Perfect Developer

Language Reference Manual

Version 5.0, September 2011

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

1. Introduction

1.1 Purpose of this manual

This manual is the definition document for the Perfect Developer Language (*Perfect*). Newcomers to *Perfect* should use it in conjunction with a tutorial such as the one to be found on the Escher Technologies Ltd. web site.

1.2 Status of this edition

The syntax and semantics given in this edition of the *Perfect Developer Language Reference Manual* are intended to be final except for the following areas which need to be completed:

- Various items flagged [TBD] are subject to review
- Further Static Constraints [SC] and Verification Conditions [PO] need to be documented
- Further clarification may be needed in some areas

1.3 Organization

This manual is structured such that each chapter builds upon the syntax described in previous chapters, with minimal need for forward reference.

Notes enclosed in square brackets have various purposes, as follows:

[TBD] - indicates an area of the language design subject to review.

[SC] - indicates a static constraint which the compiler must check; it must report an error if the condition is not satisfied.

[PO] - indicates a verification condition (or *proof obligation*) which must be generated by the compiler and passed to the theorem prover for proving.

[IMP] - implementation note.

1.4 Syntax used to describe the grammar of Perfect

Character sequences appearing in bold are keywords of the language and represent themselves.

Character sequences appearing in italics represent nonterminal symbols of the language.

Strings of one or more punctuation characters within double quotes represent themselves.

Where a construct appears within square brackets, the construct is optional. If the opening square bracket is preceded by an asterisk, the construct may appear zero or more times in sequence at that point.

Two or more constructs separated by a vertical bar indicates that the constructs are alternatives. Round brackets may be used to delimit the scope of the vertical bar.

The colon separates the left hand side of a production from the right hand side. The semicolon separates multiple alternate right-hand sides from each other (i.e. it is similar to a vertical bar but has a lower precedence).

Comments in the grammar are introduced by a hyphen and terminated by end-of-line.

For example, the lexical form of a *Perfect* identifier can be described as:

> *Identifier* :
>   (*letter* | "_") *[*letter* | *digit* | "_"].

2. Goals and principles of the language

2.1 Design goals

The main design goals of the *Perfect* language are as follows:

- Provability. It should be possible to construct proofs of program correctness.
- Expressibility. The language should be sufficiently expressive for a very wide range of problems.
- Safety. The language must be safe to use for the development of critical systems.
- Object-oriented support. The language should support object-oriented development without forcing object-oriented methods on the user where they convey no benefit.
- Portability. There should be no implementation-dependent language features. Any implementation-dependent limits (other than run-time memory and other resource limitations) should be such that the compiler can report if they are exceeded. If a program which does not refer to features of the environment compiles successfully under two implementations, and its specification is deterministic, it should produce identical results in both cases (except for execution time differences). [Note: this requirement may be relaxed in the case of floating-point types, as a concession to execution efficiency, and in recognition that hardware conformance to the IEEE standard is of variable quality.]
- Clarity. The language must be strongly typed and make it clear when the value of a variable or parameter is changed.
- Productivity. An experienced user should be able to construct correct programs rapidly, and to read programs written by others.
- Ease of learning. The language should not be difficult for a typical analyst/programmer to learn. Mastery of *Perfect* should be easier to achieve than mastery of C++.
- Implementability. It must be possible to translate *Perfect* programs to run efficiently in a wide range of computer environments.

2.2 Principles

Major principles of *Perfect* include the following:

- Specification of entities is done separately from describing their implementations. Descriptions of implementations are optional in those cases where the translator is capable of generating an implementation.
- Minimisation of side-effects (to promote provability). The evaluation of an expression is not permitted to have side-effects apart from the creation of new objects. All changes to variables must be explicit (i.e. the variable must be identified and the context must clearly indicate that its value is changed).
- Reduced need for explicit pointers in comparison to traditional programming languages. This is because the use of pointers makes programs very hard to reason about (due to the pervasive nature of aliasing and side-effects which abound).
- Economy of concepts (to promote ease of learning).
- Orthogonality. Language constructs may be used in combination and their semantics are combined naturally.
- Automatic storage management (to promote productivity and correctness).

- Minimal implicit type conversions (to promote correctness, clarity and safety).
- Conciseness of frequently-used constructs and symbols (to promote productivity). Most keywords are abbreviated (usually to a single syllable). Infrequently-used keywords are written in full (or at least have longer abbreviations) for clarity.
- The meaning of the program does not depend in any way on characters that are typically invisible when the source is printed out or viewed using an editor. For example, non-displayable control characters are illegal; blanks and tabs at the end of a line have no significance. This promotes clarity and safety.

2.3 Context

It is assumed that in the future, *Perfect* developers will work in the context of a document-centred Integrated Development Environment (although it is also required that any *Perfect* program can be expressed in plain ASCII text form for listing and porting purposes). For this reason, *Perfect* differs from some other programming languages in the following ways:

- A wider range of symbols is used when displaying programs; however, since users will generally have keyboards supporting standard character sets, such symbols have simple equivalents (e.g. the predecessor operator will normally be displayed as a downward-pointing arrow, but the equivalent symbol for this is "<").
- The concept of a file containing part of a complete program is not central to *Perfect*. Information hiding is intended to be done by the IDE, not by splitting a specification into separate parts in different files. However, a traditional multiple-file compilation model is also supported (initially, it is the only model supported).
- Since a *Perfect* program may be held as a single large document, the language allows for the compiler to recompile only affected sections of the document when changes are made.
- Most declarations may be forward-referenced without restriction.
- The grammar does not require the parser to distinguish identifiers used as type names from other identifiers (making it possible to parse portions of *Perfect* text without the need for context information).

3. Lexical form

3.1 Overview

Although *Perfect* programs are normally represented using an extended character set in an integrated development environment, they can be represented in the standard ASCII character set or in Unicode. This chapter describes how the character set is used to express the various tokens of the language.

3.2 Character set

The character set used by *Perfect* comprises the letters A through Z and a through z, the digits 0 through 9 and the following special characters:

> . , ; : ? ! ' " ` + - * / % & | ( ) { } [ ] < > ~ # ^ _ = \ @

[SC] Other printable characters supported by the underlying character set are legal only in comments and in character and string literals. Nonprintable characters other than those characters or character combinations used to represent space, newline and horizontal tab are illegal in *Perfect* text, except that a nonprintable character or character combination representing end-of-file may be present at the very end of the program if allowed by the underlying file system and character representation.

It is recommended that when printing or displaying *Perfect* text, tab stops are considered to exist every 4 space-character widths from the left hand margin.

[Note: the character "$" is the only printable 7-bit character in the ASCII set that is not used.]

3.3 Comments

Comments are introduced by two adjacent forward slash characters ("//") and terminate at the end of the line. The last line in a file is always considered as having an end, even if there is no end-of-line marker before the end of the file.

3.4 White space

Comments, and newline, space and tab characters (other than those within comments, and space characters within string and character literals) are collectively known as whitespace. Multiple adjacent whitespace elements are equivalent to a single whitespace.

