AB30.3.2 ALGOL-N⁺


Introduction

The authors of MR93 contributed remarkably to the development of WG2.1 project on prospected ALGOL, realizing in a unified system a huge amount of diverse demands on its power of expression. That document, however, is far from being readable because of the too complicated mechanism of the language and its description. Simplification should be made without essentially spoiling the power and applicability of the language.

We, members of a working group in Nippon (Japan), present here a design of simplified language, tentatively named ALGOL N, with a simplified method of description. It is designed to be an algorithmic language, not distorted by compiler oriented contrivances, though the efficiency of compilation is not totally neglected. We hope that we have achieved some improvement, particularly in the following points at least.

Plainness of syntax description:

Though we recognize the theoretical importance of the method used for the description of syntax in MR93, we find it preferable, from a practical point of view, to use another method due to T. Simauti, previously reported to the WG2.1 and WG2.2 meetings, summer 1968, which reproduces the ALGOL 60 style of description, modified and enriched by a few conventions; two of the new conventions, continned dots for 'and so on' and metalinguistic parentheses, are borrowed from usage in mathematics which are familiar to many people. It enables us to describe our syntax very plainly without non-essential construction of intermediate notions.

Compactness of syntax description:

The syntax of ALGOL N is designed to meet the requirement that it should be learned very quickly by a beginner and become the firm foundation of further understanding. It is extremely simple and compact; it needs even less space than ALGOL 60 syntax. This is achieved by dealing only with the broad framework and the subjects of general character. Details such as coincidence of type are treated separately, soon after the main syntax. Many particular subjects such as assignment statement, conditional expression, etc. are to be found among standard declarations, not in the description of syntax. Such separation of particular subjects serves a great deal to simplify the whole structure of language.
Rigor of semantics:
Simplification of syntax causes simplification of semantics which, together with some conceptual rearrangement such as strict separation of notation and value, facilitate a uniform and rigorous description of semantics for various syntactic elements. A reader can easily follow the rigorous construction of elaboration and can find, if acquainted with MR93, how the bewildering concept of scope is made unnecessary in ALGOL N.

Types above modes:
It is observed that the single levelled specification by modes has caused a lot of inconvenience and immaterial complexity. We distinguish it into two levels, namely the classification of values by "types", e.g. real, and the specification by 'modes' (in the new sense), e.g. integer. A mode is, so to say, contained in just one type, and there is given a 'rounding' or 'projection' map from (the set of values of) the type to (that of) the mode, such as the usual rounding of real number to integer. Types are subject to syntactic analysis, but modes are not. Thus a 'procedure' is concerned with parameters belonging to fixed types, not to fixed modes; its action, however, may deal with the mode of some parameter, as is the case of assignment, where the source value is rounded to the mode of the destination.

Simplified system of types:
Corresponding to the system of composite modes in MR93, we have a much simpler system of composite types. The essential simplicity is that there is no united type and no composite type of reference style. All references are of the same type, independent of what is referred to. This prevents any possibility of looped type. Inconvenience caused by the lack of united types is more than compensated by 'form' declarations, which is a generalization and modification of operation declaration.

Coercion eliminated:
Coercions are not considered in the principal part of ALGOL N, since they are partly dissolved in projections and the most popular coercions will spoil the simplicity of the general framework if they are taken in. They are treated as a matter of declarations, and so a matter of adaptation which a programmer may choose. Some are found in standard declarations, but the invocation can be prevented or modified.

S. Igarashi: Research Institute for Mathematical Sciences, Kyoto University.
T. Iwamura: Faculty of Science, Rikkyo University.
K. Sakuma: Research Institute for Mathematical Sciences, Kyoto Univ.
T. Simauti: Faculty of Science, Rikkyo University.
T. Simizu: College of General Education, Tokyo University.
S. Takasu: Research Institute for Mathematical Sciences, Kyoto Univ.
E. Wada: Faculty of Engineering, Tokyo University.
N. Yoneda: Faculty of Science, Gakushuin University.
I. Nakata (Observer): Central Research Laboratory, Hitachi Ltd.
1. Syntax

1.1 Meta-Language for Syntax Description

To describe the syntax of ALGOL N we use the following meta-language, which is a simple modification of the Backus notation.

1.1.1 Meta-symbols: =, (, ), [ , ] , { , } , | , / , ....

1.1.2 Meta-constants: begin, ;, (, a, etc.

Those are basic symbols of ALGOL N.

1.1.3 Meta-variables: <expression>, <variable>, <procedure call> etc. Those are used to represent the form of syntactical elements.

1.1.4 Meta-expressions:

A meta-expression represents the forms of figures which are finite sequences of basic symbols, and are defined recursively as follows:

1) Let \( \alpha \) stand for a meta-constant. Then

\[
\alpha
\]

is a meta-expression.

A figure \( \phi \) is of the form \( \alpha \), if and only if \( \phi \) consists of just one basic symbol \( \alpha \).

2) Let \( \alpha \) stand for a meta-variable. Then

\[
\alpha
\]

is a meta-expression.

A figure \( \phi \) is of the form \( \alpha \), if and only if \( \phi \) is of the form meta-variable \( \alpha \).

3) Let \( \alpha \) stand for a meta-expression. Then

\[
(\alpha)
\]

is a meta-expression, and is used to express the precedence of connections.

A figure \( \phi \) is of the form \( (\alpha) \), if and only if \( \phi \) is of the form \( \alpha \).

4) Let \( \alpha \) stand for a meta-expression. Then

\[
[\alpha]
\]

is a meta-expression.

A figure \( \phi \) is of the form \( [\alpha] \), if and only if \( \phi \) is the empty figure or is of the form \( \alpha \).
Let \( \alpha, \beta \) stand for meta-expressions. Then
\[
\alpha \beta
\]
is a meta-expression.

A figure \( \phi \) is of the form \( \alpha \beta \), if and only if \( \phi \) is the concatenation of a figure of the form \( \alpha \) and a figure of the form \( \beta \).

Let \( \alpha, \beta \) stand for meta-expressions. Then
\[
\alpha \mid \beta
\]
is a meta-expression.

A figure \( \phi \) is of the form \( \alpha \mid \beta \), if and only if either \( \phi \) is of the form \( \alpha \) or \( \phi \) is of the form \( \beta \).

Let \( \alpha \) stand for a meta-expression. Then
\[
\alpha ...
\]
is a meta-expression.

A figure \( \phi \) is of the form \( \alpha \ldots \), if and only if either \( \phi \) is of the form \( \alpha \) or \( \phi \) is the concatenation of a figure of the form \( \alpha \ldots \) and a figure of the form \( \alpha \).

Let \( \alpha, \beta \) stand for meta-expressions. Then
\[
\alpha \{ \beta \} ...
\]
is a meta-expression.

A figure \( \phi \) is of the form \( \alpha \{ \beta \} \ldots \), if and only if either \( \phi \) is of the form \( \alpha \{ \beta \} \ldots \), a figure of the form \( \alpha \{ \beta \} \ldots \), a figure of the form \( \beta \), and a figure of the form \( \alpha \).

Let \( \alpha, \beta \) stand for meta-expressions. Then
\[
\alpha / \beta
\]
is a meta-expression.

A figure \( \phi \) is of the form \( \alpha / \beta \), if and only if \( \phi \) is of the form \( \alpha / \beta \).

The ranking of the priorities of connections is as follows:

- **first**: \( \alpha \ldots, \alpha \{ \beta \} \ldots \)
- **second**: \( \alpha \beta \)
- **third**: \( \alpha \mid \beta, \alpha / \beta \)
1.1.5 Meta-statement:

A meta-statement is used to define a meta-variable. Let \( \alpha \) stand for a meta-variable, and \( \beta \) stand for a meta-expression. Then

\[
\alpha = \beta
\]

is a meta-statement. This meta-statement represents the sentence:

"A figure \( \phi \) is of the form of \( \alpha \), if and only if \( \phi \) is of the form \( \beta \)."

In the following, we shall say simply

"\( \phi \) is \( \alpha \)" instead of "\( \phi \) is of the form \( \alpha \)."

1.2 Standard Language of ALGOL N

\[
\begin{align*}
\langle \text{expression} \rangle & = \langle \text{secondary} \rangle / \\
& \quad \langle \text{form call} \rangle \\
\langle \text{secondary} \rangle & = \langle \text{primary} \rangle / \\
& \quad \langle \text{array element} \rangle / \\
& \quad \langle \text{structure element} \rangle / \\
& \quad \langle \text{procedure call} \rangle \\
\langle \text{primary} \rangle & = \langle \text{variable} \rangle / \\
& \quad \langle \text{go to statement} \rangle / \\
& \quad \langle \text{dummy statement} \rangle / \\
& \quad \langle \text{code call} \rangle / \\
& \quad \langle \text{closed expression} \rangle / \\
& \quad \langle \text{block} \rangle / \\
& \quad \langle \text{notation} \rangle \\
\langle \text{notation} \rangle & = \langle \text{effect notation} \rangle / \\
& \quad \langle \text{real notation} \rangle / \\
& \quad \langle \text{bits notation} \rangle / \\
& \quad \langle \text{string notation} \rangle / \\
& \quad \langle \text{reference notation} \rangle / \\
& \quad \langle \text{array notation} \rangle / \\
& \quad \langle \text{structure notation} \rangle / \\
& \quad \langle \text{procedure notation} \rangle \\
\langle \text{declaration} \rangle & = \langle \text{variable declaration} \rangle / \\
& \quad \langle \text{form declaration} \rangle / \\
& \quad \langle \text{mark declaration} \rangle \\
\langle \text{go to statement} \rangle & = \langle \text{go to} \rangle / \langle \text{label} \rangle \\
\langle \text{dummy statement} \rangle & = \langle \text{dummy} \rangle \\
\langle \text{code call} \rangle & = \langle \text{code} \rangle \left( ( \langle \text{selector} \rangle \langle \text{expression} \rangle ) [ , ] \ldots \right) \\
& \quad \langle \text{primary typifier} \rangle \text{ by } \langle \text{code body} \rangle
\end{align*}
\]
<closed expression> == ( <expression> )

<block> == begin [ <declaration>; ] ... 
((<label>; ) ... <expression>){;}{;} ...
end

<array element> == <secondary> [ <expression> ]

<structure element> == <secondary> [ <selector> ]

<procedure call> == <secondary> ( [ <expression>{,} ] ... )

<form call> == [ <expression> ] <mark> [ <expression> ] <mark> ... 
<effect notation> == effect

<real notation> == real <real modifier> <real donor>

<real modifier> == [ [ [ <expression> ] : [ <expression> ] :
[ <expression> ] ] ]

<real donor> == [ <number> ]

<number> == <integer donor> | <fraction donor>

<integer donor> == <digit> ...

<fraction donor> == [ <digit> ... ] ( . <digit> ... )

<real modifier> == [ [ ( exact | varying ) [ <expression> ] ] ]

<real donor> == [ <bits> ]

<bits> == ( 0 | 1 ) ...

<string notation> == string <string modifier> <string donor>

<string modifier> == [ [ ( exact | varying ) [ <expression> ] ] ]

<string donor> == [ <string> ]

<string> == [ [ non' ] [ <string> ] ] ...

<reference notation> == reference <reference donor>

<reference donor> == [ nil ]

<array notation> == array <array modifier> <primary>
array ( <expression> {,} ... )

<array modifier> == [ [ <expression> : <expression> ] ]

<structure notation> == structure [ [ <selector> <expression> {,} ... ] ]

<procedure notation> == procedure [ [ <typifier> {,} ... ] ]
<primary typifier> <procedure donor>
<procedure donor> = \([\text{by}] (([\text{variable}] \{,\} \ldots)) \langle\text{expression}\rangle\)

<variable declaration> = \(\text{let} \langle\text{variable}\rangle = \langle\text{expression}\rangle\)

<form declaration> = \(\text{let} \langle\text{form}\rangle \text{ represent} \langle\text{expression}\rangle\)

<form> = \([\text{typifier}] \langle\text{mark}\rangle \langle\text{typifier}\rangle \langle\text{mark}\rangle \ldots\)

<mark declaration> = \(\text{let} \langle\text{mark}\rangle \text{ operate} \langle\text{left priority}\rangle \langle\text{right priority}\rangle\)

<left priority> = \([\text{before}] (\langle\text{mark}\rangle \{,\} \ldots) \langle\text{all}\rangle \langle\text{left}\rangle\)

<right priority> = \([\text{after}] (\langle\text{mark}\rangle \{,\} \ldots) \langle\text{all}\rangle \langle\text{right}\rangle\)

<typifier> = \langle\text{expression}\rangle

<primary typifier> = \langle\text{primary}\rangle

<variable> = \langle\text{identifier}\rangle

<label> = \langle\text{identifier}\rangle \_i

<selector> = \langle\text{letter}\rangle \langle\text{letter}\rangle \langle\text{digit}\rangle \ldots

<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

<digit> = 0|1|2|3|4|5|6|7|8|9

<basic symbol> = \langle\text{non" } \rangle | " "

<non" \rangle = \langle\text{delimiter}\rangle | \langle\text{letter}\rangle \langle\text{digit}\rangle | \ldots | 0 | 1

<delimiter> = \langle\text{standard symbol}\rangle | \langle\text{extension symbol}\rangle | \langle\text{mark}\rangle

