Modula-3 Action Semantics[1]

David A. Watt[2] Deryck F. Brown[3]

Version 0.1: 1997–05–12 (09.41)

[1]This work was supported partly by EPSRC grant GR/J86292, and partly by the Computing Science Research Festival (University of Glasgow 1995).
[2]Department of Computing Science, University of Glasgow, Glasgow G12 8QQ, Scotland.
[3]School of Computer and Mathematical Sciences, The Robert Gordon University, St Andrew Street, Aberdeen AB25 1HG, Scotland.

Contents

Chapter 1

Abstract Syntax

closed except Literals .

grammar:

1.1 Identifiers

(1) Word = ⟦ letter (letter | digit | "_")* ⟧ .

(2) Word = Keyword | Identifier (*disjoint*) .

(3) Identifier = Reserved-Identifier | Ordinary-Identifier (*disjoint*) .

(4) Keyword = "AND" | "ANY" | "ARRAY" | "AS" |
"BEGIN" | "BITS" | "BRANDED" | "BY" |
"CASE" | "CONST" | "DIV" | "DO" |
"ELSE" | "ELSIF" | "END" | "EVAL" |
"EXCEPT" | "EXCEPTION" | "EXIT" | "EXPORTS" |
"FINALLY" | "FOR" | "FROM" | "GENERIC" |
"IF" | "IMPORT" | "IN" | "INTERFACE" |
"LOCK" | "LOOP" | "METHODS" |
"MOD" | "MODULE" | "NOT" |
"OBJECT" | "OF" | "OR" | "OVERRIDES" |
"PROCEDURE" |
"RAISE" | "RAISES" | "READONLY" |
"RECORD" | "REF" | "REPEAT" | "RETURN" |
"REVEAL" | "ROOT" | "SET" |
"THEN" | "TO" | "TRY" | "TYPE" | "TYPECASE" |
"UNSAFE" | "UNTIL" | "UNTRACED" |
"VALUE" | "VAR" | "WHILE" | "WITH" .

(5) Reserved-Identifier = ??? .

(6) Identifiers = ⟨ Identifier ⟨ "," Identifier ⟩* ⟩ .

(7) Qualified-Identifier = ⟨ Identifier ⟨ "." Identifier ⟩? ⟩ .

(8) Qualified-Identifiers = ⟨ Qualified-Identifier ⟨ "," Qualified-Identifier ⟩* ⟩ .

1.2 Literals

includes: **[2]/Data Notation/Characters** (letter, digit, octal-digit) .

(1) Literal = Integer-Numeral | Real-Numeral |
Character-Literal | Text-Literal .

(2) Integer-Numeral = ⟦ digit$^+$ ⟧ | ⟦ digit$^+$ “_” hex-digit$^+$ ⟧ .

(3) Real-Numeral = ⟦ digit$^+$ “.” digit$^+$ Exponent$^?$ ⟧ .

(4) Exponent = ⟨ (“E” | “e” | “D” | “d” | “X” | “x”)
(“+” | “-”)$^?$ digit$^+$ ⟩ .

(5) Character-Literal = ⟦ “'” (printing-character | “"” | Escape) “'” ⟧ .

(6) Text-Literal = ⟦ “"” (printing-character | “'” | Escape)* “"” ⟧ .

(7) Escape = “\n” | “\t” | “\r” | “\f” | “\\” | “\'” | “\"” |
⟨ “\” octal-digit^3 ⟩ .

(8) printing-character = letter | digit | other-character .

(9) other-character = “ ” | “!” | “#” | “$” | “%” | “&” | “(” | “)” |
“*” | “+” | “,” | “-” | “.” | “/” | “:” | “;” |
“<” | “=” | “>” | “?” | “@” | “[” | “]” | “^” |
“_” | “`” | “{” | “|” | “}” | “~” |
extended-character .

(10) extended-character = □ .

(11) hex-digit = digit |
“A” | “B” | “C” | “D” | “E” | “F” |
“a” | “b” | “c” | “d” | “e” | “f” .

1.3 Expressions

needs: **Identifiers**, **Literals**, **Types** .

(1) Expression = Identifier |
Literal |
Constructor |
⟦ “(” Expression “)” ⟧ |
⟦ Monadic-Operator Expression ⟧ |
⟦ Expression (Dyadic-Operator | “AND” | “OR”)
Expression ⟧ |
⟦ Expression “^” ⟧ |
⟦ Expression “.” Identifier ⟧ |
⟦ Type “.” Identifier ⟧ |
⟦ Expression “[” Expressions “]” ⟧ |
⟦ Expression “(” Actuals “)” ⟧ .

(2) Expressions = ⟨ Expression ⟨ “,” Expression ⟩* ⟩ .

(3) Constructor = ⟦ Type “{” (Set-Constructor |
Array-Constructor | Record-Constructor)$^?$ “}” ⟧ .

(4) Set-Constructor = Expression | ⟦ Expression ".." Expression ⟧ |
⟨ Set-Constructor "," Set-Constructor ⟩ .

(5) Array-Constructor = ⟨ Expressions ⟨ "," ".." ⟩$^?$ ⟩ .

(6) Record-Constructor = ⟨ Record-Element ⟨ "," Record-Element ⟩* ⟩ .

(7) Record-Element = Expression | ⟦ Identifier ":=" Expression ⟧ .

(8) Actuals = ⟨ Actual ⟨ "," Actual ⟩* ⟩$^?$.

(9) Actual = Expression | ⟦ Identifier ":=" Expression ⟧ | Type .

(10) Constant-Expression ≤ Expression .

(11) Monadic-Operator = "+" | "-" | "NOT" .

(12) Dyadic-Operator = "+" | "-" | "*" | "/" | "DIV" | "MOD" |
"=" | "#" | "<=" | ">=" | "<" | ">" | "IN" |
"&" .

1.4 Statements

needs: **Identifiers**, **Expressions**, **Blocks and Declarations**, **Types** .

(1) Statements = ⟨ Statement ⟨ ";" Statement ⟩* ";"$^?$ ⟩$^?$.

(2) Statement = Block |
⟦ Expression ":=" Expression ⟧ |
⟦ Expression "(" Actuals ")" ⟧ |
⟦ "CASE" Expression "OF" Cases
⟨ "ELSE" Statements ⟩$^?$ "END" ⟧ |
⟦ "EXIT" ⟧ |
⟦ "EVAL" Expression ⟧ |
⟦ "FOR" Identifier ":=" Expression "TO" Expression
⟨ "BY" Expression ⟩$^?$ "DO" Statements "END" ⟧ |
⟦ "IF" Expression "THEN" Statements
⟨ "ELSIF" Expression "THEN" Statements ⟩*
⟨ "ELSE" Statements ⟩$^?$ "END" ⟧ |
⟦ "LOCK" Expression "DO" Statements "END" ⟧ |
⟦ "LOOP" Statements "END" ⟧ |
⟦ "RAISE" Qualified-Identifier ⟨ "(" Expression ")" ⟩$^?$ ⟧ |
⟦ "REPEAT" Statements "UNTIL" Expression ⟧ |
⟦ "RETURN" Expression$^?$ ⟧ |
⟦ "TRY" Statements "EXCEPT" Handlers
⟨ "ELSE" Statements ⟩$^?$ "END" ⟧ |
⟦ "TRY" Statements "FINALLY" Statements "END" ⟧ |
⟦ "TYPECASE" Expression "OF" Type-Cases
⟨ "ELSE" Statements ⟩$^?$ "END" ⟧ |
⟦ "WHILE" Expression "DO" Statements "END" ⟧ |
⟦ "WITH" Bindings "DO" Statements "END" ⟧ .

(3) Cases = ⟨ Case$^?$ ⟨ "|" Case ⟩* ⟩ .

(4) Case = ⟦ Case-Labels "=>" Statements ⟧ .

(5) Case-Labels = ⟨ Case-Label ⟨ "," Case-Label ⟩* ⟩ .

(6) Case-Label = Constant-Expression |
⟦ Constant-Expression ".." Constant-Expression ⟧ .

(7) Type-Cases = ⟨ Type-Case$^?$ ⟨ "|" Type-Case ⟩* ⟩ .

(8) Type-Case = ⟦ Types ⟨ "(" Identifier ")" ⟩$^?$ "=>" Statements ⟧ .

(9) Types = ⟨ Type ⟨ "," Type ⟩* ⟩ .

(10) Handlers = ⟨ Handler$^?$ ⟨ "|" Handler ⟩* ⟩ .

(11) Handler = ⟦ Qualified-Identifiers ⟨ "(" Identifier ")" ⟩$^?$ "=>" Statements ⟧ .

(12) Bindings = ⟨ Binding ⟨ "," Binding ⟩* ⟩ .

(13) Binding = ⟦ Identifier "=" Expression ⟧ .

1.5 Blocks and Declarations

needs: **Identifiers**, Expressions, Statements, Types .

(1) Block = ⟦ Declaration* "`BEGIN`" Statements "`END`" ⟧ .

(2) Declaration = ⟦ "`CONST`" ⟨ Constant-Declaration ";" ⟩* ⟧ |
⟦ "`VAR`" ⟨ Variable-Declaration ";" ⟩* ⟧ |
⟦ "`TYPE`" ⟨ Type-Declaration ";" ⟩* ⟧ |
⟦ "`REVEAL`" ⟨ Reveal-Declaration ";" ⟩* ⟧ |
⟦ "`EXCEPTION`" ⟨ Exception-Declaration ";" ⟩* ⟧ |
⟦ Procedure-Declaration ";" ⟧ .

(3) Constant-Declaration = ⟦ Identifier ⟨ ":" Type ⟩$^?$ "=" Constant-Expression ⟧ .

(4) Variable-Declaration = ⟦ Identifiers ⟨ ":" Type ⟩$^?$ ":=" Expression ⟧ |
⟦ Identifiers ":" Type ⟧ .

(5) Type-Declaration = ⟦ Identifier ("=" | "<:") Type ⟧ .

(6) Reveal-Declaration = ⟦ Qualified-Identifier ("=" | "<:") Type ⟧ .

(7) Exception-Declaration = ⟦ Identifier ⟨ "(" Type ")" ⟩$^?$ ⟧ .

(8) Procedure-Declaration = ⟦ Procedure-Heading ⟨ "=" Block Identifier ⟩$^?$ ⟧ .

(9) Procedure-Heading = ⟨ "`PROCEDURE`" Identifier Signature ⟩ .

(10) Signature = ⟨ "(" Formals ")" ⟨ ":" Type ⟩$^?$
⟨ "`RAISES`" Raises ⟩$^?$ ⟩ .

(11) Formals = ⟨ Formal ⟨ ";" Formal ⟩* ";"$^?$ ⟩$^?$.

(12) Formal = ⟦ Mode Identifiers ⟨ ":" Type ⟩$^?$
":=" Constant-Expression ⟧ |
⟦ Mode Identifiers ":" Type ⟧ .

(13) Mode = "`VALUE`" | "`VAR`" | "`READONLY`" .

(14) Raises = ⟦ "{" Qualified-Identifiers$^?$ "}" ⟧ | ⟦ "`ANY`" ⟧ .

1.6 Types

needs: Identifiers, Expressions, Blocks and Declarations .

(1) Type-Name = Qualified-Identifier | ⟦ “UNTRACED”$^?$ “ROOT” ⟧ .

(2) Type = Type-Name |
⟦ “ARRAY” Types$^?$ “OF” Type ⟧ |
⟦ “BITS” Constant-Expression “FOR” Type ⟧ |
⟦ “{” Identifiers$^?$ “}” ⟧ |
Object-Type |
⟦ “PROCEDURE” Signature ⟧ |
⟦ “RECORD” Fields “END” ⟧ |
⟦ “UNTRACED”$^?$ Brand$^?$ “REF” Type ⟧ |
⟦ “SET” “OF” Type ⟧ |
⟦ “[” Constant-Expression “..” Constant-Expression “]” ⟧ |
⟦ “(” Type “)” ⟧ .

(3) Object-Type = ⟦ (Type-Name | Object-Type)$^?$ Brand$^?$
“OBJECT” Fields ⟨ “METHODS” Methods ⟩$^?$
⟨ “OVERRIDES” Overrides ⟩$^?$ “END” ⟧ .

(4) Brand = ⟦ “BRANDED” Constant-Expression$^?$ ⟧ .

(5) Fields = ⟨ Field ⟨ “;” Field ⟩* “;”$^?$ ⟩$^?$.

(6) Field = ⟦ Identifiers ⟨ “:” Type ⟩$^?$ “:=” Constant-Expression ⟧ |
⟦ Identifiers “:” Type ⟧ .

(7) Methods = ⟨ Method ⟨ “;” Method ⟩* “;”$^?$ ⟩$^?$.

(8) Method = ⟦ Identifier Signature ⟨ “:=” Constant-Expression ⟩$^?$ ⟧ .

(9) Overrides = ⟨ Override ⟨ “;” Override ⟩* “;”$^?$ ⟩$^?$.

(10) Override = ⟦ Identifier “:=” Constant-Expression ⟧ .

1.7 Compilation Units

needs: Identifiers, Blocks and Declarations .

(1) Compilation = ⟦ “UNSAFE”$^?$ (Interface | Module) ⟧ |
Generic-Interface | Generic-Module .

(2) Interface = ⟦ “INTERFACE” Identifier “;”
Imports Declaration*
“END” Identifier “.” ⟧ |
⟦ “INTERFACE” Identifier “=”
Identifier “(” Identifiers$^?$ “)”
“END” Identifier “.” ⟧ .

(3) Module = ⟦ “MODULE” Identifier ⟨ “EXPORTS” Identifiers ⟩$^?$ “;”
Imports Block Identifier “.” ⟧ |
⟦ “MODULE” Identifier ⟨ “EXPORTS” Identifiers ⟩$^?$ “=”
Identifier “(” Identifiers$^?$ “)”
“END” Identifier “.” ⟧ .

(4) Generic-Interface = ⟦ "`GENERIC`" "`INTERFACE`"
Identifier "(" Identifiers$^{?}$ ")" ";"
Imports Declaration*
"`END`" Identifier "`.`" ⟧ .

(5) Generic-Module = ⟦ "`GENERIC`" "`MODULE`" Identifier "(" Identifiers$^{?}$ ")" ";"
Imports Block Identifier "`.`" ⟧ .

(6) Imports = Import* .

(7) Import = ⟦ "`IMPORT`" Import-Items ";" ⟧ **|**
⟦ "`FROM`" Identifier "`IMPORT`" Identifiers ";" ⟧ .

(8) Import-Items = ⟨ Import-Item ⟨ "," Import-Item ⟩* ⟩ .

(9) Import-Item = Identifier **|** ⟦ Identifier "`AS`" Identifier ⟧ .

1.8 Programs

needs: Compilation Units .

(1) Program = Compilation^{+} .

It is assumed that the compilations are ordered according to the initialisation rule.

Chapter 2

Semantic Functions

needs: **Abstract Syntax**, **Semantic Entities**, **[2]/Action Notation**.

2.1 Identifiers

introduces: the identifiers of _ , the entity identified by _ .

- the identifiers of _ :: Identifiers$^?$ $\rightarrow$ token* .

(1) the identifiers of $\langle\ \rangle$ = ( ) .

(2) the identifiers of I:Identifier = I .

(3) the identifiers of $\langle$ I_1:Identifier "," I_2:Identifiers $\rangle$ =
  (I_1, the identifiers of I_2) .

- the entity identified by _ :: Qualified-Identifier $\rightarrow$
  yielder [of a bindable] [using current bindings] .

(4) the entity identified by I:Identifier =
  the bindable bound to I .

(5) the entity identified by (I_1:Identifier "." I_2:Identifier) =
  the component I_2 of the interface bound to I_1 .

2.2 Literals

introduces: the value of _ .

privately introduces: the character-value of _ , the character-values of _ .

- the value of _ :: Literal $\rightarrow$ literal-value .

- the value of _ :: Integer-Numeral $\rightarrow$ integer-number .

(1) the value of $[\![$ c:digit$^+$ $]\!]$ =
  the integer-number of the numerical-value of c .

(2) the value of $[\![$ c:$\langle$ digit$^+$ "_" hex-digit$^+$ $\rangle$ $]\!]$ =
the integer-number of the numerical-value of c .

- the value of _ :: Real-Numeral → real-number .

