6.3.5.2. Semantics

a) A disjunction *D* is tested in an environment *E* as follows:
The tight-test is tested, offering an environment *E'*.
If the tight-test succeeds:
    D succeeds and offers *E'*.
Otherwise:
    The disjunct is tested in *E'* and its outcome is the outcome of the testing of *D*.

(The environment in which *COM* is executed in the context

IF (SOME i IN x HAS p(i)) OR (SOME j IN y HAS q(j)): COM

depends on which of the two quantifications succeeded. However, because of the static content check (7.2.a), neither *i* nor *j* is accessible from *COM*, so the effect is the same.

In the context of

IF (SOME i IN x HAS p(i)) OR (SOME i IN y HAS q(i)): COM

i is filled in either case if *COM* is reached, so it is accessible from there.)

6.3.6. Negations

6.3.6.1. Syntax

a) negation:
        not-keyword, right-test.

Example:
a) negation:
   NOT i in keys t

6.3.6.2. Semantics

a) A negation *N* is tested in an environment *E* as follows:
The right-test is tested, offering an environment *E′*.
If the right-test fails:
    N succeeds and offers *E′*.
Otherwise:
    N fails and offers *E′*.

6.3.7. Quantifications

6.3.7.1. Syntax

A) TEXTUALS::
        textual-textual;
        textual-TEXTUALS.

a) quantification:
        quantifier, ranger, has-keyword, right-test.

b) quantifier:
        each-keyword;
        some-keyword;
        no-keyword.

c) ranger:
        in-ranger;
        parsing-ranger.

d) parsing-ranger:
        TEXTUALS-identifier, parsing-keyword, textual-expression.

(For in-rangers, see 5.2.4.1.b.)

Examples:
a) quantification:
```
EACH i, j IN keys t HAS t[i, j] = t[j, i]
SOME p, q, r PARSING line HAS q in {'. '; '? '; '! '}
NO d IN {2..n−1} HAS n mod d = 0
```

6.3.7.2. Semantics

```
a) A quantification Q is tested, in an environment E, as follows:
Let E′ be a transparent copy of E.
The collateral- or TEXTUALS-identifier I of the in- or parsing-ranger is located
        in E′, giving a new location L, and the tags with which the
        domain of E′ is thereby augmented, are bound.
The expression of the in- or parsing-ranger is evaluated in E, giving a value
        G.
Let K be the keyword of the quantifier.
For each value V "generated" (5.2.4.2.b, c, d, 6.3.7.2.b) by G, "for" I, in turn:
    V is put in L (in E′).
    The right-test is tested in E′, offering an environment E″.
    Case A: K is an each-keyword and the test fails:
        The testing of Q is complete.
        Q fails and offers the environment E″.
    Case B: K is a some-keyword and the test succeeds:
        The testing of Q is complete.
        Q succeeds and offers the environment E″.
    Case C: K is a no-keyword and the test succeeds:
        The testing of Q is complete.
        Q fails and offers the environment E″.
    Other Cases:
        The testing of Q is continued (until G is exhausted).
Case A: K is an each- or no-keyword:
    Q succeeds and offers E.
Case B: K is a some-keyword
    Q fails and offers E.
```

b) The values "generated by" a text *T*, "for" a TEXTUALS-identifier, are all compounds of the type compound-with-TEXTUALS-fields, taken in order, such that the concatenation of the fields of each compound is *T*.

(For example, the values generated by *'abba'* for *p, q, r* are

```
'', '', 'abba'
'', 'a', 'bba'
'', 'ab', 'ba'
'', 'abb', 'a'
'', 'abba', ''
'a', '', 'bba'
'a', 'b', 'ba'
'a', 'bb', 'a'
'a', 'bba', ''
'ab', '', 'ba'
```

```
'ab', 'b', 'a'
'ab', 'ba', ''
'abb', '', 'a'
'abb', 'a', ''
'abba', '', ''
```

in that order, which is the order in which these values would be sorted in a list. The quantification

```
SOME p, q, r PARSING 'abba' HAS p = r
```

would succeed with *p* and *r* set to *''* and *q* to *'abba'*.)