<standard symbol> = \langle\text{begin}\rangle \langle\text{end}\rangle | \{\} | \[\] | \langle\text{left}\rangle \langle\text{right}\rangle | \langle\text{before}\rangle \langle\text{left}\rangle \langle\text{after}\rangle | \langle\text{right}\rangle \langle\text{effect}\rangle | \langle\text{real}\rangle | \langle\text{bits}\rangle | \langle\text{string}\rangle | \langle\text{reference}\rangle | \langle\text{array}\rangle | \langle\text{structure}\rangle | \langle\text{procedure}\rangle | \langle\text{precision}\rangle | \langle\text{exact}\rangle | \langle\text{varying}\rangle | \langle\text{nil}\rangle | \langle\text{code}\rangle | \langle\text{by}\rangle | \langle\text{let}\rangle | \langle\text{be}\rangle | \langle\text{represent}\rangle | \langle\text{operate}\rangle | \langle\text{all}\rangle | \langle\text{go to}\rangle | \langle\text{dummy}\rangle ; \ : \ \_i$

<extension symbol> = \langle\text{integer}\rangle | \langle\text{none}\rangle | \langle\text{comment}\rangle | \langle\text{silent}\rangle | \langle\text{2 array}\rangle | \ldots

<mark> = _i := \langle\text{if}\rangle \langle\text{then}\rangle \langle\text{else}\rangle | \langle\text{for}\rangle | \langle\text{do}\rangle | \langle\text{from}\rangle | \langle\text{step}\rangle | \langle\text{until}\rangle | \langle\text{while}\rangle | \langle\text{case}\rangle | \langle\text{of}\rangle | \langle\text{+}\rangle | \langle\text{-}\rangle | \langle\text{\#}\rangle | \langle\text{complex}\rangle | \langle\text{copy}\rangle | \langle\text{enproc}\rangle | \langle\text{mode}\rangle | \langle\text{length}\rangle | \langle\text{has}\rangle | \langle\text{type}\rangle | \langle\text{as}\rangle | \langle\text{type}\rangle | \langle\text{etc.}\rangle

<code body> is not specified.

1.3 Extensions

1.3.1 When \(E_i\) is empty or an \(\langle\text{expression}\rangle\) for \(i = 1, 2\), "real \([E_1 : : E_2]\)" may be replaced by "\(\text{integer} \ [E_1 : E_2]\)".
1.3.2 "integer [ ; ]" may be replaced by "integer"

1.3.3 When J is a <number>, "real J" may be replaced by "J".

1.3.4 "precision" may be omitted.

1.3.5 When J is a <bits>, "bits J" may be replaced by "J".

1.3.6 When J is a <string>, "string J" may be replaced by "J".

1.3.7 "exact" may be omitted.

1.3.8 "reference nil" may be replaced by "nil"

1.3.9 "array (" may be replaced by "[]".

1.3.10 "structure (" may be replaced by "{".

1.3.11 When $E_1$ and $E_2$ are <expression>'s, "$[E_1 : E_2] array [" may be replaced by "array $[E_1 : E_2] "$.

1.3.12 When $E_1$ and $E_2$ are <expression>'s, "$[E_1 : E_2] array array" may be replaced by "array $[E_1 : E_2] array$."

1.3.13 "]" may be replaced by "", ".

1.3.14 "array array" may be replaced by "2 array". "array array array" may be replaced by "3 array". etc.

1.3.15 ") effect" may be replaced by ")".

1.3.16 " effect" may be replaced by "\)

1.3.17 When $V_1, ..., V_n$ are <variable>'s and $E$ is an <expression>,

"let $V_1$ be $E$;
let $V_2$ be $E$;

... ...

let $V_n$ be $E$;"

may be replaced by

"let $V_1, V_2, ..., V_n$ be $E$;"

1.3.18 Let $V$ be a <variable> or a sequence of <variable>'s separated by commas, and $E$ be an <expression>. If the right-most symbol of $E$ is

"effect, real, bits, string, reference, nil, end, ], ), \) or \" then "let $V$ be $E$;" may be replaced by "EV;".

1.3.19 Let $T_i$ be a <typifier> for $i = 1, 2, ..., n$;
$P_i$ be a sequence of <mark>'s for $i = 1, 2, ..., n$;
empty or a sequence of <mark>'s for $i = 0, n$.

"let $P_0(T_1)P_1(T_2)P_2...P_{n-1}(T_n)P_n$ represent" may be replaced by

"let $P_0()P_1()P_2...P_{n-1}()P_n$ represent"
1.3.20 When G is a sequence of <mark>'s and parentheses, and \( E_1, \ldots, E_n \) are <expression>'s,

\[
\text{"let G represent } E_1; \\
\text{let G represent } E_2; \\
\text{.........} \\
\text{let G represent } E_n;"
\]

can be replaced by

\[
\text{"let G represent } E_1, E_2, \ldots, E_n;"
\]

1.3.21 When \( P_1, \ldots, P_n \) are <mark>'s and \( Z \) or \( Z' \) is a <left priority> or a <right priority> respectively,

\[
\text{"let } P_1 \text{ operate } Z Z'; \\
\text{let } P_2 \text{ operate } Z Z'; \\
\text{.........} \\
\text{let } P_n \text{ operate } Z Z';"
\]

can be replaced by

\[
\text{"let } P_1, P_2, \ldots, P_n \text{ operate } Z Z';"
\]

1.3.22 "before left" may be replaced by "before none left"

1.3.23 "after right" may be replaced by "after none right".

1.3.24 Let \( n \) be an integer \((\geq 0)\), \( U_i \) be a <basic symbol> other than comment and silent for \( i = 1, 2, \ldots, n \).

"comment \( U_1 \ldots U_n \) silent" may be inserted between two symbols.

2. Static Structures of Programs

2.1 Syllables.

We consider a program as an <expression> which is a figure consisted of a finite sequence of <basic symbol>'s. A program is divided to several number of subsequences called "syllables". A syllable is of the form <identifier>, <number>, <bits>, <string>, <code body> or <delimiter>. In a program two syllables other than <delimiter>'s must be separated by one or more <delimiter>'s. Under these conditions, the division of a program is unique. In the following, a sequence of syllables of the form \( a \) is called "to be \( a \) in the program" or simply "to be \( a \)", where \( a \) is a meta-variable. Each <identifier> in a program is used either as a <variable> or as a <label> or as a part of a <selector>. 
2.2 Block-Structures and Declarations

For each <block> and <procedure notation>, we define its proper <variable>'s, proper <form>'s, proper <mark>'s, proper <label>'s and proper interior:

2.2.1 Let E be a <block> of the form

\[ \text{"begin } D_1; \]
\[ \text{....} \]
\[ D_n; \]
\[ L_1: \ldots : L^1_1 : E_1; \]
\[ \text{....} \]
\[ L^k_1: \ldots : L^k_k : E_k \text{ end}" \]

with <declaration>'s \( D_1, \ldots, D_n \), <label>'s \( L_1^1, \ldots, L^k_1 \), \( \ldots, L^k_k \), <expression>'s \( E_1, \ldots, E_k \), where \( n \) is an integer \( (\geq 0) \), \( k \) is an integer \( (\geq 1) \), \( i_1, \ldots, i_k \) are integers \( (\geq 0) \). Let \( i \) be an integer, and \( 1 \leq i \leq n \).

1) If \( D_i \) is a <variable declaration> of the form

\[ \text{"let } V_i \text{ be } F_i" \]

with a <variable> \( V_i \) and an <expression> \( F_i \), then \( V_i \) is a proper <variable> of \( E \), and we say that "(<declaration> in the program) \( D_i \) is a <declaration> for \( V_i \)."

2) If \( D_i \) is a <form declaration> of the form

\[ \text{"let } G_i \text{ represent } F_i" \]

with a <form> \( G_i \) and an <expression> \( F_i \), then \( G_i \) is a proper <form> of \( E \), and we say that "\( D_i \) is a <declaration> for \( G_i \)."

3) If \( D_i \) is a <mark declaration> of the form

\[ \text{"let } P_i \text{ operate } Z_i Z_i'" \]

with a <mark> \( P_i \), a <left priority> \( Z_i \) and a <right priority> \( Z_i' \), then \( P_i \) is a proper <mark> of \( E \), and we say that "\( D_i \) is a <declaration> for \( P_i \)."

Furthermore,

3.1) If \( Z_i \) is of the form

\[ \text{"before } P_1, \ldots, P_m \text{ left"} \]

then we say that "\( D_i \) is a reverse <declaration> for the ordered pair \( <P_j', P_i>" \text{ for } j=1,2,\ldots,m."
3.2) If $Z_i$ is of the form
"before all left",
then "$D_i$ is a reverse <declaration> for the pair $<P', P_i>$ " for each <mark> $P'$.

3.3) If $Z_i'$ is of the form
"after $P_1$, ..., $P_m$ right",
then "$D_i$ is a reverse <declaration> for the pair $<P_i, P_j'>$ " for $j=1, 2, ..., m$.

3.4) If $Z_i'$ is of the form
"after all right",
then "$D_i$ is a reverse <declaration> for the pair $<P_i, P_j'>$ " for each <mark> $P'$.

4) $L_1^1, ..., L_m^1, ..., L_k^1, ..., L_k^k$ are proper <label> 's of E.

2.2.2 Let E be a <procedure notation> of the form
"procedure (T_1, ..., T_n)T J"
with <typifier> 's $T_1, ..., T_n$, T and <procedure donor> J, where n is an integer ($\geq 0$).
If J is empty, then E has no proper <variable> 's.
If J is of the form
"by ((V_1, ..., V_n) F)"
with <variable> 's $V_1, ..., V_n$ and an <expression> F, then $V_1, ..., V_n$ are proper <variable> of E.
A <procedure notation> has neither proper <form> 's, no proper <mark> 's, and no proper <label> 's.

2.2.3 Let E be a <block> or a <procedure notation> in a program, and let F be an <expression>, or a <label>, or a <declaration> in that program. If E is a <block> and F is a subsequence of E, then we say that "F is in the interior of E". If E is a <procedure notation> of the form
"procedure (T_1, ..., T_n)T J"
with <typifier> 's $T_1, ..., T_n$, T and a <procedure donor> J, and F is a subsequence of J, then we say that "F is in the interior of E". If F is in the interior of E and there is no <block> or <procedure notation> E', such that E' is in the interior of E, and F is in the interior of E', then we say that "F is in the proper interior of E".
2.2.4 Let A be a \textit{variable} (or a \textit{form} or a \textit{mark}), and E be a \textit{block} or a \textit{procedure notation} (in a program). And let F be the minimum \textit{block} or \textit{procedure notation} (in the program), which contains E in its interior and A is its proper \textit{variable} (or \textit{form} or \textit{mark}), and D be a \textit{declaration} (in the program) which is a \textit{declaration} of A and is in the proper interior of F. Then we say that "D is a \textit{declaration} of A in the interior of E", or "A in the proper interior of E is declared on F". Let L be a \textit{label}, and E be a \textit{block} or a \textit{procedure notation} (in a program). And let F be the minimum \textit{block} or \textit{procedure notation} (in the program), which contains E in its interior and L is its proper \textit{label}. Then we say that "L in the proper interior of E is declared on F".

Let P, P' be \textit{mark} 's, and E be a \textit{block} or a \textit{procedure notation} (in a program). And let F be the minimum \textit{block} or \textit{procedure notation} (in the program), on which P or P' in the proper interior of E is declared. Furthermore D be a \textit{declaration} of either P or P' and is in the proper interior of F. Then we say that "D is a \textit{declaration} of the pair \(P, P'\) in the proper interior of E". If there is a reverse \textit{declaration} of \(P, P'\), which is a \textit{declaration} of \(P, P'\) in the proper interior of E, then we say that " \(P, P'\) is reverse in the interior of E", and in other cases " \(P, P'\) is natural in the proper interior of E".

2.3 Parsing of Expressions

The parsing of an expression is syntactically unique except for constructions of \textit{form call} 's. To obtain the complete uniqueness, we restrict \textit{mark declaration} 's and the construction of \textit{form call} 's as follows:

(R1) In the proper interior of each \textit{block\(\)}, there must be at most one \textit{declaration} for each \textit{mark}.

(R2) For each \textit{mark} P in a program, there must be a \textit{declaration} (in the program) by which P is declared or P is declared by a standard \textit{declaration}.

Therefore each \textit{mark} P in a program has just one (standard or not standard) \textit{declaration} D by which P is declared.

Let D be of the form

"let P operate ZZ""

where Z is a \textit{left priority} and Z' is a \textit{right priority}.

If both Z and Z' are not empty, then P is called "be independent".

If Z is not empty and Z' is empty, then P is called "be initial".

If Z is empty and Z' is not empty, then P is called "be terminal".

If both Z and Z' are empty, then P is called "be connecting".

Let an \textit{expression} in a program G be a \textit{form call} of the form

"E P E P E .... E P E"  
\begin{array}{c} o \ 1 \ 1 \ 2 \ 2 \ \ n-1 \ n \ n \end{array}
where \( n' \) is an integer (\( \geq 1 \)),

\[ P_i' \] is a \(<\text{mark}>\) for \( i = 1, 2, \ldots, n' \),

\[ E_i' \] is empty of an \(<\text{expression}>\) for \( i = 0, 1, \ldots, n' \),

then \(<P_n', P_1'>\) must be natural.