Whitespace may occur between any two program tokens but not within an identifier, literal, reserved word or multi-character token. Whitespace may, however, appear between two tokens that construct a new operator from an existing one according to the rules of the language. Whitespace must occur between a pair of adjacent tokens if the beginning of the second would otherwise be a legal continuation of the first (e.g. between a reserved word and an identifier).

3.5 Multi-character tokens

The multi-character tokens of the language are:

> **<= >= << >> <<= >>= <== ==> <==> ‖ ~~ ^= :- :: -> <- <-> ++ -- ** ## ..**
> **...**

Where an input character sequence can be interpreted in more than one way, the lexical analyser picks the longest leading sub-sequence that forms a token, then applies the same rule to the remainder of the input sequence. For example, "=>>" would be interpreted as "=>"followed by ">", not as "=" followed by ">>", even if the former interpretation gave rise to a parsing or other error message and the latter did not.

3.6 Reserved words

The reserved words of the language are:

> **abstract absurd after any anything as assert associative axiom bag begin bool build byte catch change char class commutative confined const decrease deferred define done early end enum exists external false fi final float for forall from function ghost goto has heap highest if idempotent identity implements import in inherits int interface internal invariant is it keep let like limited loop lowest map name nonmember null of on opaque operator out over pair par pass post pragma pre proof property public rank real redefine ref repeated require result satisfy schema selector self seq set storable super supports tag that then those throw total trace triple true try until value var via void when within yield**

Reserved words are case-sensitive. Note that **catch**, **float**, **implements**, **supports**, **throw**, **trace** and **try** are not used at present, but are reserved for future use.

The following words are not reserved but are names for built-in global methods and are therefore best avoided:

> debugHalt debugPrint flatten interleave loadObject max min storeObject swap

Similarly, the following words are names for built-in classes and are also best avoided:

> ByteData ByteStream CharDecoder CharEncoder CharEncoderDecoder
> Comparator DebugType Environment FileAttribute FileError FileHandle
> FileMode FileModeType FilePath FileRef FileStats FileStream GuardedObject
> InputStream nat OsInfo OsType OutputStream ReverseComparator SerialError
> SerialErrorType SimpleComparator Socket SocketError SocketMode
> StandardInputStream StandardOutputStream Storable StreamBase StreamHeap
> string Time

3.7 Identifiers

Identifiers comprise a letter or underscore character optionally followed by any number of characters each of which is a letter, digit or underscore character, provided only that the resulting string is not a reserved word. There is no limit on the length of an identifier. All characters in an identifier are significant. The case of letters is significant.

3.8 Character literals

Literals of type char are written as the desired character between opening-single-quote symbols thus:

> `` `a` ``

The backslash character has a special meaning within character literals in that the backslash character and one or more of the characters following it are replaced by a single character, as follows:

\a	alert (bell)
\b	backspace
\f	form feed
\n	line feed
\r	carriage return
\t	horizontal tab
\v	vertical tab
\\	\
\`	`
\"	"
\(*ddd*)	the character represented by the integer literal *ddd*

In the case of the form `` `\(ddd)` ``, *ddd* is any integer literal such that the resulting integer is within the range appropriate to the character set in use. There must be no whitespace between the brackets and the integer literal.

The use of any other character following the initial backslash is illegal. The amount of storage associated with each character and the character set supported are implementation dependent (a typical implementation might offer a choice of ASCII or Unicode).

[SC] Exactly one printable character, space character or backslash combination equivalent to one character

must appear between the quotes. If the `\(*ddd*)` form is used then the integer literal *ddd* must in be the range of the supported character set.

3.9 String literals

Literals of type "sequence of characters" (**seq of char**) are written as a sequence of characters enclosed in double quotes. The backslash character has the same special meaning as it does in character literals and every backslash sequence gives rise to a single character in the string. The closing quote must be on the same line as the opening quote.

Within a character or string constant, nonprintable characters (including newline and tab characters) are not permitted, and comments are not recognised.

[SC] The sequence between the quotes must comprise only printable characters, space characters and valid backslash sequences.

3.10 Integer literals

An integer literal is written as a sequence of decimal digits, or as a sequence of hexadecimal digits (0-9 and A-F or a-f) preceded by **0X** or **0x**, or as a sequence of binary digits preceded by **0B** or **0b**. There is no fixed limit on the size of integer literals, however if the compiler or target uses bounded integers, an error message will be generated in respect of any integer literal that cannot be represented. The case of any letter forming part of an integer literal is not significant.

Underscore characters may be inserted within the sequence of digits (but not at the start or the end) to improve readability.

3.11 Real literals

Real literals are written in the form *s*.*s* or *s*e*s* or *s*.*s*e*s* where *s* is any sequence of decimal digits and *e* is the letter **e** or the letter **E** optionally followed by a minus sign. The digit string following **e** or **E** is interpreted as a decimal exponent. White space is not permitted within a real literal. Each digit string *s* may contain embedded underscore characters to improve readability.

4. Classes and Types

4.1 Overview

Perfect provides a number of fundamental data types, from which more complex types may be built.

4.2 Concepts of type

4.2.1 Classes

In *Perfect* the term class is used to mean a set of allowed values which is not a proper subset of any other *Perfect* class nor a union of types.

There are eight predefined non-template classes in *Perfect* (**anything**, **int**, **real**, **bool**, **char**, **byte**, **void**, and **rank**). Note that **int** is not considered to be a subset of **real**, and **byte** is not considered to be a subset of **int**.

Each instance in a program of the keyword **enum** or the keyword **tag** creates a new class which is distinct from all other classes (including classes created by identical declarations elsewhere in the program).

Each occurrence of an abstract class declaration also creates a new class that is distinct from all other classes (including classes created by identical class declarations elsewhere in the program).

4.2.2 Types

In *Perfect* a type defines a set of allowed values and may be a class, a class associated with a constraint, or a union of types.

A constraint is a predicate that takes a single parameter representing values of the class and returns **true** for values that are permissible values of the type.

When binding occurrences of functions and operators to their declarations, most constraints are ignored, however, constraints within template parameters must match exactly.

Declaring entities with constrained rather than unconstrained types has the following effects:

4.2.2.1 Constraints on variables and data members

When declaring a variable or a data member of a class, there is a proof obligation that the constraint is satisfied whenever the value of the variable or data member is changed, thereby increasing the degree of validation performed.

A second effect of constraining a variable or data member is that it permits the code generator to allocate a more efficient form of storage. For example, an integer variable with a constraint that the value lies in the range 0 to 100 permits the code generator to use a simple fixed-length binary representation instead of the more general format used to store unbounded integers.

4.2.2.2 Constraints on parameters

When declaring the type of a parameter in a function, operator or schema declaration, the constraint becomes a part of the precondition of that function, operator or schema. It also allows the code generator to optimize the mechanism used for parameter passing.

4.2.2.3 Constraints on results

When declaring the type of a function or operator result, or of a schema parameter which is modified by the schema, the constraint gives rise to an additional proof obligation (i.e. that the result or final value of the parameter belongs to the specified type). It may also allow the code generator to optimize the mechanism used to return the result.

4.2.2.4 Constraints on bound variables

When declaring the type of a bound variable (i.e. following one of the keywords **exists**, **forall**, **that**, **any** or **for**), the effect is that the bound variable ranges only over the permitted values, instead of over all values of the class.