7. The static checks

(The following treatment is informal.)

a) The "static checks" are the "static definedness check", the "static content check" and the "static type check". They are performed on immediate-commands (4.b) and, in a relaxed form, on units.

b) Next to these checks, an implementer should feel free to implement a "static no-nonsense check". This check allows to signal statically as erroneous any construction of which it can be shown that its elaboration would result in a run-time error or an infinite execution, or would make no sense otherwise. (The other static checks may, to some extent, be viewed as special cases of this catch-all check.)

(Since there is no end to the ingenuity that can be put in devising such a check, this document does not attempt to define it. However, some likely candidates are:

```
FOR x IN {'aa'..'zz'}: ...

FOR c IN {'a'..'Z'}: ...

PUT 0, 1 IN a, a

DELETE t@1
```

```
IF t = {}: PUT min t IN m

WHILE t <> {}:
    INSERT min t IN u
    REMOVE min t FROM u
```

The latter is a border-line case (it could be a defiant, though roundabout, way to express one's assuredness that $t = \{\}$), and care should be taken not to declare constructions nonsensical, however weird, that might express a lawful intention of the user. An approach in such cases (e.g., put something in a target that is never inspected again), is to have the system ask the user if this is really her intention. If so, the system will not bother her again.)

c) The static checks are described below in terms of infinite pseudo-algorithms, using "execution paths". However, they can be performed by finite algorithms. Developing fast but simple algorithms for incremental checks is still a matter of research. For pilot implementations these checks, if implemented statically at all, may be performed using (by now) conventional data-flow-analysis techniques, amounting to computing the closures of certain relations.

d) An "execution path" is a finite sequence of simple-commands. Given the set of units, it is possible to characterize, statically, the set of execution paths for a given command. The execution paths may be viewed as the paths in an execution tree that splits at choice points (on tests or on for-commands).

Informally, an execution path consists of successive commands that could be caused, by the execution of the given command, to be executed, assuming that tests succeed or fail at random and that TYPE-expressions return arbitrary values from the TYPE-domain. User-defined- and refined-commands are replaced by the execution paths of their command-suites, and the command-suites of (user-defined) formulas and propositions and of refined-expressions and tests are, in some way, likewise interpolated, all after shielding (4.0.2.c) as necessary. In interpolating the execution paths for formulas, put-commands are inserted to copy the expressions to the formal-operands. Return-commands are replaced by put-commands, putting the returned expression in a fresh basic-target coming in place of the formula. (Propositions can be handled like formulas if a fictitious type "boolean" is introduced). A terminating-command (*QUIT* etc.) finishes, of course, an execution path of a command-suite. Where scratch-pad copying occurs, a put-command of the form *PUT a, b, ... IN a′, b′, ...* is inserted, copying all live basic-targets to fresh basic-targets, and corresponding adjustments are made in the controlled commands. Finally, each read-command *READ t EG e* is replaced by *PUT e IN t*.

7.1. The static definedness check

a) The "static definedness check" for a given command is:
The functions and predicates occurring in formulas and tests in each execution path must be functions and predicates defined, with the same adicity, by some unique unit or by predefinition (and the execution paths are defined, in fact, only by virtue of this fact). Moreover, for each TYPE-formula the yield-unit defining its function has a formal-TYPE0-formula such that there is a polytype refinement (1.b) from TYPE0 to TYPE.

b) Since not all potentially invoked units need be defined at the time of entering a unit (which is what the static definedness check checks), this check should not be applied to units as such. Because of the incomplete knowledge of execution paths then, the following static checks should correspondingly be relaxed for units.