**(R4.2)** If \( E_n' \) is a \(<\text{form call}>\) of the form

\[ E_0''P_1''E_1''P_2''E_2''\ldots E_{n''-1}''P_n''E_n'' \]

where \( n'' \) is an integer (\( \geq 1 \)),

\[ P_i'' \] is a \(<\text{mark}>\) for \( i = 1, 2, \ldots, n'' \),

\[ E_i'' \] is empty or an \(<\text{expression}>\) for \( i = 0, 1, \ldots, n'' \),

then \(<P_n'', P_1''>\) must be reverse.

Those restrictions (R3.1), (R3.2), (R4.1) and (R4.2) are also applied for every \(<\text{typifier}>\)'s and \(<\text{form}>\)'s as \(<\text{expression}>\)'s.

### 2.4 Direct Constituents of Expressions

Let \( E \) and \( E' \) be \(<\text{expression}>\)'s.

\( E \) is said to embrace \( E' \) if and only if \( E \) is of the form

\[ AE'B \]

where \( A \) and \( B \) are figures and at least one of them is non-empty.

\( E' \) is called a direct constituent of \( E \) if and only if the following three conditions are satisfied.

1) \( E \) embraces \( E' \);
2) \( E \) embraces no \(<\text{expression}>\) which embraces \( E' \);
3) \( E' \) is used neither as a \(<\text{typifier}>\) nor as a \(<\text{prime typifier}>\) in the construction of \( E \).

### 2.5 Types

**2.5.1** Types are defined recursively as follows:

1) \( \text{effect} \) is a type
2) \( \text{real} \) is a type
3) \( \text{bits} \) is a type
4) \( \text{string} \) is a type
5) \( \text{reference} \) is a type
6) Let \( T \) be a type, then \( \text{array T} \) is a type, and called array style.
7) Let \( n \) be an integer (\( \geq 0 \)); \( S_i \) be a \(<\text{selector}>\) different from each other, for \( i = 1, 2, \ldots, n \), and \( T_i \) be a type for \( i = 1, 2, \ldots, n \); then \( \text{structure} (S_1T_1, \ldots, S_nT_n) \) is a type, and called structure style.
8) Let $n$ be an integer ($\geq 0$);

$T_i$ be a type for $i = 1, 2, \ldots, n$;

and $T$ be a type; then

procedure $(T_1, \ldots, T_n)T$ is a type, and called procedure style.

We shall use the following notations:

$T_{array}$: The set $\{ T \mid T$ is a type of array style $\}$. 

$T_{structure}$: The set $\{ T \mid T$ is a type of structure style $\}$. 

$T_{procedure}$: The set $\{ T \mid T$ is a type of procedure style $\}$. 

Let $T$ stand for an arbitrary type, then $T$ is of the form some typifier. We shall denote this typifier simply by $T$.

In a legal program, each expression and each form has its type. And some semantical notion (quantity, value and mode) has its type.

Let $A$ be an expression, form, quantity, value or mode, then we shall denote its type by $t(A)$.

2.5.2 To define the types' of expressions', we introduce some restrictions:

(R5.1) In the proper interior of each block in a program, there must be at most one declaration for each variable.

(R5.2) For each procedure donor in a program of the form

"by \(((V_1, \ldots, V_n)E)\)"

with variables' $V_1, \ldots, V_n$ and an expression $E$, $V_1, \ldots, V_n$ must be different from each other.

(R6) Each variable in a program must be declared by a declaration in the program, or by a standard declaration.

Further restrictions on types are introduced recursively with the definition of the types of expressions'.

The type of an expression $E$ in a program is abstracted by the form of $E$ and types of expressions' contained in $E$. Those types of sub expressions' are abstracted from left to right in the contextual order.

(R7) By this process, the type of each expression must be able to be defined.

1) In the beginning of the type abstraction of a block (in a program) $E$, each variable declaration and form declaration are processed from left to right.

1.1) Let a variable declaration be of the form

"let $V$ be $F$"

with a variable $V$ and an expression $F$.

Then the type of $E$ ($t(F)$) is abstracted, and $t(F)$ is represent the type of a variable (in the program) of the form $V$ and declared on $E$. 

1.2) Let a form declaration \( D \) be of the form

\[
\text{"let } G \text{ represent } F \text{"}
\]

with a form \( G \) and an expression \( F \), and let \( G \) be of the form

\[
E_0 P_1 E_1 P_2 E_2 \ldots \ldots E_{n-1} P_n E_n
\]

where \( n \) is an integer \( \geq 1 \),

\( P_i \) is a mark for \( i = 1, 2, \ldots, n \),

\( E_i \) is empty or an expression for \( i = 0, 1, \ldots, n \).

Let \( T_i \) stand for \( t(E_i) \) if \( E_i \) is an expression,

empty if \( E_i \) is empty, for \( i = 0, 1, \ldots, n \).

Then the figure

\[
\text{(} T_0 P_1 T_1 P_2 T_2 \ldots \ldots T_{n-1} P_n T_n \text{)}
\]

is called the operator form of \( G \), and we say that \( D \) is a declaration for this operator form. And the figure, which is made from the operator form of \( G \) eliminating all marks 's and insert a comma "", between each succession of two types, is called the argument - types of \( G \).

(R8) In this case, \( t(F) \) must be procedure style, and if \( t(F) \) is of the form

\[
\text{procedure } (T_1, \ldots, T_n) T
\]

with types \( T_1, \ldots, T_n, T \), then the argument-types of \( G \) must be \( (T_1, \ldots, T_n) \). And we say that in this declaration the result type of \( G \) is \( T \).

2) In the case of a procedure notation (in a program) of the form

\[
\text{"procedure } (T_1, \ldots, T_n) T \text{ by } ((V_1, \ldots, V_m) E)"
\]

with expression 's \( T_1, \ldots, T_n, T, E \) and variable 's \( V_1, \ldots, V_m \).

(R9) In this case, \( m \) must be \( n \), and \( t(E) \) must be \( t(T) \).

\( t(T_i) \) is represent the type of a variable (in the program) of the form \( V_i \) and declared on \( E \), for \( i = 1, 2, \ldots, n \).

2.5.3 Let \( E \) be a block or procedure notation (in a program) and \( 0 \) be an operator form.
If \( F \) is the minimum \(<\text{block}>\) (in a program) which contains \( E \) (or \( E \) itself), and \( \bar{O} \) is declared by a \(<\text{declaration}>\) \( D \) in its proper interior, then we say that "\( \bar{O} \) in the proper interior of \( E \) is declared on \( F \)" or "\( \bar{O} \) in the proper interior of \( E \) is declared by \( D \)".

(R10) In the proper interior of each \(<\text{block}>\) (in a program) there must be at most one \(<\text{declaration}>\) for each operator form.

2.5.4

1) Let \( E \) be a \(<\text{variable}>\) \( V \) in a program. The type of \( V \) \((t(V))\) is defined as above.

2) Let \( E \) be a \(<\text{go to statement}>\) or \(<\text{dummy statement}>\). Then \( t(E) \) is \text{effect}.

3) Let \( E \) be a \(<\text{code call}>\) of the form

\[
\text{"code (S}_1E_1, \ldots, S_nE_n)\text{ by (A)} \]

with \(<\text{selector}>\) \( S_1, \ldots, S_n \), \(<\text{expression}>\) \( E_1, \ldots, E_n, T \), and \(<\text{code body}>\) \( A \).

Then, \( t(E) \) is \( t(T) \).

(R11) In this case, \( S_1, \ldots, S_n \) must be different from each other.

4) Let \( E \) be a \(<\text{closed expression}>\) of the form

\[
\text{"(F)"} \]

with an \(<\text{expression}>\) \( F \). Then \( t(E) \) is \( t(F) \).

5) Let \( E \) be a \(<\text{block}>\) of the form

\[
\text{"begin } D_1; \ldots ; D_n; \text{ L}_1; \ldots ; \text{ L}_k; E_1; \ldots ; \text{ L}_k; \text{ E}_k \text{ end }" \]

with \(<\text{declaration}>\) \( D_1, \ldots, D_n \), \(<\text{label}>\) \( L_1, \ldots, L_{k1}, \ldots, L_{kk} \), \(<\text{expression}>\) \( E_1, \ldots, E_k \).

Then \( t(E) \) is \( t(E_k) \).

6) Let \( E \) be an \(<\text{array element}>\) of the form

\[
\text{"F[E']"} \]

with \(<\text{expression}>\) \( F \) and \( E' \).

(R12) In this case, \( t(F) \) must be array style, and \( t(E') \) must be \text{real}.  

\[\text{AB30 p 53}\]
Let \( t(F) \) be of the form

\[
\text{array } T
\]

with a type \( T \). Then \( t(E) \) is \( T \).

7) Let \( E \) be a \(<\text{structure element}>\) of the form

\[
[F[S]]
\]

with an \(<\text{expression}>\) \( F \) and a \(<\text{selector}>\) \( S \).

(R13) In this case, \( t(F) \) must be structure style, and when \( t(F) \) is of the form

\[
\text{structure } (S_1 T_1, \ldots, S_n T_n)
\]

with \(<\text{selector}>\)'s \( S_1, \ldots, S_n \) and types \( T_1, \ldots, T_n \).

And \( n \) must be \( >1 \), and \( S \) must be one of \( S_1, \ldots, S_n \).

If \( S \) is \( S_i (1 \leq i \leq n) \), then \( t(E) \) is \( T_i \).

8) Let \( E \) be a \(<\text{procedure call}>\) of the form

\[
[F(E_1, \ldots, E_n)]
\]

with \(<\text{expression}>\)'s \( F, E_1, \ldots, E_n \).

(R14) In this case, \( t(F) \) must be procedure style, and when \( t(F) \) is of the form

\[
\text{procedure } (T_1, \ldots, T_m)T
\]

with types \( T_1, \ldots, T_m, T \), and \( m \) must be \( n \), and \( t(E_i) \) must be \( T_i \) for \( i = 1, 2, \ldots, n \).

Then \( t(E) \) is \( T \).

9) Let \( E \) be a \(<\text{form call}>\) of the form

\[
E_0 P_1 E_1 P_2 E_2 \ldots E_{n-1} P_n E_n
\]

where \( m \) is an integer (\( \geq 1 \)),

\( P_i \) is a \(<\text{mark}>\) for \( i = 1, 2, \ldots, n \),

\( E_i \) is \(<\text{expression}>\) or an \(<\text{expression}>\) for \( i = 0, 1, \ldots, n \).

and be in the proper interior of a \(<\text{block}>\) or \(<\text{procedure notation}>\) \( E' \).

Let \( T_i \) stand for \( t(E_i) \) if \( E_i \) is an \(<\text{expression}>\), empty if \( E_i \) is empty, for \( i = 1, 2, \ldots, n \), and let \( U \) stand for the operator form

\[
(T_0 P_1 T_1 P_2 T_2 \ldots T_{n-1} P_n T_n)
\].
In this case, $U$ in the proper interior of $E'$ must be declared by a <declaration> in the program or by a standard <declaration>.

Let $D$ be the <declaration> for $0$ in the proper interior of $F$, of the form

\[
\text{"let } G \text{ represent } F\"
\]

with a <form> $G$ and an <expression> $F$, and let $T$ be the result type of $G$.

Then $t(E)$ is $T$.

10) The type of a <effect notation> is effect.

11) The type of a <real notation> is real.

(R16) In a <real modifier> of the form

\[
[ E_1 : E_2 : E_3 ] \quad \text{or} \quad [ \text{precision } E_4 ]
\]

if $E_i$ is an <expression>, then $t(E_i)$ must be real for $i = 1, 2, 3, 4$.

12) The type of a <bits notation> is bits.

(R17) In a <bits modifier> of the form

\[
[\text{exact } E_1 ] \quad \text{or} \quad [\text{varying } E_1 ]
\]

if $E_1$ is an <expression> then $t(E_1)$ must be real.

13) The type of a <string notation> is string.

(R18) In a <string modifier> of the form

\[
[\text{exact } E_1 ] \quad \text{or} \quad [\text{varying } E_1 ]
\]

if $E_1$ is an <expression> then $t(E_1)$ must be real.

14) The type of a <reference notation> is reference.

15) Let $E$ be a <array notation> of the form

\[
\text{"array } HJ \"
\]

with an <array modifier> $H$ and an <expression> $J$.

Then $t(E)$ is array $t(J)$.

(R19) In a <array modifier> of the form

\[
[ E_1 : E_2 ]
\]

with <expression> $'s$, $t(E_1)$ and $t(E_2)$ must be real.
Let $E$ be a array notation of the form

\[
\text{array} \left( E_1, \ldots, E_n \right)
\]

with expression $'s \ E_1, \ldots, E_n$.

(R20) In this case, $t(E_1), t(E_2), \ldots, t(E_n)$ must be equal.

Then $t(E)$ is array $t(E_1)$.

17) Let $E$ be a structure notation of the form

\[
\text{structure} \left( S_1 E_1, \ldots, S_n E_n \right)
\]

with selector $'s \ S_1, \ldots, S_n$ and expression $'s \ E_1, \ldots, E_n$.