4.3 Predefined classes

The predefined classes of *Perfect* are as follows:

anything	The base class for all other classes, containing member function "toString"
bool	The Boolean class, comprising the values **true** and **false**
byte	The class of eight bit bytes
char	The character set supported by the environment (including control characters)
int	The (unbounded) positive and negative integers (including zero)
real	The real numbers. These are represented using the double-precision (64-bit) format defined in standard IEC559:1989 (IEEE 754)
void	The void type, comprising the single value **null**
rank	The enumeration comprising the values "below", "same" and "above" (in that order)

The base class **anything** is **deferred**; all other predefined classes are **final** classes (see Chapter 7 for the definitions of these terms).

4.4 Predefined types

There are two predefined types that are not classes:

nat	Subset of **int** consisting of unbounded positive integers and zero
string	Equivalent to **seq of char**

4.5 Literals for predefined classes

The section on lexical form has already described character literals (which have type **char**), string literals (which have type **seq of char**), integer literals (which yield non-negative values of class **int**) and real literals (which yield non-negative values of class **real**). Negative literals of class **real** and **int** are not directly represented but suitable constants can be constructed from positive literals and the negation operator.

Literals of the **bool** and **void** classes are written directly (i.e. as **true**, **false** and **null**).

4.6 Class declarations

New classes may be added by three mechanisms: enumeration, tag generation, and abstract class declaration. Each such class must be named at the point of generation using the syntax:

> *ClassDeclaration*:
>   *EnumerationDeclaration*;
>   *TagDeclaration*;
>   *AbstractClassDeclaration*.

4.6.1 Enumeration generator

Classes may be generated by enumeration of values. The syntax for this is:

> *EnumerationDeclaration*:
>   **class** *Identifier* "**^=**" **enum** *Identifier* *["**,**" *Identifier*] **end**.

An enumeration is collection of values having an ordering relation. The **lowest** and **highest** operators (see section 5.4.1) may be applied to the enumeration class to obtain the first and last elements respectively. Predecessor and successor operators may be applied to values of enumerated classes (except for the first and last elements respectively). The comparison operator "<" is defined such that each value is considered to be less than all values declared later in the enumeration list.

Every enumeration in a program yields a new class. Two enumerations with identical lists of values yield distinct classes. The identifiers in the list are treated as non-member constants of the enumeration class, and thus must always be referred to by using the enumeration class name to resolve the scope of the value name. For example, given the following declaration:

> **class** Color = **enum** red, green, blue **end**

then a value would be referred to as e.g. *Color red*. See section 5.4.14 for more on scope resolution.

4.6.2 Tag generator

The tag generator yields a class comprising a set of ordered un-named values. It may be thought of as an enumeration where we have asked the compiler to generate the names for us. The syntax is:

> *TagDeclaration*:
>   **class** *Identifier* "^=" **tag** ["(" *Expression* ")"].

The optional expression (which must yield a positive integer) is the number of distinct values we require. If no expression is given, the set of distinct values provided is sufficiently large that it will not be exhausted at run-time before some other resource (e.g. memory) is exhausted.

The usual comparison operators are defined on the values. The **lowest** operator may be used to obtain the lowest tag available and the predecessor and successor operators may be used on tag values. If an expression is given, the **highest** operator may be used to obtain the highest member of the set.

Every tag constructor yields a (notionally) unique set of values.

[TBD: should we require the expression to be constant at run-time throughout the scope of the declaration? Or constant at compile-time?]

4.6.3 Abstract class declarations

These are covered fully in Chapter 7.

4.7 Type expressions

Types may be expressed in terms of classes and other types in a number of ways:

- By using a non-template class name
- By providing a class template name with type or class names to use as actual parameters, thereby instantiating it
- By associating an existing type or class name with a constraint
- By uniting two or more types
- By forming the union of all abstract classes that inherit from a particular abstract class

The full syntax for type expressions is:

> *PossConstrainedTypeExpression*:
>   *ConstrainedTypeExpression*;
>   *TypeExpression*.

> *ConstrainedTypeExpression*:
>   **those** *Identifier* ":" *TypeExpr1* ":-" *Predicate*;
>   *TypeExpr3 ComparisonOperator Expr4*;
>   "(" *ConstrainedTypeExpression* ")".

TypeExpression:
 TypeExpression "||" *TypeExpr1*;
 TypeExpr1.

TypeExpr1:
 ref *TypeExpr2* **on** *Identifier*;
 TypeExpr2.

TypeExpr2:
 from *ClassName*;
 TypeExpr3.

TypeExpr3:
 "(" *TypeExpression*")";
 ClassName.

ClassName:
 TemplateName **of** *ActualTemplateParameters*;
 TypeName.

TypeName:
 anything;
 bool;
 byte;
 char;
 int;
 rank;
 real;
 Identifier.

TemplateName:
 bag;
 map;
 pair;
 seq;
 set;
 triple;
 Identifier.

ActualTemplateParameters:
 TypeExpr1;
 class *Identifier*;
 "(" *TypeExpression* *[*Separator TypeExpression*] ")".

Separator:
 ",";
 "->";
 "<-";

"<->".

ComparisonOperator:
  "~" *CompareOp*;
  CompareOp.

CompareOp:
  "=";
  ">";
  "<";
  ">=";
  "<=";
  ">>";
  "<<";
  ">>=";
  "<<=";
  in.

4.8 Instantiating class templates

A class template is instantiated by following its name with the keyword **of** and a list of class parameters, which are types. If there is only one parameter and it is a type name or template instantiation, it need not be enclosed in brackets, otherwise brackets are required. The separators in the parameter list must match the separators in the corresponding template class declaration.

4.9 Constrained types

A type may be formed by constraining an existing class or type. Two forms of syntax are available for expressing this.

The more general form is "**those** *BoundVariableDeclaration* :- *Predicate*" where *Predicate* is a function of the bound variable. Since the **those** construct is delimited by the end of the type expression, if this form is used within a larger type expression, either it must be the last element, or it must be enclosed in brackets. An example of this form would be "**those** x: nat :- x <= 100".

The abbreviated form is "*TypeExpr4 CompareOp Expr4*" which is equivalent to the full form "**those** xx: *Type* :- xx *CompareOp Expr4*". So the above example could be written "nat <= 100".

Another abbreviated form is possible in some contexts (see <u>section 6.4</u>).

4.10 United types

Sometimes the value of a variable, parameter or function return may belong to one of several types. In this case we can create a united type using the uniting operator "||"; for example "nat || **void**". The "||" symbol is pronounced "united with" or simply "or".

The uniting operator is commutative and associative, for example "nat || (**void** || **char**)" yields the same union as "**char** || nat || **void**".

[SC] The operands of "||" must be disjoint.

4.11 Union of all derived classes

Sometimes it is desirable to express the type comprising the union of the values of all classes derived from some base class (including the values of the base class itself). This is expressed using the notation "**from** *ClassName*", where *ClassName* is the name of a class or a class template instantiation.

4.12 Reference types

An entity of a reference type refers to some value created on a heap. Other entities of reference type may refer to the same value; any changes to the value are therefore visible to all reference entities which refer to it.

