1. Overview

1.1 Scope

This standard defines the Rosetta system-level description language. It specifies the concrete syntax, abstract syntax and denotational semantics of the language. It also specifies the predefined libraries, predefined packages and predefined domains for models of computation.

1.2 Purpose

The discipline of systems engineering benefits from formal models of systems under design. Such models allow systems engineers to analyze and predict properties of systems, including behavior, performance, power and adherence to requirements and constraints.

Large systems comprise a complex assembly of heterogeneous elements, including digital, analog and radio-frequency electronics, software, mechanical, hydraulic and thermal elements. Different models of computation are appropriate for representing aspects of these elements and their interactions. Models of computation include discrete and continuous state-based representations in time, space, frequency, or abstract state, and trace-based representations.

Rosetta is a language that can be used to describe aspects, or *facets*, of elements of a system in different models of computation and to describe the interactions between the facets. The language is extensible, allowing domains for specific modeling disciplines to be described as extensions of basic domains. The language has formal semantics, expressed using denotational semantic techniques, thus admitting of formal analysis of descriptions.

1.3 Structure of this standard

2. References

This clause defines ...

3. Definitions

This clause defines ...

4. Acronyms and abbreviations

This clause defines ...

5. Values and types

This clause defines the values that are denoted by Rosetta specification. It also defines the notions of types and subtypes.

5.1 The Rosetta value space

Specifications and expressions in Rosetta shall denote abstract values defined according to the mathematics of domain theory. The Rosetta value space shall be the domain of all denotable values, defined by the following equation:

$$Values = Numbers + Characters + Labels + Powerdomain + Unions + Functions + Facets \quad (1)$$

where

—*Values* is the domain of denotable values
—*Numbers* is the domain of complex number values, including complex values with infinite real or imaginary components
—*Characters* is the domain of Unicode character values
—*Labels* is the domain of label values
—*Powerdomain* is the powerdomain of values from *V*
—*Unions* is the disjoint union of disjoint unions of products of subdomains of *V*
—*Functions* is the domain of functions from subdomains of *V* to subdomains of *V*
—*Facets* is the domain of categories of coalgebras representing facets and facet-equivalents
—the operator + is the disjoint union operator on domains

In addition, each of the domains shall contain the undefined value denoted by the lexical token _|_.

5.2 Types

The domain of Rosetta types, *T*, shall be the union of:

—*Powerdomain*,
—*Facets*, and
—The subdomain of *Functions* containing functions that map to *T*

A Rosetta type shall be a value that is a member of *T*. The undefined value _|_ shall be a member of every type.

A type that is a member of *Powerdomain* or *Facets* shall be called a *monomorphic type*. A type that is a function whose domain values are types shall be called a *polymorphic type*. A type that is a function whose domain values are not types shall be called a *dependent type*.

NOTES:

1 — Since the definition of *T* is recursive, the value is defined by the fixed-point solution of the recursive equation for *T*.

2 — Usually, only facet values that are declared using domain declarations are used as types.

Given two types T_1 and T_2, T_1 shall be called a subtype of T_2 if and only if every member of T_1 is also a member of T_2. T_1 shall be called a proper subtype of T_2 if and only if T_1 is a subtype of T_2 and T_1 is not equal to T_2. T_1 shall be called a supertype of T_2 if and only if T_2 is a subtype of T_1. T_1 shall be called a proper supertype of T_2 if and only if T_2 is a proper subtype of T_1.

6. The Rosetta kernel language

The Rosetta kernel language shall be a subset of the full Rosetta language. The abstract syntax of the kernel language shall be represented by the type `rosetta.lang.reflect.kernel.kernel_nonterminal`. A specification in the kernel language shall be ascribed an interpretation by the denotational semantics in this standard. The denotation of a specification in the full language shall be the denotation of the equivalent kernel language specification derived thorough the process of elaboration (see 20.2).

7. Lexical Elements

This clause defines the lexical elements used to write a Rosetta specification. A Rosetta specification shall consist of one or more design files. A design file shall contain a sequence of lexical elements, each of which shall consist of one or more characters.

A lexical element shall be one of:

—A token
—A label
—A literal

7.1 Character set

The text of a Rosetta design file shall comprise a sequence of characters drawn from the Unicode character set defined in The Unicode Standard, Version 4.0.1 (hereafter referred to as the Unicode Standard). An implementation may choose to store and transmit characters of a Rosetta specification using using the UTF-32, UTF-16 or UTF-8 encodings defined in the Unicode Standard. However, all characters in the specification shall be interpretted as UTF-32 characters when the specification is processed by an implementation.

The means by which an implementation renders characters into glyphs is implementation dependent. In this standard, characters are specified using either character names or short identifiers, as defined in Unicode Standard. Where the character name is used, it is written using small capital letters.

NOTE — The Unicode UTF-32 character set is equivalent to the UCS-4 character set defined in ISO/IEC 10646.

7.1.1 Separators

A separator shall be a separator character, the end of a line or a comment. A separator character shall be one of:

—A character in the Separator class defined in the Unicode Standard
—The HORIZONTAL TABULATION character
—The VERTICAL TABULATION character
—The CARRIAGE RETURN character
—The LINE FEED character
—The FORM FEED character

The cause a line end is implementation defined. An implementation may interpret a specific control character or a specific sequence of control characters as a line end. Alternatively, it may determine a line end using some mechanism other than by interpretation of a character.

Lexical elements may be separated by one or more separators. If the characters of two successive lexical elements can, without an intervening separator, be interpreted as a single lexical element, one or more separators shall be included between the lexical elements. In any case, separators may be included between lexical tokens to enhance readability of a specification.

7.1.2 Comments

A comment shall be either a single-line comment or a delimited comment. The purpose of a comment is to document the specification for the human reader. The presence of a comment shall not influence the legality or semantics of a Rosetta specification.

A single line comment shall comprise a sequence of characters starting with two consecutive SOLIDUS characters (//) that are not included within a string literal and extending to the end of the line. If an implementation signifies the end of the line with one or more control characters, those control characters shall not form part of the single-line comment.

A delimited comment shall comprise a sequence of characters starting with a SOLIDUS character immediately followed by an ASTERISK character (/*), both not included within a string literal. The delimited comment shall extend to include the first subsequent pair consisting of an ASTERISK character immediately followed by a SOLIDUS character (*/). The closing character pair may be on the same line as the opening character pair or on a subsequent line.

Examples:

```
// A comment containing documentation.
// A comment that is too long for one line may be
// split over several lines

/* A comment that extends over
   more than one line. */

/* Delimited comments may be used to exclude part of a specification...
a :: real;  // A variable that we might want later.
   ... provided the excluded part contains no nested delimited comments. */
```

NOTE — Delimited comments do not nest. The first */ character pair encountered after an opening /* character pair closes the first-encountered delimited comment.

7.2 Tokens

A token shall be one of

—A keyword
—A delimiter

7.2.1 Keywords

A keyword is a sequence of characters that forms a token. If a sequence of characters conforms to the rules for forming a label and is included in the list of keywords below, it shall be interpreted as a keyword. The keywords are listed in Table 1. A sequence of characters that differs from a keyword listed in Table 1 only in the case of letters shall be considered equivalent to that keyword.

Table 1—Rosetta keywords

all	enumeration	mod
and	export	nand
assumptions	facet	nor
be	if	not
begin	implications	or
body	implies	package
case	in	rem
component	instance	sub
constant	interaction	subtype
data	interface	then
definitions	is	type
div	justification	use
domain	let	where
else	library	with
elsif	max	xnor
end	min	xor

7.2.2 Delimiters

A delimiter token is a sequence of characters listed in Table 2. A sequence of characters listed in Table 2 shall be interpreted as a single delimiter, unless it occurs as part of a larger sequence of characters forming a lexical element. Sequences of characters listed together in a cell of Table 2 shall be considered as equivalent delimiter tokens.

Table 2—Delimiters

<table>
<tr><td>"</td><td>`</td><td>== ≡</td></tr>
<tr><td>'</td><td><<</td><td>=> ⇒</td></tr>
<tr><td>%</td><td>>></td><td><= ⇐</td></tr>
<tr><td>&</td><td><</td><td>/= ≠</td></tr>
<tr><td>&&</td><td>=</td><td>=< ≤</td></tr>
<tr><td>#</td><td>></td><td>>= ≥</td></tr>
<tr><td>(</td><td>@</td><td>::</td></tr>
<tr><td>)</td><td>[</td><td>,..</td></tr>
<tr><td>*</td><td>]</td><td>||</td></tr>
<tr><td>+</td><td>^</td><td>-> →</td></tr>
<tr><td>,</td><td>_</td><td><*</td></tr>
<tr><td>-</td><td>{</td><td>*></td></tr>
<tr><td>.</td><td>|</td><td>{*</td></tr>
<tr><td>/</td><td>}</td><td>*}</td></tr>
<tr><td>:</td><td>>>></td><td>~</td></tr>
<tr><td>;</td><td>=>></td><td></td></tr>
</table>

7.3 Labels

A label shall be a sequence of characters that may be used as a name. A label shall be a simple label or an operator interpretation label.

Two labels that differ only by one or more characters in the formatting code class shall be considered the same label. Furthermore, two labels that differ only in the case of letters shall be considered the same label. Two labels differ only in case if and only if they fold to the same Unicode character sequence when folded using the simple-case-folding mapping defined in Section 3.13 of the Unicode Standard.

Concrete syntax:

label ::=
  simple_label | operator_interpretation_label

7.3.1 Simple labels

A simple label may be used as the name of an item or a term.

Concrete syntax:

simple_label ::=
  label_start_character { [label_connecting_character] label_extending_character }

A label start character is a Unicode character in one of the classes uppercase letter, lowercase letter, titlecase letter, modifier letter, other letter or letter number defined in the Unicode Standard. A label extending character is a label start character or a character in one of the classes nonspacing mark, spacing combining mark, decimal number or formatting code defined in the Unicode Standard. A label connecting character is a character in the class connector punctuation defined in the Unicode Standard.

A sequence of characters that conforms to the rules for a simple label and that can be interpreted as a keyword shall be interpreted as a keyword and not as a simple label.

7.3.2 Operator interpretation labels

An operator interpretation label is used as the name of a function that provides a semantic interpretation of an operator. The operator interpretation labels are listed in Table 3. A sequence of characters that differs from an operator interpretation label listed in Table 3 only in the case of letters shall be considered equivalent to that operator interpretation label.

Table 3—Operator interpretation labels

`not__`	`__nor__`	`__sub__`
`+__`	`__xor__`	`__+__`
`-__`	`__xnor__`	`__-__`
`*__`	`__max__`	`__&__`
`#__`	`__and__`	`__&&__`
`%__`	`__nand__`	`__||__`
`~__`	`__min__`	`__*__`
`__>>>__`	`__=__`	`__/__`
`__=>>__`	`__/=__`	`__mod__`
`__==__`	`__<__`	`__div__`
`__=>__`	`__>__`	`__rem__`
`__implies__`	`__=<__`	`__^__`
`__<=__`	`__>=__`	`__::__`
`__or__`	`__in__`	

7.4 Literals

A literal is a sequence of characters that denotes a value.

Concrete syntax:

literal ::=
  undefined_literal | real_literal | infinity_literal | character_literal | string_literal | bitvector_literal

7.4.1 The undefined literal

The undefined literal shall denote the undefined value ⊥ in the Rosetta value space.

Concrete syntax:

undefined_literal ::=
_ | _ | ⊥

7.4.2 The infinity literal

The infinity literal shall denote the infinite number value in the Rosetta value space.

Concrete syntax:

infinity_literal ::=
∞

7.4.3 Real literals

A real literal denotes a value of the type .

Concrete syntax:

real_literal ::=
decimal_literal | based_literal

decimal_literal ::=
decimal_digits [. decimal_digits] [exponent]

exponent ::=
(e | E) [+ | -] decimal_digits

decimal_digits ::=
decimal_digit { decimal_digit }

decimal_digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

based_literal ::=
base \ based_digits [. based_digits] \ [exponent]

base ::=
decimal_digits

based_digits ::=
based_digit { based_digit }

based_digit ::=
0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | a | A | b | B | c | C | d | D | e | E | f | F

A decimal literal shall consist of a sequence of decimal digits representing the whole-number part of a value in decimal radix, an optional post-decimal-point sequence of decimal digits representing the fractional part of the value in decimal radix, and an optional exponent.

A based literal shall consist of a sequence of decimal digits representing the radix of a value, a sequence of based digits representing the whole-number part of a value in the specified radix, an optional post-radix-point sequence of based digits representing the fractional part of the value in the specified radix, and an optional exponent.

Both the base number and the exponent shall be interpretted as decimal numbers. The value denoted by the literal shall be the value specified by the whole-number and post-radix-point parts multiplied by the radix raised to the power of the exponent.

The based digits a and A shall have the value 10, b and B shall have the value 11, c and C shall have the value 12, d and D shall have the value 13, e and E shall have the value 14, and f and F shall have the value 15. In a based literal, the radix shall be in the range 2 to 16 inclusive, and only the based digits between 0 inclusive and the radix exclusive shall be used.

NOTE — No distinction is made between real literals and literal values of type `rational`, `integer`, `natural` or `bit`, as all of these types are subtypes of `real`. For example, the real literal 3.0 has the integer value 3, and so is a member of the types `real`, `rational`, `integer` and `natural`.

7.4.4 Character literals

A character literal denotes a value of the type `rosetta.lang.prelude.character`.

Concrete syntax:

character_literal ::=
  graphic_character_literal | character_code_literal

graphic_character_literal ::=
  ' character '

character_code_literal ::=
  ' character_code_specifier '

character_code_specifier
  short_character_code_specifier | full_character_code_specifier

short_character_code_specifier ::=
  (u | U) + [[based_digit] based_digit] based_digit based_digit based_digit based_digit

full_character_code_specifier ::=
  (u | U) - based_digit based_digit based_digit based_digit based_digit based_digit based_digit based_digit

The value denoted by a graphic character literal shall be the Unicode character enclosed between the two APOSTROPHE characters.

The value denoted by a character code literal shall be the Unicode character whose character code is an eight-digit hexadecimal value specified by the character code specifier. A short character code specifier shall specify a hexadecimal value formed from the based digits in the short character code specifier extended to the left with sufficient 0 digits to form a total of eight digits. A full character code specifier shall specify all eight digits of the character code.

Examples:

```
'A'            // LATIN CAPITAL LETTER A
' '            // SPACE
'''            // APOSTROPHE
'U+00B1'       // PLUS-MINUS SIGN (±)
'u+274F'       // LOWER RIGHT DROP-SHADOWED WHITE SQUARE
'U+10347’      // GOTHIC LETTER IGGWS
'U+00FFFF'     // not a character
'U-0001040F'   // DESERET CAPITAL LETTER YEE
```


NOTE — A graphic character literal may contain a Unicode character that cannot be rendered by an implementation.

7.4.5 String literals

A string literal shall denote a value of the type `rosetta.lang.prelude.string`. A string literal shall be formed by enclosing a sequence of zero or more Unicode characters between two QUOTATION MARK characters.

Concrete syntax:

string_literal ::=
  " { character } "

The value denoted by a string literal shall be the sequence of characters contained between the QUOTATION MARK characters. If there are no characters between the QUOTATION MARK characters, the value denoted by the string literal shall be the empty sequence. Otherwise, the first character in the string literal shall be the first element of the value and subsequent characters in the string literal shall be subsequent elements of the value, in the same order, except that two successive QUOTATION MARK characters in a string literal shall denote a single occurrence of a QUOTATION MARK character in the sequence of characters denoted by the string literal. The first QUOTATION MARK character that is not immediately followed by another QUOTATION MARK character shall be the terminating QUOTATION MARK character of the string literal.

Examples:

```
"Enter command: "    // A prompt string
"¿Que?"              // Almost any character can appear in a string literal
""                   // An empty string
"++""++"             // A string containing a QUOTATION MARK character
"This string extends "
   & "over two lines."
"This string contains" & 'U+000A' & "a line separator character."
```

NOTE — A string literal cannot extend over more than one line. A string can be written on two lines by concatenating two substrings, each written on a separate line. A string containing a line separator character can be formed by concatenating string literals and an character code literal denoting the line separator character.

7.4.6 Bitvector literals

A bitvector literal shall denote a value of the type `rosetta.lang.prelude.bitvector`.

Concrete syntax:

bitvector_literal ::=
  binary_bitvector_literal | octal_bitvector_literal | hexadecimal_bitvector_literal

binary_bitvector_literal ::=
  (b | B) " binary_digit { binary_digit } "

binary_digit ::=
  0 | 1

octal_bitvector_literal ::=
  (o | O) " octal_digit {octal_digit } "

octal_digit ::=

0 | 1 | 2 | 3 | 4 | 5 | 6 | 7

hexadecimal_bitvector_literal ::=
(x | X) " based_digit {based_digit } "

The binary digit 0 shall denote bitvector values as listed in Table 4.

Table 4—Values denoted by binary digits

Binary digit	Bitvector value
0	[0]
1	[1]

The octal digits shall denote bitvector values as listed in Table 5.

Table 5—Values denoted by octal digits

Octal digit	Bitvector value	Octal digit	Bitvector value
0	[0, 0, 0]	4	[0, 0, 1]
1	[1, 0, 0]	5	[1, 0, 1]
2	[0, 1, 0]	6	[0, 1, 1]
3	[1, 1, 0]	7	[1, 1, 1]

The hexadecimal digits shall denote bitvector values as listed in Table 6.

Table 6—Values denoted by hexadecimal digits

Hexadecimal digit	Bitvector value	Hexadecimal digit	Bitvector value
0	[0, 0, 0, 0]	8	[0, 0, 0, 1]
1	[1, 0, 0, 0]	9	[1, 0, 0, 1]
2	[0, 1, 0, 0]	a A	[0, 1, 0, 1]
3	[1, 1, 0, 0]	b B	[1, 1, 0, 1]
4	[0, 0, 1, 0]	c C	[0, 0, 1, 1]
5	[1, 0, 1, 0]	d D	[1, 0, 1, 1]
6	[0, 1, 1, 0]	e E	[0, 1, 1, 1]
7	[1, 1, 1, 0]	f F	[1, 1, 1, 1]

Let the rightmost digit of a bitvector literal be called d_0, and the remaining digits from right to left be called d_1, d_2, and so on. Let the bitvector value denoted by d_0 be called s_0, by d_1 be s_1, and so on. Then the bitvector literal shall denote the concatenation

s_0 & s_1 & s_2 & ...

Examples:

```
b"001101" == [1, 0, 1, 1, 0, 0]
O"35" == [1, 0, 1, 1, 1, 0]
x"71" == [1, 0, 0, 0, 1, 1, 1, 0]
```

NOTE — An octal bitvector literal always denotes a sequence that is a multiple of three in length, and a hexadecimal bitvector literal always denotes a sequence that is a multiple of four in length.

8. Expressions

This clause defines the various forms of Rosetta expressions. The abstract syntax definitions in this clause are declared in the package `rosetta.lang.reflect.abstract_syntax`. The simplification definitions are declared in the package `rosetta.lang.reflect.simplification`.

An expression shall denote a value. Expressions shall be formed by combining primary expressions with operators. The concrete syntax for expressions shall determine the precedence of operators. Where operators in an expression are of equal precedence, the expression shall be interpreted by applying operators in left-to-right order.

Concrete Syntax:

expression ::=
  precedence_0_expression

optional_expression_list ::=
  [expression { `,` expression }]

precedence_0_expression ::=
  precedence_1_expression
  | precedence_0_operation

precedence_0_operation ::=
  *left_operand:*precedence_0_expression *operator:*precedence_0_operator
    *right_operand:*precedence_1_expression

precedence_0_operator ::=
  `>>>` | `=>>`

precedence_1_expression ::=
  precedence_2_expression
  | precedence_1_operation

precedence_1_operation ::=
  *left_operand:*precedence_1_expression *operator:*precedence_1_operator
    *right_operand:*precedence_2_expression

precedence_1_operator ::=
  `==`

precedence_2_expression ::=
  precedence_3_expression
  | precedence_2_operation

precedence_2_operation ::=
  *left_operand:*precedence_2_expression *operator:*precedence_2_operator
    *right_operand:*precedence_3_expression

precedence_2_operator ::=
  `=>` | **`implies`** | `<=`

precedence_3_expression ::=
  precedence_4_expression

| precedence_3_operation

precedence_3_operation ::=
*left_operand:*precedence_3_expression *operator:*precedence_3_operator
*right_operand:*precedence_4_expression

precedence_3_operator ::=
or | **nor** | **xor** | **xnor** | **max**

precedence_4_expression ::=
precedence_5_expression
| precedence_4_operation

precedence_4_operation ::=
*left_operand:*precedence_4_expression *operator:*precedence_4_operator
*right_operand:*precedence_5_expression

precedence_4_operator ::=
and | **nand** | **min**

precedence_5_expression ::=
precedence_6_expression
| precedence_5_operation

precedence_5_operation ::=
*left_operand:*precedence_5_expression *operator:*precedence_5_operator
*right_operand:*precedence_6_expression

precedence_5_operator ::=
= | /= | < | =< | > | >= | **in**

precedence_6_expression ::=
precedence_7_expression
| precedence_6_operation

precedence_6_operation ::=
*left_operand:*precedence_6_expression *operator:*precedence_6_operator
*right_operand:*precedence_7_expression

precedence_6_operator ::=
sub

precedence_7_expression ::=
precedence_8_expression
| precedence_7_operation

precedence_7_operation ::=
*left_operand:*precedence_7_expression *operator:*precedence_7_operator
*right_operand:*precedence_8_expression

precedence_7_operator ::=
+ | - | & | && | ||

precedence_8_expression ::=

precedence_9_expression
| precedence_8_operation

precedence_8_operation ::=
*left_operand:*precedence_8_expression *operator:*precedence_8_operator
*right_operand:*precedence_9_expression

precedence_8_operator ::=
`*` | `/` | **`div`** | **`mod`** | **`rem`**

precedence_9_expression ::=
precedence_10_expression
| precedence_9_operation

precedence_9_operation ::=
*left_operand:*precedence_9_expression *operator:*precedence_9_operator
*right_operand:*precedence_10_expression

precedence_9_operator ::=
`^` | `#`

precedence_10_expression ::=
precedence_11_expression
| precedence_10_operation

precedence_10_operation ::=
*operator:*precedence_10_operator *operand:*precedence_10_expression

precedence_10_operator ::=
`not` | `+` | `-` | `%` | `#` | `~`

precedence_11_expression ::=
precedence_12_expression
| at_operation

at_operation ::=
*left_operand:*name `@` *right_operand:*precedence_12_expression

precedence_12_expression ::=
precedence_13_expression
| function_application
| facet_instantiation

precedence_13_expression ::=
primary
| precedence_13_operation

precedence_13_operation ::=
*left_operand:*primary *operator:*precedence_13_operator *right_operand:*primary

precedence_13_operator ::=
`::`

primary ::=

parenthesized_expression
| literal
| name
| tick_operation
| anonymous_function
| anonymous_facet
| set_formation
| multiset_formation
| sequence_formation
| type_formation
| let_expression
| if_expression
| case_expression
| quantified_expression
| template_expression

Abstract syntax:

```
prefix_unary_operation :: ...

infix_binary_operation :: ...

expression :: ...

optional_expression :: ...

optional_expression_list :: ...

precedence_0_expression :: ...

precedence_0_operation :: ...

precedence_0_operator :: ...

precedence_1_expression :: ...

precedence_1_operation :: ...

precedence_1_operator :: ...

precedence_2_expression :: ...

precedence_2_operation :: ...

precedence_2_operator :: ...

precedence_3_expression :: ...

precedence_3_operation :: ...

precedence_3_operator :: ...

precedence_4_expression :: ...
```

```
precedence_4_operation :: ...

precedence_4_operator :: ...

precedence_5_expression :: ...

precedence_5_operation :: ...

precedence_5_operator :: ...

precedence_6_expression :: ...

precedence_6_infix_binary_operation :: ...

precedence_6_binary_operator :: ...

precedence_7_expression :: ...

precedence_7_operation :: ...

precedence_7_operator :: ...

precedence_8_expression :: ...

precedence_8_operation :: ...

precedence_8_operator :: ...

precedence_9_expression :: ...

precedence_9_operation :: ...

precedence_9_operator :: ...

precedence_10_expression :: ...

precedence_10_operation :: ...

precedence_10_operator :: ...

precedence_11_expression :: ...

at_operation :: ...

precedence_12_expression :: ...

precedence_13_expression :: ...

precedence_13_operation :: ...

precedence_13_operator :: ...
```

NOTE — There is no abstract syntax corresponding to the concrete nonterminal `template_expression`, since a template expression is expanded into an equivalent anstract syntax tree during parsing of a Rosetta specification.

Simplification:

```
simplified_expression :: ...

simplified_optional_expression_list :: ...

simplified_singleton_expression_list :: ...

simplify_expression ...
```

NOTE — The function `simplify_expression` yields a simplified expression according to the subtype of the given Rosetta expression. In the case of an at operation, the result is a simplified tick operation in which the operand is the left operand of the at operation, and the tick operator list contains one element where the tick label is “at”, and the tick arguement list is a singleton list containing the right operand of the at operation. In the case of an expression that is a literal or a name, the expression is also the simplified expression.

Examples:

The Rosetta expression `#e` simplifies to the expression `#__ ( e )`.

The Rosetta expression `e1 * e2` simplifies to the expression `__*__ ( e1, e2 )`.

The Rosetta expression `n @ e` simplifies to the expression `n'at ( e )`.

Name Expansion:

```
resolved_expression :: ...

resolved_optional_expression_list :: ...

resolved_singleton_expression_list :: ...

expand_names_in_expression ...
```

NOTE — The function `expand_names_in_expression` yields a resolved expression according to the subtype of the given simplified expression. If the given expression is a name that is a label list, the resolved expression is an equivalent value of `rosetta.lang.reflect.name_expansion.resolved_reference`, while if the given expression is a name that is a resolved reference then the given name is yielded as is.

8.1 Operators

Static semantics:

For each operator, their shall be a corresponding operator interpretation function label, given in Table 7. If a given operator occurs in an expression, an item with the corresponding operator interpretation function label shall be directly visible at the place of the expression, and that item shall denote a function, called the operator interpretation function. An expression that is an operation shall be equivalent to an expression that is a function application in which the applied function expression is the operator interpretation function name. For unary operators, the operator interpretation function shall be applied to the single operand of the operator. For binary operators, the operator interpretation function shall be applied with the left operand as the first argument and the right operand as the second argument.

Table 7—Operator interpretation function names

<table>
<tr><th>Operator</th><th>Operator interpretation function label</th><th>Operator</th><th>Operator interpretation function label</th></tr>
<tr><td>>>></td><td>__>>>__</td><td>+ (binary)</td><td>__+__</td></tr>
<tr><td>=>></td><td>__=>>__</td><td>- (binary)</td><td>__-__</td></tr>
<tr><td>==</td><td>__==__</td><td>&</td><td>__&__</td></tr>
<tr><td>=></td><td>__=>__</td><td>&&</td><td>__&&__</td></tr>
<tr><td>implies</td><td>__implies__</td><td>||</td><td>__||__</td></tr>
<tr><td><=</td><td>__<=__</td><td>*</td><td>__*__</td></tr>
<tr><td>or</td><td>__or__</td><td>/</td><td>__/__</td></tr>
<tr><td>nor</td><td>__nor__</td><td>div</td><td>__div__</td></tr>
<tr><td>xor</td><td>__xor__</td><td>mod</td><td>__mod__</td></tr>
<tr><td>xnor</td><td>__xnor__</td><td>rem</td><td>__rem__</td></tr>
<tr><td>max</td><td>__max__</td><td>^</td><td>__^__</td></tr>
<tr><td>and</td><td>__and__</td><td># (binary)</td><td>__#__</td></tr>
<tr><td>nand</td><td>__nand__</td><td>not</td><td>not__</td></tr>
<tr><td>min</td><td>__min__</td><td>+ (unary)</td><td>+__</td></tr>
<tr><td>=</td><td>__=__</td><td>- (unary)</td><td>-__</td></tr>
<tr><td>/=</td><td>__/=__</td><td>%</td><td>%__</td></tr>
<tr><td><</td><td>__<__</td><td># (unary)</td><td>#__</td></tr>
<tr><td>=<</td><td>__=<__</td><td>~</td><td>~__</td></tr>
<tr><td>></td><td>__>__</td><td>::</td><td>__::__</td></tr>
<tr><td>>=</td><td>__>=__</td><td></td><td></td></tr>
<tr><td>in</td><td>__in__</td><td></td><td></td></tr>
<tr><td>sub</td><td>__sub__</td><td></td><td></td></tr>
</table>

NOTE — Operator interpretation functions for all operators except `>>>` and `=>>` are defined in the package `rosetta.lang.prelude`, which is used in most domains. Hence the functions will usually be directly visible. Operator interpretation functions for `>>>` and `=>>` are defined in the package `rosetta.lang.reflect.semantics`. These functions can be made directly visible by means of a use clause referring to the package.

Simplification:

```
simplify_prefix_unary_operation ...
```

NOTE — The function `simplify_prefix_unary_operation` yields the simplification of a function application in which the applied function expression is a label formed by appending "__" to the operator token and the argument is the operand expression.

```
simplify_infix_binary_operation ...
```


NOTE — The function `simplify_infix_binary_operation` yields the simplification of a function application in which the applied function expression is a label formed by prefixing and appending “__” to the operator token, the first argument is the left operand expression, and the second arguement is the right operand expression.

8.2 Primaries

This clause defines the various forms of primary that shall be used to form expressions.

8.2.1 Parenthesized expressions

Concrete syntax:

parenthesized_expression ::=
  (*contained_expression:*expression)

Abstract syntax:

```
parenthesized_expression :: ...
```

Simplification:

```
simplify_parenthesized_expression ...
```

NOTE — The function `simplify_parenthesized_expression` yields the simplification of the contained expression.

8.2.2 Literals

The value denoted by a primary that is a literal shall be the value of the literal.

NOTE — Literals are defined in 7.4.

8.2.3 Names

To do: sort this out properly.