(R21) In this case, $S_1, \ldots, S_n$ must be different from each other.

Then $t(E)$ is structure $\left( S_1 \ t(E_1), \ldots, S_n \ t(E_n) \right)$.

18) Let $E$ be a procedure notation of the form

\[
\text{procedure} \left( T_1, \ldots, T_n \right) \ TJ
\]

with typifier $'s \ T_1, \ldots, T_n, T$, and procedure donor $J$.

Then $t(E)$ is procedure $\left( t(T_1), \ldots, t(T_n) \right) t(T)$.

2.6 Legal Programs

A program is called legal, if it suffices the restrictions (R1) - (R21) and the following (R22) and (R23).

(R22) For each block in the program of the form

\[
\text{begin} \ D_1 ; \ldots ; D_n ; \ L_1 : \ldots : L_{i_1} : E_1 ; \ldots ; L_{i_k} : \ldots : L_{i_k} \ E_k \ 	ext{end}
\]

with declaration $'s \ D_1, \ldots, D_n$;

label $'s \ L_1, \ldots, L_{i_1}, \ldots, L_{i_1}, \ldots, L_{i_k}$;

and expression $'s \ E_1, \ldots, E_k$;

$L_{i_1}, \ldots, L_{i_1}, \ldots, L_{i_k}, \ldots, L_{i_k}$ must be different from each other.

(R23) For each block or procedure notation $E$ in the program, and for each label $L$ in the proper interior of $E$, there must be a block or procedure notation in the program, which contains $E$ and on which $L$ declared.
3. Semantical Notions

3.1 Quantities

Quantities are abstract elements, and are introduced for describing the course of the elaborations of expressions. Each quantity has its mode, type and value. Let Q be a quantity, then, we shall denote its mode by \( m(Q) \), type by \( t(Q) \) and value by \( w(Q) \). Then it holds

\[ t(Q) = t(m(Q)) = t(w(Q)). \]

pragmatics

Let \( V \) be a variable, and \( E \) be an expression. In a course of a normal program, if \( V \) has its ability "able", then \( V \) has its quantity denoted by \( q(V) \). As the result of the elaboration of \( E \), we shall obtain a quantity \( Q' \) or a label \( L \). For describing such conclusion, we use the notation

\[ e(E) \Rightarrow Q' \]

or

\[ e(E) \Rightarrow L \]

respectively. end of pragmatics

3.2 Values

Values are classified according to their types (or their styles) as follows:

3.2.1 effect type.

There is a sole value done in the effect type.

3.2.2 real type.

A value of the real type is a real number. We shall use the following notations:

\[ \mathbb{R} : \text{the set of all real numbers.} \]

\[ \mathbb{I} : \text{the set of all integers, in the sense of the subset of } \mathbb{R}. \]

\[ \text{round}(R) : \text{the integer obtained by rounding } R, \text{ where } R \text{ is a real number.} \quad (\text{round}(R) = \text{entier}(R+0.5).) \]

3.2.3 bits type.

A value of the bits type is a bit-string. Bit-strings are defined with its length \( (\in \mathbb{I}) \), recursively as follows:
1) $\epsilon$ is a bit-string of length 0.

2) $\mathbf{0}$ is a bit-string of length 1.

3) $\mathbf{1}$ is a bit-string of length 1.

4) Let $B$ be a bit-string of length $n$ ( $\geq 1$). $B\mathbf{0}$ is a bit-string of length $n+1$.

5) Let $B$ be a bit-string of length $n$ ( $\geq 1$). $B\mathbf{1}$ is a bit-string of length $n+1$.

We shall use the following notations:

$\mathbf{B}$ : the set of all bit-string.

$\mathbf{B}_I$ : the set of all bit-string whose length is $I$, where $I$ is an integer ( $\geq 0$).

length($B$) : length of $B$, where $B$ is a bit-string.

3.2.4 string type.

A value of the string type is a <string>. <string>'s are defined with its length ( $\in \mathbb{N}$ ), recursively as follows:

1) " " is a <string> of length 0.

2) Let $n$ be an integer ( $\geq 1$); and $A_i$ be a <basic symbol> other than " and ", or a <string> , and if $A_i$ is a <basic symbol> then let $m_i$ stand for 1, if $A_i$ is a <string> of length m then let $m_i$ stand for $m+2$, for (i=1,2,....,n); then

$A_1A_2...A_n$

is a string of length $m_1+m_2+...+m_n$.

We shall use the following notations:

$\mathbf{C}$ : the set of all <string>.

$\mathbf{C}_I$ : the set of all <string> whose length is $I$, where $I$ is an integer ( $\geq 0$).

length($C$) : the length of $C$, where $C$ is a <string>.

3.2.5 reference type

A value of the reference type is the empty set $\emptyset$ or a set (Q) with a sole element Q, where Q is a quantity.
3.2.6 **array** style.

Let T be a type of the form

\[ \text{array } T' \]

where T' is a type.

1) The Empty set \( \emptyset \) is a value of type T.
2) Let \( v \) be an integer, \( u \) be an integer \((\geq v)\), and \( Q_i \) be a quantity of type T', for \( i = v, v+1, \ldots, u \).

Then the set

\[ \{ <v, Q_v>, <v+1, Q_{v+1}>, \ldots, <u, Q_u> \} \]

is a value of type T. \((<v, Q>\) denotes the ordered pair of \( v \) and \( Q \)).

3.2.7 **structure** style

3.2.7.1 Let T be a type of the form

\[ \text{structure } (S_1, T_1, \ldots, S_n, T_n) \]

where \( n \) is an integer \((\geq 1)\);

- \( S_i \) is a selector for \( i = 1, 2, \ldots, n \);
- \( T_i \) is a type for \( i = 1, 2, \ldots, n \).

Let \( Q_i \) be a quantity of type \( T_i \), for \( i = 1, 2, \ldots, n \). Then the set

\[ \{ <S_1, Q_1>, <S_2, Q_2>, \ldots, <S_n, Q_n> \} \]

is a value of type T.

3.2.7.2 Let T be a type of the form

\[ \text{structure } () \]

There is a sole value, the empty set \( \emptyset \), in the type T.

3.2.8 **procedure** style.

Let T be a type of the form

\[ \text{procedure } (T_1, \ldots, T_n)T' \]

where \( n \) is an integer \((\geq 0)\);

- \( T_i \) is a type for \( i = 1, 2, \ldots, n \);
- T' is a type.
Let $V_i$ be a <variable> different from each other, for $i=1,2,\ldots,n$, and let $E$ be an <expression>; then

$$(V_1,\ldots,V_n)E$$

is a value of type $T$.

### 3.3 Mode

Modes and their types are defined recursively as follows:

1) **effect** is a mode of type **effect**.

2.1) Let $R_i$ be a real number for $i=1,2,3$, then

$$\text{real } [R_1:R_2:R_3]$$

is a mode of type **real**.

2.2) Let $R$ be a real number, then

$$\text{real } [\text{precision } R]$$

is a mode of type **real**.

3.1) Let $I$ be an integer, then

$$\text{bits } [\text{exact } I]$$

is a mode of type **bits**.

3.2) Let $I$ be an integer, then

$$\text{bits } [\text{varying } I]$$

is a mode of type **bits**.

4.1) Let $I$ be an integer, then

$$\text{string } [\text{exact } I]$$

is a mode of type **string**.

4.2) Let $I$ be an integer, then

$$\text{string } [\text{varying } I]$$

is a mode of type **string**.

5) **reference** is a mode of type **reference**.

6) Let $I_1$ be an integer for $i=1,2$; and let $T$ be a type; then

$$\text{array } [I_1:I_2]T$$

is a mode of type **array** $T$.

7) Let $T$ be a type of structure style, then $T$ is a mode of type $T$.

8) Let $T$ be a type of procedure style, then $T$ is a mode of type $T$. 
pragmatics

Let Q be a quantity. Q has its mode, and we shall denote it by m(Q). end of pragmatics

A mode specifies a domain of values. Let M be a mode. The domain of values specified by M is denoted by \( W(M) \),

and is defined as follows:

3.3.1 \( W(\text{effect}) \) is \{ done \}.

3.3.2.1 Let \( R_i \) be a real number for \( i = 1, 2, 3 \). Then, \( W(\text{real } [R_1;R_2;R_3]) \) is the finite set
\[
\{ x \mid x \in \mathbb{R} \land R_1 \leq x < R_3 \land \text{there exist an integer } y \text{ such that } x = y \times R_2 \}.
\]

3.3.2.2 Let \( R \) be a real number. Then \( W(\text{real } [\text{precision } R]) \) is some finite set of real numbers which satisfies following conditions:

For the sake of simplicity, we represent \( W(\text{real } [\text{precision } R]) \) by \( W_1 \).

a) If \( 0 \neq x \in W \) and \( 0 \neq y \in W \) and \( x < y \) and there are no elements \( z \) of such that \( x < z < y \), then
\[
y - x < \frac{1}{2} (|x| + |y|) \times |R|.
\]
b) There exists a positive number in \( W \) with a sufficiently large absolute value.
c) There exists a negative number in \( W \) with a sufficiently large absolute value.
d) There exists a positive number in \( W \) with a sufficiently small absolute value.
e) There exists a negative number in \( W \) with a sufficiently small absolute value.

(The meaning of the adverb "sufficiently" is unspecified.)

3.3.3 Let \( I \) be an integer.

\( W(\text{bits } [\text{exact } I]) \) is \( B_I \) if \( I \geq 0 \),
\( \emptyset \) if \( I < 0 \).

\( W(\text{bits } [\text{varying } I]) \) is \( B_0 \cup B_1 \cup \ldots \cup B_I \) if \( I \geq 0 \),
\( \emptyset \) if \( I < 0 \).

3.3.4 Let \( I \) be an integer

\( W(\text{string } [\text{exact } I]) \) is \( C_I \) if \( I \geq 0 \),
\( \emptyset \) if \( I < 0 \).

\( W(\text{string } [\text{varying } I]) \) is \( C_0 \cup C_1 \cup \ldots \cup C_I \) if \( I \geq 0 \),
\( \emptyset \) if \( I < 0 \).
3.3.5 \( W \) (reference) is \( \{ \emptyset \} \cup \{ \{ q \} \mid q \in Q \} \).

3.3.6 Let \( I \) be an integer, \( I' \) be an integer, and let \( T \) be a type.

Then \( W(\text{array } [I:I'] T) \) is
\[
\{ \langle i, q_i \rangle, \langle i+1, q_{i+1} \rangle, \ldots, \langle I', q_{I'} \rangle \mid q_i \in Q \land (q_i) = T, \\
\text{for } i = I, I+1, \ldots, I' \} \text{ if } I \leq I', \\
\emptyset \text{ if } I > I'.
\]

3.3.7 Let \( n \) be an integer \( (\geq 0) \); \( S_i \) be a selector different from each other, and \( T_i \) be a type for \( i = 1, 2, \ldots, n \). Then \( W(\text{structure } (S_1 T_1, \ldots, S_n T_n)) \) is
\[
\{ \langle S_1, q_1 \rangle, \langle S_2, q_2 \rangle, \ldots, \langle S_n, q_n \rangle \mid q_i \in Q \land (q_i) = T_i, \\
\text{for } i = 1, 2, \ldots, n \}
\]

3.3.8 Let \( n \) be an integer \( (\geq 0) \); \( T_i \) be a type for \( i = 1, 2, \ldots, n \); and \( T \) be a type. Then \( W(\text{procedure } (T_1, \ldots, T_n) T) \) is the set
\[
\{ (V_1, \ldots, V_n) \mid \text{E is variable for } i = 1, 2, \ldots, n \land E \text{ is expression without mark } \land \\
\text{"begin let } V_1 \text{ be } T_1; \\
\text{.....} \\
\text{let } V_n \text{ be } T_n; \\
\text{E end"} \\
\text{is a legal expression} \}\}
\]

3.4 Implementation Dependent Factor

When we are concerned with a particular implementation, it is usual that not all values are realized in the implementation. So, the domain of values are restricted, and biased in the form of implementation dependent. In the following, we use the notation \( W_M \) for such an implementation dependent set, transformed from \( W(M) \). In each implementation, modes are classified by the coincidence of the set \( W_M \). And we shall denote the representative of the class, which contains a mode \( M \), by \( c(M) \).

We shall use the following notations:

R1: An (implementation dependent) fixed negative real number with sufficiently large absolute value. It acts as a proxy in a <real modifier> of the form \([E_1; E_2; E_3]\) when \( E_1 \) is absent.
R2: An (implementation dependent) fixed positive real number with sufficiently large absolute value. It acts as a proxy in a <real modifier> of the form \([E_1:E_2:E_3]\) when \(E_3\) is absent.

R3: An (implementation dependent) fixed positive real number with sufficiently small absolute value. It acts as a proxy in a <real modifier> of the form \([\text{precision } E]\) when \(E\) is absent.

I1: An (implementation dependent) fixed positive integer. It acts as a proxy in a <bits modifier> of the form \([\text{exact } I]\) when \(I\) is absent.

I2: An (implementation dependent) fixed positive integer (usually \(\geq I_1\)). It acts as a proxy in a <bit modifier> of the form \([\text{varying } I]\) when \(I\) is absent.