When declaring a reference entity, the type (or types) of the value to which it refers must be declared, together with the name of the heap on which the entity will be created.

Note that the use of reference types is the sole mechanism for creating aliases.

4.13 Type naming

It is possible to introduce new names for types, with optional generic parameters. The syntax for this is:

> *TypeDeclaration*:
>   **class** *Identifier* [**of** *FormalTemplateParameters*] "**^=**" *PossConstrainedTypeExpression*.

> *FormalTemplateParameters*:
>   *Identifier*;
>   "(" *Identifier* *[*Separator Identifier*] ")".

For example, we could declare:

>

> **class** Word **^= seq of char**

or:

> **class** ListOfLists **of** X **^= seq of seq of** X

4.14 Predefined class templates

Predefined class templates are provided for six common ways of collecting values of similar types: sets, bags, sequences, pairs, triples and mappings.

4.14.1 Sets

A set comprises a collection which includes none, some or all the values of its base type. It is meaningless to speak of a value occurring in the set more than once. For example, if we consider the set of prime numbers, every positive integer either does or does not belong to the set.

To build a set in a non-**ghost** context (i.e. in a context where code must be generated) requires that the base type has a non-**ghost** equality operator. See <u>section 7.1.7</u> for a full explanation of when the equality operator of a type is **ghost**.

The name of the class template is **set** so that, for example, "**set of char**" is the type expression for a set of characters.

4.14.2 Bags

A bag is a collection of values in which it does make sense to ask how many times a value occurs. For example, if we were interested in collecting statistics on examination results but wished to preserve the anonymity of the candidates concerned, we would be interested only in the marks obtained. Since multiple candidates might score the same marks and we wish to record this, the appropriate data type would be a bag of examination marks.

Like a set, a bag may only be constructed in a **ghost** context, or with a base type with non-**ghost** equality.

We construct a bag type using the template name **bag**, e.g. "**bag of nat**".

4.14.3 Sequences

The third type of collection is the sequence, or list of values. Sequences are used where the order of values is important. We construct a sequence type using the template name **seq**, e.g. "**seq of char**". The same value may occur more than once in a sequence. There is no requirement on the equality operator for constructing sequences.

4.14.4 Pairs

The class "**pair of** (X, Y)" is used to describe and construct mapping classes. The abstract data members of a pair are a variable of type X named "x" and a variable of type Y named "y".

4.14.5 Triples

The class "**triple of** (X, Y, Z)" is similar to "**pair**". The abstract data members of a triple are variables of types X, Y and Z named "x", "y" and "z" respectively.

4.14.6 Mappings

A mapping has a set of values (the domain) paired with a bag (the range) such that every member of the domain is associated with exactly one member of the range. Mappings are useful for describing lookup tables and relational databases. We express a mapping class using an arrow-symbol pointing from the type on the

left (the domain type) to the type on the right (the range type) as the separator in the class parameter list, e.g. "**map of** (**char** -> nat)".

Like sets and bags, a mapping in a non-**ghost** context requires a non-**ghost** equality operator on both the domain and range types.

Note that a sequence can be regarded as a mapping from a range of integers to some other type. This is reflected in the fact that the indexing and domain member functions may be applied to both sequences and mappings. However, the class "**seq of** X" is considered distinct from "**map of** (nat -> X)" and all its subclasses.

The abstract data of "**map of** (X ->Y)" is "**those** v:**set of pair of** (X, Y) :- (**for** x::v.rep(1) **yield** x.dom).unique".

4.15 Type compatibility

4.15.1 Overview

The presence of unions and subtyping in the type system of *Perfect* gives rise to various ways in which two types can be related. All semantic rules governing the use of types can be expressed in terms of four fundamental binary relations on types: *disjoint* (have no common values), *same* (are equivalent), *overlapping* (have a common value) and *nested* (the first type is a subtype of the second). Three of the relations are symmetric; the only exception is 'nested'. Interdependencies between the four relations are shown on the following Venn diagram where each point represents an ordered pair of *Perfect* types:
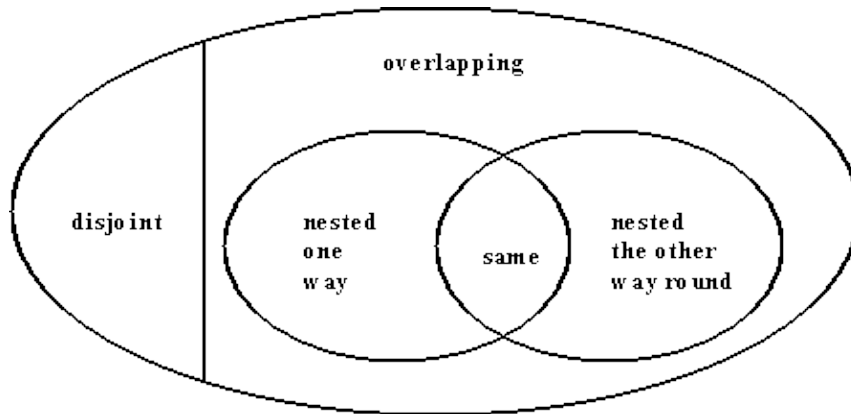
For example, if the type given by type expression X is nested in the type given by Y then types X and Y are also overlapping, not disjoint and can be (but do not have to be) equivalent.

In simple cases the way in which two given types are related trivially follows from their definitions. However, the high expressive power of *Perfect* means that it is possible to define types that are complex enough to make relations between them less than obvious. Therefore, each of the four fundamental relations is given an unambiguous formal definition. The role of these definitions is similar to that of the formal grammar of the language: both can be used to verify validity of arbitrarily complex language constructs and both are instrumental in the compilation process. The latter role of formal type relations has some far-reaching implications: for example, in *Perfect* it is possible to guarantee that any conceivable call to an overloaded

function will be unambiguous, i.e. no call can possibly type-match more than one variant of the function declaration. This is not even a proof obligation - the "collective consistency" of overloaded declarations is checked by the type inference logic of the compiler based on the formally defined type relations, and any potential ambiguity is reported as an error.

4.15.2 Fundamental type relations

Each type relation is defined by means of a binary predicate on types: for example, the type given by expression X is regarded as a subtype of the type given by expression Y if and only if nested(X,Y) is true. Each of the four predicates is, in turn, defined by collections of clauses; the clause that applies to a particular case is chosen by means of pattern matching according to the syntactical structure of the operands (*Perfect* type constructors are shown in the definitions in bold face, as is the auxiliary tag 'strict' that has no counterpart in *Perfect* syntax). Upper-case letters in patterns denote arbitrary type expressions while lower-case Latin letters denote type names. Symbol *x* in rules (n1) and (n2) represents a polymorphic type name introduced in the declaration of a polymorphic function: in actual *Perfect* programs this position will be occupied by an identifier. It is possible for a pair of parameters to match more than one clause in the same predicate definition. Such ambiguities are resolved by textual precedence: for example, nesting of a union into another union is governed by the rule (n8) rather than (n9) or (n12). Definitions of all four fundamental type relations are complete, as each predicate definition contains a clause that will match any pair of parameters.