(3) the value of $[\![$ c:$\langle$ digit$^+$ "." digit$^+$ Exponent$^?$ $\rangle$ $]\!]$ =
the real-number of the numerical-value of c .

- the value of _ :: Character-Literal → character .

(4) the value of _ $[\![$ "'" c:(printing-character | """ | Escape) "'" $]\!]$ =
the character-value of c .

- the value of _ :: Text-Literal → text-value .

(5) the value of $[\![$ """ c:(printing-character | "'" | Escape)* """ $]\!]$ =
the text-value of the character-values of c .

- the character-value of _ ::
(printing-character | "'" | """ | Escape) → character .

(6) the character-value of c:(printing-character | "'" | """) = c .

(7) the character-value of "`\n`" = control 'J' .

(8) the character-value of "`\t`" = control 'I' .

(9) the character-value of "`\r`" = control 'M' .

(10) the character-value of "`\f`" = control 'L' .

(11) the character-value of "\\" = '\' .

(12) the character-value of "\'" = ''' .

(13) the character-value of "\"" = '"' .

(14) code c = octal (string of d) ⇒
the character-value of $\langle$ "\" d:octal-digit^3 $\rangle$ = c .

- the character-values of _ ::
(printing-character | "'" | Escape)* → character* .

(15) the character-values of $\langle$ $\rangle$ = () .

(16) the character-values of $\langle$ c_1:(printing-character | "'" | Escape)
c_2:(printing-character | "'" | Escape)* $\rangle$ =
(the character-value of c_1, the character-values of c_2) .

2.3 Expressions

needs: **Identifiers**, **Literals**, **Types**.

introduces: evaluate _ , investigate _ , apply to _ ,
the value of _ ,
the type of _ .

privately introduces: memberwise evaluate _ ,
elementwise evaluate _ , fieldwise evaluate _ ,
formally evaluate _ ,
the member-set of _ ,
the element-values of _ , the field-values of _ ,
the monadic-result of _ , the dyadic-result of _ .

- evaluate _ :: Expression $\rightarrow$
  action [giving a value | storing | diverging | escaping with an exception | erring]
  [using current bindings | current storage] .

(1) evaluate E:Expression =
investigate E then
| give the given value or
| give the value assigned to the given variable .

- investigate _ :: Expression $\rightarrow$
  action [giving a (value | variable | method) |
  storing | diverging | escaping with an exception | erring]
  [using current bindings | current storage] .

(2) investigate I:Identifier =
give the (value | variable | method) bound to I .

(3) investigate L:Literal =
give the value of L .

(4) investigate ⟦ "(" E:Expression ")" ⟧ =
investigate E .

(5) investigate ⟦ O:Monadic-Operator E:Expression ⟧ =
evaluate E then
| give the monadic-result of O reporting any given error .

(6) investigate ⟦ E_1:Expression O:Dyadic-Operator E_2:Expression ⟧ =
| evaluate E_1 and evaluate E_2
then
| give the dyadic-result of O reporting any given error .

(7) investigate ⟦ E_1:Expression "`AND`" E_2:Expression ⟧ =
evaluate E_1 then
| evaluate E_2 else
| give false .

Note use of '_ else _', here and elsewhere.

(8) investigate ⟦ E_1:Expression "`OR`" E_2:Expression ⟧ =
evaluate E_1 then
| give true else
| evaluate E_2 .

```
(9)  investigate [[ E:Expression "^" ]] =
        evaluate E then
        | give the referent of the given reference-value reporting any given error .

(10) investigate [[ E:Expression "." I:Identifier ]] =
        investigate E then
        | give the field I of the given (record-value | record-variable)
        | or
        | | | | give the referent of the given reference-value or
        | | | | give the referent of the reference-value assigned to the given variable
        | | | reporting any given error
        | | then give the field I of the given record-variable
        | or
        | | | | give the referent of the given object-value or
        | | | | give the referent of the object-value assigned to the given variable
        | | | reporting any given error
        | | then
        | | | give the field I of the given object or
        | | | give the method I of the given object .

(11) investigate [[ T:Type "." I:Identifier ]] =
        give the type denoted by T then
        | give the enumerand named I of the given enumeration-type or
        | give the method I of the given object-type .

(12) investigate [[ E1:Expression "[" E2:Expression "]" ]] =
        | investigate E1 and evaluate E2
        then
        | give the element indexed by the given ordinal-value#2
        |          in the given (array-value | array-variable)#1
        |     reporting any given error .

(13) investigate [[ E1:Expression "[" E2:Expression "," E3:Expressions "]" ]] =
        investigate [[ [[ E1 "[" E2 "]" ]] "[" E3 "]" ]] .

(14) investigate [[ E:Expression "(" A:Actuals ")" ]] =
        evaluate E then
        | | give the given (procedure | method) then
        | | apply to A then
        | | | give the given value or
        | | | | check there is given () then
        | | | | report the missing-function-result-error
        | or
        | | check there is given nil then
        | | report the nil-procedure-or-method-error
        | or
        | | check there is given a (stub-procedure | stub-method) then
        | | report the stub-procedure-or-method-error .
```

2.3.1 Constructors

- evaluate _ :: Constructor →
        action [giving a (set-value | array-value | record-value) |
                  storing | diverging | escaping with an exception | erring]
              [using current bindings | current storage] .

(1) evaluate ⟦ T:Type "{" "}" ⟧ =
give the type denoted by T then
| check there is the given set-type then give the empty-set
or
| check there is the given array-type and then
| give the array of ((), the index-type of the given array-type)
or
| check there is the given record-type and then
| give the record of the defaulted-field-values of
| the fields of the given record-type .

(2) evaluate ⟦ T:Type "{" S:Set-Constructor "}" ⟧ =
memberwise evaluate S .

(3) evaluate ⟦ T:Type "{" A:Array-Constructor "}" ⟧ =
give the index-type of the type denoted by T then
| | give the cardinality of the given ordinal-type then
| | elementwise evaluate A
| and give the given ordinal-type
then give the array of the given (value$^+$, ordinal-type) .

(4) evaluate ⟦ T:Type "{" R:Record-Constructor "}" ⟧ =
give the fields of the type denoted by T then
| fieldwise evaluate R and
| give the given associations[(type, value$^?$)]
then give the record of the defaulted-field-values of
the given (associations[value], associations[(type, value$^?$)]) .

- memberwise evaluate _ :: Set-Constructor →
action [giving a set-value | storing | diverging |
escaping with an exception | erring]
[using current bindings | current storage] .

(5) memberwise evaluate E:Expression =
evaluate E
then give the set of the given ordinal-value .

(6) memberwise evaluate ⟦ E_1:Expression ".." E_2:Expression ⟧ =
| evaluate E_1 and evaluate E_2
then give the set of the interval-values of the given ordinal-value^2 .

(7) memberwise evaluate ⟨ S_1:Set-Constructor "," S_2:Set-Constructor ⟩ =
| memberwise evaluate S_1 and memberwise evaluate S_2
then give the union of the given set-value^2 .

- elementwise evaluate _ :: Array-Constructor →
action [giving value* | storing | diverging | escaping with an exception | erring]
[using the given natural | current bindings | current storage] .

(8) elementwise evaluate E:Expression =
evaluate E .

(9) elementwise evaluate ⟨ E:Expression "," ".." ⟩ =
| give the given natural and evaluate E
then give (the given value#2)$^{\text{(the given natural\#1)}}$.

(10) elementwise evaluate ⟨ E:Expression "," A:Array-Constructor ⟩ =
evaluate E and
| give the predecessor of the given natural then elementwise evaluate A .

- fieldwise evaluate _ :: (Record-Constructor | Record-Element) →
action [giving associations[value] | storing | diverging |
escaping with an exception | erring]
[using the given associations[type] |
current bindings | current storage] .

(11) fieldwise evaluate E:Expression =
| give the first of the given associations[type] and
| evaluate E
then give the association of the identifier of the given association[type]#1
to the given value#2 .

(12) fieldwise evaluate ⟨ E:Expression "," R:Record-Constructor ⟩ =
fieldwise evaluate E and
| give the rest of the given associations[type] then
| fieldwise evaluate R .

(13) fieldwise evaluate ⟦ I:Identifier ":=" E:Expression ⟧ =
evaluate E
then give the association of I to the given value#2 .

(14) fieldwise evaluate ⟨ R_1:⟦ Identifier ":=" Expression ⟧ ","
R_2:Record-Constructor ⟩ =
fieldwise evaluate R_1 and fieldwise evaluate R_2 .

2.3.2 Actuals

- apply to _ :: Actuals →
action [giving a value? | storing | diverging |
escaping with an exception | erring]
[using the given (procedure | method) |
current bindings | current storage] .

(1) apply to A:Actuals =

| give the body-abstraction of the given procedure and
| | give the formal-parameters of the signature of the given procedure then
| | | formally evaluate A and
| | | give the given formal-parameter*
| | then give the defaulted-arguments of
| | the given (actual-parameter*, formal-parameter*)
| then enact the application of the given abstraction#1 to
| the actual-parameter* yielded by the rest of the given data
or
| give the body-abstraction of the given method and
| give the association of the self-identifier of the given method
| to a traced reference to the object of the given method and
| | give the formal-parameters of the signature of the given method then
| | | formally evaluate A and
| | | give the given formal-parameter*
| | then give the defaulted-arguments of
| | the given (actual-parameter*, formal-parameter*)
| then enact the application of the given abstraction#1 to
| the actual-parameter* yielded by the rest of the given data .

'defaulted-arguments (a:associations[argument], f:associations[mode, type, value?])' yields the tuple of associations a defaulted (where necessary) according to f.

- formally evaluate _ :: Actuals →
  action [giving actual-parameter* | storing | diverging |
  escaping with an exception | erring]
  [using the given formal-parameter* | current bindings | current storage] .

(2) formally evaluate ⟨ ⟩ =
give () .

(3) formally evaluate E:Expression =
| | give the given (value-parameter | read-only-parameter) and then
| | evaluate E
| then give the association of the identifier of the given formal-parameter#1
| to (the mode of the given formal-parameter#1, the given value#2)
or
| | give the given variable-parameter and then
| | investigate E
| then give the association of the identifier of the given formal-parameter#1
| to (the variable-mode, the given variable#2) .

(4) formally evaluate ⟨ E:Expression "," A:Actuals ⟩ =
| give the first of the given formal-parameter$^+$ then
| formally evaluate E
and
| give the rest of the given formal-parameter$^+$ then
| formally evaluate A .

(5) formally evaluate ⟦ I:Identifier ":=" E:Expression ⟧ =

give the component associated with I in the given formal-parameter$^+$ then
| | give the given (value-parameter | read-only-parameter) and then
| | evaluate E
| then give the association of I
| to (the mode of the given formal-parameter#1, the given value#2)
or
| | give the given variable-parameter and then
| | investigate E
| then give the association of I
| to (the variable-mode, the given variable#2) .

(6) formally evaluate $\langle A_1$:⟦ Identifier ":=" Expression ⟧ "," A_2:Actuals $\rangle$ =
formally evaluate A_1 and formally evaluate A_2 .

(7) formally evaluate T:Type =
give the type denoted by T .

2.3.3 Constant-Expressions

introduces: the value of _ .

- the value of _ :: Constant-Expression →
  yielder [of a value] [using current bindings] .

(1) the value of I:Identifier =
the value bound to I .

(2) the value of ⟦ "(" E:Expression ")" ⟧ =
the value of E .

(3) the value of ⟦ O:Monadic-Operator E:Expression ⟧ =
the value yielded by
(the monadic-result of O taking the value of E) .

(4) the value of ⟦ E_1:Expression O:Dyadic-Operator E_2:Expression ⟧ =
the value yielded by
(the dyadic-result of O taking (the value of E_1, the value of E_2)) .

Note use of the proposed new yielder operation '_ taking _'.

(5) the value of ⟦ E_1:Expression "AND" E_2:Expression ⟧ =
if the value of E_1 is true then the value of E_2 else false .

(6) the value of ⟦ E_1:Expression "OR" E_2:Expression ⟧ =
if the value of E_1 is true then true else the value of E_2 .

(7) the value of [E:Expression "^" ⟧ =
nothing .

(8) the value of ⟦ E:Expression "." I:Identifier ⟧ =
the field I of the record-value yielded by the value of E .

(9) the value of ⟦ T:Type "." I:Identifier ⟧ =
the enumerand named I of
the enumeration-type yielded by the type denoted by T .

(10) the value of ⟦ E_1:Expression "[" E_2:Expression "]" ⟧ =
the value yielded by the element indexed by
the ordinal-value yielded by the value of E_1
in the array-value yielded by the value of E_2 .

(11) the value of ⟦ E_1:Expression "[" E_2:Expression "," E_3:Expressions "]" ⟧ =
the value of ⟦ ⟦ E_1 "[" E_2 "]" ⟧ "[" E_3 "]" ⟧ .

(12) the value of ⟦ E:Expression "(" A:Actuals ")" ⟧ =
nothing .

BUG: This is too restrictive. A constant-expression may call functions in the Word interface, type operations, etc.

- the value of _ :: Constructor →
yielder [of a (set-value | array-value | record-value)]
[using current bindings] .

(13) the value of ⟦ T:Type "{" "}" ⟧ =
when there is the set-type yielded by the type denoted by T then
the empty-set |
when there is the array-type yielded by the type denoted by T then
the array of ((), the index-type of the type denoted by T) |
the record of the defaulted-field-values of
the fields of the record-type yielded by the type denoted by T .

(14) the value of ⟦ T:Type "{" S:Set-Constructor "}" ⟧ =
the member-set of S .

(15) the value of ⟦ T:Type "{" A:Array-Constructor "}" ⟧ =
the array of (
the element-values of A
taking the cardinality of the index-type of the type denoted by T,
the index-type of the type denoted by T) .

(16) the value of ⟦ T:Type "{" R:Record-Constructor "}" ⟧ =
the record of the defaulted-field-values of (
the field-values of R taking the fields of the type denoted by T,
the fields of the type denoted by T) .

- the member-set of _ :: Set-Constructor →
yielder [of a set-value] [using current bindings] .

(17) the member-set of E:Expression =
the set of the value of E .

(18) the member-set of ⟦ E_1:Expression ".." E_2:Expression ⟧ =
the set of the interval-values of
(the value of E_1, the value of E_2) .

(19) the member-set of ⟨ S_1:Set-Constructor "," S_2:Set-Constructor ⟩ =
the union of (the member-set of S_1, the member-set of S_2) .

- the element-values of _ :: Array-Constructor →
yielder [of value*] [using the given natural | current bindings] .

(20) the element-values of E:Expression =
the value of E .

(21) the element-values of $\langle$ E:Expression "," ".." $\rangle$ =
(the value of E)$^{\text{(the given natural)}}$.

(22) the element-values of $\langle$ E:Expression "," A:Array-Constructor $\rangle$ =
(the value of E,
the element-values of A taking the predecessor of the given natural) .

- the field-values of _ :: (Record-Constructor | Record-Element) $\rightarrow$
yielder [of associations[value]]
[using the given associations[type] | current bindings] .

(23) the field-values of E:Expression =
the association of the identifier of the first of the given associations[type]
to the value of E .

(24) the field-values of $\langle$ E:Expression "," R:Record-Constructor $\rangle$ =
(the field-values of E,
the field-values of R taking the rest of the given associations[type] .

(25) the field-values of $[\![$ I:Identifier ":=" E:Expression $]\!]$ =
the association of I to the value of E .

(26) the field-values of $\langle$ R_1:$[\![$ Identifier ":=" Expression $]\!]$ ","
R_2:Record-Constructor $\rangle$ =
(the field-values of R_1, the field-values of R_2) .

2.3.4 Operators

- the monadic-result of _ :: Monadic-Operator $\rightarrow$
yielder [of a (value | error)] [using the given value] .

(1) the monadic-result of "+" =
the given integer-number |
the given real-number .

(2) the monadic-result of "–" =
the negation of the given integer-number |
the negation of the given real-number .

(3) the monadic-result of "`NOT`" =
not (the given truth-value) .

- the dyadic-result of _ :: Dyadic-Operator $\rightarrow$
yielder [of a (value | error)] [using the given value^2] .

(4) the dyadic-result of "+" =
the sum of the given integer-number^2 |
the sum of the given real-number^2 |
the union of the given set-value^2 .