Concrete syntax:

name ::=
  label [{ . label }]

Abstract syntax:

```
name :: ...
```

Name Expansion:

```
resolved_reference :: ...
```

```
referable_item :: ...
```

NOTE — The type `referable_item` is the union of those resolved AST types that declare an item or term or define a parameter.

```
get_resolved_reference ...
```


NOTE — Given a visibility context corresponding to the declarative region in which a label list occurs as a name, and a label list occuring as a name, the function `get_resolved_reference` yields a value of `rosetta.lang.reflect.name_expansion.resolved_reference` that denotes the label of the declared item or declared term or defined parameter that is visible within the given context and is refered to by the given name.

8.2.4 Tick operations

Concrete syntax:

tick_operation ::=
*operand:*name *operators:*tick_operator_list

tick_operator_list ::=
tick_operator { tick_operator }

tick_operator ::=
' [*tick_label:*label [(*tick_arguments:*expression_list)]]

Abstract syntax:

```
tick_operation :: ...

tick_operator_list :: ...

tick_operator :: ...
```

Simplification:

```
simplified_tick_operation :: ...

simplified_tick_operator :: ...

simplified_tick_operator_list :: ...

simplify_tick_operation ...
```

NOTE — The function `simplify_tick_operation` yields a simplified tick operation with the same left operand. For each tick operator, the result simplified tick operation has a corresponding tick operator with the same operator label, but with each argument expression simplified.

Name Expansion:

```
resolved_tick_operation :: ...

resolved_tick_operator :: ...

resolved_tick_operator_list :: ...

resolve_tick_operation ...
```

NOTE — The function `resolve_tick_operation` yields a resolved tick operation where the left operand is a resolved reference corresponding to the name denoted by the left operand. For each simplified tick operator, the result has a corresponding tick operator with the same operator label, but where any names within each argument expression are expanded.

8.2.5 Set formations

Concrete syntax:

set_formation ::=
  { *set_formation_contents:*set_formation_content }

set_formation_content ::=
  optional_expression_list
  | range

range ::=
  *lower_bound:*expression , . . *upper_bound:*expression

Abstract syntax:

```
set_formation :: ...

set_formation_content :: ...

range :: ...
```

Simplification:

```
simplify_set_formation ...
```

NOTE — The function `simplify_set_formation` yields a simplified expression formed as follows:

—If the set formation content is an optional expression list, each element of the list is wrapped as the argument in a function application expression in which the applied function is the name `rosetta.lang.prelude.singleton_set`. The resulting expressions are wrapped as arguments in nested function application expressions in which each applied function is the name `rosetta.lang.prelude.set_union`, starting with the name `rosetta.lang.prelude.empty_set` as the degenerate case for an empty expression list. The nested function application expression is simplifed to yield the final result.
—If the set formation content is a range, the result is the simplification of a function application in which the applied function is the name `rosetta.lang.prelude.range_set`, the first argument is the lower bound of the set content, and the second argument is the upper bound of the set content.

Examples:

The Rosetta expression `{ }` simplifies to the expression

```
rosetta.lang.prelude.empty_set
```

The Rosetta expression `{ e }` simplifies to the expression

```
rosetta.lang.prelude.set_union (
  rosetta.lang.prelude.empty_set,
  rosetta.lang.prelude.singleton_set ( e ) )
```

The Rosetta expression `{ e1, e2, e3 }` simplifies to the expression

```
rosetta.lang.prelude.set_union (
  rosetta.lang.prelude.set_union (
    rosetta.lang.prelude.set_union (
```

```
        rosetta.lang.prelude.empty_set,
        rosetta.lang.prelude.singleton_set ( e1 ) ),
      rosetta.lang.prelude.singleton_set ( e2 ) ),
    rosetta.lang.prelude.singleton_set ( e3 ) )
```

The Rosetta expression `{ e1,..e2 }` simplifies to the expression

```
rosetta.lang.prelude.range_set ( e1, e2 )
```

8.2.6 Multiset formations

Concrete syntax:

multiset_formation ::=
`{*` *multiset_formation_contents:*multiset_formation_content `*}`

multiset_formation_content ::=
optional_multiset_expression_list
| range

optional_multiset_expression_list ::=
[multiset_expression { `,` multiset_expression }]

multiset_expression ::=
*occurrences:*optional_occurrence_count *value:*expression

optional_occurrence_count ::=
[expression `:`]

Abstract syntax:

```
multiset_formation :: ...

multiset_formation_content :: ...

optional_multiset_expression_list :: ...

multiset_expression :: ...
```

Simplification:

```
simplify_multiset_formation ...
```

NOTE — The function `simplify_multiset_formation` yields a simplified expression formed as follows:

—If the multiset formation content is an optional multiset expression list, each element of the list is wrapped as the second argument in a function application expression in which the applied function is the name `rosetta.lang.prelude.single_value_multiset`. The first argument is either the occurrence count, if present, or the literal 1 otherwise. The resulting expressions are wrapped as arguments in nested function application expressions in which each applied function is the name `rosetta.lang.prelude.multiset_union`, starting with the name `rosetta.lang.prelude.empty_multiset` as the degenerate case for an empty multiset expression list. The nested function application expression is simplifed to yield the final result.
—If the multiset formation content is a range, the result is the simplification of a function application in which the applied function is the name `rosetta.lang.prelude.range_multiset`, the first argument is the lower bound of the multiset content, and the second argument is the upper bound of the multiset content.

Examples:

The Rosetta expression `{* *}` simplifies to the expression

```
rosetta.lang.prelude.empty_multiset
```

The Rosetta expression `{* c:e *}` simplifies to the expression

```
rosetta.lang.prelude.multiset_union (
  rosetta.lang.prelude.empty_multiset,
  rosetta.lang.prelude.single_value_multiset ( c, e ) )
```

The Rosetta expression `{* e *}` simplifies to the expression

```
rosetta.lang.prelude.multiset_union (
  rosetta.lang.prelude.empty_multiset,
  rosetta.lang.prelude.single_value_multiset ( 1, e ) )
```

The Rosetta expression `{* c1:e1, c2:e2, c3:e3 *}` simplifies to the expression

```
rosetta.lang.prelude.multiset_union (
  rosetta.lang.prelude.multiset_union (
    rosetta.lang.prelude.multiset_union (
      rosetta.lang.prelude.empty_multiset,
      rosetta.lang.prelude.single_value_multiset ( c1, e1 ) ),
    rosetta.lang.prelude.single_value_multiset ( c2, e2 ) ),
  rosetta.lang.prelude.single_value_multiset ( c3, e3 ) )
```

The Rosetta expression `{* e1,..e2 *}` simplifies to the expression

```
rosetta.lang.prelude.range_multiset ( e1, e2 )
```

8.2.7 Sequence formations

Concrete syntax:

sequence_formation ::=
  `[` *sequence_content:*sequence_formation_content `]`

sequence_formation_content ::=
  optional_expression_list
  | range

Abstract syntax:

```
sequence_formation :: ...

sequence_formation_content :: ...
```

Simplification:

```
simplify_sequence_formation ...
```

NOTE — The function `simplify_sequence_formation` yields a simplified expression formed as follows:

—If the sequence formation content is an optional expression list, the expressions are wrapped as arguments in nested function application expressions in which each applied function is the name `rosetta.lang.prelude.cons`, starting with the name `rosetta.lang.prelude.empty_sequence` as the degenerate case for an empty expression list. The nested function application expression is simplifed to yield the final result.

—If the sequence formation content is a range, the result is the simplification of a function application in which the applied function is the name `rosetta.lang.prelude.range_sequence`, the first argument is the lower bound of the sequence content, and the second argument is the upper bound of the sequence content.

Examples:

The Rosetta expression `[ ]` simplifies to the expression

```
rosetta.lang.prelude.empty_sequence
```

The Rosetta expression `[ e ]` simplifies to the expression

```
rosetta.lang.prelude.cons (
  e,
  rosetta.lang.prelude.empty_sequence )
```

The Rosetta expression `[ e1, e2, e3 ]` simplifies to the expression

```
rosetta.lang.prelude.cons (
  e1,
  rosetta.lang.prelude.cons (
    e2
    rosetta.lang.prelude.cons (
      e3,
      rosetta.lang.prelude.empty_sequence ) ) )
```

The Rosetta expression `[ e1,..e2 ]` simplifies to the expression

```
rosetta.lang.prelude.range_sequence ( e1, e2 )
```

8.2.8 Type formations

Concrete syntax:

type_formation ::=
  subtype_formation
  | universal_type_formation
  | function_type_formation

subtype_formation ::=
  `subtype` `(` *base_type:*expression `)`

universal_type_formation ::=
  `type`

Abstract syntax:

```
type_formation :: ...

subtype_formation :: ...
```



```
universal_type_formation :: ...
```

Simplification:

To do: revise the following to take account of types that are functions returning types. Specifically, having **`type`** simplify to set(universal) is not sufficient, since types denoted by functions returning types are not sets.

```
simplify_type_formation ...
```

NOTE — If the type formation is a subtype formation or a universal type formation, the function `simplify_type_formation` yields the simplification of a function application in which the applied function is the name `rosetta.lang.prelude.set` and the argument is the base type expression of the subtype formation or the name `rosetta.lang.prelude.universal`, respectively. If the type formation is a function type formation, the function `simplify_type_formation` yields the simplification of the function type formation.

Examples:

The Rosetta expression **`subtype`** `( T )` simplifies to the expression

```
rosetta.lang.prelude.set ( T )
```

The Rosetta expression **`type`** simplifies to the expression

```
rosetta.lang.prelude.set ( rosetta.lang.prelude.universal )
```

8.2.9 Let expressions

Concrete syntax:

let_expression ::=
  `let`
    *binding_list:*let_binding_list
  `in`
    *encapsulated_expression:*expression
  `end let`

let_binding_list ::=
  let_binding { let_binding_list }

let_binding ::=
  let_function_binding
  | let_parameter_binding

let_function_binding ::=
  *function_label:*label *signature:*function_signature **`be`**
    *encapsulated_expression:*expression `;`

let_parameter_binding ::=
  *parameter:*parameter_definition **`be`** *bound_expression:*expression `;`

Abstract syntax:

```
let_expression :: ...
```

```
let_binding_list :: ...
```

```
let_binding :: ...

let_function_binding :: ...

let_parameter_binding :: ...
```

Static semantics:

The labels of all of the let bindings in a let expression shall be distinct.

Simplification:

```
simplified_let_expression :: ...

simplified_let_binding_list :: ...

simplified_let_binding :: ...

labels_from_simplified_let_binding_list ...
```

NOTE — The function `labels_from_simplifed_list_binding_list` yields a list of the labels of the parameters in the given simplified let binding list.

```
simplify_let_expression ...
```

NOTE — The function `simplify_let_expression` yields a let expression in which the elements of the binding list are the simplification of the binding list elements of the given let expression, and the encapsulated expression is the simplification of the encapsulated expression of the given let expression.

```
simplify_let_binding ...
```

NOTE — The function `simplify_let_binding` yields a simplified let function binding if the let binding is a let function binding, or a simplified let parameter binding if the let binding is a let parameter binding.

```
simplify_let_function_binding ...
```

NOTE — The function `simplify_let_function_binding` yields the simplification of a let parameter binding in which the parameter label is the label of the given let function binding, the parameter type is a function type formation with the same signature as the given let function binding. The and the bound expression is an anonymous function in which the signature is that of the given let function binding and the encapsulated expression is that of the given let function binding.

```
simplify_let_parameter_binding ...
```

NOTE — The function `simplify_let_parameter_binding` yields a let parameter binding in which the parameter label is that of the given let parameter binding, the parameter type is the simplification of the type of the given let parameter binding, and the bound expression is the simplification of the bound expression of the given let parameter binding.

Example:

The Rosetta expression

```
let
  f ( x :: T ) :: R be e1
in
  e2
```



```
end let
```

simplifies to the expression

```
let
  f :: <* ( x :: T ) :: R > be <* ( x :: T ) :: R is e1 *>
in
  e2
end let
```

Name expansion:

```
resolved_let_expression :: ...

resolved_let_binding_list :: ...

resolved_let_binding :: ...

expand_names_in_let_expression ...
```

NOTE — The function `expand_names_in_let_expression` yields a resolved let expression in which the elements of the binding list are the resolved let bindings corresponding to the bindings of the given simplified let expression, and the encapsulated expression is the resolved expression corresponding to the encapsulated expression of the given simplified let expression in which any names are resolved references.

```
expand_names_in_let_binding ...
```

NOTE — The function `expand_names_in_let_binding` yields a resolved let binding in which the parameter label is that of the given simplified let binding, the parameter type corresponds to the type of the given simplified let binding in which any names are resolved references, and the bound expression corresponds to the bound expression of the given simplified let binding in which any names are resolved references.

Denotational semantics:

The value denoted by a let expression shall be the value denoted by the encapsulated expression interpreted in a context formed by adding bindings for the parameter labels in the binding list to the head of the context in which the let expression is interpreted. The binding for each parameter label shall consist of an item whose label is the parameter label and whose value is the value denoted by the corresponding bound expression. The bindings may be mutually recursive, in which case a least-fixpoint solution to the bindings shall be determined.

8.2.10 If expressions

Concrete syntax:

if_expression ::=
  if *if_condition:*expression **then**
    *true_alternative:*expression
    *elsif_alternatives:*optional_elsif_expression_list
    *false_alternative:*optional_else_expression
  end if

optional_elsif_expression_list ::=
  { elsif_expression }

elsif_expression ::=

elsif *elsif_condition:*expression **then**
*elsif_alternative:*expression

optional_else_expression ::=
[**else** expression]

Abstract syntax:

```
if_expression :: ...

optional_elsif_expression_list :: ...

elsif_expression :: ...

optional_else_expression :: ...
```

Simplification:

```
simplified_if_expression :: ...

simplify_if_expression ...
```

NOTE — The function `simplify_if_expression` yields an if expression in which the if condition and true expression are the simplified if condition and true expression, respectively of the given if expression. The false expression is formed from zero or more nested if expressions. The innermost false expression is either the simplified false expression of the given if expression, if present, or the bottom token otherwise. This is wrapped as the false expression in nested if expressions, starting from the last elsif expression of the given if expression and working outward to the first elsif expression. For each nested if expression, the if condition is the simplified form of the corresponding elsif condition and the true alternative is the simplified form of the corresponding elsif alternative.

Examples:

The Rosetta expression

```
if c1 then
  e1
end if
```

simplifies to the expression

```
if c1 then
  e1
else
  _|_
end if
```

The Rosetta expression

```
if c1 then
  e1
elsif c2 then
  e2
elsif c3 then
  e3
end if
```

simplifies to the expression

```
if c1 then
  e1
else
  if c2 then
    e2
  else
    if c3 then
      e3
    else
      _|_
    end if
  end if
end if
```

The Rosetta expression

```
if c1 then
  e1
elsif c2 then
  e2
elsif c3 then
  e3
else
  e4
end if
```

simplifies to the expression

```
if c1 then
  e1
else
  if c2 then
    e2
  else
    if c3 then
      e3
    else
      e4
    end if
  end if
end if
```

Name Expansion:

```
resolved_if_expression :: ...

expand_names_in_if_expression ...
```

NOTE — The function `expand_names_in_if_expression` yields a resolved if expression in which the if condition, true expression, and false expression correspond to the if condition, true expression, and false expression, respectively of the given simplified if expression but where any names are resolved references. The else if alternatives of the yielded value is an empty list.

Denotational semantics:

The value denoted by the condition expression in an if expression shall be of type `rosetta.lang.prelude.boolean`.

The value denoted by an if expression shall depend on the value denoted by the condition. If the value denoted by the condition is _|_, the value denoted by the if expression shall be _|_. If the value denoted by the condition is `true`, the value denoted by the if expression shall be the value denoted by the true alternative expression. If the value denoted by the condition is `false`, the value denoted by the if expression shall be the value denoted by the false alternative.

8.2.11 Case expressions

Concrete syntax:

case_expression ::=
  case *selector:*expression **is**
    *alternatives:*case_alternative_list
  end case

case_alternative_list ::=
  case_alternative { | case_alternative }

case_alternative ::=
  *tag:*expression -> *result:*expression

Abstract syntax:

```
case_expression :: ...

case_alternative_list :: ...

case_alternative :: ...
```

Simplification:

```
simplify_case_expression ...
```

NOTE — The function `simplify_case_expression` yields the simplification of an if expression in which there is a nested if expression for each case alternative, with the first case alternative corresponding to the outermost if expression and the last case alternative corresponding to the innermost if expression. For each if expression, the if condition is an application of the function rosetta.lang.prelude.type_member in which the first argument is the selector of the given case expression and the second argument is the tag expression of the corresponding alternative of the given case expression. The true alternative is the result expression of the corresponding alternative of the given case expression. There are no elsif expressions, and the false alternative is the next nested if expression. The false expression for the innermost if expression is absent.

Examples:

The Rosetta expression

```
case s is
  t1 -> e1
end case
```

simplifies to the expression

```
if rosetta.lang.prelude.universal_type_member ( s, t1 ) then
  e1
end if
```

The Rosetta expression

```
case s is
  t1 -> e1 |
  t2 -> e2 |
  t3 -> e3
end case
```

simplifies to the expression

```
if rosetta.lang.prelude.universal_type_member ( s, t1 ) then
  e1
else
  if rosetta.lang.prelude.universal_type_member ( s, t2 ) then
    e1
  else
    if rosetta.lang.prelude.universal_type_member ( s, t3 ) then
      e3
    end if
  end if
end if
```

NOTES:

1 — The values of the tag expressions need not be mutually exclusive.

2 — The effect of a default alternative, to be selected when no other alternative is selected, can be achieved by writing an alternative with the tag expression `universal` as the last alternative in a case expression. For example:

```
case char is
    {'.'}       -> ... // result for decimal point
  | {'0',..'9'} -> ... // result for digit
  | character   -> ... // result for other character
  | universal   -> ... // result for non-character
end case
```

8.2.12 Quantified expressions

Concrete syntax:

quantified_expression ::=
  *quantifier_name:*name (*parameters:*parameter_list | *encapsulated_expression:*expression)

Abstract syntax:

```
quantified_expression :: ...
```

Simplification:

```
simplify_quantified_expression ...
```

NOTE — The function `simplify_quantified_expression` yields a simplified nested function application. The parameter list of the quantified expression is expanded to obtain a list of individual parameter definitions. Starting with the last parameter definition in the list, the encapsulated expression of the quantified expression is wrapped as the encapsulated expression in an anonymous function, in which the anonymous function parameter definition is the last parameter definition of the list, and the return type expression is `rosetta.lang.prelude.universal`. The anonymous function then forms the argument of a function application in which the applied function is the quantifier name of the quantified expression. The resulting function application is further wrapped as the encapsulated expression in a similar anonymous function and function application, once for each of the parameter definitions in the list, if any.

Examples:

The Rosetta expression

```
q ( x1 :: T1; x2 :: T2 | e )
```

simplifies to the expression

```
q ( <* ( x1 :: T1 ) :: rosetta.lang.prelude.universal is
       q ( <* ( x2 :: T2 ) :: rosetta.lang.prelude.universal is
              e *> ) *> )
```

8.2.13 Template expressions

Concrete syntax:

template_expression ::=
  << *nonterminal_name:*label *template_elements:*template_element_list >>

template_element_list ::=
  template_element { template_element }

template_element ::=
  non_template_token
  | template_quoted_expression

template_quoted_expression ::=
  ` *quoted_expression:*expression `

A non template token shall be any lexical token except <<, >> or `.

A template expression shall be expanded into an abstract syntax tree during parsing of the concrete syntax of the Rosetta specification. The abstract syntax tree shall be formed by parsing the lexical tokens in the template element list. The goal nonterminal of the parse shall be the nonterminal of the Rosetta concrete syntax that corresponds to the abstract syntax type in `rosetta.lang.reflect.abstract_syntax` with the nonterminal name label of the template expression. The abstract syntax tree shall be of type `rosetta.lang.reflect.abstact_syntax.expression` and shall represent nested application of abstract syntax constructor functions that construct the parse of the sequence of template elements.

Where the parse encounters a literal in the sequence of template elements, the literal value shall be used as the abstract syntax subtree corresponding to the place of the literal in the parse.

Where the parse encounters a label in the sequence of template elements, the abstract syntax subtree corresponding to the place of the label in the parse shall be an abstract syntax tree for a function application. The applied function abstract syntax tree shall be an abstract syntax tree for a qualified name representing the item `rosetta.lang.reflect.labels.make_label`, and the argument abstract syntax tree shall be a list of one element, that element being a string literal containing the characters of the label.

Where the parse encounters a terminal token, other than a literal or a label, that represents itself in an abstract syntax tree, the value of type `rosetta.lang.reflect.lexical_elements.token` corresponding to the encountered token shall be used as the abstract syntax subtree corresponding to the place of the token in the parse.

Where the parse encounters a template quoted expression in the sequence of template elements, a type assertion operation shall take the place of an abstract syntax subtree constructor application. The type assertion operation constructor application shall have as its left operand argument the quoted expression and as its right operand argument a qualified name constructor application that shall construct a qualified name for the expected nonterminal type in `rosetta.lang.reflect.abstract_syntax`. The expected nonterminal type shall be the most general nonterminal that can satisfy the parse goal at the point of the template quoted expression in the template element sequence.

Examples:

The template expression

```
<<expression x + 2>>
```

is expanded into an abstract syntax tree that corresponds to the Rosetta expression

```
make_infix_binary_operation (
  rosetta.lang.reflect.make_label ( "x" ), plus_token, 2 )
```

The template expression

```
<<name rosetta.lang.prelude.empty_set>>
```

is expanded into an abstract syntax tree that corresponds to the Rosetta expression

```
make_qualified_name (
  make_qualified_name (
    make_qualified_name (
      rosetta.lang.reflect.make_label ( "rosetta" ),
      rosetta.lang.reflect.make_label ( "lang" ) ),
    rosetta.lang.reflect.make_label ( "prelude" ) ),
  rosetta.lang.reflect.make_label ( "empty_set" ) )
```

The template expression

```
<<optional_export_clause export all>>
```

is expanded into an abstract syntax tree that corresponds to the Rosetta expression

```
present ( all_token )
```

The template expression

```
<<function_type_formation <* `signature(n)` *> >>
```

is expanded into an abstract syntax tree that corresponds to the Rosetta expression

```
make_function_type_formation (
  ( signature(n) ) :: function_signature )
```

The template expression

```
<<function_application
  `applied_function(n)` ( `[head(arguments(n))]` ) ( `tail(arguments(n))` )>>
```

is expanded into an abstract syntax tree that corresponds to the Rosetta expression

```
make_funtion_application (
  make_function_application (
    ( applied_function(n) ) :: expression,
    ( [head(arguments(n))] ) :: argument_list ),
    ( tail(arguments(n)) ) :: argument_list )
```

NOTE — In this example, the quoted expression `[head(arguments(n))]` is expected to be on the nonterminal type `argument_list`, since that is the most general nonterminal type that can be substituted at that point in the parse. Hence, the value of `head(arguments(n))` is formed into a single-element list to meet the type requirement.

8.3 Constant expressions

Certain expressions are said to be constant expressions. The value denoted by such expressions shall not be dependent upon any state variable defined in a domain. A constant expression shall be an expression formed according to the following rules:

- a literal
- a name that denotes an item whose declaration includes a constant expression or the keyword **`constant`**
- a name that denotes a parameter of kind `rosetta.lang.static.design`
- ...

8.4 Static expressions

Certain constant expressions are said to be static expressions. The value denoted by such expressions shall be determined at the time of elaboration of the specification containing expressions.

9. Functions

This clause defines Rosetta function type formations, anonymous function values, function declarations and function application. The abstract syntax definitions in this clause are declared in the package `rosetta.lang.reflect.abstract_syntax`. The simplification definitions are declared in the package `rosetta.lang.reflect.simplification`.

9.1 Function signatures

A function signature shall define the parameter labels and types and the return type for a function type formation or an anonymous function.

Concrete syntax:

function_signature ::=
`(` *parameters:*parameter_list `)` `::` *return_type:*expression

parameter_list ::=
parameter_list_element { `;` parameter_list_element }

parameter_list_element ::=
multiple_parameter_definition
| parameter_definition

multiple_parameter_definition ::=
*parameter_labels:*label_list `::` *parameter_type:*expression

label_list ::=
label { `,` label }

parameter_definition ::=
*parameter_label:*label `::` *parameter_type:*expression

Abstract syntax:

```
function_signature :: ...

parameter_list :: ...

optional_parameter_list :: ...

parameter_list_element :: ...

multiple_parameter_definition :: ...

label_list :: ...

parameter_definition :: ...
```

Simplification:

```
simplified_function_signature :: ...
```



```
simplified_singleton_parameter_list :: ...
```

```
simplified_optional_parameter_list :: ...
```

```
simplified_parameter_definition :: ...
```

```
label_from_simplified_singleton_parameter_list ...
```

NOTE — The function `label_from_simplified_singleton_parameter_list` yields the label of the single parameter definition in the given parameter list.

```
labels_from_simplified_optional_parameter_list ...
```

NOTE — The function `labels_from_simplified_optional_parameter_list` yields a list of the labels of the parameter definitions, if any, in the given parameter list.

```
expand_parameter_list ...
```

NOTE — The function `expand_parameter_list` yields a parameter list that contains the concatenation of the results of expanding each element of the given parameter list. An empty list is yielded if the given list is empty.

```
expand_multiple_parameter_definition ...
```

NOTE — The function `expand_multiple_parameter_definition` yields a parameter list in which, for each label of the given multiple parameter definition, there is a parameter definition consisting of the label and the type expression of the given multiple parameter definition.

```
expand_parameter_definition ...
```

NOTE — The function `expand_parameter_definition` yields a singleton parameter list containing the given parameter definition.

```
simplify_parameter_definition ...
```

NOTE — The function `simplify_parameter_definition` yields a parameter definition in which the label is the label of the given parameter definition and the type expression is the simplification of the type expression of the given parameter definition

Examples:

The parameter list

```
p1 :: T1; p2, p3, p4 :: T2
```

expands to the parameter list

```
p1 :: T1; p2 :: T2; p3 :: T2; p4 :: T2
```

Name Expansion:

```
resolved_function_signature :: ...
```

```
expand_names_in_function_signature ...
```

NOTE — The function `expand_names_in_function_signature` yields a resolved function signature in which the parameter definition label is the parameter definition label of the given function signature, the parameter definition type expression corresponds to the parameter definition type expression of the given function signature where all names are resolved references, and the return type expression corresponds to the return type expression of the given function signature where all names are resolved references.

Denotational Semantics:

A parameter definition defines a label and a type for a parameter. The value denoted by the parameter type expression in a parameter definition shall be a type value, and is called the type of the parameter.

The value denoted by the return type expression of a function signature shall be a type value, and is called the return type of the function signature.

9.2 Function type formations

A function type formation shall denote a set of function values.

Concrete syntax:

function_type_formation ::=
  `<*` *signature:*function_signature `*>`

Abstract syntax:

```
function_type_formation :: ...
```

Simplification:

```
simplified_function_type_formation :: ...

simplify_function_type_formation ...
```

NOTE — The function `simplify_function_type_formation` yields a function type formation in which the return type is a nested function type formation, recursively, with the innermost return type being the simplification of the return type of the given function type formation. There is a nested function type formation corresponding to each parameter definition in the expanded parameter definition list derived from the parameter list of the given function type formation, with the first expanded parameter definition corresponding to the outermost type formation and the last parameter list correponding to the innermost type formation. For each of the nested type formations, there is a single parameter definition consisting of the simplification of the corresponding expanded parameter definition from the given function type formation.

Examples:

The Rosetta expression

```
<* ( p1 :: T1; p2 :: T2 ) :: R *>
```

simplifies to the expression

```
<* ( p1 :: T1 ) :: <* ( p2 :: T2 ) :: R *> *>
```

Name Expansion:

```
resolved_function_type_formation :: ...

expand_names_in_function_type_formation ...
```


NOTE — The function `expand_names_in_function_type_formation` yields a resolved function type formation in which the signature corresponds to the signature of the given function type formation where any names are resolved references.

Denotational semantics:

A function type formation shall denote the set of all function values whose domains are supertypes of the type of the parameter of the signature and whose ranges are subtypes of the return type of the signature.

NOTE — The domain of a function value that is a member of the type denoted by a type formation need not be a proper supertype of the type of the parameter of the signature. Similarly, the range of the function need not be a proper subtype of the return type of the signature.

9.3 Anonymous functions

An anonymous function expression shall denote a function value.

Concrete syntax:

anonymous_function ::=
`<*` *universally_quantified_variables:*optional_universally_quantified_variable_list
*signature:*function_signature **`is`** *encapsulated_expression:*expression `*>`

optional_universally_quantified_variable_list ::=
[`[` parameter_list `]`]

NOTE — The inner square brackets enclosing the parameter list in the rule for an optional univesally quantified variable list are delimiter tokens in the concrete syntax. The outer square brackets are EBNF meta-symbols.

Abstract syntax:

```
anonymous_function :: ...
```

Simplification:

```
simplified_anonymous_function :: ...

simplify_anonymous_function ...
```

NOTE — The function `simplify_anonymous_function` yields an anonymous function in which the universally quantified variable list contains the simplifications of the elements of the expanded universally quantified variable list of the given anonymous function. Further,

—If the expanded parameter list of the signature of the given anonymous function is singleton, then
- The parameter of the signature of the resulting anonymous function is the simplification of that element,
- The return type of the signature of the resulting anonymous function is the simplification of the result type of signature of the given anonymous function, and
- The encapsulated expression of the resulting anonymous function is the simplification of the encapsulated expression of the given anonymous function.

—Otherwise,
- The parameter of the signature of the resulting anonymous function is the simplification of the first element of the expanded parameter list of the signature of the given anonymous function,
- The return type of the signature of the resulting anonymous function is the simplification of a function type formation where the parameter list contains those elements of the expanded parameter list of the given anonymous function other than the first and the return type is the simplification of the return type of the signature of the given anonymous function, and
- The encapsulated expression of the resulting function is the simplification of an anonymous function which has no universally quantified variables, a parameter list that contains those elements of the expanded parameter list of the

given anonymous function other than the first, a return type that is the simplification of the return type of the signature of the given anonymous function, and an encapsulated expression that is the simplification of the encapsulated expression of the given anonymous function.