Type namings without constraints do not affect relations between types, and are therefore removed before any type relation is evaluated. In addition, constraints are removed except from within template parameters. For example, "**seq of** string" becomes "**seq of seq of char**", whereas "**seq of** nat" remains the same.

Where removing all constraints from within template parameters would change the result of evaluating a type relation we say that the relation is not defined. For example, "same(**seq of** nat, **seq of int**)" is undefined. When we say "X *relation* Y" we mean the relationship is true, not false or undefined.

Further to that, the following transformations are assumed to be performed before applying the type relation rules:

- All unions are flattened: "(a || b) || (c || d)" is replaced by "a || b || c || d"
- Instantiations of templates by a single class are rewritten in the bracketed form: "t **of** x" is replaced by "t **of** (x)"
- All occurrences of polymorphic type names in declarations are prefixed by the keyword "**class**": "f(a:**seq of class** x, b:**ref** x)" is replaced by "f(a:**seq of class** x, b:**ref class** x)"

The purpose of these transformations is to get rid of syntactical sugar and to represent the type expressions in a uniform format; this reduces the number of clauses required to define the fundamental predicates without affecting their semantics. Actual implementations of *Perfect* do not have to transform the types and may use more complex systems of rules instead, as long as the final result stays the same in all cases.

Auxiliary binary predicate 'derived' is defined on type names and reflects the inheritance hierarchy that exists in the program. Namely, derived(a,b) yields true if and only if both 'a' and 'b' denote classes and class 'a' is a descendant of class 'b' or coincides with it.

Predicate disjoint(type1, type2): no value can belong to both type1 and type2

This predicate is simply a logical negation of overlapping:

(d1)   disjoint(A, B) == ~ overlapping(A, B)

Predicate same(type1, type2): type1 and type2 are equivalent

Equivalence is defined as mutual nesting (see the diagram):

(s1)   same(A, B) == nested(A, B) & nested(B, A)

Predicate overlapping(type1, type2): a value can belong to both type1 and type2

A type overlaps with a union if it overlaps with at least one of its members:

(o1)   overlapping(A, B_1 || B_2 || ... || B_k) == **exists** i::1..k :-overlapping(A, B_i)

It does not matter whether a union occurs as the first or the second argument, as 'overlapping' is a symmetric relation:

(o2)   overlapping(A_1 || A_2 || ... || A_k, B) == overlapping (B, A_1 || A_2 || ... || A_k)

There are no other non-trivial cases of overlapping; the rest can only be clean nesting. (Note that because of the absence of multiple inheritance, two 'from' families cannot overlap without being nested.)

(o3)   overlapping(A, B) == nested(A, B) or nested(B, A)

Predicate nested(type1,type2): all values of type1 also belong to type2

The first four rules deal with polymorphic type names in function declarations. When a function call is checked for compliance to a function declaration (which is essentially parameter-wise type nesting plus fulfilling the constraints), polymorphic type names can occur on the declaration side only. However, because of the symmetric way in which equivalence is defined in (s1), rules (n2) and (n4) are also necessary. Note that rules (n1) and (n2) have side-effects: all further occurrences of the polymorphic type name in question are replaced by the opposite operand prefixed by a special tag "**strict**". Together with rules (n3) and (n4) this enforces the requirement that all types occurring in the positions of a function call that correspond to different occurrences of the same polymorphic type name must be equivalent.

(n1)   nested(A, **class** X) == **true**, **class** X <- **strict** A
        nested(**class** X, A) == **true**, **class** X <- **strict** A

(n3)   nested(A, **strict** B)   == same (A, B)

(n4)   nested(**strict** A, B) == same (A, B)

Trivial cases with type names and derived families:

(n5)   nested(a, b) == a = b

(n6)   nested(a, **from** b) == derived(a, b)

(n7)   nested(**from** a, **from** b) == derived(a, b)

A union can only be nested in other type if all its members are nested in that type:

(n8)   nested(A_1 || A_2 || ... || A_k, B) == **forall** i::1..k :- nested(A_i, B)

The only way for a (non-union) type to be nested in a union is to be nested in one of its members:

(n9)   nested(A, B_1 || B_2 || ... || B_k) == **exists** i::1..k :- nested(A, B_i)

References are not transparent for subtyping:

(n10)   nested(**ref** A, **ref** B) == same(A, B)

Two template instantiations are either disjoint or equivalent:

(n11)   nested(a **of**(A_1, ..., A_k), b **of**(B_1, ..., B_n)) == a = b & k = n & **forall** i::1..k :- same(A_i, B_i)

There are no other ways in which one type can be nested into another:

(n12)   nested(A, B) == **false**

4.15.3 Examples

Example 1

Is "**seq of** b || **seq of** d" a disjoint union, provided class "d" is a descendant of class "b" ?

	disjoint(**seq of** (b), **seq of** (d))	(d1)
=	~ overlapping(**seq of** (b), **seq of** (d))	(o3)
=	~ (nested(**seq of** (b), **seq of** (d)) | nested(**seq of** (d), **seq of** (b)))	(n11)
=	~ (same(b,d) | same(d,b))	(s1)
=	~ ((nested(b,d) & nested(d,b)) | (nested(d,b) & nested(b,d)))	(n5)
=	~ ((**false** & **false**) | (**false** & **false**))	
=	**true**	

Therefore this is a valid union.

Example 2

Is "a || **int**" a subtype of "**int** || **real** || **from** a" ?

	nested(a || **int**, **int** || **real** || **from** a)	(n8)
=	nested(a, **int** || **real** || **from** a) & nested(**int**, **int** || **real** || **from** a)	(n9)
=	(nested(a, **int**) | nested(a, **real**) | nested(a, **from** a)) &	
	(nested(**int**, **int**) | nested(**int**, **real**) | nested(**int**, **from** a))	(n5)

=   (**false** | **false** | nested(a, **from** a)) & (**true** | **false** | **false**)   (n6)

=   (**false** | **true**) & **true**

=   **true**

Yes: the first of these types can be used where the second one is required.

Example 3

Is "**ref** (a || b)" equivalent to "**ref** a || **ref** b" ?

       same(**ref** (a || b), **ref** a || **ref** b)   (s1)

=   nested(**ref** (a || b), **ref** a || **ref** b) & nested (**ref** a || **ref** b, **ref** (a || b))

Starting with the left operand of 'and',

       nested(**ref** (a || b), **ref** a || **ref** b)   (n9)

=   nested(**ref** (a || b), **ref** a) | nested(**ref** (a || b), **ref** b)   (n10)

=   same(a || b, a) | same (a || b, b)   (s1)

=   (nested(a || b, a) & nested(a, a || b)) | (nested(a || b, b) & nested(b, a || b))

Let us deal with the leftmost term first:

       nested(a || b, a)   (n8)

=   nested(a, a) and nested(b, a)   (n5)

=   **true** and **false**

=   **false**

In the same way, nested(a || b, b) = **false** and therefore the answer to the original question is negative.