(5) the dyadic-result of "–" =
the difference of the given integer-number^2 |
the difference of the given real-number^2 |
the difference of the given set-value^2) .

(6) the dyadic-result of "`*`" =
  the product of the given integer-number^2 |
  the product of the given real-number^2 |
  the intersection of the given set-value^2 .

(7) the dyadic-result of "/" =
  the quotient of the given real-number^2 |
  the symmetric-difference of the given set-value^2 .

(8) the dyadic-result of "`DIV`" =
  the integer-quotient of the given integer-number^2 .

(9) the dyadic-result of "`MOD`" =
  the modulo of the given integer-number^2 |
  the modulo of the given real-number^2 .

(10) the dyadic-result of "=" =
  (the given value#1 is the given value#2) .

(11) the dyadic-result of "`#`" =
  not (the given value#1 is the given value#2) .

(12) the dyadic-result of "<=" =
  not (the given ordinal-value#1 is greater than the given ordinal-value#2) |
  not (the given real-number#1 is greater than the given real-number#2) |
  not (the given address#1 is greater than the given address#2) |
  (the given set-value#1 is included in the given set-value#2) .

(13) the dyadic-result of ">=" =
  not (the given ordinal-value#1 is less than the given ordinal-value#2) |
  not (the given real-number#1 is less than the given real-number#2) |
  not (the given address#1 is less than the given address#2) |
  (the given set-value#2 is included in the given set-value#1) .

(14) the dyadic-result of "<" =
  (the given ordinal-value#1 is less than the given ordinal-value#2) |
  (the given real-number#1 is less than the given real-number#2) |
  (the given address#1 is less than the given address#2) |
  (the given set-value#1 is properly included in the given set-value#2) .

(15) the dyadic-result of ">" =
  (the given ordinal-value#1 is greater than the given ordinal-value#2) |
  (the given real-number#1 is greater than the given real-number#2) |
  (the given address#1 is greater than the given address#2) |
  (the given set-value#2 is properly included in the given set-value#1) .

(16) the dyadic-result of "`IN`" =
  the given ordinal-value#1 is in the given set-value#2 .

(17) the dyadic-result of " & " =
  the concatenation of the given text-value^2 .

2.3.5 Types of Expressions

- the type of _ :: Expression → type .

  Not yet defined.

2.4 Statements

needs: **Identifiers**, **Expressions**, **Blocks and Declarations**, **Types**.

introduces: execute _ .

privately introduces: select from _ , selectable from _ ,
type-select from _ , type-selectable from _ ,
handle in _ , possible to handle in _ .

- execute _ :: (Statements | Statement) $\rightarrow$
  action [completing | storing | diverging | escaping with an exception | erring]
  [using current bindings | current storage] .

(1) execute $\langle\ \rangle$ =
  complete .

(2) execute $\langle$ S_1:Statements ";" S_2:Statement $\rangle$ =
  execute S_1 and then execute S_2 .

(3) execute B:Block =
  activate B .

(4) execute ⟦ E_1:Expression ":=" E_2:Expression ⟧ =
  | investigate E_1 and evaluate E_2
  then assign the given value#2 to the given variable#1 .

(5) execute ⟦ E:Expression "(" A:Actuals ")" ⟧ =
  evaluate E then
  | | give the given (procedure | method) then
  | | apply to A
  | or
  | | check there is given nil then
  | | report the nil-procedure-or-method-error
  | or
  | | check there is given a (stub-procedure | stub-method) then
  | | report the stub-procedure-or-method-error .

(6) execute ⟦ "CASE" E:Expression "OF" C:Cases "END" ⟧ =
  evaluate E then
  | select from C or
  | | check not selectable from C and then
  | | report the case-error .

(7) execute ⟦ "CASE" E:Expression "OF" C:Cases
  "ELSE" S:Statements "END" ⟧ =
  evaluate E then
  | select from C or
  | | check not selectable from C and then
  | | execute S .

(8) execute ⟦ "EXIT" ⟧ =
  escape with the exit-exception .

(9) execute ⟦ "EVAL" Expression ⟧ =
  evaluate E then complete .

(10) execute ⟦ "FOR" I:Identifier ":=" E_1:Expression "TO" E_2:Expression
"DO" S:Statements "END" ⟧ =
| evaluate E_1 and evaluate E_2
then give the interval-values of the given ordinal-value^2 then
| unfolding
| | | | check there is given () then complete
or
| | | | furthermore bind I to the first of the given ordinal-value$^+$
| hence execute S
and then
| give the rest of the given ordinal-value$^+$ then unfold
trap handle the exit-exception .

(11) execute ⟦ "FOR" I:Identifier ":=" E_1:Expression "TO" E_2:Expression
"DO" S:Statements "END" ⟧ =
execute ⟦ "FOR" I ":=" E_1 "TO" E_2 "BY" "1" "DO" S "END" ⟧ .

Above equation is redundant.

(12) execute ⟦ "FOR" I:Identifier ":=" E_1:Expression "TO" E_2:Expression
"BY" E_3:Expression "DO" S:Statements "END" ⟧ =
| evaluate E_1 and evaluate E_2 and evaluate E_3
then give the interval-values of the given (ordinal-value^2, integer-number) then
| unfolding
| | | | check there is given () then complete
or
| | | | furthermore bind I to the first of the given ordinal-value$^+$
| hence execute S
and then
| give the rest of the given ordinal-value$^+$ then unfold
trap handle the exit-exception .

(13) execute ⟦ "IF" E:Expression "THEN" S:Statements "END" ⟧ =
evaluate E then
| execute S else
| complete .

(14) execute ⟦ "IF" E:Expression "THEN" S:Statements "END" ⟧ =
execute ⟦ "IF" E "THEN" S "ELSE" "END" ⟧ .

Above equation is redundant.

(15) execute ⟦ "IF" E:Expression "THEN" S_1:Statements
"ELSE" S_2:Statements "END" ⟧ =
evaluate E then
| execute S_1 else
| execute S_2 .

(16) execute ⟦ "IF" E_1:Expression "THEN" S_1:Statements
"ELSIF" E_2:Expression "THEN" S_2:Statements
X:⟨ "ELSIF" Expression "THEN" Statements ⟩*
Y:⟨ "ELSE" Statements ⟩$^?$ "END" ⟧ =
execute ⟦ "IF" E_1 "THEN" S_1 "ELSE"
⟦ "IF" E_2 "THEN" S_2 X Y "END" ⟧
"END" ⟧ .

(17) execute ⟦ "`LOCK`" E:Expression "`DO`" S:Statements "`END`" ⟧ =
    evaluate E then
    | store locked in the given primitive-variable and then
    | execute S and then
    | store unlocked in the given primitive-variable .

(18) execute ⟦ "`LOOP`" S:Statements "`END`" ⟧ =
    unfolding
    | | execute S and then unfold
    | trap handle the exit-exception .

(19) execute ⟦ "`RAISE`" I:Qualified-Identifier ⟧ =
    escape with the primitive-exception yielded by the entity identified by I .

(20) execute ⟦ "`RAISE`" I:Qualified-Identifier "(" E:Expression ")" ⟧ =
    evaluate E then
    escape with exception of (the entity identified by I, the given value) .

(21) execute ⟦ "`REPEAT`" S:Statements "`UNTIL`" E:Expression ⟧ =
    unfolding
    | | execute S and then
    | | | evaluate E then
    | | | | complete else
    | | | | unfold
    | trap handle the exit-exception .

(22) execute ⟦ "`RETURN`" ⟧ =
    escape with the return-exception .

(23) execute ⟦ "`RETURN`" E:Expression ⟧ =
    evaluate E then
    escape with the exception of (the return-exception, the given value) .

(24) execute ⟦ "`TRY`" S:Statements "`EXCEPT`" H:Handlers "`END`" ⟧ =
    execute S trap
    | | handle in H
    | or
    | | check not possible to handle in H and then
    | | escape with the given exception .

(25) execute ⟦ "`TRY`" S_1:Statements "`EXCEPT`" H:Handlers
        "`ELSE`" S_2:Statements "`END`" ⟧ =
    execute S_1 trap
    | | handle in H
    | or
    | | check not possible to handle in H and then
    | | execute S_2 .

(26) execute ⟦ "`TRY`" S_1:Statements "`FINALLY`" S_2:Statements "`END`" ⟧ =
    execute S_1 and then terminally execute S_2 .

(27) execute ⟦ "`TYPECASE`" E:Expression "`OF`" C:Type-Cases "`END`" ⟧ =
    evaluate E then
    | type-select from C or
    | | check not type-selectable from C and then
    | | report the type-case-error .

(28) execute ⟦ "TYPECASE" E:Expression "OF" C:Type-Cases
"ELSE" S:Statements "END" ⟧ =
evaluate E then
| type-select from C or
| | check not type-selectable from C and then
| | execute S .

(29) execute ⟦ "WHILE" E:Expression "DO" S:Statements "END" ⟧ =
unfolding
| | evaluate E then
| | | | execute S and then unfold
| | | else complete
| trap handle the exit-exception .

(30) execute ⟦ "WITH" ⟦ I:Identifier "=" E:Expression ⟧ "DO" S:Statements "END" ⟧ =
furthermore
| | investigate E then bind I to the given (value | variable)
| hence execute S .

(31) execute ⟦ "WITH" B_1:Bindings "," B_2:Binding "DO" S:Statements "END" ⟧ =
execute ⟦ "WITH" B_1 "DO" ⟦ "WITH" B_2 "DO" S "END" ⟧ "END" ⟧ .

2.4.1 Cases

- select from _ :: (Case | Cases) →
  action [completing | storing | diverging | escaping with an exception | erring]
  [using the given ordinal-value | current bindings | current storage] .

(1) select from ⟨ ⟩ =
fail .

(2) select from ⟦ L:Case-Labels "=>" S:Statements ⟧ =
check selectable from L then
execute S .

(3) select from ⟨ C_1:Case "|" C_2:Cases ⟩ =
select from C_1 or select from C_2 .

- selectable from _ :: (Cases | Case | Case-Labels) →
  yielder [of a truth-value] [using the given ordinal-value | current bindings] .

(4) selectable from ⟨ ⟩ = false .

(5) selectable from ⟦ L:Case-Labels "=>" S:Statements ⟧ = selectable from L .

(6) selectable from ⟨ C_1:Cases "|" C_2:Case ⟩ =
either (selectable from C_1, selectable from C_2) .

(7) selectable from E:Constant-Expression =
(the given ordinal-value is the value of E) .

(8) selectable from ⟦ E_1:Constant-Expression ".."
E_2:Constant-Expression ⟧ =
not either (
the given ordinal-value is less than the value of E_1,
the given ordinal-value is greater than the value of E_2) .

(9) selectable from ⟨ L_1:Case-Label "," L_2:Case-Labels ⟩ =
either (selectable from L_1, selectable from L_2) .

2.4.2 Type-Cases

- type-select from _ :: (Type-Case | Type-Cases) →
  action [completing | storing | diverging | escaping with an exception | erring]
  [using the given reference-value | current bindings | current storage] .

(1) type-select from ⟨ ⟩ =
fail .

(2) type-select from ⟦ T:Types "=>" S:Statements ⟧ =
check type-selectable from T and then
execute S .

(3) type-select from ⟦ T:Type "(" I:Identifier ")" "=>" S:Statements ⟧ =
check type-selectable from T and then
allocate a variable of the type denoted by T
initialized with the given value then
furthermore bind I to the given variable
hence
execute S and then terminally
dispose of the given variable .

(4) type-select from ⟨ C_1:Type-Case "|" C_2:Type-Cases ⟩ =
give type-selectable from C_1 then
type-select from C_1 else type-select from C_2 .

Note use of '_ else _', rather than '_ or _'.

- type-selectable from _ :: (Type-Cases | Type-Case | Types) →
  yielder [of a truth-value] [using the given reference-value | current bindings] .

(5) type-selectable from ⟨ ⟩ = false .

(6) type-selectable from ⟦ T:Types "=>" S:Statements ⟧ =
type-selectable from T .

(7) type-selectable from ⟦ T:Type "(" I:Identifier ")" "=>" S:Statements ⟧ =
type-selectable from T .

(8) type-selectable from ⟨ C_1:Type-Case "|" C_2:Type-Cases ⟩ =
either (type-selectable from C_1, type-selectable from C_2) .

(9) type-selectable from T:Type =
the type possessed by the given reference-value is the type denoted by T .

(10) type-selectable from ⟨ T_1:Type "," T_2:Types ⟩ =
either (type-selectable from T_1, type-selectable from T_2) .

2.4.3 Handlers

- handle in _ :: (Handlers | Handler) →
  action [completing | storing | diverging | escaping with an exception | erring]
  [using the given exception | current bindings | current storage] .

(1) handle in $\langle\ \rangle$ =
  fail .

(2) handle in $[\![$ I:Qualified-Identifiers "=>" S:Statements $]\!]$ =
  check possible to handle in I and then
  execute S .

(3) handle in $[\![$ I:Qualified-Identifier "(" I_1:Identifier ")" "=>" S:Statements $]\!]$ =
  check possible to handle in I and then
  | furthermore
  | | allocate a variable of ???
  | | initialized with the component of the given exception then
  | | bind I_1 to the given variable#1
  | hence execute S .

BUG: incomplete.

(4) handle in $\langle$ H_1:Handlers "|" H_2:Handler $\rangle$ =
  handle in H_1 or handle in H_2 .

- possible to handle in _ :: (Handlers | Handler | Qualified-Identifiers) $\rightarrow$
  yielder [of a truth-value]
    [using the given exception | current bindings] .

(5) possible to handle in $\langle\ \rangle$ =
  false .

(6) possible to handle in $[\![$ I:Qualified-Identifiers "=>" S:Statements $]\!]$ =
  possible to handle in I .

(7) possible to handle in $[\![$ I:Qualified-Identifier "(" I_1:Identifier ")"
    "=>" S:Statements $]\!]$ =
  possible to handle in I .

(8) possible to handle in $\langle$ H_1:Handlers "|" H_2:Handler $\rangle$ =
  either (possible to handle in H_1, possible to handle in H_2) .

(9) possible to handle in I:Qualified-Identifier =
  the given exception matches the entity identified by I .

(10) possible to handle in $\langle$ I_1:Qualified-Identifiers "," I_2:Qualified-Identifier $\rangle$ =
  either (possible to handle in I_1, possible to handle in I_2) .

2.5 Blocks and Declarations

needs: **Identifiers**, **Expressions**, **Statements**, **Types**.

introduces: activate _ , elaborate _ .

privately introduces: pre-elaborate _ , re-elaborate _ , initialize _ , relinquish _ ,
constant-elaborate _ ,
variable-elaborate _ , variable-initialize _ ,
variable-relinquish _ ,
type-pre-elaborate _ , type-re-elaborate _ ,
exception-elaborate _ ,
procedure-pre-elaborate _ , procedure-re-elaborate _ ,
the signature denoted by _ ,
the formal-parameters denoted by _ ,
the formal-mode denoted by _ , the raises denoted by _ .

- activate _ :: Block →
  action [completing | storing | diverging | escaping with an exception | erring]
  [using current bindings | current storage] .

(1) activate ⟦ D:Declaration* "BEGIN" S:Statements "END" ⟧ =
furthermore elaborate D
hence
| execute S and then terminally
| relinquish D .

- elaborate _ :: Declaration* →
  action [binding | redirecting | storing | diverging |
  escaping with an exception | erring]
  [using current bindings | current storage] .

(2) elaborate D:Declaration* =
pre-elaborate D before
| re-elaborate D and then initialize D .

- pre-elaborate _ :: Declaration* →
  action [binding | storing]
  [using current bindings | current storage] .

(3) pre-elaborate ⟨ ⟩ =
complete .

(4) pre-elaborate ⟦ "CONST" C:⟨ Constant-Declaration ";" ⟩* ⟧ =
constant-elaborate C .

(5) pre-elaborate ⟦ "VAR" V:⟨ Variable-Declaration ";" ⟩* ⟧ =
variable-elaborate V .

(6) pre-elaborate ⟦ "TYPE" T:⟨ Type-Declaration ";" ⟩* ⟧ =
type-pre-elaborate T .

(7) pre-elaborate ⟦ "REVEAL" R:⟨ Reveal-Declaration ";" ⟩* ⟧ =
complete .

