Specware 4.1 Language Manual

Specware 4.1 Language Manual

Table of Contents

Disclaimer

As experience is gained with Specware 4.1, both the operation of the Specware system and the Metaslang language are bound to undergo changes, which may not always be fully “backwards compatible”.

For updates, news and bug reports, visit the Specware web site http://www.specware.org.

Chapter 1. Introduction to Specware

1.1. What Is Specware?

Specware is a tool for building and manipulating a collection of related specifications. Specware can be considered:

- a design tool, because it can represent and manipulate designs for complex systems, software or otherwise
- a logic, because it can describe concepts in a formal language with rules of deduction
- a programming language, because it can express programs and their properties
- a database, because it can store and manipulate collections of concepts, facts, and relationships

Specifications are the primary units of information in Specware. A specification, or theory, describes a concept to some degree of detail. To add properties and extend definitions, you create new specifications that import or combine earlier specifications. Within a specification, you can reason about objects and their relationships. You declare sorts (data types) and operations (ops, functions), axioms that state properties of operations, and theorems that follow logically from axioms.

A morphism is a relationship between specifications that describes how the properties of one map to the properties of another. Morphisms describe both part-of and is-a relationships. You can propagate theorems from one specification to another using morphisms; for example, if the QEII is a ship, and ships cannot fly, then the QEII cannot fly.

1.2. What Is Specware For?

Specware is a general-purpose tool that you can use to develop specifications for any system or realm of knowledge. You can do this as an abstract process, with no reference to computer programming; or you can produce a computer program that is provably a correct implementation of a specification; or you can use the process to redesign an existing program.

You can use Specware to:

- **Develop domain theories**

  You can use Specware to do “ontological engineering” -- that is, to describe a real-world domain of knowledge in explicit or rigorous terms. You might wish to develop a domain theory in abstract terms that are not necessarily intended to become a computer program. You can use the inference engine to test the internal logic of your theory, derive conclusions, and propose theorems.

  You can use specifications and morphisms to represent abstract knowledge, with no refinement to any kind of concrete implementation.

  More commonly, you would use Specware to model expert knowledge of engineering design. In this case you would refine your theoretical specifications and morphisms to more concrete levels.

- **Develop code from specifications**

  You can use Specware to develop computer programs from specifications. One advantage of using Specware for this task is that you can prove that the generated code does implement the specification correctly. Another advantage is that you can develop and compare different implementations of the same specification.

- **Develop specifications from code**

  You can use Specware for reverse engineering -- that is, to help you derive a specification from existing code. To do this, you must examine the code to determine what problems are being solved by it, then use Specware’s language Metaslang to express the problems as specifications. In addition to providing a notation tool for this process, Specware allows you to operate on the derived specification. Once you have derived a specification from the original code, you can analyze the specification for correctness and completeness, and also generate different and correct implementations for it.

1.3. The Design Process in Specware

To solve real problems, programs typically combine domain theories about the physical world with problem solving theories about the computational world. Your domain theory is an abstract representation of a real-world problem domain. To implement it, you must transform the domain theory to a concrete computational model. The built-in specification libraries describe mathematical and computational concepts, which are building blocks for an implementation. Your specifications combine real-world knowledge with this built-in computational knowledge to generate program code that solves real-world problems in a rigorous and provable way.

You interpret designs relative to an initial universe of models. In software design, for example, the models are programs, while in engineering design, they are circuits or pieces of metal. To design an object is to choose it from among the universe of possible models. You make this choice by beginning with an initial description and augmenting it until it uniquely describes the model you desire. In Specware, this process is called refinement.

Composition and refinement are the basic techniques of application building in Specware. You compose simpler specifications into more complex ones, and refine more abstract specifications into more concrete ones. When you refine a specification, you create a more specific case of it; that is, you reduce the number of possible models of it.

The process of refinement is also one of composition. To begin the refinement, you construct primitive interpretations that show how to implement an abstract concept in terms of a concrete concept. You then compose interpretations to deepen and widen the refinement.

Specware provides two types of composition for interpretations; horizontal (or parallel), and vertical (or sequential).

- When you compose interpretations horizontally, you increase the scope of what is refined. In the same way you compose specifications to create a more complex specification from simpler parts, you compose refinements horizontally to create a complex refinement from simpler parts.
- When you compose interpretations vertically, you increase the degree of refinement. You compose interpretations sequentially, in a simple, linear progression, to create a deeper refinement from a shallower one.

For example, suppose you are designing a house. A wide but not deep view of the design specifies several rooms but gives no details. A deep but not wide view of the design specifies one room in complete detail. To complete the refinement, you must create a view that is both wide and deep; however, it makes no difference which view you create first.

The final refinement implements a complex, abstract specification by interpreting it to code.

1.4. Stages of Application Building

Conceptually, there are two major stages in producing a Specware application. In the actual process, steps from these two stages may alternate.

1. Building a specification
2. Refining your specifications to constructive specifications

1.4.1. Building a Specification

You must build a specification that describes your domain theory in rigorous terms. To do this, you first create small specifications for basic, abstract concepts, then specialize and combine these to make them more concrete and complex.

To relate concepts to each other in Specware, you use specification morphisms. A specification morphism shows how one concept is a specialization or part of another. For example, the concept “fast car” specializes both “car” and “fast thing”. The concept “room” is part of the concept “house”. You can specialize “room” in different ways, one for each room of the house.

You specialize in order to derive a more concrete specification from a more abstract specification. Because the specialization relation is transitive (if A specializes B and B specializes C, then A specializes C as well), you can combine a series of morphisms to achieve a step-wise refinement of abstract specifications into increasingly concrete ones.

You combine specifications in order to construct a more complex concept from a collection of simpler parts. In general, you increase the complexity of a specification by adding more structural detail.

Specware helps you to handle complexity and scale by providing composition operators that take small specifications and combine them in a rigorous way to produce a complex specification that retains the logic of its parts. Specware provides several techniques for combining specifications, that can be used in combination:

- The import operation allows you to include earlier specifications in a later one.
- The translate operation allows you to rename the parts of a specification.
- The colimit operation glues concepts together into a shared union along shared subconcepts.

A shared union specification combines specializations of a concept. For example, if you combine “red car” with “fast car” sharing the concept “car”, you obtain the shared union “fast, red car”. If you combine them sharing nothing, you obtain “red car and fast car”, which is two cars rather than one. Both choices are available.

1.4.2. Refining your specifications to constructive specifications

You combine specifications to extend the refinement iteratively. The goal is to create a refinement between the abstract specification of your problem domain and a concrete implementation of the problem solution in terms of sorts and operations that ultimately are defined in the Specware libraries of mathematical and computational theories.

For example, suppose you want to create a specification for a card game. An abstract specification of a card game would include concepts such as card, suit, and hand. A refinement for this specification might map cards to natural numbers and hands to lists containing natural numbers.

The Specware libraries contains constructive specifications for various sorts, including natural numbers and lists.

To refine your abstract specification, you build a refinement between the abstract Hand specification and the List-based specification. When all sorts and operations are refined to concrete library-defined sorts and operations, the Specware system can generate code from the specification.

1.5. Reasoning About Your Code

Writing software in Metaslang, the specification and programming language used in Specware, brings many advantages. Along with the previously-mentioned possibilities for incremental development, you have the option to perform rigorous verification of the design and implementation of your code, leading to the a high level of assurance in the correctness of the final application.

1.5.1. Abstractness in Specware

Specware allows you to work directly with abstract concepts independently of implementation decisions. You can put off making implementation decisions by describing the problem domain in general terms, specifying only those properties you need for the task at hand.

In most languages, you can declare either everything about a function or nothing about it. That is, you can declare only its type, or its complete definition. In Specware you must declare the signature of an operation, but after that you have almost complete freedom in stating properties of the operation. You can declare nothing or anything about it. Any properties you have stated can be used for program transformation.

For example, you can describe how squaring distributes over multiplication:

```
axiom square_distributes_over_times is
  fa(a, b) square(a * b) = square(a) * square(b)
```

This property is not a complete definition of the squaring operation, but it is true. The truth of this axiom must be preserved as you refine the operation. However, unless you are going to generate code for `square`, you do not need to supply a complete definition.

The completeness of your definitions determines whether you can create interpretations to code. An interpretation must completely define the operations of the source theory in terms of the target theory. This guarantees that, if the target is implementable, the source is also implementable. However, Specware also allows you to construct interpretation schemes, which need not be definitional extensions. This allows you to make considerable progress in building up and refining an abstract specification in advance of providing complete definitions.

1.5.2. Logical Inference in Specware

Specware performs inference using external theorem provers; the distribution includes SRI's SNARK theorem prover. External provers are connected to Specware through logic morphisms, which relate logics to each other.

You can apply external reasoning agents to refinements in different ways (although only verification is fully implemented in the current release of Specware).

- Verification tests the correctness of a refinement. For example, you can prove that quicksort is a correct refinement of the sorting specification.
- Simplification is a complexity-reducing refinement. For example, given appropriate axioms, you can rewrite `3*a+3*b` to `3*(a+b)`.

- Synthesis derives a refinement for a given specification by combining the domain theory with computational theory. For example, you can derive quicksort semi-automatically from the sorting specification as a solution to a sorting problem, if you describe exactly how the problem is a sorting problem.

Chapter 2. Metaslang

This chapter introduces the Metaslang specification language.

The following sections give the grammar rules and meaning for each Metaslang language construct.

2.1. Preliminaries

2.1.1. The grammar description formalism

The grammar rules used to describe the Metaslang language use the conventions of (extended) BNF. For example, a grammar rule like:

```
wiffle ::= waffle [ waffle-tail ] | piffle { + piffle }*
```

defines a wiffle to be: either a waffle optionally followed by a waffle-tail, or a sequence of one or more piffles separated by terminal symbols **+** . (Further rules would be needed for waffle, waffle-tail and piffle.) In a grammar rule the left-hand side of `::=` shows the kind of construct being defined, and the right-hand side shows how it is defined in terms of other constructs. The sign `|` separates alternatives, the square brackets `[ ... ]` enclose optional parts, and the curly braces plus asterisk `{ ... }*` enclose a part that may be repeated any number of times, including zero times. All other signs stand for themselves, like the symbol **+** in the example rule above.

In the grammar rules terminal symbols appear in a bold font. Some of the terminal symbols used, like **|** and **{**, are very similar to the grammar signs like `|` and `{` as described above. They can hopefully be distinguished by their bold appearance.

Grammar rules may be *recursive*: a construct may be defined in terms of itself, directly or indirectly. For example, given the rule:

```
piffle ::= 1 | M { piffle }*
```

here are some possible piffles:

```
1      M      M1      M111    MMMM    M1M1
```

Note that the last two examples of piffles are ambiguous. For example, `M1M1` can be interpreted as: `M` followed by the two piffles `1` and `M1`, but also as: `M` followed by the three piffles `1`, `M`, and another `1`. Some of the actual grammar rules allow ambiguities; the accompanying text will indicate how they are to be resolved.

2.1.2. Models

op ::= op-name

In Metaslang, *op* is used used as an abbreviation for “op-name”, where op-names are declared names representing values. (*Op* for *operator*, a term used for historical reasons, although including things normally not considered operators.)

spec ::= spec-form

The term spec is used as short for spec-form. The *semantics* of Metaslang specs is given in terms of classes of *models*. A model is an assignment of sets of values (called “types”) to all the type-names and of “typed” values to all the ops declared -- explicitly or implicitly -- in the spec. The notion of *value* includes numbers, strings, arrays, functions, etcetera. A typed value can be thought of as a pair (*T*, *V*), in which *T* is a type and *V* is a value that is an inhabitant of *T*. For example, the expressions `0 : Nat` and `0 : Integer` correspond, semantically, to the typed values (*N*, 0) and (*Z*, 0), respectively, in which *N* stands for the set of natural numbers {0, 1, 2, ...}, and *Z* for the set of integers {..., -2, -1, 0, 1, 2, ...}. For example, given this spec:

```
spec
  type Even
  op next : Even -> Even
  axiom nextEffect is
    fa(x : Even) ~(next x = x)
endspec
```

one possible model (out of many!) is the assignment of the even integers to `Even`, and of the function that increments an even number by 2 to `next`.

Each model has to *respect typing*; for example, given the above assignment to `Even`, the function that increments a number by 1 does not map all even numbers to even numbers, and therefore can not -- in the same model -- be assigned to `next`. Additionally, the axioms of the spec have to be satisfied by the model. The axiom labeled `nextEffect` above states that the function assigned to op-name `next` maps any value of the type assigned to type-name `Even` to a different value. So the squaring function, although type-respecting, could not be assigned to `next` since it maps 0 to itself.

If all type-respecting combinations of assignments of types to type-names and ops to op-names fail the axioms test, the spec has no models and is called *inconsistent*. Although usually undesirable, an inconsistent spec is not by itself considered erroneous. The Specware system does not attempt to detect inconsistencies, but associated provers can sometimes be used to find them. Not always; in general it is undecidable whether a spec is consistent or not.

In general, the meaning of a construct in a model depends on the assignments of that model, and more generally on an *environment*: a model possibly extended with assignments to local-variables. For example, the meaning of the claim `fa(x : Even) ~(next x = x)` in the axiom `nextEffect` depends on the meanings of `Even` and `next`, while the sub-expression `next x`, for example, also depends on an assignment (of an "even" value) to `x`. To avoid laborious phrasings, the semantic descriptions use language like "the function `next` applied to `x`" as shorthand for this lengthy phrase: "the function assigned in the environment to `next` applied to the value assigned in the environment to `x`".

When an environment is extended with an assignment to a local-variable, any assignments to synonymous ops or other local-variables are superseded by the new assignment in the new environment. In terms of Metaslang text, within the scope of the binding of local-variables, synonymous ops and earlier introduced local-variables (that is, having the same name) are "hidden"; any use of the name in that scope refers to the textually most recently introduced local-variable. For example, given:

```
def x = "op-x"
def y = let v = "let-v" in x
def z = let x = "let-x" in x
```

the value of `y` is `"op-x"` (op `x` is not hidden by the local-variable `v` of the let-binding), whereas the value of `z` is `"let-x"` (op `x` *is* hidden by the local-variable `x` of the let-binding).

2.1.3. Type-correctness

Next to the general requirement that each model respects typing, there are specific correctness requirements for various constructs that constrain the possible types for the components. For example, in an application `f(x)`, the type of the actual-parameter `x` has to match the domain type of function `f`. These requirements are stated in the relevant sections of this language manual. If no type-respecting combinations of assignments meeting all these requirements exist for a given spec, it is considered *incorrect*, and is said to have a type error. This is determined by Specware while elaborating the spec, and signaled as an error. Type-incorrectness differs from inconsistency in that the meaning of the axioms does not come into play, and the question whether an incorrect spec is consistent is moot.

To be precise, there are subtle and less subtle differences between type-incorrectness of a spec and its having no type-respecting combinations of assignments. For example, the following spec is type-correct but has no models:

```
spec
  type Empty = | Just Empty
  op IdoNotExist : Empty
endspec
```

The explanation is that the type-definition for `Empty` generates an *implicit* axiom that all inhabitants of the type `Empty` must satisfy, and for this recursive definition the axiom effectively states that such creatures can't exist: the type `Empty` is uninhabited. That by itself is no problem, but precludes a type-respecting assignment of an inhabitant of `Empty` to op `IdoNotExist`. So the spec, while type-correct, is actually inconsistent. See further under *Type-definitions*.

Here is a type-incorrect spec that has type-respecting combinations of assignments:

```
spec
  type Outcome = | Positive | Negative
  type Sign = | Positive | Zero | Negative
  def whatAmI = Positive
endspec
```

Here there are two constructors `Positive` of different types, the type `Outcome` and the type `Sign`. That by itself is fine, but when such "overloaded" constructors are used, the context must give sufficient information which is meant. Here, the use of `Positive` in the definition for op `whatAmI` leaves both possibilities open; as used it is *type-ambiguous*. Metaslang allows omitting type information provided that, given a type assignment to all local-type-variables in scope, unique types for all typed constructs, such as expressions and patterns, can be inferred from the context. If no complete and unambiguous type-assignment can be made, the spec is not accepted by the Specware system. Type-ambiguous expressions can be disambiguated by using a type annotation, as described under *Annotated-expressions*. In the example, the definition of `whatAmI` can be disambiguated in either of the following ways:

```
def whatAmI : Sign = Positive
def whatAmI = Positive : Sign
```

Also, if the spec elsewhere contains something along the lines of:

```
op signToNat : Sign -> Nat
def sw = signToNat whatAmI
```

that is sufficient to establish that `whatAmI` has type `Sign` and thereby disambiguate the use of `Positive`. See further under *Op-definitions* and *Structors*.

2.1.4. Constructive

When code is generated for a spec, complete "self-contained" code is only generated for type-definitions and op-definitions that are fully *constructive*.
Non-constructiveness is "contagious": a definition is only constructive if all components of the definition are. The type of a type-name without definition is not constructive. A type is only constructive if all component types are. An op without definition is non-constructive, and so is an op whose type is non-constructive. A quantification is non-constructive. The built-in polymorphic equality predicate = is only constructive for *discrete types* (see below).