Name Expansion:

```
resolved_anonymous_function :: ...
```

```
expand_names_in_anonymous_function ...
```

NOTE — The function `expand_names_in_anonymous_function` yields a resolved anonymous function in which the universally quantified variable list corresponds to the universally quantified variable list of the given anonymous function where any names are resolved references, the signature corresponds to the signature of the given anonymous function where any names are resolved references, and the encapsulated expression corresponds to the encapsulated expression of the given anonymous function where any names are resolved references.

```
expand_names_in_optional parameter list ...
```

NOTE — The function `expand_names_in_optional_parameter_list` yields a resolved optional parameter list in which the parameters correspond to the parameters of the given optional parameter list where any names in the type expressions of the parameters are resolved references.

Denotational semantics:

An anonymous function shall denote a function value.

If the universally quantified variable list is absent, the function value shall be of a monomorphic type. The domain of the function value shall be the type of the parameter of the signature. For each value in the domain, the image shall be the value denoted by the encapsulated expression interpreted in a context formed by adding a binding for the parameter to the head of the context in which the anonymous function is interpreted. The binding for the parameter shall consist of an item whose label is the parameter label and whose value is the domain value. The image value shall be a member of the return type of the signature of the anonymous function. The range of the function value shall be the set of image values for all values in the domain of the function value.

If the universally quantified variable list is present, the function value shall be of a polymorphic or dependent type. The function value shall be determined by considering each possible combination of binding the labels of the universally quantified variable list to values of their corresponding types. The domain of the function value shall be the union of all of the types denoted by the type expression of the signature interpreted in each of the contexts given by the combinations of bindings for the universally quantified variables. For each value in the domain, the image shall be the value denoted by the encapsulated expression interpreted in a context formed by adding a binding for the parameter and the corresponding bindings for the universally quantified variables to the head of the context in which the anonymous function is interpreted. The binding for the parameter shall consist of an item whose label is the parameter label and whose value is the domain value. The image value shall be a member of the type denoted by the return type expression of the signature of the anonymous function, interpreted in the same context as the encapsulated expression. The range of the function value shall be the set of image values for all values in the domain of the function value.

NOTE — Since the undefined value _|_ is a member of every type, it is a member of the domain of every function value. A function need not be strict; that is, the image of _|_ need not be _|_.

9.4 Function declarations

Concrete syntax:

function_declaration ::=
  *function_label:*label

*universally_quantified_variables:*optional_universally_quantified_variable_list
*signature:*function_signature
*definition:*optional_definition_clause *property:*optional_property_clause ;

Abstract syntax:

```
function_declaration :: ...
```

Simplification:

```
simplify_function_declaration ...
```

NOTE — The function `simplify_function_declaration` yields the simplification of a variable declaration where

—The label is that of the given function declaration;
—The declared type is either
- A function type formation with the same signature as the given function declaration, if the list of universally quantified variables of the given function declaration is empty, otherwise
- An anonymous function with a parameter list that is the same as the universally quantified variable list of the given function declaration, **type** as the return type, and a function type formation with the same signature as the given function declaration as the encapsulated expression;

—The definition is one of
- Absent, if the definition of the given function declaration is absent,
- The function definition **is constant**, if the definition of the given function declaration is constant, otherwise
- An anonymous function where the universally quantified variable list is the same as that of the given function declaration, the signature is the same as that of the given function declaration, and the encapsulated expression is the same as the expression of the definition of the given function declaration;

—The property clause is either
- Absent, if the property clause of the given function declaration is absent, otherwise
- A quantified expression where the quantifier is `rosetta.lang.prelude.forall`, the parameter list is the concatenation of the universally quantified variable list and parameter list of the given function declaration, and the encapsulated expression is the expression of the property clause of the given function declaration.

Examples:

The Rosetta function declaration

```
f ( x1 :: T1; x2 :: T2 ) :: Tr;
```

simplifies to the simplification of the declaration

```
f :: <* ( x1 :: T1; x2 :: T2 ) :: Tr *>;
```

The Rosetta function declaration

```
f [ v1 :: Tv1; v2 :: Tv2 ] ( x1 :: T1; x2 :: T2 ) :: Tr;
```

simplifies to the simplification of the declaration

```
f :: <* ( v1 :: Tv1; v2 :: Tv2 ) :: type is
        <* ( x1 :: T1; x2 :: T2 ) :: Tr *> *>;
```

The Rosetta function declaration

```
f ( x1 :: T1; x2 :: T2 ) :: Tr is E;
```

simplifies to the simplification of the declaration

```
f :: <* ( x1 :: T1; x2 :: T2 ) :: Tr *> is
     <* ( x1 :: T1; x2 :: T2 ) :: Tr is E *>;
```

The Rosetta function declaration

```
f [ v1 :: Tv1; v2 :: Tv2 ] ( x1 :: T1; x2 :: T2 ) :: Tr is E;
```

simplifies to the simplification of the declaration

```
f :: <* ( v1 :: Tv1; v2 :: Tv2 ) :: type is
        <* ( x1 :: T1; x2 :: T2 ) :: Tr *> *> is
     <* [ v1 :: Tv1; v2 :: Tv2 ] ( x1 :: T1; x2 :: T2 ) :: Tr is E *>;
```

The Rosetta function declaration

```
f ( x1 :: T1; x2 :: T2 ) :: Tr is constant;
```

simplifies to the simplification of the declaration

```
f :: <* ( x1 :: T1; x2 :: T2 ) :: Tr *> is constant;
```

The Rosetta function declaration

```
f [ v1 :: Tv1; v2 :: Tv2 ] ( x1 :: T1; x2 :: T2 ) :: Tr is constant;
```

simplifies to the simplification of the declaration

```
f :: <* ( v1 :: Tv1; v2 :: Tv2 ) :: type is
        <* ( x1 :: T1; x2 :: T2 ) :: Tr *> *> is constant;
```

The Rosetta function declaration

```
f ( x1 :: T1; x2 :: T2 ) :: Tr where P ( x1, x2 );
```

simplifies to the simplification of the declaration

```
f :: <* ( x1 :: T1; x2 :: T2 ) :: Tr *>
  where rosetta.lang.prelude.forall ( x1 :: T1; x2 :: T2 | P ( x1, x2 ) );
```

The Rosetta function declaration

```
f [ v1 :: Tv1; v2 :: Tv2 ] ( x1 :: T1; x2 :: T2 ) :: Tr
  where P ( v1, v2, x1, x2 );
```

simplifies to the simplification of the declaration

```
f :: <* ( v1 :: Tv1; v2 :: Tv2 ) :: type is
        <* ( x1 :: T1; x2 :: T2 ) :: Tr *> *>
  where rosetta.lang.prelude.forall ( v1 :: Tv1; v2 :: Tv2; x1 :: T1; x2 :: T2 |
                  P ( v1, v2, x1, x2 ) );
```

The Rosetta function declaration

```
f ( x1 :: T1; x2 :: T2 ) :: Tr is E where P ( x1, x2 );
```


simplifies to the simplification of the declaration

```
f :: <* ( x1 :: T1; x2 :: T2 ) :: Tr *> is
     <* ( x1 :: T1; x2 :: T2 ) :: Tr is E *>
  where rosetta.lang.prelude.forall ( x1 :: T1; x2 :: T2 | P ( x1, x2 ) );
```

The Rosetta function declaration

```
f [ v1 :: Tv1; v2 :: Tv2 ] ( x1 :: T1; x2 :: T2 ) :: Tr is E
  where P ( v1, v2, x1, x2 );
```

simplifies to the simplification of the declaration

```
f :: <* ( v1 :: Tv1; v2 :: Tv2 ) :: type is
        <* ( x1 :: T1; x2 :: T2 ) :: Tr *> *> is
     <* [ v1 :: Tv1; v2 :: Tv2 ] ( x1 :: T1; x2 :: T2 ) :: Tr is E *>
  where rosetta.lang.prelude.forall ( v1 :: Tv1; v2 :: Tv2; x1 :: T1; x2 :: T2 |
                                       P ( v1, v2, x1, x2 ) );
```

The Rosetta function declaration

```
f ( x1 :: T1; x2 :: T2 ) :: Tr is constant where P ( x1, x2 );
```

simplifies to the simplification of the declaration

```
f :: <* ( x1 :: T1; x2 :: T2 ) :: Tr *> is constant
  where rosetta.lang.prelude.forall ( x1 :: T1; x2 :: T2 | P ( x1, x2 ) );
```

The Rosetta function declaration

```
f [ v1 :: Tv1; v2 :: Tv2 ] ( x1 :: T1; x2 :: T2 ) :: Tr is constant
  where P ( v1, v2, x1, x2 );
```

simplifies to the simplification of the declaration

```
f :: <* ( v1 :: Tv1; v2 :: Tv2 ) :: type is
        <* ( x1 :: T1; x2 :: T2 ) :: Tr *> *> is constant
  where rosetta.lang.prelude.forall ( v1 :: Tv1; v2 :: Tv2; x1 :: T1; x2 :: T2 |
                                       P ( v1, v2, x1, x2 ) );
```

9.5 Function application

Concrete syntax:

function_application ::=
  *applied_function:*precedence_12_expression (*arguments:*expression_list)

expression_list ::=
  expression { , expression }

Abstract syntax:

```
function_application :: ...
```



```
expression_list :: ...
```

Simplification:

```
simplified_function_application :: ...

simplify_function_application ...
```

NOTE — If the argument list of the given function application is a singleton list, then the result of the `simplify_function_application` function is a function application where the applied function is the simplification of the applied function of the given function application and the argument is the simplification of the argument of the given function application. Otherwise, the result is the simplification of a Curried function application where

—The applied function is itself a function application in which the applied function is the applied function of the given function application and the single argument is the first argument of the given function application,
—The arguments are those of the given function application excluding the first element.

Examples:

The Rosetta function application `f ( x )` elaborates to a function application of the same form in the kernel language.

The Rosetta function application

```
f ( e1, e2, e3 )
```

simplifies to the function application

```
f ( e1 ) ( e2 ) ( e3 )
```

Name Expansion:

```
resolved_function_application :: ...

expand_names_in_function_application ...
```

NOTE — The function `expand_names_in_function_application` yields a resolved function application in which the applied function corresponds to the applied function of the given function application where any names are resolved references, and the argument list corresponds to the argument list of the given function application where any names are resolved references.

Denotational semantics:

The applied function expression of a function application shall denote a function value, called the applied function value. The value denoted by the argument expression shall be a member of the domain type of the applied function value. The value denoted by the function application, called the result value, shall be the image of the value denoted by the argument expression mapped by the applied function value.

10. Declarations

This clause defines declarations that may be included in facets, packages, components and domains. The abstract syntax definitions in this clause are declared in the package `rosetta.lang.reflect.abstract_syntax`.

10.1 Declaration lists

Concrete syntax:

optional_declaration_list ::=
  { declaration }

declaration ::=
  variable_declaration
  | enumeration_type_declaration
  | constructed_type_declaration
  | function_declaration
  | facet_declaration
  | package_declaration
  | component_declaration
  | domain_declaration
  | interaction_declaration
  | use_clause

Abstract syntax:

```
optional_declaration_list :: ...

declaration :: ...
```

A declaration list element shall declare one or more items in the declarative region containing the optional declaration list. The labels of the all of the declared items, both explicitly declared and implicitly declared, in a declarative region shall be distinct.

Static semantics:

A declaration list element shall declare one or more items in the declarative region containing the optional declaration list. The labels of the all of the declared items, both explicitly declared and implicitly declared, in a declarative region shall be distinct.

Simplification:

```
simplified_declaration :: ...

simplified_optional_declaration_list :: ...

labels_from_simplified_optional_declaration_list ...
```

NOTE — The function `labels_from_simplified_optional_declaration_list` yields a list of the labels implicitly and explicitly declared by each of the declarations, if any, in the given declaration list.

```
labels_from_simplified_declaration ...
```


NOTE — The function `labels_from_simplified_declaration` yields a list of labels as follows:

—In the case of a variable declaration and property simplification, the list of labels resulting from application of the function `labels_from_variable_declaration_and_property_simplification` to the given simplified declaration.
—In the case of a constructed type declaration, the list of labels resulting from application of the function `labels_from_constructed_type_declaration` to the given simplified declaration.
—In the case of a domain declaration, the domain label of the domain declaration.
—In the case of a use clause, the empty list.

```
simplify_optional_declaration_list ...
```

NOTE — The function `simplify_optional_declaration_list` yields a list of simplifications that result from

—Applying `simplify_variable_declaration` to any variable declarations in the given declaration list,
—Applying `simplify_enumeration_declaration` to any enumeration type declarations in the given declaration list,
—Applying `simplify_constructed_type_declaration` to any constructed type declarations in the given declaration list,
—Applying `simplify_facet_declaration_list` to the list of any facet declarations (be they complete, interface, or body declarations) in the given declaration list,
—Applying `simplify_package_declaration_list` to the list of any package declarations (be they complete, interface, or body declarations) in the given declaration list,
—Applying `simplify_compenent_declaration_list` to the list of any component declarations (be they complete, interface, or body declarations) in the given declaration list,
—Applying `simplify_domain_declaration` to any domain declarations in the given declaration list,
—Applying `simplify_interaction_declaration` to any interaction declarations in the given declaration list, and
—Any use clauses in the given declaration list.

10.2 Variable declarations

Concrete syntax:

variable_declaration ::=
*labels:*label_list `::` *declared_type:*expression
*definition:*optional_definition_clause *property:*optional_property_clause `;`

optional_definition_clause ::=
[**`is`** definition_clause]

definition_clause ::=
expression | **`constant`**

optional_property_clause ::=
[**`where`** expression]

Abstract syntax:

```
variable_declaration :: ...

optional_definition_clause :: ...

definition_clause :: ...

optional_property_clause :: ...
```

Static semantics:

The declared type expression shall be a constant expression. If the definition is present and take the form of an expression, the expression shall be a constant expression.

NOTE — An item declared with the keyword **`constant`** may have its value specified by the interpretations of terms in the specification, or it may remain unspecified.

Simplification:

```
variable_declaration_and_property_simplification :: ...

variable_declaration_simplification :: ...

variable_declaration_simplification_list :: ...

labels_from_variable_declaration_and_property_simplification ...
```

NOTE — The function `labels_from_variable_declaration_and_property_simplification` yields a list of labels of the simplified variable declarations in the given simplification.

```
simplify_variable_declaration ...
```

NOTE — The function `simplify_variable_declaration` yields a simplification containing of a list of simplified variable declarations. For each label of the given variable declaration, there is a variable declaration simplification in which:

—The label is the label from the given variable declaration.
— The declared type is the simplification of the declared type of the given variable declaration.
—If the given variable declaration has a definition that is an expression, the definition term is present and is the simplification of a simple term in which the term expression is a function application. The applied function is the name `rosetta.lang.prelude.universal_equals`, the first argument is the label, and the second argument is the definition expression of the given variable declaration.
—If the given variable declaration has a definition that is **constant**, the definition term is present and is the simplification of a simple term in which the term expression is a tick operation. The operand of the tick operation is the label, and the single tick operator has the tick label `is_constant` with no tick arguments.
—If the given variable declaration has no definition, the definition term is absent.

If the given variable declaration has a property clause, the property term is present in the result and is the simplification of a simple term where the term expression is the expression of the property clause. Otherwise, the property term is absent in the result.

Examples:

The Rosetta variable declaration

```
v :: T;
```

simplifies to a single variable declaration simplification with no definition term and no property term.

The Rosetta variable declaration

```
v1, v2, v3 :: T;
```

simplifies to three variable declaration simplifications corresponding to the declarations

```
v1 :: T;
v2 :: T;
v3 :: T;
```

The Rosetta variable declaration


```
v :: T is E;
```

simplifies to a variable declaration simplification corresponding to the declaration

```
v :: T;
```

combined with the definition term that is the simplification of the term

```
v = E;
```

The Rosetta variable declaration

```
v :: T is constant;
```

simplifies to a variable declaration simplification corresponding to the declaration

```
v :: T;
```

combined with the definition term that is the simplification of the term

```
T'is_constant;
```

The Rosetta variable declaration

```
v :: T where P ( v );
```

simplifies to a variable declaration simplification corresponding to the declaration

```
v :: T;
```

and with no definition term. The simplification includes the property term that is the simplification of the term

```
P ( v );
```

The Rosetta variable declaration

```
v1, v2, v3 :: T is E where P ( v1, v2, v3 );
```

simplifies to three variable declarations simplification corresponding to the declarations

```
v1 :: T;
v2 :: T;
v3 :: T;
```

combined with the definition terms that are, respectively, the simplifications of the terms

```
v1 = E;
v2 = E;
v3 = E;
```

The simplification includes the property term that is the simplification of the term

```
P ( v1, v2, v3 );
```

The Rosetta variable declaration

```
v1, v2, v3 :: T is constant where P ( v1, v2, v3 );
```

simplifies to three variable declarations simplification corresponding to the declarations

```
v1 :: T;
v2 :: T;
v3 :: T;
```

combined with the definition terms that are, respectively, the simplifications of the terms

```
v1'is_constant;
v2'is_constant;
v3'is_constant;
```

The simplification includes the property term that is the simplification of the term

```
P ( v1, v2, v3 );
```

Name Expansion:

```
resolved_variable_declaration_and_property_simplification :: ...

resolved_variable_declaration_simplification :: ...

resolved_variable_declaration_simplification_list :: ...

expand_names_in_variable_declaration_and_property_simplification ...
```

NOTE — The function `expand_names_in_variable_declaration_and_property_simplification` yields a resolved simplification containing of a list of resolved variable declarations. For each variable declaration of the given simplification, there is a corresponding variable declaration in which any names are resolved references. If the given variable declaration has a property clause, the property term is present in the result and corresponds to the given property clause where any names are resolved references. Otherwise, the property term is absent in the result.

```
expand_names_in_variable_declaration ...
```

NOTE — The function `expand_names_in_variable_declaration` yields a resolved variable declaration in which the variable label is that of the given declaration, the declared type corresponds to the declared type of the given declaration where any names are resolved references, and the definition expression corresponds to the definition expression of the given declaration where any names are resolved references.

Denotational semantics:

A variable declaration shall declare an item whose label is the label in the variable declaration simplification and whose type is the value denoted by the declared type.

10.3 Enumeration type declarations

Concrete syntax:

enumeration_type_declaration ::=
  *type_label:*label `::` *subtype:*subtype_or_universal_type_formation **is**
    *enumeration_formation:*enumeration_type_formation `;`

subtype_or_universal_type_formation ::=
subtype_formation
| universal_type_formation

enumeration_type_formation ::=
enumeration (*values:*label_list)

Abstract syntax:

```
enumeration_type_declaration :: ...

subtype_or_universal_type_formation :: ...

enumeration_type_formation :: ...
```

Simplification:

```
simplify_enumeration_type_declaration ...
```

NOTE — The function `simplify_enumeration_type_declaration` yields the simpification of a constructed type declaration where the type label is that of the given enumeration type declaration, there are no parameters, the subtype is that of the given enumeration type declaration, and the alternatives list is such that, for each value of the given enumeration type declaration, there is a constructor with the same constructor label, no observers, and no recognizer label.

Example:

The Rosetta declaration

```
T :: type is enumeration ( a, b, c );
```

simplifies to the declaration

```
T :: type is data a | b | c;
```

10.4 Constructed type declarations

Concrete syntax:

constructed_type_declaration ::=
*type_label:*label *parameters:*optional_parameter_list
:: *subtype:*subtype_or_universal_type_formation **is data**
*alternatives:*constructor_definition_list ;

optional_parameter_list ::=
[(parameter_list)]

constructor_definition_list ::=
constructor_definition { | constructor_definition }

constructor_definition ::=
*constructor_label:*label *observers:*optional_parameter_list *recognizer_label:*optional_recognizer

optional_recognizer ::=
[:: label]

Abstract syntax:

```
constructed_type_declaration :: ...

constructor_definition_list :: ...

constructor_definition :: ...

optional_recognizer :: ...
```

Simplification:

```
simplified_constructed_type_declaration :: ...

simplified_constructor_definition :: ...

labels_from_simplified_constructed_type_declaration ...
```

NOTE — The function `labels_from_simplified_constructed_type_declaration` yields a list of the labels explicitly and implicitly declared by the given constructed type declaration. The type label is the only explicitly declared label. The implicitly declare labels are, for each alternative, the constructor label, the label of the recognizer, if present, and the label of each observer, if any.

```
simplify_constructed_type_declaration ...
```

NOTE — The function `simplify_constructed_type_declaration` yields a constructed type declaration where the type label is that of the given constructed type declaration, the parameter list contains simplifications of the parameter definitions that result from applying `expand_parameter_list` to the parameter list of the given constructed type declaration, the subtype expression that is the simplification of the subtype expression of the given constructed type declaration, and the constructor alternatives are the simplifications of the constructor alternatives of the given constructed type declaration.

```
simplify_constructor_definition ...
```

NOTE — The function `simplify_constructor_definition` yields a constructor definition where the constructor label is that of the given constructor, the observer list contains the simplifications of the observer definitions that result from applying `expand_parameter_list` to the observer list of the given constructor definitiion, and the recongnizer label is that of the given constructor definition.

Example:

The Rosetta declaration

```
Tc ( x1, x2 :: Tx ) :: subtype ( Ts ) is data
   c1 ( p1, p2 :: Tp ) :: r1
 | c2 ( p3 :: Tp ) :: r2;
```

simplifies to the declaration

```
Tc ( x1 :: Tx; x2 :: Tx ) :: subtype ( Ts ) is data
   c1( p1 :: Tp; p2 :: Tp ) :: r1
 | c2( p3 :: Tp ) :: r2;
```

Name Expansion:

```
resolved_constructed_type_declaration :: ...
```



```
resolved_constructor_definition :: ...

expand_names_in_constructed_type_declaration ...
```

NOTE — The function `expand_names_in_constructed_type_declaration` yields a resolved constructed type declaration where the type label is that of the given constructed type declaration, the parameter list corresponds to the result from applying `expand_names_in_optional_parameter_list` to the parameter list of the given constructed type declaration, the subtype expression corresponds to the subtype expression of the subtype expression of the given constructed type declaration, and the constructor alternatives are the constructor alternatives that result from applying `expand_names_in_constructor_definition` to the constructor definitions of the given constructed type declaration.

```
expand_names_in_constructor_definition ...
```

NOTE — The function `expand_names_in_constructor_definition` yields a constructor definition where the constructor label is that of the given constructor, the observer list corresponds to the observer list of the given constructor definitiion where any names in the type expressions are resolved references, and the recongnizer label is that of the given constructor definition.

Denotational semantics:

A constructed type declaration shall declare in the declarative region containing the constructed type declaration an item, called the *constructed type item*. The label of the constructed type item shall be the type label of the constructed type declaration. The type of the constructed type item shall be the value denoted by the subtype of the constructed type declaration.

If the constructed type declaration omits the optional constructed type parameters, the constructed type item shall have a monomorphic type as its value. The value shall be the disjoint union of the types of the constructor items corresponding to constructor definitions with no observers, if any, and the ranges of the values of the constructor items corresponding to constructor definitions with observers, if any.

If the constructed type declaration includes a parameter list, the constructed type item shall have a polymorphic or dependent type as its value. The value shall be a function whose parameters are the parameters of the constructed type declaration and whose return type is the value denoted by the universal type formation. The result yielded by the function shall be the disjoint union of the types of the constructor items corresponding to constructor definitions no observers, if any, and the ranges of the values of the constructor items corresponding to constructor definitions with observers, if any.

A constructed type declaration shall also implicitly declare items corresponding to the constructor definitions. These implicit declarations shall be included in the declarative region containing the constructed type declaration.

10.4.1 Constructor definitions with no observers

Corresponding to a constructor definition in which the observers parameter list is empty, the facet equivalent containing the constructed type declaration shall contain an implicitly declared item, called the *constructor item* corresponding to the constructor definition. The label of the constructor item shall be the constructor label of the constructor definition. The type of the constructor item shall be the domain-theoretic *Unit* domain, namely, the domain containing just one value in addition to $\bot$. The value of the constructor item shall be the non-$\bot$ value of the *Unit* domain tagged as being a member of the alternative of the disjoint union corresponding to the constructor definition.

If the constructor definition includes a recognizer label, the facet equivalent containing the constructed type declaration shall contain an implictly declared item, called the *recognizer item* corresponding to the constructor definition. The label of the recognizer item shall be the recognizer label of the constructor definition. The type of the recognizer item shall be the function type consisting of function values whose domain is the value of the constructed type item and whose return type is `rosetta.lang.prelude.boolean`. The value of the recognizer item shall be the function that maps the value of the constructor item to `true` and other values in its domain to `false`.

10.4.2 Constructor definitions with observers

A constructor definition in which the observers parameter list is not empty shall specify the type that is a domain-theoretic product being the range of the constructor item value specified in this clause.

Corresponding to the constructor definition, the facet equivalent containing the constructed type declaration shall contain an implicitly declared item, called the *constructor item* corresponding to the constructor definition. The label of the constructor item shall be the constructor label of the constructor definition.

If the constructed type declaration omits the optional constructed type parameters, the constructor item shall have a function of monomorphic type as its value. The type of the constructor item shall be the function type whose parameter list is the observers parameter list of the constructor definition and whose return type is the value of the constructed type item.

Example:

Given the constructed type declaration

```
Tc :: subtype(Ts) is data
  C ( O1 :: T1; O2 :: T2 ) :: R;
```

the type of the constructor item corresponding to `C` is

```
<* ( O1 : T1;  O2 :: T2 ) :: Tc *>
```

If the constructed type declaration includes a parameter list, the constructor item shall have a function of polymorphic or dependent type as its value. The type of the constructor item shall be a function whose parameters are the parameters of the constructed type declaration and whose return type is the value denoted by the universal type formation. The result yielded by the function shall be the function type whose parameter list is the observers parameter list of the constructor definition and whose return type is the value of the constructed type item.

Example:

Given the constructed type declaration

```
Tc ( x :: Tp ) :: subtype(Ts) is data
  C ( O1 :: T1; O2 :: T2 ) :: R;
```

the type of the constructor item corresponding to `C` is

```
<* ( x :: Tp ) :: type is <* ( O1 : T1;  O2 :: T2 ) :: Tc *> *>
```

The value of the constructor item shall be a function that is a member of the type of the constructor item and that yields a tuple formed from the function arguments in the order of occurrence of the arguments. The tuple shall be tagged as being a member of the alternative of the disjoint union corresponding to the constructor definition.

Corresponding to each parameter definition in the observers parameter list of the constructor definition, the facet equivalent containing the constructed type declaration shall contain an implictly declared item, called the *observer item* corresponding to the parameter definition. The label of the observer item shall be the parameter label of the observer parameter definition. The type of the observer item shall be the function type consisting of function values whose domain is the range of the value of the constructor item and whose return type is the value denoted by the parameter type of the observer parameter definition. The value of the observer item shall be the function item that projects a tuple component from the argument. The component projected is the component in the position corresponding to the position of the observer parameter definition in the observers parameter list of the constructor definition.

If the constructor definition includes a recognizer label, the facet equivalent containing the constructed type declaration shall contain an implictly declared item, called the *recognizer item* corresponding to the constructor definition. The label of the recognizer item shall be the recognizer label of the constructor definition. The type of the recognizer item shall be the function type consisting of function values whose domain is the value of the constructed type item and whose return type is `rosetta.lang.prelude.boolean`. The value of the recognizer item shall be the function that maps values in the range of the value of the constructor item to `true` and other values in its domain to `false`.

11. Terms

This clause defines the syntax and semantics of terms. The abstract syntax definitions in this clause are declared in the package `rosetta.lang.reflect.abstract_syntax`. The simplification definitions are declared in the package `rosetta.lang.reflect.simplification`.

Concrete syntax:

optional_term_list ::=
  { term }

term_list ::=
  term { term }

term ::=
  simple_term
  | let_term

simple_term ::=
  *term_label:*optional_label *term_expression:*expression *justification:*optional_justification;

optional_label ::=
  [label **:**]

optional_justification ::=
  [**justification** expression]

let_term ::=
  let
    *binding_list:*let_binding_list
  in
    *terms:*term_list
  end let ;

Abstract syntax:

```
optional_term_list :: ...

term_list :: ...

term :: ...

simple_term :: ...

optional_label :: ...

let_term :: ...
```

Static semantics:

A simple term declares the term label, if present, and specifies a term expression.

The term expression in a simple term shall be a facet expression or an expression of type `rosetta.lang.prel-`

`ude.boolean`. If the term expression in a simple term is a facet expression, the term label shall be present in the simple term. If the justification exists in a simple term, the expression in the justification shall be an expression of type `rosetta.lang.prelude.string`.

NOTE — Justifications are used to record the reason that a term is believed to be true. The result of the string expression is outside the semantics of the language, and may be used by a tool or may be used as a documentation device by the user.

Simplification:

```
simplified_optional_term_list :: ...

simplified_term_list :: ...

simplified_term :: ...

simplified_optional_justification :: ...

labels_from_simplified_term_list ...
```

NOTE — The function `labels_from_simplified_term_list` yields a list of labels of simple terms in the given term list. If a simple term is labelled, its label is included in the result. If a simple term is not labelled, no label corresponding to that term is included in the result.

```
expand_optional_term_list ...
```

NOTE — The function `expand_optional_term_list` yields the concatenation of the lists that result from expansion of each element in the given term list. For a simple term the expansion is a singleton list containing the simple term. For a let term, the expansion is a list of simple terms in which, for each term in the term list of the given let term, there is a simple term in which the term label is that of the given let term and the term expression is a let expression with the same binding list as the given let term term and the same encapsulated expression as the given let term term.

```
simplify_simple_term ...
```

NOTE — The function `simplify_simple_term` yields a simple term in which the term label is that of the given simple term, the term expression is the simplification of the term expression of the given simple term, and the justification is the simplification of the justification expression of the given simple term, if the justification expression is present, or absent otherwise.

```
expand_and_simplify_optional_term_list ...
```

NOTE — The function `expand_and_simplify_optional_term_list` yields a list of simplified terms where the elements are the simplifications of the elements of the expansion of the given term list.