7.2. The static content check

a) The "static content check" for a given command is:
Let initially only those tags be "filled" that occur in the domain of the permanent environment (2.2.a). All other tags are "empty". Proceed along an execution path. Set the status of an empty tag to filled if it occurs as the tag in a basic-target targeted in a put-command or if it occurs as a bound tag (i.e., a tag in the collateral- or TEXTUALS-identifier of a ranger) and the sequel of the execution path takes a turn where the Semantics would put a value in the corresponding location. Set the tag to empty if it is targeted in a delete-command.
It is then required that in all execution paths at all occurrences of a basic-identifier in a target-content, trimmed-text-target or table-selection-target, or in the target of a remove-, insert- or delete-command, the corresponding tag be filled. Moreover, a tag occurring as a bound tag may only occur in commands governed by the construction to which the tag is bound.

(For example, after

```
SELECT:
    0 = 1: PUT '' IN t
    1 = 2: PUT '' IN t
```

t may be considered filled, since execution must pass through one of the alternatives. Also, given

```
HOW'TO FILL it:
    IF 0 = 1:
        PUT '' IN it
        QUIT
    FILL it
```

t may be considered filled after *FILL t*, since execution must have passed through the put-command to leave the body of *FILL*. Of course, the static no-nonsense check might reject these constructions.

The unlawful use of a bound tag occurs in

```
    FOR i IN {1..n}: SET'KEY
    DELETE t[i]
SET'KEY: PUT 0 IN t[i]
```

in which the delete-command is in error. The use of *i* in the refinement is allowed, as long as it is only invoked under the control of constructions in which *i* is bound. Note that the following is also allowed:

```
FOR i IN {1..n}: SET'KEY
FOR i IN {p..q}: SET'KEY
```

as long as the separate bindings have the same type. Finally, the following is not clearly excluded by the present wording, but should be outlawed:

```
FOR i IN {1..m}:
    FOR i IN {1..n}: ....)
```

(Once a collection of units passes the static definedness and type checks, for purposes of the static content test the behavior of each unit can be abstracted to: which parameters and permanent basic-targets are potentially inspected before they receive a content, and which parameters and permanent basic-targets must be filled on leaving the body. Using this abstraction, the effort in performing the static content test for an immediate-command may be greatly reduced. Similar optimizations are possible for the other static checks.)

7.3. The static type check

a) The "static type check" for a given command is:

First, it is required that the process of substituting actual-parameters for formal-parameters in commands in determining the execution paths (7.d) for the given command result everywhere in (syntactically correct) commands. (This should exclude, e.g., a user-defined-command *INCR 0* in the context of

```
HOW'TO INCR i: PUT i+1 IN i
```

since the form *PUT (0)+1 IN (0)* is not a command.)

Second, let initially only those tags "carry" a type that occur in the domain of the permanent environment (2.2.a). The type they carry is the type of the location they access. All other tags are "empty". Proceed along an execution path. Set the type of an empty tag, if it occurs as the tag in a basic-TYPE-identifier, to TYPE, which is required to be a type (and not a proper polytype, i.e., a polytype still containing "poly").

It is then required that in all execution paths at all occurrences of a basic-TYPE-identifier the type of the tag, if previously set, be TYPE.

(This definition is non-constructive in the sense that one may have to "guess" types such that no clash will occur later on, and especially so when units are checked. For an algorithm for the static type check, see MILNER[6].)

A problem may occur by the combination of "{}" and a read-command. A simple example of the problem is given by

```
READ x EG {}
```

and a more complicated one by

```
HOW'TO TYPE'PUZZLE:
    PUT {[1]: ('a', {})} IN a
    READ x EG a[1]
```

By the check as formulated until now, these are allowed. Even at run-time, however, the type of the expression to be read will be unknown. Therefore, such cases are not allowed. They are also caught by the static type check. (The treatment of "{}" requires some special care anyway, but should cause no insuperable difficulties.)