A type is called discrete if the equality predicate = for that type is constructive. The built-in types `Integer`, `Nat`, `Boolean`, `Char` and `String` are all discrete. Type `List` *`T`* is discrete when *`T`* is. All function types are non-discrete (even when the domain type is the unit type). Sum types, product types and record types are discrete whenever all component types are. Subtype `(`*`T`*` | `*`P`*`)` is discrete when supertype *`T`* is. (Predicate *`P`* need not be constructive: the equality test is that of the supertype.) Quotient type *`T`*` / `*`Q`* is discrete when predicate *`Q`* is constructive. (Type *`T`* need not be discrete: the equality test on the quotient type is just the predicate *`Q`* applied to pairs of members of the *`Q`*-equivalence classes.)

2.2. Lexical conventions

A Metaslang text consists of a sequence of symbols, possibly interspersed with whitespace. The term *whitespace* refers to any non-empty sequence of spaces, tabs, newlines, and comments (described below). A symbol is presented in the text as a sequence of one or more "marks" (ASCII characters). Within a composite (multi-mark) symbol, no whitespace is allowed, but whitespace may be needed between two symbols if together they could be taken for one symbol; in particular, two names that follow each other should be separated by whitespace. More precisely, whitespace is required between two adjacent symbols for each of the following combinations, in which "`abc`" stands for an arbitrary word-symbol, "`<*>`" stands for an arbitrary non-word-symbol, "`?:!`" stands for an arbitrary non-word-symbol starting with a `?`-mark, and "`123`" stands for an arbitrary literal (see below for the definitions of the various classes of symbols):

```
abc  abc
abc  ?:!
abc  123
<*>  <*>
123  abc
123  123
abc  _
  (  *
```

Apart from the last two cases, no whitespace is ever needed adjacent to a special-symbol.

Inside literals (constant-denotations) whitespace is also disallowed, except for "significant-whitespace" as described under *String-literals*.

Other than that, whitespace -- or the lack of it -- has no significance. Whitespace can be used to lay-out the text for readability, but as far as only the meaning is concerned, the following two presentations of the same spec are entirely equivalent:

```
spec
  type Even
  op next : Even -> Even
  axiom nextEffect is
    fa(x : Even) ~(next x = x)
endspec

spec type   Even op   next : Even -> Even axiom nextEffect
is fa(x : Even)~(next     x            = x)endspec
```

2.2.1. Symbols and Names

symbol `::=` name | literal | special-symbol

name `::=` word-symbol | non-word-symbol

word-symbol `::=` word-start-mark `{` word-continue-mark `}*`

word-start-mark `::=` letter

word-continue-mark `::=`
    letter | decimal-digit | `_` | **`?`**

letter `::=`
```
      A | B | C | D | E | F | G | H | I | J | K | L | M
    | N | O | P | Q | R | S | T | U | V | W | X | Y | Z
    | a | b | c | d | e | f | g | h | i | j | k | l | m
    | n | o | p | q | r | s | t | u | v | w | x | y | z
```

decimal-digit `::=` **`0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9`**

non-word-symbol `::=` non-word-mark `{` non-word-mark `}*`

non-word-mark `::=`
```
      ' | ~ | ! | @ | $ | ^ | & | * | -
    | = | + | \ | | | : | < | > | / | ?
```

special-symbol `::=` `_ | ( | ) | [ | ] | { | } | ; | , | .`

Sample names:

```
Date                    $$          ?!
yymmdd2date             <*>         :=:
well_ordered?           ~==
```

For convenience, here are the 14 printing ASCII marks that, next to letters and decimal-digits, can *not* occur in a non-word-symbol:

```
#   %   '   "   _   (   )
[   ]   {   }   ;   ,   .
```

Restriction. As mentioned before, no whitespace is allowed in symbols: while `anode` is a single name, both `a node` and `an ode` consist of two names. Further, the case (lower or upper) of letters in names is significant: `grandparent`, `grandParent` and `grandpaRent` are three different names.

Restriction. In general, names can be chosen freely. However, the following *reserved words* have a special meaning and must not be used for names:

```
as           endspec      infixr         relax
axiom        ex           is             restrict
case         fa           let            spec
choose       false        morphism       then
colimit      fn           obligations    theorem
conjecture   from         of             translate
def          generate     op             true
diagram      if           project        type
else         import       prove          where
embed        in           qualifying
embed?       infixl       quotient
```

They each count as a single symbol, and no whitespace is allowed inside any reserved word.

Likewise, the following *reserved non-words* must not be used for names:

```
:       ::      |       =>      ||      &&
<-      ->      +->     =       ~=      <<
```

In addition, several names -- for example = -- are pre-defined in built-in libraries. While strictly speaking not reserved, they must not be redefined. See further the *Libraries* Appendix.

The non-word-symbols can be used to choose convenient names for infix-operators that, conventionally, are written with non-alphabetic marks.

Some Metaslang users follow the convention of using names that start with a capital letter for unit-identifiers and type-names and for constructors, while word-symbols chosen for op-names and field-names start with a lower-case letter. Both plain local-variables and local-type-variables are often chosen to be single lower-case letters: `x`, `y`, `z`, `a`, `b`, `c`, with the start of the alphabet preferred for local-type-variables. Op-names of predicates (that is, having some type *T* `-> Boolean`) often end with the mark `?`. These are just conventions that users are free to follow or ignore, but in particular some convention distinguishing constructors from op-names and local-variables is recommended.

2.2.2. Comments

comment `::=` line-end-comment | block-comment

line-end-comment `::=` **`%`** line-end-comment-body

line-end-comment-body `::=`
    any-text-up-to-end-of-line

block-comment `::=` **`(*`** block-comment-body **`*)`**

block-comment-body `::=`
    any-text-including-newlines-and-nested-block-comments

Sample comments:

```
% keys must be unique
(* op yymmdd2Date : String -> Date *)
```

Metaslang allows two styles of comments. The `%`-style is light-weight, for adding comment on a line *after* the formal text (or taking a line on its own, but always confined to a single line). The `(*`...`*)`-style can be used for blocks of text, spanning several lines, or stay within a line. Any text remaining on the line after the closing `*)` is processed as formal text. Block-comments may be nested, so the pairs of brackets `(*` and `*)` must be balanced.

A block-comment can not contain a line-end-comment and vice versa: whichever starts first has "the right of way". For example, `(* 100 % or more! *)` is a block-comment with block-comment-body `100 % or more!` . The `%` here is a mark like any other; it does not introduce a line-end-comment. Conversely, in the line-end-comment `% op <*> stands for (*)` the `(*` is part of the line-end-comment-body; it does not introduce a block-comment. Note also that `%` and `(*` have no special significance in literals (which must not contain whitespace, including comments): `"100 % or more!"` is a well-formed string-literal.

2.3. Units

A "unit" is an identifiable unit-term, where "identifiable" means that the unit-term can be referred to by a unit-identifier. Unit-terms can be "elaborated", resulting in specs, morphisms, diagrams or other entities. The effect of elaborating a unit-definition is that its unit-term is elaborated and becomes associated with its unit-identifier.

A Specware project consists of a collection of Metaslang unit-definitions. They can be recorded in one or more Specware files. There are basically two styles for recording unit-definitions using Specware files. In the single-unit style, the file, when processed by Specware, contributes a single unit-definition to the project. In the multiple-unit style, the file may contribute several unit-definitions. The two styles may be freely mixed in a project (but not in the same Specware file). This is explained in more detail in what follows.

```
unit-definition ::= unit-identifier = unit-term

unit-term ::=
      spec-term
    | morphism-term
    | diagram-term
    | target-code-term
    | proof-term

specware-file-contents ::=
      unit-term
    | infile-unit-definition { infile-unit-definition }*

infile-unit-definition ::= fragment-identifier = unit-term

fragment-identifier ::= word-symbol
```

Unit-definitions may use other unit-definitions, including standard libraries, which in Specware 4.1 are supposed to be part of each project. However, the dependencies between units must not form a cycle; it must always be possible to arrange the unit-definitions in an order in which later unit-definitions only depend on earlier ones. How unit-definitions are processed by Specware is further dealt with in the Specware User Manual.

As mentioned above, unit-definitions are collected in Specware files, which in Specware 4.1 must have an `.sw` extension. The Specware files do not directly contain the unit-definitions that form the project. In fact, a user never writes unit-definition explicitly. These are instead determined from the contents of the Specware files using the following rules. There are two possibilities here. The first is that the specware-file-contents consists of a single unit-term. If `P.sw` is the path for the

Specware file, the unit being defined has as its unit-identifier `P`. For example, if file `C:/units/Layout/Fixture.sw` contains a single unit-term `U`, the unit-identifier is `/units/Layout/Fixture`, and the unit-definition it contributes to the project is

```
/units/Layout/Fixture = U
```

(Note that this is not allowed as an infile-unit-definition in a specware-file-contents, since the unit-identifier is not a fragment-identifier.)

The second possibility is that the Specware file contains one or more infile-unit-definitions. If `I` is the fragment-identifier of such an infile-unit-definition, and `P.sw` is the path for the Specware file, the unit being defined has as its unit-identifier `P#I`. For example, if file `C:/units/Layout/Cart.sw` contains an infile-unit-definition `Pos = U`, the unit-identifier is `/units/Layout/Cart#Pos`, and the unit-definition it contributes to the project is

```
/units/Layout/Cart#Pos = U
```

2.3.1. Unit Identifiers

unit-identifier `::=` swpath-based-path `|` relative-path

swpath-based-path `::=` `/` relative-path

relative-path `::=` `{` path-element `/` `}*` path-element `[` `#` fragment-identifier `]`

path-element `::=` word-symbol

Warning. Note that unit-identifiers are processed by the tokenizer like everything else. This means that whitespace is removed and marks not allowed in word-symbols, even if otherwise permitted in filenames (such as `.`), cannot appear in unit-identifiers. For this reason, some care must be taken when naming units.

Unit-identifiers are used to identify unit-terms. Typically, only a final part of the full unit-identifier is used. When Specware is started with environment variable `SWPATH` set to a semicolon-separated list of pathnames for directories, the Specware files are searched for relative to these pathnames; for example, if `SWPATH` is set to `C:/units/Layout;C:/units/Layout/Cart`, then

`C:/units/Layout/Fixture.sw` may be shortened to `/Fixture`, and `C:/units/Layout/Cart.sw` to `/Cart`. How unit-definitions are processed by Specware is further dealt with in the Specware User Manual.

Further, unit-identifiers can be relative to the directory containing the Specware file in which they occur. So, for example, both in file `C:/units/Layout/Fixture.sw` and in file `C:/units/Layout/Cart.sw`, unit-identifier `Tools/Pivot` refers to the unit-term contained in file `C:/units/Layout/Tools/Pivot.sw`, while `Props#SDF` refers to the unit-term of infile-unit-definition `SDF = ...` contained in file `C:/units/Layout/Props.sw`. As a special case, a unit-term with the same name as the file may be referenced without a fragment-identifier. For example, in the current case, if the file `C:/units/Layout/Props.sw` contains the unit-term of infile-unit-definition `Props = ...`, then this unit-term can be referred to either by `Props#Props` or `Props`.

The unit-identifier must identify a unit-definition as described above; the elaboration of the unit-identifier is then the result of elaborating the corresponding unit-term, yielding a spec, morphism, diagram, or other entity.

2.3.2. Specs

```
spec-term ::=
      unit-identifier
    | spec-form
    | spec-qualification
    | spec-translation
    | spec-substitution
    | diagram-colimit
    | obligator
```

Restriction. When used as a spec-term, the elaboration of a unit-identifier must yield a spec.

The elaboration of a spec-term, if defined, yields a “closed” spec-form as defined in the next subsection.

2.3.2.1. Spec Forms

```
spec-form ::= spec { declaration }* endspec
```

Restriction. Spec-forms must be type-correct.

Sample spec-forms:

```
spec import Measures import Valuta endspec
```

A *closed* spec-form is a spec-form containing no import-declarations.

The elaboration of a spec-form yields the Metaslang text which is that spec itself, after expanding any import-declarations. The *meaning* of that text is the class of models of the spec, as described throughout this Chapter.

2.3.2.2. Qualifications

Names of types and ops may be *unqualified* or *qualified*. The difference is that unqualified-names do not contain a dot sign "`.`", while qualified-names are prefixed with a "qualifier" followed by a dot. Examples of unqualified names are `Date`, `today` and `<*>`. Examples of qualified-names are `Calendar.Date`, `Calendar.today` and `Monoid.<*>`.

Qualifiers can be used to disambiguate. For example, there may be reason to use two different ops called `union` in the same context: one for set union, and one for bag (multiset) union. They could then more fully be called `Set.union` and `Bag.union`, respectively. Unlike in earlier versions of Specware, there is no rigid relationship between qualifiers and the unit-identifiers identifying specs. The author of a collection of specs may use the qualifier deemed most appropriate for any type-name or op-name. For example, there could be a single spec dubbed `SetsAndBags` that introduces two new ops, one called `Set.union` and one called `Bag.union`. Generally, types and ops that "belong together" should receive the same qualifier. It is up to the author of the specs to determine what belongs together.

Type-names and op-names are *introduced* in a declaration or definition, and may then be *employed* elsewhere in the same spec. Thus, all occurrences of a type-name or op-name can be divided into "introductions" and "employs". The name as introduced in an introduction is the *full name* of the type or op. If that name is unqualified, the full name is unqualified. If the name as introduced is qualified, then so is the full name.

For employs the rules are slightly different. First, if the name employed occurs just like that in an introduction, then it is the full name. Also, if the name employed is qualified, it is the full name. Otherwise, the name as employed may be unqualified shorthand for a qualified full name. For example, given an employ of the unqualified type-name `Date`, possible qualified full names for it are `Calendar.Date`, `DateAndTime.Date`, `Diary.Date`, and so on. But, of course, the full name must be one that is introduced in the spec. If there is precisely one qualified-name introduced whose last part is the same as the unqualified-name employed, then that name is the full name. Otherwise, type information may be employed to disambiguate ("resolve overloading").

Here is an illustration of the various possibilities:

```
spec
  type Apple
  type Fruit.Apple
  type Fruit.Pear
  type Fruit.Date
  type Calendar.Date
  type Fruit.Basket = Apple * Pear * Date
endspec
```

In the definition of type `Fruit.Basket` we have three unqualified employs of type-names, viz. `Apple`, `Pear` and `Date`. The name `Apple` is introduced like that, so the employ `Apple` already uses the full name; it does not refer to `Fruit.Apple`. The name `Pear` is nowhere introduced just like that, so the employ must be shorthand for some qualified full name. There is only one applicable introduction, namely `Fruit.Pear`. Finally, for `Date` there are two candidates: `Fruit.Date` and `Calendar.Date`. This is ambiguous, and in fact an error. To correct the error, the employ of `Date` should be changed into either `Fruit.Date` or `Calendar.Date`, depending on the intention.

It is possible to give a qualification in one go to all unqualified-names introduced in a spec. If *`Q`* is a qualifier, and *`S`* is a term denoting a spec, then the term *`Q`* `qualifying` *`S`* denotes the same spec as *`S`*, except that each introduction of an unqualified-name *`N`* is replaced by an introduction of the qualified-name *`Q.N`*. Employs that before referred to the unqualified introduction are also accordingly qualified, so that they now refer to the qualified introduction. For example, the value of

```
Company qualifying spec
  type Apple
  type Fruit.Apple
  type Fruit.Pear
  type Fruit.Basket = Apple * Pear
endspec
```

is the same as that of

```
spec
  type Company.Apple
```

```
    type Fruit.Apple
    type Fruit.Pear
    type Fruit.Basket = Company.Apple * Fruit.Pear
  endspec
```

spec-qualification `::=` qualifier **`qualifying`** spec-term

qualifier `::=` word-symbol

qualifiable-name `::=` unqualified-name `|` qualified-name

unqualified-name `::=` name

qualified-name `::=` qualifier **`.`** name

Sample spec-qualification:

```
Weight qualifying /Units#Weights
```

Sample qualifiable-names:

```
Key
$
Calendar.Date
Monoid.<*>
```

Let *`R`* be the result of elaborating spec-term *`S`*. Then the elaboration of qualification *`Q`* `qualifying` *`S`*, where *`Q`* is a qualifier, is *`R`* with each unqualified type-name, op-name or claim-name *`N`* introduced there replaced by the qualified-name *`Q.N`*. The same replacement applies to all uses of *`N`* identifying that introduced name.

For example, the elaboration of

```
Buffer qualifying spec
  op size : Nat
  axiom LargeSize is size >= 1024
endspec
```

results in:

```
spec
  op Buffer.size : Nat
  axiom Buffer.LargeSize is Buffer.size >= 1024
```

```
endspec
```

Note that the claim-name `LargeSize` is unchanged. In later versions of Specware this may change, so that also unqualified claim-names get qualified.

2.3.2.3. Translations

spec-translation `::=` **`translate`** spec-term **`by`** name-map

name-map `::=` **`{`** `[` name-map-item `{` **`,`** name-map-item `}*` `]` **`}`**

name-map-item `::=` type-name-map-item `|` op-name-map-item

type-name-map-item `::=` `[` **`type`** `]` qualifiable-name **`+->`** qualifiable-name

op-name-map-item `::=`
`[` **`op`** `]` annotable-qualifiable-name **`+->`** annotable-qualifiable-name

annotable-qualifiable-name `::=` qualifiable-name `[` **`:`** type-descriptor `]`

Sample spec-translation:

```
translate A by {Counter +-> Nat}
```

Let *R* be the result of elaborating spec-term *S*. Then the elaboration of `translate` *S* `by {` M_1 `+->` N_1`, ...` M_n `+->` N_n `}` is *R* with each occurrence of a qualifiable-name M_i replaced by N_i.