Examples:

A simple term expands to a term of the same form.

The let term

```
let
  v :: T be E_v
in
  L1 : E1;
  L2 : E2;
end let;
```

expands and simplifies to the same sequence of simple terms as the simple terms


```
L1 : let
       v :: T be E_v
     in
       E1
     end let;
L2 : let
       v :: T be E_v
     in
       E2
     end let;
```

The let term

```
let
  v1 :: T1 be E_v1
in
  L1 : E1;
  let
    v2 :: T2 be E_v2
  in
    L2 : E2;
    L3 : E3;
  end let;
end let;
```

expands and simplifies to the same sequence of simple terms as the simple terms

```
L1 : let
       v1 :: T1 be E_v1
     in
       E1
     end let;
L2 : let
       v1 :: T1 be E_v1
     in
       let
         v2 :: T2 be E_v2
       in
         E2
       end let
     end let;
L3 : let
       v1 :: T1 be E_v1
     in
       let
         v2 :: T2 be E_v2
       in
         E3
       end let
     end let;
```

Name Expansion:

```
  resolved_optional_term_list :: ...
```



```
resolved_term_list :: ...

resolved_term :: ...

resolved_optional_justification :: ...

expand_names_in_optional_term_list ...
```

NOTE — The function `expand_names_in_optional_term_list` yields the list of resolved terms that result from application of the function `simplify_simple_term` to each member of the given list.

```
expand_names_in_term ...
```

NOTE — The function `expand_names_in_term` yields a resolved term in which the label is that of the given term, the expression corresponds to the expression of the given term where any names are resolved references, and the justification corresponds to the justification of the given term where any names are resolved references.

Denotational semantics:

To do: complete cross refs...

If the term expression is of type `rosetta.lang.prelude.boolean`, the expression shall be interpreted as a constraint that applies to the items referenced by the expression (see ??—clause defining denotation of a facet). If the term expression is a facet expression, the facet value denoted by the term expression shall be included in the facet denoted by the immediately enclosing facet (see ??—clause defining facet inclusion).

The value denoted by the justification expression, if present, shall not affect the interpretation of a Rosetta specification.

12. Facets

This clause defines facets. The abstract syntax definitions in this clause are declared in the package `rosetta.lang.reflect.abstract_syntax`.

12.1 Facet signatures

Concrete syntax:

facet_signature ::=
  *parameters:*optional_facet_parameter_list `::` *facet_domain:*expression

optional_facet_parameter_list ::=
  [`(` facet_parameter { `;` facet_parameter } `)`]

facet_parameter ::=
  multiple_facet_parameter_definition
  | facet_parameter_definition

multiple_facet_parameter_definition ::=
  *parameter_labels:*label_list `::` *parameter_kind:*optional_kind *parameter_type:*expression

facet_parameter_definition ::=
  *parameter_label:*label `::` *parameter_kind:*optional_kind *parameter_type:*expression

optional_kind ::=
  [kind]

kind ::=
  label

Abstract syntax:

```
facet_signature :: ...

optional_facet_parameter_list :: ...

facet_parameter :: ...

multiple_facet_parameter_definition :: ...

facet_parameter_definition :: ...

optional_kind :: ...

kind :: ...
```

Static semantics:

The domain expression of a facet signature shall be a static domain expression. The domain value denoted by the domain expression shall be called the *domain of the facet signature*.

NOTE — The domain expression is usually the name of a domain declared using a domain declaration.

If a facet parameter definition includes a parameter kind label, that label shall be the label of a function declared in the domain of the facet signature.

Simplification:

```
simplified_facet_signature :: ...
```

```
simplified_optional_facet_parameter_list :: ..
```

```
simplified_facet_parameter_definition :: ...
```

```
simplified_optional_kind :: ...
```

```
facet_parameter_list_simplification :: ...
```

```
labels_from_simplified_facet_parameter_list ...
```

NOTE — The function `labels_from_simplified_facet_parameter_list` yields a list of labels of parameters, if any, in the given facet parameter list.

```
expand_facet_parameter_list ...
```

NOTE — The function `expand_facet_parameter_list` yields a facet parameter list which contains the concatenation of the results of expanding each element of the given facet parameter list, if any.

```
expand_multiple_facet_parameter_definition ...
```

NOTE — The function `expand_multiple_facet_parameter_definition` yields a facet parameter list in which, for each label of the given multiple facet parameter definition, there is a facet parameter definition consisting of the label, the kind of the given multiple facet parameter definition, and the type expression of the given multiple facet parameter definition.

```
expand_facet_parameter_definition ...
```

NOTE — The function `expand_facet_parameter_definition` yields a singleton facet parameter list containing the given facet parameter definition.

```
simplify_facet_parameter_definition ...
```

NOTE — The function `simplify_facet_parameter_definition` yields a facet parameter list simplification. The parameter list part of the simplification is a singleton list containing a simplified parameter definition in which the label is that of the given facet parameter definition, and the type expression is the elaboration of the type expression of the given facet parameter definition. If the given facet parameter definition has a kind label, the term list part of the simplification is a simpified term in which the term expression is a tick_operation where the operand is the label of the given facet parameter definition, the label of the single tick operator is the kind label of the given facet parameter definition, and there are no tick arguments. Otherwise, the the term list part of the simplification is the empty list.

```
expand_and_simplify_facet_parameter_list ...
```

NOTE — The function `expand_and_simplify_facet_parameter_list` yields a facet parameter list simplification in which the parameter list part and the term list part are the concatenations of the parameter list parts and term list parts, respectively, of the simplifications of each of the facet parameter definitions resulting from expanding the given facet parameter list.

Examples:

The facet parameter definition

```
p :: k T
```

simplifies to a simplification containing the parameter definition

```
p :: T
```

and the term

```
p'k;
```

Name Expansion:

```
resolved_facet_signature :: ...

resolved_optional_facet_parameter_list :: ..

resolved_facet_parameter_definition :: ...

resolved_optional_kind :: ...

expand_names_in_facet_parameter_list ...
```

NOTE — The function `expand_names_in_facet_parameter_list` yields a resolved facet parameter list in which there are elements corresponding to each element of the given facet parameter list where any names are resolved references.

Denotational Semantics:

A simplified facet parameter definition defines a label and a type for a facet parameter. The value denoted by the parameter type expression in a simplified facet parameter definition shall be a type value, and is called the type of the facet parameter.

12.2 Anonymous facets

Concrete syntax:

anonymous_facet ::=
  facet *signature:*facet_signature **is**
    *exports:*optional_export_clause
    *declarations:*optional_declaration_list
  begin
    *terms:*optional_term_list
  end facet

optional_export_clause ::=
  [**export** export_list ;]

export_list ::=
  label_list
  | **all**

Abstract syntax:

```
anonymous_facet :: ...
```

```
optional_export_clause :: ...

export_list :: ...
```

Static semantics:

The parameters of an anonymous facet shall consist of the parameters defined in the parameter list of the signature of the anonymous facet followed by the parameters of the domain of the signature of the anonymous facet.

The declared items of an anonymous facet shall consist of the explicitly and implicitly declared items of the domain of the signature of the anonymous facet followed by the explictly and implicitly declared items of the optional declaration list of the anonymous facet.

The use clauses of an anonymous facet shall consist of the use clauses of the domain of the signature of the anonymous facet followed by the use clauses of the optional declaration list of the anonymous facet.

The terms of an anonymous facet shall consist of the terms of the domain of the signature of the anonymous facet followed by the terms of the optional term list of the anonymous facet.

If an anonymous facet omits an export clause, only those labels exported by the domain of the signature of the anonymous facet shall be exported by the anonymous facet. If an anonymous facet includes an export clause that includes a label list, the labels in the label list shall be exported in addition to those labels exported by the domain of the signature of the anonymous facet. If a label occurs more than once in the label list, the effect shall be the same as if the label occurred exactly once in the list. Each label in the label list shall be one of

- —the label of a parameter in the parameter list of the signature of the anonymous facet
- —the label of an explicitly declared item of the optional declaration list of the anonymous facet
- —the label of an implitly declared item of the optional declaration list of the anonymous facet
- —the label of a term of the optional term list of the anonymous facet

If an anonymous facet includes an export clause that includes the keyword **`all`**, then, in addition to those labels exported by the domain of the signature of the anonymous facet, the labels of all parameters in the parameter list of the signature of the anonymous facet, all explicitly declared items of the optional declaration list of the anonymous facet, all implitly declared items of the optional declaration list of the anonymous facet and all labeled terms of the optional term list of the anonymous facet shall be exported.

NOTE — The exported labels of a domain are all exported by a facet that extends the domain. The facet can only add labels to the collection of exported labels. This ensures that every facet that is of the type denoted by a domain exports at least the labels exported by the domain.

Simplification:

```
simplified_anonymous_facet :: ...

simplified_optional_export_clause :: ...

simplify_anonymous_facet ...
```

NOTE — The function `simplify_anonymous_facet` yields an anonymous facet where

- —The parameter list contains the parameter definitions from expansion and simplification of the parameters of the given anonymous facet;
- —The domain is the simplification of the domain expression of the given anonymous facet;
- —The export clause label list is present and contains a label list being one of

- •An empty list, if the export clause of the given anonymous facet is absent,
- •A list containing all parameter and definition labels and any term labels found in the result if the export clause of the given anonymous facet is present and **export all**, or
- •A list containing the same labels as the export clause of the given anonymous facet if the export clause of the given anonymous facet is present and a label list;

—The declaration list contains the declarations from simplification of the declarations of the given anonymous facet; and
—The term list contains the terms from simplification of the parameters and declarations of the given anonymous facet, and the simplification of the terms of the given anonymous facet.

Name Expansion:

```
resolved_anonymous_facet :: ...

resolved_optional_export_clause :: ...

expand_names_in_anonymous_facet ...
```

NOTE — The function `expand_names_in_anonymous_facet` yields a resolved anonymous facet where

—The parameter list corresponds to the parameter definition list of the given anonymous facet in which any names are resolved references;
—The domain expression corresponds to the domain expression of the given anonymous facet in which any names are resolved references;
—The export clause label list is that of the given anonymous facet;
—The declaration list corresponds to the declarations of the given anonymous facet without use clauses and in which any names are resolved references; and
—The term list corresponds to the term list of the given anonymous facet without use clauses and in which any names are resolved references.

Denotational semantics:

An anonymous facet expression shall denote a value that is a member of the *Facets* semantic domain described in 5.1.

To do: Work out facet denotation. This should be based on the transformed facet that has all of the parent domain stuff in it and is transformed into the null domain. May need to involve flattening of hierarchy by inclusion. Along the lines of a category of coalgebras whose observers are denoted by the items of the transformed facet and whose behaviors satisfy the boolean terms of the transformed facet.

12.3 Facet declarations

Concrete syntax:

facet_declaration ::=
  complete_facet_declaration
  | facet_interface_declaration
  | facet_body_declaration

complete_facet_declaration ::=
  facet *facet_label:*label *signature:*facet_signature **is**
    *exports:*optional_export_clause
    *declarations:*optional_declaration_list
  begin
    *terms:*optional_term_list
  end facet [*facet_label:*label] ;

facet_interface_declaration ::=
  facet interface *facet_label:*label *signature:*facet_signature **is**
    *exports:*optional_export_clause
    *declarations:*optional_declaration_list
  end facet interface [*facet_label:*label] ;

facet_body_declaration ::=
  facet body *facet_label:*label **is**
    *declarations:*optional_declaration_list
  begin
    *terms:*optional_term_list
  end facet body [*facet_label:*label] ;

Abstract syntax:

```
facet_declaration :: ...

complete_facet_declaration :: ...

facet_interface_declaration :: ...

facet_body_declaration :: ...
```

Static semantics:

If a complete facet declaration includes a facet label after the keywords **end facet**, that facet label shall be the same label as the label occurring at the beginning of the complete facet declaration. If a facet interface declaration includes a facet label after the keywords **end facet interface**, that facet label shall be the same label as the label occurring at the beginning of the facet interface declaration. If a facet body declaration includes a facet label after the keywords **end facet body**, that facet label shall be the same label as the label occurring at the beginning of the facet body declaration.

A facet may be declared by exactly one complete facet declaration or by a combination of exactly one facet interface declaration and exactly one facet body declaration. A facet declared by a facet interface declaration and a facet body declaration shall be equivalent to a facet declared by a complete facet declaration with the same label and signature as the facet interface declaration; the concatenation of the declaration lists of the facet interface declaration and a facet body declaration; and the term list of the facet body declaration.

If the facet interface declaration omits the export clause, the equivalent complete facet declaration shall also omit the export clause. If the facet interface declaration includes an export clause that includes a list of labels, the equivalent complete facet declaration shall include an export list with the same list of labels, and each label in the label list shall be one of

- —the label of a parameter in the parameter list of the signature of the facet interface declaration
- —the label of an explicitly declared item of the optional declaration list of the facet interface declaration
- —the label of an implitly declared item of the optional declaration list of the facet interface declaration

If the facet interface declaration includes an export clause that includes the keyword **all**, then the equivalent complete facet declaration shall include an export list that includes a label list that includes the labels of all parameters in the parameter list of the signature of the facet interface declaration, all explicitly declared items of the optional declaration list of the facet interface declaration and all implitly declared items of the optional declaration list of the facet interface declaration.

NOTE — The rules governing the export list of a facet interface declaration ensures that only those labels defined in the facet interface declaration are exported (in addition to those exported by the domain of the facet). Labels defined in the corresponding facet body declaration cannot be exported.

If a facet is declared by a facet interface declaration and a facet body declaration, those declarations shall occur in the same declarative region. A facet body declaration shall not occur in an enclosing interface declaration part unless the corresponding facet interface declaration also occurs in the enclosing interface declarative part. If a facet interface declaration occurs in an enclosing interface declarative part, the corresponding facet body declaration may occur either in the same enclosing interface declarative part or in the body declarative part corresponding to the enclosing interface declarative part.

Simplification:

```
simplify_facet_declaration_list ...
```

NOTE — The function `simplify_facet_declaration_list` yields a list of declaration simplifications that includes, for each complete facet declaration in the given facet declaration list, the result of applying `simplify_complete_facet_declaration` to the complete facet declaration, and for each matching pair of facet interface and body declarations, the result of applying `simplify_facet_interface_and_body` to the pair.

```
simplify_complete_facet_declaration ...
```

NOTE — The function `simplify_complete_facet_declaration` yields the simplification of a variable declaration where the declaration label is that of the given facet declaration, the declared type is the domain of the given facet, the definition expression is an anonymous facet, and the properry clause is absent. The signature, export clause, declaration list and term list of the anonymous facet are those of the given facet declaration.

```
simplify_facet_interface_and_body_declaration ...
```

NOTE — The function `simplify_facet_interface_and_body_declaration` yields the simplification of a complete facet declaration where the label is that shared by the given facet interface and body declarations, the signature is that of the given facet interface declaration, the export clause is that of the given facet interface declaration, the declaration list is the concatenation of the declaration lists of the given facet interface and body declarations, and the term list is that of the given facet body declaration.

Examples:

The Rosetta facet declaration

```
facet F ( x1 :: T1; x2, x3 :: T2 ) :: D is
   export y1, y2;
   ... declarations ...
begin
  ... terms ...
end facet F;
```

simplifies to the simplification of the variable declaration

```
F :: D is
  facet ( x1 :: T1; x2, x3 :: T2 ) :: D is
     export y1, y2;
     ... declarations ...
  begin
    ... terms ...
  end facet;
```

The Rosetta facet declarations


```
facet interface F ( x1 :: T1; x2, x3 :: T2 ) :: D is
   export y1, y2;
   ... interface declarations ...
begin
  ... terms ...
end facet F;

facet body F is
   ... body declarations ...
begin
  ... terms ...
end facet F;
```

simplifies to the simplification of the variable declaration

```
F :: D is
  facet ( x1 :: T1; x2, x3 :: T2 ) :: D is
     export y1, y2;
     ... interface declarations ...
     ... body declarations ...
  begin
    ... terms ...
  end facet;
```

12.4 Facet instantiation

Concrete syntax:

facet_instantiation ::=
  instance *instantiated_facet*:expression (*arguments:*facet_instantiation_argument_list)

facet_instantiation_argument_list ::=
  facet_instantiation_argument { , facet_instantiation_argument }

facet_instantiation_argument ::=
  expression | _

Abstract syntax:

```
facet_instantiation :: ...

facet_instantiation_argument_list :: ...

facet_instantiation_argument :: ...
```

Static semantics:

The instantiated facet expression shall be a facet expression.

Simplification:

```
simplified_facet_instantiation :: ...
```



```
simplified_facet_instantiation_argument_list :: ...

simplify_facet_instatiation ...
```

NOTE — The function `simplify_facet_instantiation` yields a facet instantiation in which the instantiated facet is the simplification of the instantiated facet of the given facet instantiation, and each argument is either the simplification of the corresponding argument of the given facet instantiation, if that argument is an expression, or the underline token otherwise.

Name Expansion:

```
resolved_facet_instantiation :: ...

resolved_facet_instantiation_argument_list :: ...

expand_names_in_facet_instatiation ...
```

NOTE — The function `expand_names_in_facet_instantiation` yields a facet instantiation in which the instantiated facet corresponds to the instantiated facet of the given facet instantiation in which any names are resolved references, and each argument is either the corresponding argument of the given facet instantiation in which any names are resolved, if that argument is an expression, or the underline token otherwise.

Denotational semantics:

To do: Define semantics of facet instantiation...

12.5 Facet expressions

An expression that involves facets or components shall be called a *facet expression*. Specifically, a facet expression shall be one of

- a name that refers to the label of a facet or component declaration
- a name that refers to the label of a variable that is declared with a variable definition that is a facet expression
- a function application of `rosetta.lang.prelude.facet_sum` or `rosetta.lang.prelude.facet_product` to arguments that are facet expressions
- a binary operation that is equivalent to a function application of `rosetta.lang.prelude.facet_sum` or `rosetta.lang.prelude.facet_product` to arguments that are facet expressions
- a facet instantiation in which the instantiated expression is a facet expression

The value denoted by a facet expression shall be called a *facet value*.

12.6 Facet inclusion

To do: Define syntax and semantics of facet inclusion.

...

13. Packages

This clause defines packages. The abstract syntax definitions in this clause are declared in the package `roset-ta.lang.reflect.abstract_syntax`.

13.1 Package declarations

Concrete syntax:

package_declaration ::=
  complete_package_declaration
  | package_interface_declaration
  | package_body_declaration

complete_package_declaration ::=
  package *package_label:*label *signature:*facet_signature **is**
    *exports:*optional_export_clause
    *declarations:*optional_declaration_list
  end package [*package_label:*label] ;

package_interface_declaration ::=
  package interface *package_label:*label *signature:*facet_signature
    *body_required:*optional_with_body_clause **is**
    *exports:*optional_export_clause
    *declarations:*optional_declaration_list
  end package interface [*package_label:*label] ;

optional_with_body_clause ::=
  [**with body**]

package_body_declaration ::=
  package body *package_label:*label **is**
    *declarations:*optional_declaration_list
  end package body [*package_label:*label] ;

Abstract syntax:

```
package_declaration :: ...

complete_package_declaration :: ...

package_interface_declaration :: ...

package_body_declaration :: ...
```

Static semantics:

If a complete package declaration includes a package label after the keywords **end package**, that package label shall be the same label as the label occurring at the beginning of the complete package declaration. If a package interface declaration includes a package label after the keywords **end package interface**, that package label shall be the same label as the label occurring at the beginning of the package interface declaration. If a package body declaration includes a package label after the keywords **end package body**, that package label shall be the same label as the label occurring at the beginning of the package body declaration.

A package may be declared by exactly one complete package declaration, or by exactly one package interface declaration, or by a combination of exactly one package interface declaration and exactly one package body declaration. A package declared by a package interface declaration that includes the words **`with body`** shall have a corresponding package body declaration. A package declared by a package interface declaration and a package body declaration shall be equivalent to a package declared by a complete package declaration with the same label and signature as the package interface declaration and the concatenation of the declaration lists of the package interface declaration and a package body declaration.

If the package interface declaration omits the export clause, the equivalent complete facet declaration shall also omit the export clause. If the facet interface declaration includes an export clause that includes a list of labels, the equivalent complete facet declaration shall include an export list with the same list of labels, and each label in the label list shall be one of

—the label of a parameter in the parameter list of the signature of the facet interface declaration
—the label of an explicitly declared item of the optional declaration list of the facet interface declaration
—the label of an implitly declared item of the optional declaration list of the facet interface declaration

If the package interface declaration omits the export clause or includes an export clause that includes the keyword **`all`**, then the equivalent complete package declaration shall include an export list that includes a label list that includes the labels of all parameters in the parameter list of the signature of the package interface declaration, all explicitly declared items of the optional declaration list of the package interface declaration and all implitly declared items of the optional declaration list of the package interface declaration.

NOTE — The rules governing the export list of a package interface declaration ensures that only those labels defined in the package interface declaration are exported (in addition to those exported by the domain of the package). Labels defined in the corresponding package body declaration cannot be exported.

If a package is declared by a package interface declaration and a package body declaration, those declarations shall occur in the same declarative region. A package body declaration shall not occur in an enclosing interface declaration part unless the corresponding package interface declaration also occurs in the enclosing interface declarative part. If a package interface declaration occurs in an enclosing interface declarative part, the corresponding package body declaration may occur either in the same enclosing interface declarative part or in the body declarative part corresponding to the enclosing interface declarative part.

Each facet parameter definition in the signature of a complete package declaration or a package interface declaration shall have the label design as the parameter kind or shall omit the parameter kind. If the parameter definition omits the parameter kind, the effect shall be as though the label `design` were included as the parameter kind.

Simplification:

```
simplify_package_declaration_list ...
```

NOTE — The function `simplify_package_declaration_list` yields a list of declaration simplifications that includes, for each complete package declaration in the given package declaration list, the result of applying `simplify_complete_package_declaration` to the complete package declaration, and for each matching pair of package interface and body declarations, the result of applying `simplify_package_interface_and_body` to the pair.

```
simplify_complete_package_declaration ...
```

NOTE — The function `simplify_complete_package_declaration` yields the simplification of a complete facet declaration where the label is that of the given package declaration, the signature is that of the given package declaration with all parameters having the kind "`design`", the export clause is that of the given package declaration if present or **export all** otherwise, the declarations are those of the given package declaration, and the term list is empty.

```
simplify_package_interface_and_body_declaration ...
```

NOTE — The function `simplify_package_interface_and_body_declaration` yields the simplification of a complete package declaration where the label is that shared by the given package interface and body declarations, the signature is that of the given package interface declaration, the export clause is that of the given package interface declaration, and the declaration list is the concatenation of the declaration lists of the given facet interface and body declarations.

Examples:

The Rosetta package declaration

```
package P ( x1 :: T1; x2, x3 :: T2 ) :: D is
   ... declarations ...
end package P;
```

simplifies to the facet declaration

```
facet P ( x1 :: design T1; x2, x3 :: design T2 ) :: D is
   export all;
   ... declarations ...
begin
end facet P;
```

The package interface and package body declarations

```
package interface P( x1 :: T1; x2, x3 :: T2 ) :: D is
   export v1, v2;
   ... interface declarations ...
end package interface P;

package body P is
   ... body declarations ...
end package body P;
```

simplify to the facet declaration

```
facet P( x1 :: design T1; x2, x3 :: design T2 ) :: D is
   export v1, v2;
   ... interface declarations ...
   ... body declarations ...
begin
end facet P;
```

13.2 Package expressions

An expression that involves packages shall be called a *package expression*. Specifically, a package expression shall be one of

- —a name that refers to the label of a package declaration
- —a name that refers to the label of a variable that is declared with a variable definition that is a package expression
- —a facet instantiation in which the instantiated expression is a package expression

The value denoted by a domain expression shall be called a *package value*.

14. Components

This clause defines component declarations. The abstract syntax definitions in this clause are declared in the package `rosetta.lang.reflect.abstract_syntax`.

14.1 Component declarations

Concrete syntax:

component_declaration ::=
  complete_component_declaration
  | component_interface_declaration
  | component_body_declaration

complete_component_declaration ::=
  component *component_label:*label *signature:*facet_signature **is**
    *exports:*optional_export_clause
    *declarations:*optional_declaration_list
  begin
    *component_assumptions:*assumptions_clause
    *component_definitions:*definitions_clause
    *component_implications:*implications_clause
  end component [*component_label:*label] ;

component_interface_declaration ::=
  component interface *component_label:*label *signature:*facet_signature **is**
    *exports:*optional_export_clause
    *declarations:*optional_declaration_list
  end component interface [*component_label:*label] ;

component_body_declaration ::=
  component body *component_label:*label **is**
    *declarations:*optional_declaration_list
  begin
    *component_assumptions:*assumptions_clause
    *component_definitions:*definitions_clause
    *component_implications:*implications_clause
  end component body [*component_label:*label] ;

assumptions_clause ::=
  assumptions
    *terms:*optional_term_list
  end assumptions ;

definitions_clause ::=
  definitions
    *terms:*optional_term_list
  end definitions ;

implications_clause ::=
  implications
    *terms:*optional_term_list

```
end implications ;
```

Abstract syntax:

```
component_declaration :: ...

complete_component_declaration :: ...

component_interface_declaration :: ...

component_body_declaration :: ...

assumptions_clause :: ...

definitions_clause :: ...

implications_clause :: ...
```

Static semantics:

If a complete component declaration includes a component label after the keywords **end component**, that component label shall be the same label as the label occurring at the beginning of the complete component declaration. If a component interface declaration includes a component label after the keywords **end component interface**, that component label shall be the same label as the label occurring at the beginning of the component interface declaration. If a component body declaration includes a component label after the keywords **end component body**, that component label shall be the same label as the label occurring at the beginning of the component body declaration.

A component may be declared by exactly one complete component declaration or by a combination of exactly one component interface declaration and exactly one component body declaration. A component declared by a component interface declaration and a component body declaration shall be equivalent to a component declared by a complete component declaration with the same label and signature as the component interface declaration; the concatenation of the declaration lists of the component interface declaration and a component body declaration; and the assumptions, definitions and implications clauses of the component body declaration.

If the component interface declaration omits the export clause, the equivalent complete component declaration shall also omit the export clause. If the component interface declaration includes an export clause that includes a list of labels, the equivalent complete component declaration shall include an export list with the same list of labels, and each label in the label list shall be one of

- —the label of a parameter in the parameter list of the signature of the component interface declaration
- —the label of an explicitly declared item of the optional declaration list of the component interface declaration
- —the label of an implitly declared item of the optional declaration list of the component interface declaration

If the component interface declaration includes an export clause that includes the keyword **all**, then the equivalent complete component declaration shall include an export list that includes a label list that includes the labels of all parameters in the parameter list of the signature of the component interface declaration, all explicitly declared items of the optional declaration list of the component interface declaration and all implitly declared items of the optional declaration list of the component interface declaration.

NOTE — The rules governing the export list of a component interface declaration ensures that only those labels defined in the component interface declaration are exported (in addition to those exported by the domain of the component). Labels defined in the corresponding component body declaration cannot be exported.

If a component is declared by a component interface declaration and a component body declaration, those declarations shall occur in the same declarative region. A component body declaration shall not occur in an enclosing interface declaration part unless the corresponding component interface declaration also occurs in the enclosing interface declarative part. If a component interface declaration occurs in an enclosing interface declarative part, the corresponding component body declaration may occur either in the same enclosing interface declarative part or in the body declarative part corresponding to the enclosing interface declarative part.

To do: Check whether facet_product is right in the above, or whether is should be facet_sum.

Simplification:

To do: Revise the definition of elaborate_complete_component_declaration depending on whether facet_product or facet_sum should be used.

```
simplify_component_declaration_list ...
```

NOTE — The function `simplify_component_declaration_list` yields a list of declaration simplifications that includes, for each complete component declaration in the given component declaration list, the result of applying `simplify_complete_component_declaration` to the complete component declaration, and for each matching pair of component interface and body declarations, the result of applying `simplify_component_interface_and_body` to the pair.

```
simplify_complete_component_declaration ...
```

NOTE — The function `simplify_complete_component_declaration` yields the simplification of a complete facet declaration where the label, signature, export clause and declaration list are those of the component declaration, and the term list contains two unlabeled terms whose term expressions are:

- —An anonymous facet that has no parameters, the same domain as the component declaration, no export clause, no declarations list, and a term list the same as the definitions term list of the component declaration; and
- —A function application where
  - •The applied function is `rosetta.lang.prelude.facet_implies`,
  - ²The first argument is a nested function application where the applied function is `rosetta.lang.prelude.facet_product`, the first argument of the nested function application is an anonymous facet that has no parameters, the same domain as the component declaration, no export clause, no declarations list, and a term list the same as the assumptions term list of the component declaration, and the second argument of the nested function application is the same anonymous facet as that of the first term of the simplification result;
  - •The second argument is an anonymous facet that has no parameters, the same domain as the component declaration, no export clause, no declarations list, and a term list the same as the implications term list of the given component declaration.

```
simplify_component_interface_and_body_declaration ...
```

NOTE — The function `simplify_component_interfec_and_body_declaration` yields the simplification of a complete component declaraion where the label is that shared by the component interface and body declarations, the signature is that of the given component interface declaration, the export clause is that of the given component interface declaration, the declarations are the concatenation of the declaration lists of the given component interface and body declarations, and the assumptions, definitions, and implications clauses are those of the given component body declaration.

Examples:

The Rosetta component declaration

```
component C ( x1 :: T1; x2, x3 :: T2 ) :: D is
   ... declarations ...
begin
  assumptions
    ... assumption terms ...
  end assumptions;
```

```
  definitions
    ... definition terms ...
  end definitions;
  implications
    ... implication terms ...
  end implications;
end component C;
```

simplifies to the facet declaration

```
facet C ( x1 :: T1; x2, x3 :: T2 ) :: D is
   ... declarations ...
begin
  facet :: D is begin
    ... definition terms ...
  end facet;
  rosetta.lang.prelude.facet_implies(
    rosetta.lang.prelude.facet_product(
      facet :: D is begin
        ... assumption terms ...
      end facet,
      facet :: D is begin
        ... definition terms ...
      end facet),
    facet :: D is begin
      ... implication terms
    end facet);
end facet C;
```


15. Domains

This clause defines ...