(8) pre-elaborate ⟦ "EXCEPTION" E:⟨ Exception-Declaration ";" ⟩* ⟧ =
exception-elaborate E .

(9) pre-elaborate ⟦ P:Procedure-Declaration ";" ⟧ =
procedure-pre-elaborate P .

(10) pre-elaborate ⟨ D_1:Declaration D_2:Declaration* ⟩ =
pre-elaborate D_1 and pre-elaborate D_2 .

- re-elaborate _ :: Declaration* →
  action [redirecting]
  [using current bindings] .

(11) D: ⟨ ⟩ |
⟦ "CONST" ⟨ Constant-Declaration ";" ⟩* ⟧ |
⟦ "VAR" ⟨ Variable-Declaration ";" ⟩* ⟧ |
⟦ "REVEAL" ⟨ Reveal-Declaration ";" ⟩* ⟧ |
⟦ "EXCEPTION" ⟨ Exception-Declaration ";" ⟩* ⟧
⇒
re-elaborate D =
complete .

(12) re-elaborate ⟦ "TYPE" T:⟨ Type-Declaration ";" ⟩* ⟧ =
type-re-elaborate T .

(13) re-elaborate ⟦ P:Procedure-Declaration ";" ⟧ =
procedure-re-elaborate P .

(14) re-elaborate ⟨ D_1:Declaration D_2:Declaration* ⟩ =
re-elaborate D_1 and re-elaborate D_2 .

- initialize _ :: Declaration* →
  action [storing | diverging | escaping with an exception | erring]
  [using current bindings | current storage] .

(15) D: ⟨ ⟩ |
⟦ "CONST" ⟨ Constant-Declaration ";" ⟩* ⟧ |
⟦ "TYPE" ⟨ Type-Declaration ";" ⟩* ⟧ |
⟦ "REVEAL" ⟨ Reveal-Declaration ";" ⟩* ⟧ |
⟦ "EXCEPTION" ⟨ Exception-Declaration ";" ⟩* ⟧ |
⟦ Procedure-Declaration ";" ⟧
⇒
initialize D =
complete .

(16) initialize ⟦ "VAR" V:⟨ Variable-Declaration ";" ⟩* ⟧ =
variable-initialize V .

(17) initialize ⟨ D_1:Declaration D_2:Declaration* ⟩ =
initialize D_1 and then initialize D_2 .

- relinquish _ :: Declaration* →
  action [storing] [using current bindings | current storage] .

(18) D: ⟨ ⟩ |
⟦ "CONST" ⟨ Constant-Declaration ";" ⟩* ⟧ |
⟦ "TYPE" ⟨ Type-Declaration ";" ⟩* ⟧ |
⟦ "REVEAL" ⟨ Reveal-Declaration ";" ⟩* ⟧ |
⟦ "EXCEPTION" ⟨ Exception-Declaration ";" ⟩* ⟧ |
⟦ Procedure-Declaration ";" ⟧
⇒
relinquish D =
complete .

(19) relinquish ⟦ "VAR" V:⟨ Variable-Declaration ";" ⟩* ⟧ =
variable-relinquish V .

(20) relinquish $\langle$ D_1:Declaration D_2:Declaration* $\rangle$ =
relinquish D_1 and relinquish D_2 .

2.5.1 Constant-Declarations

- constant-elaborate _ ::
(Constant-Declaration | $\langle$ Constant-Declaration ";" $\rangle^*$ $\rightarrow$
action [binding] [using current bindings] .

(1) constant-elaborate $[\![$ I:Identifier T:$\langle$ ":" Type $\rangle^?$
"=" E:Constant-Expression $]\!]$ =
bind I to the value of E .

(2) constant-elaborate $\langle$ $\rangle$ =
complete .

(3) constant-elaborate $\langle$ C_1:Constant-Declaration ";"
C_2:$\langle$ Constant-Declaration ";" $\rangle^*$ $\rangle$ =
constant-elaborate C_1 and constant-elaborate C_2 .

2.5.2 Variable-Declarations

- variable-elaborate _ ::
(Variable-Declaration | $\langle$ Variable-Declaration ";" $\rangle^*$) $\rightarrow$
action [binding | storing]
[using current bindings | current storage] .

(1) variable-elaborate $[\![$ I:Identifiers ":" T:Type E:$\langle$ ":=" Expression $\rangle^?$ $]\!]$ =
respectively allocate variables of
(the type denoted by T)$^{(\text{the count of the identifiers of } I)}$ then
respectively bind the identifiers of I to the given variable$^+$.

(2) variable-elaborate $[\![$ I:Identifiers ":=" E:Expression $]\!]$ =
respectively allocate variables of
(the type of E)$^{(\text{the count of the identifiers of } I)}$ then
respectively bind the identifiers of I to the given variable$^+$.

(3) variable-elaborate $\langle$ $\rangle$ =
complete .

(4) variable-elaborate $\langle$ V_1:Variable-Declaration ";"
V_2:$\langle$ Variable-Declaration ";" $\rangle^*$ $\rangle$ =
variable-elaborate V_1 and variable-elaborate V_2 .

- variable-initialize _ ::
(Variable-Declaration | $\langle$ Variable-Declaration ";" $\rangle^*$) $\rightarrow$
action [storing | diverging | escaping with an exception | erring]
[using current bindings | current storage] .

(5) variable-initialize $[\![$ I:Identifiers ":" T:Type $]\!]$ =
complete .

(6) variable-initialize $[\![$ I:Identifier T:$\langle$ ":" Type $\rangle^?$ ":=" E:Expression $]\!]$ =
evaluate E then
assign the given value to the variable bound to I .

(7) variable-initialize $[\![$ I_1:Identifier "," I_2:Identifiers T:$\langle$ ":" Type $\rangle^?$
":=" E:Expression $]\!]$ =
variable-initialize $[\![$ I_1 T ":=" E $]\!]$ and then
variable-initialize $[\![$ I_2 T ":=" E $]\!]$.

(8) variable-initialize $\langle\ \rangle$ =
complete .

(9) variable-initialize $\langle$ V_1:Variable-Declaration ";"
V_2:$\langle$ Variable-Declaration ";" $\rangle^*$ $\rangle$ =
variable-initialize V_1 and then variable-initialize V_2 .

- variable-relinquish _ ::
  (Variable-Declaration | $\langle$ Variable-Declaration ";" $\rangle^*$) $\rightarrow$
  action [storing] [using current bindings | current storage] .

(10) variable-relinquish $[\![$ I:Identifiers ":" T:Type $]\!]$ =
dispose of all the variables respectively bound to the identifiers of I .

(11) variable-relinquish $[\![$ I:Identifiers T:$\langle$ ":" Type $\rangle^?$ ":=" E:Expression $]\!]$ =
dispose of all the variables respectively bound to the identifiers of I .

(12) variable-relinquish $\langle\ \rangle$ =
complete .

(13) variable-relinquish $\langle$ V_1:Variable-Declaration ";"
V_2:$\langle$ Variable-Declaration ";" $\rangle^*$ $\rangle$ =
variable-relinquish V_1 and variable-relinquish V_2 .

2.5.3 Type-Declarations

- type-pre-elaborate _ ::
  (Type-Declaration | $\langle$ Type-Declaration ";" $\rangle^*$ $\rightarrow$
  action [binding] .

(1) type-pre-elaborate $[\![$ I:Identifier ("=" | "<:") T:Type $]\!]$ =
indirectly bind I to unknown .

(2) type-pre-elaborate $\langle\ \rangle$ =
complete .

(3) type-pre-elaborate $\langle$ T_1:Type-Declaration ";"
T_2:$\langle$ Type-Declaration ";" $\rangle^*$ $\rangle$ =
type-pre-elaborate T_1 and type-pre-elaborate T_2 .

- type-re-elaborate _ ::
  (Type-Declaration | $\langle$ Type-Declaration ";" $\rangle^*$ $\rightarrow$
  action [redirecting] [using current bindings] .

(4) type-re-elaborate $[\![$ I:Identifier "=" T:Type $]\!]$ =
redirect I to the type denoted by T .

(5) type-re-elaborate $[\![$ I:Identifier "<:" T:Type $]\!]$ =
redirect I to the opaque-type of the type denoted by T .

(6) type-re-elaborate $\langle\ \rangle$ =
complete .

(7) type-re-elaborate $\langle$ T_1:Type-Declaration ";"
T_2:$\langle$ Type-Declaration ";" $\rangle^*$ $\rangle$ =
type-re-elaborate T_1 and type-re-elaborate T_2 .

2.5.4 Exception-Declarations

- exception-elaborate _ ::
(Exception-Declaration | $\langle$ Exception-Declaration ";" $\rangle^*$ $\rightarrow$
action [binding | storing] [using current storage] .

(1) exception-elaborate $[\![$ I:Identifier T:$\langle$ "(" Type ")" $\rangle^?$ $]\!]$ =
allocate an exception-name then bind I to it .

(2) exception-elaborate $\langle\ \rangle$ =
complete .

(3) exception-elaborate $\langle$ E_1:Exception-Declaration ";"
E_2:$\langle$ Exception-Declaration ";" $\rangle^*$ $\rangle$ =
exception-elaborate E_1 and exception-elaborate E_2 .

2.5.5 Procedure-Declarations

- procedure-pre-elaborate _ :: Procedure-Declaration $\rightarrow$
action [binding] [using current bindings] .

(1) procedure-pre-elaborate $[\![$ "`PROCEDURE`" I:Identifier S:Signature $]\!]$ =
indirectly bind I to unknown .

(2) procedure-pre-elaborate $[\![$ "`PROCEDURE`" I:Identifier S:Signature
"=" B:Block I':Identifier $]\!]$ =
| check there is the stub-procedure bound to I
or
| check not (there is the stub-procedure bound to I) and then
| indirectly bind I to unknown .

- procedure-re-elaborate _ :: Procedure-Declaration $\rightarrow$
action [redirecting] [using current bindings] .

(3) procedure-re-elaborate $[\![$ "`PROCEDURE`" I:Identifier S:Signature $]\!]$ =
redirect I to the stub-procedure of the signature denoted by S .

(4) procedure-re-elaborate $[\![$ "`PROCEDURE`" I:Identifier S:Signature
"=" B:Block I':Identifier $]\!]$ =
redirect I to the procedure of (the signature denoted by S,
the closure of the abstraction of
| furthermore
| | formally associate all the given actual-parameter*
| hence
| | | activate B and then terminally
| | | dissociate all the formal-parameters of
| | | the signature denoted by S
| | trap handle the return-exception) .

- the signature denoted by _ :: Signature $\rightarrow$
  yielder [of a signature] [using current bindings] .

(5) the signature denoted by $\langle$ "(" F:Formals ")" ":" T:Type R:$\langle$ "`RAISES`" Raises $\rangle^?$ $\rangle$ =
the signature of (the formal-parameters denoted by F,
the type denoted by T, the raises denoted by R) .

(6) the signature denoted by $\langle$ "(" F:Formals ")" R:$\langle$ "`RAISES`" Raises $\rangle^?$ $\rangle$ =
the signature of (the formal-parameters denoted by F,
the raises denoted by R) .

- the formal-parameters denoted by _ :: Formals $\rightarrow$
  yielder [of formal-parameter*]
  [using current bindings] .

(7) the formal-parameters denoted by $\langle\ \rangle$ = () .

(8) the formal-parameters denoted by $[\![$ M:Mode I:Identifiers ":" T:Type $]\!]$ =
the associations of the identifiers of I
to (the formal-mode denoted by M, the type denoted by T) .

(9) the formal-parameters denoted by $[\![$ M:Mode I:Identifiers ":" T:Type
":=" E:Constant-Expression $]\!]$ =
the associations of the identifiers of I
to (the formal-mode denoted by M, the type denoted by T,
the value of E) .

(10) the formal-parameters denoted by $[\![$ M:Mode I:Identifiers
":=" E:Constant-Expression $]\!]$ =
the associations of the identifiers of I
to (the formal-mode denoted by M, the type of E, the value of E) .

(11) the formal-parameters denoted by $\langle$ F_1:Formal ";"
F_2:$\langle$ Formal $\langle$ ";" Formal $\rangle^*$ $\rangle$ $\rangle$ =
(the formal-parameters denoted by F_1, the formal-parameters denoted by F_2)
.

(12) the formal-parameters denoted by $\langle$ F:$\langle$ Formal $\langle$ ";" Formal $\rangle^*$ $\rangle$ ";" $\rangle$ =
the formal-parameters denoted by F .

- the formal-mode denoted by _ :: Mode $\rightarrow$ parameter-mode .

(13) the formal-mode denoted by "`VALUE`" = the value-mode .

(14) the formal-mode denoted by "`VAR`" = the variable-mode .

(15) the formal-mode denoted by "`READONLY`" = the read-only-mode .

- the raises denoted by _ :: $\langle$ "`RAISES`" Raises $\rangle^?$ $\rightarrow$
  yielder [of a raise-set] [using current bindings] .

(16) the raises denoted by $[\![$ "`RAISES`" R:Raises $]\!]$ =
the raises denoted by R .

(17) the raises denoted by () = the empty-set .

- the raises denoted by _ :: Raises $\rightarrow$
  yielder [of a raise-set$^?$] [using current bindings] .

(18) the raises denoted by $[\![$ "{" I:Qualified-Identifiers$^?$ "}" $]\!]$ = ??? .

... not yet defined.

(19) the raises denoted by $[\![$ "`ANY`" $]\!]$ = () .

2.6 Types

needs: Identifiers, Expressions, Blocks and Declarations.

introduces: the type denoted by _ .

privately introduces: the brand of _ ,
the field-types denoted by _ ,
the method-types denoted by _ .

- the type denoted by _ :: Type $\rightarrow$
  yielder [of a type] [using current bindings] .

(1) the type denoted by I:Qualified-Identifier =
the type yielded by the entity bound to I .

(2) the type denoted by $[\![$ "`ROOT`" $]\!]$ =
the traced-root-object-type .

(3) the type denoted by $[\![$ "`UNTRACED`" "`ROOT`" $]\!]$ =
the untraced-root-object-type .

(4) the type denoted by $[\![$ "`ARRAY`" "`OF`" T:Type $]\!]$ =
the open-array-type of the type denoted by T .

(5) the type denoted by $[\![$ "`ARRAY`" T_1:Type "`OF`" T_2:Type $]\!]$ =
the fixed-array-type of (the type denoted by T_1,
the type denoted by T_2) .

(6) the type denoted by $[\![$ "`ARRAY`" T_1:Type "," T_2:Types "`OF`" T_3:Type $]\!]$ =
the type denoted by $[\![$ "`ARRAY`" T_1 "`OF`" $[\![$ "`ARRAY`" T_2 "`OF`" T_3 $]\!]$ $]\!]$.

(7) the type denoted by $[\![$ "`BITS`" E:Constant-Expression "`FOR`" T:Type $]\!]$ =
the packed-type of (the value of E, the type denoted by T) .

(8) the type denoted by $[\![$ "{" Identifiers$^?$ "}" $]\!]$ =
the enumeration-type of the identifiers of I .

(9) the type denoted by $[\![$ "`PROCEDURE`" S:Signature $]\!]$ =
the procedure-type of the signature denoted by S .

(10) the type denoted by $[\![$ "`RECORD`" F:Fields "`END`" $]\!]$ =
the record of the field-types denoted by F .

(11) the type denoted by $[\![$ B:Brand$^?$ "`REF`" T:Type $]\!]$ =
the untraced-reference-type of (the type denoted by T, the brand of B) .

(12) the type denoted by $[\![$ "`UNTRACED`" B:Brand$^?$ "`REF`" T:Type $]\!]$ =
the traced-reference-type of (the type denoted by T, the brand of B) .

(13) the type denoted by ⟦ "`SET`" "`OF`" T:Type ⟧ =
the set-type of the type denoted by T .

(14) the type denoted by ⟦ "`[`" E_1:Constant-Expression "`..`"
E_2:Constant-Expression "`]`" ⟧ =
the subrange-type of (the value of E_1, the value of E_2) .

(15) the type denoted by ⟦ "`(`" T:Type "`)`" ⟧ =
the type denoted by T .