For example, the elaboration of

```
translate spec
  type E
  op i : E
endspec by {
  E +-> Counter,
  i +-> zero
}
```

results in:

```
spec
  type Counter
  op zero : Counter
```

```
endspec
```

2.3.2.4. Substitutions

spec-substitution ::= spec-term **[** morphism-term **]**

Sample spec-substitution:

```
Routing#Basis[morphism /Coll/Lattice ->
                       /Coll/LatticeWithTop {} ]
```

The elaboration of spec-substitution `S[M]` yields the spec `T` obtained as follows. Let spec `R` be the result of elaborating `S`, and morphism `N` that of `M`. Let specs `D` and `C` be the domain and codomain of `N`. First, remove from `R` all declarations of `D`, and subject the result to the effect of `N`, meaning that all name translations of `N` and all extensions with declarations are performed. Then add the declarations of `C`, but without duplications, i.e., as if `C` is imported. The result obtained is `T`.

Restriction. Spec `D` must be a "sub-spec" of spec `R`, meaning that each declaration of `D` is also a declaration of `R`.

Informally, `T` is to `R` as `C` is to `D`.

Except when `R` introduces, next to the type-names and op-names it has in common with `D`, new type-names or op-names that also occur in `C`, the result spec `T` is a categorical colimit of this pushout diagram:

```
D ---------> R
|            |
|            |
|            |
v            v
C ---------> T
```

Although isomorphic to the result that would be obtained by using a diagram-colimit, `T` is more "user-oriented" in two ways: the names in `T` are names from `C`, and axioms of `D` not repeated in `C` are not repeated here either.

For example, assume we have:

```
A = spec
  type Counter
  op reset: Counter
  op tick : Counter -> Counter
  axiom Effect is
    fa (c : Counter) ~(tick c = c)
endspec

B = spec
  def reset = 0
  def tick c = c+1
endspec

M = morphism A -> B {Counter +-> Nat}

AA = spec
  import A
  type Interval = {start: Counter, stop: Counter}
  op isEmptyInterval? : Interval -> Boolean
  def isEmptyInterval? {start = x, stop = y} = (x = y)
endspec
```

Then the result of `AA[M]` is the same as that of this spec:

```
spec
  import B
  type Interval = {start: Nat, stop: Nat}
  op isEmptyInterval? : Interval -> Boolean
  def isEmptyInterval? {start = x, stop = y} = (x = y)
endspec
```

2.3.2.5. Diagram Colimits

diagram-colimit `::=` **`colimit`** diagram-term

The result of elaborating a diagram-colimit is the spec which is the apex of the cocone forming the colimit in the category of specs and spec-morphisms. See further the Specware Tutorial.

2.3.2.6. Obligators

obligator `::=` **`obligations`** unit-term

Restriction. The unit-term of an obligator must either be a spec-term or a morphism-term.

The result of elaborating an obligator is a spec containing the proof obligations engendered by the spec or morphism resulting from elaborating its unit-term. These proof obligations are expressed as conjectures; they can be discharged by proving them, using proof-terms. See further the Specware User Manual.

2.3.3. Morphisms

morphism-term `::=`
  unit-identifier
| spec-morphism

spec-morphism `::=` **`morphism`** spec-term **`->`** spec-term name-map

A morphism is a formal mapping between two closed specs that describes exactly how one is translated or extended into the other.

Restriction. When used as a morphism-term, the elaboration of a unit-identifier must yield a spec morphism.

Restriction ("proof obligations"). Given spec-morphism `morphism` *`S`* `->` *`T`* `{` *`M`* `}`, let *`S′`* be the result of elaborating `translate` *`S`* `by {` *`M`* `}`, and let *`T′`* be the result of elaborating *`T`*. Then, first, each type-name or op-name introduced in *`S′`* must also be introduced in *`T′`*. Further, no type-name or op-name originating from a library spec may have been subject to translation. Finally, each axiom in *`S′`* must be a theorem that follows from the axioms of *`T′`*. Collectively, the axioms in *`S′`* are known as the *proof obligations* engendered by the morphism. They are the formal expression of the requirement that the step from *`S′`* to *`T′`* is a proper refinement.

2.3.4. Diagrams

diagram-term `::=`

```
      unit-identifier
    | diagram-form

diagram-form ::= diagram { diagram-element { , diagram-element }* }

diagram-element ::=
      diagram-node
    | diagram-edge

diagram-node ::= name +-> spec-term

diagram-edge ::= name : name -> name +-> morphism-term
```

Restriction. When used as a diagram-term, the elaboration of a unit-identifier must yield a diagram.

Restriction. In a diagram, the first name of each diagram-node and diagram-edge must be unique (i.e., not be used more than once in that diagram). Further, for each diagram-edge `E : ND -> NC +-> M`, there must be diagram-nodes `ND +-> D` and `NC +-> C` of the diagram such that, after elaboration, `M` is a morphism from `D` to `C`.

Sample diagram:

```
diagram {
   A          +-> /Coll/Lattice,
   B          +-> /Coll/LatticeWithTop,
   m : A -> B +-> /Coll/AddTop,
   C          +-> Routing#Basis,
   i : A -> C +-> morphism /Coll/Lattice ->
                                Routing#Basis {}
}
```

The result of elaborating a diagram-form is the categorical diagram whose nodes are labeled with the specs and whose edges are labeled with the morphisms that result from elaborating the corresponding spec-terms and morphism-terms.

2.3.5. Target Code Terms

```
target-code-term ::=
```

generate target-language-name spec-term [**in** string-literal]

target-language-name ::= **c** | **java** | **lisp**

Sample target-code-term:

```
generate lisp /Vessel#Contour
                  in "C:/Projects/Vessel/Contour.lisp"
```

The elaboration of a target-code-term for a correct spec-term generates code in the language suggested by the target-language-name (currently only C, Java, and Common Lisp); see further the Specware User Manual.

2.3.6. Proof Terms

proof-term ::=
  prove claim-name **in** spec-term
    [**with** prover-name]
    [**using** **{** claim-list **}**]
    [**options** prover-options]

prover-name ::= **snark**

claim-list ::= claim-name { **,** claim-name }*

prover-options ::= string-literal

Restriction. The claim-names must occur as claim-names in the spec that results from elaborating the spec-term.

Sample proof-term:

```
prove Effect in translate A by {Counter +-> Nat}
                     options "(use-paramodulation t)"
```

The elaboration of a proof-term invokes the prover suggested by the prover-name (currently only SNARK). The property to be proved is the claim of the first claim-name; the claim-list lists the hypotheses (assumptions) that may be used in the proof. The prover-options are prover-specific and are not further described here. For details, see the Specware User Manual.

2.4. Declarations

```
declaration ::=
        import-declaration
      | type-declaration
      | op-declaration
      | definition

definition ::=
        type-definition
      | op-definition
      | claim-definition

equals ::= is | =
```

Sample declarations:

```
import Lookup
type Key
op present : Database * Key -> Boolean
type Key = String
def present(db, k) = embed? Some (lookup (db, k))
axiom norm_idempotent is fa(x) norm (norm x) = norm x
```

2.4.1. Import-declarations

A spec may contain one or more import-declarations. On elaboration, these are "expanded". The effect is as if the bodies of these imported specs (themselves in elaborated form, which means that all import-declarations have been expanded, all translations performed and all shorthand employs of names have been resolved to full names, after which only declarations or definitions of types, ops and claims are left) is inserted in place in the receiving spec.

For example, the result of

```
spec
  import spec
           type A.Z
           op b : Nat -> Z
         end
  type A.Z = String
  def b = toString
endspec
```

is this “expanded” spec:

```
spec
  type A.Z
  op b : Nat -> A.Z
  type A.Z = String
  def b = toString
endspec
```

For this to be correct, the imported specs must be correct by themselves; in addition, the result of expanding them in place must result in a correct spec.

There are a few restrictions, which are meant to catch unintentional naming mistakes. First, if two different imported specs each introduce a type or op with the same (full) name, the introductions must be identical declarations or definitions, or one may be a declaration and the other a “compatible” definition. For example, given

```
S1 = spec op e : Integer end
S2 = spec op e : Char end
S3 = spec def e = 0 end
```

the specs `S1` and `S3` can be imported together, but all other combinations of two or more co-imported specs are incorrect. This restriction is in fact a special case of the general requirement that import expansion must result in a correct spec. Secondly, a type-name introduced in any of the imported specs cannot be re-introduced in the receiving spec except for the case of an “imported” declaration together with a definition in the receiving spec. Similarly for op-names, with the addition that an op-definition in the receiving spec must be compatible with an op-declaration for the same name in an imported spec. The latter is again a special case of the general requirement that import expansion must result in a correct spec.

What is specifically excluded by the above, is giving a definition of a type or op in some spec, import it, and then redefining or declaring that type or op with the same full name in the receiving spec.

import-declaration `::=` **`import`** spec-term

Sample import-declarations

```
import Lookup
```

An import-declaration is contained in some spec-form, and to elaborate that spec-form the spec-term of the import-declaration is elaborated first, giving some spec *S*. The import-declaration has then the effect as if the declarations of the imported spec *S* are expanded in place. This cascades: if spec *A* imports *B*, and spec *B* imports *C*, then effectively spec *A* also imports *C*. An important difference with earlier versions of Specware than version 4 is that multiple imports of the same spec have the same effect as a single import.

If spec *A* is imported by *B*, each model of *B* is necessarily a model of *A* (after "forgetting" any names newly introduced by *B*). So *A* is then refined by *B*, and the morphism from *A* to *B* is known as the "import morphism". As it does not involve translation of type-names or op-names, it can be denoted by `morphism A -> B {}`.

2.4.2. Type-declarations

type-declaration `::=` **`type`** type-name `[` formal-type-parameters `]`

formal-type-parameters `::=` local-type-variable `|` **`(`** local-type-variable-list **`)`**

local-type-variable `::=` name

local-type-variable-list `::=` local-type-variable `{` **`,`** local-type-variable `}*`

Restriction. Each local-type-variable of the formal-type-parameters must be a different name.

Sample type-declarations:

```
type Date
```

```
type Array a
type Map(a, b)
```

Every type-name used in a spec must be declared (in the same spec or in an imported spec, included the “built-in” specs that are always implicitly imported). A type-name may have *type parameters*. Given the example type-declarations above, some valid type-descriptors that can be used in this context are `Array Date`, `Array (Array Date)` and `Map (Nat, Boolean)`.

In a model of the spec, a type is assigned to each unparameterized type-name, while an infinite *family* of types is assigned to parameterized type-names “indexed” by tuples of types, that is, there is one family member, a type, for each possible assignment of types to the local-type-variables. So for the above example type-declaration of `Array` one type must be assigned to `Array Nat`, one to `Array Boolean`, one to `Array (Array Date)`, and so on. These assigned types could all be the same type, or perhaps all different, as long as the model respects typing.

2.4.3. Type-definitions

type-definition `::=` **`type`** type-name `[` formal-type-parameters `]` equals type-descriptor

Sample type-definitions:

```
type Date = {year : Nat, month : Nat, day : Nat}
type Array a = List a
type Map(a, b) = (Array (a * b) | key_uniq?)
```

In each model, the type assigned to the type-name must be the same as the right-hand-side type-descriptor. For parameterized types, this extends to all possible assignments of types to the local-type-variables, taking the right-hand type-descriptors as interpreted under each of these assignments. So, for the example, `Map(Nat, Char)` is the same type as `(Array (Nat * Char) | key_uniq?)`, and so on.

With *recursive* type-definitions, there are additional requirements. For example, consider

```
type Stack a =
  | Empty
  | Push {top : a, pop : Stack a}
```

This means that for each type a there is a value `Empty` of type `Stack a`, and further a function `Push` that maps values of type `{top : a, pop : Stack a}` to `Stack a`. Furthermore, the type assigned to `Stack a` must be such that all its inhabitants can be constructed *exclusively* and *uniquely* in this way: there is one inhabitant `Empty`, and all others are the result of a `Push`. Finally -- this is the point -- the type in the model must be such that its inhabitants can be constructed this way in *a finite number of steps*. So there can be no “bottom-less” stacks: deconstructing a stack using

```
def fa(a) hasBottom? (s : Stack a) : Boolean =
  case s of
     | Empty -> true
     | Push {top, pop = rest} -> hasBottom? rest
```

is a procedure that is guaranteed to terminate, always resulting in `true`.

In general, type-definitions generate implicit axioms, which for recursive definitions imply that the type is not “larger” than necessary. In technical terms, in each model the type is the least fixpoint of a recursive domain equation.

2.4.4. Op-declarations

op-declaration `::=` **`op`** op-name `[` fixity `]` **`:`** type-scheme

fixity `::=` associativity priority

associativity `::=` **`infixl`** `|` **`infixr`**

priority `::=` nat-literal

type-scheme `::=` `[` type-variable-binder `]` type-descriptor

type-variable-binder `::=` **`fa (`** local-type-variable-list **`)`**

Sample op-declarations:

```
op usage : String
```

```
op o infixl 24 : fa(a,b,c) (b -> c) * (a -> b) -> a -> c
```

An op-declaration introduces an op-name with an associated type. The type can be "monomorphic", like `String`, or "polymorphic" (indicated by a type-variable-binder). In the latter case, an indexed family of values is assigned to parameterized type-names "indexed" by tuples of types, that is, there is one family member, a typed value, for each possible assignment of types to the local-type-variables of the type-variable-binder, and the type of that value is the result of the corresponding substitution of types for local-type-variables on the polymorphic type of the op. In the example above, the declaration of polymorphic `o` can be thought of as introducing a family of (fictitious) ops, one for each possible assignment to the local-type-variables `a`, `b` and `c`:

$o_{\texttt{Nat,String,Char}}$ `: (String -> Char) * (Nat -> String) -> Nat -> Char`

$o_{\texttt{Nat,Nat,Boolean}}$ `: (Nat -> Boolean) * (Nat -> Nat) -> Nat -> Boolean`

$o_{\texttt{Char,Boolean,Nat}}$ `: (Boolean -> Nat) * (Char -> Boolean) -> Char -> Nat`

and so on. Any op-definition for `o` must be likewise accommodating.

Only binary ops (those having some type *`S * T -> U`*) may be declared with a fixity. When declared with a fixity, the op-name may be used in infix notation, and then it is called an infix-operator. For `o` above, this means that `o(f, g)` and `f o g` may be used, interchangeably, with no difference in meaning. If the associativity is `infixl`, the infix-operator is called *left-associative*; otherwise, if the associativity is `infixr`, it is called *right-associative*. If the priority is `priority` *`N`*, the operator is said to have *priority* *`N`*. The nat-literal *`N`* stands for a natural number; if infix-operator *`O1`* has priority *`N1`*, and *`O2`* has priority *`N2`*, with *`N1`* < *`N2`*, we say that *`O1`* has *lower priority* than *`O2`*, and that *`O2`* has *higher priority* than (or *takes priority over*) *`O1`*. For the role of the associativity and priority, see further at *Infix-applications*.

2.4.5. Op-definitions

op-definition `::=`
    `def` `[` type-variable-binder `]` formal-expression `[` **`:`** type-descriptor `]` equals
        expression

formal-expression `::=` op-name `|` formal-application

formal-application `::=` formal-application-head formal-parameter

formal-application-head `::=` op-name `|` formal-application formal-parameter ::= closed-pattern

Sample op-definitions:

```
def usage = "Usage: Lookup key [database]"

def fa(a,b,c) o(f : b -> c, g: a -> b) : a -> c =
  fn (x : a) -> f(g x)

def o(f, g) x = f(g x)
```

Restriction. See the restriction under *Op-declarations* on redeclaring/redefining ops.

Note that a formal-expression always contains precisely one op-name, which is the op *being defined* by the op-definition. Note further that the formal-application of an op-definition always uses prefix notation, also for infix-operators.

An op can be defined without having been declared. In that case the op-definition generates an implicit op-declaration for the op, provided a monomorphic type for the op can be unambiguously determined from the op-definition together with the uses of the op in applications and other contexts. In general, typing information on ops may be omitted, but sufficient information must be supplied when used, so that all expressions can be assigned a type in the context in which they occur while uniquely associating the ops with op-declarations or op-definitions. If two different associations both give type-correct specs, the spec is ambiguous and incorrect.

As for op-definitions, the presence of a type-variable-binder signals that the op being defined is polymorphic. Note that the optional type annotation in an op-definition can not be a polymorphic type-scheme, unlike for op-declarations. For example, the following is ungrammatical:

```
def o : fa(a,b,c) (b -> c) * (a -> b) -> a -> c =
  fn (f, g) -> fn (x) -> f(g x)
```

The presumably intended effect is achieved by

```
def fa(a,b,c) o : (b -> c) * (a -> b) -> a -> c =
  fn (f, g) -> fn (x) -> f(g x)
```

In a model of the spec, an indexed family of typed values is assigned to a polymorphic op, with one family member for each possible assignment of types to the local-type-variables of the type-variable-binder, and the type of that value is the result of the corresponding type-instantiation for the polymorphic type of the op.

