PASCAL ACTION SEMANTICS

Peter D. Mosses[1] David A. Watt[2]

Version 0.6: 1993–03–22 (17.08)

[1]Computer Science Department, Aarhus University, Ny Munkegade Bldg. 540, DK–8000 Aarhus C, Denmark
[2]Department of Computing Science, University of Glasgow, Lilybank Gardens, Glasgow G12 8QQ, Scotland

Preface

This report, version 0.6 of the Pascal Action Semantics, provides an almost complete formal specification of the dynamic semantics of Standard Pascal (Level 0). It reflects the authors' understanding of the Pascal Standard[1] . However, it is not intended primarily as a contribution to the understanding of Pascal. Rather, it is intended as a 'showcase' example of the use of action semantics, aimed at readers who are already acquainted with action notation.

Contents

Chapters 1–3 of this report are modules named Abstract Syntax, Semantic Functions, and Semantic Entities, respectively. (This is quite literally true: our specification language allows modules and sub-modules to be presented in the form of chapters and sections of a written document.)

The Abstract Syntax module specifies the abstract syntax of Standard Pascal, in the sense that it defines a set of syntax trees that correspond to Pascal source programs. In contrast to some other formal specifications of Pascal, this one keeps the abstract syntax very close to the concrete syntax, so readers should have no difficulty in mentally translating between the abstract and concrete syntax.

The Semantic Functions module specifies the mapping of syntactic entities to semantic entities. The simplest syntactic entities such as literals are mapped to simple semantic entities such as numbers and characters. The more complex syntactic entities such as expressions, statements, and declarations are mapped to actions. The operations that map syntactic entities to semantic entities correspond to the semantic functions of the denotational framework, and the equations that define these operations correspond to the semantic equations of that framework.

The Semantic Entities module specifies the meanings of the semantic entities themselves. Many of the semantic entities used in the Pascal Action Semantics are generic, and are borrowed from standard action notation and data notation. The Semantic Entities module serves to specify additional semantic entities that are specific to Pascal. The Typed Entities, Primitive Typed Entities, Ordinals, Booleans, Numbers, ..., Arrays, Records, and Files sub-modules specify the properties of values and variables of the various types and type classes of Pascal. The Procedures and Functions sub-module specifies the properties of procedures, functions, and parameters. The Errors sub-module loosely[2] specifies error handling. The External Entities sub-module loosely specifies the association of program parameters with external entities.

The order of the three modules is intended to encourage reading of the Pascal Action Semantics in a top-down fashion. (Indeed, the modules were written in this same order. It was the authors' experience that only minor changes to Abstract Syntax were required during the writing of Semantic Functions, and that only minor changes to the latter were required during the writing of Semantic Entities.)

The report finally contains a symbol index. At present, this contains entries for all *definitions* of symbols. *Uses* of symbols have not yet been fully indexed.

[1] International Standard 7185 (1982), which is identical to British Standard 6192 (1982). A 1992 draft revision clarifies certain obscure points in the 1982 standard.

[2] In the Pascal Standard, such things are described as *implementation-defined* or *implementation-dependent*.

Accuracy

To the best of our knowledge, this version of the Pascal Action Semantics is a complete and accurate specification of the semantics of Standard Pascal (Level 0), with the following exceptions:

- When assignment to the tag field of a tagged variant record causes a change of active variant, the fields of the previously-active variant should become uninitialized.
- When '_is_' is used to compare the identity of two variables, it fails to distinguish between two empty record variables, even if they were allocated at different times.

Readership

This version of the Pascal Action Semantics is intended only for readers who are already familiar with the action semantics framework, as described in the recent books by the authors[3] [4] .

A future version will be designed for a more general readership. That version will include:

- a motivating introduction;
- notes, explaining some of the finer points of Pascal and motivating some of the choices made in formulating the semantic description;
- appendices, including an informal summary of standard action notation and data notation, and a brief explanation of the meta-notation;
- a cross-reference with the Pascal Standard;
- a more comprehensive index, to include entries for the symbols of standard action and data notation, and to include entries for *uses* as well as definitions of symbols introduced by the Semantic Entities module;
- automated checking of the internal consistency of the specification; and
- a bibliography.

Feedback

Please help by sending errata, comments, and suggestions for improvements to the authors, preferably by e-mail (`pdmosses@daimi.aau.dk`, `daw@dcs.glasgow.ac.uk`). Further versions will be announced on the action semantics mailing list as soon as they become available.

Peter D. Mosses, David A. Watt
March 1993.

[3] Mosses, P.D. (1992) *Action Semantics*, Cambridge Tracts in Theoretical Computer Science 26, Cambridge University Press, Cambridge, England.

[4] Watt, D.A. (1991) *Programming Language Syntax and Semantics*, Prentice Hall International, Hemel Hempstead, England.

Contents

Chapter 1

Abstract Syntax

Identifiers.

Labels.

Literals

needs: String Characters.

String Characters.

Constants

needs: Identifiers, Literals.

Expressions

needs: Identifiers, Literals.

Statements

needs: Identifiers, Labels, Constants, Expressions.

Blocks and Declarations

needs: Identifiers, Labels, Constants, Expressions, Statements, Type Denoters.

Type Denoters

needs: Identifiers, Constants.

Programs

needs: Identifiers, Blocks and Declarations.

closed except Literals/String Characters.

grammar:

1.1 Identifiers

introduces: Identifier .

Note: Abstract identifiers are already lowercase.

- Word = Word-Symbol | Identifier (*disjoint*) .
- Word = ⟦ (lowercase letter) (lowercase letter | digit)* ⟧ .
- Word-Symbol = "and" | "array" | "begin" | "case" | "const" | "div" | "do" |
  "downto" | "else" | "end" | "file" | "for" | "function" |
  "goto" | "if" | "in" | "label" | "mod" | "nil" | "not" |
  "of" | "or" | "packed" | "procedure" | "program" | "record" | "repeat" |
  "set" | "then" | "to" | "type" |
  "until" | "var" | "while" | "with" .
- Identifiers = ⟨ Identifier ⟨ "," Identifier ⟩* ⟩ .

1.2 Labels

- Label = ⟦ digit$^+$ ⟧ .

1.3 Literals

- Literal = Unsigned-Integer | Unsigned-Real | Character-String .
- Unsigned-Integer = ⟦ digit$^+$ ⟧ .
- Unsigned-Real = ⟦ digit$^+$ '.' digit$^+$ ⟧ | ⟦ digit$^+$ 'e' ('+' | '−')$^?$ digit$^+$ ⟧ |
  ⟦ digit$^+$ '.' digit$^+$ 'e' ('+' | '−')$^?$ digit$^+$ ⟧ .
- Character-String = ⟦ ''' String-Element$^+$ ''' ⟧ .
- String-Element = string-character | ⟨ ''' ''' ⟩ .

1.3.1 String Characters

includes: [Mosses92]/**Data Notation/Characters** (character) .

introduces: string-character .

- character $\geq$ string-character | ''' (*disjoint*) .

open.

1.4 Constants

- Constant = Identifier | Literal | ⟦ ("+" | "−") Constant ⟧ .
- Constants = ⟨ Constant ⟨ "," Constant ⟩* ⟩ .

1.5 Expressions

- Variable-Access = Identifier | ⟦ Variable-Access "[" Expression "]" ⟧ |
  ⟦ Variable-Access "." Identifier ⟧ | ⟦ Variable-Access "^" ⟧ .
- Expression = Identifier | Literal | "nil" |
  Variable-Access | ⟦ "(" Expression ")" ⟧ |
  ⟦ Monadic-Operator Expression ⟧ |
  ⟦ Expression Dyadic-Operator Expression ⟧ |
  ⟦ "[" Members$^?$ "]" ⟧ |
  ⟦ Identifier "(" Actual-Parameters ")" ⟧ .
- Monadic-Operator = "+" | "−" | "not" .
- Dyadic-Operator = "+" | "−" | "*" | "/" | "div" | "mod" |
  "=" | "<>" | "<" | "<=" | ">" | ">=" |
  "and" | "or" | "in" .
- Member = Expression | ⟦ Expression ".." Expression ⟧ .
- Members = ⟨ Member ⟨ "," Member ⟩* ⟩ .
- Actual-Parameter = Expression | ⟦ Expression ":" Expression ⟧ |
  ⟦ Expression ":" Expression ":" Expression ⟧ .
- Actual-Parameters = ⟨ Actual-Parameter ⟨ "," Actual-Parameter ⟩* ⟩ .

1.6 Statements

- Statements = ⟨ Statement ⟨ ";" Statement ⟩* ⟩ .
- Statement = ⟨ ⟨ Label ":" ⟩$^?$ Unlabeled-Statement ⟩ .
- Unlabeled-Statement = ⟦ "begin" Statements "end" ⟧ |
  ⟦ "goto" Label ⟧ |
  ⟦ Variable-Access ":=" Expression ⟧ |
  ⟦ "if" Expression "then" Statement ⟨ "else" Statement ⟩$^?$ ⟧ |
  ⟦ "case" Expression "of" Cases "end" ⟧ |
  ⟦ "repeat" Statements "until" Expression ⟧ |
  ⟦ "while" Expression "do" Statement ⟧ |
  ⟦ "for" Identifier ":=" Expression
  ("to" | "downto") Expression "do" Statement ⟧ |
  ⟦ Identifier ⟨ "(" Actual-Parameters ")" ⟩$^?$ ⟧ |
  ⟦ "with" Variable-Access "do" Statement ⟧ .
- Case = ⟦ Constants ":" Statement ⟧ .
- Cases = ⟨ Case ⟨ ";" Case ⟩* ⟩ .

1.7 Blocks and Declarations

- Block = ⟦ Label-Declaration-Part$^?$ Constant-Definition-Part$^?$
  Type-Definition-Part$^?$ Variable-Declaration-Part$^?$
  (Procedure-Declaration | Function-Declaration)*
  "begin" Statements "end" ⟧ .

- Label-Declaration-Part = ⟦ “label” Labels “;” ⟧ .
- Labels = ⟨ Label ⟨ “,” Label ⟩* ⟩ .
- Constant-Definition-Part = ⟦ “const” Constant-Definition$^+$ ⟧ .
- Constant-Definition = ⟦ Identifier “=” Constant “;” ⟧ .
- Type-Definition-Part = ⟦ “type” Type-Definition$^+$ ⟧ .
- Type-Definition = ⟦ Identifier “=” Type-Denoter “;” ⟧ .
- Variable-Declaration-Part = ⟦ “var” Variable-Declaration$^+$ ⟧ .
- Variable-Declaration = ⟦ Identifiers “:” Type-Denoter “;” ⟧ .
- Procedure-Declaration = ⟦ “procedure” Identifier Formal-Parameter-List$^?$ “;”
  (Block | “forward”) “;” ⟧ .
- Function-Declaration = ⟦ “function” Identifier Formal-Parameter-List$^?$
  “:” Identifier “;” (Block | “forward”) “;” ⟧ |
  ⟦ “function” Identifier “;” Block “;” ⟧ .
- Formal-Parameter-List = ⟨ “(” Formal-Parameter-Sections “)” ⟩ .
- Formal-Parameter-Section = ⟦ “var”$^?$ Identifiers “:” Identifier ⟧ |
  ⟦ “procedure” Identifier Formal-Parameter-List$^?$ ⟧ |
  ⟦ “function” Identifier Formal-Parameter-List$^?$ “:” Identifier ⟧ .
- Formal-Parameter-Sections = ⟨ Formal-Parameter-Section ⟨ “;” Formal-Parameter-Section ⟩* ⟩ .

1.8 Type-Denoters

- Type-Denoter = Identifier | ⟦ “(” Identifiers “)” ⟧ | ⟦ Constant “..” Constant ⟧ |
  ⟦ “packed”$^?$ “array” “[” Type-Denoter “]” “of” Type-Denoter ⟧ |
  ⟦ “packed”$^?$ “record” Field-List “end” ⟧ |
  ⟦ “packed”$^?$ “set” “of” Type-Denoter ⟧ |
  ⟦ “packed”$^?$ “file” “of” Type-Denoter ⟧ |
  ⟦ “^” Identifier ⟧ .
- Field-List = ⟦ Field-Declaration* Variant-Part$^?$ ⟧ .
- Field-Declaration = ⟦ Identifiers “:” Type-Denoter “;” ⟧ .
- Variant-Part = ⟨ “case” ⟨ Identifier “:” ⟩$^?$ Identifier “of” Variants ⟩ .
- Variant = ⟦ Constants “:” “(” Field-List “)” ⟧ .
- Variants = ⟨ Variant ⟨ “;” Variant ⟩* ⟩ .

1.9 Programs

- Program = ⟦ “program” Identifier ⟨ “(” Identifiers “)” ⟩$^?$ “;” Block “.” ⟧ .

Chapter 2

Semantic Functions

needs: **Abstract Syntax, Semantic Entities.**

Identifiers.

Labels.

Literals

  needs: **String Characters.**
  Numerals.
  Character Strings.

Constants

  needs: **Identifiers, Literals.**

Expressions

  needs: **Identifiers, Literals.**
  Investigating Expressions.
  Evaluating Expressions.
  Evaluating Actual Parameters.

Statements

  needs: **Identifiers, Labels, Constants, Expressions.**
  Executing Statements.
  Executing Labeled Statements.

Blocks and Declarations

  needs: **Identifiers, Labels, Constants, Expressions, Statements, Type Denoters.**
  Activating Blocks.
  Establishing Declarations and Definitions.
  Relinquishing Variables.

Type Denoters

  needs: **Identifiers, Constants.**

Programs

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

2.1 Identifiers

introduces: the _ , the identifiers of _ .

- the _ :: Identifier $\rightarrow$ identifier-token .

(1) the I:Identifier $= I$.
Note: Thus we may write I instead of 'the I'.

- the identifiers of _ :: Identifiers $\rightarrow$ identifier-token$^+$.

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

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

2.2 Labels

introduces: the spelling of _ .

- the spelling of _ :: Label $\rightarrow$ label-token .

(1) the spelling of $[\![$ d:digit$^+$ $]\!]$ = (the numerical-value of d) [max 9999] .

2.3 Literals

introduces: the value of _ .

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

2.3.1 Numerals

- the value of _ :: Unsigned-Integer $\rightarrow$ integer-number ,
  Unsigned-Real $\rightarrow$ real-number .

(1) the value of $[\![$ d:digit$^+$ $]\!]$ = integer-number & the integer-number of the numerical-value of d .
Note: literal error reporting is part of static semantics.

(2) the value of $[\![$ c:$\langle$ digit$^+$ 'e' ('+' | '−')$^?$ digit$^+$ $\rangle$ $]\!]$ = the real-number of the numerical-value of c .

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

2.3.2 Character Strings

- the value of _ :: Character-String $\rightarrow$ character | string-value .

(1) the value of $[\![$ ''' S:String-Element ''' $]\!]$ = the value of S .

(2) the value of $[\![$ ''' S:$\langle$ String-Element String-Element$^+$ $\rangle$ ''' $]\!]$ =
the string-value of the character-values of S .

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

- the character-value of _ :: String-Element $\rightarrow$ character .

(3) the character-value of c:string-character = c .

(4) the character-value of $\langle$ '' '' $\rangle$ = the apostrophe-image-character .
Note: Not necessarily the same as ''!

- the character-values of _ :: String-Element$^+$ $\rightarrow$ character$^+$.

(5) the character-values of S:String-Element = the character-value of S .

(6) the character-values of $\langle$ S_1:String-Element S_2:String-Element$^+$ $\rangle$ =
(the character-value of S_1, the character-values of S_2) .

2.4 Constants

introduces: the constant-value of _ , the case-constant-set of _ .

- the constant-value of _ :: Constant $\rightarrow$
  yielder [of a constant-value] [using current bindings] .

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

(2) the constant-value of L:Literal = the value of L .

(3) the constant-value of $[\![$ "+" C:Constant $]\!]$ =
the number yielded by the constant-value of C .

(4) the constant-value of $[\![$ "−" C:Constant $]\!]$ =
the negation of the number yielded by the constant-value of C .

- the case-constant-set of _ :: Constants $\rightarrow$
  yielder [of a set-value] [using current bindings] .

(5) the case-constant-set of C:Constant =
the set of the constant-value of C .

(6) the case-constant-set of $\langle$ C_1:Constant "," C_2:Constants $\rangle$ =
the union of (the set of the constant-value of C_1, the case-constant-set of C_2) .

2.5 Expressions

introduces: investigate _ , evaluate _ , modally evaluate _ .

- investigate _ :: Expression $\rightarrow$ action .
- evaluate _ :: Expression $\rightarrow$ action .
- modally evaluate _ :: Actual-Parameters $\rightarrow$ action .

2.5.1 Investigating Expressions

- investigate _ :: Expression →
  action [giving a variable | giving a value | storing | diverging | jumping | erring]
  [using current bindings | current storage] .

(1) investigate I:Identifier =
give the constant-value bound to I or
give the variable bound to I or
give the result-variable of the function bound to I .

(2) investigate ⟦ V:Variable-Access "[" E:Expression "]" ⟧ =
| investigate V and evaluate E
then
| give the component indexed by the given ordinal-value#2 in the given array-variable#1
| reporting any given error .

(3) investigate ⟦ V:Variable-Access "." I:Identifier ⟧ =
investigate V then
| access the component designated by I in the given record-variable .

(4) investigate ⟦ V:Variable-Access "^" ⟧ =
investigate V then
| | give the target-variable of the pointer-value attributed to the given pointer-variable
| | reporting any given error
| or
| | give the buffer-part of the given file-variable .

(5) E & Variable-Access = nothing ⇒
investigate E:Expression = evaluate E .

2.5.2 Evaluating Expressions

- evaluate _ :: Expression →
  action [giving a value | storing | diverging | jumping | erring]
  [using current bindings | current storage] .

(1) evaluate I:Identifier =
give the constant-value bound to I
or
| give the value attributed to the variable bound to I
| reporting any given error
or
enact the body-abstraction of the function bound to I .

(2) evaluate L:Literal = give the value of L .

(3) evaluate "nil" = give the nil-pointer .

(4) V & Identifier = nothing ⇒
evaluate V:Variable-Access =
investigate V then
| give the value attributed to the given variable
| reporting any given error .

(5) evaluate ⟦ "(" E:Expression ")" ⟧ = evaluate E .

(6) evaluate $[\![$ O:Monadic-Operator E:Expression $]\!]$ =
  evaluate E then
  | give the monadic-result of O reporting any given error .

(7) evaluate $[\![$ 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 .

(8) evaluate $[\![$ "[" M:Members "]" $]\!]$ = collect M .

(9) evaluate $[\![$ I:Identifier "(" A:Actual-Parameters ")" $]\!]$ =
  give the formal-parameter-modes of the function bound to I then
  modally evaluate A then
  enact the application of (the body-abstraction of the function bound to I)
    to the given argument$^+$.

privately introduces: the monadic-result of _ , the dyadic-result of _ , collect _ .

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

(10) the monadic-result of "+" = the given number .

(11) the monadic-result of "−" = the negation of the given number .

(12) the monadic-result of "not" = not (the given truth-value) .

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

(13) the dyadic-result of "+" =
  the sum of (the given number#1, the given number#2) |
  the union of (the given set-value#1, the given set-value#2) .

(14) the dyadic-result of "−" =
  the difference of (the given number#1, the given number#2) |
  the difference of (the given set-value#1, the given set-value#2) .