APPENDIX. PRELIMINARY THOUGHTS ON THE *B* SYSTEM

The language *B* is embedded in a system that is *fully* dedicated to *B*, without any exceptions. If the system is embedded again in a larger operating system, that fact should be invisible to the user as far as possible. (An OS may have nasty features that cannot be hidden.) In particular, a user should and need not be aware of peculiar animals such as load modules, file systems etc.

If the *B* system is embedded in a larger system and it should be possible to transfer data between the systems, this should be done by means of an interface that displays to the *B* user a face in the spirit of *B*.

A.1. Command language

Usually, a general purpose system is addressed in some lingo called "JCL" or "command language". The ideal is that the command language of the *B* system is *B* itself. It is a matter of research to see to what extent this ideal can be reached.

One part is relatively easy. Assume a how-to-unit has been defined:

```
HOW'TO ADVENTURE: ...
```

The "command" to invoke this unit would simply be the *B* command

```
ADVENTURE
```

just as it might occur in a program:

```
HOW'TO PLAY:
     READ name EG ''
     SELECT:
          name = 'adventure': ADVENTURE
          ...
```

There is no need for the user to request "compilation"; to the user the system behaves as if the computer has *B* as its code.

This means that a *B* system can be used as a glorified calculator, since all commands are usable as "JCL" commands:

```
System:  At your command
User:    PUT 5 IN n
System:  At your command
User:    FOR i IN {1..n}:
             WRITE i*i
System:  1 4 9 16 25
         At your command
```

If there is no compelling reason why the request for a specific system action should not be a *B* command, it is preferable to introduce a new simple-command. Not all requests to the system to perform some action, however, can be *B* commands. All such requests are given such a form that it is entirely obvious that they cannot be part of a command-suite. The requests might be given by pressing special "function keys", or at least be of a form radically different from that of the commands. Requests are always obeyed immediately. This also solves (largely) the problem of having different "modes".

A.2. Files

Among the global, permanent objects in a given workspace—how-to-, yield- and test-units—there are also variables. This means that we can use normal *B* values instead of traditional "files". Character files may be modelled in *B*, e.g., by a table of texts whose keys are *{1..n}*.

In many cases, however, a file is conceptually not a sequence of text lines, but a structured object. Processing such a file often requires parsing to detect a structure that was known to the program that created the file. In such cases, it is much simpler to keep the original structure.

Tables are quite similar to indexed files, and the fields of a compound give a simple way to model record fields.

A.3. The editor

Let us assume that part of the hardware configuration is the screen of a not too dumb terminal. Although the *B* system should not have different "modes", the history of keys that have been struck up till now has a bearing on the interpretation of further key strokes. This current "status" should be clear from the contents of the screen. A reserved status bottom line may be helpful (and can also be put to further uses.)

Some keys must allow easy editing of existing objects. Those, in particular, should not be command-like; otherwise, the editor needs a command and a text mode. Functions like listing (displaying) an object, available in most operating systems both as an editor capability and at the command language level, exist only once. The editor is also a pretty-printer, displaying objects in a canonic representation (0.4.2.e).

The editor is also invoked for interactive input; to the user, it should make little difference whether *{['linda']: (2, 29, 72); ['john']: (4, 1, 74); ['peter']: (7, 16, 75)}* is entered as part of a command, or in response to the prompt from a read-command *READ birthday EG {['name']: (31, 12, 99)}*. In particular, even in the latter case *all* of the input should be available for editing as long as the whole has not been finally, and irrevocably, entered. A difference is that in a read-command the editor knows the type of the expression to be entered, so it can offer more editorial assistance.

Because the whole system is dedicated to *B*, the editor is cognizant of the syntax, and (relevant) semantics, of *B*. If we distinguish between the abstract syntax of *B*, which defines more or less tree-like forms, and the concrete syntax giving a linearized representation, the typical situation might be that editing is (conceptually, at least) performed on the *abstract* form, even though a concrete representation is continually displayed.