Thus, we can reduce the meaning of a polymorphic op-definition to a family of (fictitious) monomorphic op-definitions.

An op-definition with formal-prefix-application

```
def H P = E
```

in which *H* is a formal-application-head, *P* is a formal-parameter and *E* an expression, is equivalent to the op-definition

```
def H = fn P -> E
```

For example,

```
def o (f, g) x = f(g x)
```

is equivalent to

```
def o (f, g) = fn x -> f(g x)
```

which in turn is equivalent to

```
def o = fn (f, g) -> fn x -> f(g x)
```

By this deparameterizing transformation for each formal-parameter, an equivalent unparameterized op-definition is reached. The semantics is described in terms of such op-definitions.

In each model, the typed value assigned to the op being defined must be the same as the value of the right-hand-side expression. For polymorphic op-definitions, this extends to all possible assignments of types to the local-type-variables.

An op-definition can be thought of as a special notation for an axiom. For example,

```
def fa(a) double (x : a) = (x, x)
```

can be thought of as standing for:

```
op double : fa(a) a -> a * a

axiom double_def is
  type fa(a) fa(x : a) double x = (x, x)
```

In fact, Specware generates such axioms for use by provers. But in the case of recursive definitions, this form of axiomatization does not adequately capture the meaning. For example,

```
def f (n : Nat) : Nat = 0 * f n
```

is an improper definition, while

```
axiom f_def is
    fa(n : Nat) f n = 0 * f n
```

characterizes the function that maps every natural number to 0. The issue is the following. Values in models can not be *undefined* and functions assigned to ops must be *total*. But in assigning a meaning to a recursive op-definition, we -- temporarily -- allow *undefined* and partial functions (functions that are not everywhere defined on their domain type) to be assigned to recursively defined ops. In the thus extended class of models, the recursive ops must be the least-defined solution to the "axiomatic" equation (the least fixpoint as in domain theory), given the assignment to the other ops. For the example of `f` above this results in the everywhere undefined function, since 0 times *undefined* is *undefined*. If the solution results in an undefined value or a function that is not total (or for higher-order functions, functions that may return non-total functions, and so on), the op-definition is improper. Specware 4.1 does not attempt to detect this condition or generate proof obligations for showing its absence.

Functions that are determined to be the value of an expression, but that are not assigned to ops, need not be total, but the context must enforce that the function can not be applied to values for which it is undefined. Otherwise, the spec is incorrect.

2.4.6. Claim-definitions

claim-definition `::=` claim-kind claim-name equals claim

claim-kind `::=` **`axiom`** `|` **`theorem`** `|` **`conjecture`**

claim-name `::=` name

claim `::=` `[` type-quantification `]` expression

type-quantification `::=` **`type`** type-variable-binder

Sample claim-definitions:

```
axiom norm_idempotent is
  norm o norm = norm
```

```
theorem o_assoc is
  type fa(a,b,c,d) fa(f : c -> d, g : b -> c, h : a -> b)
    f o (g o h) = (f o g) o h

conjecture pivot_hold is
  let p = pivot hold in
    fa (n : {n : Nat | n < p}) ~(hold n = hold p)
```

Restriction. The type of the claim must be `Boolean`.

When a type-quantification is present, the claim is polymorphic. The claim may be thought of as standing for an infinite family of monomorphic claims, one for each possible assignment of types to the local-type-variables.

The claim-kind `theorem` should only be used for claims that have actually been proved to follow from the (explicit or implicit) axioms. In other words, giving them axiom status should not change the class of models. Theorems can be used by provers.

Conjectures are meant to represent proof obligations that should eventually attain theoremhood. Like theorems, they can be used by provers.

The Specware system passes on the claim-name of the claim-definition with the claim for purposes of identification. Both may be transformed to fit the requirements of the prover, and appear differently there. Not all claims can be faithfully represented in all provers, and even when they can, the logic of the prover may not be up to dealing with them.

Remark. It is a common mistake to omit the part “claim-name equals” from a claim-definition. A defensive style against this mistake is to have the claim always start on a new text line. This is additionally recommended because it may become required in future revisions of Metaslang.

2.5. Type-descriptors

```
type-descriptor ::=
      type-sum
    | type-arrow
    | slack-type-descriptor
```

```
slack-type-descriptor ::=
        type-product
      | tight-type-descriptor

tight-type-descriptor ::=
        type-instantiation
      | closed-type-descriptor

closed-type-descriptor ::=
        type-name
      | local-type-variable
      | type-record
      | type-restriction
      | type-comprehension
      | type-quotient
      | ( type-descriptor )
```

(The distinctions “slack-”, “tight-” and “closed-” before “type-descriptor” have no semantic significance. The distinction merely serves the purpose of diminishing the need for parenthesizing in order to avoid grammatical ambiguities.)

Sample type-descriptors:

```
| Point XYpos | Line XYpos * XYpos
List String * Nat -> Option String
a * Order a * a
PartialFunction (Key, Value)
Key
a
{center : XYpos, radius : Length}
(Nat | even)
{k : Key | present (db, k)}
Nat / (fn (m, n) -> m rem 3 = n rem 3)
(Nat * Nat)
```

The meaning of a parenthesized type-descriptor `(`*T*`)` is the same as that of the enclosed type-descriptor *T*.

The various other kinds of type-descriptors not defined here are described each in their following respective sections, with the exception of local-type-variable, whose (lack of) meaning as a type-descriptor is described below.

Restriction. A local-type-variable may only be used as a type-descriptor if it occurs in the scope of a formal-type-parameters or type-variable-binder in which it is introduced.

Disambiguation. A single name used as a type-descriptor is a local-type-variable when it occurs in the scope of a formal-type-parameters or type-variable-binder in which it is introduced, and then it identifies the textually most recent introduction. Otherwise, the name is a type-name.

A local-type-variable used as a type-descriptor has no meaning by itself, and where relevant to the semantics is either "indexed away" (for parameterized types) or "instantiated away" (when introduced in a formal-type-parameters or type-variable-binder) before a meaning is ascribed to the construct in which it occurs. Textually, it has a scope just like a plain local-variable.

2.5.1. Type-sums

type-sum `::=` type-summand `{` type-summand `}*`

type-summand `::=` **|** constructor `[` slack-type-descriptor `]`

constructor `::=` name

Sample type-sum:

```
| Point XYpos | Line XYpos * XYpos
```

Restriction. The constructors of a type-sum must all be different names.

The ordering of the type-summands has no significance: `| Zero | Succ Peano` denotes the same "sum type" as `| Succ Peano | Zero`.

A type-sum denotes a *sum type*, which is a type that is inhabited by "tagged values". A tagged value is a pair (*C*, *v*), in which *C* is a constructor and *v* is a typed value.

A type-sum introduces a number of embedders, one for each type-summand. In the discussion, we omit the optional `embed` keyword of the embedders. The embedders are similar to ops, and are explained as if they were ops, but note the Restriction specified under *Structors*.

For a type-sum *T* with type-summand *C S*, in which *C* is a constructor and *S* a type-descriptor, the corresponding pseudo-op introduced is typed as follows:

```
op C : S -> T
```

It maps a value *v* of type *S* to the tagged value (*C*, *v*). If the type-summand is a single *parameter-less* constructor (the slack-type-descriptor is missing), the pseudo-op introduced is typed as follows:

```
op C : T
```

It denotes the tagged value (*C*, ()), in which () is the inhabitant of the unit type (see under *Type-records*).

The sum type denoted by the type-sum then consists of the union of the ranges (for parameter-less constructors the values) of the pseudo-ops for all constructors.

The embedders are individually, jointly and severally *injective*, and jointly *surjective*.

This means, first, that for any pair of constructors *C1* and *C2* of *any* type-sum, and for any pair of values *v1* and *v2* of the appropriate type (to be omitted for parameter-less constructors), the value of *C1 v1* is only equal to *C2 v2* when *C1* and *C2* are the same constructor of the *same* sum type, and *v1* and *v2* (which then are either both absent, or else must have the same type) are both absent or are the same value. In other words, whenever the constructors are different, or are from different type-sums, or the values are different, the results are different. (The fact that synonymous constructors of different types yield different values already follows from the fact that values in the models are typed.)

Secondly, for any value *u* of any sum type, there is a constructor *C* of that sum type and a value *v* of the appropriate type (to be omitted for parameter-less constructors), such that the value of *C v* is *u*. In other words, all values of a sum type can be constructed with an embedder.

For example, consider

```
type Peano =
  | Zero
  | Succ Peano

type Unique =
  | Zero
```

This means that there is a value `Zero` of type `Peano`, and further a function `Succ` that maps values of type `Peano` to type `Peano`. Then `Zero` and `Succ n` are guaranteed to be different, and each value of type `Peano` is either `Zero : Peano`, or expressible in the form `Succ (n : Peano)` for a suitable expression `n`. The expressions `Zero : Peano` and `Zero : Unique` denote different, entirely unrelated, values. (Note that `Unique` is *not* a subtype of `Peano`. Subtypes of a type can only be made with a type-restriction, for instance as in `(Peano | embed? Zero)`.) For recursively defined type-sums, see also the discussion under *Type-definitions*.

Note. Although the sum types `| Mono` and `| Mono ()` have exactly the same set of inhabitants when considered as untyped values, these two types are different, and the pseudo-ops they introduce have different types, only the second of which is a function type:

```
Mono : | Mono
```

```
Mono : () -> | Mono ()
```

2.5.2. Type-arrows

type-arrow `::=` arrow-source **`->`** type-descriptor

arrow-source `::=` type-sum `|` slack-type-descriptor

Sample type-arrow:

```
(a -> b) * b -> List a -> List b
```

In this example, the arrow-source is `(a -> b) * b`, and the (target) type-descriptor `List a -> List b`.

The *function type* *`S -> T`* is inhabited by precisely all *partial or total* functions from *`S`* to *`T`*. That is, function *`f`* has type *`S -> T`* if, and only if, for each value *`x`* of type *`S`* such that the value of *`f x`* is defined, that value has type *`T`*. Functions can be constructed with lambda-forms, and be used in applications.

In considering whether two functions (of the same type) are equal, only the meaning on the domain type is relevant. Whether a function is undefined outside its domain type, or might return some value of some type, is immaterial to the semantics of Metaslang. (For a type-correct spec, the difference is unobservable.)

2.5.3. Type-products

type-product `::=` tight-type-descriptor **`*`** tight-type-descriptor `{` **`*`** tight-type-descriptor `}*`

Sample type-product:

```
(a -> b) * b * List a
```

Note that a type-product contains at least two constituent tight-type-descriptors.

A type-product denotes a *product type* that has at least two "component types", represented by its tight-type-descriptors. The ordering of the component types is significant: unless `S` and `T` are the same type, the product type `S * T` is different from the type `T * S`. Further, the three types `(S * T) * U`, `S * (T * U)` and `S * T * U` are all different; the first two have two component types, while the last one has three. The inhabitants of the product type $T_1 * T_2 * ... * T_n$ are precisely all *n*-tuples (v_1, v_2, ... , v_n), where each v_i has type `T\`$_{i\backslash}$, for `i` = 1, 2, ... , *n*. Values of a product type can be constructed with tuple-displays, and component values can be extracted with tuple-patterns as well as with projectors.

2.5.4. Type-instantiations

type-instantiation `::=` type-name actual-type-parameters

actual-type-parameters `::=` closed-type-descriptor `|` **(** proper-type-list **)**

proper-type-list `::=` type-descriptor **,** type-descriptor `{` **,** type-descriptor `}*`

Sample type-instantiation:

```
Map (Nat, Boolean)
```

Restriction. The type-name must have been declared or defined as a parameterized type (see *Type-declarations*), and the number of type-descriptors in the actual-type-parameters must match the number of local-type-variables in the formal-type-parameters of the type-declaration and/or type-definition.

The type-descriptor represented by a type-instantiation is the type assigned for the combination of types of the actual-type-parameters in the indexed family of types for the type-name of the type-instantiation.

2.5.5. Type-names

type-name `::=` qualifiable-name

Sample type-names:

```
Key
Calendar.Date
```

Restriction. At the spec level, a type-name may only be used if there is a type-declaration and/or type-definition for it in the current spec or in some spec that is imported (directly or indirectly) in the current spec. If there is a unique qualified-name for a given unqualified ending, the qualification may be omitted for a type-name used as a type-descriptor.

The type of a type-name is the type assigned to it in the model. (In this case, the context can not have superseded the original assignment.)

2.5.6. Type-records

type-record `::=` **`{`** `[` field-typer-list `]` **`}`** `|` **`( )`**

field-typer-list `::=` field-typer `{` **`,`** field-typer `}*`

field-typer `::=` field-name **`:`** type-descriptor

field-name `::=` name

Sample type-record:

```
{center : XYpos, radius : Length}
```

Restriction. The field-names of a type-record must all be different.

Note that a type-record contains either no constituent field-typers, or else at least two.

A type-record is like a type-product, except that the components, called “fields”, are identified by name instead of by position. The ordering of the field-typers has no significance: `{center : XYpos, radius : Length}` denotes the same *record type* as `{radius : Length, center : XYpos}`. Therefore we assume in the following, without loss of generality, that the fields are ordered lexicographically according to their field-names (as in a dictionary: `a` comes before `ab` comes before `b`) using some fixed collating order for all marks that may comprise a name. Then each field of a record type with n fields has a *position* in the range 1 to n. The inhabitants of the record type $\{F_1 : T_1, F_2 : T_2, \ldots, F_n : T_n\}$ are precisely all n-tuples (v_1, v_2, ... , v_n), where each v_i has type T_i, for i = 1, 2, ... , n. The field-names of that record type are the field-names F_1, ... , F_n, and, given the lexicographic ordering, field-name F_i *selects* position `i`, for `i` = 1, 2, ... , `n`. Values of a record type can be constructed with record-displays, and field values can be extracted with record-patterns and (as for product types) with projectors.

For the type-record `{}`, which may be equivalently written as `()`, the record type it denotes has zero components, and therefore no field-names. This zero-component type has precisely one inhabitant, and is called the *unit type*. The unit type may equally well be considered a product type, and is the only type that is both a product and a record type.

2.5.7. Type-restrictions

type-restriction `::=` **(** slack-type-descriptor **|** expression **)**

Sample type-restriction:

```
(Nat | even)
```

Restriction. In a type-restriction `(T | P)`, the expression `P` must be a predicate on the type `T`, that is, `P` must be a function of type `T -> Boolean`.

Note that the parentheses in `(T | P)` are mandatory.

The inhabitants of type `(T | P)` are precisely the inhabitants of type `T` that satisfy the predicate `P`, that is, they are those values `v` for which the value of `P v` is `true`.

If `P1` and `P2` are the same function, then `(T | P1)` and `(T | P2)` are equivalent, that is, they denote the same type. Furthermore, `(T | fn _ -> true)` is equivalent to `T`.

The type `(T | P)` is called a *subtype* of *supertype* `T`. Values can be shuttled between a subtype and its supertype and vice versa with relaxators and restrict-expressions; see also *Relax-patterns*.

Metaslang does not require the explicit use of a relaxator to relax an expression from a subtype to its supertype if the context requires the latter. Implicit relaxation will take place when needed. For example, in the expression `-1` the nat-literal `1` of type `Nat` is implicitly relaxed to type `Integer` to accommodate the unary negation operator `-`, which has type `Integer -> Integer`.

Likewise, Metaslang does not require the explicit use of a restrict-expression to restrict an expression from a type to a subtype if the context requires the latter.

Implicit restriction will take place when needed. For example, in the expression `7 div 2` the nat-literal `2` of type `Nat` is implicitly restricted to type `PosNat`, a subtype of `Nat`, to accommodate the division operator `div`, whose second argument has type `PosNat`. But note that implicit restriction engenders the same proof obligation as results when using an explicit restrict-expression.

2.5.8. Type-comprehensions

type-comprehension `::=` **{** annotated-pattern **|** expression **}**

Sample type-comprehension:

```
{n : Nat | even n}
```

Restriction. In a type-comprehension `{`*P* `:` *T* `|` *E*`}`, the expression *E* must have type `Boolean`.

Type-comprehensions provide an alternative notation for type-restrictions that is akin to the common mathematical notation for set comprehensions. The meaning of type-comprehension `{`*P* `:` *T* `|` *E*`}` is the same as that of the type-restriction `(`*T* `| fn` *P* `->` *E*`)`. So the meaning of the example type-comprehension above is `(Nat | fn n -> even n)`.

2.5.9. Type-quotients

type-quotient `::=` closed-type-descriptor `/` closed-expression

Sample type-quotient:

```
Nat / (fn (m, n) -> m rem 3 = n rem 3)
```

Restriction. In a type-quotient *T* `/` *Q*, the expression *Q* must be a (binary) predicate on the type *T* `*` *T* that is an equivalence relation, as explained below.

Equivalence relation. Call two values *x* and *y* of type *T* “*Q*-related” if (*x*, *y*) satisfies *Q*. Then *Q* is an *equivalence relation* if, for all values *x*, *y* and *z* of type *T*, *x* is *Q*-related to itself, *y* is *Q*-related to *x* whenever *x* is *Q*-related to *y*, and *x* is *Q*-related to *z* whenever *x* is *Q*-related to *y* and *y* is *Q*-related to *z*. The equivalence relation *Q* then partitions the inhabitants of *T* into *equivalence classes*, being the maximal subsets of `T` containing mutually `Q`-related members. These equivalence classes will be called "`Q`-equivalence classes".