(15) the dyadic-result of "*" =
  the product of (the given number#1, the given number#2) |
  the intersection of (the given set-value#1, the given set-value#2) .

(16) the dyadic-result of "/" =
  the quotient of (the given number#1, the given number#2) .

(17) the dyadic-result of "div" =
  the integer-quotient of (the given integer-number#1, the given integer-number#2) .

(18) the dyadic-result of "mod" =
  the modulo of (the given integer-number#1, the given integer-number#2) .

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

(20) the dyadic-result of "<>" = not (the given value#1 is the given value#2) .

(21) the dyadic-result of "<" =
    (the given ordinal-value#1 is less than the given ordinal-value#2) **|**
    (the given number#1 is less than the given number#2) **|**
    (the given string-value#1 is lexicographically less than
     the given string-value#2) .

(22) the dyadic-result of "<=" =
    not (the given ordinal-value#1 is greater than the given ordinal-value#2) **|**
    not (the given number#1 is greater than the given number#2) **|**
    not (the given string-value#1 is lexicographically greater than
        the given string-value#2) **|**
    (the given set-value#1 is included in the given set-value#2) .

(23) the dyadic-result of ">" =
    (the given ordinal-value#1 is greater than the given ordinal-value#2) **|**
    (the given number#1 is greater than the given number#2) **|**
    (the given string-value#1 is lexicographically greater than
     the given string-value#2) .

(24) the dyadic-result of ">=" =
    not (the given ordinal-value#1 is less than the given ordinal-value#2) **|**
    not (the given number#1 is less than the given number#2) **|**
    not (the given string-value#1 is lexicographically less than
        the given string-value#2) **|**
    (the given set-value#2 is included in the given set-value#1) .

(25) the dyadic-result of "and" = both (the given truth-value#1, the given truth-value#2) .

(26) the dyadic-result of "or" = either (the given truth-value#1, the given truth-value#2) .

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

- collect _ :: Members$^?$ $\rightarrow$
    action [giving a set-value **|** storing **|** diverging **|** jumping **|** erring]
        [using current bindings **|** current storage] .

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

(29) collect ⟦ 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#1, the given ordinal-value#2) .

(30) collect ⟨ M_1:Member "," M_2:Members ⟩ =
    | collect M_1 and collect M_2
    then give the union of (the given set-value#1, the given set-value#2) .

(31) collect ⟨ ⟩ = give the empty-set .

2.5.3 Evaluating Actual Parameters

- modally evaluate _ :: Actual-Parameters $\rightarrow$
    action [giving argument$^+$ **|** storing **|** diverging **|** jumping **|** erring]
        [using the given parameter-modes **|** current bindings **|** current storage] .

Note: The only case when modally evaluate E receives more than a single parameter-mode (in a legal program) is when dealing with the first actual parameter of a file-handling or allocation procedure or function! Moreover, whether the first parameter-mode is read- or write-file-variable-mode determines whether any second parameter-mode is a read-variable- or write-value-mode.

(1) modally evaluate E:Expression =

| check there is given a value-mode#1 then
| evaluate E

or

| check there is given a variable-mode#1 then
| investigate E then give the given variable

or

| check there is given a procedural-mode#1 then
| give the procedure yielded by the entity identified by E

or

| check there is given a functional-mode#1 then
| give the function yielded by the entity identified by E

or

| check there is given a write-value-mode#1 then
| give the unevaluated-argument of the closure of abstraction of evaluate E

or

| check there is given a read-variable-mode#1 then
| give the unevaluated-argument of the closure of abstraction of investigate E

or

| check there is given a file-variable-mode#1 and then
| | | check there is the file-type yielded by the variable-type of E then
| | | investigate E then give the given file-variable
| | or
| | check there is the nonfile-type yielded by the variable-type of E and then
| | | | check there is given a write-file-variable-mode then
| | | | give (the text-file-variable bound to the output-token,
| | | | the unevaluated-argument of the closure of abstraction of evaluate E)
| | | or
| | | | check there is given a read-file-variable-mode then
| | | | give (the text-file-variable bound to the input-token,
| | | | the unevaluated-argument of the closure of abstraction of investigate E) .

(2) modally evaluate $[\![$ E_1:Expression ":" E_2:Expression $]\!]$ =

| evaluate E_1 and evaluate E_2

then

| give the text-field of (the given textually-writable#1, the given integer-number#2)
| reporting any given error .

(3) modally evaluate $[\![$ E_1:Expression ":" E_2:Expression ":" E_3:Expression $]\!]$ =

| evaluate E_1 and evaluate E_2 and evaluate E_3

then

| give the text-field of (the given real-number#1,
| the given integer-number#2, the given integer-number#3)
| reporting any given error .

(4) modally evaluate $\langle$ A_1:Actual-Parameters "," A_2:Actual-Parameter $\rangle$ =

| give the penultimate of the given parameter-modes then
| modally evaluate A_1

and

| give the ultimate of the given parameter-modes then
| modally evaluate A_2 .

privately introduces: the entity identified by _ , the variable-type of _ .

- the entity identified by _ :: Expression →
  yielder [of a bindable] [using current bindings] .

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

(6) E & Identifier = nothing ⇒
the entity identified by E:Expression = nothing .

- the variable-type of _ :: Expression →
  yielder [of a type] [using current bindings] .

(7) the variable-type of I:Identifier =
the type possessed by the variable bound to I .

(8) the variable-type of ⟦ V:Variable-Access "[" E:Expression "]" ⟧ =
the component-type of the array-type yielded by the variable-type of V .

(9) the variable-type of ⟦ V:Variable-Access "." I:Identifier ⟧ =
the component-type designated by I in
the record-type yielded by the variable-type of V .

(10) the variable-type of ⟦ V:Variable-Access "^" ⟧ =
the target-type of the pointer-type yielded by the variable-type of V |
the component-type of the file-type yielded by the variable-type of V.

(11) E & Variable-Access = nothing ⇒
the variable-type of E:Expression = nothing .

2.6 Statements

introduces: execute _ , jumpily execute _ .

- execute _ :: Statements → action .
- jumpily execute _ :: Statements → action .

2.6.1 Executing Statements

- execute _ :: Statements →
  action [completing | storing | diverging | jumping | erring]
  [using current bindings | current storage] .

(1) execute ⟨ S_1:Statement ";" S_2:Statements ⟩ = execute S_1 and then execute S_2 .

(2) execute ⟨ L:Label ":" S:Unlabeled-Statement ⟩ = execute S .

(3) execute ⟦ "begin" S:Statements "end" ⟧ = jumpily execute S .

(4) execute ⟦ "goto" L:Label ⟧ = escape with the program-point bound to the spelling of L .

(5) execute ⟦ V:Variable-Access ":=" E:Expression ⟧ =
| investigate V and evaluate E
then assign the given value#2 to the given variable#1 .

(6) execute ⟦ "if" E:Expression "then" S:Statement ⟧ =
evaluate E then
| | check (the given truth-value is true) then jumpily execute S
| or
| | check (the given truth-value is false) .

(7) execute ⟦ "if" E:Expression "then" S_1:Statement "else" S_2:Statement ⟧ =
evaluate E then
| | check (the given truth-value is true) then jumpily execute S_1
| or
| | check (the given truth-value is false) then jumpily execute S_2 .

(8) execute ⟦ "case" E:Expression "of" C:Cases "end" ⟧ =
evaluate E then
| | select from C
| or
| | check not possible to select from C and then
| | report the error#51 .

(9) execute ⟦ "repeat" S:Statements "until" E:Expression ⟧ =
unfolding
| jumpily execute S and then
| | evaluate E then
| | | | check (the given truth-value is true)
| | | or
| | | | check (the given truth-value is false) then unfold .

(10) execute ⟦ "while" E:Expression "do" S:Statement ⟧ =
unfolding
| evaluate E then
| | | check (the given truth-value is true) then
| | | | jumpily execute S and then unfold
| | or
| | | check (the given truth-value is false) .

(11) execute ⟦ "for" I:Identifier ":=" E_1:Expression T:("to" | "downto")
E_2:Expression "do" S:Statement ⟧ =
| evaluate E_1 and evaluate E_2
then give the control-values of T then
| unfolding
| | | check there is given ( )
| | or
| | | | | assign the first of the given ordinal-value$^+$ to
| | | | | the ordinal-variable bound to I and then
| | | | | jumpily execute S
| | | | and give the rest of the given ordinal-value$^+$
| | | then unfold .
then uninitialize the ordinal-variable bound to I .

(12) execute ⟦ I:Identifier ⟧ =
enact the body-abstraction of the procedure bound to I .

(13) execute ⟦ I:Identifier "(" A:Actual-Parameters ")" ⟧ =
give the formal-parameter-modes of the procedure bound to I then
modally evaluate A then
enact the application of
(the body-abstraction of the procedure bound to I)
to the given argument$^+$.

(14) execute ⟦ "with" V:Variable-Access "do" S:Statement ⟧ =
furthermore
| investigate V then
| produce the field-bindings of the given record-variable
hence jumpily execute S .

privately introduces: the control-values of _ , select from _ , possible to select from _ .

- the control-values of _ :: ("to" | "downto") →
yielder [of ordinal-value*] [using the given (ordinal-value, ordinal-value)] .

(15) the control-values of "to" =
the interval-values of (the given ordinal-value#1, the given ordinal-value#2) .

(16) the control-values of "downto" =
the reverse of
the interval-values of (the given ordinal-value#2, the given ordinal-value#1) .

- select from _ :: Cases →
action [completing | storing | diverging | jumping | erring]
[using the given ordinal-value | current bindings | current storage] .

(17) select from ⟦ C:Constants ":" S:Statement ⟧ =
check (the given ordinal-value is in the case-constant-set of C) then
execute S .

(18) select from ⟨ C_1:Cases ";" C_2:Cases ⟩ = select from C_1 or select from C_2 .

- possible to select from _ :: Cases →
yielder [of a truth-value] [using the given ordinal-value | current bindings] .

(19) possible to select from ⟦ C:Constants ":" S:Statement ⟧ =
the given ordinal-value is in the case-constant-set of C .

(20) possible to select from ⟨ C_1:Cases ";" C_2:Cases ⟩ =
either (possible to select from C_1, possible to select from C_2) .

2.6.2 Executing Labeled Statements

- jumpily execute _ :: Statements →
  action [completing | storing | diverging | jumping | erring]
  [using current bindings | current storage] .

(1) jumpily execute S:Statements =
execute S trap
| unfolding
| | | check not possible to jump into S and then
| | | escape with the given program-point
| | or
| | | jump into S trap unfold .

privately introduces: jump into _ , possible to jump into _ .

- jump into _ :: Statements →
  action [completing | storing | diverging | jumping | erring]
  [using the given program-point | current bindings | current storage] .

(2) jump into ⟨ S_1:Statement ";" S_2:Statements ⟩ =
| jump into S_1 and then execute S_2
or jump into S_2 .

(3) jump into ⟨ L:Label ":" S:Unlabeled-Statement ⟩ =
check (the given program-point is the program-point bound to the spelling of L)
and then execute S .

(4) jump into S:Unlabeled-Statement = fail .

- possible to jump into _ :: Statements →
  yielder [of a truth-value] [using the given program-point | current bindings] .

(5) possible to jump into ⟨ S_1:Statement ";" S_2:Statements ⟩ =
either (possible to jump into S_1, possible to jump into S_2) .

(6) possible to jump into ⟨ L:Label ":" S:Unlabeled-Statement ⟩ =
the given program-point is the program-point bound to the spelling of L .

(7) possible to jump into S:Unlabeled-Statement = false .

2.7 Blocks and Declarations

introduces: activate _ , establish _ , relinquish _ .

- activate _ :: Block → action .
- establish _ :: Label-Declaration-Part? | Constant-Definition-Part? |
  Type-Definition-Part? | Variable-Declaration-Part? |
  (Procedure-Declaration | Function-Declaration)* → action .
- relinquish _ :: Variable-Declaration-Part? → action .

2.7.1 Activating Blocks

- activate _ :: Block $\rightarrow$
  action [completing | storing | diverging | jumping | erring]
  [using current bindings | current storage] .

(1) activate $\llbracket$ L:Label-Declaration-Part$^?$ C:Constant-Definition-Part$^?$
T:Type-Definition-Part$^?$ V:Variable-Declaration-Part$^?$
P:(Procedure-Declaration | Function-Declaration)*
"begin" S:Statements "end" $\rrbracket$ =
furthermore
unbind all (the labels declared by L, the identifiers declared by C,
the identifiers declared by T, the identifiers declared by V,
the identifiers declared by P) before
establish L before establish C before establish T before
establish V before establish P
hence
jumpily execute S thereafter relinquish V .

privately introduces: the labels declared by _ , the identifiers declared by _ ,
the identifiers declared-or-enumerated by _ , the identifiers enumerated by _ .

- the labels declared by _ :: Label-Declaration-Part$^?$ $\rightarrow$ label-token* .

(2) the labels declared by $\llbracket$ "label" L:Labels ";" $\rrbracket$ = the labels declared by L .

(3) the labels declared by $\langle\ \rangle$ = ( ) .

- the labels declared by _ :: Labels $\rightarrow$ label-token* .

(4) the labels declared by L:Label = the spelling of L .

(5) the labels declared by $\langle$ L_1:Label "," L_2:Labels $\rangle$ =
(the labels declared by L_1, the labels declared by L_2) .

- the identifiers declared by _ ::
  Constant-Definition-Part$^?$ | Type-Definition-Part$^?$ | Variable-Declaration-Part$^?$ |
  (Procedure-Declaration | Function-Declaration)* $\rightarrow$ identifier-token* .

(6) the identifiers declared by $\llbracket$ "const" C:Constant-Definition$^+$ $\rrbracket$ = the identifiers declared by C .

(7) the identifiers declared by $\llbracket$ "type" T:Type-Definition$^+$ $\rrbracket$ =
the identifiers declared-or-enumerated by T .

(8) the identifiers declared by $\llbracket$ "var" V:Variable-Declaration$^+$ $\rrbracket$ = the identifiers declared by V .

(9) the identifiers declared by $\langle$ P_1:(Procedure-Declaration | Function-Declaration)
P_2:(Procedure-Declaration | Function-Declaration)$^+$ $\rangle$ =
(the identifiers declared by P_1, the identifiers declared by P_2) .

(10) the identifiers declared by $\langle\ \rangle$ = ( ) .

- the identifiers declared by _ :: Constant-Definition$^+$ $\longrightarrow$ identifier-token$^+$.

(11) the identifiers declared by $[\![$ I:Identifier "=" C:Constant ";" $]\!]$ = I .

(12) the identifiers declared by $\langle$ C_1:Constant-Definition C_2:Constant-Definition$^+$ $\rangle$ =
(the identifiers declared by C_1, the identifiers declared by C_2) .

- the identifiers declared by _ :: Variable-Declaration$^+$ $\longrightarrow$ identifier-token$^+$.

(13) the identifiers declared by $[\![$ I:Identifiers ":" T:Type-Denoter ";" $]\!]$ =
(the identifiers of I, the identifiers enumerated by T) .

(14) the identifiers declared by $\langle$ V_1:Variable-Declaration V_2:Variable-Declaration$^+$ $\rangle$ =
(the identifiers declared by V_1, the identifiers declared by V_2) .

- the identifiers declared by _ :: Procedure-Declaration $\longrightarrow$ identifier-token .

(15) the identifiers declared by $[\![$ "procedure" I:Identifier F:Formal-Parameter-List$^?$ ";"
X:(Block | "forward") ";" $]\!]$ = I .

- the identifiers declared by _ :: Function-Declaration $\longrightarrow$ identifier-token .

(16) the identifiers declared by $[\![$ "function" I_1:Identifier F:Formal-Parameter-List$^?$
":" I_2:Identifier ";" X:(Block | "forward") ";" $]\!]$ = I_1 .

(17) the identifiers declared by $[\![$ "function" I:Identifier ";" B:Block ";" $]\!]$ = I .

- the identifiers declared-or-enumerated by _ :: Type-Definition$^+$ $\longrightarrow$ identifier-token$^+$.

(18) the identifiers declared-or-enumerated by $[\![$ I:Identifier "=" T:Type-Denoter ";" $]\!]$ =
(I, the identifiers enumerated by T) .

(19) the identifiers declared-or-enumerated by $\langle$ T_1:Type-Definition T_2:Type-Definition$^+$ $\rangle$ =
(the identifiers declared-or-enumerated by T_1, the identifiers declared-or-enumerated by T_2) .

- the identifiers enumerated by _ :: Type-Denoter $\longrightarrow$ identifier-token* .

(20) the identifiers enumerated by I:Identifier = ( ) .

(21) the identifiers enumerated by $[\![$ "(" I:Identifiers ")" $]\!]$ = the identifiers of I .

(22) the identifiers enumerated by $[\![$ C_1:Constant ".." C_2:Constant $]\!]$ = ( ) .

(23) the identifiers enumerated by $[\![$ X:"packed"$^?$ "array" "[" T_1:Type-Denoter "]"
"of" T_2:Type-Denoter $]\!]$ =
(the identifiers enumerated by T_1, the identifiers enumerated by T_2) .

(24) the identifiers enumerated by $[\![$ X:"packed"$^?$ "record" F:Field-List "end" $]\!]$ =
the identifiers enumerated by F .

(25) the identifiers enumerated by $[\![$ X:"packed"$^?$ "set" "of" T:Type-Denoter $]\!]$ =
the identifiers enumerated by T .

(26) the identifiers enumerated by $[\![$ X:"packed"$^?$ "file" "of" T:Type-Denoter $]\!]$ =
the identifiers enumerated by T .

(27) the identifiers enumerated by ⟦ "^" I:Identifier ⟧ = ( ) .

- the identifiers enumerated by _ :: Field-List → identifier-token* .

(28) the identifiers enumerated by ⟦ F:Field-Declaration* ⟧ =
the identifiers enumerated by F .

(29) the identifiers enumerated by ⟦ F:Field-Declaration*
"case" I_1:⟨ Identifier ":" ⟩? I_2:Identifier "of" V:Variants ⟧ =
(the identifiers enumerated by F, the identifiers enumerated by V) .

- the identifiers enumerated by _ :: Field-Declaration* → identifier-token* .

(30) the identifiers enumerated by ⟦ I:Identifiers ":" T:Type-Denoter ⟧ =
the identifiers enumerated by T .

(31) the identifiers enumerated by ⟨ F_1:Field-Declaration F_2:Field-Declaration* ⟩ =
(the identifiers enumerated by F_1, the identifiers enumerated by F_2) .

(32) the identifiers enumerated by ⟨ ⟩ = ( ) .

- the identifiers enumerated by _ :: Variants → identifier-token* .

(33) the identifiers enumerated by ⟦ C:Constants "(" F:Field-List ")" ⟧ =
the identifiers enumerated by F .

(34) the identifiers enumerated by ⟨ V_1:Variant ";" V_2:Variants ⟩ =
(the identifiers enumerated by V_1, the identifiers enumerated by V_2) .

2.7.2 Establishing Declarations and Definitions

- establish _ :: Label-Declaration-Part? | Constant-Definition-Part? |
Type-Definition-Part? | Variable-Declaration-Part? |
(Procedure-Declaration | Function-Declaration)* →
action [completing | binding | storing | redirecting]
[using current bindings | current storage | current redirections] .