Advantages are:
- The user will obtain a keen feeling for the abstract syntax underlying the concrete representations.
- The opportunity for making syntax errors is greatly reduced.
- The editor will immediately point out most other syntax errors.
- The number of key strokes is significantly reduced, since the editor immediately fills out redundant information. (This is important in view of the verbosity of concrete *B*.)

The following ideas on "screen editing" are tentative. Only a pilot implementation can show how to make a smooth and friendly system.

Instead of the usual cursor, we have a *focus* of attention. The focus corresponds to a node (with offspring) in the syntax tree. It is displayed in some way distinguishing it from the context (e.g., brighter).

In the concrete representation the focus may consist of non-contiguous elements. (E.g.,

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

or

IF x > 0 **AND** *y > 0* **AND** *z > 0: ....*)

Special keys allow the user to shift the focus in the tree, e.g., Up, Down, Left and Right (where Up and Down, especially, bear no relationship to the usual cursor up and cursor down).

The editor uses its knowledge of the *B* language to help minimize key strokes. On entering a command, the editor tries to guess the kind of command from the first letters of the first keyword. For example, after entering "*P*", the screen has already

PUT □ *IN* □ .

(The boxes □ stand for a dummy node.) If the next key entered would be an "*R*", the screen would change to

PRINT □

(assuming the user has defined a print-command). Sticking to the case of "*PUT* □ *IN* □", the Down or Right key puts the focus on the first box. A subsequent Right focusses on the second box.

If there is no right node on the same level, a sufficient number of Ups is assumed before a Right (and similarly for Left). Down goes down on the left-most branch if the current "direction", determined by the last Right/Left given, is Right, and on the right-most branch if the current direction is Left. If there is no down branch, a sufficient numbers of Right/Lefts, in the current direction, is assumed before a Down. To complicate the issue, a Down immediately following an Up is equivalent to giving a Right/Left instead of that Up+Down.

To give an example, consider a tree with three branches, each having a node with again three branches. The linearized tree is represented thus:

abc def ghi

Starting from a situation where the focus is on the *a*, subsequent Rights will give the following sequence:

a_bc_ _def_ _ghi_
_a_**b**_c_ _def_ _ghi_
_ab_**c** _def_ _ghi_
abc **def** _ghi_
abc _def_ **ghi**

after which the next Right leaves the tree. Starting from a situation where the focus is on the whole tree, subsequent Downs will produce:

abc **def** **ghi**
abc *def* *ghi*
a*bc* *def* *ghi*
abc **def** *ghi*
abc **d***ef* *ghi*
abc *def* **ghi**
abc *def* **g***hi*

after which the next Down leaves the tree.

On entering a syntactic construction with a variable number of constituents such as a command suite, the editor behaves as follows. Assume the user enters "*I*". The screen displays

```
IF □: □.
```

If the user enters now Down, followed by "*i > 0*", the screen has

```
IF i > 0□: □.
```

A Right focusses on the right box. The command is now automatically opened up, i.e., displayed over several lines:

```
IF i > 0:
    □
```

After entering a "*P*", e.g., the situation becomes

```
IF i > 0:
    PUT □ IN □
    □
```

with focus on the put-command, but with a new box so that a Right or Down, after finishing the put-command, e.g., in the following situation:

```
IF i > 0:
    PUT i−1 IN i
    □
```

does not leave the command-suite prematurely. An Up, however, now finishes the if-command. Since the command-suite contains only one short command, it closes up and the screen displays

```
IF i > 0: PUT i−1 IN i.
```

An Insert key creates a new box node in the tree at a position such that the last Up/Down/Left/Right given would have focused there, had the node already been present. The ambiguity if there are several such positions is resolved by the editor, using its knowledge of the syntax.

A Suck key allows to delete the focus. It is not irretrievably lost, but pushed on a stack. A box is left, so that no Insert is needed if the purpose is to change the node. That box is volatile, however, if part of a construction with an arbitrary number of constituents: each subsequent action (including another Suck) makes it disappear.