The inhabitants of the *quotient type* `T / Q` are precisely the `Q`-equivalence classes into which the inhabitants of `T` are partitioned by `Q`. For the example above, there are three equivalence classes of natural numbers leaving the same remainder on division by 3: the sets {0, 3, 6, ...}, {1, 4, 7, ...} and {2, 5, 8, ...}, and so the quotient type has three inhabitants.

2.6. Expressions

```
expression ::=
          lambda-form
        | case-expression
        | let-expression
        | if-expression
        | quantification
        | annotated-expression
        | tight-expression

tight-expression ::=
          application
        | restrict-expression
        | closed-expression

closed-expression ::=
          op-name
        | local-variable
        | literal
        | field-selection
        | tuple-display
        | record-display
        | sequential-expression
        | list-display
        | structor
        | ( expression )
        | ( inbuilt-infix )

inbuilt-infix ::= => | || | && | = | ~= | <<
```

(The distinctions tight- and closed- for expressions lack semantic significance, and merely serve the purpose of avoiding grammatical ambiguities.)

Sample expressions:

```
fn (s : String) -> s ^ "."
case z of {re = x, im = y} -> {re = x, im = -y}
let x = x + 1 in f(x, x)
if x <= y then x else y
fa(x,y) (x <= y)  <=>  ((x < y) or (x = y))
f(x, x)
[] : List Arg
<=>
x
3260
z.re
("George", Poodle : Dog, 10)
{name = "George", kind = Poodle : Dog, age = 10}
(writeLine "key not found"; embed Missing)
["Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"]
project 2
(n + 1)
```

Restriction. Like all polymorphic or type-ambiguous constructs, an expression can only be used in a context if its type can be inferred uniquely, given the expression and the context. This restriction will not be repeated for the various kinds of expressions defined in the following subsections.

The meaning of a parenthesized expression `(`*E*`)` is the same as that of the enclosed expression *E*. The meaning of a parenthesized inbuilt-infix `(`*I*`)` is the same as that of the lambda-form `fn (x,y) -> x` *I* `y`.

The various other kinds of expressions not defined here are described each in their following respective sections, with the exception of local-variable, whose meaning as an expression is described below.

Restriction. A local-variable may only be used as an expression if it occurs in the scope of the local-variable-list of a quantification or of a variable-pattern in which it is introduced.

Disambiguation. A single name used as an expression is a local-variable when it occurs in the scope of a local-variable-list or variable-pattern in which a synonymous local-variable is introduced, and then it identifies the textually most recent introduction. Otherwise, the name is an op-name or an embedder; for the disambiguation between the latter two, see *Embedders*.

A local-variable used as an expression has the typed value assigned to it in the environment.

2.6.1. Lambda-forms

lambda-form `::=` **`fn`** match

Sample lambda-form:

```
fn (s : String) -> s ^ "."
```

The value of a lambda-form is a partial or total function. If the value determined for a lambda-form as described below is not a total function, the context must enforce that the function can not be applied to values for which it is undefined. Otherwise, the spec is incorrect. Specware 4.1 does not attempt to generate proof obligations for establishing this.

The type of a lambda-form is that of its match. The meaning of a given lambda-form of type `S -> T` is the function `f` mapping each inhabitant `x` of `S` to a value `y` of type `T`, where `y` is the return value of `x` for the match of the lambda-form. If the match accepts each `x` of type `S` (for acceptance and return value, see the section on *Matches*) function `f` is total; otherwise it is partial, and undefined for those values `x` rejected.

In case of a recursive definition, the above procedure may fail to determine a value for `y`, in which case function `f` is not total, but undefined for `x`.

2.6.2. Case-expressions

case-expression `::=` **`case`** expression **`of`** match

Sample case-expressions:

```
case z of {re = x, im = y} -> {re = x, im = -y}

case s of
   | Empty -> true
   | Push {top = _, pop = rest} -> hasBottom? rest
```

The value of a case-expression `case E of M` is the same as that of the application `(fn M) (E)`.

2.6.3. Let-expressions

let-expression `::=` **`let`** let-bindings **`in`** expression

let-bindings `::=` recless-let-binding `|` rec-let-binding-sequence

recless-let-binding `::=` pattern equals expression

rec-let-binding-sequence `::=` rec-let-binding `{` rec-let-binding `}*`

rec-let-binding `::=`
    `def` name formal-parameter-sequence `[` **`:`** type-descriptor `]` equals expression

formal-parameter-sequence `::=` formal-parameter `{` formal-parameter `}*`

Sample let-expressions:

```
let x = x + e in f(x, x)
let def f x = x + e in f (f x)
```

In the case of a recless-let-binding (recless = recursion-less), the value of the let-expression `let` *`P`* `=` *`A`* `in` *`E`* is the same as that of the application `(fn` *`P`* `->` *`E`*`)` `(`*`A`*`)`. For the first example above, this amounts to `f(x + e, x + e)`. Note that `x = x + e` is not interpreted as a recursive definition.

In case of a rec-let-binding-sequence (rec = recursive), the rec-let-bindings have the role of "local" op-definitions; that is, they are treated exactly like op-definitions except that they are interpreted in the local environment instead of the global model. For the second example above, this amounts to `(x + e) + e`. (If `e` is a local-variable in this scope, the definition of `f` can not be "promoted" to an op-definition, which would be outside the scope binding `e`.) A spec with rec-let-bindings can be transformed into one without such by creating op-definitions for each rec-let-binding that take additional arguments, one for each of the local-variables referenced. For the example, in which `f` references local-variable `e`, the op-definition for the "extended" op `f`$^+$ would be `def f`$^+$` e x = x + e`, and the let-expression would become `f`$^+$` e (f`$^+$` e x)`. The only difference in meaning is that the models of the transformed spec assign a value to the newly introduced op-name `f`$^+$.

Note that the first occurrence of `x` in the above example of a rec-let-binding is a variable-pattern and the second-occurrence is in its scope; the third and last occurrence of `x`, however, is outside the scope of the first `x` and identifies an op or local-variable `x` introduced elsewhere. So, without change in meaning, the rec-let-binding can be changed to:

```
let def f xena = xena + e in f (f x)
```

2.6.4. If-expressions

if-expression `::=` **`if`** expression **`then`** expression **`else`** expression

Sample if-expression:

```
if x <= y then x else y
```

The value of an if-expression `if` *B* `then` *T* `else` *F* is the same as that of the case-expression `case` *B* `of true -> (`*T*`) | false -> (`*F*`)`.

2.6.5. Quantifications

quantification `::=` quantifier **`(`** local-variable-list **`)`** expression

quantifier `::=` **`fa`** `|` **`ex`**

local-variable-list `::=` annotable-variable `{` **`,`** annotable-variable `}*`

annotable-variable `::=` local-variable `[` **`:`** type-descriptor `]`

local-variable `::=` name

Sample quantifications:

```
fa(x) norm (norm x) = norm x
ex(e : M) fa(x : M) x <*> e = x & e <*> x = x
```

Restriction. Each local-variable of the local-variable-list must be a different name.

Quantifications are non-constructive, even when the domain type is finitely enumerable. The main uses are in type-restrictions and type-comprehensions, and claims. The type of a quantification is `Boolean`. There are two kinds of quantifications: `fa`-quantifications (or “universal quantifications”; fa = for all), and `ex`-quantifications (or “existential quantifications”; ex = there exists).

The value of a `fa`-quantification `fa` *V E*, in which *V* is a local-variable-list and *E* is an expression, is determined as follows. Let *M* be the match *V* `->` *E*. If *M* has return value `true` for each value *x* in its domain (note that rejection cannot happen here), the value of the quantification is `true`; otherwise it is `false`.

The value of an `ex`-quantification `ex` *V E* is the same as that of the `fa`-quantification `~(fa` *V* `~(`*E*`))`.

Note that `fa` and `ex` must be followed by an opening parenthesis `(` . So `fa x (x = x)`, for example, is ungrammatical.

2.6.6. Annotated-expressions

annotated-expression `::=` tight-expression `:` type-descriptor

Restriction. In an annotated-expression *E* `:` *T*, the expression *E* must have type *T*.

Sample annotated-expression:

```
[] : List Arg
Positive : Sign
```

The value of an annotated-expression *E* `:` *T* is the value of *E*.

The type of some expressions is polymorphic. For example, for any type *T*, `[]` denotes the empty list of type `List` *T*. Likewise, constructors of parameterized sum types can be polymorphic, as the constructor `None` of

```
type Option a = | Some a | None
```

Further, overloaded constructors have an ambiguous type. By annotating such polymorphic or type-ambiguous expressions with a type-descriptor, their type can be disambiguated, which is required unless an unambiguous type can already be inferred from the context. Annotation, even when redundant, can further help to increase clarity.

2.6.7. Applications

application `::=` prefix-application `|` infix-application

prefix-application `::=` application-head actual-parameter application-head `::=` closed-expression | prefix-application

actual-parameter `::=` closed-expression

infix-application `::=` operand infix-operator operand

operand `::=` tight-expression

infix-operator `::=` op-name | inbuilt-infix

Sample applications:

```
f (x, x)
f x (g y)
x + 1
```

Restriction. An infix-operator, whether qualified or unqualified, can not be used without more as an actual-parameter or operand (and in the case of an inbuilt-infix, it can not be used without more as any other kind of expression either). To use an infix-operator in such cases, it must be enclosed in parentheses, as for example in the prefix-applications `foldl (+) 0` and `foldl ( *) 1` or the infix-application `(<) o ival`. Note the space between "`(`" and "`*`", since without space "`(*`" signals the start of a comment.

Restriction. An op-name can be used as an infix-operator only if it has been declared as such in an op-declaration (see under *Op-declarations*).

Disambiguation. An infix-application *`P M Q N R`*, in which *`P`*, *`Q`* and *`R`* are operands and *`M`* and *`N`* are infix-operators, is interpreted as either `(`*`P M Q`*`)` *`N R`* or *`P M`* `(`*`Q N R`*`)`. The choice is made as follows. If *`M`* has higher priority than *`N`*, or the priorities are the same but *`M`* is left-associative, the interpretation is `(`*`P M Q`*`)` *`N R`*. In all other cases the interpretation is *`P M`* `(`*`Q N R`*`)`. For example, given

```
op @ infixl 10: Nat * Nat -> Nat
op $ infixr 20: Nat * Nat -> Nat
```

the following interpretations hold:

```
1 $ 2 @ 3  =  (1 $  2) @ 3
1 @ 2 @ 3  =  (1 @  2) @ 3
1 @ 2 $ 3  =   1 @ (2  $ 3)
1 $ 2 $ 3  =   1 $ (2  $ 3)
```

Note that no type information is used in the disambiguation. If `(1 @ 2) $ 3` is type-correct but `1 @ (2 $ 3)` is not, the formula `1 @ 2 $ 3` is type-incorrect, since its interpretation is.

For the application of this disambiguation rule, the inbuilt-infix operators have fixity as suggested by the following pseudo-op-declarations:

```
op =>  infixr 13 : Boolean * Boolean -> Boolean
op ||  infixr 14 : Boolean * Boolean -> Boolean
op &&  infixr 15 : Boolean * Boolean -> Boolean
op =   infixr 20 : fa(a) a * a       -> Boolean
op ~=  infixr 20 : fa(a) a * a       -> Boolean
op <<  infix? ?? : {a:X, b:Y} * {a:X, c:Z} ->
                                          {a:X, b:Y, c:Z}
```

Restriction. In an application *H P*, in which *H* is an application-head and *P* an actual-parameter, the type of *P* must be some function type *S* `->` *T*, and then *H* must have the domain type *S*. The type of the whole application is then *T*.

The value of application *H P* is the value returned by function *H* for the argument value *P*.

The meaning of infix-application *P N Q*, in which *P* and *Q* are operands and *N* is an op-name, is the same as that of the prefix-application *N(P, Q)*.

The meaning of infix-application *P* `=>` *Q*, in which *P* and *Q* are operands, is the same as that of the if-expression `if` *P* `then` *Q* `else true`.

The meaning of infix-application *P* `||` *Q*, in which *P* and *Q* are operands, is the same as that of the if-expression `if` *P* `then true else` *Q*.

The meaning of infix-application *P* `&&` *Q*, in which *P* and *Q* are operands, is the same as that of the if-expression `if` *P* `then` *Q* `else false`.

The value of infix-application *P* `=` *Q*, in which *P* and *Q* are operands, is `true` if *P* and *Q* have the same value, and `false` otherwise. *P* and *Q* must have the same type, or else have types that are subtypes of the same supertype. In the latter case, the comparison is the same as for the values of the operands relaxed to the supertype, so, for example, the value of `(1:Nat)=(1:PosNat)` is `true`.

The meaning of infix-application *P* `~=` *Q*, in which *P* and *Q* are operands, is the same as that of the prefix-application `~(`*P* `=` *Q*`)`.

An infix-application *P* `<<` *Q* is also called a “record update”. In a record update *P* `<<` *Q*, in which *P* and *Q* are operands, *P* and *Q* must have record types, referred to as *S* and *T*, respectively. Moreover, for each field-name *F* these types *S* and *T* have in common, the field types for *F* in *S* and *T* must be the same, or be subtypes of the same supertype. The type of `P << Q` is then the record type *R* whose field-names are formed by the union of the field-names of *S* and *T*, where for each field-name *F* in that union, the type of field *F* in *R* is that of field *F* in *T* if *F* is a field of *T*, and otherwise the type of field *F* in *S*. Likewise, the value of `P << Q` is the record value of type *R* whose field value of each field *F* is that of field *F* in *Q* if *F* is a field of *T*, and otherwise the field value of field *F* in *P*. So, for example, the value of `{a=1, b=#z} << {a=2, c=true}` is `{a=2, b=#z, c=true}`: fields of the right-hand side operand take precedence over the left-hand side when present in both.

2.6.8. Restrict-expressions

restrict-expression `::=` **`restrict`** closed-expression closed-expression

Sample restrict-expression:

```
restrict posNat? (n+1)
```

Restriction. In a restrict-expression `restrict` *P E* the expression *P* must have function type *T* `-> Boolean` and the expression *E* must have type *T* for some *T*.

The type of a restrict-expression `restrict` *P E*, where *P* has type *T* `-> Boolean`, is the type `(`*T* `|` *P*`)`.

A restrict-expression `restrict` *P E* is a convenient notation for the let-expression `let relax` *P V* `=` *E* `in` *V*, where *V* is some unique fresh name, that is, it is any name that does not already occur in the spec, directly or indirectly through an import.

The use of this restrict-expression engenders a proof obligation that the value of *E* satisfies predicate *P*.

For example, assuming the definitions from the Base Library for `Nat`, the restrict-expression `restrict posNat? (n+1)` has type `PosNat`. The proof obligation here is that, in the context, `(n+1) > 0`.

The purpose of restrict-expressions is to make it explicit that an expression whose a-priori type is some supertype *T* actually is guaranteed (or required) to have subtype `(`*T* `|` *P*`)`. Note, however, that Metaslang does not require the explicit use of a restrict-expression to restrict an expression from a type to a subtype if the context requires the latter. Implicit restriction will take place when needed. For example, in the expression `7 div 2` the nat-literal `2` of type `Nat` is implicitly restricted to type

`PosNat`, a subtype of `Nat`, to accommodate the division operator `div`, whose second argument has type `PosNat`. But note that implicit restriction engenders the same proof obligation as results when using an explicit restrict-expression.

2.6.9. Op-names

op-name `::=` qualifiable-name

Sample op-names:

```
length
>=
DB_LOOKUP.Lookup
```

Restriction. An op-name may only be used if there is an op-declaration and/or op-definition for it in the current spec or in some spec that is imported (directly or indirectly) in the current spec. If there is a unique qualified-name for a given unqualified ending that is type-correct in the context, the qualification may be omitted for an op-name used as an expression. So overloaded ops may only be used as such when their type can be disambiguated in the context.

The value of an op-name is the value assigned to it in the model. (In this case, the context can not have superseded the original assignment.)

2.6.10. Literals

```
literal ::=
      boolean-literal
    | nat-literal
    | char-literal
    | string-literal
```

Sample literals:

```
true
3260
#z
"On/Off switch"
```

Restriction: No whitespace is allowed anywhere inside any kind of literal, except for “significant” whitespace in string-literals, as explained there.

Literals provide denotations for the inhabitants of the “built-in” types `Boolean`, `Nat`, `Char` and `String`. The value of a literal is independent of the environment.

(There are no literals for the built-in type `Integer`. For nonnegative integers, a nat-literal can be used. For negative integers, apply the built-in op `-`, which negates an integer: `-1` denote the negative integer -1.)

2.6.10.1. Boolean-literals

boolean-literal `::=` **`true`** `|` **`false`**

Sample boolean-literals:

```
true
false
```

The type `Boolean` has precisely two inhabitants, the values of `true` and `false`.

Note that `true` and `false` are not constructors. So `embed true` is ungrammatical.