(1) establish ⟦ "label" L:Labels ";" ⟧ = establish L .

(2) establish ⟦ "const" C:Constant-Definition+ ⟧ = establish C .

(3) establish ⟦ "type" T:Type-Definition+ ⟧ = pre-establish T before re-establish T .

(4) establish ⟦ "var" V:Variable-Declaration+ ⟧ = establish V .

(5) establish ⟨ P_1:(Procedure-Declaration | Function-Declaration)
P_2:(Procedure-Declaration | Function-Declaration)+ ⟩ = establish P_1 before establish P_2 .

(6) establish ⟨ ⟩ = complete .

privately introduces: pre-establish _ , re-establish _ .

2.7.2.1 Establishing Labels

- establish _ :: Labels $\rightarrow$
  action [completing | binding | storing] [using current bindings | current storage] .

(1) establish L:Label = generate a program-point then
bind the spelling of L to the given program-point .

(2) establish ⟨ L_1:Label "," L_2:Labels ⟩ = establish L_1 and establish L_2 .

2.7.2.2 Establishing Constants

- establish _ :: Constant-Definition$^+$ $\rightarrow$
  action [completing | binding] [using current bindings] .

(1) establish ⟦ I:Identifier "=" C:Constant ";" ⟧ = bind I to the constant-value of C .

(2) establish ⟨ C_1:Constant-Definition C_2:Constant-Definition$^+$ ⟩ = establish C_1 before establish C_2 .

2.7.2.3 Establishing Types

- pre-establish _ :: Type-Definition$^+$ $\rightarrow$
  action [completing | binding] [using current bindings] .

(1) pre-establish ⟦ I:Identifier "=" T:Type-Denoter ";" ⟧ = indirectly bind I to unknown .

(2) pre-establish ⟨ T_1:Type-Definition T_2:Type-Definition$^+$ ⟩ = pre-establish T_1 and pre-establish T_2 .

- re-establish _ :: Type-Definition$^+$ $\rightarrow$
  action [completing | redirecting] [using current bindings] .

(3) re-establish ⟦ I:Identifier "=" T:Type-Denoter ";" ⟧ = typify T then redirect I to the given type .

(4) re-establish ⟨ T_1:Type-Definition T_2:Type-Definition$^+$ ⟩ = re-establish T_1 and re-establish T_2 .

2.7.2.4 Establishing Variables

- establish _ :: Variable-Declaration$^+$ $\rightarrow$
  action [completing | binding | storing] [using current bindings | current storage] .

(1) establish ⟦ I:Identifiers ":" T:Type-Denoter ";" ⟧ =
typify T then
respectively allocate variables of (the given type)$^{(\text{count of the identifiers of } I)}$ then
respectively bind the identifiers of I to the given variable$^+$.

(2) establish ⟨ V_1:Variable-Declaration V_2:Variable-Declaration$^+$ ⟩ =
establish V_1 and establish V_2 .

2.7.2.5 Establishing Procedures and Functions

- establish _ :: Procedure-Declaration →
  action [completing | binding | redirecting] [using current bindings] .

(1) establish ⟦ "procedure" I:Identifier "(" F:Formal-Parameter-Sections ")" ";" B:Block ";" ⟧ =
  recursively bind I to
    the procedure of (the modes of the formal-parameters of F,
      the closure of abstraction of
        furthermore respectively associate the formal-parameters of F with the given argument$^+$
        hence
          activate B thereafter
          dissociate all the formal-parameters of F) .

(2) establish ⟦ "procedure" I:Identifier ";" B:Block ";" ⟧ =
    check not (I is known to be bound) then
    recursively bind I to
      the procedure of the closure of abstraction of activate B
  or
    give the forward-procedure bound to I then
    redirect I to
      the procedure of (the modes of the parameters of the given forward-procedure,
        the closure of
          furthermore the abstraction for associating
            the parameters of the given forward-procedure
          hence
            abstraction of activate B thereafter
            the abstraction for dissociating
              the parameters of the given forward-procedure .

(3) establish ⟦ "procedure" I:Identifier "(" F:Formal-Parameter-Sections ")" ";" "forward" ";" ⟧ =
  indirectly bind I to the forward-procedure of the formal-parameters of F .

(4) establish ⟦ "procedure" I:Identifier ";" "forward" ";" ⟧ =
  indirectly bind I to the forward-procedure of ( ) .

- establish _ :: Function-Declaration →
  action [completing | binding | redirecting] [using current bindings] .

(5) establish ⟦ "function" I_1:Identifier ":" I_2:Identifier ";" B:Block ";" ⟧ =
  recursively bind I_1 to
    the function of
      the closure of abstraction of
        furthermore
          allocate a variable of the result-type bound to I_2 then
          bind I_1 to
            the result-variable-inclusion of (the given result-variable, the function bound to I_1)
        hence
            activate B and then
              give the value attributed to the result-variable of the function bound to I_1 then
                give the given result-value or
                check there is given uninitialized then report error#48
          thereafter dispose of the result-variable of the function bound to I_1 .

(6) establish ⟦ "function" I_1:Identifier "(" F:Formal-Parameter-Sections ")"
":" I_2:Identifier ";" Block ";" ⟧ =
recursively bind I_1 to
the function of (the modes of the formal-parameters of F,
the closure of abstraction of
| furthermore
| | allocate a variable of the result-type bound to I_2 then
| | bind I_1 to
| | the result-variable-inclusion of (the given result-variable, the function bound to I_1)
| hence
| | | | furthermore respectively associate
| | | | the formal-parameters of F with the given argument$^+$
| | | hence
| | | | activate B thereafter
| | | | dissociate all the formal-parameters of F
| | | and then
| | | | give the value attributed to the result-variable of the function bound to I_1 then
| | | | | give the given result-value or
| | | | | | check there is given uninitialized then report error#48
| | thereafter dispose of the result-variable of the function bound to I_1) .

(7) establish ⟦ "function" I_1:Identifier ":" I_2:Identifier ";" "forward" ";" ⟧ =
indirectly bind I_1 to
the forward-function of the result-type bound to I_2 .

(8) establish ⟦ "function" I_1:Identifier "(" F:Formal-Parameter-Sections ")"
":" I_2:Identifier ";" "forward" ";" ⟧ =
indirectly bind I_1 to
the forward-function of (the formal-parameters of F, the result-type bound to I_2) .

(9) establish ⟦ "function" I:Identifier ";" B:Block ";" ⟧ =
redirect I to
the function of (the modes of the parameters of the forward-function bound to I,
the closure of
| furthermore
| | the abstraction for including a result-variable of
| | the result-type of the forward-function bound to I
| | in the binding for I
| hence
| | | | furthermore
| | | | | the abstraction for associating
| | | | | the parameters of the forward-function bound to I
| | | | hence
| | | | | abstraction of activate B thereafter
| | | | | the abstraction for dissociating
| | | | | the parameters of the forward-function bound to I
| | | and then abstraction of
| | | | give the value attributed to
| | | | the result-variable of the function bound to I then
| | | | | give the given result-value or
| | | | | | check there is given uninitialized then report error#48
| | thereafter abstraction of
| | dispose of the result-variable of the function bound to I) .

privately introduces: the formal-parameters of _ .

- the formal-parameters of _ :: Formal-Parameter-Sections $\rightarrow$
  yielder [of formal-parameter$^+$] [using current bindings] .

(10) the formal-parameters of $[\![$ I_1:Identifiers ":" I_2:Identifier $]\!]$ =
the formal value-mode parameters of (the identifiers of I_1, the type bound to I_2) .

(11) the formal-parameters of $[\![$ "var" I_1:Identifiers ":" I_2:Identifier $]\!]$ =
the formal variable-mode parameters of (the identifiers of I_1, the type bound to I_2) .

(12) the formal-parameters of $[\![$ "procedure" I:Identifier F:Formal-Parameter-List$^?$ $]\!]$ =
the formal procedural-mode parameter of I .

(13) the formal-parameters of $[\![$ "function" I_1:Identifier F:Formal-Parameter-List$^?$ ":" I_2:Identifier $]\!]$ =
the formal functional-mode parameter of I_1 .

(14) the formal-parameters of $\langle$ F_1:Formal-Parameter-Section ";" F_2:Formal-Parameter-Sections $\rangle$ =
(the formal-parameters of F_1, the formal-parameters of F_2) .

2.7.3 Relinquishing Variables

- relinquish _ :: Variable-Declaration-Part$^?$ $\rightarrow$
  action [completing | storing] [using current bindings | current storage] .

(1) relinquish $[\![$ "var" V:Variable-Declaration$^+$ $]\!]$ = relinquish V .

(2) relinquish $\langle\ \rangle$ = complete .

- relinquish _ :: Variable-Declaration$^+$ $\rightarrow$
  action [completing | storing] [using current bindings | current storage] .

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

(4) relinquish $\langle$ V_1:Variable-Declaration V_2:Variable-Declaration$^+$ $\rangle$ = relinquish V_1 and relinquish V_2 .

2.8 Type-Denoters

introduces: typify _ .

- typify _ :: Type-Denoter $\rightarrow$
  action [giving a type | binding] [using current bindings] .

(1) typify I:Identifier = give the type bound to I .

(2) typify $[\![$ "(" I:Identifiers ")" $]\!]$ =
give the enumerated-type of the count of the identifiers of I then
| respectively bind the identifiers of I to
| the interval-values of
| (minimum of the given enumerated-type, maximum of the given enumerated-type) and
| give the given enumerated-type .

(3) typify $[\![$ C_1:Constant ".." C_2:Constant $]\!]$ =
give the subrange-type of (the constant-value of C_1, the constant-value of C_2) .

(4) typify ⟦ X:"packed"? "array" "[" T_1:Type-Denoter "]" "of" T_2:Type-Denoter ⟧ =
| typify T_1 and typify T_2
then give the array-type of (the given type#2, the given ordinal-type#1) .

(5) typify ⟦ X:"packed"? "record" F:Field-List "end" ⟧ =
record-typify F .

(6) typify ⟦ X:"packed"? "set" "of" T:Type-Denoter ⟧ =
typify T then give the set-type of the given ordinal-type .

(7) typify ⟦ X:"packed"? "file" "of" T:Type-Denoter ⟧ =
typify T then give the file-type of the given type .

(8) typify ⟦ "^" I:Identifier ⟧ = give the pointer-type of the indirection bound to I .

privately introduces: record-typify _ , fields-typify _ , variants-typify _ .

- record-typify _ :: Field-List →
  action [giving a record-type | binding] [using current bindings] .

(9) record-typify ⟦ F:Field-Declaration* ⟧ =
fields-typify F then
give the fixed-record-type of the given field[type]* .

(10) record-typify ⟦ F:Field-Declaration*
"case" I_1:Identifier ":" I_2:Identifier "of" V:Variants ⟧ =
| fields-typify F and
| give the field of (I_1, the ordinal-type bound to I_2) and
| variants-typify V
then give the variant-record-type of the given (field[type]*, field[ordinal-type], variant[type]+) .

(11) record-typify ⟦ F:Field-Declaration*
"case" I:Identifier "of" V:Variants ⟧ =
| fields-typify F and
| give the ordinal-type bound to I and
| variants-typify V
then give the variant-record-type of the given (field[type]*, ordinal-type, variant[type]+) .

- fields-typify _ :: Field-Declaration* →
  action [giving field[type]*] | binding] [using current bindings] .

(12) fields-typify ⟨ ⟩ = give () .

(13) fields-typify ⟦ I:Identifiers ":" T:Type-Denoter ";" ⟧ =
typify T then
give the fields of (the identifiers of I, (the given type)^(the count of the identifiers of I)) .

(14) fields-typify ⟨ F_1:Field-Declaration F_2:Field-Declaration+ ⟩ = fields-typify F_1 and fields-typify F_2 .

- variants-typify _ :: Variants →
  action [giving variant[record-type]+ | binding] [using current bindings] .

(15) variants-typify ⟦ C:Constants ":" "(" F:Field-List ")" ⟧ =
record-typify F then
give the variant of (the case-constant-set of C, the given record-type) .

(16) variants-typify ⟨ V_1:Variant ";" V_2:Variant+ ⟩ = variants-typify V_1 and variants-typify V_2 .

2.9 Programs

introduces: run _ .

- run _ :: Program →
  action [completing | storing | diverging | erring]
  [using the given external-entity* | current storage] .

(1) run ⟦ "program" I:Identifier B:Block "." ⟧ = produce the required-bindings hence activate B .

(2) run ⟦ "program" I_1:Identifier "(" I_2:Identifiers ")" ";"
L:Label-Declaration-Part? C:Constant-Definition-Part?
T:Type-Definition-Part? V:Variable-Declaration-Part?
P:(Procedure-Declaration | Function-Declaration)*
"begin" S:Statements "end" "." ⟧ =
| produce the required-bindings moreover
| externally associate the identifiers of I_2 with the given external-entity+
hence
| furthermore
| | establish L before establish C before establish T before
| | | open the input text-file if appropriate and
| | | open the output text-file if appropriate and
| | | globally establish V
| | before establish P
| hence
| | jumpily execute S thereafter relinquish V .

privately introduces: globally establish _ .

- globally establish _ :: Variable-Declaration-Part? →
  action [completing | binding | storing] [using current bindings | current storage] .

(3) globally establish ⟦ "var" V:Variable-Declaration+ ⟧ = globally establish V .

(4) globally establish ⟨ ⟩ = complete .

- globally establish _ :: Variable-Declaration+ →
  action [completing | binding | storing] [using current bindings | current storage] .

(5) globally establish ⟦ I:Identifiers ":" T:Type-Denoter ";" ⟧ =
typify T then
respectively globally allocate variables of (the given type)$^{(\text{count of the identifiers of } I)}$
for the identifiers of I then
respectively bind the identifiers of I to the given variable+ .

(6) globally establish ⟨ V_1:Variable-Declaration V_2:Variable-Declaration+ ⟩ =
globally establish V_1 and globally establish V_2 .

Chapter 3

Semantic Entities

Data

needs: **Typed Entities, Records, Files, Procedures and Functions, Program Points, External Entities, Errors, Bindings/Tokens, Action Notation/Imperative/Data.**

Typed Entities

Values

needs: **(Primitive Typed Entities, Ordinals, Numbers, Characters)/Values, (Arrays, Records)/Values, Action Notation/Imperative/Data.**

Types

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

Variables

needs: **Typed Entities/(Values, Types), Primitive Typed Entities, Booleans/Values, Arrays, Records, Files, Errors, Action Notation.**

Primitive Typed Entities

Values

needs: **(Ordinals, Reals, Pointers, Sets)/Values.**

Types

needs: **(Ordinals, Reals, Pointers, Sets)/Types.**

Variables

needs: **(Ordinals, Reals, Pointers, Sets)/Variables, Primitive Typed Entities/Types, Action Notation/Imperative/Data.**

Ordinals

Values

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

Types

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

Variables

needs: **Ordinals/Types, Primitive Typed Entities/Variables.**

Booleans

Values

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

Types

needs: **Ordinals/Types.**

Variables

needs: **Booleans/Types, Primitive Typed Entities/Variables.**

Numbers

Values

needs: **(Integers, Reals)/Values, Errors.**

Integers

Values

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

Types

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

Variables

needs: **Integers/Types, Primitive Typed Entities/Variables.**

Reals

Values

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

Types

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

Variables

needs: **Reals/Types, Primitive Typed Entities/Variables.**

Characters

Values

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

Types

needs: **Ordinals/Types, Characters/Values.**

Variables

needs: **Characters/Types, Primitive Typed Entities/Variables.**

Enumerands

Values

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

Types.

Variables

needs: **Enumerands/Types, Primitive Typed Entities/Variables.**

Pointers

Values

needs: **Typed Entities/Variables.**

Types

needs: **Pointers/Values, Typed Entities/Types,**
Action Notation/Directive/Yielders.

Variables

needs: **Pointers/Types, Primitive Typed Entities/Variables.**

Sets

Values

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

Types

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

Arrays

Values

needs: **Typed Entities/Values.**

Strings

needs: **(Characters, Booleans)/Values.**

Types

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

Variables

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

Records

General Records

Values

needs: **(Typed Entities, Fixed Records, Variant Records)/Values, Record Fields.**

Types

needs: **General Records/Values,**
(Typed Entities, Fixed Records, Variant Records)/Types,
Record Fields, Record Variants, Bindings/Tokens.

Variables

needs: **Booleans/Values,**
(Typed Entities, Fixed Records, Variant Records)/Variables,
Record Fields, Errors,
Bindings/Tokens, Action Notation.

Fixed Records

Values

needs: **Typed Entities/Values, Record Fields.**

Types

needs: **Typed Entities/Types, Record Fields, Bindings/Tokens.**

Variables

needs: **Fixed Records/(Values, Types),**
Typed Entities/Variables, Record Fields,
Bindings/Tokens, Action Notation.

Variant Records

Values

needs: **(Typed Entities, Ordinals, General Records)/Values,**
Record Fields, Record Variants.

Types

needs: **(Typed Entities, Ordinals, General Records)/Types,**
Record Fields, Record Variants, Bindings/Tokens.

Variables

needs: **Typed Entities, Booleans/Values, Ordinals/Variables,**
General Records, Fixed Records/Variables,
Variant Records/(Values,Types), Record Fields, Record Variants,
Errors, Bindings/Tokens, Action Notation.

Record Fields

needs: **Typed Entities, Bindings/Tokens.**

Record Variants

needs: **Ordinals/Values, General Records, Bindings/Tokens.**

Files

General Files

Types

needs: (Typed Entities, Plain Files, Text Files)/Types.

Variables

needs: Typed Entities/(Values, Variables), Booleans/Values, General Files/Types, (Plain Files, Text Files)/Variables, Text Fields, Errors, Action Notation.

Plain Files

Types

needs: Typed Entities/Types.

Variables

needs: Typed Entities/(Values, Variables), Plain Files/Types, General Files/Variables, Errors, Action Notation.

Text Files

Types

needs: (General Files, Characters)/Types.

Variables

needs: Booleans/Values, (Integers, Reals, Characters)/(Values, Variables), Text Files/Types, General Files/Variables, Text Fields, Decimal Conversions, Errors, Action Notation.

Text Fields

needs: (Booleans, Integers, Reals, Characters)/Values, Arrays/Values/Strings, Decimal Conversions, Errors.

Character Fields.

Integer Fields.

Real Fields.

Boolean Fields.

String Fields.

Decimal Conversions

needs: (Integers, Reals, Characters)/Values.

Procedures and Functions

Procedures

needs: **Parameter Modes, Action Notation.**

Functions

needs: **Parameter Modes, Function Results, Action Notation.**

Function Results

needs: **Ordinals, Reals, Pointers.**

Arguments

needs: **Typed Entities/(Values, Variables), Procedures, Functions, Program Points, Errors.**

Parameter Modes.

Formal Parameters

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

Parameter Association

needs: **Typed Entities, Parameter Modes, Arguments, Formal Parameters, Bindings/Tokens, Action Notation.**

Forward Procedures

needs: **Formal Parameters.**

Forward Functions

needs: **Formal Parameters, Function Results.**

Program Points

needs: **Action Notation.**

External Entities

needs: **Typed Entities, Files/Text Files, Bindings/Tokens, Action Notation.**

Errors

needs: **Data, Action Notation.**

Bindings

needs: **Typed Entities, Procedures and Functions, Program Points, External Entities.**

Tokens.