15.1 Domain declarations

Concrete syntax:

domain_declaration ::=
  domain *domain_label:*label *signature:*domain_signature **with** *extending_facet_AST:*label **is**
    exports *exports:*optional_export_list **with** *additional_exports_expression:*expression ;
    *declarations:*optional_declaration_list **with** *additional_declarations_expression:*expression
  begin
    *terms:*optional_term_list **with** *additional_terms_expression:*expression
  end domain [*domain_label:*label] ;

domain_signature ::=
  *parameters:*optional_facet_parameter_list *parent_domain:*optional_parent_domain

optional_parent_domain ::=
  [:: *parent_domain:*optional_expression]

optional_export_list ::=
  [*exports:*export_list]

Abstract syntax:

```
domain_declaration :: ...

domain_signature :: ...

optional_export_list :: ...
```

Static semantics:

If a domain declaration includes a domain label after the keywords **end domain**, that domain label shall be the same label as the label occurring immediately before the signature.

Each facet parameter definition in the signature of a domain declaration shall have the label design as the parameter kind or shall omit the parameter kind. If the parameter definition omits the parameter kind, the effect shall be as though the label design were included as the parameter kind.

If the signature of a domain declaration includes a parent domain, the parent domain shall be a domain expression.

The additional exports expression of a domain declaration shall be an expression of type `rosetta.lang.reflect.optional_export_list`. The additional declarations expression of a domain declaration shall be an expression of type `rosetta.lang.reflect.optional_declaration_list`. The additional terms expression of a domain declaration shall be an expression of type `rosetta.lang.reflect.optional_term_list`.

Simplification:


```
simplified_domain_declaration :: ...

simplified_domain_signature :: ...

simplify_domain_declaration ...
```

NOTE — The function `simplify_domain_declaration` yields a simplification in which

—Tthe domain label is that of the domain declaration,
—The signature contains the simplified parameter definitions resulting from simplification of the domain declaration signature, and the simplification of the domain expression of the domain declaration signature, if present or no domain expression otherwise,
—The exports clause that is absent if the domain declaration has no export clause, or that contains all of the parameter, declaration and term labels of the domain declaration if the domain declaration export clause is **export all**, or the domain declaration export clause if that clause contains a list of labels,
—The declarations are the simplified declarations resulting from simplification of the declarations of the domain declaration,
—The terms are the simplified terms resulting from simplification of the parameters, declarations and terms of the domain declaration,
—The additional exports, declarations and terms expressions are the simplifications of the additional exports, declarations and terms expressions, respectively, of the domain declaration.

Name Expansion:

```
resolved_domain_declaration :: ...

resolved_domain_signature :: ...

expand_names_in_domain_declaration ...
```

NOTE — The function `expand_names_in_domain_declaration` yields a resolved domain declaration in which

—Tthe domain label is that of the domain declaration,
—The signature contains the parameter definitions corresponding to the parameter definitions of the signature of the given domain in which any names are resolved references, and, if present, the domain expression corresponds to the domain expression of the signature of the given domain declaration in which any names are resolved references, or no domain expression otherwise,
—The exports clause is that of the given domain declaration,
—The declarations correspond to the declarations of the given domain declaration in which any names are resolved references,
—The terms correspond to the terms of the given domain declaration in which any names are resolved references,
—The additional exports, declarations and terms expressions correspond to the additional exports, declarations and terms expressions, respectively, of the given domain declaration in which any names are resolved references.

Denotational semantics:

To do: put something sensible in here. Domains are not really involved in the interpretation of a specification per se. They are used in the elaboration of facets; they describe how to transform a facet that extends the domain into a facet that extends the parent of the domain. This occurs recursively to produce a facet in a root domain, such as the null domain. So transformation of a facet involves using the domain it refines and all the ancestors of that domain to transform a facet into one that explicitly describes all the semantics that were implicitly obtained by extending a particular domain. This gives us a forest of facets that are all “domainless” and which we will proceed to elaborate by “flattening” to “the coalgebra”, and the domain declarations have disappeared in the wash so to speak since describing the interpretation of the forest will not involve domains anymore. However I can still refer to the things declared in domains, so the interpretation of the forest will need to appeal to the “facet-equivalent value” of the domain (i.e. the domain with all the bits that can’t be in a facet stripped out in some way).

15.2 Domain expressions

An expression that involves domains shall be called a *domain expression*. Specifically, a domain expression shall be one of

- a name that refers to the label of a domain declaration
- a name that refers to the label of a variable that is declared with a variable definition that is a domain expression
- a facet instantiation in which the instantiated expression is a domain expression

The value denoted by a domain expression shall be called a *domain value*.

16. Interactions

This clause defines interactions, which specify the way in which properties described by a facet in one domain are manifest in another domain.

16.1 Interaction declarations

Concrete syntax:

interaction_declaration ::=
interaction *interaction_label:*label *signature:*interaction_signature **is**
with *source_AST_label:*label
*target_AST:*target_AST_definition
end interaction [*interaction_label:*label] ;

interaction_signature ::=
(*parameter:*parameter_definition) :: *return_type:*expression

target_AST_definition ::=
let_target_AST_definition
| template_target_AST_definition

let_target_AST_definition ::=
let
*binding_list:*let_binding_list
in
*template:*template_target_AST_definition
end let ;

template_target_AST_definition ::=
facet (*parameters_expression:*expression) :: *domain_expression:*expression **is**
export *exports_expression:*expression ;
*declarations_expression:*expression ;
begin
*terms_expression:*expression ;
end facet ;

Abstract syntax:

```
interaction_declaration :: ...

interaction_signature :: ...

target_AST_definition :: ...

let_target_AST_definition :: ...

template_target_AST_definition :: ...
```

The values denoted by the parameter type and the return type of the interaction signature shall be of the type `rosetta.lang.null`. The value denoted by the parameters expression shall be of the type `rosetta.lang.reflect.abstract_syntax.optional_facet_parameter_list`. The value denoted by the domain expression shall be of the type `rosetta.lang.reflect.abstract_syntax.expression`. The value denoted by the exports expression shall be of the type `rosetta.lang.reflect.abstract_syntax.optional_export_clause`. The value denoted by the declarations expression shall be of the type `rosetta.lang.reflect.abstract_syntax.optional_declaration_list`. The value denoted by the terms expression shall be of the type `rosetta.lang.reflect.abstract_syntax.optional_term_list`.

An interaction declaration shall be equivalent to a function declaration. An interaction declaration of the form

```
interaction interaction_label ( parameter_label :: parameter_type ) :: return_type is
  with source_AST_label
  facet ( parameters_expression ) :: domain_expression is
    export exports_expression ;
    declarations_expression ;
  begin
    terms_expression ;
  end facet ;
end interaction [ interaction_label:label ] ;
```

shall be equivalent to a function declaration of the form

```
interaction_label ( parameter_label :: parameter_type ) :: return_type where
  rosetta.lang.prelude.forall (
    source_AST_label :: rosetta.lang.prelude.sel (
      _source_AST_label :: rosetta.lang.reflect.abstract_syntax.anonymous_facet |
          rosetta.lang.reflect.semantics.interpretation ( _source_AST_label )
              in parameter_type ) |
    rosetta.lang.prelude.__=__ (
      interaction_label (
        rosetta.lang.reflect.semantics.interpretation ( source_AST_label ) ),
      rosetta.lang.reflect.semantics.interpretation (
        rosetta.lang.reflect.abstract_syntax.make_anonymous_facet (
          parameters_expression,
          domain_expression,
          exports_expression,
          declarations_expression,
          terms_expression ) ) ) );
```

Example:

Provided all names from `rosetta.lang.prelude`, `rosetta.lang.reflect.abstract_syntax` and `rosetta.lang.reflect.semantics` are directly visible and not hidden, the interaction declaration

```
interaction F ( x :: D1 ) :: D2 is
  with x_AST
  facet ( make_parameters ( x_AST ) ) :: make_domain ( x_AST ) is
    export make_exports ( x _AST );
    make_declarations ( x_AST );
  begin
    make_terms ( x_AST );
  end facet;
end interaction F;
```

is equivalent to the function declaration


```
F ( x :: D1 ) :: D2 where
  forall ( x_AST :: sel ( _x_AST :: anonymous_facet |
                          interpretation ( _x_AST ) in D1 ) |
          F ( interpretation ( x_AST )
          = interpretation (
             make_anonymous_facet (
                  make_parameters ( x_AST ),
                  make_domain ( x_AST ),
                  make_exports ( x _AST ),
                  make_declarations ( x_AST ),
                  make_terms ( x_AST ) ) ) ) );
```

An interaction declaration of the form

```
interaction interaction_label ( parameter_label :: parameter_type ) :: return_type is
  with source_AST_label
  let
    binding_list
  in
    facet ( parameters_expression ) :: domain_expression is
      export exports_expression ;
      declarations_expression ;
    begin
      terms_expression ;
    end facet ;
  end let ;
end interaction [ interaction_label:label ] ;
```

shall be equivalent to a function declaration of the form

```
interaction_label ( parameter_label :: parameter_type ) :: return_type where
  rosetta.lang.prelude.forall (
    source_AST_label :: rosetta.lang.prelude.sel (
      _source_AST_label :: rosetta.lang.reflect.abstract_syntax.anonymous_facet |
          rosetta.lang.reflect.semantics.interpretation ( _source_AST_label )
              in parameter_type ) |
    rosetta.lang.prelude.__=__ (
      interaction_label (
        rosetta.lang.reflect.semantics.interpretation ( source_AST_label ) ),
      rosetta.lang.reflect.semantics.interpretation (
        let
          binding_list
        in
          rosetta.lang.reflect.abstract_syntax.make_anonymous_facet (
            parameters_expression,
            domain_expression,
            exports_expression,
            declarations_expression,
            terms_expression )
        end let ) ) );
```

Example:

Provided all names from `rosetta.lang.prelude`, `rosetta.lang.reflect.abstract_syntax` and `rosetta.lang.reflect.semantics` are directly visible and not hidden, the interaction declaration

```
interaction F ( x :: D1 ) :: D2 is
  with x_AST
  let
    y :: T be E
  in
    facet ( make_parameters ( x_AST, y ) ) :: make_domain ( x_AST, y ) is
      export make_exports ( x _AST, y );
      make_declarations ( x_AST, y );
    begin
      make_terms ( x_AST, y );
    end facet;
  end let;
end interaction F;
```

is equivalent to the function declaration

```
F ( x :: D1 ) :: D2 where
  forall ( x_AST :: sel ( _x_AST :: anonymous_facet |
                              interpretation ( _x_AST ) in D1 ) |
            F ( interpretation ( x_AST )
            = interpretation (
                  let
                    y :: T be E
                  in
                    make_anonymous_facet (
                          make_parameters ( x_AST, y ),
                          make_domain ( x_AST, y ),
                          make_exports ( x _AST, y ),
                          make_declarations ( x_AST, y ),
                          make_terms ( x_AST, y ) )
                  end let ) ) );
```

17. Libraries

This clause defines libraries that contain Rosetta specifications. It also defines the rules governing order of analysis of design units that comprise Rosetta specifications.

17.1 Design units

A *design unit* shall be a portion of a Rosetta specification that may be independently analyzed and inserted into a design library. A *design file* shall be a sequence of one or more design units. The representation of a design file shall be implementation dependent.

NOTE — An implementation will typically represent a design file as a text file containing the text of one or more design units in sequence.

Concrete syntax:

```
design_file ::=
   design_unit { design_unit }

design_unit ::=
   complete_or_interface_design_unit
   | body_design_unit

complete_or_interface_design_unit ::=
   context:context_clause_list unit:complete_or_interface_declaration

complete_or_interface_declaration ::=
   complete_declaration
   | interface_declaration

complete_declaration ::=
   complete_facet_declaration
   | complete_package_declaration
   | complete_component_declaration
   | domain_declaration
   | interaction_declaration
   | variable_declaration

interface_declaration ::=
   facet_interface_declaration
   | package_interface_declaration
   | component_interface_declaration

context_clause_list ::=
   { context_clause }

context_clause ::=
   library_clause
   | use_clause

body_design_unit ::=
   context:library_clause_list unit:body_declaration
```

body_declaration ::=
  facet_body_declaration
  | package_body_declaration
  | component_body_declaration

library_clause_list ::=
  { library_clause }

Abstract syntax:

```
design_file :: ...

design_unit :: ...

complete_or_interface_design_unit :: ...

complete_or_interface_declaration :: ...

complete_declaration :: ...

interface_declaration :: ...

optional_context_clause_list :: ...

context_clause :: ...

body_design_unit :: ...

body_declaration :: ...
```

Static semantics:

A design unit that is a variable declaration shall have a declared type that is a facet, package, component or domain type; a definition that is a constant facet expression, a constant package expression or a constant domain expression; and no property clause.

Simplification:

```
design_unit_simplification :: ...

optional_design_unit_simplification_list :: ...

labels_from_design_unit_simplification_list ...
```

NOTE — The function `labels_from_design_unit_simplification_list` yields a list of labels of design units in the given design unit simplification list.

```
simplify_design_unit_list
  ( l :: sequence ( design_unit ) )
  :: optional_design_unit_simplification_list is ...
```

NOTE — The function `simplify_design_unit_list` yields a list of design unit simplifications obtained by simplifying each of the design units in the list `l`. Each of the variables bound in the let expression represents those design units from the list `l` of a given type. The function filters the design units from the list `l` by applying the observer function `unit` to obtain the declaration without the context clause, composed with the recognizer predicate for the given declaration type. The function applies simplification functions to each of the variables and concatenates the results.

```
merge_use_names ( cc :: context_clause ) :: optional_name_list is ...
```

NOTE — The function `merge_use_names` yields the list of names referred to in the use clauses of a context clause. The function filters the context clause to obtain the use clauses, then applies the `used_names` observer to each use clause. The function then concatenates the the lists of used names to obtain its result.

```
simplify_variable_units ( n :: sequence ( design_unit ) )
  :: optional_design_unit_simplification_list is ...
```

NOTE — The function `simplify_variable_units` yields a list of design unit simplifications obtained by simplifying each of the variable declarations in the list of design units, `l`. The function simplifies each variable declaration, yielding a list of simplified declarations. For each of those simplified declarations, the function makes a design unit simplification from the list of used names in the context clause of the design unit, together with the simplified declaration. The function then concatenates the lists of design unit simplifications to obtain its result.

```
simplify_facet_units
   ( n :: sequence ( design_unit ) )
   :: optional_design_unit_simplification_list is ...
```

NOTE — The function `simplify_facet_units` yields a list of design unit simplifications obtained by simplifying each of the facet declarations in the list of design units, n. The function filters out the complete, interface and body design units into separate lists. For each complete design unit, the function makes a design unit simplification from the list of used names in the context clause of the design unit, together with the simplified facet declaration. For each interface design unit, the function finds the matching body design unit, then makes a design unit simplification from the list of used names in the context clause of the interface design unit, together with the simplified facet interface and body declarations. The function concatenates the two lists of design unit simplifications to obtain its result.

```
simplify_package_units
   ( n :: sequence ( design_unit ) )
   :: optional_design_unit_simplification_list is ...
```

NOTE — The function `simplify_package_units` is similar to the function `simplify_facet_units`; it yields a list of design unit simplifications obtained by simplifying each of the package declarations in the list of design units, n. The function filters out the complete, interface and body design units into separate lists. For each complete design unit, the function makes a design unit simplification from the list of used names in the context clause of the design unit, together with the simplified package declaration. For each interface design unit, the function finds the matching body design unit, then makes a design unit simplification from the list of used names in the context clause of the interface design unit, together with the simplified package interface and body declarations. The function concatenates the two lists of design unit simplifications to obtain its result.

```
simplify_component_units
   ( n :: sequence ( design_unit ) )
   :: optional_design_unit_simplification_list is ...
```

NOTE — The function `simplify_component_units` is similar to the function `simplify_facet_units`; it yields a list of design unit simplifications obtained by simplifying each of the component declarations in the list of design units, n. The function filters out the complete, interface and body design units into separate lists. For each complete design unit, the function makes a design unit simplification from the list of used names in the context clause of the design unit, together with the simplified component declaration. For each interface design unit, the function finds the matching body design unit, then makes a design unit simplification from the list of used names in the context clause of the interface design unit, together with the simplified component interface and body declarations. The function concatenates the two lists of design unit simplifications to obtain its result.

```
simplify_domain_units
   ( n :: sequence ( design_unit ) )
   :: optional_design_unit_simplification_list is ...
```

NOTE — The function `simplify_domain_units` yields a list of design unit simplifications obtained by simplifying each of the domain declarations in the list of design units, n. For each design unit, the function makes a design unit simplification from the list of used names in the context clause of the design unit, together with the simplified domain declaration. The result of the function is the list of design unit simplifications.

```
simplify_interaction_units
   ( n :: sequence ( design_unit ) )
   :: optional_design_unit_simplification_list is ...
```

NOTE — The function `simplify_interaction_units` yields a list of design unit simplifications obtained by simplifying each of the interaction declarations in the list of design units, n. For each design unit, the function makes a design unit simplification from the list of used names in the context clause of the design unit, together with the simplified interaction declaration. The result of the function is the list of design unit simplifications.

Name Expansion:

```
expand_names_in_design_unit
   :: resolved_variable_declaration_and_property_simplification is ...
```

NOTE — The function `expand_names_in_design_unit` yields a resolved variable declaration and property simplification if the declaration of the given simplified design unit is a variable declaration and property simplification, or a resolved domain declaration if the declaration of the given simplified design unit is a domain declaration, in either case corresponding to the declaration of the given simplified domain declaration where any names are resolved references.

17.2 Design libraries

A *design library* shall be a hierarchical repository for storing analyzed design units. The representation of a design library shall be implementation dependent.

A design library shall contain all of the analyzed design units that are stored in it. A design library may also contain one or more subordinate design libraries. A design library that contains a subordinate design library shall be called the *parent library* of the subordinate library.

A design library that is not a subordinate design library of any other design library shall be called a *root design library*. A root design library that is recursively the parent of a subordinate design library shall be called the *ultimate parent* of the subordinate design library. A root design library shall be its own ultimate parent.

Abstract syntax:

```
design_library :: ...
```

Static semantics:

A design library shall be considered equivalent to a complete package declaration in which:

—The label is defined in an implementation-dependent manner
—The parameter list is empty
—The domain is rosetta.lang.null
—The export clause is absent
—The declarations comprise the declarations of the analyzed design units store in the library, together with the declarations of the equivalent package declarations of each subordinate design library of the design library

The label of the equivalent package declaration shall be called the label of the design library.

NOTE — Since a subordinate design library is considered as a package declared within the package corresponding to the parent library, the subordinate library may be referred to by a qualified name in which the prefix denotes the parent library and the suffix is the label of the subordinate library. Furthermore, since an analyzed design unit is considered to be declared within the equivalent package of the containing design library, the design unit may be referred to by a qualified name in which the prefix denotes the containing library and the suffix is the label of the design unit.

Simplification:

```
design_library_simplification :: ...

optional_design_library_simplification_list :: ...

simplify_design_library
  ( l :: design_library ) :: design_library_simplification is ...
```

NOTE — The function `simplify_design_library` yields a design library simplification whose label is that of the argument design library, whose design units are the simplified design units of the argument design library, and whose sublibraries are the simplified sublibraries of the argument design library.

Name Expansion:

```
expand_names_in_design_library
  :: resolved_variable_declaration_and_property_simplification is ...
```

NOTE — The function `expand_names_in_design_unit` yields a resolved variable declaration and property simplification which is equivalent to the result of simplifying and expanding the names in a package declaration where the declared label is the same as the library label, the domain expression is a reference to the predefined domain rosetta.lang.null, and the elements of the declaration list correspond to the results of expanding names in the design units and sublibraries of the given design library.

17.2.1 Predefined design libraries

There shall be a predefined root design library with the label `rosetta`. The predefined library `rosetta` shall contain a subordinate library with the label `lang`.

The predefined library `rosetta.lang` shall contain the following predefined design units:

—The package `prelude`, as defined in Clause 21
—The package `unicode`, as defined in Clause 22
—The predefined domains defined in Clause 24

The predefined library `rosetta.lang` shall contain a subordinate library with label `reflect`. The predefined library `rosetta.lang.reflect` shall contain the predefined design units defined in Clause 23.

17.2.2 Library clauses

Concrete syntax:

library_clause ::=
  library library_name_list ;

library_name_list ::=
  library_name { , library_name }

library_name ::=
  *library_label:*label *library_identifier:*optional_library_identifier

optional_library_identifier ::=
  [**is** string_literal]

Abstract syntax:

```
library_clause :: ...

optional_library_clause_list :: ...

library_name :: ...

library_name_list :: ...

optional_library_identifier :: ...
```

Static semantics:

A library label shall be the label of a root design library. If a library name omits the optional library identifier, the root design library shall be determined in an implementation-dependent manner. If a library name includes the optional library identifier, the string literal shall be in the form of a Universal Resource Identifier (URI) that shall identify the root design library.

Examples:

```
library design_lib, standard_components;

library company_lib is "file:///usr/local/lib/rosetta/company_lib";

library widgets is "http://www.widget.com/libraries/widgets";
```

18. Type Rules

This clause defines the static type rules governing the language. The definitions in this clause are declared in the package `rosetta.lang.reflect.type_rules`.

18.1 Type contexts

A type context shall be a sequence of type bindings. A type binding shall be a pair whose first element is a name and whose second element is an abstract syntax value of an expression that denotes a type.

```
type_binding :: ......
```

A type expression is an expression defined as:

```
type_expression :: ...
```

```
type_constructor :: ...
```

```
type_constructor_application :: ...
```

A well formed type context shall be a type context for which the function `well_formed` is true.

```
well_formed...
```

```
well_formed_type_context :: ...
```

```
name_in_type_context...
```

```
type_binding_in_type_context...
```

18.2 Type statements

A type statement shall be an expression of the form

type_context |– *expression* ∈ *type_expression*

where *type_context* shall be a Rosetta expression of type `type_context`, *expression* shall be a Rosetta expression of type `expression`, and *type_expression* shall be a Rosetta expression of type `type_expression`. A type statement shall be equivalent to an expression of the form

`in_type (` *type_context* `,` *expression* `,` *type_expression* `)`

where the function `in_type` shall be defined as

```
in_type ( G :: well_formed_type_context;
          e :: expression; T :: type_expression )
  :: boolean;
```


18.3 Type rules for expressions

18.3.1 Type rules for literals

```
T_Literal_Bottom :...
```

```
T_Literal_Real :...
```

```
T_Character_Literal :...
```

```
T_String_Literal :...
```

```
T_Bitvector_Literal :...
```

18.3.2 Type rule for names

```
T_Name :...
```

18.3.3 Type rule for let expressions

```
T_Let_Expression :...
```

18.3.4 Type rule for if expressions

```
T_If_Expression :...
```

18.4 Type rules for functions

18.4.1 Type rule for function type formations

```
T_Function_Type_Formation :...
```

18.4.2 Type rule for anonymous functions

```
T_Anonymous_Function :...
```

18.4.3 Type rule for function application

```
T_Function_Application :...
```

18.5 Type rules for declarations

18.5.1 Type rule for variable declarations

```
T_Variable_Declaration :...
```

18.5.2 Type rule for constructed type declarations

```
T_Constructed_Type_Declaration :...
```

19. Scope and visibility

This clause defines the scope and visibility rules that govern the definition and use of labels within Rosetta specifications.

19.1 Declarative regions

A declarative region shall be a portion of the text of a Rosetta specification, and shall be one of

—A complete facet declaration (excluding the facet label)
—A facet interface declaration (excluding the facet label) together with the corresponding facet body declaration
—A complete package declaration (excluding the package label)
—A package interface declaration (excluding the package label) together with the corresponding package body declaration (if any)
—A complete component declaration (excluding the component label)
—A component interface declaration (excluding the component label) together with the corresponding component body declaration
—A domain declaration (excluding the domain label)
—A let expression
—A quantified expression
—A function type formation
—An anonymous function
—A function declaration (excluding the function label)
—A constructed type declaration (excluding the type label)
—A let term
—A root declarative region for the purpose of analyzing a design unit

A declaration, parameter definition or term is said to *declare* one or more labels. A label is said to be declared *immediately within* a declarative region if the declarative region is the innermost declarative region in which the declaration, parameter definition or term occurs. If a label is declared immediately within a declarative region, the declarative region is said to *immediately declare* the label.

Certain declarative regions consist of disjoint portions of text. Each such declarative regions shall nontheless be considered as a single declarative region comprising the disjoint portions of text but excluding any intervening portions of text.

19.2 Scope of a label

Associated with each declared label there shall be a portion of the text of the Rosetta specification called the *scope* of the label. The scope shall comprise the *immediate scope* and may also be extended beyond the immediate scope.

The immediate scope of a label declared immediately within a complete facet declaration, a complete package declaration, a complete component declaration or a domain declaration shall extend from the keyword **`is`** in the enclosing declaration to the end of the enclosing declaration.

The immediate scope of a label declared immediately within a facet interface declaration, a package interface declaration or a component interface declaration shall extend from the keyword **`is`** in the enclosing interface declaration to the end of the enclosing interface declaration. The immediate scope shall further extend from the keyword **`is`** in the corresponding body declaration to the end of the corresponding body declaration (if any).

The immediate scope of a label declared immediately within a facet body declaration, a package body declaration or a component body declaration shall extend from the keyword **is** in the enclosing body declaration to the end of the enclosing body declaration.

The immediate scope of a label declared in a parameter definition in a facet, package, component or domain parameter list shall extend from the beginning of the parameter definition to the end of the enclosing declaration. Where the enclosing declaration is an interface declaration, the immediate scope of the label shall further extend from the keyword **is** in the corresponding body declaration to the end of the corresponding body declaration (if any).

The immediate scope of a label declared in a parameter definition in a let expression shall extend from the beginning of the let expression to the end of the let expression.

The immediate scope of a label declared in a parameter definition in a quantified expression shall extend from the beginning of the parameter definition to the end of the quantified expression.

The immediate scope of a label declared in a parameter definition in a function type formation shall extend from the beginning of the parameter definition to the end of the function type formation.

The immediate scope of a label declared in a parameter definition in an anonymous function shall extend from the beginning of the parameter definition to the end of the anonymous function.

The immediate scope of a label declared in a parameter definition in a function declaration shall extend from the beginning of the parameter definition to the end of the function declaration.

The immediate scope of a label declared in a parameter definition in a constructed type declaration shall extend from the beginning of the parameter definition to the end of the constructed type declaration.

The immediate scope of a label declared in a parameter definition in a let term shall extend from the beginning of the let term to the end of the let term.

NOTE — The rules specifying the scope of a label declared in a declaration refer to both explicit and implicit declarations.

The scope of a label that is exported by an export clause shall be further extended beyond the immediate scope of the label.

19.3 Visibility

Within the scope of a label, there shall be places where the label is said to be *visible*. An occurrence of the label in a place where it is visible shall be a reference to the declaration, parameter definition or term that declares the label.

A label shall be visible only in places where it is *directly visible* or where it is *visible by selection*.

A label shall be directly visible in parts of its immediate scope except in those places where it is *hidden*. A label shall be hidden in a declarative region that is nested within the scope of the label if the nested declarative region immediately declares the same label.

A label declared in a declaration or a term shall be directly visible in all of its immediate scope, except where the label is hidden.

A label declared in a parameter definition shall be directly visible in all of its immediate scope except in the parameter definition that declares the label and except where the label is hidden.

A label shall also be directly visible in places where it is made directly visible by a use clause.

A label shall be visible by selection at places defined as follows:

a) For a label that is immediately declared within an anonymous facet that is the value of a variable: at the place of the suffix in a qualified name that occurs within the immediate scope of the label and whose prefix denotes the variable.
b) For a label that is immediately declared within a facet declaration, a package declaration, a component declaration or a domain declaration: at the place of the suffix in a qualified name that occurs within the immediate scope of the label and whose prefix refers to the label of the declaration that immediately declares the label.
c) For an exported label of an anonymous facet that is the value of a variable: at the place of the suffix in a qualified name whose prefix denotes the variable.
d) For an exported label of a facet declaration, a package declaration or a component declaration: at the place of the suffix in a qualified name whose prefix refers to the label of the facet declaration, package declaration or component declaration.
e) For an exported label of an anonymous facet that is instantiated in a term: at the place of the suffix in a qualified name whose prefix refers to the label of the term.
f) For an exported label of a facet declaration, a package declaration or a component declaration that is instantiated in a term: at the place of the suffix in a qualified name whose prefix refers to the label of the term.

19.4 Use clauses

Concrete syntax:

use_clause ::=
  use *used_names:*name_list ;

name_list ::=
  name { , name }

Abstract syntax:

```
use_clause :: ...

name_list :: ...
```

Static semantics:

A use clause shall make directly visible exported labels that are visible by selection.

A use clause is said to occur *immediately within* a declarative region if the declarative region is the innermost declarative region in which the use clause explicitly or implicitly occurs.

Each name in the name list of a use clause shall be a static expression that denotes a package value. Two names that occur in the same use clause or in different use clauses that both occur immediately within the same declarative region shall denote different package values. A name in a use clause that appears immediately within a declarative region shall not be or have as a prefix a label that is made directly visible by the same use clause or another use clause that appears immediately within the same declarative region.

Rationale: The restriction on labels in use clauses eliminates cases where a used package name is a name imported from another package. It would otherwise be possible to introduce ambiguity and potentially circular imports. Checking for these problems could be complex and burdensome.

Associated with a use clause there shall be a portion of the text of the Rosetta specification called the *scope* of the use clause.

The scope of a use clause that occurs immediately within a complete facet declaration, a complete package declaration, a complete component declaration or a domain declaration shall extend from the keyword **is** in the enclosing declaration to the end of the enclosing declaration.

The scope of a use clause that occurs immediately within a facet interface declaration, a package interface declaration or a component interface declaration shall extend from the keyword **is** in the enclosing interface declaration to the end of the enclosing interface declaration. The scope shall further extend from the keyword **is** in the corresponding body declaration to the end of the corresponding body declaration (if any).