5. Expressions and Operators

5.1 Overview

Like traditional programming languages, *Perfect* provides a rich set of operators for constructing expressions. *Perfect* operators can be overloaded (i.e. the meaning of the operator depends on the types of its operands). To avoid ambiguity, no conversions between classes are automatically applied to operands, apart from widening conversions where the parameter is declared with a union type (note that **from** types are unions) and the actual parameter is of a type nested in that union.

However, when binding an occurrence of a function or operator to the correct definition, any operand of a type which is not a complete class will be treated as if it belongs to the complete class. For example, an operand of type **nat** will be treated as if it were of type **int**. Likewise, if the type naming is used to give a name to some original type with an added constraint, the operand will be treated as belonging to the original type (unless that type is also a type-naming, in which case this rule is applied recursively). Where necessary, verification conditions are generated to the effect that any constraints on the formal parameters are satisfied (this is the only place where the compiler generates an implicit type narrowing).

The full grammar for expressions is as follows:

> *Expression*:
>   *PrimableExpression*;
>   *UnprimableExpression*.
>
> *PrimableExpression*:
>   *PrimableExpr8*;
>   *PrimableCastExpression*.
>
> *UnprimableExpression*:
>   *UnprimableCastExpression*;
>   *TypeWideningExpression*;
>   *TypeEnquiryExpression*;
>   *TypeAssertionExpression*;
>   *SubjunctiveExpression*;
>   *ChooseExpression*;
>   *QuantifiedExpression*;
>   *TransformExpression*.
>
> *Expr0*:
>   *Expr0 BooleanImplicationOperator Expr1*;
>   *Expr1*.
>
> *Expr1*:
>   *Expr1* "|" *Expr2*;
>   *Expr2*.

Expr2:
    Expr2 "**&**" *Expr3*;
    Expr3.

Expr3:
    Expr3 ComparisonOperator *[*Expr3 ComparisonOperator*] *CompareExpr*;
    CompareExpr.

CompareExpr:
    Expr4 "**~~**" *Expr4*;
    Expr4 ["**~**"] [**in** | **like**] *Expr4*;
    Expr4.

Expr4:
    Expr4 AddingOperator Expr5;
    Expr5.

Expr5:
    Expr5 MultiplyingOperator Expr6;
    Expr6.

Expr6:
    Expr6 "**^**" *Expr7*;
    Expr6 "**..**" *Expr7*;
    Expr7.

Expr7:
    UnaryPrefixOperator Expr7;
    AllowedOverOperator **over** *Expr7*;
    TypeOp TypeName;
    PrimableExpr8;
    UnPrimableExpr8.

UnPrimableExpr8:
    UnprimableBracketedExpression;
    UnPrimableExpr8 "**[**" *Expression* "**]**";
    UnPrimableExpr8 "**.**" **value**;
    PrimableExpr8 "**'**";
    UnPrimableExpr8 "**.**" *IdentifierOrSuper* [*ActualParameterList*];
    ref *Expression* **on** *Identifier*;
    ConstructorExpression;
    Literal;
    "**?**".

PrimableExpr8:
    PrimableBracketedExpression;
    PrimableExpr8 "**[**" *Expression* "**]**";
    PrimableExpr8 "**.**" **value**;

PrimableExpr8 "**.**" *IdentifierOrSuper* [*ActualParameterList*];
GeneralIdentifierOrSuper [*ActualParameterList*];
self;
result;
it.

IdentifierOrSuper:
    [**super**] *Identifier*.

GeneralIdentifierOrSuper:
    ClassName Identifier;
    IdentifierOrSuper.

ActualParameterList:
    "**(**" *Expression* *[*Separator Expression*] "**)**".

ConstructorExpression:
    ClassName "**{** " [*Expression* *[*Separator Expression*]] "**}**".

Literal:
    IntegerLiteral;
    RealLiteral;
    StringLiteral;
    CharacterLiteral;
    BooleanLiteral;
    VoidLiteral.

BooleanLiteral:
    true;
    false.

VoidLiteral:
    null.

AllowedOverOperator:
    AddingOperator;
    MultiplyingOperator;
    "**^**"; "**..**".

BooleanImplicationOperator:
    "**==>**"; "**<==**"; "**<==>**".

AddingOperator:
    "**+**"; "**++**"; "**-**"; "**--**".

MultiplyingOperator:
    "*****"; "******"; "**/**"; "**%**"; "**%%**"; "**#**"; "**##**".

UnaryPrefixOperator:
  "/"; "%"; "#"; "*"; "+"; "-", "<"; ">"; "^"; "~".

5.2 Functions, Selectors and Constructors

Perfect provides three variations of the concept of function, together with constructors. All share the property of having no side-effects, i.e. they behave like mathematical functions.

A function takes one or more parameters and yields a result (in the case of a class member function, it is permitted for a function to take no parameters). An operator is identical to a function except that it is written as a symbol and takes exactly one or two parameters.

A selector is similar to a function except that it must be a member of a class, its return value must be a sub-object of the current object and a selector expression may appear primed in a postcondition; that is, the selector provides a mechanism to access a sub-object in such a way that it may be modified.

A constructor is used to build objects of a given class. It is a special form of function declared within a class and returning an object of that class.

Definitions of predefined global functions, and of functions, selectors and constructors declared by predefined classes are given in the *Library Reference*, which is Appendix A of this document.

5.2.1 Using functions and selectors

A function or selector is called by writing its name followed by its parameter list in brackets, if the function has any parameters; if there are no parameters, no brackets are required. Where there is more than one parameter, adjacent parameters are separated using any of the separators ",", "->", "<-" and "<->". The separators used in the actual parameter list must match those given in the formal parameter list (overloading of functions with identical argument types by using different separators is possible). For details of type compatibility of the parameters, and how to match **repeated** parameter groups, see section 6.11.

As an alternative to being given a name, a selector may be represented by square brackets, in which case it must take a single parameter. In this case it is called by following an expression of the type that the selector is a member of by the parameter enclosed in square brackets.

5.2.2 Using constructors

A constructor for a class is called by following the name of the class (including any template parameters) by a parameter list enclosed in curly brackets "{...}". The curly brackets are always used in a constructor call, even if the parameter list is empty (unlike the parameter list for function and selector calls). As with function and selector calls, a constructor may be declared with **repeated** parameters, see section 6.11.

5.3 Operators

Definitions of operators declared by predefined classes are given in the *Library Reference*, which is Appendix A of this document.

5.3.1 Unary operators

The following unary operator symbols are available.

Symbol	Typical use	Redefinable?	Comments
*		Yes	
/		Yes	
%		Yes	
#	Count	Yes	
+	Convert to integer	Yes	
-	Negation	Yes	
<	Predecessor	Yes	May be represented by (down-arrow) if the character set allows
>	Successor	Yes	May be represented by (up-arrow) if the character set allows
^		Yes	
~	Logical negation	No	

A unary operator is invoked by prefixing the operand symbol to the parameter.

5.3.2 Binary operators

The following binary operator symbols are available.