I3: An (implementation dependent) fixed positive integer. It acts as a proxy in a <string modifier> of the form \([\text{exact } I]\) when \(I\) is absent.

I4: An (implementation dependent) fixed positive integer (usually \(\geq I_3\)). It acts as a proxy in a <string modifier> of the form \([\text{varying } I]\) when \(I\) is absent.

3.5 Projection

When a value is assigned for a quantity, it is adjusted (or rounded) for the quantity's mode. Let \(W\) be a value, and let \(M\) be a mode of the same type with \(W\). We shall denote such an adjusted value by \(p(M,W)\) or \(p_W(W)\), and call it "the projection of \(W\) for \(M\)." Obviously it suffices that \(p_W(W)\in W\), and \(p_W(W)\) is not defined if \(W\neq W\). Since the set \(W\) is implementation dependent, \(p_M\) is implementation dependent, too. So the following directions are not compulsory, though the implementors and users are suggested to refer to it.

3.5.1 \(p(\text{effect},\text{done})\) is \text{done}.

3.5.2 Let \(M\) be a mode of real type, and let \(R\) be a real number. Then \(p(M,R)\) is a real number in \(W\) which is nearest to \(R\).

3.5.3 Let \(I\) be an integer (\(\geq 0\)), and let \(B\) be a bit-string. Then \(p(\text{bits } [\text{exact } I],B)\) is

\[
\begin{cases}
  \text{if } I=0; \\
  0 \ldots 0 \text{ if } I>0 \text{ and } B \text{ is } \epsilon; \\
  I \\
  b_1 \ldots b_I \text{ if } I>0 \text{ and } B \text{ is } b_1 \ldots b_n \text{ where } n=\text{length}(B) \geq I, \text{ and } b_i \text{ is } 0 \text{ or } 1 \text{ for } i=1,2,\ldots,n; \\
  \text{ and } b_1 \ldots b_{I-n} \ldots 0 \text{ if } I>0 \text{ and } B \text{ is } b_1 \ldots b_n \text{ where } n=\text{length}(B)<I, \\
  b_{I-n} \ldots 0 \text{ and } b_i \text{ is } 0 \text{ or } 1 \text{ for } i=1,2,\ldots,n.
\end{cases}
\]
p(bits varying I ,B) is
\[ \begin{cases} 
\epsilon & \text{if } I=0 \text{ or } B \text{ is } \epsilon; \\
b_1 \ldots b_i & \text{if } I > 0 \text{ and } B \text{ is } b_1 \ldots b_n \text{ where } n=\text{length}(B) \geq I, \\
& \text{and } b_i = 0 \text{ or } 1 \text{ for } i=1,2,\ldots,n; \\
B & \text{if } I > 0 \text{ and } \text{length}(B) < I.
\end{cases} \]

3.5.4 Let I be an integer (≠ 0), and let C be a <string> of the form \[ c_1 c_2 \ldots c_n \]
where \( n=\text{length}(C) \), \( c_i \) is a <basic symbol> for \( i=1,2,\ldots,n \).

Then \( p(\text{string} \{ \text{exact I} \} ,C) \) is
\[ \begin{cases} 
c_1 \ldots c_n & \text{if } n \geq I, \\
c_1 \ldots c_n \ldots 0 & \text{if } n < I.
\end{cases} \]

\( I-n \)

\( p(\text{string} \{ \text{varying I} \} ,C) \) is
\[ \begin{cases} 
c_1 \ldots c_n & \text{if } n \geq I, \\
C & \text{if } n < I.
\end{cases} \]

3.5.5 Let \( W \) be a value of reference type.

Then \( p(\text{reference},W) \) is \( W \).

3.5.6 Let I be an integer, \( I' \) be an integer ≠ I, and let T be a type. And let \( W \) be a set of the form
\[ \{ <v,Q_v>, <v+1,Q_{v+1}>, \ldots, <u,Q_u> \} , \]
where \( Q_i \) is a quantity of type T.

Then \( p(\text{array} \{ I:I' \} T,W) \) is
\[ \{ <I,Q_I',> , <I+1,Q_{I+1}'> , \ldots, <I',Q_{I'}'> \} \]
where if \( v \leq j \leq u \) then \( Q_j' \) is \( Q_j \), else \( Q_j' \) is unspecified, (but \( t(Q_j') \) is T) for \( j=I,I+1,\ldots,I' \).

3.5.7 If \( t(M) \) is structure style and \( t(W) \) is coincide with \( t(M) \), then \( p(H,W) \) is \( W \).

3.5.8 If \( t(M) \) is procedure style, and \( t(W) \) is coincide with \( t(M) \), then \( p(M,W) \) is \( W \).

3.6 Abilities

During the elaboration, each <variable> is in a state of ability which is either able or inable, and may be changed to its alternative. We shall denote such ability of a <variable> \( V \) by \( a(V) \). If a <variable> \( V \) is able, \( V \) is associated with some quantity, which we shall denote by \( q(V) \).

Pragmatics Briefly speaking, a <variable> is made inable at the entrance of the <block> , and is made able at the end of the elaboration of its <declaration> . End of pragmatics
4. Operators

4.1 Core Language

To describe the action of operations, the elaboration (e), the generation (g), the refinement (r), etc. we shall use the "core language". This language consists of usual mathematical notations, simple English sentences and ALGOL-like structures. Moreover we shall use several conventions as follows:

Let K be a label in the core language.

1) \( \rightarrow K \) : "go to K".
2) \( \rightarrow \) next : "go to next statement".

Let \( f \) be an operator, and let \( A \) be an operand.

3) \( f(A) \) : "operate \( f \) on \( A \)".
4) \( f(A) \Rightarrow A' \) : "let \( A' \) stand for the result of the operation \( f \) on \( A \)".
5) \( \Rightarrow A' \) : "the operation is now completed with the result \( A' \)".
6) \( A \leftarrow A' \) : "let \( A \) stand for \( A' \)".
7) \( \text{Let } A = A' \) : "let \( A \) be of the form \( A' \)".

When \( f \) is the elaboration "e", and \( E \) is an <expression>,
8) \( e(E) \) if \( Q \rightarrow K, L \rightarrow K' \) : "e(E); let the result be \( A' \); if \( A' \notin Q \), then let \( Q \) stand for \( A' \) and go to \( K \), else if \( A' \in Q \), then let \( L \) stand for \( A' \) and go to \( K' \)".

4.2 Generations

4.2.1 \( g(Q) \) is

\begin{verbatim}
core
let Q stand for a new abstract element \( \notin Q \);
Q \leftarrow Q \cup \{Q\} ;
=> Q
end of core
\end{verbatim}

4.2.2 \( g(V) \) is

\begin{verbatim}
core
let V stand for a new identifier \( \notin V \cup L \);
V \leftarrow V \cup \{V\} ;
=> V
end of core
\end{verbatim}

4.2.3 \( g(L) \) is

\begin{verbatim}
core
let L stand for a new identifier \( \notin L \cup V \);
L \leftarrow L \cup \{L\} ;
=> L
end of core
\end{verbatim}
4.3 Substitution

Let $E$ be an expression; $n$ be an integer ($\geq 0$); $V_i$ be a variable or a label, different from each other, for $i=1,2,\ldots,n$; and $E_i$ be an expression or a label, for $i=1,2,\ldots,n$. Then

$$E(V_1 \ldots V_n)$$

$$E_1 \ldots E_n$$

denotes the expression obtained from $E$ after substitution of $E_1, \ldots, E_n$ for $V_1, \ldots, V_n$ respectively and simultaneously, except those $V_i$, if any, contained in a string. (The exact and too long definition, carried out recursively along the course of construction, of $E$ is omitted.)

4.4 Refinement

When $E$ is an expression the result of its refinement is roughly speaking, the expression obtained from $E$ after substitution of the result of

$$g(V)$$

for each variable or label which is local to $E$ or to a block or procedure notation in $E$. Exactly, it is defined by recursion on the construction of $E$:

4.4.1 Let $E$ be an expression of the form

$$A_0 E_1 A_1 E_2 A_2 \ldots A_{n-1} E_n A_n$$

where $n$ is an integer ($\geq 0$); $E_1, E_2, \ldots, E_n$ are all direct constituents of $E$; and $A_i$ is a sequence of basic symbol's or empty for $i=0,1,2,\ldots,n$.

Then $r_0(E)$ is

**core**

$$r(E_1) \Rightarrow F_1;$$

$$r(E_2) \Rightarrow F_2;$$

$$\ldots$$

$$r(E_n) \Rightarrow F_n;$$

let $F$ stand for (the expression )

$$A_0 F A_1 F A_2 \ldots A_{n-1} F A_n;$$

$$\Rightarrow F$$

**end of core**
4.4.2 Let E be a <block>, let $V_1, \ldots, V_n$ be all its proper <variable> 's, and let $L_1, \ldots, L_m$ be all its proper <label> 's, where $n, m$ are integers ($\geq 0$).

$r(E)$ is

### core

$$r_0(E) \Rightarrow E';$$

$$g(V) \Rightarrow V_1';$$

$$\ldots$$

$$g(V) \Rightarrow V_n';$$

$$g(L) \Rightarrow L_1';$$

$$\ldots$$

$$g(L) \Rightarrow L_m';$$

let $F$ stand for (the <expression>)

$$E'(V_1' \ldots V_n' L_1' \ldots L_m');$$

$$\Rightarrow F$$

end of core

4.4.3 Let E be a <procedure notation> of the form

"procedure ($T_1', \ldots, T_n$)$T J$"

where $n$ is an integer ($\geq 0$); $T_i$ is a <typifier> for $i=1, 2, \ldots, n$; $T$ is a <typifier>; $J$ is a <procedure donor>.

1) If $J$ is empty, then $r(E)$ is

### core

$$r_0(E) \Rightarrow F;$$

$$\Rightarrow F$$

end of core

2) If $J$ is of the form

"by $((V_1', \ldots, V_n')F)$" 

where $V_i$ is a <variable> different from each other, for $i=1, 2, \ldots, n$; $F$ is an <expression> .

Then $r(E)$ is

### core

$$r(F) \Rightarrow F';$$
Let $E$ be an $\langle$expression$\rangle$ other than $\langle$block$\rangle$ or $\langle$procedure notation$\rangle$. Then $r(E)$ is

$\begin{align*}
&\text{core} \\
r_0(E) \Rightarrow F; \\
&\Rightarrow F
\end{align*}$

end of core

4.5 Elaboration

Let $E$ be an $\langle$expression$\rangle$. The elaboration of $E$ is defined recursively as follows:

$\begin{align*}
&e(E) \text{ is} \\
&\text{core} \\
&\text{if } E = \langle\text{variable}\rangle, \text{ then } \Rightarrow K_{\text{variable}}, \text{ else } \Rightarrow \text{next}; \\
&\text{if } E = \langle\text{go to statement}\rangle, \text{ then } \Rightarrow K_{\text{goto statement}}, \text{ else } \Rightarrow \text{next}; \\
&\text{if } E = \langle\text{dummy statement}\rangle, \text{ then } \Rightarrow K_{\text{dummy statement}}, \text{ else } \Rightarrow \text{next}; \\
&\text{if } E = \langle\text{code call}\rangle, \text{ then } \Rightarrow K_{\text{code call}}, \text{ else } \Rightarrow \text{next}; \\
&\text{if } E = \langle\text{closed expression}\rangle, \text{ then } \Rightarrow K_{\text{closed expression}}, \text{ else } \Rightarrow \text{next}; \\
&\text{if } E = \langle\text{block}\rangle, \text{ then } \Rightarrow K_{\text{block}}, \text{ else } \Rightarrow \text{next}; \\
&\text{if } E = \langle\text{array element}\rangle, \text{ then } \Rightarrow K_{\text{array element}}, \text{ else } \Rightarrow \text{next}; \\
&\text{if } E = \langle\text{structure element}\rangle, \text{ then } \Rightarrow K_{\text{structure element}}, \text{ else } \Rightarrow \text{next}; \\
&\text{if } E = \langle\text{procedure call}\rangle, \text{ then } \Rightarrow K_{\text{procedure call}}, \text{ else } \Rightarrow \text{next}; \\
&\text{if } E = \langle\text{effect notation}\rangle, \text{ then } \Rightarrow K_{\text{effect notation}}, \text{ else } \Rightarrow \text{next}; \\
&\text{if } E = \langle\text{real notation}\rangle, \text{ then } \Rightarrow K_{\text{real notation}}, \text{ else } \Rightarrow \text{next}; \\
&\text{if } E = \langle\text{bits notation}\rangle, \text{ then } \Rightarrow K_{\text{bits notation}}, \text{ else } \Rightarrow \text{next}; \\
&\text{if } E = \langle\text{string notation}\rangle, \text{ then } \Rightarrow K_{\text{string notation}}, \text{ else } \Rightarrow \text{next}; \\
&\text{if } E = \langle\text{reference notation}\rangle, \text{ then } \Rightarrow K_{\text{reference notation}}, \text{ else } \Rightarrow \text{next}; \\
&\text{if } E = \langle\text{array notation}\rangle, \text{ then } \Rightarrow K_{\text{array notation}}, \text{ else } \Rightarrow \text{next}; \\
&\text{if } E = \langle\text{structure notation}\rangle, \text{ then } \Rightarrow K_{\text{structure notation}}, \text{ else } \Rightarrow \text{next}; \\
&\text{if } E = \langle\text{procedure notation}\rangle, \text{ then } \Rightarrow K_{\text{procedure notation}}, (\text{else } \Rightarrow \text{LO}); \\
\end{align*}$

$K_{\text{variable}}$:

$\begin{align*}
&\text{if } a(E) = \text{able} \text{ then } \Rightarrow \text{next}, \text{ else } \Rightarrow \text{LO}; \\
&\text{let } Q \text{ stand for } q(E); \\
&\Rightarrow Q;
\end{align*}$
Kgotostatement:

let E be of the form

"go to L"

where L is a <label> ;

⇒ L;

Kdummystatement:

⇒ QO;

Kcodecall:

let E be of the form

"code (S_1 E_1, ...., S_n E_n) T by (A)"

where n is an integer (≥ 0),

S_i is a <selector> for i=1,2,....,n,

E_i is an <expression> for i=1,2,....,n,

T is a <typifier> ,

A is a <code body> ;

let F stand for the expression

"structure (S_1 E_1, ...., S_n E_n)";

e(F) if Q ⇒ next, L ⇒ L;

e(A(Q)) if Q' ⇒ Q', L ⇒ L;

Kclosedexpression:

let E be of the form "(F)" where F is an <expression> ;

e(F) if Q ⇒ Q, L ⇒ L;

Kblock:

let E be of the form

"begin let V_1 be E_1;

....

let V_n be E_n; L_1^l:....:L_n^l : F_1;

....