A Squirt pops the stack at boxes, assuming Inserts if necessary. The direction of Squirts is reversed, so that a repeated Suck followed by a repeated Squirt does what one should hope. (This mechanism is not general enough, since stacks are too well-behaved.)

Finally, there is some way to Undo unwanted changes without retyping.

It will take a user some time to get proficient at such a system. Luckily, the syntax of *B* is largely such that the editor is able to accept input by a user who ignores the tree-structure and simply enters the linearized concrete representation (see GEURTS & MEERTENS[4]). For example, in *WRITE* □, an "*(*" results in *WRITE (*□*)*. Next, "*i+j*" results in *WRITE (i+j*□*)*. The "normal" way to proceed would be an Up or Right, but a "*)*" is also accepted, with the same effect: *WRITE (i+j)*□.

A.4. Illegal constructions

The editor does more than context-free syntax checking. In particular, it checks the type-consistency of the whole work-space incrementally as and when units are entered. The editor refuses attempts to enter unlawful constructions that violate the context-free syntax. If the construction is wrong because of an inconsistency with other constructions, but acceptable from a context-free viewpoint, the editor protests, but the user may override it. (The actual error may be in the other constructions.) The inconsistent parts are displayed, e.g., in red.

For example, in the situation

```
HOW'TO INCR x:
    PUT □ IN □
    □
```

with focus on the first box, the editor refuses all of the keys

! $ % &) , : ; < = > ? @ [\ ] ^ _ ` | }

since an expression may not start with these. In the situation

```
HOW'TO INCR x:
    PUT x+1 IN z□
    □
```

it protests against a "*/*", since *z* is un unlawful target here (being uninitialized). If the user, however, Resets, the next "*/*" is obeyed, but *z* is displayed in red. If the user then inserts *ALLOW z*, the red *z* receives the normal color. A request to display the headings of the current units in the workspace will display them in red if they could not be invoked as they stand.

An attempt to execute an unlawful command (not all errors can be caught on entering, and a command may still contain boxes) will also bring the focus on the offending part. A Help will explain what is wrong (in understandable terms).

An important case is given by refinements. These are usually recognizable to the editor because the refined-command, -expression or -test is normally undefined without corresponding refinement. If properly treated, the user will have an automatic reminder of the next step to be done (keeping track of loose ends). For example, when the user has entered the tag *entries* in

```
YIELD aliases name:
    ALLOW address'list
    FOR i IN entries: □
    □
```

the editor will already display

```
YIELD aliases name:
    ALLOW address'list
    FOR i IN entries: □
    □
entries: RETURN □
```

REFERENCES

[1] CLEAVELAND, J.C. & R.C. UZGALIS, Grammars for Programming Languages, Elsevier, 1977.

[2] GEHANI, N., Generic procedures: an implementation and an undecidability result, Computer Languages **5** (1980) 155-161.

[3] GEURTS, L.J.M. & L.G.L.T. MEERTENS, Designing a beginners' programming language, *in* New Directions in Programming Languages 1975, 1-18, (S.A. Schuman, ed.), IRIA, Rocquencourt, 1976.

[4] GEURTS, L.J.M. & L.G.L.T. MEERTENS, Keyword grammars, *in* Implementation and Design of Algorithmic Languages, 1-12, (J. André & J.-P. Banâtre, eds), IRIA, Rocquencourt, 1978.

[5] MEERTENS, L.G.L.T., Issues in the design of a beginners' programming language, *in* Algorithmic Languages, 167-184, (J.W. de Bakker & J.C. van Vliet, eds), North-Holland, 1981.

[6] MILNER, R., A theory of type polymorphism in programming, Journal of Computer and System Sciences **17** (1978) 348-375.

[7] VAN WIJNGAARDEN, A., & al., Revised Report on the Algorithmic Language ALGOL 68, Acta Informatica **5** (1975) 1-236.

INDEX