2.6.10.2. Nat-literals

nat-literal `::=` decimal-digit `{` decimal-digit `}*`

Sample nat-literals:

```
3260
007
```

The type-descriptor `Nat` is, by definition, the subtype of `Integer` restricted to the nonnegative integers 0, 1, 2, ... , which we identify with the natural numbers. The value of a nat-literal is the natural number of which it is a decimal representation; for example, the nat-literal `3260` denotes the natural number 3260. Leading decimal-digits `0` have no significance: both `007` and `7` denote the number 7.

2.6.10.3. Char-literals

```
char-literal ::= #char-literal-glyph

char-literal-glyph ::= char-glyph | "

char-glyph ::=
      letter
    | decimal-digit
    | other-char-glyph

other-char-glyph ::=
      ! | : | @ | # | $ | % | ^ | & | * | ( | ) | _ | - | + | =
    | | | ~ | ‘ | . | , | < | > | ? | / | ; | ’ | [ | ] | { | }
    | \\ | \"
    | \a | \b | \t | \n | \v | \f | \r | \s
    | \x hexadecimal-digit hexadecimal-digit

hexadecimal-digit ::=
      decimal-digit
    | a | b | c | d | e | f
    | A | B | C | D | E | F
```

Sample char-literals:

```
#z
#\x7a
```

The type `Char` is inhabited by the 256 8-bit *characters* occupying decimal positions 0 through 255 (hexadecimal positions 00 through FF) in the ISO 8859-1 code table. The first 128 characters of that code table are the traditional ASCII characters (ISO 646). (Depending on the operating environment, in particular the second set of 128 characters -- those with “the high bit set” -- may print or otherwise be visually presented differently than intended by the ISO 8859-1 code.) The value of a char-literal is a character of type `Char`.

The value of a char-literal `#`***`G`***, where ***`G`*** is a char-glyph, is the character denoted by ***`G`***. For example, `#z` is the character that prints as `z`. The two-mark char-literal `#"` provides a variant notation of the three-mark char-literal `#\"` and yields the character `"` (decimal position 34).

Each one-mark char-glyph `C` denotes the character that "prints" as `C`. The two-mark char-glyph `\\` denotes the character `\` (decimal position 92), and the two-mark char-glyph `\"` denotes the character `"` (decimal position 34).

Notations are provided for denoting eight "non-printing" characters, which, with the exception of the first, are meant to regulate lay-out in printing; the actual effect may depend on the operating environment:

glyph	decimal	name
`\a`	7	bell
`\b`	8	backspace
`\t`	9	horizontal tab
`\n`	10	newline
`\v`	11	vertical tab
`\f`	12	form feed
`\r`	13	return
`\s`	32	space

Finally, every character can be obtained using the hexadecimal representation of its position. The four-mark char-glyph `\x`H_1H_0 denotes the character with hexadecimal position H_1H_0, which is decimal position 16 times the decimal value of hexadecimal-digit H_1 plus the decimal value of hexadecimal-digit H_0, where the decimal value of the digits `0` through `9` is conventional, while the six extra digits `A` through `F` correspond to 10 through 15. The case (lower or upper) of the six extra digits is not significant. For example, `\x7A` or equivalently `\x7a` has decimal position 16 times 7 plus 10 = 122, and either version denotes the character `z`. The "null" character can be obtained by using `\x00`.

2.6.10.4. String-literals

string-literal `::=` `"` string-body `"`

string-body `::=` `{` string-literal-glyph `}*`

string-literal-glyph `::=` char-glyph `|` significant-whitespace

significant-whitespace `::=` space `|` tab `|` newline

The presentation of a significant-whitespace is the whitespace suggested by the name (space, tab or newline).

Sample string-literals:

```
""
"see page"
"see\spage"
"the symbol ’ is a single quote"
"the symbol \" is a double quote"
```

The type `String` is inhabited by the *strings*, which are (possibly empty) sequences of characters. The type `String` is primitive; it is a different type than the isomorphic type `List Char`, and the list operations can not be directly applied to strings.

The value of a string-literal is the sequence of characters denoted by the string-literal-glyphs comprising its string-body, where the value of a significant-whitespace is the whitespace character suggested by the name (space, horizontal tab or newline). For example, the string-literal `"seepage"` is different from `"see page"`; the latter denotes an eight-character string of which the fourth character is a space. The space can be made explicit by using the char-glyph `\s`.

When a double-quote character `"` is needed in a string, it must be escaped, as in `"[6’2\"]"`, which would print like this: `[6’2"]`.

2.6.11. Field-selections

field-selection `::=` closed-expression `.` field-selector

field-selector `::=` nat-literal `|` field-name

Disambiguation. A closed-expression of the form *`M.N`*, in which *`M`* is a word-symbol and *`N`* is a name, is interpreted as an op-name if *`M.N`* occurs as the op-name of an op-declaration or op-definition in the spec in which it occurs or in the set of names imported from another spec through an import-declaration. Otherwise, *`M.N`* is interpreted as a field-selection. (The effect of a field-selection can always be obtained with a projector.)

Sample field-selections:

```
triple.2
z.re
```

A field-selection `E.F` is a convenient notation for the equivalent expression `(project F E)`. (See under *Projectors*.)

2.6.12. Tuple-displays

tuple-display `::=` **(** tuple-display-body **)**

tuple-display-body `::=` `[` expression **,** expression `{` **,** expression `}*` `]`

Sample tuple-display:

```
("George", Poodle : Dog, 10)
```

Note that a tuple-display-body contains either no expressions, or else at least two.

The value of a tuple-display whose tuple-display-body is not empty, is the tuple whose components are the respective values of the expressions of the tuple-display-body, taken in textual order. The type of that tuple is the “product” of the corresponding types of the components. The value of `()` is the empty tuple, which is the sole inhabitant of the unit type `()`. (The fact that the notation `()` does double duty, for a type-descriptor and as an expression, creates no ambiguity. Note also that -- unlike the empty list-display `[]` -- the expression `()` is monomorphic, so there is no need to ever annotate it with a type-descriptor.)

2.6.13. Record-displays

record-display `::=` **{** record-display-body **}**

record-display-body `::=` `[` field-filler `{` **,** field-filler `}*` `]`

field-filler `::=` field-name equals expression

Sample record-display:

```
{name = "George", kind = Poodle : Dog, age = 10}
```

The value of a record-display is the record whose components are the respective values of the expressions of the record-display-body, taken in the lexicographic order of the field-names, as discussed under *Type-records*. The type of that record is the record type with the same set of field-names, where the type for each field-name *F* is the type of the corresponding type of the component selected by *F* in the record. The value of `{}` is the empty tuple, which is the sole inhabitant of the unit type `()`. (For expressions as well as for type-descriptors, the notations `{}` and `()` are fully interchangeable.)

2.6.14. Sequential-expressions

sequential-expression `::=` **(** open-sequential-expression **)**

open-sequential-expression `::=` void-expression **;** sequential-tail

void-expression `::=` expression

sequential-tail `::=` expression | open-sequential-expression

Sample sequential-expression:

```
(writeLine "key not found"; embed Missing)
```

A sequential-expression `(`*V*`;` *T*`)` is equivalent to the let-expression `let _ =` *V* `in (`*T*`)`. So the value of a sequential-expression `(`V_1`;` ... `;` V_n`;` *E*`)` is the value of its last constituent expression *E*.

Sequential-expressions can be used to achieve non-functional “side effects”, effectuated by the elaboration of the void-expressions, in particular the output of a message. This is useful for tracing the execution of generated code. The equivalent effect of the example above can be achieved by a let-binding:

```
let _ = writeLine "key not found" in
embed Missing
```

(If the intent is to temporarily add, and later remove or disable the tracing output, this is probably a more convenient style, as the modifications needed concern a single full text line.) Any values resulting from elaborating the void-expressions are discarded.

2.6.15. List-displays

list-display `::=` **[** list-display-body **]** list-display-body `::= [` expression `{ ,` expression `}* ]`

Sample list-display:

```
["Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"]
```

Restriction. All expressions of the list-display-body must have the same type.

Note that a list-display `[]` with empty list-display-body is polymorphic, and may need to be type-disambiguated, for example with a type annotation. In a case like `[[], [1]]`, there is no need to disambiguate `[]`, since the above restriction already implies that `[]` here has the same type as `[1]`, which has type `List Nat`.

The parameterized type `List`, although built-in, is actually not primitive, but defined by:

```
type List a =
  | Nil
  | Cons a * List a
```

The empty list-display `[]` denotes the same list as the expression `Nil`, a singleton list-display `[`*E*`]` denotes the same list as the expression `Cons (`*E*`, Nil)`, and a multi-element list-display `[`E_1`,` E_2`, ... ,` $E\_{n\}$`]` denotes the same list as the expression `Cons (`E_1`, [`E_2`, ... ,` $E\_{n\}$`])`.

2.6.16. Structors

structor `::=`
    projector
  `|` relaxator
  `|` quotienter
  `|` chooser
  `|` embedder
  `|` embedding-test

The structors are a medley of constructs, all having polymorphic or type-ambiguous function types and denoting special functions that go between structurally related types, such as the constructors of sum types and the destructors of product types.

Restriction. Like all polymorphic or type-ambiguous constructs, a structor can only be used in a context where its type can be inferred uniquely. This restriction will not be repeated for the various kinds of structors described in the following subsections.

For example, the following correct spec becomes incorrect when any of the type annotations is omitted:

```
spec
  def fa(a) p2 = project 2 : String *  a  ->  a
  def       q2 = project 2 : String * Nat -> Nat
endspec
```

2.6.16.1. Projectors

projector `::=` **`project`** field-selector

Sample projectors:

```
project 2
project re
```

When the field-selector is some nat-literal with value i, it is required that i be at least 1. The type of the projector is a function type (whose domain type is a product type) of the form T_1 * T_2 * ... * T_n -> T_i, where n is at least i, and the value of the projector is the function that maps each n-tuple (v_1, v_2, ... , v_n) inhabiting the domain type to its ith component v_i.

When the field-selector is some field-name F, the type of the projector is a function type (whose domain type is a record type) of the form {F_1 : T_1, F_2 : T_2, ... , F_n : T_n} -> T_i, where F is the same field-name as F_i for some natural number i in the range 1 through n. Assuming that the fields are lexicographically ordered by field-name (see under *Type-records*), the value of the projector is the function that maps each n-tuple (v_1, v_2, ... , v_n) inhabiting the domain type to its ith component v_i.

2.6.16.2. Relaxators

relaxator `::=` **`relax`** closed-expression

Sample relaxator:

```
relax even
```

Restriction. The closed-expression of a relaxator must have some function type `T -> Boolean`.

The type of relaxator `relax P`, where `P` has type `T -> Boolean`, is the function type (whose domain is a subtype) `(T | P) -> T`. The value of the relaxator is the function that maps each inhabitant of subtype `(T | P)` to the same value -- apart from the type information -- inhabiting supertype `T`.

For example, given

```
type Even = (Nat | even)
```

we have the typing

```
relax even : Even -> Nat
```

for the function that injects the even natural numbers back into the supertype of `Even`.

Metaslang does not require the explicit use of a relaxator to relax an expression from a subtype to its supertype if the context requires the latter. Implicit relaxation will take place when needed. For example, in the expression `-1` the nat-literal `1` of type `Nat` is implicitly relaxed to type `Integer` to accommodate the unary negation operator `-`, which has type `Integer -> Integer`.

Note the remarks about equivalence of type-restrictions in the corresponding section.

2.6.16.3. Quotienters

quotienter `::=` **`quotient`** closed-expression

Sample quotienter:

```
quotient (fn (m, n) -> m rem 3 = n rem 3)
```

Restriction. The closed-expression of a quotienter must have some type `T * T -> Boolean`; in addition, it must be an equivalence relation, as explained under *Type-quotients*.

The type of quotienter `quotient Q`, where `Q` has type `T * T -> Boolean`, is the function type `T -> T / Q`, that is, it goes from some type to one of its quotient types. The value of the quotienter is the function that maps each inhabitant of type `T` to the `Q`-equivalence class inhabiting `T / Q` of which it is a member.

For example, given

```
def congMod3 : Nat * Nat -> Boolean =
  (fn (m, n) -> m rem 3 = n rem 3)

type Z3 = Nat / congMod3
```

we have the typing

```
quotient congMod3 : Nat -> Z3
```

and the function maps, for example, the number 5 to the equivalence class {2, 5, 8, ...}, which is one of the three inhabitants of `Z3`.

2.6.16.4. Choosers

chooser `::=` **`choose`** closed-expression

Sample chooser:

```
choose congMod3
```

Restriction. In a chooser `choose` *`Q`*, expression *`Q`* must have some type *`T`* `*` *`T`* `-> Boolean`, and must be an equivalence relation (see under *Type-quotients*).

The type of a chooser `choose` *`Q`*, where *`Q`* has type *`S`* `*` *`S`* `-> Boolean`, is a function type of the form *`R`* `-> (`*`S`* `/`*`Q`* `->`*`T`*`)`, where *`R`* is the subtype of *`S`* `->` *`T`* consisting of the *`Q`*-constant (explained below) functions. Expressed more formally, *`R`* is the type `{f :`*`S`* `->`*`T`* `| fa((x,y) :`*`S`* `*` *`S`*`)` *`Q`*`(x,y) => f x = f y}`, where the names `f`, `x` and `y` must be replaced by "fresh" names not clashing with names already in use in *`S`*, *`T`* or *`Q`*.

The value of the chooser is the function mapping each *`Q`*-constant (explained below) function *`f`* inhabiting type *`S`* `->` *`T`* to the function that maps each inhabitant *`C`* of *`S`* `/` *`Q`* to *`f x`*, where *`x`* is any member of *`C`*. Expressed symbolically, using a pseudo-function `any` that arbitrarily picks any member from a nonempty set, this is the function

```
fn f -> fn C -> f (any C)
```

The requirement of *`Q`*-constancy is precisely what is needed to make this function insensitive to the choice made by `any`.

Function *`f`* is *`Q`*-constant if, for each *`Q`*-equivalence class *`C`* inhabiting *`S`* `/` *`Q`*, *`f x`* equals *`f y`* for any two values *`x`* and *`y`* that are members of *`C`*, or *`f`* is undefined on all members of *`C`*. (Since the result of *`f`* is constant across each equivalence class, it does not matter which of its elements is selected by `any`.) For example -- continuing the example of the previous section -- function `fn n -> n*n rem 3` is `congMod3`-constant; for the equivalence class {2, 5, 8, ...}, for example, it maps each member to the same value 1. So `choose congMod3 (fn n -> n*n rem 3)` maps the inhabitant {2, 5, 8, ...} of type `Z3` to the natural number 1.

The most discriminating *`Q`*-constant function is `quotient` *`Q`*, and `choose` *`Q`* `quotient` *`Q`* is the identity function on the quotient type for *`Q`*.

The meaning of `choose` *`Q`* `(fn x ->` *`E`*`)` *`A`* is the same as that of the let-expression `let quotient` *`Q`* `x =` *`A`* `in` *`E`*. Indeed, often a quotient-pattern offers a more convenient way of expressing the intention of a chooser. Note, however, the remarks on the proof obligations for quotient-patterns.

2.6.16.5. Embedders

embedder `::= [` **`embed`** `]` constructor

Sample embedders:

```
Nil
embed Nil
Cons
embed Cons
```

Disambiguation. If an expression consists of a single name, which, in the context, is both the name of a constructor and the name of an op or a local-variable in scope, then it is interpreted as the latter of the various possibilities. For example, in the context of

```
type Answer = | yes | no

def yes = no : Answer

def which (a : Answer) = case a of
  | yes -> "Yes!"
  | no  -> "Oh, no!"
```

the value of `which yes` is `"Oh, no!"`, since `yes` here is disambiguated as identifying the op `yes`, which has value `no`. The interpretation as embedder is forced by using the `embed` keyword: the value of `which embed yes` is `"Yes!"`. By using names that begin with a capital letter for constructors, and names that do not begin with a capital letter for ops and local-variables, the risk of an accidental wrong interpretation can be avoided.

The semantics of embedders is described in the section on *Type-sums*. The presence or absence of the keyword `embed` is not significant for the meaning of the construct (although it may be required for grammatical disambiguation, as described above).

2.6.16.6. Embedding-tests

embedding-test `::=` **`embed?`** constructor

Sample embedding-test:

```
embed? Cons
```

Restriction. The type of an embedding-test `embed?` *`C`* must be of the form *`T/replaceable>[[ -> Boolean]], where`* *`T/replaceable> is a sum type that has a`* constructor *`C`*.

The value of embedding-test `embed?` *`C`* is the predicate that returns `true` if the argument value -- which, as inhabitant of a sum type, is tagged -- has tag *`C`*, and otherwise `false`. The embedding-test can be equivalently rewritten as

```
fn
 | C _  ->  true
 | _    ->  false
```

where the wildcard _ in the first branch is omitted when *`C`* is parameter-less.

In plain words, `embed?` *`C`* tests whether its sum-typed argument has been constructed with the constructor *`C`*. It is an error when *`C`* is not a constructor of the sum type.

2.7. Matches and Patterns

2.7.1. Matches

match ::= [**|**] branch { **|** branch }*

branch ::= pattern **->** expression

Sample matches:

```
{re = x, im = y} -> {re = x, im = -y}

  Empty -> true
| Push {top = _, pop = rest} -> hasBottom? rest

| Empty -> true
| Push {top = _, pop = rest} -> hasBottom? rest
```

Restriction. In a match, given the environment, there must be a unique type *T* to which the pattern of each branch conforms, and a unique type *T* to which the expression of each branch conforms, and then the match has type *S* `->` *T*. The pattern of each branch then has type *S*.