The scope of a use clause that occurs immediately within a facet body declaration, a package body declaration or a component body declaration shall extend from the keyword **is** in the enclosing body declaration to the end of the enclosing body declaration.

The scope of a use clause that occurs in the context clause of a complete or interface design unit shall extend from the beginning of the design unit to the end of the design unit. If the design unit is a facet interface declaration, a package interface declaration or a component interface declaration, the scope of the use clause shall further extend from the keyword **is** of the corresponding body declaration to the end of the corresponding body declaration (if any).

For each name in the name list of a use clause, the exported labels of the denoted package shall be made *potentially visible* in the scope of the use clause. A label shall be made directly visible by a use clause in those places where it is potentially visible and

a)It is not hidden, and
b)it is not the same label as an exported label from a different package also made potentially visible by the same use clause or by a use clause that occurs immediately within the same declarative region, and
c)it is not the same label as a label declared immediately within the declarative region in which the use clause immediately occurs.

20. Analysis, elaboration and interpretation

This clause defines three steps of processing a Rosetta specification by a tool: analysis, elaboration and interpretation.

20.1 Analysis of design units

A design unit shall be analyzed by ensuring that the text of the design unit is lexically and syntactically well-formed, and that it conforms to the static semantic rules of the language. An implementation may also perform type checks at the time of analyzing a design unit, provided that the type checks performed can be decided at time of analysis of the design unit.

If any error is detected during analysis, the design unit shall not be stored in a design library. If analysis reveals no errors, the analyzed design unit may be stored in a design library. The selection of the design library and the form in which the analyzed design unit is stored shall be implementation dependent.

A design file shall be analyzed by analyzing each of the design units in the design file in the order of their occurrence in the design file. The effect of analyzing a design file shall be the same as the effect of independently analyzing the design units in the file one at a time in the order of their occurrence in the design file.

During analysis of a design unit, the label of the design unit shall be considered to be declared in the equivalent package declaration of the design library into which the analyzed design unit is to be stored. The scope of labels of analyzed design units previously stored in the design library and the scope of labels of subordinate design libraries of the deisgn library shall extend over the design unit being analyzed.

During analysis of a design unit, there shall be a root declarative region immediately within which the equivalent packages of the following root design libraries shall be considered to be declared:

a)The predefined root library `rosetta`
b)The ultimate parent library of the design library into which the analyzed design unit is stored
c)Each root library whose label occurs in a library name in the context clause of the design unit
d)For a design unit that is a body declaration: each root library whose label occurs in a library name in the context clause of the corresponding interface declaration

The root declarative region shall contain an implicit use clause of the form:

```
use rosetta.lang;
```

Corresponding to an analyzed design unit there shall be a value of type `rosetta.lang.reflect.abstract_syntax.design_unit` that represents the abstract syntax tree of the design unit.

For a label that is declared by a single design unit, an implementation shall yield the abstract syntax tree of the analyzed design unit as the result of interpretation of a function application of the function `rosetta.lang.reflect.abstract_syntax.get_design_unit` to a fully qualified name that refers to the declared label.

For a label that is declared by a pair of design units containing an interface declaration and a body declaration, an implementation shall yield the pair of abstract syntax trees of the analyzed design units as the result of interpretation of a function application of the function `rosetta.lang.reflect.abstract_syntax.get_design_unit` to a fully qualified name that refers to the declared label.

An implementation may store abstract syntax trees explicitly or may generate an abstract syntax tree when required.

NOTES:

1 — As a consequence of the above rules and of the scope and visibility rules, a design unit when analyzed has direct visibility of labels of all root libraries declared in the root declarative region, of all parent libraries of the design library into which it is stored, and of all other design units previously stored in the design library into which it is stored.

2 — The static semantic rules governing correspondence of interface and body declarations require that an interface declaration and a corresponding body declaration be stored in the same design library.

20.1.1 Order of analysis

A design unit that references the label of a second design unit is said to *depend on* the second design unit. Furthermore, a design unit that is a body declaration is said to depend on the corresponding interface declaration. A design unit must be analyzed without error before any design unit that depends on it is analyzed.

If a previously analyzed design unit in a design library is reanalyzed and stored the design library, replacing the previously stored version, all analyzed design units that depend on the first design unit shall become obsolete. If they continue to form part of the Rosetta specification, they shall be reanalyzed and replace their previously stored versions.

20.2 Elaboration of a specification

Elaboration of a specification shall involve construction of a language specification from a collection of analyzed Rosetta design units. An implementation may perform type checks at the time of elaborating a specification, provided that the type checks performed can be decided at time of elaborating the specification.

An implementation shall select a *root design name* from which to start elaboration. The manner in which the root design name is selected shall be implementation defined.

The result of elaborating a root design name shall be equivalent to the specification resulting from application of the function `rosetta.lang.reflect.elaboration.elaborate_root_design_name` to the root design name. The form in which the result of elaboration is represented is implementation defined.

The function `elaborate_root_design_name` shall yield a Rosetta specification by determining the closure of all design units referenced by the design unit denoted by the root design name, simplifying those design units, expanding the names in the simplified design units, and applying transformations defined by the domains of the design units. Simplification is described in 20.2.1, name expansion is described in 20.2.2, and transformation is described in 20.2.3.

<u>**To do:**</u> There will probably be a fourth that performs "flattening" of facets to produce the final monolithic anonymous facet value that is "the coalgebra".

Elaboration:

```
elaborate_root_design_name ( n :: label_list ) ...
```

NOTE — The function `elaborate_root_design_name` determines the root library referenced by the name formed by the list of labels, `n`, and yields the result of elaborating transitively from that root library. It gets the closure of all referenced libraries from the design units in the library denoted by `n`. It then determines the simplified form of those libraries and expands the names in the simplified forms. The function yields ..

```
get_root_design_library ( l :: library_name ) :: design_library is
   constant;
```

NOTE — The function `get_root_design_library` shall yield an AST for the root design library denoted by the name `l`. The manner in which the AST is determined shall be implementation defined.

```
referenced_libraries_from_target_library_name ( n :: library_name )
   :: set ( library_name ) is ...
```


NOTE — The function `referenced_libraries_from_target_library_name` yields the set of names of libraries that are directly or indirectly referenced from design units in the library denoted by `n`.

```
closure_of_referenced_libraries ( n :: library_name;
                                  s :: optional_library_name_list )
   :: optional_library_name_list is ...
```

NOTE — The function closure_of_referenced_libraries yields a list of names of libraries that are directly or indirectly referenced from design units in the library denoted by `n`, appended to the list `s`, with no duplicates. If the name n is already in the list `s`, the function yields `s` unchanged. Otherwise, the function determines a list of names of libraries directly referenced by design units in the library denoted by `n`. For each name in that list, the function recursively applies itself, thus determining the names of libraries indirectly referenced.

```
libraries_referenced_from_design_library ( d :: design_library )
   :: optional_library_name_list is ...
```

NOTE — The function `libraries_referenced_from_design_library` yields a list of library names referenced in library clauses of design units in the design library `d` and the sublibraries of `d`.

```
libraries_referenced_from_design_unit ( n :: design_unit )
   :: optional_library_name_list is ...
```

NOTE — The function `libraries_referenced_from_design_unit` yields a list of library names referenced in library clauses of the design unit `n`.

```
unique_libraries ( l :: optional_library_name_list )
   :: optional_library_name_list ...
```

NOTE — The function `unique_libraries` yields a list of library names that contains exactly the same set of names that are in `l` and that contains no duplicates. Two name are duplicatse if they both denotes the same design library.

```
library_name_in_list ( n :: library_name; l :: optional_library_name_list )
   :: boolean is ...
```

NOTE — The function `library_name_in_list` tests whetherthe library denoted by `n` is denoted by a library name in the list `l`.

```
identical_libraries ( l1, l2 : library_name ) :: boolean is
   constant;
```

NOTE — The function `identical_libraries` tests whether the library names `l1` and `l2` denote the same design library.

20.2.1 Simplification

A simplified Rosetta specification is a Rosetta specification in which all abstract syntax constructs are members of `rosetta.lang.reflect.simplification.simplified_nonterminal`. Simplification of a Rosetta specification shall yield an equivalent simplified Rosetta specification in which those abstract syntax constructs that are not of a simplified form are replaced by the equivalent simplified form. The result of simplifying a design library shall be equivalent to the simplified design library resulting from application of the function `rosetta.lang.reflect.simplification.simplify_design_library` to the design library.

```
simplified_nonterminal :: subtype ( nonterminal ) is ...
```

NOTE — The type `simplified_nonterminal` is the union of each of the simplified AST types.

20.2.2 Name expansion

A resolved Rosetta specification is a simplified Rosetta specification in which all abstract syntax constructs are members of `rosetta.lang.reflect.name_expansion.resolved_nonterminal`. Name expansion of a simplified Rosetta specification shall yield an equivalent resolved Rosetta specification in which those abstract syntax constructs that are not of a resolved form are replaced by the equivalent resolved form. The result of name expansion of a simplified design library shall be equivalent to the resolved design library resulting from application of the function `rosetta.lang.reflect.name_expansion.resolve_names_in_design_library` to the simplified design library.

```
resolved_nonterminal :: subtype ( simplified_nonterminal ) is ...
```

NOTE — The type `resolved_nonterminal` is the union of each of the AST types in which name references have been expanded to resolved names.

20.2.2.1 Name expansion contexts

A context for name expansion is described by a value of the type `rosetta.lang.name_expansion.context`.

```
context :: ...
```

NOTE — A context for name expansion is a list of context regions. Each element describes a declarative region nested within the declarative region described by the subsequent element, except for the final element which describes the root declarative region for elaboration.

```
context_region :: ...
```

NOTE — A context region is an pair, the first element being a declarative region AST value, and the second element being a set of context pair.

```
declarative_region :: ...
```

NOTE — The type `declarative_region` is the union of those resolved AST types that may declare items or terms or define parameters.

```
context_pair :: ...
```

NOTE — A context pair describes a label that is directly or potentially visible within a declarative region. For a label that is directly visible, the context name will be a singleton list containing the label. For a label that is potentially visible, the context name will be a label list consisting of the label prepended with the name of the package that appeared in a use clause to make the label potentially visible.

```
make_local_binding_context_pair ...
```

NOTE — Yields a context pair corresponding to a directly visible label for the given label value.

```
make_local_context_region ...
```

NOTE — Yields a context region in which the declarer is the given AST value and the context pairs are formed from the result applying `make_local_binding_context_pair` to each label of the given label list.

```
label_in_region ...
```

NOTE — Yields true if the any region of the given context contains a context pair that describes a binding for the given label.

```
label_in_context ...
```

NOTE — Yields true if the given context region contains a context pair that describes a binding for the given label.

```
get_name_from_region ...
```

NOTE — Yields the binding contained in a context pair of the given region that describes a binding for the given label.

```
add_labels_to_head_region ...
```

NOTE — Yields a context which is equivalent to the given context except for the context pairs of the head region being the union of the context pairs of the head region of the given context with the results of applying `make_local_binding_context_pair` to each label of the given label list.

```
add_used_names_to_head_region ...
```

NOTE — Yields a context which is equivalent to the given context except for the context pairs of the head region being the union of the context pairs of the head region of the given context with the context pairs that describe the labels that are directly visible by virtue of the association of a use clause in which the given names appear to the declarative region corresponding to the head of the context.

20.2.2.2 Resolution of references

The value of the type `rosetta.lang.reflect.name_expansion.resolved_reference` equivalent to a given label list name `n` occuring within a given declarative region described by context `c` is the result of application of the function `rosetta.lang.reflect.name_expansion.get_resolved_reference_internal` with parameters `c`, `0`, and `n`.

```
get_resolved_reference_internal :: ...
```

NOTE — If the head region of the given context does not contain a binding for the head of the given name, then the result is the recursive application of `get_resolved_reference_internal` where the parameters are the tail of the given context, the sum of the given parameter `u` with `1`, and the given name. If the head of the given context contains a binding for the head of the given name and the binding is not a singleton list then the result is the recursive application of `get_resolved_reference_internal` where the parameters are the tail of the given context, the sum of the given parameter `u` with `1`, and the name formed by prepending the binding for the head of the given name to the tail of the given name. If the head of the given context contains a binding for the head of the given name and the binding is a singleton label list then a resolved reference is produced such that

—The up levels component is the value given for parameter u;
—The top level declarer component is the declarer of the head of the given context;
—The labels component is the label list formed by prepending the binding for head of the given name to tail of the given name;
—The item component is the result of application of `get_item_from_declarative_region` where the parameters are the labels compenent described above and the declarer of the head of the given context;
—The declared type component is the result of application of `get_declared_type` to the item component described above;
—The declared value component is the result of application of `get_declared_value` to the item component described above.

```
get_item_from_declarative_region :: ...
```

NOTE — Yields a referable item according to the subtype of the given declarative region

```
get_item_from_let_expression :: ...
```

NOTE — If the given name is a singleton label list, then the result is the let binding of the given let expression that declares the same label as the given label. Otherwise, the result is the application of `get_item_from_declarative_region` where the first parameter is the tail of the given name and the second parameter is the application of `get_declared_value` to the let binding of the given let expression that declares the same label as the head of the given name.


```
get_item_from_function_type_formation :: ...
```

NOTE — Yields the parameter definition of the signature of the given function type formation that declares the same parameter label as the head of the given label list.

```
get_item_from_anonymous_function :: ...
```

NOTE — Yields the parameter definition of the signature of the given anonymous that declares the same parameter label as the head of the given label list.

```
get_item_from_constructed_type_declaration :: ...
```

NOTE — Yields the parameter definition of the construct type definition that declares the same parameter label as the head of the given label list.

```
get_item_from_anonymous_facet :: ...
```

NOTE — If the given name is a singleton label list, then the result is the parameter, declaration, or term of the given anonymous facet that declares the same label as the given label. Otherwise, the result is the application of `get_item_from_declarative_region` where the first parameter is the tail of the given name and the second parameter is the application of `get_declared_value` to the parameter, declaration, or term of the given anonymous facet that declares the same label as the head of the given name.

```
get_item_from_domain_declaration :: ...
```

NOTE — If the given name is a singleton label list, then the result is the parameter, declaration, or term of the given domain declaration that declares the same label as the given label. Otherwise, the result is the application of `get_item_from_declarative_region` where the first parameter is the tail of the given name and the second parameter is the application of `get_declared_value` to the parameter, declaration, or term of the given domain declaration that declares the same label as the head of the given name.

```
referable_item_from_declaration :: ...
```

NOTE — The type `referable_item_from_declaration` is the union of those resolved AST types that may declare items.

```
referable_items_in_declaration :: ...
```

NOTE — Yields a sequence of values of type `referable_item_from_declaration` corresponding to the items declared by the given resolved declaration. For resolved declarations that are resolved variable simplification and property clause values, the declarations of variable simplification and property clause is yielded. For a resolved declaration that is a resolved constructed type declaration, a singleton list containing the resolved constructed type declaration is yielded. For a resolved declaration that is a resolved domain declaration, a singleton list containing the resolved domain declaration is yielded.

```
is_matching_item :: ...
```

NOTE — Yields true if the given referable item is a declaration, term, or parameter that declares the given label.

```
get_declared_type :: ...
```

NOTE —

```
get_declared_value :: ...
```

NOTE —

```
get_value_from_equality_term :: ...
```

NOTE — If the given term is equivalent to a term produced by simplification of the simple term `rosetta.lang.prel-`

`ude.universal_equals ( l , v )` for some expression `v`, then `v` is yeilded. Otherwise the result is _|_.

To do: Tie up loose ends in code and comments for get_declared_type and get_declared_value.

20.2.3 Transformation

A transformed Rosetta specification is a resolved Rosetta specification in which all abstract syntax constructs are members of `rosetta.lang.reflect.transformation.transformed_nonterminal`. Transformation of a resolved Rosetta specification shall yield an equivalent transformed Rosetta specification in which those abstract syntax constructs that are not of a transformed form are replaced by the equivalent transformed form. The result of transformation of a resolved design library shall be equivalent to the transformed design library resulting from application of the function `rosetta.lang.reflect.transformation.transform_design_library` to the resolved design library.

20.3 Interpretation of a specification

Interpretation of a specification shall involve a computation upon the value denoted by the result of elaboration of the specification. The computation to be performed is implemenation dependent. However, this clause defines a computation, consistency verification, that an implementation may perform.

20.3.1 Consistency checking

A specification is consistent if and only iff

- it conforms to the type rules of the language
-

To do: Define interpretation of terms.

21. The package rosetta.lang.prelude

The library `rosetta.lang` shall contain a predefined package named `prelude`. The domain of the package shall be `rosetta.lang.null`. The package shall contain declarations of items as specified in this clause. The package shall not contain declarations of any other items.

Except as otherwise specified, all functions declared in `rosetta.lang.prelude` shall be strict; that is, if any argument is _|_, the function shall yield _|_.

21.1 Universal types

The package `rosetta.lang.prelude` shall contain the following declarations of universal types:

```
// universal type is not formally defined

universal :: type is constant;

element :: type is boolean + number + character;
```

The type `universal` shall denote the set of all denotable values (see 5.1).

The type `element` shall denote the set of all Rosetta values that are not decomposable into component values. The value of `element` shall be the union of

—All Boolean values
—All numbers
—All characters

21.1.1 Universal functions

The package `rosetta.lang.prelude` shall contain the following declarations of functions on universal types:

```
universal_equals ( L, R :: universal ) :: boolean is constant;

universal_not_equals ( L, R :: universal ) :: boolean is
  if L = R then
    false
  else
    true
  end if;

universal_equivalent :: <* ( L, R :: universal ) :: boolean *> is
  universal_equals;
```

The function `universal_equals` shall test whether its two arguments are the same value. If so, the function shall yield the value `true`; otherwise the function shall yield the value `false`. The function `universal_not_equals` shall test whether the two arguments are the same value. If so, the function shall yield the value `false`; otherwise the function shall yield the value `true`. The function `universal_equivalent` shall be the same function as `universal_equals`.

21.2 Type functions

The package `rosetta.lang.prelude` shall contain the following declarations of functions on types:

```
type_assertion ( L :: universal; R :: type ) :: R is
  if L in R then
    L
  else
    _|_
  end if;

// type_member function is not formally defined

type_member ( L :: universal; R :: type) :: boolean is constant;

type_proper_subtype ( L, R :: type ) :: boolean is
  forall ( V :: L | V in R ) and L /= R;

type_subtype ( L, R :: type ) :: boolean is
  L < R or L = R;

type_proper_supertype ( L, R :: type ) :: boolean is
  R < L;

type_supertype ( L, R :: type ) :: boolean is
  R < L or R = L;

type_union [ T :: type ] ( L, R :: subtype ( T ) ) :: subtype ( T ) is
  sel ( X :: T | X in L or X in R );

type_intersection [ T :: type ] ( L, R :: subtype ( T ) ) :: subtype ( T ) is
  sel ( X :: T | X in L and X in R );

type_difference [ T :: type ] ( L, R :: subtype ( T ) ) :: subtype ( T ) is
  sel ( X :: T | X in L and not ( X in R ) );
```

The function `type_assertion` shall test whether its left argument is a member of the type denoted by its right argument. If so, the function shall yield the first argument value; otherwise it shall yield _|_. The function `type_member` shall test whether its first argument is a member of the type denoted by its second argument. If so, the function shall yield the value `true`; otherwise it shall yield the value `false`.

The function `type_proper_subtype` shall test whether its first argument is a proper subtype of the type denoted by its second argument. If so, the function shall yield the value `true`; otherwise it shall yield the value `false`. The function `type_subtype` shall test whether its first argument is a subtype of the type denoted by its second argument. If so, the function shall yield the value `true`; otherwise it shall yield the value `false`. The function `type_proper_supertype` shall test whether its first argument is a proper supertype of the type denoted by its second argument. If so, the function shall yield the value `true`; otherwise it shall yield the value `false`. The function `type_supertype` shall test whether its first argument is a supertype of the type denoted by its second argument. If so, the function shall yield the value `true`; otherwise it shall yield the value `false`.

The function `type_union` shall yield the union of its two arguments, that is, the set of values that are members of the first argument or of the second argument. The function `type_intersection` shall yield the intersection of its two arguments, that is, the set of values that are members of both the first argument and the second argument. The function `type_difference` shall yield the relative complement of its two arguments, that is, the set of values that are members of the first argument and not of the second argument.

21.3 The Boolean type

The package `rosetta.lang.prelude` shall contain the following declaration of the Boolean types:

```
boolean :: type is enumeration (false, true);
```

21.3.1 Boolean functions

The package `rosetta.lang.prelude` shall contain the following declarations of functions on the Boolean type:

```
boolean_to_bit ( R :: boolean ) :: bit is
  case R is
    {false} -> 0
  | {true}  -> 1
  end case;

boolean_not ( R :: boolean ) :: boolean is
  if R then false else true end if;

boolean_and ( L, R :: boolean ) :: boolean is
  if L then R else false end if;

boolean_or ( L, R :: boolean ) :: boolean is
  if L then true else R end if;

boolean nand ( L, R :: boolean ) :: boolean is
  not ( L and R );

boolean_nor ( L, R :: boolean ) :: boolean is
  not ( L or R );

boolean_xor ( L, R :: boolean ) :: boolean is
  ( L or R ) and not ( L and R );

boolean_xnor ( L, R :: boolean ) :: boolean is
  not ( L xor R );

boolean_implies ( L, R :: boolean ) :: boolean is
  not L or R;

boolean_implied_by ( L, R :: boolean ) :: boolean is
  R => L;
```

The function `boolean_to_bit` shall yield the bit value 0 when applied to the Boolean value `false`, and shall yield the bit value 1 when applied to the Boolean value `true`.

The function `boolean_not` shall yield the Boolean negation of its argument. The function `boolean_and` shall yield the Boolean conjunction of its arguments. The function `boolean_or` shall yield the Boolean disjunction of its arguments. The function `boolean_nand` shall yield the Boolean negated conjunction of its arguments. The function `boolean_nor` shall yield the Boolean negated disjunction of its arguments. The function `boolean_xor` shall yield the Boolean exclusive disjunction of its arguments. The function `boolean_xnor` shall yield the Boolean negated exclusive disjunction of its arguments. The function `boolean_implies` shall yield the Boolean implication of the second argument by the first argument. The function `boolean_implied_by` shall yield the Boolean implication of the first argument by the second argument.

The functions `boolean_and`, `boolean_or`, `boolean_nand`, `boolean_nor`, `boolean_implies` and `boolean_implied_by` shall be non-strict; that is, if the value of one argument is sufficient to determine a proper result, the function shall yield the proper result regardless of whether the other argument is _|_.

21.4 Numeric types

The package `rosetta.lang.prelude` shall contain the following declarations of numeric types:

```
number :: type is complex_with_infinity;

// complex type is not formally defined

complex :: subtype ( number ) is constant;

complex_with_infinity :: type is
  complex + { -∞ , +∞ };

real :: subtype ( complex ) is
  sel ( z :: complex | im(z) = 0 );

real_with_infinity :: subtype ( complex_with_infinity ) is
  real + { -∞ , +∞ };

posreal :: subtype ( real ) is
  sel ( x :: real | x > 0 );

posreal_with_infinity :: subtype ( real_with_infinity ) is
  posreal + { +∞ };

negreal :: subtype ( real ) is
  sel ( x :: real | x < 0 );

negreal_with_infinity :: subtype ( real_with_infinity ) is
  negreal + { -∞ };

imaginary :: subtype ( complex ) is
  sel ( z :: complex | re(z) = 0 );

imaginary_with_infinity :: subtype (complex_with_infinity  ) is
  imaginary + { -∞ , +∞ };

rational :: subtype ( real ) is
  sel ( x :: real | exists ( y, z :: integer | x = y / z and z /= 0 ) );

rational_with_infinity :: subtype ( real_with_infinity ) is
  rational + { -∞ , +∞ };

// integer type is not formally defined
```

```
integer :: subtype ( rational ) is;

integer_with_infinity :: subtype ( rational_with_infinity ) is
  integer + { -∞ , +∞ };

natural :: subtype ( integer ) is
  sel ( x :: integer | x >= 0 );

natural_with_infinity :: subtype ( integer_with_infinity ) is
  natural + { +∞ };

posint :: subtype ( natural ) is
  sel ( x :: integer | x > 0 );

posint_with_infinity :: subtype ( natural_with_infinity ) is
  posint + { +∞ };

negint :: subtype ( integer ) is
  sel ( x :: integer | x < 0 );

negint_with_infinity :: subtype ( integer_with_infinity ) is
  negint + { -∞ };

bit :: subtype ( natural ) is {0, 1};
```

The type `number` shall denote the set of Rosetta number values, and shall be the same as the type `complex_with_infinity`.

The type `complex` shall denote the set of mathematical complex numbers.

NOTE — Literal values of type `complex` may be written using real literals, either in Cartesian form (for example, `3 + 2*j`) or in polar form (for example, `2*exp(pi/2*j)`).

The type `real` shall denote the set of mathematical real numbers, namely, those complex numbers whose imaginary part is 0. The type `posreal` shall denote the subset of real numbers that are strictly greater than 0. The type `negreal` shall denote the subset of real numbers that are strictly less than 0.

The type `imaginary` shall denote the set of mathematical imaginary numbers, namely, those complex numbers whose real part is 0.

NOTE — Literal values of type `imaginary` may be written by multiplying a real literal by the constant `j` (for example, `2*j`).

The type `rational` shall denote the set of mathematical rational numbers, namely, those real numbers that can be expressed as the quotient of two integers with the divisor being non-zero.

The type `integer` shall denote the set of mathematical integers. The type `natural` shall denote the subset of integers that are greater than or equal to 0. The type `posint` shall denote the subset of integers that are strictly greater than 0. The type `negint` shall denote the subset of integers that are strictly less than 0.

The type `bit` shall denote the subset of natural numbers 0 and 1.

Corresponding to the types `complex`, `real`, `imaginary`, `rational` and `integer`, the types `complex_with_infinity`, `real_with_infinity`, `imaginary_with_infinity`, `rational_with_infinity` and `integer_with_infinity`, respectively, shall include all of the values of the type to which they correspond and the infinite values $-\infty$ and $+\infty$. Corresponding to the types posreal, natural, and posint, the types posreal_with_infinity, natural_with_infinity, and posint_with_infinity, respectively, shall include all of the values of the type to which they correspond and the infinite value $+\infty$. Corresponding to the types negreal and negint, the types negreal_with_infinity and negint_with_infinity, respectively, shall include all of the values of the type to which they correspond and the infinite value $-\infty$.

21.4.1 Numeric constants

The package rosetta.lang.prelude shall contain the following declarations of numeric constants:

```
j :: imaginary is constant
     where j * j = -1;

e :: real is exp ( 1 );

// pi is not formally defined

π :: real is constant;
pi :: real is π;

infinity :: number is ∞;
```

The constant j shall denote the imaginary unit value.

The constant e shall denote the real value of the mathematical constant e.

The constants pi and π shall denote the real value of the mathematical constant π.

The constant infinity shall denote the positive infinite number value.

NOTE — The negative infinite value can be denoted using the expression -infinity.

21.4.2 Numeric functions

The package rosetta.lang.prelude shall contain the following declarations of functions on the numeric types:

```
// Include numeric function declarations here...
```

The function number_identity shall yield its argument. The unary function number_negation shall yield the numerical negation of its argument.

The function complex_addition shall yield the first argument added to the second argument. The function complex_subtraction shall yield the second argument subtracted from the first argument. The function complex_multiplication shall yield the first argument multiplied by the second argument. The function complex_division shall yield the first argument divided by the second argument, provided the mathematical result of the division is defined; otherwise the function shall yield _|_. The function complex_exponentiation shall yield the first argument raised to the power of the second argument, provided the mathematical result of the power is defined; otherwise the function shall yield _|_.

The function real_min shall yield the lesser of its two arguments. The function real_max shall yield the greater of its two arguments.

The function real_less_than shall yield true if the first argument is strictly less than the second argument; otherwise it shall yield `false`. The function `real_less_than_or_equals` shall yield `true` if the first argument is less than or equal to the second argument; otherwise it shall yield `false`. The function `real_greater_than` shall yield `true` if the first argument is strictly greater than the second argument; otherwise it shall yield `false`. The function `real_greater_than_or_equals` shall yield `true` if the first argument is greater or equal to the second argument; otherwise it shall yield `false`.

The function `imaginary_min` shall yield the lesser of its two arguments. The function `imaginary_max` shall yield the greater of its two arguments.

The function `imaginary_less_than` shall yield `true` if the first argument is strictly less than the second argument; otherwise it shall yield `false`. The function `imaginary_less_than_or_equals` shall yield `true` if the first argument is less than or equal to the second argument; otherwise it shall yield `false`. The function `imaginary_greater_than` shall yield `true` if the first argument is strictly greater than the second argument; otherwise it shall yield `false`. The function `imaginary_greater_than_or_equals` shall yield `true` if the first argument is greater or equal to the second argument; otherwise it shall yield `false`.

The function `re` shall yield the real part of its argument. The function `im` shall yield the imaginary part of its argument. The function `abs` shall yield the absolute value (that is, the modulus) of its argument. The function `arg` shall yield the argument (that is, the angle between the positive real axis and the vector represented by the complex number) of its argument. The function `conj` shall yield the complex conjugate of its argument.

The function `sin` shall yield the circular sine of its argument. The function `cos` shall yield the circular cosine of its argument. The function `tan` shall yield the circular tangent of its argument, provided the mathematical result of the division is defined; otherwise the function shall yield _|_. The function `arcsin` shall yield the inverse circular sine of its argument, provided the mathematical result of the division is defined; otherwise the function shall yield _|_. The function `arccos` shall yield the inverse circular cosine of its argument, provided the mathematical result of the division is defined; otherwise the function shall yield _|_. The function `arctan` shall yield the inverse circular tangent of its argument.