Required

needs: **Ordinals, Numbers, Characters, Pointers, Arrays, Files, Procedures and Functions, Errors.**

3.1 Data

- datum = value | uninitialized | type | variable |
  field | variant | file-mode | end-of-line | end-of-page | text-field |
  procedure | function | forward-procedure | forward-function |
  parameter-mode | formal-parameter |
  program-point | external-entity | error |
  token .

3.2 Typed Entities

3.2.1 Values

introduces: value , constant-value , literal-value ,
putative _ .

- value = primitive-value | array-value | record-value $(disjoint)$.
- constant-value = ordinal-value | number | string-value .
- literal-value = number | character | string-value .
- putative ($v \leq$ value) = v | uninitialized $(disjoint)$.

3.2.2 Types

introduces: type , nonfile-type ,
_ made assignment-compatible with _ .

- type = nonfile-type | file-type $(disjoint)$.
- nonfile-type = primitive-type | array-type | record-type $(disjoint)$.
- _ made assignment-compatible with _ :: value, nonfile-type $\rightarrow$ value | error $(partial)$.

(1) v:value made assignment-compatible with t:nonfile-type =
the ordinal-value yielded by v made assignment-compatible with the ordinal-type yielded by t |
the number yielded by v made assignment-compatible with the real-type yielded by t |
the set-value yielded by v made assignment-compatible with the set-type yielded by t |
the pointer-value yielded by v |
the array-value yielded by v |
the record-value yielded by v .

3.2.3 Variables

introduces: variable , nonfile-variable ,
type possessed by _ , _ is _ ,
assign _ to _ , the value attributed to _ , attribute _ to _ ,
uninitialize _ , allocate a variable of _ , dispose of _ ,
the values respectively attributed to _ , respectively attribute _ to _ ,
uninitialize all _ , respectively allocate variables of _ , dispose of all _ .

- variable = nonfile-variable | file-variable $(disjoint)$.
- nonfile-variable = primitive-variable | array-variable | record-variable $(disjoint)$.

- type possessed by _ :: variable → type (*total*) .

(1) the type possessed by v:variable =
  the type possessed by the primitive-variable yielded by v **|**
  the type possessed by the array-variable yielded by v **|**
  the type possessed by the record-variable yielded by v **|**
  the type possessed by the file-variable yielded by v .

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

(2) v_1:variable is v_2:variable =
  the cell of the primitive-variable yielded by v_1 is
    the cell of the primitive-variable yielded by v_2 **|**
  the array-variable yielded by v_1 is the array-variable yielded by v_2 **|**
  the record-variable yielded by v_1 is the record-variable yielded by v_2 **|**
  the file-variable yielded by v_1 is the file-variable yielded by v_2 .

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

(3) assign v_1:yielder [of a value] to v_2:yielder [of a nonfile-variable] =
  | give (v_1 made assignment-compatible with the type possessed by v_2)
  | reporting any given error
  then attribute the given value to v_2 .

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

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

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

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

(6) 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 or
  uninitialize the file-variable yielded by v .

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

(7) 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, t)
  or allocate a variable of the array-type yielded by t
  or allocate a variable of the record-type yielded by t
  or allocate a variable of the file-type yielded by t .

- dispose of _ :: yielder [of a variable] → action [storing] .

(8) 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
dispose of the file-variable yielded by v .

- the values respectively attributed to _ :: yielder [of nonfile-variable*] → yielder [of (putative value)*] [using current storage] .

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

- respectively attribute _ to _ :: yielder [of value*], yielder [of nonfile-variable*] → action [storing] .

(10) respectively attribute v_1:yielder [of value*] to v_2:yielder [of nonfile-variable*] =
| check both (v_1 is (), v_2 is ())
or
| attribute the first of v_1 to the first of v_2 and
| respectively attribute the rest of v_1 to the rest of v_2 .

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

(11) 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*] → action [giving variable* | storing] [using current storage] .

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

- dispose of all _ :: yielder [of variable*] → action [storing] .

(13) 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

introduces: primitive-value .

- primitive-value = ordinal-value | real-number | pointer-value | set-value (*disjoint*) .

3.3.2 Types

introduces: primitive-type .

- primitive-type = ordinal-type | real-type | pointer-type | set-type (*disjoint*) .

3.3.3 Variables

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

- primitive-variable = ordinal-variable | real-variable | pointer-variable | set-variable (*disjoint*) .
- primitive-variable of _ :: (cell, primitive-type) $\rightarrow$ primitive-variable (*total*, *injective*) .
- cell _ :: primitive-variable $\rightarrow$ cell (*total*) .
- type possessed by _ :: primitive-variable $\rightarrow$ primitive-type (*total*) .

(1) v = the primitive-variable of (c:cell, t:primitive-type) $\Rightarrow$

(1) the cell of v = c ;

(2) the type possessed by v = t .

- storable = primitive-value | $\square$.

3.4 Ordinals

3.4.1 Values

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

- ordinal-value = truth-value | integer-number | character | enumerated-value (*disjoint*) .
- comparable-ordinals = truth-value^2 | integer-number^2 | character^2 |
comparable-enumerated-values (*disjoint*) .
- ordinal-number _ :: ordinal-value $\rightarrow$ integer-number (*total*) .
- successor _ , predecessor _ :: ordinal-value $\rightarrow$ ordinal-value | error (*total*) .

(1) (o_1, o_2) : comparable-ordinals ;
ordinal-number of o_1 = integer-number of i ;
ordinal-number of o_2 = integer-number of successor of i $\Rightarrow$

(1) o_2 = successor of o_1 ;

(2) predecessor of o_2 = o_1 .

(2) successor of maximum of t:ordinal-type = error#38 .

(3) predecessor of minimum of t:ordinal-type = error#39 .

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

(4) (o_1, o_2) : comparable-ordinals $\Rightarrow$

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

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

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

(4) interval-values of (o_1, o_2) =
if o_1 is greater than o_2 then ( ) else
$(o_1$, interval-values of (successor of o_1, o_2)) .

3.4.2 Types

introduces: ordinal-type , minimum _ , maximum _ , cardinality _ ,
subrange-type , subrange-type of _ , host-type _ .

- ordinal-type = boolean-type | integer-type | character-type |
enumerated-type | subrange-type $(disjoint)$.

- minimum _ , maximum _ :: ordinal-type $\rightarrow$ ordinal-value $(total)$.

- cardinality _ :: ordinal-type $\rightarrow$ positive-integer $(total)$.

(1) ordinal-number of minimum of t = integer-number of i_1:integer ;
ordinal-number of maximum of t = integer-number of i_2:integer $\Rightarrow$
cardinality of t:ordinal-type = successor of difference of (i_2, i_1) .

- _ made assignment-compatible with _ :: ordinal-value, ordinal-type $\rightarrow$ ordinal-value | error $(partial)$.

(2) o:ordinal-value made assignment-compatible with t:ordinal-type =
if either (o is less than the minimum of t, o is greater than the maximum of t)
then error#49
else o .

- subrange-type = boolean-subrange-type | integer-subrange-type |
character-subrange-type | enumerated-subrange-type $(disjoint)$.

- subrange-type of _ :: comparable-ordinals $\rightarrow$ subrange-type $(partial)$.

(3) (o_1, o_2) : comparable-ordinals ; o_1 is greater than o_2 = false $\Rightarrow$
subrange-type of (o_1, o_2) : subrange-type .

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

(1) minimum of $t = o_1$;

(2) maximum of $t = o_2$.

- host-type _ :: ordinal-type $\rightarrow$ boolean-type | integer-type | character-type | enumerated-type $(total)$.

(5) host-type of (t:(boolean-type | integer-type | character-type | enumerated-type)) = t .

3.4.3 Variables

introduces: ordinal-variable .

- primitive-variable of _ :: (cell, ordinal-type) $\rightarrow$ ordinal-variable $(partial)$.

3.5 Booleans

3.5.1 Values

(1) ordinal-number of false = integer-number of 0 .

(2) ordinal-number of true = integer-number of 1 .

3.5.2 Types

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

- boolean-type : ordinal-type .

(1) minimum of boolean-type = false .

(2) maximum of boolean-type = true .

- subrange-type of _ :: truth-value^2 $\rightarrow$ boolean-subrange-type ($partial$) .

(3) host-type of t:boolean-subrange-type = boolean-type .

3.5.3 Variables

introduces: boolean-variable .

- primitive-variable of _ :: (cell, boolean-type) $\rightarrow$ boolean-variable ($partial$) .

3.6 Numbers

3.6.1 Values

introduces: number , negation _ , absolute-value _ , square _ , sum _ , difference _ , product _ ,
_ is _ , _ is less than _ , _ is greater than _ .

- number = integer-number | real-number ($disjoint$) .
- negation _ , absolute-value _ , square _ :: number $\rightarrow$ number | error ($total$) .
- sum _ , difference _ , product _ :: (number, number) $\rightarrow$ number | error ($total$) .
- _ is _ , _ is less than _ , _ is greater than _ :: number, number $\rightarrow$ truth-value ($total$) .

3.7 Integers

3.7.1 Values

introduces: integer-number , integer-number of _ ,
min-implemented-integer , max-implemented-integer , maxint-value ,
negation _ , absolute-value _ , square _ ,
sum _ , difference _ , product _ ,
integer-quotient _ , modulo _ ,
_ is _ , _ is less than _ , _ is greater than _ .

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

- min-implemented-integer , max-implemented-integer : integer .

(1) integer-number of _ ::
    integer [min min-implemented-integer] [max max-implemented-integer]
        $\rightarrow$ integer-number $(total,\ injective)$,
    integer [min successor of max-implemented-integer] $\rightarrow$ error#47 $(total)$,
    integer [max successor of min-implemented-integer] $\rightarrow$ error#47 $(total)$.

(2) ordinal-number of i:integer-number $= i$.

- maxint-value : integer-number .

(3) maxint-value is less than integer-number of 0 = false .
Note: maxint = 0 is consistent with PS!

- negation _ , absolute-value _ , square _ :: integer-number $\rightarrow$ integer-number | error $(total)$.

(4) $n =$ (integer-number of i:integer) : integer-number $\Rightarrow$

    (1) negation of $n =$ integer-number of negation of i ;
    (2) absolute-value of $n =$ integer-number of absolute i but error#47 ;
    (3) square of $n =$ integer-number of product of $(i,\ i)$ but error#32 .

- sum _ , difference _ , product _ , integer-quotient _ , modulo _ ::
    (integer-number, integer-number) $\rightarrow$ integer-number | error $(total)$.

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

(5) $n_1 =$ (integer-number of i_1:integer) : integer-number ;
$n_2 =$ (integer-number of i_2:integer) : integer-number $\Rightarrow$

    (1) sum of $(n_1,\ n_2) =$ integer-number of sum of $(i_1,\ i_2)$;
    (2) difference of $(n_1,\ n_2) =$ integer-number of difference of $(i_1,\ i_2)$;
    (3) product of $(n_1,\ n_2) =$ integer-number of product of $(i_1,\ i_2)$;
    (4) integer-quotient of $(n_1,\ n_2) =$
        if i_2 is 0 then error#45 else
        integer-number of integer-quotient of $(i_1,\ i_2)$;
    (5) modulo of $(n_1,\ n_2) =$
        if not (i_2 is greater than 0) then error#46 else
        integer-number of modulo of $(i_1,\ i_2)$;
    (6) n_1 is $n_2 = i_1$ is i_2 ;
    (7) n_1 is less than $n_2 = i_1$ is less than i_2 ;
    (8) n_1 is greater than $n_2 = i_1$ is greater than i_2 .

3.7.2 Types

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

- integer-type : ordinal-type .

(1) minimum of integer-type = integer-number of min-implemented-integer .

(2) maximum of integer-type = integer-number of max-implemented-integer .

- subrange-type of _ :: integer-number^2 $\rightarrow$ integer-subrange-type $(partial)$.

(3) host-type of t:integer-subrange-type = integer-type .

3.7.3 Variables

introduces: integer-variable .

- primitive-variable of _ :: (cell, integer-type) $\rightarrow$ integer-variable ($partial$) .

3.8 Reals

3.8.1 Values

introduces: real-number , real-number of _ , negation _ , absolute-value _ , square _ ,
sum _ , difference _ , product _ , quotient _ ,
sine _ , cosine _ , arctangent _ ,
exponential _ , natural-logarithm _ , square-root _ ,
truncated _ , rounded _ , floated _ ,
_ is _ , _ is less than _ , _ is greater than _ .

privately introduces: approximated _ .

- real-number of _ :: rational $\rightarrow$ real-number ($total$) .

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

- approximated _ :: rational $\rightarrow$ rational ($total$) .

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

(3) approximated 0 = 0 .

(4) r_1 is greater than r_2 = false $\Rightarrow$
approximated r_1:rational is greater than approximated r_2:rational = false .

- negation _ , absolute-value _ , square _ :: real-number $\rightarrow$ real-number ($total$) .

(5) n = real-number of r:approximated rational $\Rightarrow$

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

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

  (3) square of n = real-number of product of (r, r) .

- sum _ , difference _ , product _ ::
  (real-number, real-number) $\rightarrow$ real-number ($total$) ,
  (integer-number, real-number) $\rightarrow$ real-number ($total$) ,
  (real-number, integer-number) $\rightarrow$ real-number ($total$) .

- quotient _ :: (number, number) $\rightarrow$ real-number | error ($total$) .

(6) n_1 = real-number of r_1:approximated rational ;
n_2 = real-number of r_2:approximated rational $\Rightarrow$

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

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

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

  (4) quotient of (n_1, n_2) =
    if n_2 is real-number of 0 then error#44 else
    real-number of quotient of (r_1, r_2) .

(7) n_1 : real-number ; n_2 : integer-number $\Rightarrow$

(1) sum of (n_1, n_2) = sum of $(n_1$, floated $n_2)$;
sum of (n_2, n_1) = sum of (floated $n_2, n_1)$;

(2) difference of (n_1, n_2) = difference of $(n_1$, floated $n_2)$;
difference of (n_2, n_1) = difference of (floated $n_2, n_1)$;

(3) product of (n_1, n_2) = product of $(n_1$, floated $n_2)$;
product of (n_2, n_1) = product of (floated $n_2, n_1)$;

(4) quotient of (n_1, n_2) = quotient of $(n_1$, floated $n_2)$;
quotient of (n_2, n_1) = quotient of (floated $n_2, n_1)$.

(8) quotient of (n_1:integer-number, n_2:integer-number) =
quotient of (floated n_1, floated n_2) .

- sine _ , cosine _ , arctangent _ , exponential _ :: number → real-number $(total)$.

- natural-logarithm _ :: number → real-number | error $(total)$.

(9) n is greater than real-number of 0 = false $\Rightarrow$
natural-logarithm of n:real-number = error#33 .

- square-root _ :: number → real-number | error $(total)$.

(10) n is less than real-number of 0 = true $\Rightarrow$
square-root of n:real-number = error#34 .

Note: We really ought to specify the exact mathematical trigonometric functions, and insist that the implementations are approximations of them!

- truncated _ , rounded _ :: real-number → integer-number | error $(total)$.

(11) truncated real-number of r:approximated rational =
integer-number of truncated r but error#35 .

(12) rounded n:real-number =
(if n is less than real-number of 0
then truncated difference (n, real-number of approximated quotient (1, 2))
else truncated sum (n, real-number of approximated quotient (1, 2)))
but error#36 .

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

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

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

(14) n_1 = real-number of r_1:approximated rational ;
n_2 = real-number of r_2:approximated rational $\Rightarrow$

(1) n_1 is n_2 = r_1 is r_2 ;

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

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

(15) n_1 : real-number ; n_2 : integer-number $\Rightarrow$

(1) n_1 is n_2 = n_1 is floated n_2 ;
n_2 is n_1 = floated n_2 is n_1 ;

(2) n_1 is less than n_2 = n_1 is less than floated n_2 ;
n_2 is less than n_1 = floated n_2 is less than n_1 ;

(3) n_1 is greater than n_2 = n_1 is greater than floated n_2 ;
n_2 is greater than n_1 = floated n_2 is greater than n_1 .

3.8.2 Types

introduces: real-type.

- real-type : primitive-type .
- _ made assignment-compatible with _ :: number, real-type → real-number (*total*) .

(1) i:integer-number made assignment-compatible with t:real-type = floated i .

(2) r:real-number made assignment-compatible with t:real-type = r .

3.8.3 Variables

introduces: real-variable .

- primitive-variable of _ :: (cell, real-type) → real-variable (*partial*) .

3.9 Characters

3.9.1 Values

includes: **Data Notation/Characters/Alphanumerics**
(character , character of _ , code _) .

introduces: maximum-character-code , space , apostrophe-image-character .

- character of _ :: natural [max maximum-character-code] → character (*total*, *injective*) .
- character of _ :: integer-number → character | error (*total*) .

(1) code of character = natural [max maximum-character-code] .

(2) ordinal-number of c:character = integer-number of code of c .

(3) interval-values of ('0', '9') = ('0', '1', '2', '3', '4', '5', '6', '7', '8', '9') .

(4) character of integer-number of i:integer =
if either (i is less than 0, i is greater than maximimum-character-code)
then error#37
else character of i .

- maximum-character-code : natural .
- space = ' ' : character .
- apostrophe-image-character : character .

3.9.2 Types

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

- character-type : ordinal-type .

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

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

- subrange-type of _ :: character^2 $\rightarrow$ character-subrange-type ($partial$) .

(3) host-type of t:character-subrange-type = character-type .

3.9.3 Variables

introduces: character-variable .

- primitive-variable of _ :: (cell, character-type) $\rightarrow$ character-variable ($partial$) .

3.10 Enumerands

3.10.1 Values

introduces: enumerated-value , comparable-enumerated-values , enumerated-value of _ .

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

(1) e_1 = enumerated-value of (n_1:natural, p:positive-integer) : enumerated-value ;
e_2 = enumerated-value of (n_2:natural, p:positive-integer) : enumerated-value $\Rightarrow$
(e_1, e_2) : comparable-enumerated-values .

- enumerated-value of _ :: (natural, positive-integer) $\rightarrow$ enumerated-value ($partial$) .

(2) n:natural is less than p:positive-integer = true $\Rightarrow$

  (1) enumerated-value of (n, p) : enumerated-value ;

  (2) ordinal-number of enumerated-value of (n, p) = integer-number of n .

3.10.2 Types

introduces: enumerated-type , enumerated-type of _ ,
enumerated-subrange-type , enumerated-value-type _ .

- enumerated-type of _ :: positive-integer $\rightarrow$ enumerated-type ($partial$) .

(1) t = enumerated-type of (p:positive-integer) $\Rightarrow$

  (1) minimum of t = enumerated-value of (0, p) ;

  (2) maximum of t = enumerated-value of (predecessor of p, p) .

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

(2) (e_1, e_2) : comparable-enumerated-values ;
t = subrange-type of (e_1, e_2) : enumerated-subrange-type $\Rightarrow$
host-type of t = enumerated-value-type of e_1 .

- enumerated-value-type _ :: enumerated-value → enumerated-type (*total*) .

(3) e = enumerated-value of (n:natural, p:positive-integer) : enumerated-value ⇒
enumerated-value-type of e = enumerated-type of p .

3.10.3 Variables

introduces: enumerated-variable .

- primitive-variable of _ :: (cell, enumerated-type) → enumerated-variable (*partial*) .

3.11 Pointers

3.11.1 Values

introduces: pointer-value , nil-pointer , pointer to _ , target-variable _ , _ is _ .