Symbol	Priority	Typical use	Prefix with "~"?	Redefinable?	Comments
[ ]	9	Indexing	No	Yes	Second operand is placed inside the brackets
^	8	Exponentiation	No	Yes	
..	8	Make sequence	No	Yes	Defined automatically for enumeration classes
*	7	Multiplication	No	Yes	

/	7	Division	No	Yes	
%	7	Remainder	No	Yes	
#	7	Count	No	Yes	
**	7	Intersection	No	Yes	
%%	7		No	Yes	
##	7	Disjoint	No	Yes	
+	6	Addition	No	Yes	Defined automatically for enumeration classes
-	6	Difference	No	Yes	
++	6	Union, concatenation	No	Yes	
--	6	Set difference	No	Yes	
~~	5	Comparison	No	Yes	Must return type **rank** and be defined so as to have the correct symmetry and transitivity properties. May be declared **total** in which case additional restrictions apply. Defined automatically (and **total**) for all enumeration classes.
in	5	Inclusion	Yes	Yes	
like	5	Type equality	Yes	No	Returns **true** if and only if the exact types of the operands are the same at run-time
=	4	Equality	Yes	No	
<	4	Less-than	Yes	No	Defined automatically from the definition of "~~"
>	4	Greater-than	Yes	No	Equivalent to "<" with operands reversed
<=	4	Less-than-or-equal	Yes	No	Defined automatically from the definition of "~~" provided

			Prefix with ~?		
					that the definition of "~~" is declared **total**
>=	4	Greater-then-or-equal	Yes	No	Equivalent to "<=" with operands reversed
<<	4	Strict inclusion	Yes	Yes	
>>	4	Strict reverse inclusion	Yes	No	Equivalent to "<<" with operands reversed
<<=	4	Inclusion	Yes	Yes	
>>=	4	Reverse inclusion	Yes	No	Equivalent to "<<=" with operands reversed
&	3	Logical 'and'	No	No	"a & b" is equivalent to "([~a]: **false**, []: b)"
|	2	Logical 'or'	No	No	"a | b" is equivalent to "([a]: **true**, []: b)"
==>	1	Implication	No	No	"a ==> b" is equivalent to "(~a) | b"
<==	1	Reverse implication	No	No	"a <== b" is equivalent to "a | ~b"
<==>	1	Equivalence	No	No	Same as "=" on Booleans but lower priority

A binary operator is invoked using infix notation; except that when invoking the "[]" operator, the first operand is placed on the left and the second operand is placed within the square brackets.

Where the column "Prefix with ~?" reads "Yes", the operator has a Boolean result and the operator may be prefixed by "~" to yield a similar operator producing the inverse result. For example, "a ~< b" has the same meaning as "~(a < b)".

The comparison operators (i.e. all the operators with priority 4) may share operands in expressions. Such expressions are expanded by duplicating the shared operand(s) and inserting "&" between each comparison. For example, the expression "a < b <= c < d" means the same as "(a < b) & (b <= c) & (c < d)".

The "[]" symbol may also be used to declare a selector.

5.3.3 Equality operator

The equality operator is treated specially in three ways:

- Equality is automatically defined by the system for all user-declared abstract classes. It is defined to yield **true** if all abstract data variables of the operands are correspondingly equal (excluding any variables within **when**-clauses whose guards yield **false**). Equality operators may not be defined by the user, although the user may provide a refinement (i.e. an implementation).
- The equality operator is not inherited; it is implicitly called by all descendent classes' equality operators. This means equality can be declared **ghost** in a class even if its parent class did not declare its equality as **ghost**. A consequence of this is that equality is normally **ghost** on the type "**from** T" unless T (or one of its ancestors) specifically declares it as non-**ghost**.
- It is permitted to apply the equality operator between two values of arbitrary types provided only that the static types of the left and right operands have a common subtype (i.e. an inspection of the types of the operands alone does not rule out the possibility of the values being equal).

5.3.4 Rank operator

The rank operator "~~" defines a partial ordering and is automatically defined by the system for all classes which do not have a user-defined relative rank operator.

Any declaration of the rank operator in some class *C* must ensure that the following relations hold for all expressions e1, e2 and e3 having type *C*:

- e1 = e2 ==> e1 ~~ e2 = same@**rank**
- e1 ~~ e2 = same@**rank** <==> e2 ~~ e1 = same@**rank**
- e1 ~~ e2 = below@**rank** <==> e2 ~~ e1 = above@**rank**
- e1 ~~ e2 = same@**rank** ==> e3 ~~ e1 = e3 ~~ e2
- e1 ~~ e2 = below@**rank** & e2 ~~ e3 = below@**rank** ==> e1 ~~ e3 = below@**rank**

The system also defines the binary operator "<&quot in terms of the rank operator using the following identity:

- e1 < e2 <==> e1 ~~ e2 = below@**rank**

Where the rank operator has been defined as **total** then only equal values may return same@**rank**, and the system will define the binary operator "<=" using the following identity:

- e1 <= e2 <==> e1 ~~ e2 ~= above@**rank**

5.3.5 Type comparison operator

The **like** operator compares the types of its operands at run-time and returns **true** if the actual types are the same, otherwise **false**. Both operands should be of united types (note that a **from** type expression is a united type), and the types of both operands must have a common subtype (so that a type match is possible).

5.3.6 Operator precedence

The relative precedence of operators in many programming languages is difficult to remember, leading to over-use of brackets. With this in mind, the precedence structure of *Perfect* has been kept simple. Operators are evaluated in the following order:

- Indexing and member are evaluated first ("[ ]" and ".").
- Unary operators come next.
- The exponentiation operators come next ("^" and "..").
- The "multiplicative" operators come next (i.e. "*", "**", "#", "##", "/", "%" and "%%").
- Then the "adding" operators ("+", "-", "++", "--").
- Then the **in**, **like** and rank ("~~") operators
- Then the comparison operators (anything which always produces a Boolean result, apart from **in** and **like**).
- Then the Boolean operators in the order: "and", "or", "implies/equivalent"

Unary operators group right-to-left (the only order which makes sense, e.g. "- # x"). All binary operators group left-to-right, except for the comparison operators, for which there is no grouping (because compound operators are formed instead).

5.3.7 Suggested operator pronunciation

When reading *Perfect* text, it is helpful to have a standard pronunciation of the more unusual operators and member names. Suggested pronunciations are:

Operator	*Suggested pronunciation*
#	"count" (or optionally "length" for the unary form with a sequence operand)
##	"disjoint"
^	"to-the-power-of" (or "exp")
..	"up-to"
++	"join" (optionally "cat" for sequences)
--	"diff"
**	"intersect"
> (unary)	"next" or "successor"
< (unary)	"previous" or "predecessor"
%	"modulo"
>>	"contains"
>>=	"includes"
<<	"is contained in"

<<=	"is included in"
~~	"compared with"

5.4 Other expression constructs

5.4.1 Operators on types

The operators **highest** and **lowest** may be applied to **char** or any enumeration type. Also, **lowest** may be applied to any tag type, and **highest** may be applied to any finite tag type. The **highest** operator yields that value such that there are no greater values in the type. The **lowest** operator yields that value such that there is no lesser value of that type. The syntax is:

> *TypeOperatorExpression*:
>   *TypeOperator Identifier*.

> *TypeOperator*:
>   **highest**;
>   **lowest**.