(16) the type denoted by ⟦ T:(Type-Name | Object-Type)$^?$ B:Brand$^?$
"`OBJECT`" F:Fields M:⟨ "`METHODS`" Methods ⟩$^?$
O:⟨ "`OVERRIDES`" Overrides ⟩$^?$ "`END`" ⟧ =
the object-type of (
the record of (the field-types denoted by F,
the method-types denoted by M),
the type denoted by T,
the brand of B) .

- the brand of _ :: Brand$^?$ →
yielder [of a brand$^?$] [using current bindings] .

(17) the brand of ⟦ "`BRANDED`" E:Constant-Expression ⟧ =
the value of E .

(18) the brand of ⟦ "`BRANDED`" ⟧ =
anonymous-brand unique in the current bindings .

BUG: This can't be guaranteed to be a unique brand.

(19) the brand of () = () .

- the field-types denoted by _ :: Fields →
yielder [of associations[(type, value$^?$)]] [using current bindings] .

(20) the field-types denoted by ⟨ ⟩ = () .

(21) the field-types denoted by ⟦ I:Identifiers "`:`" T:Type ⟧ =
the associations of the identifiers of I to the type denoted by T .

(22) the field-types denoted by ⟦ I:Identifiers "`:`" T:Type "`:=`"
E:Constant-Expression ⟧ =
the associations of the identifiers of I to
(the type denoted by T, the value of E) .

(23) the field-types denoted by ⟦ I:Identifiers "`:=`" E:Constant-Expression ⟧ =
the associations of the identifiers of I to
(the type of E, the value of E) .

(24) the field-types denoted by ⟨ F_1:Field "`;`" F_2:⟨ Field ⟨ "`;`" Field ⟩* ⟩ ⟩ =
(the field-types denoted by F_1, the field-types denoted by F_2) .

(25) the field-types denoted by ⟨ F:⟨ Field ⟨ "`;`" Field ⟩* ⟩ "`;`" ⟩ =
the field-types denoted by F .

- the method-types denoted by _ ::
  (Method | Methods | ⟨ "METHODS" Methods ⟩$^?$) →
  yielder [of associations[procedure-type]] [using current bindings] .

(26) the method-types denoted by ⟨ ⟩ = ( ) .

(27) the method-types denoted by ⟦ I:Identifier S:Signature ⟧ =
the association of I to the method of (the signature denoted by S, nil) .

(28) the method-types denoted by ⟦ I:Identifier S:Signature
":=" E:Constant-Expression ⟧ =
the association of I to the method of the value of E .

(29) the method-types denoted by ⟨ M_1:Method ";"
M_2:⟨ Method ⟨ ";" Method ⟩* ⟩ ⟩ =
(the method-types denoted by M_1, the method-types denoted by M_2) .

(30) the method-types denoted by ⟨ M:⟨ Method ⟨ ";" Method ⟩* ⟩ ";" ⟩ =
the method-types denoted by M .

(31) the method-types denoted by ⟨ "METHODS" M:Methods ⟩ =
the method-types denoted by M .

2.7 Compilations

needs: **Identifiers, Blocks and Declarations.**

introduces: elaborate _ , activate _ .

privately introduces: import _ , expose _ .

- elaborate _ :: Compilation →
  action [binding | storing | diverging | escaping with an exception | erring]
  [using current bindings | current storage] .

(1) elaborate ⟦ "UNSAFE"$^?$ I:Interface ⟧ =
elaborate I .

(2) elaborate ⟦ "UNSAFE"$^?$ M:Module ⟧ =
activate M .

(3) elaborate G:Generic-Interface =
elaborate G .

(4) elaborate G:Generic-Module =
activate G .

- elaborate _ :: Interface →
  action [binding] [using current bindings] .

(5) elaborate ⟦ "INTERFACE" I:Identifier ";"
Y:Imports D:Declaration*
"END" I':Identifier "." ⟧ =
import Y hence
elaborate D hence
bind I to the interface of the current bindings .

(6) elaborate ⟦ "`INTERFACE`" I:Identifier "="
I_1:Identifier "(" I_2:Identifiers$^?$ ")"
"`END`" I':Identifier "." ⟧ =
???

- activate _ :: Module →
  action [completing | storing | diverging | escaping with an exception | erring]
  [using current bindings | current storage] .

(7) activate ⟦ "`MODULE`" I:Identifier "`EXPORTS`" I_1:Identifiers ";"
Y:Imports B:Block I':Identifier "." ⟧ =
| expose components of I_1 and
| import Y
hence activate B .

(8) activate ⟦ "`MODULE`" I:Identifier ";"
Y:Imports B:Block I':Identifier "." ⟧ =
| expose components of I and
| import Y
hence activate B .

(9) activate ⟦ "`MODULE`" I:Identifier ";"
Y:Imports B:Block I':Identifier "." ⟧ =
activate ⟦ "`MODULE`" I "`EXPORTS`" I ";" Y B I' "." ⟧ .

The above item is redundant.

(10) activate ⟦ "`MODULE`" I:Identifier X:⟨ "`EXPORTS`" Identifiers ⟩$^?$ "="
I_1:Identifier "(" I_2:Identifiers$^?$ ")"
"`END`" I':Identifier "." ⟧ =
???

- elaborate _ :: Generic-Interface → ??? .

(11) elaborate ⟦ "`GENERIC`" "`INTERFACE`" I:Identifier "(" I_1:Identifiers$^?$ ")" ";"
Y:Imports D:Declaration* "`END`" I':Identifier "." ⟧ =
???

- activate _ :: Generic-Module → ??? .

(12) activate ⟦ "`GENERIC`" "`MODULE`" I:Identifier "(" I_1:Identifiers$^?$ ")" ";"
Y:Imports B:Block I':Identifier "." ⟧ =
???

- import _ :: (Imports | Import-Items) →
  action [binding] [using current bindings] .

(13) import ⟨ ⟩ =
complete .

(14) import ⟦ "`IMPORT`" Y:Import-Items ";" ⟧ =
import Y .

(15) import ⟦ "`FROM`" I_1:Identifier "`IMPORT`" I_2:Identifiers ";" ⟧ =
give the interface bound to I_1 then expose I_2 .

(16) import $\langle$ Y_1:Import Y_2:Import$^+$ $\rangle$ =
  import Y_1 and import Y_2 .

(17) import I:Identifier =
  bind I to the interface bound to I .

(18) import $[\![$ I_1:Identifier "AS" I_2:Identifier $]\!]$ =
  bind I_2 to the interface bound to I_1 .

(19) import $\langle$ Y_1:Import-Item "," Y_2:Import-Items $\rangle$ =
  import Y_1 and import Y_2 .

- expose components of _ :: Identifiers $\rightarrow$
  action [binding] [using current bindings] .

(20) expose components of I:Identifier =
  produce the bindings of the interface bound to I .

(21) expose components of $\langle$ I_1:Identifier "," I_2:Identifiers $\rangle$ =
  expose components of I_1 and expose components of I_2 .

- expose _ :: Identifiers $\rightarrow$
  action [binding] [using the given interface] .

(22) expose I:Identifier =
  bind I to the component I of the given interface .

(23) expose $\langle$ I_1:Identifier "," I_2:Identifiers $\rangle$ =
  expose I_1 and expose I_2 .

2.8 Programs

needs: **Compilations**.

introduces: run _ .

privately introduces: successively elaborate _ .

- run _ :: Program $\rightarrow$
  action [binding | storing | diverging | escaping with an exception | erring] .

(1) run C:Compilation$^+$ =
  produce the standard-interface-bindings hence
  successively elaborate C .

- successively elaborate _ :: Compilation$^+$ $\rightarrow$
  action [binding | storing | diverging | escaping with an exception | erring]
    [using current bindings | current storage] .

(2) successively elaborate C:Compilation =
  elaborate C .

(3) elaborate $\langle$ C_1:Compilation C_2:Compilation$^+$ $\rangle$ =
  elaborate C_1 before successively elaborate C_2 .

It is assumed that the compilations are ordered according to the initialisation rule.

Chapter 3

Semantic Entities

needs: **[2]/Data Notation**.

3.1 Data

needs: **Typed Entities**, **Exceptions**, **Errors**,
Associations, **Bindings/Tokens**,
[2]/Action Notation/Imperative/Data.

- datum = value | type | variable |
  mode | signature |
  exception | error | association |
  token | uninitialized .

  BUG: incomplete.

3.2 Typed Entities

3.2.1 Values

needs: **(Primitive Typed Entities, Integers, Reals, Characters, Texts, Arrays, Records)/Values**,
[2]/Action Notation/Imperative/Data.

introduces: value , literal-value ,
putative _ .

- value = primitive-value | array-value | record-value (*disjoint*) .

- literal-value = integer-number | real-number | character | text-value .

- $v \leq$ value $\Rightarrow$
  putative v = v | uninitialized (*disjoint*) .

3.2.2 Types

needs: **(Primitive Typed Entities, Arrays, Records)/Types**,
Errors.

introduces: type .

- type = primitive-type | array-type | record-type (*disjoint*) .

3.2.3 Variables

needs: **Typed Entities, Primitive Typed Entities, Arrays, Records, Errors, [2]/Action Notation.**

introduces: variable ,
_ is _ ,
the value assigned to _ , assign _ to _ ,
uninitialize _ , allocate a variable of _ ,
allocate a variable of _ initialized with _ , dispose of _ ,
the values respectively assigned to _ ,
uninitialize all _ , respectively allocate variables of _ , dispose of all _ .

- variable = primitive-variable | array-variable | record-variable (*disjoint*) .

- _ is _ :: variable, variable → truth-value (*partial, commutative*) .

- the value assigned to _ :: yielder [of a variable] →
  yielder [of a putative value] [using current storage] .

(1) the value assigned to v:yielder [of a variable] =
  the current storage at the cell of the primitive-variable yielded by v |
  the value assigned to the array-variable yielded by v |
  the value assigned to the record-variable yielded by v .

- assign _ to _ :: yielder [of a value], yielder [of a variable] → action [storing] .

(2) assign v_1:yielder [of a value] to v_2:yielder [of a variable] =
  store the primitive-value yielded by v_1 in
    the cell of the primitive-variable yielded by v_2 or
  assign the array-value yielded by v_1 to the array-variable yielded by v_2 or
  assign the record-value yielded by v_1 to the record-variable yielded by v_2 .

- uninitialize _ :: yielder [of a variable] → action [storing] [using current storage] .

(3) uninitialize v:yielder [of a variable] =
  unstore the cell of the primitive-variable yielded by v or
  uninitialize the array-variable yielded by v or
  uninitialize the record-variable yielded by v .

- allocate a variable of _ :: yielder [of a type] →
  action [giving a variable | storing] [using current storage] .

(4) allocate a variable of t:yielder [of a type] =
  | check there is the primitive-type yielded by t then
  | allocate a cell then
  | give the primitive-variable of the given cell
  or allocate a variable of the array-type yielded by t
  or allocate a variable of the record-type yielded by t .

- allocate a variable of _ initialized with _ ::
  yielder [of a type], yielder [of a value] $\rightarrow$
  action [giving a variable | storing] [using current storage] .

(5) allocate a variable of t:yielder [of a type] initialized with v:yielder [of a value] =
| give v and allocate a variable of t
then
| store the given value#1 in the given variable#2 and
| give the given variable#2 .

- dispose of _ :: yielder [of a variable] $\rightarrow$ action [storing] .

(6) dispose of v:yielder [of a variable] =
unreserve the cell of the primitive-variable yielded by v or
dispose of the array-variable yielded by v or
dispose of the record-variable yielded by v .

- the values respectively assigned to _ :: yielder [of variable*] $\rightarrow$
  yielder [of (putative value)*] [using current storage] .

(7) the values respectively assigned to v:yielder [of variable*] =
if v is ()
then ()
else (the value assigned to the first of v,
the values respectively assigned to the rest of v) .

- uninitialize all _ :: yielder [of variable*] $\rightarrow$ action [storing] [using current storage] .

(8) uninitialize all v:yielder [of variable*] =
| check (v is ())
or
| uninitialize the first of v and
| uninitialize all the rest of v .

- respectively allocate variables of _ :: yielder [of type*] $\rightarrow$
  action [giving variable* | storing] [using current storage] .

(9) respectively allocate variables of t:yielder [of type*] =
| check (t is ())
or
| allocate a variable of the first of t and
| respectively allocate variables of the rest of t .

- dispose of all _ :: yielder [of variable*] $\rightarrow$ action [storing] .

(10) dispose of all v:yielder [of variable*] =
| check (v is ())
or
| dispose of the first of v and
| dispose of all the rest of v .

3.3 Primitive Typed Entities

3.3.1 Values

needs: **(Ordinals, Reals, Texts, Sets,
References, Objects, Procedures, Mutexes)/Values**.

introduces: primitive-value .

- primitive-value =
  ordinal-value | real-number | text-value | set-value |
  proper-reference-value | proper-object-value |
  proper-procedure-value | nil | mutex-value (*disjoint*) .

  ... written this way because reference-value, object-value, and procedure-value are not disjoint – their join is nil.

3.3.2 Types

needs: **(Ordinals, Reals, Texts, Sets,
References, Objects, Procedures)/Types**.

introduces: primitive-type .

- primitive-type =
  ordinal-type | real-type | text-type | set-type |
  reference-type | object-type | procedure-type (*disjoint*) .

3.3.3 Variables

needs: **Primitive Typed Entities/Types**,
**(Ordinals, Reals, Texts, Sets,
References, Objects, Procedures)/Variables**,
[2]/Action Notation/Imperative/Data.

introduces: primitive-variable ,
primitive-variable of _ , cell _ .

- primitive-variable of _ :: cell → primitive-variable (*total*, *injective*) .

- cell _ :: primitive-variable → cell (*total*) .

(1) cell (primitive-variable of c:cell) = c .

- _ is _ :: primitive-variable, primitive-variable → truth-value (*total*, *commutative*) .

(2) v_1:primitive-variable is v_2:primitive-variable = cell v_1 is cell v_2 .

- storable = primitive-value .

3.4 Ordinals

3.4.1 Values

needs: **Ordinals/Types**,
(Enumerands, Integers)/Values.

introduces: ordinal-value ,
ordinal-number _ , successor _ , predecessor _ , increase _ , decrease _ ,
_ is _ , _ is less than _ , _ is greater than _ ,
interval-values _ ,
comparable-ordinal-values .

- ordinal-value = enumerated-value | integer-number (*disjoint*) .

  Note that truth-value, character $\leq$ *enumerated-value.*

- comparable-ordinal-values =
  comparable-enumerated-values | integer-number^2 (*disjoint*) .

- ordinal-number _ :: ordinal-value $\rightarrow$ integer-number (*total*) .

- ordinal-val _ :: (integer-number, ordinal-type) $\rightarrow$ ordinal-value (*total*) .

  ... These are defined in Enumerands/Values, Integers/Values, and below.

(1) ordinal-val (i:integer-number, t:subrange-type) =
ordinal-val (i, host-type t) .

- successor _ , predecessor _ :: ordinal-value $\rightarrow$ (ordinal-value | error) (*total*) .

(2) successor (o:ordinal-value) = increase (o, integer-number of 1) .

(3) predecessor (o:ordinal-value) = decrease (o, integer-number of 1) .

- increase _ , decrease _ ::
  (ordinal-value, integer-number) $\rightarrow$ (ordinal-value | error) (*total*) .

(4) sum (ordinal-number o, i) = ordinal-number o' $\Rightarrow$
increase (o:ordinal-value, i:integer-number) = o':ordinal-value .

(5) increase (o:ordinal-value, i:integer-number) = o':ordinal-value $\Rightarrow$
sum (ordinal-number o, i) = ordinal-number o' .

... fully defined in Enumerands/Values and Integers/Values.

(6) decrease (o:ordinal-value, i:integer-number) =
increase (o, negation i) .

- _ is _ :: ordinal-value, ordinal-value $\rightarrow$ truth-value (*partial*, *commutative*) .

- _ is less than _ , _ is greater than _ ::
  ordinal-value, ordinal-value $\rightarrow$ truth-value (*partial*) .

(7) (o_1, o_2) : comparable-ordinal-values $\Rightarrow$

  (1) o_1 is o_2 = ordinal-number o_1 is ordinal-number o_2 ;

  (2) o_1 is less than o_2 = ordinal-number o_1 is less than ordinal-number o_2 ;

  (3) o_1 is greater than o_2 = ordinal-number o_1 is greater than ordinal-number o_2 .