L_1^k:....:L_k^k : F_k end"

where n is an integer (≥ 0),

k is an integer (≥ 1),

i_u is an integer (≥ 0) for u=1,2,....,k,

V_j is a <variable> for j=1,2,....,n,

E_j is an <expression> for j=1,2,....,n,

F_u is an <expression> for u=1,2,....,k,

L_u^v is a <label> for u=1,2,....,k,

v=1,2,....,i_u;
\( a(V_j) \leftarrow \text{inable for } j=1,2,\ldots,n; \ (e(E_j) \text{ if } Q \rightarrow \text{next, } L \Rightarrow L; \)

\( a(V_j) \leftarrow \text{able; } q(V_j) \leftarrow Q; \) for \( j=1,2,\ldots,n \)

\( K_{111}: \ e(F_1) \text{ if } Q \rightarrow \text{next, } L \Rightarrow K_{121}; \)

\( K_{112}: \ e(F_2) \text{ if } Q \rightarrow \text{next, } L \Rightarrow K_{121}; \)

\( \ldots \)

\( K_{11k}: \ e(F_k) \text{ if } Q \rightarrow \text{next, } L \Rightarrow K_{121}; \)

\( K_{121}: \Rightarrow Q; \)

\( \text{if } i_1=0 \text{ then } \rightarrow K_{122}, \text{ else } \rightarrow \text{next; } \)

\( \text{(if } L=L^v \text{ then } K_{111}, \text{ else } \rightarrow \text{next; ) for } v=1,2,\ldots,i_1 \)

\( K_{122}: \text{if } i_2=0 \text{ then } \rightarrow K_{123}, \text{ else } \rightarrow \text{next; } \)

\( \text{(if } L=L^v \text{ then } K_{112}, \text{ else } \rightarrow \text{next; ) for } v=1,2,\ldots,i_2 \)

\( K_{123}: \text{\ldots} \)

\( K_{12k}: \text{if } i_k=0 \text{ then } \rightarrow K_{13}, \text{ else } \rightarrow \text{next; } \)

\( \text{(if } L=L^v \text{ then } K_{11k}, \text{ else } \rightarrow \text{next; ) for } v=1,2,\ldots,i_k \)

\( K_{13}: \Rightarrow L; \)

**Karrayelement:**

let E be of the form \( \"F \{ E' \}\" \)

where F is an expression , E' is an expression; \( t(F) \text{ is a type of array style then }\rightarrow \text{next, else } \Rightarrow L_0; \)

if \( t(E') \text{ is real then }\rightarrow \text{next, else } \Rightarrow L_0; \)

e(F) if \( Q \rightarrow \text{next, } L \Rightarrow L; \text{ if } w(F) \neq \emptyset \text{ then } \rightarrow \text{next, else } \rightarrow K_{21}; \)

let v stand for the integer 1; let u stand for the integer 0;

\( \rightarrow K_{22}; \)

\( K_{21}: \text{let } w(F)=\{<v,Q_v>,<v+1,Q_{v+1}>,\ldots,<u,Q_u>\} \)

where v is an integer, u is an integer \((\geq v), \)

\( Q_j \text{ is a quantity for } j=v,v+1,\ldots,u; \)

\( K_{22}: \text{e}(E') \text{ if } Q' \rightarrow \text{next, } L \Rightarrow L; \)

let i stand for the integer round \((w(Q'))\);

\( \text{if } i \geq v \text{ then } \rightarrow \text{next, else } \Rightarrow L_0; \text{ if } i \leq u \text{ then } \Rightarrow Q_i, \text{ else } \Rightarrow L_0; \)

**Kstructureelement:**

let E be of the form \( \"F \{ S \}\" \)

where F is an expression , S is a selector; \( t(F) \text{ is a type of structure style then } \rightarrow \text{next, else } \Rightarrow L_0; \)

e(F) if \( Q \rightarrow \text{next, } L \Rightarrow L; \)

let \( w(F)=\{<S_1,Q_1>,\ldots,<S_n,Q_n>\} \)

where n is an integer \((\geq 0), \)

\( S_j \text{ is a selector for } j=1,2,\ldots,n, \)

\( Q_j \text{ is a quantity for } j=1,2,\ldots,n; \)
if $S = S_i$ for some $i (1 \leq i \leq n)$ then $\Rightarrow Q_i$, else $\Rightarrow L_0$;

Kprocedurecall:

let $E$ be of the form $"F(E_1, \ldots, E_n)"$

where $n$ is an integer $(\geq 0)$,

$F$ is an expression, $E_j$ is an expression for $j = 1, 2, \ldots, n$;

if $t(F)$ is a type of procedure style, then $\Rightarrow$ next, else $\Rightarrow L_0$;

$e(F)$ if $Q \Rightarrow$ next, $L \Rightarrow L$;

let $t(Q)$ be of the form $"\text{procedure } (T_1, \ldots, T_m)T"$

where $m$ is an integer $(\geq 0)$, $T_i$ is a type for $i = 1, 2, \ldots, m$,

$T$ is a type;

let $w(Q)$ be of the form $"(V_1, \ldots, V_m)F'"$

where $V_i$ is a variable for $i = 1, 2, \ldots, m$,

$F'$ is an expression;

let $E'$ stand for the block of the form

"begin let $V_1$ be $T_1$;

\ldots

let $V_m$ be $T_m$;

$F'$ end";

if $E'$ is legal then $\Rightarrow$ next, else $\Rightarrow L_0$;

if $n = m$ then $\Rightarrow$ next, else $\Rightarrow L_0$;

(if $t(E_i)$ is $T_i$ then $\Rightarrow$ next, else $\Rightarrow L_0$) for $i = 1, 2, \ldots, m$

if $t(E')$ is $T$ then $\Rightarrow$ next, else $\Rightarrow L_0$;

(let $E_i$ stand for the expression of the form

"$(E_i)$"; for $i = 1, 2, \ldots, n$

$E' = E_i E'$

let $F'' = F'(V_1, \ldots, V_m)$; $r(F'') \Rightarrow F''$; $e(F'')$ if $Q' \Rightarrow Q'$,

$L \Rightarrow L$;

Keffectnotation:

let $M$ stand for the mode effect;

let $W$ stand for the value done;

$g(Q) \Rightarrow Q$; $n(Q) \leftarrow M$; $w(Q) \leftarrow W$; $\Rightarrow Q$;

Krealnotation:

let $E$ be of the form $"\text{real } H J"$

where $H$ is a real modifier, $J$ is a real donor;

if $H$ is empty then $\Rightarrow K31$, else $\Rightarrow$ next;

if $H$ is of the form $"[E_1; E_2; E_3]\"$

where $E_i$ is an expression or empty for $i = 1, 2, 3$,

then $\Rightarrow K32$, else $\Rightarrow$ next;
if \( H \) is of the form " [\textbf{precision} \( F \)] "
where \( F \) is an \textbf{expression} or empty, then \( \rightarrow K_{33}, \) (else \( \rightarrow L_0 \));

\( K_{31}: \) if \( J \) is empty then \( \rightarrow K_{331}, \) else \( \rightarrow \) next;
if \( J \) is an \textbf{integer donor} then \( \rightarrow K_{311}, \) else \( \rightarrow \) next;
if \( J \) is a \textbf{fraction donor} then \( \rightarrow K_{312}, \) (else \( \rightarrow L_0 \));

\( K_{311}: \) let \( R_3 \) stand for the integer \( w(J) \);
let \( R_2 \) stand for the integer \( l \); let \( R_1 \) stand for the integer \(-R_3\);
\( \rightarrow K_{326}; \)

\( K_{312}: \) let \( R \) stand for the real number \( w(J) \);
\( \rightarrow K_{332}; \)

\( K_{32}: \) if \( E_1 \) is empty then \( \rightarrow K_{321}, \) else \( \rightarrow \) next;
if \( t(E_1) \) is \textbf{real} then \( \rightarrow \) next, else \( \rightarrow L_0; \)
e(\( E_1 \)) if \( Q_1 \) \( \rightarrow \) next, \( L \rightarrow L; \)
let \( R_1 \) stand for the real number \( w(Q_1) \);
\( \rightarrow K_{322}; \)

\( K_{321}: \) let \( R_1 \) stand for the real number \( R_1 \) (of implementation dependent);

\( K_{322}: \) if \( E_2 \) is empty then \( \rightarrow K_{323}, \) else \( \rightarrow \) next;
if \( t(E_2) \) is \textbf{real} then \( \rightarrow \) next, else \( \rightarrow L_0; \)
e(\( E_2 \)) if \( Q_2 \) \( \rightarrow \) next, \( L \rightarrow L; \)
let \( R_2 \) stand for the real number \( w(Q_2) \);
\( \rightarrow K_{324}; \)

\( K_{323}: \) let \( R_2 \) stand for the integer \( l \);

\( K_{324}: \) if \( E_3 \) is empty then \( \rightarrow K_{325}, \) else \( \rightarrow \) next;
if \( t(E_3) \) is \textbf{real} then \( \rightarrow \) next, else \( \rightarrow L_0; \)
e(\( E_3 \)) if \( Q_3 \) \( \rightarrow \) next, \( L \rightarrow L; \)
let \( R_3 \) stand for the real number \( w(Q_3) \);
\( \rightarrow K_{326}; \)

\( K_{325}: \) let \( R_3 \) stand for the real number \( R_2 \) (of implementation dependent);

\( K_{326}: \) let \( M \) stand for the mode \textbf{real} \([R_1;R_2;R_3]\);
\( \rightarrow K_{34}; \)

\( K_{33}: \) if \( F \) is empty then \( \rightarrow K_{331}, \) else \( \rightarrow \) next;
if \( t(F) \) is \textbf{real} then \( \rightarrow \) next, else \( \rightarrow L_0; \)
e(\( F \)) if \( Q \) \( \rightarrow \) next, \( L \rightarrow L; \)
let \( R \) stand for the real number \( w(Q) \);
\( \rightarrow K_{332}; \)

\( K_{331}: \) let \( R \) stand for the real number \( R_3 \) (of implementation dependent);
K32: let $M$ stand for the mode real [precision $R$];

K33: let $M'$ stand for the mode $d(M)$;

if $J$ is empty then $\rightarrow K34$, else $\rightarrow$ next;
let $W$ stand for the real number $w(J)$;
$\rightarrow K35$;

K34: let $W$ stand for an arbitrary real number;

K35: if $W \neq \emptyset$ then $\rightarrow$ next, else $\rightarrow$ LO;
let $X$ stand for the real number $p(M', W)$;

g($W$) $\rightarrow Q'$;
w($Q'$) $\leftarrow M'$;
w($Q'$) $\leftarrow X$;
$\Rightarrow Q'$;

Kbitsnotation:

let $E$ be of the form "bits H J" where $H$ is a bits modifier, $J$ is a bits donor;
if $H$ is empty then $\rightarrow K41$, else $\rightarrow$ next;
if $H$ is of the form "[exact F]" where $F$ is an expression, then $\rightarrow K411$, else $\rightarrow$ next;
if $H$ is of the form "$\leftarrow J$" then $\rightarrow K412$, else $\rightarrow$ next;
if $H$ is of the form "$J$" then $\rightarrow K415$, (else $\rightarrow$ LO);

K41: if $J$ is empty then $\rightarrow K412$, else $\rightarrow$ next;
let $I$ stand for the integer length($w(J)$); $\rightarrow K413$;

K411: if $t(F)$ is real then $\rightarrow$ next, else $\rightarrow$ LO;
e($F$) if $Q$ $\rightarrow$ next, $L$ $\Rightarrow$ $L$;
let $I$ stand for the integer round($w(Q)$); $\rightarrow K413$;

K412: let $I$ stand for the integer $I1$ (of implementation dependent);