The function `sinh` shall yield the hyperbolic sine of its argument. The function `cosh` shall yield the hyperbolic cosine of its argument. The function `tanh` shall yield the hyperbolic tangent of its argument. The function `arcsinh` shall yield the inverse hyperbolic sine of its argument, provided the mathematical result of the division is defined; otherwise the function shall yield _|_. The function `arccosh` shall yield the inverse hyperbolic cosine of its argument, provided the mathematical result of the division is defined; otherwise the function shall yield _|_. The function `arctanh` shall yield the inverse hyperbolic tangent of its argument, provided the mathematical result of the division is defined; otherwise the function shall yield _|_.

The function `exp` shall yield the exponential function of its argument. The function `log` shall yield the natural logarithm of its argument, provided the mathematical result of the division is defined; otherwise the function shall yield _|_. The function `log10` shall yield the logarithm to base 10 of its argument, provided the mathematical result of the division is defined; otherwise the function shall yield _|_. The function `log2` shall yield the logarithm to base 2 of its argument, provided the mathematical result of the division is defined; otherwise the function shall yield _|_.

The function `sqrt` shall yield the square root of its argument.

The function `floor` shall yield the largest integer that is less than or equal to the argument of the function. The function `cieling` shall yield the smallest integer that is greater than or equal to the argument of the function. The function `trunc` shall yield the integer of greatest absolute value that is of the same sign as the argument of the function and whose absolute value is less than or equal to the absolute value of the argument of the function. The function `round` shall yield the nearest integer to the argument of the function. In the case of an argument that is an odd multiple of 0.5, the result shall be the integer further from zero. The function `sign` shall yield –1 when applied to a negative argument value, 0 when applied to 0, and +1 when applied to a positive argument value.

The function `num` shall yield the numerator of its argument, and the function `den` shall yield the denominator of its argument. For a given rational number, the result of num shall be be of the same sign as the number, the result of den shall be positive, and the greatest common divisor of the two results shall be 1. The result of den(0) shall be 1.

The function integer_division shall yield the integer quotient of the first argument divided by the second argument, truncated towards zero, provided the mathematical result of the division is defined; otherwise the function shall yield _|_. The function integer_modulus shall yield the integer modulus of the first argument divided by the second argument, provided the mathematical result of the division is defined; otherwise the function shall yield _|_. The result shall have the same sign as the value of the second argument and shall have absolute value less than the absolute value of the second argument. The function rem shall yield the integer remainder of the first argument divided by the second argument, provided the mathematical result of the division is defined; otherwise the function shall yield _|_. The result shall have the same sign as the value of the first argument and shall have absolute value less than the absolute value of the second argument.

21.4.3 Bit functions

The package rosetta.lang.prelude shall contain the following declarations of functions on the bit type:

```
bit_to_boolean ( R :: bit ) :: boolean is
  case R is
    {0} -> false
  | {1} -> true
  end case;

bit_not ( R :: bit ) :: bit is
  if R = 1 then 0 else 1 end if;

bit_and ( L, R :: bit ) :: bit is
  if L = 1 then R else 0 end if;

bit_or ( L, R :: bit ) :: bit is
  if L = 1 then 1 else R end if;

bit nand ( L, R :: bit ) :: bit is
  not ( L and R );

bit_nor ( L, R :: bit ) :: bit is
  not ( L or R );

bit_xor ( L, R :: bit ) :: bit is
  ( L or R ) and not ( L and R );

bit_xnor ( L, R :: bit ) :: bit is
  not ( L xor R );

bit_implies ( L, R :: bit ) :: bit is
  not L or R;

bit_implied_by ( L, R :: bit ) :: bit is
  R => L;
```

The function bit_to_boolean shall yield the Boolean value false when applied to the bit value 0, and shall yield the Boolean value true when applied to the Boolean value 1.

The function bit_not shall yield the logical negation of its argument. The function bit_and shall yield the logical conjunction of its arguments. The function `bit_or` shall yield the logical disjunction of its arguments. The function `bit_nand` shall yield the logical negated conjunction of its arguments. The function `bit_nor` shall yield the logical negated disjunction of its arguments. The function `bit_xor` shall yield the logical exclusive disjunction of its arguments. The function `bit_xnor` shall yield the logical negated exclusive disjunction of its arguments. The function `bit_implies` shall yield the logical implication of the second argument by the first argument. The function `bit_implied_by` shall yield the logical implication of the first argument by the second argument.

21.5 Character types

The package `rosetta.lang.prelude` shall contain the following declarations of the character type:

```
character :: type is { 'U-00000000' ,.. 'U-FFFFFFFF' };
```

The type `character` shall denote the set of all UTF-32 characters, as defined in The Unicode Standard, Version 3.2.

21.5.1 Character functions

The package `rosetta.lang.prelude` shall contain the following declarations of functions on the character type:

```
character_less_than ( L, R :: character ) :: boolean is
  unicode.ord ( L ) < unicode.ord ( R );

character_less_than_or_equals ( L, R :: character ) :: boolean is
  unicode.ord ( L ) =< unicode.ord ( R );

character_greater_than ( L, R :: character ) :: boolean is
  unicode.ord ( L ) > unicode.ord ( R );

character_greater_than_or_equals ( L, R :: character ) :: boolean is
  unicode.ord ( L ) >= unicode.ord ( R );
```

The function `character_less_than` shall yield `true` if the Unicode code value of the first argument is strictly less than the Unicode code value of the second argument; otherwise it shall yield `false`. The function `character_less_than_or_equals` shall yield `true` if the Unicode code value of the first argument is less than or equal to the Unicode code value of the second argument; otherwise it shall yield `false`. The function `character_greater_than` shall yield `true` if the Unicode code value of the first argument is strictly greater than the Unicode code value of the second argument; otherwise it shall yield `false`. The function `character_greater_than_or_equals` shall yield `true` if the Unicode code value of the first argument is greater or equal to the Unicode code value of the second argument; otherwise it shall yield `false`.

NOTE — Further functions upon characters are defined in the package `rosetta.lang.unicode`.

21.6 Function types

The package `rosetta.lang.prelude` shall contain the following declarations of function types:

```
function ( D, R :: type ) :: type is <* ( P :: D ) :: R *>;

predicate ( D :: type ) :: type is <* ( P :: D ) :: boolean *>;
```

The type constructor `function` shall denote the set of all Rosetta function values. The type constructor shall yield the set of functions whose domain is the first argument of the type constructor and whose range is the second argument of the type constructor.

The type constructor `predicate` shall denote the set of all Rosetta function values that return a result of type `boolean`. The type constructor shall yield the set of Boolean-valued functions whose domain is the argument of the type constructor.

21.6.1 Higher-order functions

The package `rosetta.lang.prelude` shall contain the following declarations of higher-order functions, that is, functions that operate on values of function types:

```
function_proper_contained_in ( L, R :: function ) :: boolean is
  dom ( L ) =< dom ( R )
  and forall ( X :: dom ( L ) | L ( X ) = R ( X ) )
  and L /= R;

function_contained_in ( L, R :: function ) :: boolean is
  L < R or L = R;

function_proper_contains ( L, R :: function ) :: boolean is
  R < L;

function_contains ( L, R :: function ) :: boolean is
  R < L or R = L;

function_member ( L :: universal; R :: function ) :: boolean is
  exists ( X :: dom ( R ) | L in R ( X ) );

function_composition
  [ T1, T2, T3 :: type ]
  ( L :: function ( T2, T3 ); R :: function ( T1, T2) )
  :: function ( T1, T3 ) is
  <* ( X :: T1) ) :: T3 is L ( R ( X ) ) *>;

// dom function is not formally defined

dom ( F :: function ) :: type is constant;

// ran function is not formally defined

ran ( F :: function ) :: type is constant;

// fix function is not formally defined

fix ( F :: function ) :: universal is constant;

minf [ D :: type; R :: dom ( __=<__ ) ]
      ( F :: function ( D, R ) ) :: R is
  choose ( sel ( X :: ran ( F ) | forall ( Y :: ran ( F ) | X =< Y ) ) );

maxf [ D :: type; R :: dom ( __=<__ ) ]
      ( F :: function ( D, R ) ) :: R is
  choose ( sel ( X :: ran ( F ) | forall ( Y :: ran ( F ) | X >= Y ) ) );

// sel function is not formally defined
```

```
sel [ D :: type ]
    ( P :: predicate ( D ) ) :: set ( D ) is constant;

// Forall and exists need to be modified to deal with
// functions of more than one parameter.  See bug #119 and bug #53.

forall ( P :: predicate ) :: boolean is
  ran ( P ) = { true };

exists ( P :: predicate ) :: boolean is
  true in ran ( P );
```

The function `function_proper_contained_in` shall yield `true` if the first argument is strictly contained in the second argument; otherwise it shall yield `false`. The function `function_contained_in` shall yield `true` if the first argument is contained in or equal to the second argument; otherwise it shall yield `false`. The function `function_proper_contains` shall yield `true` if the first argument strictly contains the second argument; otherwise it shall yield `false`. The function `function_contains` shall yield `true` if the first argument contains or is equal to the second argument; otherwise it shall yield `false`.

NOTE — The function-containment functions applied to functions that denote polymorphic or dependent types are equivalent to the corresponding subtype and supertype functions. However, the function containment functions are also applicable to functions that do not denote types.

The function `function_composition` shall yield the function that is the composition of the first argument with the second argument. The domain of the composition shall be the domain of the second argument. For each value in the domain of the second argument, the result of applying the composition to the value shall be the result of applying the first argument to the result of applying the second argument to the value.

The function `dom` shall yield the set of values that are in the domain of the argument function, that is, values to which the argument function can be applied to yield a proper result. The function `ran` shall yield the set of values that are in the range of the argument function, that is, values yielded as results of the argument function. The function `fix` shall yield the function that is the least fixed point of the argument function.

The function `minf` shall yield the minimum value in the range of the argument function. The function `maxf` shall yield the maximum value in the range of the argument function.

The function `sel` shall yield the set of values in the domain of the argument for which the argument function yields the result `true`. The function `forall` shall yield `true` if the range of the argument function is the singleton set containing the value `true`; otherwise it shall yield `false`. The function `exists` shall yield true if the range of the argument function is contains the value `true`; otherwise it shall yield `false`.

21.7 Set types

A set shall be an unordered collection of values without duplicates. A set value shall be a member of the powerdomain of denotable values (see 5.1) The number of elements in a set shall be called the cardinality of the set. A set may be empty, in which case its cardinality is 0. A set may have an unbounded number of elements, in which case its cardinality is not defined.

The package `rosetta.lang.prelude` shall contain the following declarations of the set type:

```
// set function is not formally defined

set ( T :: type ) :: type is constant;
```


The type constructor `set` shall denote the set of all Rosetta set values. The type constructor shall yield the set of sets whose elements are members of the argument of the type constructor.

21.7.1 Set constants

The package `rosetta.lang.prelude` shall contain the following declarations of a set constant:

```
// empty_set constant is not formally defined

empty_set :: set is constant;
```

The constant `empty_set` shall denote the empty set.

21.7.2 Set functions

The package `rosetta.lang.prelude` shall contain the following declarations of functions on set types:

```
set_cardinality ( R :: set ) :: natural is
  if R = { } then
    0
  else
    1 + #( R - { set_choose ( R ) } )
  end if;

set_contents [ T :: type ] ( S :: set ( T ) ) :: set ( T ) is
  S;

// singleton_set function is not formally defined

singleton_set [ T :: type ] ( V :: T ) :: set ( T ) is constant;

range_set
  [ T :: subtype ( integer ) + subtype ( character ) ]
  ( L, R :: T ) :: set ( T ) is
  if L > R then
    { }
  elsif L = R then
    { L }
  else
    { L }
    + range_set (
        if T =< integer then
          L + 1
        else
          unicode.char ( unicode.ord ( L ) + 1 )
        end if,
        R )
  end if;

// set_choose function is not formally defined

set_choose [ T :: type ] ( S :: set ( T ) ) :: T is constant;

set_image
```

```
    [ D, R :: type ]
    ( F :: function ( D, R ); S :: set ( D ) ) :: set ( R ) is
    ran ( <* ( X :: S ) :: R is F ( X ) *> );

  set_filter
    [ D :: type ]
    ( P :: predicate ( D ); S :: set ( D ) ) :: set ( D ) is
    if S = { } then then
      { }
    else
      let ( V :: D be set_choose ( S ) ) in
        if P ( V ) then { V } else { } end if
        + set_filter ( P, S - { V } )
      end let
    end if;
```

The function `set_cardinality` shall yield the cardinality of its argument, if defined; otherwise the function shall yield _|_.

The function `set_contents` shall yield the set of values that are elements of its argument. Thus, the function shall be an identify function.

The function `singleton_set` shall yield the singleton set containing the value of the argument. The function `range_set` shall yield the set of values that are greater than or equal to the first argument and less than or equal to the second argument.

The function `choose` applied to a non-empty argument shall yield an arbirarily chosen member of the argument. The function `choose` applied to an empty argument shall yield _|_.

The function `set_image` shall yield the set of results of applying the first argument to members of the second argument. The function `set_filter` shall yield the set of elements of the second argument for which the second argument yields a result value of `true`.

NOTE — The subtype and supertype functions applicable to types are applicable to sets, since sets are a form of type. When applied to sets, the subtype and supertype functions are equivalent to subset and superset functions.

21.8 Multiset types

A multiset shall be an unordered collection of values with duplicates allowed. The number of elements in a multiset shall be called the cardinality of the multiset. A multiset may be empty, in which case its cardinality is 0. A multiset may have an unbounded number of elements, in which case its cardinality is not defined.

The package `rosetta.lang.prelude` shall contain the following declarations of the multiset type:

```
  // multiset function is not formally defined

  multiset ( T :: type ) :: type is constant;
```

The type constructor `multiset` shall denote the set of all Rosetta multiset values. The type constructor shall yield the set of multisets whose elements are members of the argument of the type constructor.

21.8.1 Multiset constants

The package `rosetta.lang.prelude` shall contain the following declarations of a multiset constant:

```
// empty_multiset constant is not formally defined

empty_multiset :: multiset is constant;
```

The constant `empty_multiset` shall denote the empty multiset.

21.8.2 Multiset functions

The package `rosetta.lang.prelude` shall contain the following declarations of functions on multiset types:

```
// multiset_union function is not formally defined

multiset_union ( L, R :: multiset ) :: multiset is constant

// multiset_intersection function is not formally defined

multiset_intersection ( L, R :: multiset ) :: multiset is constant

// multiset_difference function is not formally defined

multiset_difference ( L, R :: multiset ) :: multiset is constant

multiset_proper_submultiset ( L, R :: multiset ) :: boolean is
  forall ( X :: ~L | X # L =< X # R ) and L /= R;

multiset_submultiset ( L, R :: multiset ) :: boolean is
  L < R or L = R;

multiset_proper_supermultiset ( L, R :: multiset ) :: boolean is
  R < L;

multiset_supermultiset ( L, R :: multiset ) :: boolean is
  R < L or R = L;

// multiset_member function is not formally defined

multiset_member ( L :: universal; R :: multiset ) :: boolean is constant;

multiset_cardinality ( R :: multiset ) :: natural is
  if R = {* *} then
    0
  else
    1 + multiset_cardinality ( R - {* choose(R) *} )
  end if;

multiset_count_occurrences ( V :: universal; M :: multiset ) :: natural is
  if not ( V in M) then
    0
  else
    1 + multiset_count_occurrences ( V, M - {* V *} )
```

```
    end if;

  multset_contents [ T :: type ] ( M :: multiset ( T ) ) :: set( T ) is
    if M = {* *} then
      { }
    else
      let ( V :: T be choose ( M ) ) in
        { V } + multiset_contents ( M - {* V *} )
      end let
    end if;

  // single_value_multiset function is not formally defined

  single_value_multiset
    [ T :: type ]
    ( C :: natural; V :: T ) :: multiset ( T ) is constant;

  range_multiset
    [ T :: subtype ( integer ) + subtype ( character ) ]
    ( L, R :: T ) :: multiset ( T ) is
    if L > R then
      {* *}
    elsif L = R then
      {* L *}
    else
      {* L *}
      + range_multiset (
          if T =< integer then
            L + 1
          else
            unicode.char ( unicode.ord ( L ) + 1 )
          end if,
          R )
    end if;

  // multiset_choose function is not formally defined

  multiset_choose [ T :: type ] ( M :: multiset ( T ) ) :: T is constant;

  multiset_image
    [ D, R :: type ]
    ( F :: function ( D, R ); M :: multiset ( D ) ) :: multiset ( R ) is
    if M = {* *} then
      {* *}
    else
      let ( V :: D be multiset_choose ( M ) ) in
        {* F ( V ) *} + multiset_image ( F, M - {* V *} )
      end let
    end if;

  multiset_filter
    [ D :: type ]
    ( P :: predicate ( D ); M :: multiset ( D ) ) :: multiset ( D ) is
    if M = {* *} then
```

```
      {* *}
    else
      let ( V :: D be multiset_choose ( M ) ) in
        if P ( V ) then {* V *} else {* *} end if
        + multiset_filter ( P, M - {* V *} )
      end let
    end if;

  set2multiset
    [ T :: type ]
    ( S :: set ( T ) ) :: multiset ( T ) is
    if S = { } then
      {* *}
    else
      let ( V :: T be multiset_choose(S) ) in
        {* V *} + set2multiset ( S - { V } )
      end let
    end if;
```

The function `multiset_union` shall yield the union of its two arguments, that is, the multiset of values that are members of the first argument or of the second argument. The number of occurrences of a value in the union is the sum of the number of occurrences of the value in each of the arguments. The function `multiset_intersection` shall yield the intersection of its two arguments, that is, the multiset of values that are members of both the first argument and the second argument. The number of occurrences of a value in the intersection is the lesser of the number of occurrences of the value in each of the arguments. The function `multiset_difference` shall yield the relative complement of its two arguments, that is, the multiset of values that are members of the first argument with a greater number of occurrences than in the second argument. The number of occurrences of a value in the relative complement is the greater of zero and the number of occurrences of the value in the first argument less the number of occurrences of the value in the second argument.

The function `multiset_proper_submultiset` shall yield `true` if the first argument is strictly contained in the second argument; otherwise it shall yield `false`. The first argument is strictly contained in the second argument if and only if the two multisets are not equal and all values in the first argument have at least as many occurrences in the second argument as they have in the first argument. The function `multiset_submultiset` shall yield `true` if the first argument is strictly contained in or equal to the second argument; otherwise it shall yield `false`. The function `multiset_proper_supermultiset` shall yield `true` if the first argument strictly contains the second argument; otherwise it shall yield `false`. The function `multiset_supermultiset` shall yield `true` if the first argument strictly contains or is equal to the second argument; otherwise it shall yield `false`.

The function `multiset_member` shall yield `true` if the first argument occurs at least once in the second argument; otherwise it shall yield `false`.

The function `multiset_cardinality` shall yield the cardinality of the argument, if defined; otherwise the function shall yield _|_. The function `multiset_count_occurrences` shall yield the number of occurrences of the first argument in the second argument.

The function `multiset_contents` shall yield the set of values that are elements of the argument.

The function `single_value_multiset` shall yield the multiset containing the second argument value with occurrences given by the first argument. The function `range_set` shall yield the multiset containing a single occurrence of each value that is greater than or equal to the first argument and less than or equal to the second argument.

The function `multiset_choose` applied to a non-empty argument shall yield an arbirarily chosen member of the argument. The function `multiset_choose` applied to an empty argument shall yield _|_.

The function `multiset_image` shall yield the multiset of results of applying the first argument to members of the second argument. The function `multiset_filter` shall yield the multiset of elements of the second argument for which the second argument yields a result value of `true`.

The function set2multiset shall yield the multiset containing exactly those members of the argument, each occurring once.

21.9 Sequence types

A Rosetta sequence shall be an ordered collection of values with duplicates being allowed. The number of elements in a sequence shall be called the length of the sequence. A sequence may be empty, in which case its length is 0. A sequence shall have a bounded number of elements. The elements of a non-empty sequence shall each have an index in the sequence. The index of the first element shall be 0. The index of each subsequent element, if any, shall be one more than the index of its preceding element.

A sequence shall be equivalent to a function whose domain is the set of integers from 0 to one less than the length of the sequence.

The package `rosetta.lang.prelude` shall contain the following declarations of sequence types:

```
// array function is not formally defined

array ( N :: natural; T :: type ) :: type is constant;

sequence [ N :: natural ] ( T :: type ) :: type is
  array ( N, T );

bitvector :: subtype(sequence) is sequence(bit);

wordtype ( N :: natural ) :: subtype(bitvector) is
  array ( N, bit );

string :: subtype(sequence) is sequence(character);
```

The type constructor `array` shall denote the set of all Rosetta sequence values. The type constructor shall yield the set of sequences whose length is the first argument of the type constructor and whose elements are members of the second argument of the type constructor.

The type constructor `sequence` shall denote the set of all Rosetta sequence values. The type constructor shall yield the set of sequences whose elements are members of the argument of the type constructor.

The type `bitvector` shall denote the set of all sequences whose elements are of type `bit`.

The type constructor `wordtype` shall denote the set of all sequences whose elements are of type `bit`. The type constructor shall yield the set of sequences whose length is the argument of the type constructor.

The type `string` shall denote the set of all sequences whose elements are of type `character`.

21.9.1 Sequence constants

The package `rosetta.lang.prelude` shall contain the following declarations of a sequence constant:

```
empty_sequence :: sequence is constant;
```


The constant `empty_sequence` shall denote the empty sequence.

21.9.2 Sequence functions

The package `rosetta.lang.prelude` shall contain the following declarations of functions on sequence types:

```
sequence_concatenation
  [ T :: type ]
  ( L, R :: sequence ( T ) ) :: sequence ( T ) is
  if L = [] then
    R
  else
    cons ( head(L), tail(L) & R );
  end if;

sequence_proper_subsequence ( L, R :: sequence ) :: boolean is
  exists ( a, b :: natural |
           ( L = R sub [ a ,.. b ] ) and ( L /= R ) );

sequence_subsequence ( L, R :: sequence ) :: boolean is
  L < R or L = R

sequence_proper_supersequence ( L, R :: sequence ) :: boolean is
  R < L;

sequence_supersequence ( L, R :: sequence ) :: boolean is
  R < L or R = L;

sequence_length ( R :: sequence ) :: natural is
  if R = [] then
    0
  else
    1 + #( tail(R) )
  end if;

sequence_contents
  [ T :: type ]
  ( R :: sequence ( T ) ) :: multiset ( T ) is
  if #R = 0 then
    {* *}
  else
    {* head(R) *} + ~( tail(r) )
  end if;

sequence_subscript
  [ T :: type ]
  ( L :: sequence ( T );
    R :: sequence ( { 0 ,.. #L - 1 } ) ) :: sequence ( T ) is
  image ( L, R );

cons
  [ T :: type ]
  ( E :: T; S :: sequence ( T ) ) :: sequence ( T ) is constant where
  head ( cons ( E, S ) ) = E and tail ( cons ( E, S ) ) = S;
```



```
range_sequence
  [ T :: subtype ( integer ) + subtype ( character ) ]
  ( L, R :: T ) :: sequence ( T ) is
  if L > R then
    [ ]
  elsif L = R then
    [ L ]
  else
    cons (
      L,
      range_sequence (
        if T =< integer then
          L + 1
        else
          unicode.char ( unicode.ord ( L ) + 1 )
        end if,
        R ) )
  end if;

sequence_value
  [ T :: type ]
  ( N :: natural; V :: T ) :: sequence ( T ) is
  if N = 0 then
    [ ]
  else
    cons (
      V,
      sequence_value ( N - 1, V ) )
  end if;

head [ T :: type ] ( S :: sequence ( T ) ) :: T is
  S(0);

tail [ T :: type ] ( S :: sequence ( T ) ) :: sequence ( T ) is
  S sub [1 ,.. #S - 1];

last [ T :: type ] ( S :: sequence ( T ) ) :: T is
  S(#S - 1);

reverse [ T :: type ] ( S :: sequence ( T ) ) :: sequence ( T ) is
  if #S = 0 then
    S
  else
    reverse ( tail(S) ) & [head(S)]
  end if;

replace
  [ T :: type ]
  ( S :: sequence ( T ); N :: { 0 ,.. #S - 1 }; E :: T ) :: sequence ( T ) is
  (S sub [0 ,.. N - 1]) & [E] & (S sub [N + 1 ,.. #S - 1]);

image
  [ D, R :: type ]
```

```
    ( F :: function ( D, R ); S :: sequence ( D ) ) :: sequence ( R ) is
    if #S = 0 then
      [ ]
    else
      cons ( F ( head ( S ) ), image ( F, tail ( S ) ) )
    end if;

  filter
    [ D :: type ]
    ( P :: predicate ( D ); S :: sequence ( D ) ) :: sequence ( D ) is
    if #S = 0 then
      [ ]
    elsif P ( head ( S ) ) then
      cons ( head ( S ), filter ( P, tail ( S ) ) )
    else
      filter ( P, tail ( S ) )
    end if;

  zip [ N :: natural; T1, T2, T3 :: type ]
      ( F :: <* ( X :: T1; Y :: T2 ) :: T3 *>;
        S1 :: array ( N, T1 );
        S2 :: array ( N, T2 ) ) :: array ( N, T3 ) is
    if N = 0 then
      [ ]
    else
      cons (
        F ( head ( S1 ), head ( S2 ) ),
        zip ( F, tail ( S1 ), tail (S2 ) ) )
    end if;

  reduce
    [ T :: type ]
    ( F :: function ( T, function ( T, T ) );
      I :: T;
      S :: sequence ( T ) ) :: T is
    if #S = 0 then
      I
    else
      reduce ( F, F ( I, head ( S ) ), tail ( S ) )
    end if;


  reduce_tail
    [ T :: type ]
    ( F :: function ( T, function ( T, T ) );
      I :: T;
      S :: sequence ( T ) ) :: T is
    if #S = 0 then
      I
    else
      F ( head ( S ), reduce ( F, I, tail ( S ) ) )
    end if;
```

The function `sequence_concatenation` shall yield the concatenation of its two arguments, that is, the sequence comprising the elements of the first argument in order followed by the elements of the second argument in order.

The function sequence_proper_subsequence shall yield true if the first argument is a proper subsequence of the second argument; otherwise it shall yield false. The function sequence_subsequence shall yield true if the first argument is a subsequence of or the same as the second argument; otherwise it shall yield false. The function sequence_proper_supersequence shall yield true if the first argument is a proper supersequence of the second argument; otherwise it shall yield false. The function sequence_supersequence shall true true if the first argument is a supersequence of or the same as the second argument; otherwise it shall yield false.

The function sequence_length shall yield the length of its argument.

The function sequence_contents shall yield the multiset of values that are elements of its argument.

The function sequence_subscript shall yield a sequence of the same length as the second argument. Each element of the result shall be the element of the first argument whose index is the element of the second argument with the same index as the result element.

The function cons shall yield the sequence whose first element is the first argument and whose subsequent elements are the elements of the second argument in order.

The function range_sequence shall yield the sequence containing a single occurrence of each value that is greater than or equal to the first argument and less than or equal to the second argument. The elements shall occur in the sequence in ascending order.

The function sequence_value shall yield the sequence whose length is the first argument and whose element values are each the second argument.

The function head shall yield the first element of the argument, provided the argument is not empty; otherwise the function shall yield _|_.

The function tail shall yield the sequence of elements of the argument in order excluding the first element, provided the argument is not empty; otherwise the function shall yield _|_.

The function last shall yield last element of the argument, provided the argument is not empty; otherwise the function shall yield _|_.

The function reverse shall yield the sequence of elements of the argument in reverse order.

The function replace shall yield the sequence of elements of the first argument in order, except that the element of the result whose index is the second argument shall be the third argument.

The function image shall yield a sequence of the same length as the second argument. Each element of the result shall be the result of applying the first argument to the element of the second argument with the same index as the result element.

The function filter shall yield a sequence of those elements of the second argument, in order, for which the result of applying the first argument to the element yields true.

The function zip shall yield a sequence of the same length as the second and third arguments. Each element of the result shall be the result of applying the first argument to the elements of the second and third arguments with the same index as the result element. The elements of the second and third argument sequences shall form the first and second arguments, respectively, of the function application.

The function reduce applied to an empty third argument shall yield the second argument. The reduce function applied to a non-empty third argument shall yield the result of recursively applying the reduce function to the same first argument, the result of applying the first argument to the second argument and the head of the third argument, and the tail of the third argument.

The function `reduce_tail` applied to an emtpy third argument shall yield the second argument. The `reduce_tail` function applied to a non-empty third argument shall yield the result of applying the first argument to the head of the third argument and the result of recursive application of the `reduce_tail` function with the same first and second arguments as the original function application and the tail of the third argument of the original function application as the third argument of the recursive application.