- interval-values _ ::
  comparable-ordinal-values, integer-number$^?$ $\rightarrow$
  ordinal-value* (*partial*) .

(8) (o_1, o_2) : comparable-ordinal-values $\Rightarrow$

(1) interval-values $(o_1, o_2) =$
if o_1 is greater than o_2 then ( )
else $(o_1,$ interval-values (successor $o_1, o_2))$;

(2) interval-values $(o_1, o_2, i$:integer-number$) =$
if o_1 is greater than o_2 then ( )
else $(o_1,$ interval-values (increase $(o_1, i), o_2))$.

3.4.2 Types

needs: **Ordinals/Values**, **Errors**,
(Typed Entities, Enumerands, Integers)/Types.

introduces: ordinal-type , least _ , greatest _ , cardinality _ ,
subrange-type , subrange-type of _ , host-type _ .

- ordinal-type = enumerated-type | integer-type | subrange-type (*disjoint*) .
- least _ , greatest _ :: ordinal-type $\to$ ordinal-value (*partial*) .
- cardinality _ :: ordinal-type $\to$ natural (*total*) .

  ... defined in Enumerands/Types, Integers/Types, and below.
- subrange-type =enumerated-subrange-type | boolean-subrange-type |
  character-subrange-type | integer-subrange-type (*disjoint*) .
- subrange-type of _ :: comparable-ordinal-values $\to$ subrange-type (*total*, *injective*)
  .
- host-type _ :: subrange-type $\to$ (enumerated-type | integer-type) (*total*) .
- least _ , greatest _ :: subrange-type $\to$ ordinal-value (*total*) .
- cardinality _ :: subrange-type $\to$ natural (*total*) .

(1) $t =$ (subrange-type of $(o_1$:ordinal-value, o_2:ordinal-value$))$: subrange-type $\Rightarrow$

(1) least $t = o_1$;

(2) greatest $t = o_2$;

(3) cardinality $t =$
if o_1 is greater than o_2
then 0
else successor (difference (ordinal-number o_2, ordinal-number o_1)) .

(2) host-type t:boolean-subrange-type = boolean-type .

3.5 Enumerands

3.5.1 Values

needs: **(Ordinals, Booleans, Characters, Integers)/Values, Enumerands/Types.**

introduces: enumerated-value , proper-enumerated-value ,
enumerand #_ of _ ,
comparable-enumerated-values .

- enumerated-value =
  proper-enumerated-value | truth-value | character (*disjoint*) .

- enumerand #_ of _ ::
  natural, enumerated-type $\rightarrow$ enumerated-value (*partial*) .

(1) n is less than cardinality t = true $\Rightarrow$
enumerand #n:natural of t:enumerated-type : enumerated-value .

(2) n is less than cardinality t = false $\Rightarrow$
enumerand #n:natural of t:enumerated-type = nothing .

- ordinal-number _ :: enumerated-value $\rightarrow$ integer-number (*total*) .

(3) ordinal-number (enumerand #n:natural of t:enumerated-type) = n .

- increase _ ::
  (enumerated-value, integer-number) $\rightarrow$ (enumerated-value | error) (*total*) .

(4) e = enumerand #n:natural of t:enumerated-type ;
n' = sum (n, i) $\Rightarrow$
increase (e:enumerated-value, integer-number of i:integer) =
if both (not (n' is less than 0), n' is less than cardinality t)
then enumerand #n' of t
else range-error .

- comparable-enumerated-values $\leq$ enumerated-value^2 .

(5) e_1 = enumerand #n_1:natural of t:enumerated-type ;
e_2 = enumerand #n_2:natural of t $\Rightarrow$ (e_1, e_2) : comparable-enumerated-values .

3.5.2 Types

needs: **Enumerands/Values, (Ordinals, Characters)/Types.**

introduces: enumerated-type , proper-enumerated-type ,
enumerated-type of _ ,
enumerand named _ of _ ,
enumerated-subrange-type .

- enumerated-type =
  proper-enumerated-type | boolean-type | character-type (*disjoint*) .

- enumerated-type of _ :: identifier-token* $\rightarrow$ proper-enumerated-type (*partial*) .

(1) distinct i' $\Rightarrow$
enumerated-type of i':identifier-token* : proper-enumerated-type .

- enumerand named _ of _ ::
  identifier-token, proper-enumerated-type $\rightarrow$
  proper-enumerated-value $(partial)$,
  identifier-token, boolean-type $\rightarrow$ truth-value $(partial)$,
  identifier-token, character-type $\rightarrow$ nothing .

(2) $t =$ enumerated-type of i':identifier-token* ;
$i =$ component #(successor n:natural) of i' $\Rightarrow$
enumerand named i:identifier-token of t:enumerated-type $=$
enumerand $\#n$ of t .

- least _ , greatest _ :: enumerated-type $\rightarrow$ enumerated-value $(partial)$.
- cardinality _ :: enumerated-type $\rightarrow$ natural $(total)$.

(3) $t =$ (enumerated-type of i':identifier-token*) : enumerated-type ;
$n =$ count i' $\Rightarrow$

(1) least $t =$ when not (n is 0) then enumerand #0 of t ;

(2) greatest $t =$ when not (n is 0) then enumerand #(predecessor n) of t ;

(3) cardinality $t = \#n$.

- subrange-type of _ :: enumerated-value^2 $\rightarrow$ enumerated-subrange-type $(partial)$.
- host-type _ :: enumerated-subrange-type $\rightarrow$ enumerated-type $(total)$.

(4) $t =$ subrange-type of (e_1, e_2) ;
$e_1 =$ enumerand $\#n_1$:natural of t':enumerated-type ;
$e_2 =$ enumerand $\#n_2$:natural of t' $\Rightarrow$

(1) t : enumerated-subrange-type ;

(2) host-type $t = t'$.

3.6 Booleans

3.6.1 Values

needs: (Ordinals, Enumerands, Integers)/Values.

includes: [2]/Data Notation/Truth-Values .

- truth-value $\leq$ enumerated-value .
- enumerand #_ of _ :: natural, boolean-type $\rightarrow$ truth-value $(partial)$.

(1) enumerand #0 of boolean-type = false .

(2) enumerand #1 of boolean-type = true .

3.6.2 Types

needs: Ordinals/Types.

introduces: boolean-type , boolean-subrange-type .

- boolean-type = enumeration-type of ("FALSE", "TRUE") .
- subrange-type of _ :: truth-value^2 $\rightarrow$ boolean-subrange-type $(total,\ injective)$.
- host-type _ :: boolean-subrange-type $\rightarrow$ boolean-type $(total)$.

3.7 Characters

3.7.1 Values

needs: (Ordinals, Integers)/Values, Errors.

includes: [2]/Data Notation/Characters/ISO-Latin-1 .

BUG: **.../ISO-Latin-1** *doesn't yet exist!*

introduces: greatest-character-code .

- character $\leq$ enumerated-value .
- greatest-character-code : natural[min 255] .
- enumerand #_of _ :: natural[max greatest-character-code], character-type $\rightarrow$ character $(total,\ injective)$.

(1) enumerand #n:natural[max greatest-character-code] of character-type = character of n .

- character of _ :: integer-number $\rightarrow$ (character | error) $(total)$.

(2) character of integer-number of i:integer =
    if either (i is less than 0, i is greater than maximimum-character-code)
    then character-error
    else character of i .

3.7.2 Types

needs: Ordinals/Types, Characters/Values.

introduces: character-type , character-subrange-type .

- character-type : enumeration-type .
- least _ , greatest _ :: character-type $\rightarrow$ character $(total)$.

(1) least character-type = character of 0 .

(2) greatest character-type = character of greatest-character-code .

- cardinality _ :: character-type $\rightarrow$ natural $(total)$.

(3) cardinality character-type = successor (greatest-character-code) .

- subrange-type of _ :: character^2 $\rightarrow$ character-subrange-type $(total,\ injective)$.
- host-type _ :: character-subrange-type $\rightarrow$ character-type $(total)$.

3.8 Integers

3.8.1 Values

needs: **Ordinals/Values**, **Errors**.

introduces: integer-number , integer-number of _ ,
least-integer , greatest-integer ,
negation _ , absolute-value _ , square _ ,
sum _ , difference _ , product _ ,
integer-quotient _ , modulo _ , minimum _ , maximum _ ,
_ is _ , _ is less than _ , _ is greater than _ .

- integer-number of _ :: integer → (integer-number | error) (*total*) .

(1) integer-number of _ ::
integer [min least-integer] [max greatest-integer] →
integer-number (*total*, *injective*) ,
integer [min successor (greatest-integer)] → integer-overflow-error (*total*) ,
integer [max predecessor (least-integer)] → integer-overflow-error (*total*) .

- least-integer , greatest-integer : integer .

(2) least-integer is less than 0 = true .

(3) greatest-integer is greater than 0 = true .

- ordinal-number _ :: integer-number → integer-number (*total*, *injective*) .

(4) ordinal-number i:integer-number = i .

- negation _ , absolute-value _ ::
integer-number → (integer-number | error) (*total*) .

(5) n = (integer-number of i:integer) : integer-number ⇒

(1) negation n = integer-number of negation i ;

(2) absolute-value n =
integer-number of absolute i but integer-overflow-error .

- sum _ , difference _ , product _ ,
integer-quotient _ , modulo _ ,
minimum _ , maximum _ ::
(integer-number, integer-number) → (integer-number | error) (*total*) .

- _ is _ :: integer-number, integer-number → truth-value (*total*, *commutative*) .

- _ is less than _ , _ is greater than _ :: integer-number, integer-number → truth-value (*total*) .

(6) n_1 = (integer-number of i_1:integer) : integer-number ;
n_2 = (integer-number of i_2:integer) : integer-number ⇒

(1) sum (n_1, n_2) = integer-number of sum (i_1, i_2) ;

(2) difference (n_1, n_2) = integer-number of difference (i_1, i_2) ;

(3) product (n_1, n_2) = integer-number of product (i_1, i_2) ;

(4) integer-quotient (n_1, n_2) =
if i_2 is 0 then integer-division-error
else integer-number of integer-quotient (i_1, i_2) ;

(5) modulo $(n_1, n_2) =$
if not $(i_2$ is greater than 0) then modulo-error
else integer-number of modulo (i_1, i_2) ;

(6) minimum $(n_1, n_2) =$
if n_1 is less then n_2 then n_1 else n_2 ;

(7) maximum $(n_1, n_2) =$
if n_1 is greater then n_2 then n_1 else n_2 ;

(8) n_1 is $n_2 = i_1$ is i_2 ;

(9) n_1 is less than $n_2 = i_1$ is less than i_2 ;

(10) n_1 is greater than $n_2 = i_1$ is greater than i_2 .

3.8.2 Types

needs: Integers/Values, (Primitive Typed Entities, Ordinals)/Types.

introduces: integer-type , integer-subrange-type .

- integer-type : ordinal-type .
- least _ , greatest _ :: integer-type → integer-number $(total)$.

(1) least integer-type = integer-number of least-integer .

(2) greatest integer-type = integer-number of greatest-integer .

- cardinality _ :: integer-type → natural $(total)$.

(3) cardinality integer-type =
successor (difference (greatest-integer, least-integer) .

- subrange-type of _ :: integer-number^2 → integer-subrange-type $(total, injective)$.
- host-type _ :: integer-subrange-type → integer-type $(total)$.

3.9 Reals

3.9.1 Values

needs: (Numbers, Integers)/Values, Errors.

introduces: real-number , real-number of _ ,
negation _ , absolute-value _ , square _ ,
sum _ , difference _ , product _ , quotient _ ,
minimum _ , maximum _ ,
_ is _ , _ is less than _ , _ is greater than _ ,
floor _ , ceiling _ , truncated _ , rounded _ , floated _ .

privately introduces: least-rational , greatest-rational , approximated _ .

This specification makes no attempt to reflect the IEEE standard.

(1) real-number of _ ::
rational [min least-rational] [max greatest-rational] →
real-number $(total, injective)$,
rational [min successor (greatest-rational)] → real-overflow-error $(total)$,
rational [max predecessor (least-rational)] → real-overflow-error $(total)$.

(2) real-number of r:rational = real-number of approximated r .

- least-rational , greatest-rational : rational .

(3) least-rational is less than 0 = true .

(4) greatest-rational is greater than 0 = true .

- approximated _ :: rational → rational ($total$) .

(5) approximated (approximated r:rational) = approximated r .

(6) approximated 0 = 0 .

(7) r_1 is greater than r_2 = false ⇒
approximated r_1:rational is greater than approximated r_2:rational = false .

- negation _ , absolute-value _ :: real-number → real-number ($total$) .

(8) n = real-number of r:approximated rational ⇒

 (1) negation n = real-number of negation r ;

 (2) absolute-value n = real-number of absolute-value r .

- sum _ , difference _ , product _ , quotient _ ,
minimum _ , maximum _ ::
(real-number, real-number) → real-number ($total$) .

- _ is _ ::
real-number, real-number → truth-value ($total$, $commutative$) .

- _ is less than _ , _ is greater than _ ::
real-number, real-number → truth-value ($total$) .

(9) n_1 = real-number of r_1:approximated rational ;
n_2 = real-number of r_2:approximated rational ⇒

 (1) sum (n_1, n_2) = real-number of sum (r_1, r_2) ;

 (2) difference (n_1, n_2) = real-number of difference (r_1, r_2) ;

 (3) product (n_1, n_2) = real-number of product (r_1, r_2) ;

 (4) quotient (n_1, n_2) =
 if n_2 is real-number of 0
 then real-division-error
 else real-number of quotient (r_1, r_2) ;

 (5) minimum (n_1, n_2) =
 if n_1 is less then n_2 then n_1 else n_2 ;

 (6) maximum (n_1, n_2) =
 if n_1 is greater then n_2 then n_1 else n_2 ;

 (7) n_1 is n_2 = r_1 is r_2 ;

 (8) n_1 is less than n_2 = r_1 is less than r_2 ;

 (9) n_1 is greater than n_2 = r_1 is greater than r_2 .

- floor _ :: real-number → (integer-number | error) ($total$) .

(10) floor (real-number of r:approximated rational) =
    integer-number of
        if r is less than 0
        then negation (truncation (negation r))
        else truncated r .

- ceiling _ :: real-number $\rightarrow$ (integer-number | error) ($total$) .

(11) ceiling (real-number of r:approximated rational) =
    integer-number of
        if r is less than 0
        then truncation r
        else negation (truncation (negation r)) .

- truncated _ :: real-number $\rightarrow$ (integer-number | error) ($total$) .

(12) truncated (real-number of r:approximated rational) =
    integer-number of truncation r but truncation-error .

- rounded _ :: real-number $\rightarrow$ (integer-number | error) ($total$) .

(13) rounded (real-number of r:approximated rational) =
    integer-number of
        if r is less than 0
        then truncation (difference (r, quotient (1, 2)))
        else truncation (sum (r, quotient (1, 2))) .

- floated _ :: integer-number $\rightarrow$ real-number ($total$) .

(14) n = (integer-number of i:integer) : integer-number $\Rightarrow$
floated n = real-number of i .

3.9.2 Types

needs: **Primitive Typed Entities/Types**,
(Numbers, Integers, Reals)/Values.

introduces: real-type.

- real-type : primitive-type .

3.10 Texts

3.10.1 Values

introduces: text-value , text-value of _ , concatenation _ , _ is _ .

- text-value $\leq$ reference-value .

- text-value of _ :: character* $\rightarrow$ text-value ($total$, $injective$) .

- concatenation _ :: text-value^2 $\rightarrow$ text-value ($total$) .

(1) concatenation (text-value of c_1:character*, text-value of c_2:character*) =
    text-value of (c_1, c_2) .

- _ is _ :: text-value, text-value $\rightarrow$ truth-value ($total$) .

(2) text-value of c_1:character* is text-value of c_2:character* =
    c_1 is c_2 .

3.10.2 Types

introduces: text-type .

- text-type : reference-type .

3.11 Sets

3.11.1 Values

needs: **(Enumerands, Integers)/Values**.

includes: **[2]/Data Notation/Sets**.

introduces: set-value , symmetric-difference _ , _ is properly included in _ .

- set-value =
  set [of enumerated-value] | set [of integer-number] (*disjoint*) .