- pointer-value = nil-pointer | pointer to variable (*disjoint*) .
- nil-pointer : pointer-value .
- pointer to _ :: variable → pointer-value (*total*) .
- target-variable _ :: pointer-value → variable | error (*total*) .

(1) target-variable of nil-pointer = error#3 .

(2) target-variable of pointer to v:variable = v .

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

(3) nil-pointer is nil-pointer = true .

(4) nil-pointer is pointer to v:variable = false .

(5) pointer to v_1:variable is pointer to v_2:variable = v_1 is v_2 .

3.11.2 Types

introduces: pointer-type , pointer-type of _ , target-type _ .

- pointer-type of _ :: indirection → pointer-type (*total*) .
- target-type _ :: yielder [of a pointer-type] → yielder [of a type] (*total*) .

(1) target-type of t:yielder [of a pointer-type] =
the type yielded by (the current redirections at the target-type-indirection of t) .

privately introduces: target-type-indirection _ .

- target-type-indirection _ :: pointer-type → indirection (*total*) .

(2) target-type-indirection of pointer-type of i:indirection = i .

3.11.3 Variables

introduces: pointer-variable .

- primitive-variable of _ :: (cell, pointer-type) → pointer-variable (*partial*) .

3.12 Sets

3.12.1 Values

introduces: set-value .

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

3.12.2 Types

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

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

  Note: This totality requires 'set-type of integer-type' to be supported!

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

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

- _ made assignment-compatible with _ :: set-value, set-type → set-value **|** error (*total*) .

(2) s:set-value made assignment-compatible with t:set-type =
    if not (s is included in the set of the interval-values of
        (the minimum of the base-type of t, the maximum of the base-type of t))
    then the error#50
    else s .

3.12.3 Variables

introduces: set-variable .

- primitive-variable of _ :: (cell, set-type) → set-variable (*partial*) .

3.13 Arrays

3.13.1 Values

introduces: array-value , array-value of _ , components _ .

- array-value of _ :: value$^+$ $\rightarrow$ array-value ($partial$) ,
  (putative value)$^+$ $\rightarrow$ putative array-value ($partial$) .

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

- components _ :: array-value $\rightarrow$ value$^+$ ($total$) .

(2) the components of the array-value of v:value$^+$ = v .

3.13.1.1 Strings

introduces: string-value , string-value of _ ,
_ is _ , _ is lexicographically less than _ , _ is lexicographically greater than _ .

- string-value = array-value of (character, character$^+$) .
- string-value of _ :: (character, character$^+$) $\rightarrow$ string-value ($total$, $injective$) .

(1) the string-value of c:(character, character$^+$) =
the array-value of c .

- _ is _ :: string-value, string-value $\rightarrow$ truth-value ($partial$) .

(2) s_1:string-value is s_2:string-value =
the components of s_1 is the components of s_2 .

- _ is lexicographically less than _ :: string-value, string-value $\rightarrow$ truth-value ($partial$) .

(3) s_1:string-value is lexicographically less than s_2:string-value =
the components of s_1 lexicographically precede the components of s_2 .

- _ is lexicographically greater than _ :: string-value, string-value $\rightarrow$ truth-value ($partial$) .

(4) s_1:string-value is lexicographically greater than s_2:string-value =
the components of s_2 lexicographically precede the components of s_1 .

privately introduces: _ lexicographically precede _ .

- _ lexicographically precede _ :: character*, character* $\rightarrow$ truth-value ($total$) .

(5) () lexicographically precede c:character$^+$ = true .

(6) c:character* lexicographically precede () = false .

(7) c_1:character$^+$ lexicographically precede c_2:character$^+$ =
if the first of c_1 is less than the first of c_2
then true
else if the first of c_1 is greater than the first of c_2
then false
else the rest of c_1 lexicographically precede the rest of c_2 .

3.13.2 Types

introduces: array-type , array-type of _ ,
component-type _ , index-type _ .

- array-type of _ :: (type, ordinal-type) → array-type (*total*, *injective*) .
- component-type _ :: array-type → type (*total*) .
- index-type _ :: array-type → ordinal-type (*total*) .

(1) t = the array-type of (t_1:type, t_2:ordinal-type) : array-type ⇒

  (1) the component-type of t = t_1 ;

  (2) the index-type of t = t_2 .

3.13.3 Variables

introduces: array-variable , array-variable of _ ,
components _ ,
type possessed by _ , _ is _ ,
component indexed by _ in _ , _ components indexed from _ in _ ,
the value attributed to _ , attribute _ to _ ,
uninitialize _ , allocate a variable of _ , dispose of _ .

- array-variable of _ :: (variable$^+$, array-type) → array-variable (*partial*) .
- components _ :: array-variable → variable$^+$ (*total*) .
- type possessed by _ :: array-variable → array-type (*total*) .

(1) a = the array-variable of (v:variable$^+$, t:array-type) : array-variable ⇒

  (1) the components of a = v ;

  (2) the type possessed by a = t .

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

(2) a_1:array-variable is a_2:array-variable =
  the first of the components of a_1 is the first of the components of a_2 .

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

(3) t = the index-type of the type possessed by a ;
the integer-number of m = the ordinal-number of the minimum of t ;
the integer-number of n = the ordinal-number of the maximum of t ;
the integer-number of i = the ordinal-number of o ;
k = the successor of the difference of (i, m)
⇒
the component indexed by o:ordinal-value in a:array-variable =
  if either (i is less than m, i is greater than n)
  then the error#1
  else the component #k of the components of a .

- _ components indexed from _ in _ ::
  positive-integer, ordinal-value, array-variable → variable$^+$ | error (*partial*) .

(4) t = the index-type of the type possessed by a ;
the integer-number of m = the ordinal-number of the minimum of t ;
the integer-number of n = the ordinal-number of the maximum of t ;
the integer-number of i = the ordinal-number of o ;
j = the sum of (i, the predecessor of p) ;
k = the successor of the difference of (i, m) ;
l = the successor of the difference of (j, m)
$\Rightarrow$
p:positive-integer components indexed from o:ordinal-value in a:array-variable =
    if either (i is less than m, i is greater than n)
    then the error#26
    else if j is greater than n
    then the error#28
    else the components from #k to #l of the components of a.

Note that the errors depend on whether this operation is invoked from "pack" or "unpack"!

- the value attributed to _ :: yielder [of an array-variable] $\rightarrow$
    yielder [of a putative array-value] [using current storage] .

(5) the value attributed to a:yielder [of an array-variable] =
    the array-value of the values respectively attributed to the components of a .

- attribute _ to _ :: yielder [of an array-value], yielder [of an array-variable] $\rightarrow$ action [storing] .

(6) attribute a_1:yielder [of an array-value] to a_2:yielder [of an array-variable] =
    respectively attribute the components of a_1 to the components of a_2 .

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

(7) uninitialize a:yielder [of an array-variable] =
    uninitialize all the components of a .

- allocate a variable of _ :: yielder [of an array-type] $\rightarrow$
    action [giving an array-variable | storing] [using current storage] .

(8) allocate a variable of t:yielder [of an array-type] =
    respectively allocate variables of
        (the component-type of t)$^{\text{(the cardinality of the index-type of } t\text{)}}$ then
    give the array-variable of (the given variable$^+$, t) .

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

(9) dispose of a:yielder [of an array-variable] =
    dispose of all the components of a .

3.14 Records

3.14.1 General Records

3.14.1.1 Values

introduces: record-value , fixed-part _ .

- record-value = fixed-record-value | variant-record-value (*disjoint*) .
- fixed-part _ :: record-value → field[value]* (*total*) .

3.14.1.2 Types

introduces: record-type , fixed-part _ ,
component-type designated by _ in _ , _ restricted by _ .

- record-type = fixed-record-type | variant-record-type (*disjoint*) .
- fixed-part _ :: record-type → field[type]* (*total*) .
- component-type designated by _ in _ :: identifier-token, record-type → type (*partial*) .
- _ restricted by _ :: record-type, ordinal-value* → record-type (*partial*) .

(1) t:record-type restricted by () = t .

(2) t:fixed-record-type restricted by o:ordinal-value$^+$ = nothing .

(3) v = (the variant selected by the first of o in the variant-part of t) : variant[type] ⇒
t:variant-record-type restricted by o:ordinal-value$^+$ =
the variant-record-type of (
the fixed-part of t,
the selector-part of t,
the variant of (the selector-set of v,
the sub-record of v restricted by the rest of o)) .

3.14.1.3 Variables

introduces: record-variable , fixed-part _ ,
_ is _ , field-bindings _ ,
access the component designated by _ in _ .

- record-variable = fixed-record-variable | variant-record-variable (*disjoint*) .
- fixed-part _ :: record-variable → field[variable]* (*total*) .
- _ is _ :: record-variable, record-variable → truth-value (*partial*) .
- field-bindings _ :: record-variable → map [identifier-token to variable] (*total*) .

(1) r_1:record-variable is r_2:record-variable =
the fixed-part of r_1 is the fixed-part of r_2 .

This definition is wrong, yielding true if both record-variables have empty fixed-parts. But even a record-variable with no fields at all has its own identity!

- access the component designated by _ in _ ::
yielder [of an identifier-token], yielder [of a record-variable] →
action [giving a variable | erring | storing] [using current storage] .

3.14.2 Fixed Records

3.14.2.1 Values

introduces: fixed-record-value , fixed-record-value of _ , fixed-part _ .

- fixed-record-value of _ :: field[value]* → fixed-record-value (*total, injective*) ,
  field[putative value]* → putative fixed-record-value (*total, injective*) .

(1) the fixed-record-value of (f_1:field[putative value]*, f_2:field[uninitialized], f_3:field[putative value]*) =
uninitialized .

- fixed-part _ :: fixed-record-value → field[value]* (*total*) .

(2) r = the fixed-record-value of f:field[value]* : fixed-record-value ⇒
the fixed-part of $r = f$.

3.14.2.2 Types

introduces: fixed-record-type , fixed-record-type of _ , fixed-part _ ,
component-type designated by _ in _.

- fixed-record-type of _ :: field[type]* → fixed-record-type (*partial*) .
- fixed-part _ :: fixed-record-type → field[type]* (*total*) .

(1) t = the fixed-record-type of f:field[type]* : fixed-record-type ⇒
the fixed-part of $t = f$.

- component-type designated by _ in _ :: identifier-token, fixed-record-type → type (*partial*) .

(2) the component-type designated by i:identifier-token in t:fixed-record-type =
the type yielded by the component of the field identified by i in the fixed-part of t .

3.14.2.3 Variables

introduces: fixed-record-variable , fixed-record-variable of _ ,
fixed-part _ , type possessed by _ ,
field-bindings _ , access the component designated by _ in _ ,
the value attributed to _ , attribute _ to _ ,
uninitialize _ , allocate a variable of _ , dispose of _ .

- fixed-record-variable of _ :: (field[variable]*, fixed-record-type) → fixed-record-variable (*partial*) .
- fixed-part _ :: fixed-record-variable → field[variable]* (*total*) .
- type possessed by _ :: fixed-record-variable → fixed-record-type (*total*) .

(1) r = the fixed-record-variable of (f:field[variable]*, t:fixed-record-type) : fixed-record-variable ⇒

  (1) the fixed-part of $r = f$;

  (2) the type possessed by $r = t$.

- field-bindings _ :: fixed-record-variable → map [identifier-token to variable] (*total*) .

(2) the field-bindings of r:fixed-record-variable =
the field-bindings of the fixed-part of r .

- field-bindings _ :: field[variable]* → map[identifier-token to variable] (*partial*) .

(3) the field-bindings of () = the empty-map .

(4) the field-bindings of f:field[variable]$^+$ =
the disjoint-union of (the map of the identifier of the first of f to
the component of the first of f,
the field-bindings of the rest of f) .

- access the component designated by _ in _ ::
yielder [of an identifier-token], yielder [of a fixed-record-variable] → action [giving a variable] .

(5) access the component designated by i:yielder [of an identifier-token]
in r:yielder [of a fixed-record-variable] =
give the component of the field identified by i in the fixed-part of r .

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

(6) the value attributed to r:yielder [of a fixed-record-variable] =
the fixed-record-value of
the fields of (the identifiers of the fixed-part of r,
the values respectively attributed to the components of the fixed-part of r) .

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

(7) attribute r_1:yielder [of a fixed-record-value] to r_2:yielder [of a fixed-record-variable] =
respectively attribute the components of the fixed-part of r_1 to
the components of the fixed-part of r_2 .

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

(8) uninitialize r:yielder [of a fixed-record-variable] =
uninitialize all the components of the fixed-part of r .

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

(9) allocate a variable of t:yielder [of a fixed-record-type] =
respectively allocate variables of
the type* yielded by the components of the fixed-part of t then
give the fixed-record-variable of (
the fields of (the identifiers of the fixed-part of t, the given variable*),
t) .

- dispose of _ :: yielder [of a fixed-record-variable] → action [storing] .

(10) dispose of r:yielder [of a fixed-record-variable] =
dispose of all the components of the fixed-part of r .

3.14.3 Variant Records

3.14.3.1 Values

introduces: variant-record-value , tagged-record-value , untagged-record-value ,
variant-record-value of _ ,
fixed-part _ , variant-part _ , selector-part _ , selector _ .

- variant-record-value = tagged-record-value | untagged-record-value (*disjoint*) .

- variant-record-value of _ ::
  (field[value]*, field[ordinal-value], variant[record-value]) → tagged-record-value (*total*, *injective*) ,
  (field[value]*, ordinal-value, variant[record-value]) → untagged-record-value (*total*, *injective*) ,
  (field[putative value]*, field[putative ordinal-value], variant[putative record-value]) →
  putative tagged-record-value (*total*, *injective*) ,
  (field[putative value]*, putative ordinal-value, variant[putative record-value]) →
  putative untagged-record-value (*total*, *injective*) .

(1) the variant-record-value of (f_1:field[putative value]*, f_2:field[uninitialized], f_3:field[putative value]*,
s:(field[putative ordinal-value] | putative ordinal-value),
v:variant[putative record-value]) =
uninitialized .

(2) the variant-record-value of (f:field[putative value]*,
s:(field[uninitialized] | uninitialized),
v:variant[putative record-value]) =
uninitialized .

(3) the variant-record-value of (f:field[putative value]*,
s:(field[putative ordinal-value] | putative ordinal-value),
v:variant[uninitialized]) =
uninitialized .

- fixed-part _ :: variant-record-value → field[value]* (*total*) .

- variant-part _ :: variant-record-value → variant[record-value] (*total*) .

- selector-part _ :: tagged-record-value → field[ordinal-value] (*total*) ,
  untagged-record-value → ordinal-value (*total*) .

- selector _ :: variant-record-value → ordinal-value (*total*) .

(4) r = the variant-record-value of (f:field[value]*,
s:(field[ordinal-value] | ordinal-value),
v:variant[record-value]) : variant-record-value
⇒
(1) the fixed-part of $r = f$;
(2) the variant-part of $r = v$;
(3) the selector-part of $r = s$;
(4) s : field[ordinal-value] ⇒ the selector of r = the component of s ;
(5) s : ordinal-value ⇒ the selector of $r = s$.

3.14.3.2 Types

introduces: variant-record-type , tagged-record-type , untagged-record-type ,
variant-record-type of _ ,
fixed-part _ , variant-part _ , selector-part _ , selector-type _ ,
component-type designated by _ in _ .

- variant-record-type = tagged-record-type | untagged-record-type $(disjoint)$.
- variant-record-type of _ ::
  (field[type]*, field[ordinal-type], variant[record-type]+) $\rightarrow$ tagged-record-type $(partial)$,
  (field[type]*, ordinal-type, variant[record-type]+) $\rightarrow$ untagged-record-type $(partial)$.
- fixed-part _ :: variant-record-type $\rightarrow$ field[type]* $(total)$.
- variant-part _ :: variant-record-type $\rightarrow$ variant[record-type]+ $(total)$.
- selector-part _ :: tagged-record-type $\rightarrow$ field[ordinal-type] $(total)$,
  untagged-record-type $\rightarrow$ ordinal-type $(total)$.
- selector-type _ :: variant-record-type $\rightarrow$ ordinal-type $(total)$.

(1) r = the variant-record-type of (f:field[type]*,
s:(field[ordinal-type] | ordinal-type),
v:variant[record-type]+) : variant-record-type
$\Rightarrow$

(1) the fixed-part of $r = f$;

(2) the variant-part of $r = v$;

(3) the selector-part of $r = s$;

(4) s : field[ordinal-type] $\Rightarrow$ the selector-type of r = the component of s ;

(5) s : ordinal-type $\Rightarrow$ the selector-type of $r = s$.

- component-type designated by _ in _ :: identifier-token, variant-record-type $\rightarrow$ type $(partial)$.

(2) the component-type designated by i:identifier-token in t:tagged-record-type =
the type yielded by the component of the field identified by i in the fixed-part of r |
the type yielded by the component of the field identified by i in the selector-part of r |
the component-type designated by i in the variant-part of t .

(3) the component-type designated by i:identifier-token in t:untagged-record-type =
the type yielded by the component of the field identified by i in the fixed-part of r |
the component-type designated by i in the variant-part of t .

- component-type designated by _ in _ :: identifier-token, variant[record-type]* $\rightarrow$ type .

(4) the component-type designated by i:identifier-token in () =
nothing .

(5) the component-type designated by i:identifier-token in t:variant[record-type]+ =
the component-type designated by i in
the record-type yielded by the sub-record of the first of t |
the component-type designated by i in the rest of t .

3.14.3.3 Variables

introduces: variant-record-variable , tagged-record-variable , untagged-record-variable ,
variant-record-variable of _ ,
fixed-part _ , variant-part _ , selector-part _ , selector _ ,
type possessed by _ ,
field-bindings _ ,
access the component designated by _ in _ ,
the value attributed to _ , attribute _ to _ ,
uninitialize _ , allocate a variable of _ , dispose of _ .

- variant-record-variable = tagged-record-variable | untagged-record-variable ($disjoint$) .
- variant-record-variable of _ ::
  (field[variable]*, field[ordinal-variable], variant[record-variable]+, tagged-record-type) $\rightarrow$
  tagged-record-variable ($partial$) ,
  (field[variable]*, ordinal-variable, variant[record-variable]+, untagged-record-type) $\rightarrow$
  untagged-record-variable ($partial$) .
- fixed-part _ :: variant-record-variable $\rightarrow$ field[variable]* ($total$) .
- variant-part _ :: variant-record-variable $\rightarrow$ variant[record-variable]+ ($total$) .
- selector-part _ :: tagged-record-variable $\rightarrow$ field[ordinal-variable] ($total$) ,
  untagged-record-variable $\rightarrow$ ordinal-variable ($total$) .
- selector _ :: variant-record-variable $\rightarrow$ ordinal-variable ($total$) .
- type possessed by _ :: variant-record-variable $\rightarrow$ variant-record-type ($total$) .

(1) r = the variant-record-variable of (f:field[variable]*,
s:(field[ordinal-variable] | ordinal-variable),
v:variant[record-variable]+,
t:variant-record-type) : variant-record-variable
$\Rightarrow$
(1) the fixed-part of $r = f$;
(2) the variant-part of $r = v$;
(3) the selector-part of $r = s$;
(4) s : field[ordinal-variable] $\Rightarrow$ the selector of r = the component of s ;
(5) s : ordinal-variable $\Rightarrow$ the selector of $r = s$;
(6) the type possessed by $r = t$.