K413: let $M$ stand for the mode bits [exact $I$]; $\rightarrow K42$;

K414: if $t(F)$ is real then $\rightarrow$ next, else $\rightarrow$ LO;
e($F$) if $Q$ $\rightarrow$ next, $L$ $\Rightarrow$ $L$;
let $I$ stand for the integer round($w(Q)$); $\rightarrow K416$;

K415: let $I$ stand for the integer $I2$ (of implementation dependent);

K416: let $M$ stand for the mode bits [varying $I$];

K42: let $M'$ stand for the mode $d(M)$;
if $J$ is empty then $\rightarrow K421$, else $\rightarrow$ next;
let $W$ stand for the bit-string $w(J)$; $\rightarrow K43$;

K421: let $W$ stand for an arbitrary bits;
K43: if $W \neq \emptyset$, then → next, else ⇒ LO;
let X stand for the bits $p(H', W)$;
$g(Q) \Rightarrow Q'$;
$m(Q') \leftarrow M'$;
$w(Q') \leftarrow X$;
⇒ $Q'$;

Kstringnotation:
let E be of the form "string H J"
where H is a <string modifier>, J is a <string donor>;
if H is empty then → K51, else → next;
if H is of the form "[exact F]"
where F is an <expression>, then → K511, else → next;
if H is of the form "[varying F]"
then → K512, else → next;

K51: if J is empty then → K512, else → next;
let I stand for the integer length(w(J)); → K513;

K511: if t(F) is real then → next, else ⇒ LO;
e(F) if Q → next, L ⇒ L;
let I stand for the integer round(w(Q)); → K513;

K512: let I stand for the integer I3 (of implementation dependent);

K513: let M stand for the mode string [exact I]; → K52;

K514: if t(F) is real then → next, else ⇒ LO;
e(F) if Q → next, L ⇒ L;
let I stand for the integer round(w(Q)); → K516;

K515: let I stand for the integer I4 (of implementation dependent);

K516: let M stand for the mode string [varying I];

K52: let M' stand for the mode d(M);
if J is empty then → K521, else → next;
let W stand for the <string> w(J) (=J); → K53;

K521: let W stand for an arbitrary <string>;

K53: if $W \neq \emptyset$, then → next, else ⇒ LO;
let X stand for the <string> $p(H', W)$;
$g(Q) \Rightarrow Q'$;
$m(Q') \leftarrow M'$;
$w(Q') \leftarrow X$;
⇒ $Q'$;
Kreferencenotation:
if E is of the form "reference"
then → K61, else → next;
if E is of the form "reference nil"
then → next, (else => LO);
let W stand for the empty set ∅; → K62;

K61: let Q stand for an arbitrary quantity in Q;
let W stand for the set {Q};

K62: g(Q) => Q';
m(Q') => reference;
w(Q') => W;
=> Q';

Karraynotation:
if E is of the form "array H F"
where H is an <array modifier> ,
F is an <expression> ,
then → K71, else → next;
if E is of the form "array (E_1,···,E_n)"
where n is an integer (≥ 1),
E_i is an <expression> for i=1,2,···,n
then → K73, (else => LO);

K71: if H is empty then → K711, else → next;
if H is of the form "[E_1;E_2]"
where E_i is an <expression> for i=1,2,
then → next, (else => LO);
if t(E_1) is real then → next, else => LO;
e(E_1) if Q_1 → next, L => L;
let I_1 stand for the integer round(w(Q_1));
if t(E_2) is real then → next, else => LO;
e(E_2) if Q_2 → next, L => L;
let I_2 stand for the integer round(w(Q_2)); → K712;

K711: let I_1 stand for the integer 1;
let I_2 stand for the integer 0;

K712: let T stand for the type t(F);
let M stand for the mode array [I_1;I_2] T;
let M' stand for the mode d(M);
let M' = array [v;u] T
where v is an integer, u is an integer;
if v > u then → K72, else → next;
(e(P) if Qj → next, L ⇒ L;) for j = v, v + 1, ..., u
let X stand for the set
\{ <v, Q_v'>, <v+1, Q_{v+1}'>, ..., <u, Q_u'> \}
→ K74;
K72: let X stand for the empty set \( \emptyset \);
→ K74;
K73: let T stand for the type t(E_1);
let W stand for the mode array [1:n]T;
let M' stand for the mode d(M);
(if t(E_j) is T then → next, else =, L0;
e(E_j) if Q_j → next, L ⇒ L;) for j = 1, 2, ..., n
let W stand for the set
\{ <1, Q_1'>, <2, Q_2'>, ..., <n, Q_n'> \};
let X stand for the set p(M', W);
K74: g(Q_0) =⇒ Q'';
m(Q'') = M'; w(Q'') = X; =⇒ Q'';
Kstructurenotation:
let E be of the form
"structure (S_1, E_1, ..., S_n, E_n)"
where n is an integer ( \( \geq 0 \)),
S_i is a <selector> for i = 1, 2, ..., n,
E_i is an <expression> for i = 1, 2, ..., n;
if n = 0 then → K81, else → next;
(e(E_j) if Q_j → next, L ⇒ L;) for j = 1, 2, ..., n
let W stand for the set
\{ <S_1, Q_1'>, ..., <S_n, Q_n'> \};
(let \( T_i \) stand for the type t(E_i);) for i = 1, 2, ..., n
let M stand for the mode
structure (S_1, T_1, ..., S_n, T_n);
→ K82;
K81: let W stand for the empty set \( \emptyset \)
let M stand for the mode
structure ();
Kprocedurenotation:

let \( E \) be of the form

\[
\text{procedure } (T_1, \ldots, T_n)T \ J
\]

where \( n \) is an integer \((\geq 0)\),

\( T_i \) is a \(<\text{typifier}>\) for \( i = 1, 2, \ldots, n \),

\( T \) is a \(<\text{typifier}>\),

\( J \) is a \(<\text{procedure donor}>\);

if \( J \) is empty, then \( \rightarrow \text{K91} \), else \( \rightarrow \text{next} \);

let \( J \) be of the form

\[
\text{by}(V_1, \ldots, V_n)E'
\]

where \( V_i \) is a \(<\text{variable}>\) for \( i = 1, 2, \ldots, n \),

\( E' \) is an \(<\text{expression}>\);

let \( W \) stand for the figure

\[
(V_1, \ldots, V_n)E';
\]

\( \rightarrow \text{K92} \);

K91: let \( E' \) be an arbitrary legal \(<\text{expression}>\) without \(<\text{mark}>\) and of the type \( T \) and of the form

\[
\text{begin let } V_1 \text{ be } T_1;
\]

\[
\ldots
\]

\[
\text{let } V_n \text{ be } T_n;
\]

\[
E' \text{' end'};
\]

let \( W \) stand for the figure

\[
(V_1, \ldots, V_n)E';
\]

K92: let \( W \) stand for the mode

\[
\text{procedure } (T_1, \ldots, T_n)T;
\]

\[
g(\xi) \Rightarrow \eta';
\]

\[
m(\eta') \leftarrow M;
\]

w(\eta') \leftarrow W;

\Rightarrow \eta
\]

end of core
5. Dynamic Behavior of Programs

5.1 Creation

In the beginning, we must create two sets $\mathcal{Q}_0$ and $\mathcal{V}_0$.

5.1.1 All <expression> 's in standard <declaration> 's are rewritten in the form of normal (as defined in the next paragraph).

5.1.2 Let $\mathcal{V}_0$ stand for the set of the all <identifier> 's declared by some standard <declaration>, and let $\alpha(V)$ be able, and $\mathcal{Q}(V)$ be an abstract element, different from each other, for each other, for each <variable> $V$ in $\mathcal{V}_0$.

5.1.3 Let $\mathcal{Q}_0$ stand for the set

$$\{ \mathcal{Q}(V) \mid V \in \mathcal{V}_0 \}$$

and let $\nu(Q)$ be the value of $Q$ for each $Q \in \mathcal{Q}_0$.

(If standard <declaration> of a <variable> $V \in \mathcal{V}_0$ is of the form

"let $V$ be procedure $(T_1, \ldots, T_n)T$ by $(V_1, \ldots, V_n)E$" and $\nu(V) = Q$, then

$\nu(Q)$ is $(V_1, \ldots, V_n)E_1$)

5.1.4 Let $Q_0$ stand for an abstract element $\mathcal{Q}_0$, and let $L_0$ stand for a <label>.

5.2 Normalization

Let $E_1$ stand for a legal program, $\mathcal{V}_1$ stand for the set of the all <identifier> 's contained in $E_1$, $\mathcal{L}_1$ stand for the set of the all <label> 's contained in $E_1$.

1) Let $\mathcal{V}$ stand for $\mathcal{V}_0 \cup \mathcal{V}_1$, and let $\alpha(V)$ be unable for $V \in \mathcal{V} - \mathcal{V}_0$.

2) Let $\mathcal{L}$ stand for $\mathcal{L}_1 \cup \{L_0\}$.

3) Let $\mathcal{Q}$ stand for $\mathcal{Q}_0 \cup \{Q_0\}$, and let $\nu(Q_0)$ be done.

4) $\nu(E_1) \Rightarrow E_2$.

5) Let $D$ be a <form declaration> in $E_2$ of the form

"let $G$ represent $F$"

with a <form> $G$ and an <expression> $F$.

$$\mathcal{Q}(V) \Rightarrow V;$$

Replace in $E_2$ $D$ with

"let $V$ be $F$;
let $G$ represent $V$;".
6) Let $E'$ be a form call in $E_2$, of the form

$$P_0 E_1 P_1 E_2 P_2 \ldots P_{n-1} E_n P_n$$

where $E_1', \ldots, E_n'$ be expressions and $P_i$ be empty or a sequence of mark's for $i = 0, 1, \ldots, n$. If the operator form $(P_0 t(E_1')) P_1 t(E_2') P_2 \ldots P_{n-1} t(E_n') P_n)$ is declared by a declaration of the form

"let $G$ represent $F$",

then, replace $E'$ in $E_2$ with

$$(F)(E_1', \ldots, E_n')$$

7) When $T$ is a typifier in $E_2$, replace $T$ in $E_2$ with $t(n)$.

8) Eliminate all form declaration and mark declaration in $E_2$, and let $E$ stand for the result.

An expression of the form as $E$ is called normal.

5.3 Elaboration of a Normal Program

$e(E)$;

if the result is a quantity $Q$, then the elaboration of $E$ is thus completed, but if the result is a label $L$, then the elaboration of $E$ is undefined.

5. Standard declarations

This section describes the ordered set SD of standard declarations supporting the use of the language. The description takes the form of orderly enumeration of a family of disjoint subsets in SD. The description of each member of this family is headed by a reference numbering of the form (SDi.j.k). The ordering in SD is such that one element under (SDi1.j1.k1) precedes another element under (SDi2.j2.k2) if $i_1.j_1.k_1$ precedes $i_2.j_2.k_2$ in the lexical order.

Each member description begins either with a '{' (not a '{') or with an 'f'.

(a) If a member description begins with a '{', then the matching '}' terminates that description, and the member has a single element, which is the enclosed text.

(b) If a member description begins with an 'f', then the part enclosed by the first following '{' and the matching '}', with which the member description is terminated, gives an element of the member when asterisked symbols ($T^*$, for example) in that part are consistently replaced by certain objects as prescribed after the leading 'f' in terms of common mathematical notations ($T^* \rightarrow T$ for example).