- symmetric-difference _ :: set-value^2 → set-value (*total*) .

(1) symmetric-difference (s_1:set-value, s_2:set-value) =
difference (union (s_1, s_2), intersection (s_1, s_2)) .

- _ is properly included in _ :: set-value^2 → truth-value (*total*) .

(2) s_1:set-value is properly included in s_2:set-value =
both (s_1 is included in s_2, not (s_1 is s_2)) .

3.11.2 Types

needs: **Sets/Values**, **(Typed Entities, Ordinals)/Types**.

introduces: set-type , set-type of _ , base-type _ .

- set-type of _ :: ordinal-type → set-type (*partial*, *injective*) .

  Note: This partiality allows 'set-type of integer-type' to be rejected.

- base-type _ :: set-type → ordinal-type (*total*) .

(1) base-type (set-type of t:ordinal-type) = t .

3.12 References

3.12.1 Values

needs: **Typed Entities/Variables**, **References/Types**.

introduces: reference-value , proper-reference-value ,
traced-reference-value , untraced-reference-value ,
nil , traced reference to _, untraced reference to _ ,
referent _ , allocated-type _ ,
address , address of _ ,
_ is _ , _ is less than _ , _ is greater than _ .

- reference-value =
  proper-reference-value | nil (*disjoint*) .
- proper-reference-value =
  traced-reference-value | untraced-reference-value (*disjoint*) .
- nil : reference-value .
- traced reference to _ ::
  (variable, traced-reference-type) $\rightarrow$ traced-reference-value (*total*, *injective*) .
- untraced reference to _ :: variable $\rightarrow$ untraced-reference-value (*total*, *injective*) .
- referent _ :: reference-value $\rightarrow$ (variable | error) (*total*) .

(1) referent (nil) = referent-error .

(2) referent (traced reference to (v:variable, t:traced-reference-type) = v .

(3) referent (untraced reference to v:variable) = v .

- allocated-type _ :: traced-reference-value $\rightarrow$ reference-type (*total*) .

(4) allocated-type (traced reference to (v:variable, t:traced-reference-type) = t .

- address $\leq$ natural .
- address _ ::
  variable $\rightarrow$ address (*total*, *injective*) ,
  reference-value $\rightarrow$ address (*total*) .

(5) address r:(traced-reference-value | untraced-reference-value) =
address (referent r) .

BUG: Two (non-global) variables with distinct lifetimes could have the same address. Think harder about 'address _' (and '_ is _' over variables, used below).

- _ is _ :: reference-value, reference-value $\rightarrow$ truth-value (*total*, *commutative*) .

(6) nil is nil = true .

(7) nil is r:(traced-reference-value | untraced-reference-value) = false .

(8) r_1:(traced-reference-value | untraced-reference-value) is
r_2:(traced-reference-value | untraced-reference-value) =
referent r_1 is referent r_2 .

- _ is less than _ , _ is greater than _ :: reference-value, reference-value $\rightarrow$ truth-value (*total*) .

(9) r_1:reference-value is less than r_2:reference-value =
address r_1 is less than address r_2 .

(10) r_1:reference-value is greater than r_2:reference-value =
r_2 is less than r_1 .

3.12.2 Types

needs: **Typed Entities/Types**, **Brands**, **[2]/Action Notation/Directive/Yielders**.

introduces: reference-type , traced-reference-type , untraced-reference-type , traced-reference-type of _ , untraced-reference-type of _ , referent-type _ , brand _ .

- reference-type = traced-reference-type | untraced-reference-type (*disjoint*) .
- traced-reference-type of _ :: (type, brand$^?$) $\rightarrow$ traced-reference-type (*total*) .
- untraced-reference-type of _ :: (type, brand$^?$) $\rightarrow$ untraced-reference-type (*total*) .
- referent-type _ :: yielder [of a reference-type] $\rightarrow$ yielder [of a type] (*total*) .

(1) t' = (traced-reference-type of (t:type, b:brand$^?$)) : traced-reference-type $\Rightarrow$

  (1) referent-type $t' = t$;

  (2) brand $t' = b$.

(2) t' = (untraced-reference-type of (t:type, b:brand$^?$)) : untraced-reference-type $\Rightarrow$

  (1) referent-type $t' = t$;

  (2) brand $t' = b$.

3.13 Objects

3.13.1 Values

needs: **Types**, **Methods**, **Errors**, **Bindings/Tokens**, **Associations**.

BUG: Relationships between object and reference values and types remain to be clarified.

introduces: object-value , proper-object-value , nil , traced reference to _ , referent_ , allocated-type _ , object , object of _ , field _ of _ , method _ of _ , _ is _ .

- object-value = proper-object-value | nil (*disjoint*) .
- nil : object-value .
- traced reference to _ :: (object, object-type) $\rightarrow$ proper-object-value (*total*, *injective*) .

  Doesn't allow for untraced references to objects: do such exist?

- referent _ :: proper-object-value $\rightarrow$ (object | error) (*total*) .

(1) referent (nil) = referent-error .

(2) referent (traced reference to (o:object, t:object-type) = o .

- allocated-type _ :: proper-object-value $\rightarrow$ object-type (*total*) .

(3) allocated-type (traced reference to (o:object, t:object-type)) = t .

- object = object of (associations[variable], associations[procedure]).

- object of _ ::
  (associations[variable], associations[procedure]) → object (*total*, *injective*) .

  Is one of the above redundant?

- field _ of _ :: identifier-token, object → variable (*partial*) .

- method _ of _ :: identifier-token, object → method (*partial*) .

(4) o = (object of (a_1:associations[variable],
a_2:associations[procedure]) : object ⇒

(1) field i:identifier-token of o =
component associated with i in a_1 ;

(2) method i:identifier-token of o =
method of (o, component associated with i in a_2) .

- field _ of _ :: identifier-token, object-value → (variable | error) (*partial*) .

(5) field i:identifier-token of o:object-value =
field i of referent o .

- method _ of _ :: identifier-token, object-value → method .

(6) method i:identifier-token of o:object-value =
method i of referent o .

Above variants of 'field _ of _' and 'method _ of _' seem to be redundant.

- _ is _ :: object-value, object-value → truth-value (*total*, *commutative*) .

(7) nil is nil = true .

(8) nil is o:object-value = false .

(9) o_1:object-value is o_2:object-value =
referent o_1 is referent o_2 .

- _ is _ :: object, object → truth-value (*total*, *commutative*) .

  ... not yet defined.

3.13.2 Types

needs: **Bindings/Tokens**.

introduces: object-type , object-type of _ ,
field _ of _ , procedure _ of _ .

- object-type =
  object-type of (associations[type], associations[procedure]).

- object-type of _ ::
  (associations[type], associations[procedure]) → object-type (*total*, *injective*)
  .

  Is one of the above redundant?

- field _ of _ :: identifier-token, object-type → type (*partial*) .
- procedure _ of _ :: identifier-token, object-type → procedure (*partial*) .

(1) t = (object-type of (a_1:associations[type],
a_2:associations[procedure])) : object-type ⇒

(1) field i:identifier-token of t =
component associated with i in a_1 ;

(2) procedure i:identifier-token of t =
component associated with i in a_2 .

Incomplete

3.13.3 Methods

needs: **Values, Methods, Procedures**.

introduces: method , method of _ ,
object _ , self-identifier _ , signature _ , body-abstraction _ .

- method of _ :: (object, procedure-value) → method (*total*) .
- object _ :: method → object (*total*) .
- self-identifier _ :: method → identifier-token (*partial*) .
- signature _ :: method → signature (*partial*) .
- body-abstraction _ :: method → abstraction (*partial*) .

(1) m = method of (o:object, p:procedure-value); signature p =
signature of (f:formal-parameter$^+$, t:type$^?$, r:raise-set$^?$)
⇒

(1) object $m = o$;

(2) self-identifier m = identifier (first f) ;

(3) signature m = signature of (rest f, t, r) ;

(4) body-abstraction m = body-abstraction p .

3.14 Procedures

3.14.1 Values

needs: **Signatures**,
[2]/Action Notation.

introduces: procedure-value , proper-procedure-value ,
nil , procedure , stub-procedure ,
procedure of _ , stub-procedure of _ ,
signature _ , body-abstraction _ ,
_ is _ .

- procedure-value = proper-procedure-value | nil (*disjoint*) .
- proper-procedure-value = procedure | stub-procedure (*disjoint*) .

- nil : procedure-value .
- procedure of _ :: (signature, abstraction) $\rightarrow$ procedure (*partial*) .

(1) s : signature ;
a : abstraction [giving a value$^?$ | storing | diverging |
escaping with an exception | erring]
[using the given actual-parameter* | current storage]
$\Rightarrow$
procedure of (s, a) : procedure .

- stub-procedure of _ :: signature $\rightarrow$ procedure (*total*) .

  Do stub-procedures really exist?

- signature _ :: procedure-value $\rightarrow$ signature (*total*) .
- body-abstraction _ :: procedure $\rightarrow$ abstraction (*total*) .

(2) p = (procedure of (s:signature, a:abstraction)) : procedure $\Rightarrow$

  (1) signature $p = s$;

  (2) body-abstraction $p = a$.

(3) p = (stub-procedure of s:signature) : stub-procedure $\Rightarrow$
signature $p = s$.

BUG: Above operations should be partial. And what about nil?

- _ is _ :: procedure, procedure $\rightarrow$ truth-value (*total*) .

  Is this to be defined?

3.14.2 Types

needs: **Signatures**.

introduces: procedure-type , procedure-type of _ , signature _ .

- procedure-type of _ :: signature $\rightarrow$ procedure-type (*total*, *injective*) .
- signature _ :: procedure-type $\rightarrow$ signature (*total*) .
- signature _ :: procedure-type $\rightarrow$ signature (*total*) .

(1) signature (procedure-type of s:signature) = s .

3.14.3 Signatures

needs: **Typed Entities/Types**, **Exceptions**, **Formal Parameters**.

introduces: signature , signature of _ ,
raise-set , formal-parameters _ , result-type of _ , raises _ .

- signature of _ :: (formal-parameter*, type$^?$, raise-set$^?$) $\rightarrow$ signature (*total*, *injective*) .

  Is the raise-set really necessary?

- raise-set = set [of exception-name] .

- formal-parameters _ :: signature $\rightarrow$ formal-parameter* (*total*) .
- result-type _ :: signature $\rightarrow$ type$^?$ (*total*) .

  'result-type s' = () means that the procedure is a function.
- raises _ :: signature $\rightarrow$ raise-set$^?$ (*total*) .

  raises s = () means that the procedure can raise any exception.

(1) s = (signature of (f:formal-parameter*, t:type$^?$, r:raise-set$^?$)
: signature $\Rightarrow$

(1) formal-parameters $s = f$;

(2) result-type $s = t$;

(3) raises $s = r$.

3.14.4 Formal Parameters

needs: **Typed Entities/Types**, **Parameter Modes**, **Bindings/Tokens**.

introduces: formal-parameter ,
value-parameter , variable-parameter , read-only-parameter ,
formal-mode , value-mode , variable-mode , read-only-mode .

- formal-parameter $\leq$ association[(formal-mode, type, value$^?$)] .
- formal-parameter = value-parameter | variable-parameter | read-only-parameter (*disjoint*) .
- value-parameter = association[(value-mode, type, value$^?$)] .
- variable-parameter = association[(variable-mode, type)] .
- read-only-parameter = association[(read-only-mode, type, value$^?$)] .
- formal-mode = value-mode | variable-mode | read-only-mode (*individual*) .

3.14.5 Actual Parameters

needs: **Typed Entities**, **Formal Parameters**.

introduces: actual-parameter , argument .

- actual-parameter = association[(formal-mode,argument)] | type .
- argument = value | variable (*disjoint*) .

3.14.6 Parameter Association

needs: **Typed Entities**, **Actual Parameters**, **Formal Parameters**,
Bindings/Tokens,
[2]/Action Notation.

introduces: formally associate _ , formally associate all _ ,
dissociate _ , dissociate all _ .

- formally associate _ ::
    yielder [of an actual-parameter] $\rightarrow$
        action [binding | storing] .

(1) formally associate a:yielder [of an actual-parameter] =
    give the (identifier-token, formal-mode, argument) yielded by
        (the identifier of a, the component of a) then
    | | check there is the given value-mode#2 and then
    | | | | give the given identifier-token#1 and
    | | | | give the given value#3 and
    | | | | allocate a variable of the given type#4
    | | | then
    | | | | bind the given identifier-token#1 to the given variable#3 and
    | | | | assign the given value#2 to the given variable#3
    | or
    | | check there is the given variable-mode#2 and then
    | | bind the given identifier-token#1 to the given variable#3
    | or
    | | check there is the given read-only-mode#2 and then
    | | bind the given identifier-token#1 to the given value#3 .

Ugh! Try to make this more lucid!

- formally associate all _ ::
    yielder [of actual-parameter*] $\rightarrow$
        action [binding | storing] .

(2) formally associate all a:yielder [of actual-parameter*] =
    | check (a is ())
    or
    | formally associate the first of a and
    | formally associate all the rest of a .

- dissociate _ :: yielder [of a formal-parameter] $\rightarrow$ action [storing] .

(3) dissociate f:yielder [of a formal-parameter] =
    give the (identifier-token, formal-mode, type, value?) yielded by
        (the identifier of f, the component of f) then
    | | check there is the given value-mode#2 and then
    | | deallocate the variable bound to the given identifier-token#1
    | or check there is the given variable-mode#2
    | or check there is the given read-only-mode#2 .

- dissociate all _ :: yielder [of formal-parameter*] $\rightarrow$ action [storing] .

(4) dissociate all f:yielder [of formal-parameter*] =
    | check (f is ())
    or
    | dissociate the first of f and
    | dissociate all the rest of f .

3.15 Mutexes

3.15.1 Values

introduces: mutex-value ,
locked , unlocked .

- mutex-value = locked | unlocked (*individual*) .

3.16 Arrays

3.16.1 Values

needs: Typed Entities/Values, **Ordinals/(Values, Types)**.

introduces: array-value , array of _ ,
elements _ , index-type _ , lower-bound _ , upper-bound _ , length _ ,
element indexed by _ in _ , _ is _ .

Note duplication of operations in Arrays/Values and Arrays/Variables.

- array of _ ::
  (value*, ordinal-type) → array-value (*partial*) ,
  ((putative value)*, ordinal-type) → putative array-value (*partial*) .

(1) the array of (v_1:(putative value)*, v_2:uninitialized, v_3:(putative value)*,
t:ordinal-type) =
uninitialized .

- elements _ :: array-value → value* (*total*) .
- index-type _ :: array-value → ordinal-type (*total*) .
- lower-bound _ , upper-bound _ :: array-value → ordinal-value (*total*) .
- length _ :: array-value → natural (*total*) .

(2) a = (array of (v:value*, t:ordinal-type)) : array-value ⇒

(1) elements a = v ;
(2) index-type a = t ;
(3) lower-bound a = least t ;
(4) upper-bound a = greatest t ;
(5) length a = cardinality t ;
(6) length a = count v .

- element indexed by _ in _ :: ordinal-value, array-value → (value | error) (*partial*) .

(3) integer-number of m = ordinal-number (lower-bound a) ;
integer-number of n = ordinal-number (upper-bound a) ;
integer-number of i = ordinal-number o ;
k = successor (difference (i, m))
⇒
element indexed by o:ordinal-value in a:array-value =
if either (i is less than m, i is greater than n)
then array-indexing-error
else component $\#k$ (elements a) .

- _ is _ :: array-value, array-value → truth-value (*partial*) .

(4) a_1:array-value is a_2:array-value =
elements a_1 is elements a_2 .

3.16.2 Types

needs: (Typed Entities, Ordinals)/Types.

introduces: array-type , fixed-array-type , open-array-type ,
fixed-array-type of _ , open-array-type of _ ,
element-type _ , index-type _ , length _ .

- array-type = fixed-array-type | open-array-type (*disjoint*) .
- fixed-array-type of _ :: (type, ordinal-type) → fixed-array-type (*total*, *injective*) .
- open-array-type of _ :: type → open-array-type (*total*, *injective*) .

- element-type _ :: array-type → type (*total*) .

(1) element-type (fixed-array-type of (t:type, t':ordinal-type)) = t .

(2) element-type (open-array-type of t:type) = t .