- field-bindings _ :: variant-record-variable $\rightarrow$ map [identifier-token to variable] ($total$) .

(2) the field-bindings of r:tagged-record-variable =
the disjoint-union of (the field-bindings of the fixed-part of r,
the field-bindings of the selector-part of r,
the field-bindings of the variant-part of r) .

(3) the field-bindings of r:untagged-record-variable =
the disjoint-union of (the field-bindings of the fixed-part of r,
the field-bindings of the variant-part of r) .

- field-bindings _ :: variant[record-variable]* $\rightarrow$ map [identifier-token to variable] ($partial$) .

(4) the field-bindings of () =
the empty-map .

(5) the field-bindings of v:variant[record-variable]$^+$ =
the disjoint-union of (the field-bindings of the sub-record of the first of v,
the field-bindings of the rest of v) .

- access the component designated by _ in _ ::
  yielder [of an identifier-token], yielder [of a variant-record-variable] $\rightarrow$
  action [giving a variable | storing | erring] [using current storage] .

(6) access the component designated by i:yielder [of an identifier-token]
in r:yielder [of a tagged-record-variable] =
give the component of the field identified by i in the fixed-part of r or
give the component of the field identified by i in the selector-part of r or
access the component designated by i in the variant-part of r
respecting the active variant of r .

(7) access the component designated by i:yielder [of an identifier-token]
in r:yielder [of an untagged-record-variable] =
give the component of the field identified by i in the fixed-part of r or
access the component designated by i in the variant-part of r
overriding the active variant of r .

- the value attributed to _ :: yielder [of a variant-record-variable] $\rightarrow$
  yielder [of a putative variant-record-value] [using current storage] .

(8) the value attributed to r:yielder [of a tagged-record-variable] =
the variant-record-value of (
the fields of (the identifiers of the fixed-part of r,
the values respectively attributed to the components of the fixed-part of r),
the field of (the identifier of the selector-part of r,
the value attributed to the selector of r),
the variant of (the selector-set of the active variant of r,
the value attributed to the sub-record of the active variant of r) .

(9) the value attributed to r:yielder [of an untagged-record-variable] =
the variant-record of (
the fields of (the identifiers of the fixed-part of r,
the values respectively attributed to the components of the fixed-part of r),
the value attributed to the selector of r,
the variant of (the selector-set of the active variant of r,
the value attributed to the sub-record of the active variant of r) .

- attribute _ to _ :: yielder [of a variant-record-value], yielder [of a variant-record-variable] $\rightarrow$
  action [storing] .

(10) attribute r_1:yielder [of a variant-record-value] to r_2:yielder [of a variant-record-variable] =
respectively attribute the components of the fixed-part of r_1 to
the components of the fixed-part of r_2
and then
| | check (the selector of r_1 selects the active variant of r_2)
| or
| | check not (the selector of r_1 selects the active variant of r_2) and then
| | uninitialize the sub-record of the active variant of r_2
and then attribute the selector of r_1 to the selector of r_2
and then attribute the sub-record of the variant-part of r_1 to
the sub-record of the active variant of r_2 .

- uninitialize _ :: yielder [of a variant-record-variable] $\rightarrow$ action [storing] [using current storage] .

(11) uninitialize r:yielder [of a variant-record-variable] =
uninitialize all the components of the fixed-part of r and then
uninitialize the sub-record of the active variant of r and then
uninitialize the selector of r .

- allocate a variable of _ :: yielder [of a variant-record-type] $\rightarrow$
action [giving a variant-record-variable | storing] [using current storage] .

(12) allocate a variable of t:yielder [of a tagged-record-type] =
| | respectively allocate variables of
| | the type* yielded by the components of the fixed-part of t then
| | give the fields of (the identifiers of the fixed-part of t, the given variable*)
| and
| | allocate a variable of the selector-type of t then
| | give the field of (the identifier of the selector-part of t, the given ordinal-variable)
| and respectively allocate variants of the variant-part of t
| and give t
then give the variant-record-variable of the given
(field[variable]*, field[ordinal-variable], variant[record-variable]$^+$, tagged-record-type) .

(13) allocate a variable of t:yielder [of an untagged-record-type] =
| | respectively allocate variables of the type* yielded by
| | the components of the fixed-part of t then
| | give the fields of (the identifiers of the fixed-part of t, the given variable*)
| and allocate a variable of the selector-type of t
| and respectively allocate variants of the variant-part of t
| and give t
then give the variant-record-variable of the given
(field[variable]*, ordinal-variable, variant[record-variable]$^+$, untagged-record-type) .

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

(14) dispose of r:yielder [of a variant-record-variable] =
dispose of all the components of the fixed-part of r and
dispose of the selector of r and
dispose of all variants of the variant-part of r .

privately introduces: _ selects _ , variant selected by _ in _ , empty-variant-variable ,
active variant _ ,
access the component designated by _ in _ respecting the active variant of _ ,
access the component designated by _ in _ overriding the active variant of _ ,
respectively allocate variants of _ , dispose of all variants _ .

- _ selects _ ::
ordinal-value, variant[record-variable] $\rightarrow$ truth-value (*partial*) ,
uninitialized, variant[record-variable] $\rightarrow$ false (*total*) .

(15) o:ordinal-value selects v:variant[record-variable] =
o is in the selector-set of v .

- variant selected by _ in _ ::
ordinal-value, variant[record-variable]* $\rightarrow$ variant[record-variable] (*partial*) ,
uninitialized, variant[record-variable]* $\rightarrow$ empty-variant-variable (*total*) .

(16) the variant selected by o:ordinal-value in () = nothing .

(17) the variant selected by o:ordinal-value in v:variant[record-variable]$^+$ =
when o selects the first of v then the first of v |
the variant selected by o in the rest of v .

- empty-variant-variable : variant[record-variable] .

(18) the empty-variant-variable =
the variant of (the empty-set, the fixed-record-variable of ()) .

- active variant _ :: yielder [of a variant-record-variable] $\rightarrow$
yielder [of a variant[record-variable]] [using current storage] .

(19) the active variant of r:yielder [of a variant-record-variable] =
the variant selected by (the value attributed to the selector of r) in the variant-part of r .

- access the component designated by _ in _ respecting the active variant of _ ::
yielder [of an identifier-token], yielder [of variant[record-variable]*],
yielder [of a tagged-record-variable] $\rightarrow$
action [giving a variable | storing | erring] [using current storage] .

(20) access the component designated by i:yielder [of an identifier-token] in ()
respecting the active variant of r:yielder [of a tagged-record-variable] =
fail .

(21) access the component designated by i:yielder [of an identifier-token]
in v:yielder [of variant[record-variable]$^+$]
respecting the active variant of r:yielder [of a tagged-record-variable] =
access the component designated by i in the sub-record of the first of v then
check (the value attributed to the selector of r selects the first of v) and then
give the given variable
or
check not (the value attributed to the selector of r selects the first of v) and then
report the error$\#2$
or access the component designated by i in the rest of v
respecting the active variant of r .

- access the component designated by _ in _ overriding the active variant of _ ::
yielder [of an identifier-token], yielder [of variant[record-variable]*],
yielder [of an untagged-record-variable] $\rightarrow$
action [giving a variable | storing | erring] [using current storage] .

(22) access the component designated by i:yielder [of an identifier-token] in ()
overriding the active variant of r:yielder [of an untagged-record-variable] =
fail .

(23) access the component designated by i:yielder [of an identifier-token]
in v:yielder [of variant[record-variable]$^+$]
overriding the active variant of r:yielder [of an untagged-record-variable] =
access the component designated by i in the sub-record of the first of v then
give the given variable and
check (the value attributed to the selector of r selects the first of v)
or
check not (the value attributed to the selector of r selects the first of v) and then
uninitialize the sub-record of the active variant of r and then
choose from the elements of the selector-set of the first of v then
attribute the given ordinal-value to the selector of r
or access the component designated by i in the rest of v
overriding the active variant of r .

- respectively allocate variants of _ :: yielder [of variant[record-type]*] →
  action [giving variant[record-variable]* | storing] [using current storage] .

(24) respectively allocate variants of t:yielder [of variant[record-type]*] =
| check t is ()
or
| | allocate a variable of the record-type yielded by the sub-record of the first of t then
| | give the variant of (the selector-set of the first of t, the given record-variable)
| and respectively allocate variants of the rest of t .

- dispose of all variants _ :: yielder [of variant[record-variable]*] → action [storing] .

(25) dispose of all variants of v:yielder [of variant[record-variable]*] =
| check v is ()
or
| dispose of the sub-record of the first of v and
| dispose of all variants of the rest of v .

3.14.4 Record Fields

introduces: field , field-component , field[_] ,
field of _ , identifier _ , component _ ,
fields of _ , identifiers _ , components _ ,
field identified by _ in _ .

- field-component = value | uninitialized | type | variable .
- field[_] :: field-component → field .
- $C \leq$ field-component ⇒
  - field of _ :: (identifier-token, C) → field[C] ($total$) ;
  - identifier _ :: field[C] → identifier-token ($total$) ;
  - component _ :: field[C] → C ($total$) ;
  - fields of _ :: (identifier-token*, C*) → field[C]* ($partial$) ;
  - identifiers _ :: field[C]* → identifier-token* ($total$) ;
  - components _ :: field[C]* → C* ($total$) ;
  - field identified by _ in _ :: identifier-token, field[C]* → field[C] ($partial$) .

(1) $C \leq$ field-component ⇒

(1) f = the field of (i:identifier-token, c:C) ⇒

(1) the identifier of f = i ;

(2) the component of f = c ;

(2) the fields of () = () ;

(3) the fields of (i:identifier-token$^+$, c:C^+) =
(the field of (the first of i, the first of c),
the fields of (the rest of i, the rest of c)) ;

(4) the identifiers of () = () ;

(5) the identifiers of f:field[C]$^+$ =
(the identifier of the first of f, the identifiers of the rest of f) ;

(6) the components of () = () ;

(7) the components of f:field$[C]^{+}$ =
(the component of the first of f, the components of the rest of f) ;

(8) the field identified by i:identifier-token in () = nothing ;

(9) the field identified by i:identifier-token in f:field$[C]^{+}$ =
when i is the identifier of the first of f then the first of f **|**
the field identified by i in the rest of f .

3.14.5 Record Variants

introduces: variant , variant-sub-record , variant[_] ,
variant of _ , selector-set _ , sub-record _ .

- variant-sub-record = record-value **|** uninitialized **|** record-type **|** record-variable .
- variant[_] :: variant-sub-record → variant .
- $R \leq$ variant-sub-record $\Rightarrow$
  - variant of _ :: (set [of ordinal-value], R) → variant[R] ($total$) ;
  - selector-set _ :: variant[R] → set [of ordinal-value] ($total$) ;
  - sub-record _ :: variant[R] → R ($total$) .

(1) v = the variant of (s:set [of ordinal-value], r:variant-sub-record) $\Rightarrow$

(1) the selector-set of $v = s$;

(2) the sub-record of $v = r$.

3.15 Files

3.15.1 General Files

3.15.1.1 Types

introduces: file-type , file-type of _ , component-type _ .

- file-type = plain-file-type **|** text-file-type ($disjoint$) .
- file-type of _ :: type → plain-file-type ($total$, $injective$) .
- component-type _ :: file-type → type ($total$) .

3.15.1.2 Variables

introduces: file-variable , file-component ,
file-mode , generation-mode , inspection-mode ,
file-variable of _ ,
left-part _ , right-part _ , mode-part _ , buffer-part _ ,
type possessed by _ , _ is _ ,
end-of-file reached in _ ,
start generation _ , advance generation _ ,
start inspection _ , advance inspection _ ,
read into _ from _ , write _ to _ ,
uninitialize _ , allocate a variable of _ , dispose of _ .

- file-variable = plain-file-variable | text-file-variable ($disjoint$) .
- file-component = plain-component | text-component .
- file-mode = generation-mode | inspection-mode ($individual$) .
- storable = □ | file-component* | file-mode .
- file-variable of _ :: (cell, cell, cell, variable, file-type) → file-variable ($partial$) .
- left-part _ , right-part _ , mode-part _ :: file-variable → cell ($total$) .
- buffer-part _ :: file-variable → variable ($total$) .
- type possessed by _ :: file-variable → file-type ($total$) .

(1) f = the file-variable of (l:cell, r:cell, m:cell, b:variable, t:file-type) ⇒

 (1) the left-part of f = l ;
 (2) the right-part of f = r ;
 (3) the mode-part of f = m ;
 (4) the buffer-part of f = b ;
 (5) the type possessed by f = t .

- _ is _ :: file-variable, file-variable → truth-value ($partial$) .

(2) f_1:file-variable is f_2:file-variable = the buffer-part of f_1 is the buffer-part of f_2 .

- end-of-file reached in _ :: yielder [of a file-variable] →
  yielder [of a truth-value | an error] [using current storage] .

(3) end-of-file reached in f:yielder [of a file-variable] =
 if f is undefined
 then the error#40
 else the count of the file-component* stored in the right-part of f is 0 .

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

(4) start generation of f:yielder [of a file-variable] =
 store the generation-mode in the mode-part of f and
 store () in the left-part of f and
 store () in the right-part of f and
 uninitialize the buffer-part f .

- advance generation _ :: yielder [of a file-variable] → action [storing | erring] [using current storage] .

(5) advance generation of f:yielder [of a file-variable] =
 assert ready to advance generation of f and then
 | give the value attributed to the buffer-part of f then .
 | | | store (the file-component* stored in the left-part of f, the given value)
 | | | in the left-part of f and
 | | | uninitialize the buffer-part of f
 | | or
 | | | check there is given uninitialized then
 | | | report the error#12

- start inspection _ :: yielder [of a file-variable] → action [storing | erring] [using current storage] .

(6) start inspection of f:yielder [of a file-variable] =
assert ready to start inspection of f and then
store the inspection-mode in the mode-part of f and then
transfer all components from left to right of f and then
update the buffer of f .

- advance inspection _ :: yielder [of a file-variable] →
action [storing | erring] [using current storage] .

(7) advance inspection of f:yielder [of a file-variable] =
assert ready to advance inspection of f and then
store (the file-component* stored in the left-part of f,
the first of the file-component+ stored in the right-part of f)
in the left-part of f and then
store (the rest of the file-component+ stored in the right-part of f)
in the right-part of f and then
update the buffer of f .

- read into _ from _ :: yielder [of a variable], yielder [of a file-variable] →
action [storing | erring] [using current storage] .

- write _ to _ :: yielder [of a value | a text-field], yielder [of a file-variable] →
action [storing | erring] [using current storage] .

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

(8) uninitialize f:yielder [of a file-variable] =
unstore the left-part of f and
unstore the right-part of f and
unstore the mode-part of f and
uninitialize the buffer-part of f .

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

(9) allocate a variable of t:yielder [of a file-type] =
| allocate a cell and allocate a cell and allocate a cell and
| allocate a variable of the component-type of t and give t
then give the file-variable of the given (cell, cell, cell, variable, file-type) .

- dispose of _ :: yielder [of a file-variable] → action [storing] .

(10) dispose of f:yielder [of a file-variable] =
unreserve the left-part of f and
unreserve the right-part of f and
unreserve the mode-part of f and
dispose of the buffer-part of f .

privately introduces: _ is undefined ,
assert ready to advance generation _ ,
assert ready to start inspection _ , assert ready to advance inspection _ ,
transfer all components from left to right _ , update the buffer _ .

- _ is undefined :: yielder [of a file-variable] → yielder [of a truth-value] [using current storage] .

(11) f:yielder [of a file-variable] is undefined =
when there is uninitialized yielded by (current storage at the mode-part of f) then true |
when there is a file-mode yielded by (current storage at the mode-part of f) then false .

- assert ready to advance generation _ :: yielder [of a file-variable] →
  action [completing | erring] [using current storage] .

(12) assert ready to advance generation of f:yielder [of a file-variable] =
| check (the file-mode stored in the mode-part of f is generation-mode)
or
| check (the file-mode stored in the mode-part of f is inspection-mode) and then
| report the error#9
or
| check (f is undefined) and then
| report the error#10 .

- assert ready to start inspection _ :: yielder [of a file-variable] →
  action [completing | erring] [using current storage] .

(13) assert ready to start inspection of f:yielder [of a file-variable] =
| check not (f is undefined)
or
| check (f is undefined) and then
| report the error#13 .

- assert ready to advance inspection _ :: yielder [of a file-variable] →
  action [completing | erring] [using current storage] .

(14) assert ready to advance inspection of f:yielder [of a file-variable] =
| check both (the file-mode stored in the mode-part of f is inspection-mode,
| not end-of-file reached in f)
or
| check both (the file-mode stored in the mode-part of f is inspection-mode,
| end-of-file reached in f) and then
| report the error#16
or
| check (the file-mode stored in the mode-part of f is generation-mode) and then
| report the error#14
or
| check (f is undefined) and then
| report the error#15 .

- transfer all components from left to right _ , update the buffer _ ::
  yielder [of a file-variable] → action [storing] [using current storage] .

3.15.2 Plain Files

3.15.2.1 Types

introduces: plain-file-type , file-type of _ , component-type _ .

- file-type of _ :: type → plain-file-type ($total$, $injective$) .

- component-type _ :: plain-file-type → type ($total$) .

(1) the component-type of the file-type of t:type = t .

3.15.2.2 Variables

introduces: plain-file-variable , plain-component ,
file-variable of _ ,
read into _ from _ , write _ to _ .

- plain-component = value .
- file-variable of _ :: (cell, cell, cell, variable, plain-file-type) → plain-file-variable (*partial*) .
- read into _ from _ :: yielder [of a variable], yielder [of a plain-file-variable] →
  action [storing | erring] [using current storage] .

(1) read into v:yielder [of a variable] from f:yielder [of a plain-file-variable] =
  assert ready to advance inspection of f and then
  | | give the value attributed to the buffer-part of f
  | | reporting any given error
  | then assign the given value to v
  and then advance inspection of f .

- write _ to _ :: yielder [of a value], yielder [of a plain-file-variable] →
  action [storing | erring] [using current storage] .

(2) write v:yielder [of a value] to f:yielder [of a plain-file-variable] =
  assert ready to advance generation of f and then
  assign v to the buffer-part of f and then
  advance generation of f .

privately introduces: transfer all components from left to right _ , update the buffer _ .

- transfer all components from left to right _ :: yielder [of a plain-file-variable] →
  action [storing] [using current storage] .

(3) transfer all components from left to right of f:yielder [of a plain-file-variable] =
  give (the plain-component* stored in the left-part of f,
    the plain-component* stored in the right-part of f) then
  | store the given plain-component* in the right-part of f and
  | store () in the left-part of f .

- update the buffer _ :: yielder [of a plain-file-variable] → action [storing] [using current storage] .

(4) update the buffer of f:yielder [of a plain-file-variable] =
  | check end-of-file reached in f and then
  | uninitialize the buffer-part of f
  or
  | attribute (the first of the plain-component+ stored in the right-part of f) to
  | the buffer-part of f .

3.15.3 Text Files

3.15.3.1 Types

introduces: text-file-type , component-type _ .

- text-file-type : file-type .
- component-type _ :: text-file-type → character-type (*total*) .