Disambiguation. If a branch could belong to several open matches, it is interpreted as being a branch of the textually most recently introduced match. For example,

```
case x of
  | A -> a
  | B -> case y of
           | C -> c
  | D -> d
```

is not interpreted as suggested by the indentation, but as

```
case x of
  | A -> a
  | B -> (case y of
            | C -> c
            | D -> d)
```

If the other interpretation is intended, the expression introducing the inner match needs to be parenthesized:

```
case x of
  | A -> a
  | B -> (case y of
            | C -> c)
  | D -> d
```

Acceptance and return value *y*, if any, of a value *x* for a given match are determined as follows. If each branch of the match rejects *x* (see below), the whole match rejects *x*, and does not return a value. Otherwise, let *B* stand for the textually first branch accepting *x*. Then *y* is the return value of *x* for *B*.

Acceptance and return value *y*, if any, of a value *x* for a branch *P* `->` *E* in an environment *C* are determined as follows. If pattern *P* rejects *x*, the branch rejects *x*, and does not return a value. (For acceptance by a pattern, see under *Patterns*.) Otherwise, *y* is the value of expression *E* in the environment *C* extended with the acceptance binding of pattern *P* for *x*.

For example, in

```
case z of
  | (x, true)  -> Some x
  | (_, false) -> None
```

if `z` has value `(3, true)`, the first branch accepts this value with acceptance binding `x = 3`. The value of `Some x` in the extended environment is then `Some 3`. If `z` has value `(3, false)`, the second branch accepts this value with empty acceptance binding (empty since there are no "accepting" local-variables in pattern `(_, false)`), and the return value is `None` (interpreted in the original environment).

2.7.2. Patterns

pattern `::=`
        annotated-pattern
      | tight-pattern

tight-pattern `::=`
        aliased-pattern
      | cons-pattern
      | embed-pattern
      | quotient-pattern

| relax-pattern
| closed-pattern

closed-pattern **::=**
variable-pattern
| wildcard-pattern
| literal-pattern
| list-pattern
| tuple-pattern
| record-pattern
| **(** pattern **)**

(As for expressions, the distinctions tight- and closed- for patterns have no semantic significance, but merely serve to avoid grammatical ambiguities.)

annotated-pattern ::= pattern **:** type-descriptor

aliased-pattern ::= variable-pattern **as** tight-pattern

cons-pattern ::= closed-pattern **::** tight-pattern

embed-pattern ::= constructor [closed-pattern]

quotient-pattern ::= **quotient** closed-expression tight-pattern

relax-pattern ::= **relax** closed-expression tight-pattern

variable-pattern ::= local-variable

wildcard-pattern ::= _

literal-pattern ::= literal

list-pattern ::= **[** list-pattern-body **]**

list-pattern-body ::= [pattern { **,** pattern }*]

tuple-pattern ::= **(** tuple-pattern-body **)**

tuple-pattern-body ::= [pattern **,** pattern { **,** pattern }*]

record-pattern ::= **{** record-pattern-body **}**

record-pattern-body ::= [field-patterner { **,** field-patterner }*] field-patterner `::=` field-name `[` equals pattern `]`

Sample patterns:

```
(i, p) : Integer * Boolean
z as {re = x, im = y}
hd :: tail
Push {top, pop = rest}
embed Empty
quotient congMod3 n
relax even e
x
_
#z
[0, x]
(c1 as (0, _), x)
{top, pop = rest}
```

Restriction. Like all polymorphic or type-ambiguous constructs, a pattern may only be used in a context where its type can be uniquely inferred.

Disambiguation. A single name used as a pattern is an embed-pattern if it is a constructor of the type of the pattern. Otherwise, the name is a variable-pattern.

Restriction. Each local-variable in a pattern must be a different name, disregarding any local-variables introduced in expressions or type-descriptors contained in the pattern. (For example, `Line (z, z)` is not a lawful pattern, since `z` is repeated; but `n : {n : Nat | n < p}` is lawful: the second `n` is "shielded" by the type-comprehension in which it occurs.)

Restriction. The closed-expression of a quotient-pattern must have some type *T* `*` *T* `-> Boolean`; in addition, it must be an equivalence relation, as explained under *Type-quotients*.

Restriction. Quotient-patterns may only be used in a definition or claim if the result is insensitive to the choice of representative from the equivalence class. Specware 4.1 does not attempt to generate proof obligations for establishing this.

Restriction. The closed-expression of a relax-pattern must have some function type *T* `-> Boolean`.

To define acceptance and acceptance binding (if any) for a value and a pattern, we introduce a number of auxiliary definitions.

The *accepting* local-variables of a pattern P are the collection of local-variables occurring in P, disregarding any local-variables introduced in expressions or type-descriptors contained in the P. For example, in pattern `u : {v : S | p v}`, `u` is an accepting local-variable, but `v` is not. (The latter is an accepting local-variable of pattern `v : S`, but not of the larger pattern.)

The *expressive descendants* of a pattern are a finite set of expressions having the syntactic form of patterns, as determined in the following three steps (of which the order of steps 1 and 2 is actually immaterial).

Step 1. From pattern P, form some *tame variant* P_t by replacing each field-patterner consisting of a single field-name F by the field-patterner $F = F$ and replacing each wildcard-pattern _ in P by a unique fresh name, that is, any name that does not already occur in the spec, directly or indirectly through an import. For example, assuming that the name `v7944` is fresh, a tame variant of

```
s0 as _ :: s1 as (Push {top, pop = rest}) :: ss
```

is

```
s0 as v7944 :: s1 as (Push {top = top, pop = rest}) :: ss
```

Step 2. Next, from P_t, form a (tamed) *construed version* P_{tc} by replacing each constituent cons-pattern H `::` T by the embed-pattern `Cons (`H`,` T`)`, where `Cons` denotes the constructor of the parameterized type `List`. For the example, the construed version is:

```
s0 as Cons (v7944,
            s1 as Cons (Push {top = top, pop = rest}, ss))
```

Step 3. Finally, from P_{tc}, form the set ED_P of *expressive descendants* of P, where expression E is an expressive descendant if E can be obtained by repeatedly replacing some constituent aliased-pattern L `as` R of P_{tc} by one of the two patterns L and R until no aliased-patterns remain, and then interpreting the result as an expression. For the example, the expressive descendants are the three expressions:

```
s0
Cons (v7944, s1)
Cons (v7944, Cons (Push {top = top, pop = rest}, ss))
```

An *accepting binding* of a pattern P for a value x in an environment C is some binding B of typed values to the accepting local-variables of the *tame* variant P_t, such that the value of each expressive descendant E in ED_P in the environment C extended with binding B, is the same typed value as x.

Acceptance and acceptance binding, if any, for a value x and a pattern P are then determined as follows. If there is no accepting binding of P for x, x is rejected. If an accepting binding exists, the value x is accepted by pattern P. There is a unique binding B among the accepting bindings in which the type of each assigned value is as "restricted" as possible in the subtype-supertype hierarchy without violating well-typedness constraints (in other words, there are no avoidable implicit relaxations). The acceptance binding is then the binding B *projected on* the accepting local-variables of P.

For the example, the accepting local-variables of P_t are the six local-variables `s0`, `s1`, `ss`, `rest` and `v7944`. In general, they are the accepting local-variables of the original pattern together with any fresh names used for taming. Let the value x being matched against the pattern be

```
Cons (Empty, Cons (Push {top = 200, pop = Empty}, Nil))
```

Under the accepting binding

```
s0 = Cons (Empty, Cons (Push {top = 200, pop = Empty}, Nil))
s1 = Cons (Push {top = 200, pop = Empty}, Nil)
ss = Nil
top = 200
rest = Empty
v7944 = Empty
```

the value of each E in ED_P amounts to the value x. Therefore, x is accepted by the original pattern, with acceptance binding

```
s0 = Cons (Empty, Cons (Push {top = 200, pop = Empty}, Nil))
s1 = Cons (Push {top = 200, pop = Empty}, Nil)
ss = Nil
top = 200
rest = Empty
```

obtained by "forgetting" the fresh name `v7944`.

Appendix A. Metaslang Grammar

This appendix lists the grammar rules of the Metaslang specification language. These rules are identical to those of the Chapter on *Metaslang*. They are brought together here, without additional text, for easy reference.

Models.

```
op ::= op-name

spec ::= spec-form
```

Symbols and Names.

```
symbol ::= name | literal | special-symbol

name ::= word-symbol | non-word-symbol

word-symbol ::= word-start-mark { word-continue-mark }*

word-start-mark ::= letter

word-continue-mark ::=
  letter | decimal-digit | _ | ?

letter ::=
    A | B | C | D | E | F | G | H | I | J | K | L | M
  | N | O | P | Q | R | S | T | U | V | W | X | Y | Z
  | a | b | c | d | e | f | g | h | i | j | k | l | m
  | n | o | p | q | r | s | t | u | v | w | x | y | z

decimal-digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

non-word-symbol ::= non-word-mark { non-word-mark }*

non-word-mark ::=
    ‘ | ~ | ! | @ | $ | ^ | & | * | -
  | = | + | \ | | | : | < | > | / | ?

special-symbol ::= _ | ( | ) | [ | ] | { | } | ; | , | .
```

Comments.

comment **::=** line-end-comment | block-comment

line-end-comment **::=** **%** line-end-comment-body

line-end-comment-body **::=**
  any-text-up-to-end-of-line

block-comment **::=** **(*** block-comment-body ***)**

block-comment-body **::=**
  any-text-including-newlines-and-nested-block-comments

Units.

unit-definition **::=** unit-identifier **=** unit-term

unit-term **::=**
    spec-term
  | morphism-term
  | diagram-term
  | target-code-term
  | proof-term

specware-file-contents **::=**
    unit-term
  | infile-unit-definition { infile-unit-definition }*

infile-unit-definition **::=** fragment-identifier **=** unit-term

fragment-identifier **::=** word-symbol

Unit Identifiers.

unit-identifier **::=** swpath-based-path | relative-path

swpath-based-path **::=** **/** relative-path

relative-path **::=** { path-element **/** }* path-element [**#** fragment-identifier]

path-element **::=** word-symbol

Specs.

spec-term `::=`
    unit-identifier
  | spec-form
  | spec-qualification
  | spec-translation
  | spec-substitution
  | diagram-colimit
  | obligator

Spec Forms.

spec-form `::=` **`spec`** `{` declaration `}*` **`endspec`**

Qualifications.

spec-qualification `::=` qualifier **`qualifying`** spec-term

qualifier `::=` word-symbol

qualifiable-name `::=` unqualified-name `|` qualified-name

unqualified-name `::=` name

qualified-name `::=` qualifier **`.`** name

Translations.

spec-translation `::=` **`translate`** spec-term **`by`** name-map

name-map `::=` **`{`** `[` name-map-item `{` **`,`** name-map-item `}*` `]` **`}`**

name-map-item `::=` type-name-map-item `|` op-name-map-item

type-name-map-item `::=` `[` **`type`** `]` qualifiable-name **`+->`** qualifiable-name

op-name-map-item `::=`
    `[` **`op`** `]` annotable-qualifiable-name **`+->`** annotable-qualifiable-name

annotable-qualifiable-name `::=` qualifiable-name `[` **`:`** type-descriptor `]`

Substitutions.

spec-substitution ::= spec-term **[** morphism-term **]**

Diagram Colimits.

diagram-colimit ::= **colimit** diagram-term

Obligators.

obligator ::= **obligations** unit-term

Morphisms.

```
morphism-term ::=
    unit-identifier
  | spec-morphism
```

spec-morphism ::= **morphism** spec-term **->** spec-term name-map

Diagrams.

```
diagram-term ::=
    unit-identifier
  | diagram-form
```

diagram-form ::= **diagram {** diagram-element { **,** diagram-element }* **}**

```
diagram-element ::=
    diagram-node
  | diagram-edge
```

diagram-node ::= name **+->** spec-term

diagram-edge ::= name **:** name **->** name **+->** morphism-term

Target Code Terms.

target-code-term ::=
    generate target-language-name spec-term [**in** string-literal]

target-language-name ::= **c** | **java** | **lisp**

Proof Terms.

```
proof-term ::=
    prove claim-name in spec-term
                        [ with prover-name ]
                        [ using { claim-list } ]
                        [ options prover-options ]

prover-name ::= snark

claim-list ::= claim-name { , claim-name }*

prover-options ::= string-literal
```

Declarations.

```
declaration ::=
      import-declaration
    | type-declaration
    | op-declaration
    | definition

definition ::=
      type-definition
    | op-definition
    | claim-definition

equals ::= is | =
```

Import-declarations.

```
import-declaration ::= import spec-term
```

Type-declarations.

```
type-declaration ::= type type-name [ formal-type-parameters ]

formal-type-parameters ::= local-type-variable | ( local-type-variable-list )

local-type-variable ::= name

local-type-variable-list ::= local-type-variable { , local-type-variable }*
```

Type-definitions.

type-definition `::=` **`type`** type-name [formal-type-parameters] equals type-descriptor

Op-declarations.

op-declaration `::=` **`op`** op-name [fixity] **`:`** type-scheme

fixity `::=` associativity priority

associativity `::=` **`infixl`** | **`infixr`**

priority `::=` nat-literal

type-scheme `::=` [type-variable-binder] type-descriptor

type-variable-binder `::=` **`fa`** **`(`** local-type-variable-list **`)`**

Op-definitions.

op-definition `::=`
`def` [type-variable-binder] formal-expression [**`:`** type-descriptor] equals expression

formal-expression `::=` op-name | formal-application

formal-application `::=` formal-application-head formal-parameter

formal-application-head `::=` op-name | formal-application

formal-parameter `::=` closed-pattern

Claim-definitions.

claim-definition `::=` claim-kind claim-name equals claim

claim-kind `::=` **`axiom`** | **`theorem`** | **`conjecture`**

claim-name `::=` name

claim `::=` [type-quantification] expression

type-quantification `::=` **`type`** type-variable-binder

Type-descriptors.

```
type-descriptor ::=
      type-sum
    | type-arrow
    | slack-type-descriptor

slack-type-descriptor ::=
      type-product
    | tight-type-descriptor

tight-type-descriptor ::=
      type-instantiation
    | closed-type-descriptor

closed-type-descriptor ::=
      type-name
    | local-type-variable
    | type-record
    | type-restriction
    | type-comprehension
    | type-quotient
    | ( type-descriptor )
```

Type-sums.

```
type-sum ::= type-summand { type-summand }*

type-summand ::= | constructor [ slack-type-descriptor ]

constructor ::= name
```

Type-arrows.

```
type-arrow ::= arrow-source -> type-descriptor

arrow-source ::= type-sum | slack-type-descriptor
```

Type-products.

```
type-product ::= tight-type-descriptor * tight-type-descriptor { * tight-type-descriptor }*
```

Type-instantiations.

type-instantiation **::=** type-name actual-type-parameters

actual-type-parameters **::=** closed-type-descriptor | **(** proper-type-list **)**

proper-type-list **::=** type-descriptor **,** type-descriptor { **,** type-descriptor }*

Type-names.

type-name **::=** qualifiable-name

Type-records.

type-record **::=** **{** [field-typer-list] **}** | **( )**

field-typer-list **::=** field-typer { **,** field-typer }*

field-typer **::=** field-name **:** type-descriptor

field-name **::=** name

Type-restrictions.

type-restriction **::=** **(** slack-type-descriptor **|** expression **)**

Type-comprehensions.

type-comprehension **::=** **{** annotated-pattern **|** expression **}**

Type-quotients.

type-quotient **::=** closed-type-descriptor **/** closed-expression

Expressions.

```
expression ::=
      lambda-form
    | case-expression
    | let-expression
    | if-expression
    | quantification
    | annotated-expression
    | tight-expression

tight-expression ::=
      application
    | restrict-expression
    | closed-expression

closed-expression ::=
      op-name
    | local-variable
    | literal
    | field-selection
    | tuple-display
    | record-display
    | sequential-expression
    | list-display
    | structor
    | ( expression )
    | ( inbuilt-infix )

inbuilt-infix ::= => | || | && | = | ~= | <<
```

Lambda-forms.

```
lambda-form ::= fn match
```

Case-expressions.

```
case-expression ::= case expression of match
```

Let-expressions.

```
let-expression ::= let let-bindings in expression

let-bindings ::= recless-let-binding | rec-let-binding-sequence
```

recless-let-binding `::=` pattern equals expression

rec-let-binding-sequence `::=` rec-let-binding `{` rec-let-binding `}*`

rec-let-binding `::=`
  `def` name formal-parameter-sequence `[` **`:`** type-descriptor `]` equals expression

formal-parameter-sequence `::=` formal-parameter `{` formal-parameter `}*`

If-expressions.

if-expression `::=` **`if`** expression **`then`** expression **`else`** expression

Quantifications.

quantification `::=` quantifier **`(`** local-variable-list **`)`** expression

quantifier `::=` **`fa`** `|` **`ex`**

local-variable-list `::=` annotable-variable `{` **`,`** annotable-variable `}*`

annotable-variable `::=` local-variable `[` **`:`** type-descriptor `]`

local-variable `::=` name

Annotated-expressions.