- index-type _ :: array-type → ordinal-type (*total*) .

(3) index-type (fixed-array-type of (t:type, t':ordinal-type)) = t' .

(4) index-type (open-array-type of t:type) = integer-type .

- length _ :: fixed-array-type → natural (*total*) .

(5) length t:fixed-array-type = cardinality (index-type t) .

3.16.3 Variables

needs: (Typed Entities, Arrays, Ordinals, Booleans)/Values, Arrays/Types, Typed Entities/Variables, Errors, [2]/Action Notation.

introduces: array-variable , array of _ ,
elements _ , index-type _ , lower-bound _ , upper-bound _ , length _ ,
element indexed by _ in _ , _ is _ ,
the value assigned to _ , assign _ to _ ,
uninitialize _ , allocate a variable of _ , dispose of _ .

- array of _ :: (variable*, ordinal-type) → array-variable (*partial*) .

- elements _ :: array-variable → variable* (*total*) .
- index-type _ :: array-variable → ordinal-type (*total*) .
- lower-bound _ , upper-bound _ :: array-variable → ordinal-value (*total*) .
- length _ :: array-variable → natural (*total*) .

(1) a = (array of (v:variable*, t:ordinal-type)) : array-variable $\Rightarrow$

(1) elements $a = v$;

(2) index-type $a = t$;

(3) lower-bound a = least t ;

(4) upper-bound a = greatest t ;

(5) length a = cardinality t ;

(6) length a = count v .

- element indexed by _ in _ :: ordinal-value, array-variable → (variable | error) (*partial*) .

(2) integer-number of m = ordinal-number (lower-bound a) ;
integer-number of n = ordinal-number (upper-bound a) ;
integer-number of i = ordinal-number o ;
k = successor (difference (i, m))
$\Rightarrow$
element indexed by o:ordinal-value in a:array-variable =
if either (i is less than m, i is greater than n)
then array-indexing-error
else component #k (elements a) .

- _ is _ :: array-variable, array-variable → truth-value (*partial*, *commutative*) .

(3) a_1:array-variable is a_2:array-variable =
first (elements a_1) is first (elements a_2) .

BUG: Crude, and doesn't work if a_1 or a_2 is empty.

- the value assigned to _ :: yielder [of an array-variable] →
yielder [of a putative array-value] [using current storage] .

(4) the value assigned to a:yielder [of an array-variable] =
the array of (
the values respectively assigned to the elements of a,
the index-type of a) .

- assign _ to _ :: yielder [of an array-value], yielder [of an array-variable] → action [storing] .

(5) assign a_1:yielder [of an array-value] to a_2:yielder [of an array-variable] =
respectively assign the elements of a_1 to the elements of a_2 .

- uninitialize _ :: yielder [of an array-variable] → action [storing] [using current storage] .

(6) uninitialize a:yielder [of an array-variable] =
uninitialize all the elements of a .

- allocate a variable of _ :: yielder [of a fixed-array-type] →
action [giving an array-variable | storing] [using current storage] .

(7) allocate a variable of t:yielder [of a fixed-array-type] =
| respectively allocate variables of
| (the element-type of t)$^{(\text{the length of } t)}$ and
| give the index-type of t
then give the array of the given (variable*, ordinal-type) .

- dispose of _ :: yielder [of an array-variable] → action [storing] .

(8) dispose of a:yielder [of an array-variable] =
dispose of all the elements of a .

3.17 Records

3.17.1 Values

needs: Typed Entities)/Values, Associations.

introduces: record-value , record of _ , fields _ ,
field _ of _ , defaulted-field-values _ ,
_ is _ .

Note duplication of operations in Record/Values and Record/Variables.

- record of _ ::
  associations[value] → record-value (*partial*) ,
  associations[putative value] → putative record-value (*partial*) .

(1) the record of (f_1:associations[putative value], f_2:association[uninitialized],
f_3:associations[putative value]) =
uninitialized .

- fields _ :: record-value → associations[value] (*total*) .

(2) r = (record of f:associations[value]) : record-value ⇒
fields $r = f$.

- field _ of _ :: identifier-token, record-value → value (*partial*) .

(3) field i:identifier-token of r:record-value =
component associated with i in fields r .

- defaulted-field-values _ :: (associations[value], associations[(type,value?)]) → associations[value] (*total*?) .

  'defaulted-field-values (a, a')' yields the tuple of associations a rearranged and defaulted (where necessary) according to a'. Initially, for every association in a there is a corresponding association in a'. Is this operation misplaced in Records?

(4) defaulted-field-values (a:associations[value], ()) = () .

(5) v = (component associated with i in a) : value ⇒
defaulted-field-values (a:associations[value],
association of i:identifier-token to (t:type, v':value?)) =
association of i to v' .

(6) component associated with i in a = nothing $\Rightarrow$
defaulted-field-values (a:associations[value],
association of i:identifier-token to (t:type, v:value)) =
association of i to v .

(7) defaulted-field-values (a:associations[value],
a':association[(type,value?)],
a'':associations[(type,value?)]) =
(defaulted-field-values (a, a'), defaulted-field-values (a, a'')) .

- _ is _ :: record-value, record-value → truth-value (*total*) .

(8) r_1:record-value is r_2:record-value =
components (fields r_1) is components (fields r_2) .

3.17.2 Types

needs: **Typed Entities/Types**, **Associations**, **Bindings/Tokens**.

introduces: record-type , record-type of _ , fields _ ,
field-type designated by _ in _.

- record-type of _ :: associations[type] → record-type (*partial*) .

- fields _ :: record-type → associations[type] (*total*) .

(1) t = (record-type of f:associations[type]) : record-type $\Rightarrow$
fields $t = f$.

- field-type designated by _ in _ :: identifier-token, record-type → type (*partial*) .

(2) field-type designated by i:identifier-token in t:record-type =
component associated with i in fields t .

3.17.3 Variables

needs: **Records/Values**, **Records/Types**, **Typed Entities/Variables**, **Associations**, **Errors**, **Bindings/Tokens**, **[2]/Action Notation**.

introduces: record-variable , record of _ ,
fields _ , field _ of _ ,
_ is _ ,
the value assigned to _ , assign _ to _ ,
uninitialize _ , allocate a variable of _ , dispose of _ .

- record of _ :: associations[variable] → record-variable (*partial*) .

- fields _ :: record-variable → associations[variable] (*total*) .

(1) r = (record of f:associations[variable]) : record-variable $\Rightarrow$
fields $r = f$.

- field _ of _ :: identifier-token, record-variable → variable (*partial*) .

(2) field i:identifier-token of r:record-variable =
the component associated with i in the fields of r .

- _ is _ :: record-variable, record-variable → truth-value (*partial, commutative*) .

(3) r_1:record-variable is r_2:record-variable =
fields r_1 is fields r_2 .

This definition is wrong, yielding true if both record-variables have empty fields. But even a record-variable with no fields at all has its own identity!

- the value assigned to _ :: yielder [of a record-variable] →
yielder [of a putative record-value] [using current storage] .

(4) the value assigned to r:yielder [of a record-variable] =
the record of
the associations of the identifiers of the fields of r to
the values respectively assigned to
the components of the fields of r) .

- assign _ to _ :: yielder [of a record-value], yielder [of a record-variable] →
action [storing] .

(5) assign r_1:yielder [of a record-value] to r_2:yielder [of a record-variable] =
respectively assign the components of the fields of r_1 to
the components of the fields of r_2 .

- uninitialize _ :: yielder [of a record-variable] →
action [storing] [using current storage] .

(6) uninitialize r:yielder [of a record-variable] =
uninitialize all the components of the fields of r .

- allocate a variable of _ :: yielder [of a record-type] →
action [giving a record-variable | storing] [using current storage] .

(7) allocate a variable of t:yielder [of a record-type] =
respectively allocate variables of
the type* yielded by the components of the fields of t then
give the record of the associations of
the identifiers of the fields of t to the given variable*) .

- dispose of _ :: yielder [of a record-variable] → action [storing] .

(8) dispose of r:yielder [of a record-variable] =
dispose of all the components of the fields of r .

3.18 Opaque Types

needs: ???.

introduces: opaque-type , opaque-type of _ .

3.19 Brands

needs: **Texts/Values**, **Typed Entities/Types**.

introduces: brand , anonymous-brand , anonymous-brand unique in _ , _ is _ .

- brand = text-value | anonymous-brand ($disjoint$) .
- anonymous-brand unique in _ :: bindings → anonymous-brand .

(1) anonymous-brand unique in b:bindings : anonymous-brand .

(2) anonymous-brand unique in b:bindings = a:anonymous-brand ; t:type branded with a':anonymous-brand is in mapped-set b = true ⇒ a is a' = false .

- _ is _ :: brand, brand → truth-value ($total$) .

3.20 Associations

needs: **Bindings/Tokens**.

introduces: association , association[_] , associations[_] ,
association of _ to _ ,
identifier _ , component _ ,
associations of _ to _ ,
identifiers _ , components _ ,
component associated with _ in _ .

- association[_] :: data → association .
- associations[_] :: data → associations .

- $D \leq$ data ⇒
  - association of _ to _ :: identifier-token, D → association[D] ($total$) ;
  - identifier _ :: association[D] → identifier-token ($total$) ;
  - component _ :: association[D] → D ($total$) ;
  - associations of _ :: (identifier-token*, D*) → association[D]* ($partial$) ;
  - identifiers _ :: associations[D] → identifier-token* ($total$) ;
  - components _ :: associations[D] → D* ($total$) ;
  - component associated with _ in _ ::
    identifier-token, association[D]* → association[D] ($partial$) .

(1) $D \leq$ data ⇒

- association[D] = association of identifier-token to D ;
- associations[D] = association[D]* ;

(1) a = association of i:identifier-token to d:D ⇒

(1) identifier $a = i$;

(2) component $a = d$;

(2) associations of () to d:D = () ;

(3) associations of i:identifier-token$^+$ to d:D =
(association of first i to d, associations of rest i to d) ;

(4) identifiers () = () ;

(5) identifiers (a:association[D], a':associations[D]) =
    (identifier a, identifiers a') ;

(6) components () = () ;

(7) components (a:association[D], a':associations[D]) =
    (component a, components a') ;

(8) component associated with i:identifier-token in () = nothing ;

(9) component associated with i:identifier-token in
    (a:association[D], a':associations[D]) =
    when i is identifier a then component a **|**
    component associated with i in a' .

3.21 Exceptions

needs: **Typed Entities/Values**,
[2]/Action Notation.

introduces: exception-name , basic-exception-name , normal-exception-name ,
_ is _ ,
exception , primitive-exception , proper-exception ,
exception of _ , cause _ , component _ , _ matches _ ,
handle _ , _ until exit , _ until return ,
_ and then terminally _ .

- exception-name =
  basic-exception-name **|** normal-exception-name (*disjoint*) .
- basic-exception-name =
  exit-exception **|** return-exception (*individual*) .
- normal-exception-name $\leq$ cell .

- _ is _ :: primitive-exception, primitive-exception $\rightarrow$ truth-value (*total*, *commutative*) .

- exception = primitive-exception **|** proper-exception (*disjoint*) .
- primitive-exception = exception-name **|** $\square$.

- exception of _ ::
  (exception-name, value) $\rightarrow$ proper-exception (*total*, *injective*) .

- cause _ :: exception $\rightarrow$ primitive-exception (*total*) .
- component _ :: exception $\rightarrow$ value$^?$ (*total*) .

(1) e : primitive-exception $\Rightarrow$

  (1) cause e = e ;

  (2) component e = () .

(2) e = (exception of (n:exception-name, v:value)) : proper-exception $\Rightarrow$

- (1) cause $e = n$;
- (2) component $e = v$.

- _ matches _ :: exception, exception-name → truth-value (*total*) .

(3) e:exception matches n:exception-name = cause e is n .

- handle _ :: yielder [of a basic-exception-name] →
  action [giving a value$^?$ | escaping with an exception]
  [using the given exception] .

(4) handle n:basic-exception-name =
  | check (the given exception matches the exception yielded by n)
  | and then give the component of the given exception
  or
  | check not (the given exception matches the exception yielded by n)
  | and then escape with the given exception .

... superfluous?

- _ until exit ::
  action [completing | storing | diverging | escaping with an exception | erring]
  [using current bindings | current storage] →
  action [completing | storing | diverging | escaping with an exception | erring]
  [using current bindings | current storage] .

(5) A:action until exit =
  A trap
  | check (the given exception matches the exit-exception) and then
  | complete
  or
  | check not (the given exception matches the exit-exception) and then
  | escape with the given exception .

- _ until return ::
  action [completing | storing | diverging | escaping with an exception | erring]
  [using current bindings | current storage] →
  action [completing | storing | diverging | escaping with an exception | erring]
  [using current bindings | current storage] .

(6) A:action until return =
  A trap
  | check (the given exception matches the return-exception) and then
  | give the component of the given exception
  or
  | check not (the given exception matches the return-exception) and then
  | escape with the given exception .

- _ and then terminally _ :: action, action → action .

(7) A_1:action and then terminally A_2:action =
| A_1
| trap
| | A_2 and then escape with the given exception
and then A_2 .

3.22 Errors

needs: **Data**,
[2]/Action Notation.

introduces: error ,
array-indexing-error , assignment-error , case-error , character-error ,
integer-division-error , integer-overflow-error ,
missing-function-result-error , modulo-error , natural-logarithm-error ,
nil-procedure-or-method-error , range-error , real-division-error ,
real-overflow-error , referent-error , square-error ,
square-root-error , stub-procedure-or-method-error , type-case-error ,
_ but _ ,
report _ , _ reporting _ , erring .

privately introduces: proper-datum .

- error = array-indexing-error | assignment-error | case-error | character-error |
  integer-division-error | integer-overflow-error |
  missing-function-result-error | modulo-error | natural-logarithm-error |
  nil-procedure-or-method-error | range-error | real-division-error |
  real-overflow-error | referent-error | square-error |
  square-root-error | stub-procedure-or-method-error | type-case-error |
  □ (*individual*) .

- primitive-exception = □ | error .

- datum = proper-datum | error (*disjoint*) .

- _ but _ :: datum, error → datum (*total*) .

(1) e:error but e':error = e' .

(2) d:proper-datum but e:error = d .

- report _ :: yielder [of an error] → action [erring] .

(3) report e:yielder [of an error] = □ .

- _ reporting _ :: action, yielder [of an error] → action (*total*) .

(4) A:action reporting e:yielder [of an error] =
A then
| give the given proper-datum or
| | check there is given an error then report e .

- erring : outcome .

(5) erring = escaping with an error | communicating .

Clean up relationship between errors and exceptions.

3.23 Interfaces

needs: ???.

introduces: ??? .

3.24 Bindings

needs: **Typed Entities**, **Exceptions**.

- bindable = value | type | variable | exception |
  interface | module | generic-interface | generic-module .

3.24.1 Tokens

introduces: identifier-token ,
respectively bind _ to _ , the variables respectively bound to _ .

- token = identifier-token .
- identifier-token = string of (letter, (letter | digit | '_')*) .
- respectively bind _ to _ :: yielder [of identifier-token*], yielder [of bindable*] →
  action [binding] .

(1) respectively bind i:yielder [of identifier-token*]
to b:yielder [of bindable*]=
| check (i is ())
or
| bind the first of i to the first of b and
| respectively bind the rest of i to the rest of b .

- the variables respectively bound to _ :: yielder [of identifier-token*] →
  yielder [of variable*] .

(2) the variables respectively bound to i:yielder [of identifier-token*] =
if i is ()
then ()
else (the variable bound to the first of i,
the variables respectively bound to the rest of i) .

3.24.2 Required

needs: ??? .

introduces: required-bindings .

- required-bindings : bindings .

(1) required-bindings = ??? .

Bibliography

[1] Cardelli, L., Donahue, J., Glassman, L., Jordan, M., Kalsow, W., and Nelson, G. (1991) Language definition, pp. 11–66 in Nelson, G. (ed.) *Systems Programming with Modula-3*, Prentice Hall, Englewood Cliffs, New Jersey.

[2] Mosses, P.D. (1992) *Action Semantics*, Cambridge University Press, Cambridge, England.

[3] Watt, D.A. (1991) *Programming Language Syntax and Semantics*, Prentice Hall International, Hemel Hempstead, England.