3.15.3.2 Variables

introduces: text-file-variable ,
text-component , end-of-line , end-of-page , _ is _ ,
file-variable of _ ,
read into _ from _ , write _ to _ ,
end-of-line reached in _ ,
skip past the next end-of-line in _ ,
write an end-of-line to _ , write an end-of-page to _ .

- text-component = character | end-of-line | end-of-page (*disjoint*) .
- end-of-line , end-of-page : text-component .
- _ is _ :: text-component, text-component → truth-value (*total*, *commutative*) .

(1) $c_1 = c_2 \Rightarrow$
c_1:text-component is c_2:text-component = true .

(2) c_1:character is c_2:(end-of-line | end-of-page) = false .

(3) end-of-line is end-of-page = false .

- file-variable of _ :: (cell, cell, cell, character-variable, text-file-type) → text-file-variable (*total*) .
- read into _ from _ :: yielder [of a character-variable | an integer-variable | a real-variable],
  yielder [of a text-file-variable] → action [storing | erring] [using current storage] .

(4) read into v:yielder [of a character-variable] from f:yielder [of a text-file-variable] =
  assert ready to advance inspection of f and then
  | | give the value attributed to the buffer-part of f
  | | reporting any given error
  | then assign the given character to v
  and then advance inspection of f .

(5) read into v:yielder [of an integer-variable] from f:yielder [of a text-file-variable] =
  assert ready to advance inspection of f and then
  skip blanks in f and then
  | get an integer-number from f then assign the given integer-number to v .

(6) read into v:yielder [of a real-variable] from f:yielder [of a text-file-variable] =
  assert ready to advance inspection of f and then
  skip blanks in f and then
  | get a real-number from f then assign the given real-number to v .

- write _ to _ :: yielder [of a textually-writable | a text-field],
  yielder [of a text-file-variable] → action [storing | erring] [using current storage] .

(7) write t:yielder [of a textually-writable] to f:yielder [of a text-file-variable] =
  write the text-representation of t to f .

(8) write t:yielder [of a text-field] to f:yielder [of a text-file-variable] =
  assert ready to advance generation of f and then
  | store (the file-component* stored in the left-part of f, the characters of t)
  | in the left-part of f and
  | uninitialize the buffer-part of f.

- end-of-line reached in _ :: yielder [of a text-file-variable] →
  yielder [of a truth-value | an error] [using current storage] .

(9) end-of-line reached in f:yielder [of a text-file-variable] =
if f is undefined
then the error#41
else if end-of-file reached in f
then the error#42
else the first of the text-component$^+$ stored in the right-part of f is the end-of-line .

- skip past the next end-of-line in _ :: yielder [of a text-file-variable] →
action [storing | erring] [using current storage] .

(10) skip past the next end-of-line in f:yielder [of a text-file-variable] =
unfolding
| | give (end-of-line reached in f) reporting any given error
| then
| | | check (it is true) and then
| | | advance inspection of f
| | or
| | | check (it is false) and then
| | | advance inspection of f and then
| | | unfold .

- write an end-of-line to _ :: yielder [of a text-file-variable] →
action [storing | erring] [using current storage] .

(11) write an end-of-line to f:yielder [of a text-file-variable] =
assert ready to advance generation of f and then
| store (the file-component* stored in the left-part of f, the end-of-line)
| in the left-part of f and
| uninitialize the buffer-part of f .

- write an end-of-page to _ :: yielder [of a text-file-variable] →
action [storing | erring] [using current storage] .

(12) write an end-of-page to f:yielder [of a text-file-variable] =
assert ready to advance generation of f and then
| store (the line-termination of the file-component* stored in the left-part of f,
| the end-of-page)
| in the left-part of f and
| uninitialize the buffer-part of f .

privately introduces: transfer all components from left to right _ , update the buffer _ ,
line-termination _ ,
separation of a signed-integer from _ ,
separation of a signed-number from _ ,
get an integer-number from _ , get a real-number from _ ,
skip blanks in _ , inspect an end-of-page in _ .

- transfer all components from left to right _ :: yielder [of a text-file-variable] →
action [storing] [using current storage] .

(13) transfer all components from left to right of f:yielder [of a text-file-variable] =
give (the text-component* stored in the left-part of f,
the text-component* stored in the right-part of f) then
| store the line-termination of the given text-component* in the right-part of f and
| store () in the left-part of f .

- update the buffer _ :: yielder [of a text-file-variable] $\rightarrow$
  action [storing] [using current storage] .

(14) update the buffer of f:yielder [of a text-file-variable] =
| check end-of-file reached in f and then
| uninitialize the buffer-part of f
or
| give the first of the text-component$^+$ stored in the right-part of f then
| | | attribute the given character to the buffer-part of f
| | or
| | | check (the given text-component is the end-of-line) and then
| | | attribute the space to the buffer-part of f
| | or
| | | check (the given text-component is the end-of-page) and then
| | | inspect an end-of-page in f .

- line-termination _ :: text-component* $\rightarrow$ text-component* .

(15) the line-termination of () = () .

(16) the line-termination of c:text-component$^+$ =
if the last of c is the end-of-line then c else (c, the end-of-line) .

- separation of a signed-integer from _ ::
  text-component* $\rightarrow$ (text-field of a signed-integer, text-component*) | error .

(17) c = (i:signed-integer, c':text-component$^+$) ;
(i, the first of t) & signed-integer = nothing
$\Rightarrow$
the separation of a signed-integer from c:text-component* = (the text-field of i, c') .

(18) c & (signed-integer, text-component$^+$) = nothing $\Rightarrow$
the separation of a signed-integer from c:text-component* = the error#54 .

- separation of a signed-number from _ ::
  text-component* $\rightarrow$ (text-field of a signed-number, text-component*) | error .

(19) c = (n:signed-number, c':text-component$^+$) ;
(n, the first of t) & signed-number = nothing
$\Rightarrow$
the separation of a signed-number from c:text-component* = (the text-field of n, c') .

(20) c & (signed-number, text-component$^+$) = nothing $\Rightarrow$
the separation of a signed-number from c:text-component* = the error#56 .

- get an integer-number from _ :: yielder [of a text-file-variable] $\rightarrow$
  action [giving an integer-number | storing | erring] [using current storage] .

(21) get an integer-number from f:yielder [of a text-file-variable] =
give the separation of a signed-integer from the text-component* stored in the right-part of f
reporting any given error
then
store (the text-component* stored in the left-part of f,
the characters of the given text-field#1)
in the left-part of f and then
store the rest of the given data in the right-part of f and then
update the buffer of f
and
give the integer-number of the numerical-value of the characters of the given text-field#1
reporting any given error .

- get a real-number from _ :: yielder [of a text-file-variable] $\rightarrow$
action [giving a real-number | storing | erring] [using current storage] .

(22) get a real-number from f:yielder [of a text-file-variable] =
give the separation of a signed-number from the text-component* stored in the right-part of f
reporting any given error
then
store (the text-component* stored in the left-part of f,
the characters of the given text-field#1)
in the left-part of f and then
store the rest of the given data in the right-part of f and then
update the buffer of f
and give the real-number of the numerical-value of the characters of the given text-field#1 .

- skip blanks in _ :: yielder [of a text-file-variable] $\rightarrow$
action [storing | erring] [using current storage] .

(23) skip blanks in f:yielder [of a text-file-variable] =
unfolding
check (the character attributed to the buffer-part of f is the space) and then
advance inspection of f and then
unfold
or
check not (the character attributed to the buffer-part of f is the space)
or
check end-of-file reached in f .

- inspect an end-of-page in _ :: yielder [of a text-file-variable] $\rightarrow$ action .

(24) inspect an end-of-page in f:yielder [of a text-file-variable] = $\Box$.

3.15.4 Text Fields

introduces: textually-writable ,
text-field, text-field of _ , characters _ ,
text-representation _ , _ spaces .

- textually-writable = character | integer-number | real-number | truth-value | string-value .

- text-field of _ :: character$^+$ $\rightarrow$ text-field ($total$) .

- characters _ :: text-field $\rightarrow$ character$^+$ ($total$) .

(1) the characters of the text-field of c:character$^+$ = c .

- text-representation _ :: textually-writable $\longrightarrow$ text-field $(total)$,
  (textually-writable, integer-number) $\longrightarrow$ text-field | error $(total)$,
  (real-number, integer-number, integer-number) $\longrightarrow$ text-field | error $(total)$.

(2) w is greater than 0 = false $\Rightarrow$
the text-representation of (t:textually-writable, the integer-number of w:integer) =
the error#58 .

(3) both (w is greater than 0, f is greater than 0) = false $\Rightarrow$
the text-representation of (r:real-number, the integer-number of w:integer,
the integer-number of f:integer) =
the error#58 .

- _ spaces :: natural $\longrightarrow$ space* .

(4) (n:natural) spaces = space^n .

3.15.4.1 Character Fields

- text-representation _ ::
  character $\longrightarrow$ text-field of character $(total)$,
  (character, integer-number of a positive-integer) $\longrightarrow$ text-field of (space*, character) $(total)$.

(1) the text-representation of c:character =
the text-field of c .

(2) the text-representation of (c:character, the integer-number of w:positive-integer) =
the text-field of ((the predecessor of w) spaces, c) .

3.15.4.2 Integer Fields

introduces: default-integer-field-width .

- default-integer-field-width : positive-integer .
- text-representation _ ::
  integer-number $\longrightarrow$ text-field of (space*, '$-$'$^?$, unsigned-integer) $(total)$,
  (integer-number, integer-number of a positive-integer) $\longrightarrow$
  text-field of (space*, '$-$'$^?$, unsigned-integer) $(total)$.

(1) the text-representation of i:integer-number =
the text-representation of (i, the integer-number of the default-integer-field-width) .

(2) d = the decimal-digits of absolute i ;
w' = the successor of the count of d
$\Rightarrow$
the text-representation of (the integer-number of i:integer,
the integer-number of w:positive-integer) =
if w is less than w'
then the text-field of (if i is less than 0 then '$-$' else (), d)
else the text-field of ((the difference of (w, w')) spaces,
if i is less than 0 then '$-$' else the space, d) .

3.15.4.3 Real Fields

introduces: default-real-field-width , exponent-width , exponent-character .

- default-real-field-width , exponent-width : positive-integer .
- exponent-character : ('E' | 'e') .
- text-representation _ ::
  real-number $\rightarrow$ text-field of (space*, '$-$'$^?$, unsigned-floating-point-real) $(total)$,
  (real-number, integer-number of a positive-integer) $\rightarrow$
  text-field of (space*, '$-$'$^?$, unsigned-floating-point-real) $(total)$,
  (real-number, integer-number of a positive-integer,
  integer-number of a positive-integer) $\rightarrow$
  text-field of (space*, '$-$'$^?$, unsigned-fixed-point-real) $(total)$.

(1) the text-representation of r:real-number =
the text-representation of (r, the integer-number of the default-real-field-width) .

(2) w' = the greater of (w, the sum of (the exponent-width, 6)) ;
f = the difference of (w', the sum of (the exponent-width, 5)) ;
m = the rounded quotient of (absolute r, the exponent of (10, n)) ;
d_1 = the decimal-digits of m ;
the count of d_1 = the successor of f ;
x = the sum of (n, f) ;
d_2 : digit$^+$ = exactly (the exponent-width) decimal-digits of absolute x
$\Rightarrow$
the text-representation of (the real-number of r:rational,
the integer-number of w:positive-integer) =
the text-field of (if both (r is less than 0, m is greater than 0) then '$-$' else the space,
the first of d_1, '.', the rest of d_1,
the exponent-character, if x is less than 0 then '$-$' else '$+$', d_2 .

(3) m = the rounded product of (absolute r, the exponent of (10, f)) ;
i = the greater of (1, the difference of (the count of the decimal-digits of m, f)) ;
w' = the sum of (i, f, if both (r is less than 0, m is greater than 0) then 2 else 1)
d = exactly (the sum of (i, f)) decimal-digits of m
$\Rightarrow$
the text-representation of (the real-number of r:rational, the integer-number of w:positive-integer,
the integer-number of f:positive-integer) =
the text-field of (if w is greater than w' then (the difference of (w, w')) spaces else (),
if both (r is less than 0, m is greater than 0) then '$-$' else (),
the components from $\#1$ to $\#i$ of d,
'.', the components from #(the successor of i) of d) .

privately introduces: greater _ .

- greater _ :: integer^2 $\rightarrow$ integer $(total)$.

(4) the greater of (i_1:integer, i_2:integer) =
if i_1 is greater than i_2 then i_1 else i_2 .

3.15.4.4 Boolean Fields

introduces: default-boolean-field-width .

- default-boolean-field-width : positive-integer .

- text-representation _ ::
  truth-value → text-field of (space*, letter$^+$) ($total$) ,
  (truth-value, integer-number of a positive-integer) →
  text-field of (space*, letter$^+$) ($total$) .

(1) the text-representation of t:truth-value =
the text-representation of (t, the integer-number of the default-boolean-field-width) .

(2) c = the representation of t ;
w' = the count of c
⇒
the text-representation of (t:truth-value, the integer-number of w:positive-integer) =
if w is less than w'
then the text-field of the components from #1 to #w of c
else the text-field of ((the difference of (w, w')) spaces, c) .

privately introduces: representation _ .

- representation _ :: : truth-value → letter$^+$.

(3) the representation of true : ('T' | 't', 'R' | 'r', 'U' | 'u', 'E' | 'e') .

(4) the representation of false : ('F' | 'f', 'A' | 'a', 'L' | 'l', 'S' | 's', 'E' | 'e') .

3.15.4.5 String Fields

- text-representation _ ::
  string-value → text-field ($total$) ,
  (string-value, integer-number of a positive-integer) → text-field ($total$) .

(1) the text-representation of the string-value of c:character$^+$ =
the text-field of c .

(2) w' = the count of c ⇒
the text-representation of (the string-value of c:character$^+$,
the integer-number of w:positive-integer) =
if w is less than w'
then the text-field of the components from #1 to #w of c
else the text-field of ((the difference of (w, w')) spaces, c) .

3.16 Decimal Conversions

introduces: signed-number , signed-integer , signed-real ,
unsigned-number , unsigned-integer , unsigned-real ,
unsigned-fixed-point-real , unsigned-floating-point-real ,
decimal-digit _ , decimal-digits _ , exactly _ decimal-digits _ ,
numerical-value _ .

- signed-number = signed-integer | signed-real .

- signed-integer = (('+' | '−')$^?$, unsigned-integer) .

- signed-real = (('+' | '−')$^?$, unsigned-real) .

- unsigned-number = unsigned-integer | unsigned-real .

- unsigned-integer = digit$^+$.

- unsigned-real = unsigned-fixed-point-real **|** unsigned-floating-point-real .
- unsigned-fixed-point-real = (digit$^+$, '.', digit$^+$) .
- unsigned-floating-point-real = (digit$^+$, ('.', digit$^+$)$^?$, ('E' **|** 'e'), signed-integer) .
- decimal-digit _ :: natural [max 9] $\rightarrow$ digit ($total$) .

(1) the decimal-digit of 0 = '0' ; the decimal-digit of 1 = '1' ;
the decimal-digit of 2 = '2' ; the decimal-digit of 3 = '3' ;
the decimal-digit of 4 = '4' ; the decimal-digit of 5 = '5' ;
the decimal-digit of 6 = '6' ; the decimal-digit of 7 = '7' ;
the decimal-digit of 8 = '8' ; the decimal-digit of 9 = '9' .

- decimal-digits _ :: natural $\rightarrow$ digit$^+$ ($total$) .

(2) the decimal-digits of n:natural [max 9] = the decimal-digit of n .

(3) the decimal-digits of n:natural [min 10] =
(the decimal-digits of the integer-quotient of (n, 10),
the decimal-digit of the integer-remainder of (n, 10) .

- exactly _ decimal-digits _ :: positive-integer, natural $\rightarrow$ digit$^+$ ($partial$) .

(4) exactly 1 decimal-digits of n:natural =
when n is less than 10 then the decimal-digit of n .

(5) exactly p:integer [min 2] decimal-digits of n:natural =
(exactly (the predecessor of p) decimal-digits of the integer-quotient of (n, 10),
the decimal-digit of the integer-remainder of (n, 10)) .

- numerical-value _ :: signed-integer $\rightarrow$ integer ($total$) ,
signed-real $\rightarrow$ rational ($total$) .

(6) the numerical-value of d:digit$^+$ = decimal d .

(7) the numerical-value of (d_1:digit$^+$, '.', d_2:digit$^+$) =
the sum of (decimal d_1,
the product of (decimal d_2,
the exponent of (10, the negation of the count of d_2))) .

(8) the numerical-value of (d:(digit$^+$, ('.', digit$^+$)$^?$), e:('E' **|** 'e'), i:signed-integer) =
the product of (the numerical-value of d,
the exponent of (10, the numerical-value of i)) .

(9) the numerical-value of ('+', n:unsigned-number) = the numerical-value of n .

(10) the numerical-value of ('−', n:unsigned-number) = the negation of the numerical-value of n .

3.17 Procedures and Functions

3.17.1 Procedures

introduces: procedure , procedure of _ , formal-parameter-modes _ , body-abstraction _ .

- procedure of _ :: (parameter-modes, abstraction) → procedure ($partial$) .

(1) m : parameter-modes ;
a : abstraction [completing | storing | diverging | erring | jumping]
[using given argument* | current storage] ⇒
procedure of (m, a) : procedure .

- formal-parameter-modes _ :: procedure → parameter-modes ($total$) .

- body-abstraction _ :: procedure → abstraction ($total$) .

(2) p = (procedure of (m:parameter-modes, a:abstraction)) : procedure ⇒

 (1) formal-parameter-modes of p = m ;

 (2) body-abstraction of p = a .

3.17.2 Functions

introduces: function , function of _ , formal-parameter-modes _ , body-abstraction _ , result-variable-inclusion of _ , result-variable _ .

- function of _ :: (parameter-modes, abstraction) → function ($partial$) .

(1) m : parameter-modes ;
a : abstraction [giving a result-value | storing | diverging | erring | jumping]
[using given argument* | current storage] ⇒
function of (m, a) : function .

- formal-parameter-modes _ :: function → parameter-modes ($total$) .

- body-abstraction _ :: function → abstraction ($total$) .

- result-variable _ :: function → result-variable ($partial$) .

(2) f = (function of (m:parameter-modes, a:abstraction)) : function ⇒

 (1) formal-parameter-modes of f = m ;

 (2) body-abstraction of f = a .

 (3) result-variable of f = nothing .

- result-variable-inclusion of _ :: (function, result-variable) → function ($partial$) .

(3) f = (function of (m:parameter-modes, a:abstraction)) : function ;
v : result-variable ⇒
result-variable-inclusion of (f, v) : function .

(4) f' = (result-variable-inclusion of (f:function, v:result-variable)) : function ⇒

 (1) formal-parameter-modes of f' = parameter-modes of f ;

 (2) body-abstraction of f' = body-abstraction of f ;

 (3) result-variable of f' = v .

3.17.3 Function Results

introduces: result-value , result-variable , result-type .

- result-value = ordinal-value **|** real-value **|** pointer-value .
- result-variable = ordinal-variable **|** real-variable **|** pointer-variable .
- result-type = ordinal-type **|** real-type **|** pointer-type .

3.17.4 Arguments

introduces: argument , evaluated-argument, unevaluated-argument ,
unevaluated-argument of _ , unevaluated-argument-abstraction _ .