† In the present document, instead of the whole family covering SD, only some illustrating subfamily will be presented. Other members, including those for input/out operations, will be found elsewhere.
6.1. Basic operations

6.1.1. Copying and enproceduring operations

(SD1.1.1) \{ \text{let copy operate before all left after copy right} \}
(SD1.1.2) \{ \text{let enproc operate before all left after all right} \}
(SD1.1.3) \int T^x \in T \{ \text{let copy() represent procedure}(T^x)T^x \text{ by}
\text{(original) code (original, d:T^x \text{ by}
\text{(core(Q)} \text{ let } w(Q)= \{ \langle c:,Q_1 \rangle, \langle d:,Q_2 \rangle \};
\quad m(Q_2) \leftarrow m(Q_1); \quad w(Q_2) \leftarrow w(Q_1); \Rightarrow Q_2 \text{ end of core})} \}
(SD1.1.4) \int T^x \in T \{ \text{let enproc() represent procedure}(T^x)T^x \text{ by}
\text{(body) procedure()}T^x \text{ by}
((() body)) \}

6.1.2. Simple assignment operations

(SD1.2.1) \{ \text{let = operate before = left after all right} \}
(SD1.2.2) \int T^x \in \{ \text{real, bits, reference}\} \cup T \text{ procedure} \quad \Delta^x \in \{ =, := \}
\{ \text{let () } \Delta^x() \text{ represent procedure}(T^x, T^x) \text{ effect by}
\text{(destination, source) code (destination, s: source) effect by}
\text{(core(Q)} \text{ let } w(Q)= \{ \langle d:,Q_1 \rangle, \langle s:,Q_2 \rangle \};
\quad w(Q_1) \leftarrow p(m(Q_1), w(Q_2)); \Rightarrow Q_0 \text{ end of core}) \}

6.1.3. Simple comparison operations

(SD1.3.1) \{ \text{let } =, \neq \text{ operate before =, } = \text{ left after all right} \}
(SD1.3.2) \{ \text{let } <, \leq, \geq, > \text{ operate before } =, \neq \text{ left after =, := right} \}
(SD1.3.3) \int T^x \in T, \Delta^x \in \{ =, \neq \}; \ T^x=\text{real}, \Delta^x \in \{ <, \leq, \geq, > \}
\{ \text{let () } \Delta^x() \text{ represent procedure}(T^x, T^x) \text{ bits by}
\text{((left,right) code (l: copy left, r: right, t:1) bits by}
\text{(core(Q)} \text{ let } w(Q)= \{ \langle l:,Q_1 \rangle, \langle r:,Q_2 \rangle, \langle t:,Q_3 \rangle \};
\quad \text{if } w(Q_1) \Delta^x w(Q_2) \text{ then } \Rightarrow Q_3, \text{ else } \Rightarrow \text{ next;}
\quad w(Q_3) \leftarrow Q; \Rightarrow Q_3 \text{ end of core}) \}

6.1.4. Conditional operations

(SD1.4.1) \{ \text{let if operate before all right} \}
(SD1.4.2) \{ \text{let then operate} \}
(SD1.4.3) \{ \text{let else, do operate after all right} \}
(SD1.4.4) \int T^x \in T \{ \text{let if() then() else() represent}
procedure(bit, $T^2$, $T^2$) $T^2$ by

((condition, then, else)

code (c:condition, t:enproc then, e:enproc else)

(enproc $T^2$) by

(core(q) let w(q)= $\{<c, Q_1>, <t, Q_2>, <e, Q_3>\}$
if the bit string $w(Q_1)$ contains at least one $1$
then $\Rightarrow Q_2$, else $\Rightarrow Q_3$ end of core)

(SD.5.4) $\{ let if()do() represent procedure(bit, effect)effect
by((condition, statement) if condition then statement
else dummy) $\}$

6.1.5. Basic arithmetic operations

(SD.5.1) $\{ let +, - operate before $\neq$, $<$, $\leq$, $\geq$, $>$ left
after $+, =, \leq, \geq$ right $\}$

(SD.5.2) $\{ let \times, / operate before $\neq$, $<$, $\leq$, $\geq$, $>$, $+, -$ left
after $\times, /, =, >, <$ right $\}$

(SD.5.3) $\Delta$ $\{ let (\Delta^*)$ represent
procedure(real, real)real by ((left, right)

code (a:copy left, b:right) real by

(core(q) let w(q)= $\{<a, Q_1>, <b, Q_2>\}$
if the arithmetic operation meant by $w(Q_1) \Delta^* w(Q_2)$
can be performed then $\Rightarrow$ next, else $\Rightarrow L0$;
let $W_3$ be the resulting value of that operation
(possibly with some implementation dependent deviation);
$w(Q_1) \leftarrow p(m(Q_1), W_3); \Rightarrow Q_1$ end of core)) $\}$

(SD.5.4) $\{ let -$() represent procedure(real)real by
((right) begin (right)b; (copy b)a; a=0; a-b end $\}$

(SD.5.5) $\{ let /() represent procedure(real)real by
((right) begin (right)b; (copy b)a; a=1; a/b end $\}$

6.1.6 Some enquiry operations

(SD.6.1) $\{ let mode, length, bd operate before all left
after right $\}$

(SD.6.2) $\{ let mode() represent procedure(real)structure
(fix:bit, min:real, step:real, max:real) by ((real)
begin (real)x; code (fix), min:copy x, step:real,
max:copy x) (mode real) by

(core(q) let w(q)= $\{<fix, Q_1>, <min, Q_2>, <step, Q_3>, <max, Q_4>\}$

let M stand for $w(Q_2)$

$w(Q_2) \leftarrow$ the minimum value in $W_M$;
w(Q_4) \leftarrow \text{the maximum value in } W_M;
if M = \text{real} [\text{precision } R] \text{ with a real number } R,
then \rightarrow \text{next, else } \rightarrow \text{Kfixed};
w(Q_3) \leftarrow p(m(Q_3), R); w(Q_1) \leftarrow Q; \rightarrow Q;
Kfixed: \text{let } M = \text{real} [R_1, R_2, R_3] \text{ with real numbers } R_1, R_2, R_3;
w(Q_3) \leftarrow p(m(Q_3), R_2); \rightarrow Q \text{ end of core) end}}

(\text{SD1.6.3}) f T^*_6 \in \text{bits, string} \{ \text{let mode() represent procedure(T^*_6) structure } (\text{exact: bits, length: real}) \} 
\text{by } ((\text{some}) \text{ code (s: some, t: (exact: 1, length: 1)) (mode T^*_6) by})
(\text{core}(Q) \text{ let } w(Q) = \{<s: Q_1>, <t: Q_2>\};
\text{let } w(Q_2) = \{<\text{exact: } Q_3>, <\text{length: } Q_4>\};
\text{let } m(Q_1) = T^*_6 \{Y I\}, \text{ where } I \text{ is an integer and } Y
\text{ is either exact or up to};
w(Q_4) \leftarrow p(m(Q_4), I);
\text{if } Y = \text{exact then } \rightarrow Q_2, \text{ else } \rightarrow \text{next};
w(Q_3) \leftarrow Q_2; \rightarrow Q_2 \text{ end of core})

(\text{SD1.6.4}) f T^*_6 \in T \{ \text{let mode() represent procedure(array T^*_6) structure } (\text{ldbd: real, ubbd: real}) \} 
\text{by } ((\text{array}) \text{ code (a: array, t: (ldbd: 1, ubbd: 1)) (mode array T^*_6) by})
(\text{core}(Q) \text{ let } w(Q) = \{<a: Q_1>, <t: Q_2>\};
\text{let } w(Q_2) = \{<\text{ldbd: } Q_3>, <\text{ubbd: } Q_4>\};
\text{let } m(Q_1) = \text{array } [I_1: I_2] T^*_6 \text{ with integers } I_1, I_2;
w(Q_3) \leftarrow p(m(Q_3), I_1); w(Q_4) \leftarrow p(m(Q_4), I_2);
\rightarrow Q_2 \text{ end of core})

(\text{SD1.6.5}) f T^*_6 \in \{\text{bits, string} \} \{ \text{let length() represent procedure(T^*_6) real by } ((\text{string}) \text{ code (s: string, t: 1) real by})
(\text{core}(Q) \text{ let } w(Q) = \{<s: Q_1>, <t: Q_2>\};
\text{let } I \text{ be the integral length of } w(Q_1);
w(Q_2) \leftarrow p(m(Q_2), I); \rightarrow Q_2 \text{ end of core})

(\text{SD1.6.6}) f T^*_6 \in T \{ \text{let bd() represent procedure(array T^*_6) real by } ((\text{array}) \text{ (mode array) [ldbd: ]})

(\text{SD1.6.7}) f T^*_6 \in T \{ \text{let ()bd represent procedure(array T^*_6) real by } ((\text{array}) \text{ (mode array) [ubbd: ]})


6.2. Extended operations

6.2.1. Repetitive operations

(SD2.1.1) \{ \textit{let while, until operate before left after all right} \}

(SD2.1.2) \{ \textit{let succ, step operate before left after },\lll,\, ,\, ,\, ,\, ,\, ,\, ,\, \, ,\, \, ,\, +,\, -,\, \times,\, / \textit{right} \}

(SD2.1.3) \{ T \in T \{ \textit{let ()succ() represent procedure(T*, T*)} \}
structure \textit{init: enproc T*, succ: enproc T*)} by 
((a, s) \textit{init: enproc a, succ: enproc s}) \}

(SD2.1.4) \{ T \in T \{ \textit{let ()while() represent procedure(T* succT*, bits)} \}
structure \textit{init: enproc T*, succ: enproc T*, while: enproc bits} by 
((r, t) \textit{begin (copy r) r1; (init: copy r[init:], succ: copy r[dom:], while: enproc t) end}) \}

(SD2.1.5) \{ \textit{let ()step() represent procedure(real, real)} \}
structure \textit{init: enproc real, step: enproc real} by 
((a, b) \textit{init: enproc a, step: enproc b}) \}

(SD2.1.6) \{ \textit{let step() represent procedure(real, real)} (real step real) \}
by ((b, l step b))

(SD2.1.7) \{ \textit{let ()until() represent procedure(real step real, real)} \}
structure \textit{init: enproc real, step: enproc real, until: enproc real} by 

((\textit{progression, limit}) \begin{align*}
\text{begin (copy progression) ab; (init: copy ab[init:], step: copy ab[step:], until: enproc limit) end})
\end{align*}

(SD2.1.8) \{ \textit{let ()until() represent procedure(real, real)} (step real until real) by ((a, c) a step l until c) \}

(SD2.1.9) \{ \textit{let until() represent procedure (real)(real until real)} \}
by ((c, l until c))

(SD2.1.10) \{ \textit{let from operate before all left} \}

(SD2.1.11) \{ T \in T \{ \textit{let ()do represent procedure(T*, T* succT* while bits, effect)} \}
effect by 
((\textit{cvar, domain, statement}) \begin{align*}
\text{begin (cvar) cv; (domain) dom; cv \leftarrow dom[init:];() ; next: if dom[while:]() do }
\text{begin statement; cv \leftarrow dom[succ:]; go to next end end})
\end{align*} \}
(SD2.1.12) \begin{align*}
\left\{ \begin{align*}
&\text{let } ()\text{ from }()\text{ do }()\text{ represent } \\
&\quad \text{procedure(real, until real, effect) effect by } \\
&\quad \begin{align*}
&\begin{align*}
&(\text{cvar, domain, statement}) \\
&\quad \text{begin } (\text{cvar}) \text{cv}; \ (\text{copy cv}) \text{vl}; \ (\text{copy domain}) \text{dom}; \\
&\quad \ (\text{copy dom[init:]}) \text{al}; \ (\text{copy dom[step:]}) \text{bl}; \\
&\quad \ (\text{copy dom[until:]}) \text{cl}; \\
&\quad \text{vl from al succ vl+bl while if bl>0 then vl<cl else if bl<0 then vl>cl else vl=cl} \\
&\quad \text{do } \text{begin cv:=vl; statement end}
\end{align*}
\end{align*}
\right. \\
\end{align*}
\end{align*}

6.2.2 Assign-and-hold operations

(SD2.1) \begin{align*}
\left\{ \begin{align*}
&\text{let the operate before all left after right } \\
\end{align*}
\right. \\
\end{align*}

(SD2.2.2) \begin{align*}
\int T^* \epsilon T \left\{ \begin{align*}
&\text{let the() represent } \\
&\quad \text{procedure(T*) procedure(effect) T* by } \\
&\quad \begin{align*}
&\begin{align*}
&(\text{expression}) \text{ procedure(effect) T* by } ((\text{dummy expression})) \\
&\end{align*}
\end{align*}
\right. \\
\end{align*}

(SD2.3.1) \begin{align*}
\int T^* \epsilon T \left\{ \begin{align*}
&\text{let ref, has type, as type operate before all left after right } \\
\end{align*}
\right. \\
\end{align*}

(SD2.3.2) \begin{align*}
\int T^* \epsilon T \left\{ \begin{align*}
&\text{let ref() represent procedure(T*)reference } \\
&\quad \text{by } ((\text{referent}) \text{ code (q:referent, r:reference ) reference } \\
&\quad \text{by (core(q)} \text{ let w(q)= \{<q:=Q_1>, <r:=Q_2>\} ; } \\
&\quad \text{w(Q_2) = \{Q_1 \}; \Rightarrow Q_2 \text{ end of core)}} \right. \\
\end{align*}
\right. \\
\end{align*}

(SD2.3.3) \begin{align*}
\int T^* \epsilon T \left\{ \begin{align*}
&\text{let () has type()} represent } \\
&\quad \text{procedure(reference, T*) bits by } ((\text{ref, type) code (r:ref, t:type, b:Q) bits by } \\
&\quad \text{(core(q)} \text{ let w(q)= \{<r:=Q_1>, <t:=Q_2>, <b:=Q_3>\} ; } \\
&\quad \text{if w(Q_1)= empty, then \Rightarrow Q_3, else \Rightarrow next; } \\
&\quad \text{let w(Q_1)= \{Q_4 \} ; } \\
&\quad \text{if t(Q_2)=t(Q_4) then \Rightarrow next, else \Rightarrow Q_3; } \\
&\quad \text{w(Q_3) \leftarrow 1; \Rightarrow Q_3 \text{ end of core)}} \right. \\
\end{align*}
\right. \\
\end{align*}
(SD2.3.4) $\forall T^* \in T, T^* \neq \text{reference} \{ \text{select as type() represent }
\}
\text{procedure(\text{reference, T}) T by ((ref, type) }
\text{code (r:ref, t:type) T by }
\text{(core(Q) let w(\epsilon) = } \{r:Q_1, t:Q_2\};
\text{if w(Q_1) = \text{empty, then } Q_2, \text{ else } \text{next: }}
\text{let w(Q_1) = } \{Q_4\};
\text{if t(Q_2) = t(Q_4) then } Q_4, \text{ else } Q_2 \text{ end of core)))}
$
+ This work was sponsored by the Research Institute for Mathematical Sciences, Kyoto University, Kyoto, Japan.