annotated-expression `::=` tight-expression **`:`** type-descriptor

Applications.

application `::=` prefix-application `|` infix-application

prefix-application `::=` application-head actual-parameter

application-head `::=` closed-expression `|` prefix-application

actual-parameter `::=` closed-expression

infix-application `::=` operand infix-operator operand

operand `::=` tight-expression

infix-operator `::=` op-name | inbuilt-infix

Restrict-expressions.

restrict-expression `::=` **`restrict`** closed-expression closed-expression

Op-names.

op-name `::=` qualifiable-name

Literals.

literal `::=`
    boolean-literal
  | nat-literal
  | char-literal
  | string-literal

Boolean-literals.

boolean-literal `::=` **`true`** | **`false`**

Nat-literals.

nat-literal `::=` decimal-digit `{` decimal-digit `}*`

Char-literals.

char-literal `::=` #char-literal-glyph

char-literal-glyph `::=` char-glyph | **"**

char-glyph `::=`
    letter
  | decimal-digit
  | other-char-glyph

other-char-glyph `::=`

```
    ! | : | @ | # | $ | % | ^ | & | * | ( | ) | _ | - | + | =
  | | | ~ | ‘ | . | , | < | > | ? | / | ; | ’ | [ | ] | { | }
  | \\ | \"
  | \a | \b | \t | \n | \v | \f | \r | \s
  | \x hexadecimal-digit hexadecimal-digit

hexadecimal-digit ::=
    decimal-digit
  | a | b | c | d | e | f
  | A | B | C | D | E | F
```

String-literals.

```
string-literal ::= " string-body "

string-body ::= { string-literal-glyph }*

string-literal-glyph ::= char-glyph | significant-whitespace

significant-whitespace ::= space | tab | newline
```

Field-selections.

```
field-selection ::= closed-expression . field-selector

field-selector ::= nat-literal | field-name
```

Tuple-displays.

```
tuple-display ::= ( tuple-display-body )

tuple-display-body ::= [ expression , expression { , expression }* ]
```

Record-displays.

```
record-display ::= { record-display-body }

record-display-body ::= [ field-filler { , field-filler }* ]

field-filler ::= field-name equals expression
```

Sequential-expressions.

sequential-expression `::=` **`(`** open-sequential-expression **`)`**

open-sequential-expression `::=` void-expression **`;`** sequential-tail

void-expression `::=` expression

sequential-tail `::=` expression | open-sequential-expression

List-displays.

list-display `::=` **`[`** list-display-body **`]`**

list-display-body `::=` `[` expression `{` **`,`** expression `}*` `]`

Structors.

structor `::=`
    projector
  | relaxator
  | quotienter
  | chooser
  | embedder
  | embedding-test

projector `::=` **`project`** field-selector

relaxator `::=` **`relax`** closed-expression

quotienter `::=` **`quotient`** closed-expression

chooser `::=` **`choose`** closed-expression

embedder `::=` `[` **`embed`** `]` constructor

embedding-test `::=` **`embed?`** constructor

Matches.

match `::=` `[` **`|`** `]` branch `{` **`|`** branch `}*`

branch `::=` pattern **`->`** expression

Patterns.

```
pattern ::=
      annotated-pattern
    | tight-pattern

tight-pattern ::=
      aliased-pattern
    | cons-pattern
    | embed-pattern
    | quotient-pattern
    | relax-pattern
    | closed-pattern

closed-pattern ::=
      variable-pattern
    | wildcard-pattern
    | literal-pattern
    | list-pattern
    | tuple-pattern
    | record-pattern
    | ( pattern )


annotated-pattern ::= pattern : type-descriptor

aliased-pattern ::= variable-pattern as tight-pattern

cons-pattern ::= closed-pattern :: tight-pattern

embed-pattern ::= constructor [ closed-pattern ]

quotient-pattern ::= quotient closed-expression tight-pattern

relax-pattern ::= relax closed-expression tight-pattern

variable-pattern ::= local-variable

wildcard-pattern ::= _

literal-pattern ::= literal

list-pattern ::= [ list-pattern-body ]

list-pattern-body ::= [ pattern { , pattern }* ]
```

tuple-pattern `::=` **(** tuple-pattern-body **)**

tuple-pattern-body `::=` [pattern **,** pattern { **,** pattern }*]

record-pattern `::=` **{** record-pattern-body **}**

record-pattern-body `::=` [field-patterner { **,** field-patterner }*]

field-patterner `::=` field-name [equals pattern]

Appendix B. Base Libraries

The base libraries are automatically imported by every user-defined spec.

This appendix provides a brief description of the sorts and ops in the current base libraries. The title of each section of this appendix is the qualifier of the sort-names and op-names given therein. For example, the full op-name for op ~ described in Section “Boolean” is `Boolean.~`.

B.1. Boolean

Sort

```
sort Boolean
```

Ops

Name	Fixity	Sort	Description
`~`		`Boolean -> Boolean`	logical negation
`&`	`infixr 15`	`Boolean * Boolean -> Boolean`	logical and
`or`	`infixr 14`	`Boolean * Boolean -> Boolean`	logical or
`=>`	`infixr 13`	`Boolean * Boolean -> Boolean`	logical implication
`<=>`	`infixr 12`	`Boolean * Boolean -> Boolean`	logical equivalence
`~=`	`infixr 20`	`fa(a) a * a -> Boolean`	inequality (logical exclusive or)

Name	Fixity	Sort	Description
toString		Boolean -> String	converts Boolean value to string
show		Boolean -> String	same as toString
compare		Boolean * Boolean -> Comparison	compares two Boolean values

B.2. Integer

Sorts

```
sort Integer

sort NonZeroInteger = {i : Integer | i ~= 0}
```

Ops

Name	Fixity	Sort	Description
~		Integer -> Integer	unary minus
+	infixl 25	Integer * Integer -> Integer	addition
-	infixl 25	Integer * Integer -> Integer	subtraction

Name	**Fixity**	**Sort**	**Description**
`*`	`infixl 27`	`Integer * Integer -> Integer`	multiplication
`div`	`infixl 26`	`Integer * NonZeroInteger -> Integer`	division (truncates towards 0)
`rem`	`infixl 26`	`Integer * NonZeroInteger -> Integer`	remainder (`x rem y` = `x - y * (x div y)`)
`<`	`infixl 20`	`Integer * Integer -> Boolean`	less-than
`<=`	`infixl 20`	`Integer * Integer -> Boolean`	less-than-or-equal
`>`	`infixl 20`	`Integer * Integer -> Boolean`	greater-than
`>=`	`infixl 20`	`Integer * Integer -> Boolean`	greater-than-or-equal
`abs`		`Integer -> Integer`	absolute value
`min`		`Integer * Integer -> Integer`	minimum
`max`		`Integer * Integer -> Integer`	maximum
`compare`		`Integer * Integer -> Comparison`	compares two integers
`toString`		`Integer -> String`	converts integer to string
`show`		`Integer -> String`	same as `toString`
`intToString`		`Integer -> String`	same as `toString`

Name	Fixity	Sort	Description
`intConvertible`		`String -> Boolean`	tests if string is representation of integer
`stringToInt`		`(String | intConvertible) -> Integer`	converts “convertible” string to integer

B.3. Nat

Sorts

```
sort Nat = {n : Integer | n >= 0}
sort PosNat = {n : Nat | n > 0 }
```

Ops

Name	Fixity	Sort	Description
`succ`		`Nat -> Nat`	successor
`pred`		`Nat -> Integer`	predecessor
`zero`		`Nat`	the natural number 0
`one`		`Nat`	the natural number 1
`two`		`Nat`	the natural number 2
`posNat?`		`Nat -> Boolean`	yields false for 0, true otherwise
`toString`		`Nat -> String`	converts natural number to string
`show`		`Nat -> String`	same as `toString`

Name	Fixity	Sort	Description
`natToString`		`Nat -> String`	same as `toString`
`natConvertible`		`String -> Boolean`	tests if string is representation of natural number
`stringToNat`		`(String | natConvertible) -> Nat`	converts “convertible” string to natural number

B.4. Char

Sort

```
sort Char
```

Ops

Name	Fixity	Sort	Description
`ord`		`Char -> Nat`	converts character to natural number
`chr`		`Nat -> Char`	converts natural number to character
`isAlpha`		`Char -> Boolean`	true for letters
`isNum`		`Char -> Boolean`	true for digits
`isAlphaNum`		`Char -> Boolean`	true for letters and digits
`isAscii`		`Char -> Boolean`	true for ASCII characters

Name	Fixity	Sort	Description
`isLowerCase`		`Char -> Boolean`	true for lower-case letters
`isUpperCase`		`Char -> Boolean`	true for upper-case letters
`toUpperCase`		`Char -> Char`	converts to upper case
`toLowerCase`		`Char -> Char`	converts to lower case
`compare`		`Char * Char -> Comparison`	compares two character values
`toString`		`Char -> String`	converts character to string
`show`		`Char -> String`	same as `toString`

B.5. String

Sort

```
sort String
```

Ops

Name	Fixity	Sort	Description
`explode`		`String -> List(Char)`	converts string to list of characters

Name	**Fixity**	**Sort**	**Description**
`implode`		`List(Char) -> String`	converts list of characters to string
`length`		`String -> Nat`	length of a string
`leq`	`infixl 20`	`String * String -> Boolean`	lexicographic less-than-or-equal
`lt`	`infixl 20`	`String * String -> Boolean`	lexicographic less-than
`++`	`infixl 11`	`String * String -> String`	string concatenation
`^`	`infixl 11`	`String * String -> String`	same as `++`
`concat`		`String * String -> String`	prefix `op` for string concatenation
`concatList`		`List String -> String`	returns the concatenation of the list elements
`sub`		`String * Nat -> Char`	returns the *n*th character in a string, counting from 0
`substring`		`String * Nat * Nat -> String`	`substring(s, m, n)` returns the substring of *s* from position *m* through position *n*-1, counting from 0
`map`		`(Char -> Char) * String -> String`	returns the concatenation of the results of applying the function given as first parameter to each character of the string
`translate`		`(Char -> String) * String -> String`	returns the concatenation of the results of applying the function given as first parameter to each character of the string

Name	Fixity	Sort	Description
`all`		`(Char -> Boolean) * String`	true if all characters in the string satisfy the predicate given as first parameter
`exists`		`(Char -> Boolean) * String`	true if some character in the string satisfies the predicate given as first parameter
`newline`		`String`	the string representing a line break
`toScreen`		`String -> ()`	prints the string on the terminal
`writeLine`		`String -> ()`	same with a newline appended
`compare`		`String * String -> Comparison`	compares two strings

B.6. List

Sort

```
sort List a = | Nil | Cons a * List a
```

Ops

Name	Fixity	Sort	Description
`nil`		`fa(a) List a`	the empty list
`null`		`fa(a) List a -> Boolean`	true for empty lists
`length`		`List a -> Nat`	length of a list

Name	**Fixity**	**Sort**	**Description**
`cons`		`fa(a) a * List a -> List a`	constructs a list consisting of a first element and a list tail
`insert`		`fa(a) a * List a -> List a`	same as `cons`
`hd`		`fa(a) List a -> a`	returns the first element of the list
`tl`		`fa(a) List a -> List a`	returns the list tail without the first element
`++`	`infixl 11`	`fa(a) List a * List a -> List a`	list concatenation
`@`	`infixl 11`	`fa(a) List a * List a -> List a`	same as `++`
`concat`		`fa(a) List a * List a -> List a`	prefix op for list concatenation
`flatten`		`fa(a) List(List(a)) -> List a`	returns the concatenation of the list elements
`diff`		`fa(a) List a * List a -> List a`	list subtraction: `diff(`*x*`,`*y*`)` returns a list containing the elements of *x* that are not in *y*, preserving the order of the elements in *x*
`member`		`fa(a) a * List a -> Boolean`	list membership
`nth`		`fa(a) List a * Nat -> a`	`nth(`*x*`, `*n*`)` returns the element at position *n* of list *x*, counting from 0
`nthTail`		`fa(a) List a * Nat -> List a`	`nthTail(`*x*`, `*n*`)` returns the tail of list *x*, starting after position *n*, counting from 0
`sublist`		`fa(a) List a * Nat * Nat -> List a`	`sublist(`*x*`, `*m*`, `*n*`)` returns the tail of list *x*, from position *m* up to but not including *n*, counting from 0

Name	Fixity	Sort	Description
`foldl`		`fa(a,b) (a*b -> b) -> b -> List a -> b`	`foldl` *f* *e* *x* successively applies function *f* to the elements of list *x* from left to right. The second argument to *f* is initially *e* and at each next step the result of the previous invocation of *f*
`foldr`		`fa(a,b) (a*b -> b) -> b -> List a -> b`	like `foldl`, but the elements of the list are processed from right to left
`map`		`fa(a,b) (a->b) -> List a -> List b`	applies function to each element of a list and returns the list consisting of the results
`mapPartial`		`fa(a,b) (a->Option b) -> List a -> List b`	like `map` but replacing each result `Some` *y* by *y* and deleting `None` results.
`filter`		`fa(a) (a -> Boolean) -> List a -> List a`	returns the list of elements satisfying the given predicate
`rev`		`fa(a) List a -> List a`	reverse list
`all`		`fa(a) (a -> Boolean) -> List a -> Boolean`	true if all elements of the list satisfy the predicate given as first parameter
`exists`		`fa(a) (a->Boolean) -> List a -> Boolean`	true if some element of the list satisfies the predicate given as first parameter
`find`		`fa(a) (a -> Boolean) -> List a -> Option(a)`	returns `Some` *x* where *x* is the first element in the list (from left to right) for which the given predicate yields true; if no such element exists, `None` is returned
`tabulate`		`fa(a) Nat * (Nat -> a) -> List a`	`tabulate(`*n*`,` *f*`)` returns the list `[`*f*`(0),` *f*`(1), ... ,` *f*`(`*n*`-1)]`

Name	Fixity	Sort	Description
`firstUpTo`		`fa(a) (a -> Boolean) -> List a -> Option (a * List a)`	returns `Some(`*e*`, `*x*`)` where *e* is the first element in the list (from left to right) satisfying the given predicate and *x* the initial list segment preceding *e*; if no such element exists, `None` is returned
`splitList`		`fa(a) (a -> Boolean) -> List a -> Option (List a * a * List a)`	returns `Some(`*x*`, `*e*`, `*y*`)` where *e* is the first element in the list (from left to right) satisfying the given predicate, *x* the initial list segment preceding *e*, and *y* the list tail following *e*; if no such element exists, `None` is returned
`locationOf`		`fa(a) List a * List a -> Option (Nat * List a)`	`locationOf(`*s*`, `*t*`)` returns `Some(`*n*`, `*x*`)` where *n* is the first position in list *t* (counting from from left to right) where list *s* occurs as a contiguous sublist, and *x* the list tail segment following *s* in *t*; if *s* does not occur in *t*, `None` is returned
`compare`		`fa(a) (a * a -> Comparison) -> List a * List a -> Comparison`	compares two list using the comparison function given as first parameter
`show`		`fa(a) String -> List String -> String`	`show(`*s*`, `*x*`)` returns the element strings in *x* concatenated, with string *s* inserted between any two elements

B.7. Compare

Sort

```
sort Comparison = | Less | Equal | Greater
```

Ops

Name	Fixity	Sort	Description
`compare`		`Comparison * Comparison -> Comparison`	compares comparison values
`show`		`Comparison -> String`	converts comparison value to string

B.8. Option

Sort

```
sort Option a = | Some a | None
```

Ops

Name	Fixity	Sort	Description
`some`		`fa(a) a -> Option a`	`op` that constructs `Some` *x*
`none`		`fa(a) Option a`	`op` that constructs `None`
`some?`		`fa(a) Option a -> Boolean`	tests if the parameter is of the form `Some` *x*

Name	Fixity	Sort	Description
`none?`		`fa(a) Option a -> Boolean`	tests if the parameter is `None`
`compare`		`fa(a) (a * a -> Comparison) -> Option a * Option a -> Comparison`	returns the result of the comparison of the two optional values, where `None` is less than `Some` *x* for all *x*; if both optional values are of the form `Some` *x*, the comparison function given as first parameter is used to compute the result
`mapOption`		`fa(a,b) (a -> b) -> Option a -> Option b`	applies the function given as first parameter to the optional value if it is `Some` *x*, otherwise `None` is returned
`show`		`fa(a,b) (a -> String) -> Option a -> String`	converts optional value to string; if the optional value is `Some` *x*, it uses the function given as first parameter to convert *x* to a string

B.9. Functions

Sorts

```
sort Injective(a,b) = ((a -> b) | injective?)

sort Surjective(a,b) = ((a -> b) | surjective?)

sort Bijective(a,b) = ((a -> b) | bijective?)
```

Ops

Name	Fixity	Sort	Description
id		fa(a) a -> a	identity function
o	infixl 24	fa(a,b,c) (b -> c) * (a -> b) -> (a -> c)	function composition
injective?		fa(a,b) (a -> b) -> Boolean	injectivity predicate; non-constructive
surjective?		fa(a,b) (a -> b) -> Boolean	surjectivity predicate; non-constructive
bijective?		fa(a,b) (a -> b) -> Boolean	bijectivity predicate; non-constructive
inverse		fa(a,b) Bijective(a,b) -> Bijective(b,a)	inverts bijective function; non-constructive