- argument = evaluated-argument **|** unevaluated-argument (*disjoint*) .
- evaluated-argument = value **|** variable **|** procedure **|** function .
- unevaluated-argument of _ :: abstraction → unevaluated-argument (*partial*) .

(1) a : abstraction [giving an evaluated-argument **|** storing **|** diverging **|** erring **|** jumping]
[using current storage] ⇒
unevaluated-argument of a : unevaluated-argument .

- unevaluated-argument-abstraction _ :: unevaluated-argument → abstraction (*total*) .

(2) t = unevaluated-argument of (a:abstraction) : unevaluated-argument ⇒
unevaluated-argument-abstraction of t = a .

3.17.5 Parameter Modes

introduces: parameter-mode , declared-parameter-mode ,
value-mode , variable-mode , procedural-mode , functional-mode ,
write-value-mode , read-variable-mode , file-variable-mode ,
read-file-variable-mode , write-file-variable-mode ,
parameter-modes , multiple _ , ultimate _ , penultimate _ .

- parameter-mode = declared-parameter-mode **|** read-variable-mode **|**
  write-value-mode **|** file-variable-mode (*disjoint*) .
- read-variable-mode , write-value-mode : parameter-mode .
- declared-parameter-mode = value-mode **|** variable-mode **|**
  procedural-mode **|** functional-mode (*individual*) .
- file-variable-mode = read-file-variable-mode **|** write-file-variable-mode (*individual*) .
- parameter-modes = declared-parameter-mode* **|** (variable-mode, multiple value-mode) **|**
  (read-file-variable-mode, (multiple read-variable-mode)$^?$) **|**
  (write-file-variable-mode, (multiple write-value-mode)$^?$) .
- multiple _ :: parameter-mode → multiple parameter-mode (*total*) .
- ultimate _ :: parameter-modes → parameter-mode (*partial*) .
- penultimate _ :: parameter-modes → parameter-modes (*partial*) .

(1) m = (m_1:declared-parameter-mode*, m_2:declared-parameter-mode) : parameter-modes ⇒

(1) ultimate of m = m_2 ;

(2) penultimate of m = m_1 .

(2) m = (m_1:parameter-mode, multiple m_2:parameter-mode) : parameter-modes ⇒

(1) ultimate of m = m_2 ;

(2) penultimate of m = m .

(3) m : file-variable-mode ⇒

(1) ultimate of m = m ;

(2) penultimate of m = nothing .

(4) ultimate of ( ) = nothing .

(5) penultimate of ( ) = nothing .

3.17.6 Formal Parameters

introduces: formal-parameter , formal _ parameter of _ , formal _ parameters of _ , modes _ , mode _ , parameter-identifier _ , parameter-type _ .

- formal _ parameter of _ :: (value-mode | variable-mode), (identifier-token, type) → formal-parameter (*total*) ,
  (procedural-mode | functional-mode), identifier-token → formal-parameter (*total*).

- formal _ parameters of _ :: (value-mode | variable-mode), (identifier-token+, type) → formal-parameter+ (*total*) .

(1) formal m:(value-mode | variable-mode) parameters of (i:identifier-token, t:type) =
formal m parameter of (i, t) .

(2) formal m:(value-mode | variable-mode) parameters of
(i_1:identifier-token, i_2:identifier-token+, t:type) =
(formal m parameter of (i_1, t), formal m parameters of (i_2, t)) .

- modes _ :: formal-parameter* → declared-parameter-mode* (*total*) .

(3) modes of (p_1:formal-parameter, p_2:formal-parameter*) =
(mode of p_1, modes of p_2) .

(4) modes of ( ) = ( ) .

- mode _ :: formal-parameter → declared-parameter-mode (*total*) .

- parameter-identifier _ :: formal-parameter → identifier-token (*total*) .

- parameter-type _ :: formal-parameter → type (*partial*) .

(5) p = (the formal m:(value-mode | variable-mode) parameter of (i:identifier-token, t:type))
: formal-parameter ⇒

(1) mode of p = m ;

(2) parameter-identifier of p = i ;

(3) parameter-type of p = t .

(6) p = (the formal m:(procedural-mode | functional-mode) parameter of (i:identifier-token))
: formal-parameter ⇒

(1) mode of $p = m$;

(2) parameter-identifier of $p = i$;

(3) parameter-type of p = nothing .

3.17.7 Parameter Association

introduces: associate _ with _ , respectively associate _ with _ ,
dissociate _ , dissociate all _
abstraction for associating _ , abstraction for dissociating _ ,
abstraction for including a result-variable of _ in the binding for _ .

- associate _ with _ ::
  yielder [of a formal-parameter], yielder [of an evaluated-argument] →
  action [binding | storing] .

(1) associate p:yielder [of a formal-parameter] with a:yielder [of an evaluated-argument] =
| check there is the value-mode yielded by
| the mode of p and then
| | allocate a variable of the parameter-type of p then
| | | bind the parameter-identifier of p to the given variable and
| | | assign the value yielded by a to the given variable
or
| check there is the variable-mode yielded by the mode of p
| and then bind the parameter-identifier of p to the variable yielded by a
or
| check there is the procedural-mode yielded by the mode of p
| and then bind the parameter-identifier of p to the procedure yielded by a
or
| check there is the functional-mode yielded by the mode of p
| and then bind the parameter-identifier of p to the function yielded by a .

- respectively associate _ with _ ::
  yielder [of formal-parameter*], yielder [of evaluated-argument*] →
  action [binding | storing] .

(2) respectively associate p:yielder [of formal-parameter*]
with a:yielder [of evaluated-argument*] =
| check (p is ())
or
| associate the first of p with the first of a and
| respectively associate the rest of p with the rest of a .

- dissociate _ :: yielder [of a formal-parameter] → action [storing] .

(3) dissociate p:yielder [of a formal-parameter] =
| check there is the value-mode yielded by
| the mode of p and then
| deallocate the variable bound to the parameter-identifier of p
or check there is the variable-mode yielded by the mode of p
or check there is the procedural-mode yielded by the mode of p
or check there is the functional-mode yielded by the mode of p .

- dissociate all _ :: yielder [of formal-parameter*] → action [storing] .

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

- abstraction for associating _ ::
formal-parameter* → abstraction [binding | storing] [using given argument+] .

(5) abstraction for associating p:formal-parameter* =
abstraction of
| respectively associate p with the given argument* .

- abstraction for dissociating _ ::
formal-parameter* → abstraction [storing] [using given argument+] .

(6) abstraction for dissociating p:formal-parameter* =
abstraction of
| dissociate all p .

- abstraction for including a result-variable of _ in the binding for _ ::
type, identifier-token → abstraction [binding | storing] .

(7) abstraction for including a result-variable of t:type
in the binding for i:identifier-token =
abstraction of
| allocate a variable of t then
| bind i to the result-variable-inclusion of (the given variable,
| the function bound to i) .

3.17.8 Forward Procedures

introduces: forward-procedure of _ , parameters _ .

- forward-procedure of _ :: formal-parameter* → forward-procedure (*total*, *injective*) .
- parameters _ :: forward-procedure → formal-parameter* (*total*, *injective*) .

(1) parameters of forward-procedure of p:formal-parameter* = p .

3.17.9 Forward Functions

introduces: forward-function of _ , parameters _ , result-type _ .

- forward-function of _ :: (formal-parameter*, result-type) → forward-function (*total*, *injective*) .
- parameters _ :: forward-function → formal-parameter* (*total*, *injective*) .
- result-type _ :: forward-function → result-type (*total*) .

(1) f = forward-function of (p:formal-parameter*, t:result-type) ⇒

  (1) parameters of f = p ;
  (2) result-type of f = t .

3.18 Program Points

introduces: program-point , jumping , _ thereafter _ , _ is _ , generate a program-point .

- jumping = escaping with a program-point .
- _ thereafter _ :: action, action → action (*total*) .

(1) A_1:action thereafter A_2:action =
| A_1 then
| | A_2 trap fail
trap
| check there is given a program-point and then
| A_2 and then escape with the given program-point .

privately introduces: program-point of _ .

- program-point of _ :: cell → program-point (*total*) .
- _ is _ :: program-point, program-point → truth-value (*total*) .

(2) program-point of c_1:cell is program-point of c_2:cell = c_1 is c_2 .

- generate a program-point :
  action [giving a program-point | storing] [using current storage] .

(3) generate a program-point =
allocate a cell then give the program-point of the given cell .

3.19 External Entities

introduces: external-entity ,
externally associate _ with _ , allocate a variable of _ associated with _ ,
globally allocate a variable of _ for _ , respectively globally allocate variables of _ for _ ,
open the input text-file if appropriate , open the output text-file if appropriate .

- external-entity = □ .
- externally associate _ with _ ::
  identifier-token$^+$, yielder [of external-entity$^+$] → action [binding] .

(1) externally associate i:identifier-token$^+$ with
e:yielder [of external-entity$^+$] = □ .

- allocate a variable of _ associated with _ ::
  yielder [of a type], yielder [of an external-entity] →
  action [giving a variable | storing] [using current storage] .

(2) allocate a variable of t:yielder [of a type]
associated with e:yielder [of an external-entity] = □ .

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

(3) globally allocate a variable of t:yielder [of a type]
for i:yielder [of an identifier-token] =
| check not (i is known to be bound)
| and then allocate a variable of t
or
| allocate a variable of t
| associated with the external-entity bound to i .

- respectively globally allocate variables of _ for _ ::
yielder [of a type*], yielder [of identifier-token*] →
action [giving variable* | storing] [using current storage] .

(4) respectively globally allocate variables of t:yielder [of a type*]
for i:yielder [of identifier-token*] =
| check (i is ())
or
| globally allocate a variable of (the first of t) for (the first of i) and
| respectively globally allocate variables of (the rest of t) for (the rest of i) .

- open the input text-file if appropriate :
action [binding | storing] [using current bindings | current storage] .

(5) open the input text-file if appropriate =
| allocate a variable of the text-file-type associated with
| the external-entity bound to the token of "input" then
| | bind the token of "input" to the given text-file-variable and
| | bind the input-token to the given text-file-variable and
| | start inspection of the given text-file-variable
or
| check not ("input" is known to be bound) .

- open the output text-file if appropriate :
action [binding | storing] [using current bindings | current storage] .

(6) open the output text-file if appropriate =
| allocate a variable of the text-file-type associated with
| the external-entity bound to the token of "output" then
| | bind the token of "output" to the given text-file-variable and
| | bind the output-token to the given text-file-variable and
| | start generation of the given text-file-variable
or
| check not ("output" is known to be bound) .

3.20 Errors

introduces: error , error of _ , error#_ ,
proper-datum , _ but _ ,
report _ , _ reporting _ , erring .

- error of _ :: string → error (*partial*) .

- error#_ :: positive-integer [max 58] → error (*total*, *injective*) .

(1) error#1 = error of "For an indexed-variable ..." .

(2) ...

(3) error#58 = error of "On writing to a text file, ..." .

- datum = proper-datum | error (*disjoint*) .
- _ but _ :: datum, error → datum (*total*) .

(4) e:error but e':error = e' .

(5) d:proper-datum but e:error = d .

- report _ :: yielder [of an error] → action [erring] (*total*) .

(6) report e:yielder [of an error] = □ .

- _ reporting _ :: action, yielder [of an error] → action (*total*) .

(7) 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 = communicating .

3.21 Bindings

- bindable = constant-value | type | variable |
  procedure | function | forward-procedure | forward-function |
  program-point | external-entity .

3.21.1 Tokens

introduces: identifier-token , label-token , input-token , output-token , _ is known to be bound , respectively bind _ to _ , the variables respectively bound to _ , unbind all _ .

- token = identifier-token | label-token | input-token | output-token (*disjoint*) .
- identifier-token = string of (lowercase letter, (lowercase letter | digit)*) .
- label-token = natural [max 9999] .
- input-token , output-token : token .
- 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 bindable*] .

(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) .

- unbind all _ :: yielder [of identifier-token*] → action [binding] .

(3) unbind all i:yielder [of identifier-token*] =
| check (i is ())
or
| unbind the first of i and
| unbind all the rest of i .

- _ is known to be bound :: identifer-token → yielder [of a truth-value] [using current bindings] .

(4) i:yielder [of identifier-token] is known to be bound =
i is in the mapped set of the known current bindings .

3.21.2 Required

introduces: required-bindings .

privately introduces: required-constants , required-types ,
required-file-handling-procedures ,
required-dynamic-allocation-procedures ,
required-transfer-procedures ,
required-arithmetic-functions , required-transfer-functions ,
required-ordinal-functions , required-Boolean-functions .

- required-bindings : bindings .

(1) required-bindings = the disjoint-union of (
the required-constants,
the required-types,
the required-file-handling-procedures,
the required-dynamic-allocation-procedures,
the required-transfer-procedures,
the required-arithmetic-functions,
the required-transfer-functions,
the required-ordinal-functions,
the required-Boolean-functions) .

(2) required-constants = disjoint-union of (
map of "false" to false ,
map of "maxint" to maxint-value ,
map of "true" to true).

(3) required-types = disjoint-union of (
map of "boolean" to the boolean-type ,
map of "char" to the character-type ,
map of "integer" to the integer-type ,
map of "real" to the real-type ,
map of "text" to the text-file-type) .

(4) required-file-handling-procedures = disjoint-union of (
map of "rewrite" to procedure of (write-file-variable-mode,
abstraction of
| start generation of the given file-variable),
map of "put" to procedure of (write-file-variable-mode,
abstraction of
| advance generation of the given file-variable),
map of "reset" to procedure of (read-file-variable-mode,
abstraction of
| start inspection of the given file-variable),
map of "get" to procedure of (read-file-variable-mode,
abstraction of
| advance inspection of the given file-variable),
map of "read" to procedure of (read-file-variable-mode, multiple read-variable-mode,
abstraction of
| respectively investigate and read into the rest of the given argument$^+$
| from the given file-variable#1),
map of "write" to procedure of (write-file-variable-mode, multiple write-value-mode,
abstraction of
| respectively evaluate and write the rest of the given argument$^+$
| to the given file-variable#1),
map of "readln" to procedure of (read-file-variable-mode, multiple read-variable-mode,
abstraction of
| respectively investigate and read into the rest of the given argument$^+$
| from the given text-file-variable#1 and then
| skip to an end-of-line in the given text-file-variable#1),
map of "writeln" to procedure of (write-file-variable-mode, multiple write-value-mode,
abstraction of
| respectively evaluate and write the rest of the given argument$^+$
| to the given text-file-variable#1 and then
| write an end-of-line to the given text-file-variable#1),
map of "page" to procedure of (write-file-variable-mode,
abstraction of
| write an end-of-page to the given text-file-variable)) .

```
(5) required-dynamic-allocation-procedures = disjoint-union of (
      map of "new" to procedure of (variable-mode, multiple value-mode,
          abstraction of
                  give the given pointer-variable and
                  allocate a variable of the target-type of
                      the type possessed by the given pointer-variable
                or
                  give the given pointer-variable#1 and
                  allocate a variable of
                      ((the variant-record-type yielded by the target-type of
                          the type possessed by the given pointer-variable#1)
                      restricted by the ordinal-value+ yielded by the rest of the given data)
              then assign (a pointer to the given variable#2) to the given pointer-variable#1 ),
      map of "dispose" to procedure of (variable-mode, multiple value-mode,
          abstraction of
                check (the value attributed to the given pointer-variable#1
                    is the nil-pointer) and then
                report error#23
              or dispose of the variable yielded by
                  the target-variable of the pointer-value attributed to the given pointer-variable#1 ) ) .

(6) required-transfer-procedures = disjoint-union of (
      map of "pack" to procedure of (variable-mode, value-mode, variable-mode,
          abstraction of
                    give (the cardinality of the index-type of
                                the type possessed by the given array-variable#3) components
                        indexed from the given ordinal-value#2
                        of the given array-variable#1
                    reporting any given error
                  then
                    give the array-value of the values respectively attributed to the given variable+
                    reporting any given error
                and give the given array-variable#3
              then
                attribute the given array-value#1 to the given array-variable#2 ),
      map of "unpack" to procedure of (variable-mode, variable-mode, value-mode,
          abstraction of
                  give the value attributed to the given array-variable#1
                  reporting any given error
                and
                  give (the cardinality of the index-type of
                              the type possessed by the given array-variable#1) components
                      indexed from the given ordinal-value#3
                      of the given array-variable#2
                  reporting any given error
              then
                respectively attribute the components of the given array-value#1
                    to the variable+ yielded by the rest of the given data ) ) .
```

(7) required-arithmetic-functions = disjoint-union of (
    map of "abs" to pure-function of the absolute-value of the given number ,
    map of "sqr" to pure-function of the square of the given number ,
    map of "sin" to pure-function of the sine of the given number ,
    map of "cos" to pure-function of the cosine of the given number ,
    map of "exp" to pure-function of the exponential of the given number ,
    map of "ln" to pure-function of the natural-logarithm of the given number ,
    map of "sqrt" to pure-function of the square-root of the given number ,
    map of "arctan" to pure-function of the arctangent of the given number) .

(8) required-transfer-functions = disjoint-union of (
    map of "round" to pure-function of the rounded given real-number ,
    map of "trunc" to pure-function of the truncated given real-number) .

(9) required-ordinal-functions = disjoint-union of (
    map of "ord" to pure-function of the ordinal-number of the given ordinal-value ,
    map of "chr" to pure-function of the character of the given integer-number ,
    map of "succ" to pure-function of the successor of the given ordinal-value ,
    map of "pred" to pure-function of the predecessor of the given ordinal-value) .

(10) required-Boolean-functions = disjoint-union of (
    map of "odd" to pure-function of
        (the modulo of (the given integer-number, integer-number of 2)
        is integer-number of 1) ,
    map of "eof" to function of (read-file-variable-mode,
        abstraction of
        | give (end-of-file reached in the given text-file-variable) reporting any given error) ,
    map of "eoln" to function of (read-file-variable-mode,
        abstraction of
        | give (end-of-line reached in the given text-file-variable) reporting any given error) .

privately introduces: pure-function of _ ,
respectively investigate and read into _ from _ ,
respectively evaluate and write _ to _ .

- pure-function of _ ::
    yielder [of a value | error] [using the given value] → function ($total$) .

(11) pure-function of Y:yielder =
    function of (value-mode,
        abstraction of (give Y reporting any given error)) .

- respectively investigate and read into _ from _ ::
    yielder [of unevaluated-argument*], yielder [of a file-variable] →
    action [storing | diverging | jumping | erring] [using current storage] ($total$) .

(12) respectively investigate and read into t:yielder [of unevaluated-argument*]
    from f:yielder [of a file-variable] =
    | check (t is ())
    or
    | | enact the first of t then
    | | read into the given variable from f
    | and then respectively investigate and read into the rest of t from f .

- respectively evaluate and write _ to _ ::
    yielder [of unevaluated-argument*], yielder [of a file-variable] →
    action [storing | diverging | jumping | erring] [using current storage] ($total$) .

(13) respectively evaluate and write t:yielder [of unevaluated-argument*]
to f:yielder [of a file-variable] =
| check (t is ())
or
| | enact the first of t then
| | write the given value to f
| and then respectively evaluate and write the rest of t to f .

Symbol Index

[0] *Underlined page numbers indicate main properties and definitions of symbols; ordinary* Roman *page numbers indicate uses of symbols; italic page numbers indicate notes and informal explanations.*