21.9.3 Bitvector functions

The package `rosetta.lang.prelude` shall contain the following declarations of functions on bitvector types:

```
bitvector_not ( R :: bitvector ) :: bitvector is
  image ( bit_not ( R ) );

bitvector_and
  [ N :: natural ]
  ( L, R :: wordtype ( N ) ) :: wordtype ( N ) is
  zip ( bit_and, L, R );

bitvector_or
  [ N :: natural ]
  ( L, R :: wordtype ( N ) ) :: wordtype ( N ) is
  zip ( bit_or, L, R );

bitvector nand
  [ N :: natural ]
  ( L, R :: wordtype ( N ) ) :: wordtype ( N ) is
  zip ( bit_nand, L, R );

bitvector_nor
  [ N :: natural ]
  ( L, R :: wordtype ( N ) ) :: wordtype ( N ) is
  zip ( bit_nor, L, R );

bitvector_xor
  [ N :: natural ]
  ( L, R :: wordtype ( N ) ) :: wordtype ( N ) is
  zip ( bit_xor, L, R );

bitvector_xnor
  [ N :: natural ]
  ( L, R :: wordtype ( N ) ) :: wordtype ( N ) is
  zip ( bit_xnor, L, R );

bitvector_implies
  [ N :: natural ]
  ( L, R :: wordtype ( N ) ) :: wordtype ( N ) is
  zip ( bit_implies, L, R );

bitvector_implied_by
  [ N :: natural ]
```



```
    ( L, R :: wordtype ( N ) ) :: wordtype ( N ) is
    zip ( bit_implied_by, L, R );

  bv2nat ( B :: bitvector ) :: natural is
    let ( weights :: sequence(natural) be map(__^__(2), [0 ,.. #B - 1]) )
    in
      reduce(__+__, 0, zip(__*__, B, weights));

  nat2bv ( N :: natural ) :: bitvector is
    if N in {0, 1} then
      [N]
    else
      nat2bv ( N div 2 ) & [ N mod 2 ]
    end if;

  bv2int ( B :: bitvector ) :: integer is
    let ( weights :: sequence(natural) be map(__^__(2), [0 ,.. #B - 1]) )
    reduce(__+__, 0, zip(__*__, B,
                         replace(weights, #weights - 1, -(last(weights))) ));

  int2bv ( I :: integer ) :: bitvector is
    if I in {-1, 0} then
      [-I]
    else
      int2bv ( ( I - I mod 2 ) div 2 ) & [ I mod 2 ]
    end if;

  twos ( B :: bitvector ) :: bitvector is
    let ( add_bit ( B :: bitvector; C :: bit ) :: bitvector be
            if #B = 0 then
              []
            else
              cons ( head(B) xor C, add_bit ( tail(B), head(B) and C ) )
            end if )
    in
      add_bit ( not B, 1 )
    end let;

  ashl ( B :: bitvector; P :: integer ) :: bitvector is
    if P = 0 or #B = 0 then
      B
    elsif P < 0 then
      ashr ( B, -P )
    else
      ashl ( cons(head(B), B sub [0 ,.. #B - 2]), P - 1 )
    end if;

  ashr ( B :: bitvector; P :: integer ) :: bitvector is
    if P = 0 or #B = 0 then
      B
    elsif P < 0 then
      ashl ( B, -P )
    else
      ashr ( tail(B) & [last(B)], P - 1 )
```



```
    end if;

  lshl ( B :: bitvector; P :: integer ) :: bitvector is
    if P = 0 or #B = 0 then
      B
    elsif P < 0 then
      lshr ( B, -P )
    else
      lshl ( cons(0, B sub [0 ,.. #B - 2]), P - 1 )
    end if;

  lshr ( B :: bitvector; P :: integer ) :: bitvector is
    if P = 0 or #B = 0 then
      B
    elsif P < 0 then
      lshl ( B, -P )
    else
      lshr ( tail(B) & [0], P - 1 )
    end if;

  rotl ( B :: bitvector; P :: integer ) :: bitvector is
    if P = 0 or #B = 0 then
      B
    elsif P < 0 then
      rotr ( B, -P )
    else
      rotl ( cons(last(B), B sub [0 ,.. #B - 2]), P - 1 )
    end if;

  rotr ( B :: bitvector; P :: integer ) :: bitvector is
    if P = 0 or #B = 0 then
      B
    elsif P < 0 then
      rotl ( B, -P )
    else
      rotr ( tail(B) & [head(B)], P - 1 )
    end if;

  padl ( B :: bitvector; V :: bit; L :: natural ) :: bitvector is
    if L = 0 then
      []
    elsif #B = 0 then
      cons ( V, padl(B, V, L - 1) )
    else
      cons ( head(B), padl(tail(B), V, L - 1) )
    endif;

  padr ( B :: bitvector; V :: bit; L :: natural ) :: bitvector is
    if L = 0 then
      []
    elsif #B = 0 then
      padr(B, V, L - 1) & [V]
    else
      padr(B sub [0 ,.. #B - 2], V, L - 1) & [last(B)]
```

```
    end if;
```

The function `bitvector_not` shall yield the elementwise logical negation of its argument. The function `bitvector_and` shall yield the elementwise logical conjunction of its arguments. The function `bitvector_or` shall yield the elementwise logical disjunction of its arguments. The function `bitvector_nand` shall yield the elementwise logical negated conjunction of its arguments. The function `bitvector_nor` shall yield the elementwise logical negated disjunction of its arguments. The function `bitvector_xor` shall yield the elementwise logical exclusive disjunction of its arguments. The function `bitvector_xnor` shall yield the elementwise logical negated exclusive disjunction of its arguments. The function `bitvector_implies` shall yield the elementwise logical implication of the second argument by the first argument. The function `bitvector_implied_by` shall yield the elementwise logical implication of the first argument by the second argument.

The function `bv2nat` shall interpret the elements of the argument as the bits of an unsigned binary-coded natural number, with the first element of the argument being the least-significant bit and the last element of the argument being the most significant bit. The function shall yield the number.

The function `nat2bv` shall yield a bitvector whose length is the minimum number of bits necessary to represent the first argument using unsigned binary coding and whose elements constitute the bits of the unsigned binary code of the first argument, with the first element of the bitvector being the least-significant bit and the last element of the bitvector being the most significant bit.

The function `bv2int` shall interpret the elements of the argument as the bits of a twos-complement binary-coded integer, with the first element of the argument being the least-significant bit and the last element of the argument being the sign bit. The function shall yield the integer.

The function `int2bv` shall yield a bitvector whose length is the minimum number of bits necessary to represent the first argument using twos-complement binary coding and whose elements constitute the bits of the twos-complement binary code of the first argument, with the first element of the bitvector being the least-significant bit and the last element of the bitvector being the sign bit.

The function `twos` shall interpret the elements of the argument as the bits of a twos-complement binary-coded integer, with the first element of the argument being the least-significant bit and the last element of the argument being the sign bit. The function shall yield twos-complement of the argument.

The function `ashl` shall yield the first argument shifted left arithmetically. The function `ashr` shall yield the first argument shifted right arithmetically. The function `lshl` shall yield the first argument shifted left logically. The function `lshr` shall yield the first argument shifted right logically. The function `rotl` shall yield the first argument rotated left. The function `rotr` shall yield the first argument rotated right.

The shift and rotate functions applied to an empty first argument or with a second argument of 0 shall yield the first argument. The functions that shift or rotate left, when applied to a negative second argument, shall shift or rotate right by the number of positions given by the negative of the second argument; otherwise, they shall shift or rotate left. Similarly, the functions that shift or rotate right, when applied to a negative second argument, shall shift or rotate left by the number of positions given by the negative of the second argument; otherwise, they shall shift or rotate right. The arithmetic shift functions shall fill the vacated bits with a copy of the original bit at the extreme end that is vacated. The logical shift functions shall fill the vacated bits with 0. In all cases, a shift or rotate to the left shall move bits from positions with lesser indices to positions with greater indices, and a shift or rotate to the right shall move bits from positions with greater indices to positions with lesser indices.

The functions `padl` and `padr` shall yield a bitvector that is a copy of the first argument with bits added or removed to produce a bitvector of length given by the third argument. If the third argument is the length of the first argument, the result shall be the first argument. If the third argument is less than the length of the first argument, the result of `padl` shall be the first argument with higher-index elements removed, and the result of `padr` shall be the first argument with lower-index elements removed. If the third argument is greater than the length of the first argument, the result of padl shall be the first argument with copies of the second argument added at higher-index positions, and the result of padr shall be the first argument with copies of the second argument added at lower-index positions.

21.10 Facet functions

The package rosetta.lang.prelude shall contain the following declarations of functions on facets and facet equivalents:

```
// facet_sum function is not formally defined

facet_sum ( L, R :: rosetta.lang.null ) :: rosetta.lang.null is constant;

// facet_product function is not formally defined

facet_product ( L, R :: rosetta.lang.null ) :: rosetta.lang.null is constant;

// facet_implies function is not formally defined

facet_implies ( L, R :: rosetta.lang.null ) :: boolean is constant;

facet_implied_by ( L, R :: rosetta.lang.null ) :: boolean is
  R => L;
```

The function facet_sum shall yield a facet that is the category-theoretic coproduct of its arguments. The function facet_product shall yield a facet that is the category-theoretic product of its arguments.

The function and facet_implies shall yield true if the first argument is a refinement of the second argument, that is, if there is a category-theoretic homomorphism from the second argument to the first argument; otherwise it shall yield false. The function facet_implied_by shall yield true if the second argument is a refinement of the first argument, that is, if there is a category-theoretic homomorphism from the first argument to the second argument; otherwise it shall yield false.

21.11 Operators

The package rosetta.lang.prelude shall contain the following declarations of functions whose labels are operator interpretation labels:

```
__::__ :: <* ( L :: universal; R :: type ) :: R *> is
  universal_type_assertion;

__in__ :: <* ( L :: universal; R :: type) :: boolean *> is
  universal_type_member;

__=__ :: <* ( L, R :: universal ) :: boolean *> is
  universal_equals;

__/=__ :: <* ( L, R :: universal ) :: boolean *> is
  universal_not_equals ;

__==__ :: <* ( L, R :: universal ) :: boolean *> is
  universal_equivalent;

+__ :: <* ( R :: number ) :: number *> is
```



```
    number_identity;

  -__ :: <* ( R :: number ) :: number *> is
    number_negation;

  __+__ [ T :: { type, complex, multiset, rosetta.lang.null } ]
        ( L, R :: T ) :: T is
    case T is
      { type } -> type_union ( L, R )
      | { complex } -> complex_addition ( L, R )
      | { multiset } -> multiset_union ( L, R )
      | { rosetta.lang.null } -> facet_sum ( L, R )
    end case;

  __-__ [ T :: { type, complex, multiset } ]
        ( L, R :: T ) :: T is
    case T is
      { type } -> type_difference ( L, R )
      | { complex } -> complex_subtraction ( L, R )
      | { multiset } -> multiset_difference ( L, R )
    end case;

  __*__ [ T :: { type, complex, multiset, rosetta.lang.null } ]
        ( L, R :: T ) :: T is
    case T is
      { type } -> type_intersection ( L, R )
      | { complex } -> complex_multiplication ( L, R )
      | { multiset } -> multiset_intersection ( L, R )
      | { rosetta.lang.null } -> facet_product ( L, R )
    end case;

  __/__ :: <* ( L, R :: complex ) :: complex *> is
    complex_division;

  __^__ :: <* ( L, R :: complex ) :: complex *> is
    complex_exponentiation;

  __min__ [ T :: { real, imaginary } ]
          ( L, R :: T ) :: T is
    case T is
      { real } -> real_min ( L, R )
      | { imaginary } -> imaginary_min ( L, R )
    end case;

  __max__ [ T :: { real, imaginary } ]
          ( L, R :: T ) :: T is
    case T is
      { real } -> real_max ( L, R )
      | { imaginary } -> imaginary_max ( L, R )
    end case;

  __<__ [ T :: { type, real, imaginary, character,
                 multiset, sequence, function } ]
        ( L, R :: T ) :: boolean is
```



```
  case T is
    { type } -> type_proper_subtype
    | { real } -> real_less_than ( L, R )
    | { imaginary } -> imaginary_less_than ( L, R )
    | { character } -> character_less_than ( L, R )
    | { multiset } -> multiset_proper_submultiset ( L, R )
    | { sequence } -> sequence_proper_subsequence ( L, R )
    | { function } -> function_proper_contained_in ( L, R )
  end case;

__=<__ [ T :: { type, real, imaginary, character,
                multiset, sequence, function } ]
      ( L, R :: T ) :: boolean is
  case T is
    { type } -> type_proper_subtype
    | { real } -> real_less_than_or_equals ( L, R )
    | { imaginary } -> imaginary_less_than_or_equals ( L, R )
    | { character } -> character_less_than_or_equals ( L, R )
    | { multiset } -> multiset_submultiset ( L, R )
    | { sequence } -> sequence_subsequence ( L, R )
    | { function } -> function_contained_in ( L, R )
  end case;

__>__ [ T :: { type, real, imaginary, character,
               multiset, sequence, function } ]
      ( L, R :: T ) :: boolean is
  case T is
    { type } -> type_proper_supertype
    | { real } -> real_greater_than ( L, R )
    | { imaginary } -> imaginary_greater_than ( L, R )
    | { character } -> character_greater_than ( L, R )
    | { multiset } -> multiset_proper_supermultiset ( L, R )
    | { sequence } -> sequence_proper_supersequence ( L, R )
    | { function } -> function_proper_contains ( L, R )
  end case;

__>=__ [ T :: { type, real, imaginary, character,
                multiset, sequence, function } ]
      ( L, R :: T ) :: boolean is
  case T is
    { type } -> type_proper_supertype
    | { real } -> real_greater_than_or_equals ( L, R )
    | { imaginary } -> imaginary_greater_than_or_equals ( L, R )
    | { character } -> character_greater_than_or_equals ( L, R )
    | { multiset } -> multiset_supermultiset ( L, R )
    | { sequence } -> sequence_supersequence ( L, R )
    | { function } -> function_containeds ( L, R )
  end case;

__div__ :: <* ( L, R :: integer ) :: integer *> is
  integer_division;

__mod__ :: <* ( L, R :: integer ) :: integer *> is
  integer_modulus;
```

```
__rem__ :: <* ( L, R :: integer ) :: integer *> is
  integer_remainder;

%__ [ T :: { boolean, bit } ]
    ( R :: T ) ::
    case T is
      { boolean } -> bit
      | { bit } -> boolean
    end case is
  case T is
    { boolean } -> boolean_to_bit ( R )
    | { bit } -> bit_to_boolean ( R )
  end case;

not__ [ T :: { boolean, bit, bitvector } ]
      ( R :: T ) :: T is
  case T is
    { boolean } -> boolean_not ( R )
    | { bit } -> bit_not ( R )
    | { bitvector } -> bitvector_not ( R )
  end case;

__and__ [ T :: { boolean, bit, bitvector } ]
      ( L, R :: T ) :: T is
  case T is
    { boolean } -> boolean_and ( L, R )
    | { bit } -> bit_and ( L, R )
    | { bitvector } -> bitvector_and ( L, R )
  end case;

__or__ [ T :: { boolean, bit, bitvector } ]
      ( L, R :: T ) :: T is
  case T is
    { boolean } -> boolean_or ( L, R )
    | { bit } -> bit_or ( L, R )
    | { bitvector } -> bitvector_or ( L, R )
  end case;

__nand__ [ T :: { boolean, bit, bitvector } ]
      ( L, R :: T ) :: T is
  case T is
    { boolean } -> boolean_nand ( L, R )
    | { bit } -> bit_nand ( L, R )
    | { bitvector } -> bitvector_nand ( L, R )
  end case;

__nor__ [ T :: { boolean, bit, bitvector } ]
      ( L, R :: T ) :: T is
  case T is
    { boolean } -> boolean_nor ( L, R )
    | { bit } -> bit_nor ( L, R )
    | { bitvector } -> bitvector_nor ( L, R )
  end case;
```

```
__xor__ [ T :: { boolean, bit, bitvector } ]
      ( L, R :: T ) :: T is
  case T is
    { boolean } -> boolean_xor ( L, R )
    | { bit } -> bit_xor ( L, R )
    | { bitvector } -> bitvector_xor ( L, R )
  end case;

__xnor__ [ T :: { boolean, bit, bitvector } ]
      ( L, R :: T ) :: T is
  case T is
    { boolean } -> boolean_xnor ( L, R )
    | { bit } -> bit_xnor ( L, R )
    | { bitvector } -> bitvector_xnor ( L, R )
  end case;

__=>__ [ T :: { boolean, bit, bitvector, rosetta.lang.null } ]
    ( R :: T ) ::
    case T is
      { boolean, bit, bitvector } -> T
      | { rosetta.lang.null } -> boolean
    end case is
  case T is
    { boolean } -> boolean_implies ( L, R )
    | { bit } -> bit_implies ( L, R )
    | { bitvector } -> bitvector_implies ( L, R )
    | { rosetta.lang.null } -> facet_implies ( L, R )
  end case;

__implies__ [ T :: { boolean, bit, bitvector, rosetta.lang.null } ]
    ( R :: T ) ::
    case T is
      { boolean, bit, bitvector } -> T
      | { rosetta.lang.null } -> boolean
    end case is
  case T is
    { boolean } -> boolean_implies ( L, R )
    | { bit } -> bit_implies ( L, R )
    | { bitvector } -> bitvector_implies ( L, R )
    | { rosetta.lang.null } -> facet_implies ( L, R )
  end case;

__implied_by__ [ T :: { boolean, bit, bitvector, rosetta.lang.null } ]
    ( R :: T ) ::
    case T is
      { boolean, bit, bitvector } -> T
      | { rosetta.lang.null } -> boolean
    end case is
  case T is
    { boolean } -> boolean_implied_by ( L, R )
    | { bit } -> bit_implied_by ( L, R )
    | { bitvector } -> bitvector_implied_by ( L, R )
    | { rosetta.lang.null } -> facet_implied_by ( L, R )
```

```
  end case;

#__ [ T :: { set, multiset, sequence } ]
    ( R :: T ) :: natural is
  case T is
    { set } -> set_cardinality ( R )
    | { multiset } ->  multiset_cardinality ( R )
    | { sequence } -> sequence_length ( R );

__#__ ( V :: universal; M :: multiset ) :: natural is
  multiset_count_occurrences

__&__ :: <* ( T :: type ) :: type is
          <* ( L, R :: sequence ( T ) ) :: sequence ( T ) *> *> is
  sequence_concatenation;

~__ [ E :: type;  T :: { set ( E ), multiset ( E ), sequence  ( E ) } ]
    ( R :: T ) ::
    case T is
      { set ( E ), multiset ( E ) } -> set ( E )
      | { sequence  ( E ) } -> multiset ( E )
    end case is
  case T is
      { set ( E ) } -> set_contents ( R )
      | { multiset ( E ) } -> multiset_contents ( R )
      | { sequence  ( E ) } -> sequence_contents ( R )
  end case;

__sub__ ::
  <* ( T :: type ) :: type is
     <* ( L :: sequence ( T ); R :: sequence ( natural ) )
        :: sequence ( T ) *> *> is
  sequence_subscript;

__&&__ ::
  <* ( T1, T2, T3 :: type ) :: type is
     <* ( L :: function ( T2, T3 ); R :: function ( T1, T2) )
        :: function ( T1, T3 ) *> *> is
  function_composition;
```

22. The package rosetta.lang.unicode

The library `rosetta.lang` shall contain a predefined package named `unicode`. The domain of the package shall the `rosetta.lang.static`. The package shall contain declarations of items as specified in this clause. The package shall not contain declarations of any other items.

Except as otherwise specified, all functions declared in `rosetta.lang.prelude` shall be strict; that is, if any argument is _|_, the function shall yield _|_.

22.1 Unicode types and subtypes

The package `rosetta.lang.unicode` shall contain the following declarations of types and subtypes:

```
code_value :: subtype(natural) is constant;

latin_1 :: subtype(character) is { 'U+0000' ,.. 'U+00FF' };

ascii :: subtype(latin_1) is { 'U+0000' ,.. 'U+007F' };

letter :: subtype(character) is constant;

letter_uppercase :: subtype(letter) is constant;
letter_lowercase :: subtype(letter) is constant;
letter_titlecase :: subtype(letter) is constant;
letter_modifier :: subtype(letter) is constant;
letter_other :: subtype(letter) is constant;

mark :: subtype(character) is constant;

mark_nonspacing :: subtype(mark) is constant;
mark_spacing_combining :: subtype(mark) is constant;
mark_enclosing :: subtype(mark) is constant;

number :: subtype(character) is constant;

number_decimal_digit :: subtype(number) is constant;
number_letter :: subtype(number) is constant;
number_other :: subtype(number) is constant;

separator :: subtype(character) is constant;

separator_space :: subtype(separator) is constant;
separator_line :: subtype(separator) is constant;
separator_paragraph :: subtype(separator) is constant;

other :: subtype(character) is constant;

other_control :: subtype(other) is constant;
other_format :: subtype(other) is constant;
other_surrogate :: subtype(other) is constant;
other_private_use :: subtype(other) is constant;
other_not_assigned :: subtype(other) is constant;
```



```
punctuation :: subtype(character) is constant;

punctuation_connector :: subtype(punctuation) is constant;
punctuation_dash :: subtype(punctuation) is constant;
punctuation_open :: subtype(punctuation) is constant;
punctuation_close :: subtype(punctuation) is constant;
punctuation_initial_quote :: subtype(punctuation) is constant;
punctuation_final_quote :: subtype(punctuation) is constant;
punctuation_other :: subtype(punctuation) is constant;

symbol :: subtype(character) is constant;

symbol_math :: subtype(symbol) is constant;
symbol_currency :: subtype(symbol) is constant;
symbol_modifier :: subtype(symbol) is constant;
symbol_other :: subtype(symbol) is constant;
```

The subtype `code_value` shall denote the set of all code values of Unicode characters.

The subtype `latin_1` shall denote the set of Unicode characters that are in the Latin-1 subset, that is, the characters in the groups C0 Controls and Basic Latin and C1 Controls and Latin-1 Supplement. The subtype `ascii` shall denote the set of Unicode characters that are in the ASCII subset, that is, the characters in the group C0 Controls and Basic Latin.

The subtype `letter_uppercase` shall denote the set of Unicode characters whose General Category is Lu (Letter, uppercase). The subtype `letter_lowercase` shall denote the set of Unicode characters whose General Category is Ll (Letter, lowercase). The subtype `letter_titlecase` shall denote the set of Unicode characters whose General Category is Lt (Letter, titlecase). The subtype `letter_modifier` shall denote the set of Unicode characters whose General Category is Lm (Letter, modifier). The subtype `letter_other` shall denote the set of Unicode characters whose General Category is Lo (Letter, other). The subtype `letter` shall denote the set of Unicode characters whose General Category is any of Lu, Ll, Lt, Lm or Lo.

The subtype `mark_nonspacing` shall denote the set of Unicode characters whose General Category is Mn (Mark, nonspacing). The subtype `mark_spacing_combining` shall denote the set of Unicode characters whose General Category is Mc (Mark, spacing combining). The subtype `mark_enclosing` shall denote the set of Unicode characters whose General Category is Me (Mark, enclosing). The subtype `mark` shall denote the set of Unicode characters whose General Category is any of Mn, Mc or Me.

The subtype `number_decimal_digit` shall denote the set of Unicode characters whose General Category is Nd (Number, decimal digit). The subtype `number_letter` shall denote the set of Unicode characters whose General Category is Nl (Number, letter). The subtype `number_other` shall denote the set of Unicode characters whose General Category is No (Number, other). The subtype `number` shall denote the set of Unicode characters whose General Category is any of Nd, Nl or No.

The subtype `separator_space` shall denote the set of Unicode characters whose General Category is Zs (Separator, space). The subtype `separator_line` shall denote the set of Unicode characters whose General Category is Zl (Separator, line). The subtype `separator_paragraph` shall denote the set of Unicode characters whose General Category is Zp (Separator, paragraph). The subtype `separator` shall denote the set of Unicode characters whose General Category is any of Zs, Zl or Zp.

The subtype `other_control` shall denote the set of Unicode characters whose General Category is Cc (Other, control). The subtype `other_format` shall denote the set of Unicode characters whose General Category is Cf (Other, format). The subtype `other_surrogate` shall denote the set of Unicode characters whose General Category is Cs

(Other, surrogate). The subtype other_private_use shall denote the set of Unicode characters whose General Category is Co (Other, private use). The subtype other_not_assigned shall denote the set of Unicode characters whose General Category is Cn (Other, not assigned). The subtype other shall denote the set of Unicode characters whose General Category is any of Cc, Cf, Cs, Co or Cn.

The subtype punctuation_connector shall denote the set of Unicode characters whose General Category is Pc (Punctuation, connector). The subtype punctuation_dash shall denote the set of Unicode characters whose General Category is Pd (Punctuation, dash). The subtype punctuation_open shall denote the set of Unicode characters whose General Category is Ps (Punctuation, open). The subtype punctuation_close shall denote the set of Unicode characters whose General Category is Pe (Punctuation, close). The subtype punctuation_initial_quote shall denote the set of Unicode characters whose General Category is Pi (Punctuation, initial quote). The subtype punctuation_final_quote shall denote the set of Unicode characters whose General Category is Pf (Punctuation, final quote). The subtype punctuation_other shall denote the set of Unicode characters whose General Category is Po (Punctuation, other). The subtype punctuation shall denote the set of Unicode characters whose General Category is any of Pc, Pd, Ps, Pe, Pi, Pf or Po.

The subtype symbol_math shall denote the set of Unicode characters whose General Category is Sm (Symbol, math). The subtype symbol_currency shall denote the set of Unicode characters whose General Category is Sc (Symbol, currency). The subtype symbol_modifier shall denote the set of Unicode characters whose General Category is Sk (Symbol, modifier). The subtype symbol_other shall denote the set of Unicode characters whose General Category is So (Symbol, other). The subtype symbol shall denote the set of Unicode characters whose General Category is any of Sm, Sc, Sk or So.

22.2 Unicode functions

The package rosetta.lang.unicode shall contain the following declarations of functions:

```
ord ( C :: character ) :: code_value is constant;

char ( V :: code_value ) :: character is constant;

to_uppercase ( S :: string ) :: string is constant;

to_lowercase ( S :: string ) :: string is constant;

to_titlecase ( S :: string ) :: string is constant;

to_case_fold ( S :: string ) :: string is constant;

is_lowercase ( S :: string ) :: boolean is constant;

is_uppercase ( S :: string ) :: boolean is constant;

is_titlecase ( S :: string ) :: boolean is constant;

is_case_folded ( S :: string ) :: boolean is constant;

is_cased ( S :: string ) :: boolean is constant;

numeric_value ( C :: number ) :: rational is constant;
```

The function ord shall yield the Unicode code calue of its argument. The function char shall yield the Unicode character whose code value is the argument.

The function `to_uppercase` shall yield a string in which the characters of the argument are mapped to uppercase according to the rules defined in The Unicode Standard. The function `to_lowercase` shall yield a string in which the characters of the argument are mapped to lowercase according to the rules defined in The Unicode Standard. The function `to_titlecase` shall yield a string in which the characters of the argument are mapped to title case according to the rules defined in The Unicode Standard. The function `to_casefold` shall yield a string in which the characters of the argument are case folded according to the rules defined in The Unicode Standard.

The function `is_uppercase` shall yield `true` if all of the characters in the argument that have case are uppercase characters; otherwise the function shall yield `false`. The function `is_lowercase` shall yield `true` if all of the characters in the argument that have case are lowercase characters; otherwise the function shall yield `false`. The function `is_titlecase` shall yield `true` if all of the characters in the argument that have case are title case characters; otherwise the function shall yield `false`. The function `is_case_folded` shall yield `true` if all of the characters in the argument that have case are case folded characters; otherwise the function shall yield `false`. The function `is_cased` shall yield `true` if the argument contains any character that has case; otherwise the function shall yield `false`.

The function `numeric_value` shall yield the numeric value property of the argument.

NOTE — The parameter of the `numeric_value` function is a character of the subtype `rosetta.lang.unicode.number`, not of the type `rosetta.lang.prelude.number`.

22.3 Unicode constants

The package `rosetta.lang.unicode` shall contain the following declarations of constants:

```
nul :: character is 'U+0000';
stx :: character is 'U+0001';
sot :: character is 'U+0002';
etx :: character is 'U+0003';
eot :: character is 'U+0004';
enq :: character is 'U+0005';
ack :: character is 'U+0006';
bel :: character is 'U+0007';
bs :: character is 'U+0008';
ht :: character is 'U+0009';
lf :: character is 'U+000A';
vt :: character is 'U+000B';
ff :: character is 'U+000C';
cr :: character is 'U+000D';
so :: character is 'U+000E';
si :: character is 'U+000F';
dle :: character is 'U+0010';
dc1 :: character is 'U+0011';
dc2 :: character is 'U+0012';
dc3 :: character is 'U+0013';
dc4 :: character is 'U+0014';
nak :: character is 'U+0015';
syn :: character is 'U+0016';
etb :: character is 'U+0017';
can :: character is 'U+0018';
em :: character is 'U+0019';
sub :: character is 'U+001A';
esc :: character is 'U+001B';
```



```
  fs :: character is 'U+001C';
  gs :: character is 'U+001D';
  rs :: character is 'U+001E';
  us :: character is 'U+001F';
  sp :: character is 'U+0020';
  del :: character is 'U+007F';
  bph :: character is 'U+0082';
  nbh :: character is 'U+0083';
  ind :: character is 'U+0084';
  nel :: character is 'U+0085';
  ssa :: character is 'U+0086';
  esa :: character is 'U+0087';
  hts :: character is 'U+0088';
  htj :: character is 'U+0089';
  vts :: character is 'U+008A';
  pld :: character is 'U+008B';
  plu :: character is 'U+008C';
  ri :: character is 'U+008D';
  ss2 :: character is 'U+008E';
  ss3 :: character is 'U+008F';
  dcs :: character is 'U+0090';
  pu1 :: character is 'U+0091';
  pu2 :: character is 'U+0092';
  sts :: character is 'U+0093';
  cch :: character is 'U+0094';
  mw :: character is 'U+0095';
  spa :: character is 'U+0096';
  epa :: character is 'U+0097';
  sos :: character is 'U+0098';
  sci :: character is 'U+009A';
  csi :: character is 'U+009B';
  st :: character is 'U+009C';
  osc :: character is 'U+009D';
  pm :: character is 'U+009E';
  apc :: character is 'U+009F';
  nbsp :: character is 'U+00A0';
  shy :: character is 'U+00AD';
```

The constants shall denote the Latin-1 control characters.

23. Reflection packages

This clause defines the packages that are used to represent the syntax and semantics of Rosetta specifications.

23.1 The package rosetta.lang.reflect.lexical

The library `rosetta.lang.reflect` shall contain a predefined package named `lexical`. The domain of the package shall the `rosetta.lang.static`. The package shall contain declarations as specified in this clause. The package shall not contain any other declarations.

The declarations of the package `rosetta.lang.reflect.lexical` shall include a use clause of the form:

```
use rosetta.lang.unicode;
```

23.2 The package rosetta.lang.reflect.abstract_syntax

23.3 The package rosetta.lang.reflect.semantics

23.4 The package rosetta.lang.reflect.simplification

23.5 The package rosetta.lang.reflect.name_expansion

24. Predefined domains

This clause defines the domains that are predefined in Rosetta.

24.1 The domain rosetta.lang.null

The domain rosetta.lang.null shall be the domain that has no parent domain and in which nothing is observed. It shall denote the category of coalgebras that extend the coalgebra that has no observers.

24.2 The domain rosetta.lang.static

24.3 The domain rosetta.lang.state_based