5.4.2 Brackets, temporary names, assertions and conditionals

Bracketed expressions may be used for the following purposes:

- To force a particular evaluation order
- To give names to subexpressions (in order to avoid repeating them, or to improve clarity)
- To insert assertions into expressions
- To express conditional expressions
- To generate multiple expressions to return from a function

The syntax for bracketed expressions is:

> *PrimableBracketedExpression*:
>   "(" *[LetDeclarationOrAssertion] PrimableExpression* ")".

> *UnprimableBracketedExpression*:
>   "(" *[LetDeclarationOrAssertion] PossMultipleExpressionOrChoice* ")".

> *LetDeclarationOrAssertion*:
>   **let** *Identifier* "**^=**" *Expression* "**;**";
>   *Assertion* "**;**".

> *Assertion*:
>   **assert** *PredicateList* [*Proof*].

PossMultipleExpressionOrChoice:
>   *PossMultipleExpression*;
>   *Choices*.

PossMultipleExpression:
>   *Expression* *["**,**" *Expression*].

Choices:
>   *GuardedExpression* *["**,**" *GuardedExpression*] ["**,**" *ElseExpression*];
>   **opaque** *GuardedExpression* "**,**" *GuardedExpression* *["**,**" *GuardedExpression*].

GuardedExpression:
>   *Guard Expression*.

Guard:
>   "**[**" *Predicate* "**]**" "**:**".

Predicate:
>   *Expression*.

ElseExpression:
>   *EmptyGuard Expression*.

EmptyGuard:
>   "**[**" "**]**" "**:**".

5.4.2.1 Let declarations

Identifiers introduced using **let** are in scope from the declaration until the closing bracket. A **let** declaration captures the value of the given expression and names that value.

5.4.2.2 Assertions

An assertion states a condition which must hold at that point. Every assertion generates a corresponding proof obligation.

5.4.2.3 Choice expressions

The choice expression replaces the conventional "if" and "case" constructs. To evaluate the expression, the guards are evaluated in the given order, until one is found to be **true**. The corresponding expression is then selected for evaluation. If none of the guards is **true** and an **else** part is present, the **else** part is selected; if no **else** part is present, the precondition of the construct is not satisfied.

For example, to compute the maximum of two expressions e1 and e2 we can use:

>   ([e1 >= e2]: e1, [e2 >= e1]: e2)

At least one of the guarded expressions must be of a type that matches or contains the types of all the other guarded expressions. The result type of a choice expression is that type.

Normally, the expression following the first guard that evaluates to **true** is chosen and any following guards are not evaluated. However, if the first guard is preceded by **opaque** then the semantics are those of nondeterministic choice between those expressions whose guards are true. In this case, no else-part (i.e. empty guard) is allowed.

[PO: if no else-part is present, at least one guard evaluates to **true**. If **opaque** was used: each guard can be evaluated; each expression can be evaluated if its guard is **true**. If **opaque** was not used: each guard can be evaluated if the preceding guards are **false**; each expression can be evaluated if its guard is **true** and the previous guards were **false**].

5.4.3 Choosing

A set, bag or sequence can be subjected to a filtering operation using the notation:

> *ChooseExpression*:
>   *ChoiceType Identifier* "**:**" *TypeExpr2* "**:-**" *Predicate*;
>   *ChoiceType Identifier* "**::**" *Expr4* "**:-**" *Predicate*;
>   *ThatOrAny Expression*.
>
> *ChoiceType*:
>   **that**;
>   **any**;
>   **those**.
>
> *ThatOrAny*:
>   **that** ;
>   **any**.

The meaning when the **any** choice type is used is: any value of the specified type or from the specified expression (which must be a set, bag or sequence) such that *Predicate* is true. There must be at least one such value. The meaning when **that** is used is similar, except that we are asserting that there is exactly one such value. The result in both cases is a value belonging to the *TypeExpr2* or the base type of which the *Expr4* is a collection. If the form without "*:- Predicate*" is used, it is equivalent to the full form with **true** substituted for *Predicate*.

The meaning when the **those** choice type is used is the set, bag or sequence (depending on whether *TypeExpr2* or *Expr4* is a class or set, a bag or a sequence respectively) comprising those values in *TypeExpr2* or *Expr4* for which *Predicate* is true.

5.4.4 Transforms

A transform takes values from one collection and maps them onto a another collection (possibly of a different type) on an element-by-element basis.

TransformExpression:
    for *Identifier* "**::**" *Expr4* **yield** *Expression*;
    for those *Identifier* "**::**" *Expr4* "**:-**" *Predicate* **yield** *Expression*.

The *Expr4* must in either case have a type which is a set, bag or sequence; the result is a set, bag or sequence (respectively) of elements of the type of the *Expression*. In the second form, only those elements of the *Expr4* which satisfy *Predicate* are chosen. If the *Expr4* is a sequence, the result is a sequence whose elements are in the same order as the elements in the *Expr4* from which they were generated.

Note that if the first form is used (i.e. without **those** keyword), then when the operand is a bag or sequence, the result has the same number of elements as the operand; however, if the operand is a set, the result may have fewer elements (because multiple elements in the operand may yield the same result value, and the result is condensed into a set).

5.4.5 Quantified expressions

A quantified expression has the form:

QuantifiedExpression:
    (**forall** | **exists**) *BoundVariableDeclarations* "**:-**" *Predicate*.

BoundVariableDeclarations:
    BoundVariableDecl *["**,**" *BoundVariableDecl*].

BoundVariableDecl:
    IdentifierList "**:**" *TypeExpr2*;
    IdentifierList "**::**" *Expr4*.

IdentifierList:
    Identifier *["**,**" *Identifier*].

The result type is Boolean. If the form containing "**::** " is used, the *Expr4* which follows "**::**" must yield a set, bag or sequence.

When the universal quantifier **forall** is used the meaning is "For all permitted values of the declared identifiers, *Predicate* is **true**". If there are no permitted values (because the type of one of the declared identifiers is an empty set or type), the expression yields **true**.

When the existential quantifier **exists** is used the meaning is "There exists a combination of permitted values of the declared identifiers such that *Predicate* is **true**".

The two types of quantifier are related in the following manner:

    (**forall** x:t **:-** p(x)) <==> ~(**exists** x:t **:-** ~p(x))

5.4.6 Cast expression

The cast expression is used to assert that some value whose static type is a union (expressed explicitly or using the keyword **from**) is known to be of a narrower type, and to cast it to that type.

>   *PrimableCastExpression*:
>   *PrimableExpr8* **is** *TypeExpression*.

>   *UnprimableCastExpression*:
>   *CompareExpr* **is** *TypeExpression*.

If a cast expression is used as an operand, it must be enclosed in brackets.

[SC] Type *TypeExpression* must be a type contained in the type of the *PrimableExpr8* or *CompareExpr*.

[PO] Each use of a cast expression *expr* **is** *type* gives rise to the verification condition *expr* **within** *type*.

5.4.7 Type widening expression

The type widening expression is used to treat a value of some class as a value of a union (expressed explicitly or using the keyword **from**) which includes that class. It may also be used to explicitly remove constraints from a type (e.g. to cast an object of type nat to **int**).

>   *PrimableCastExpression*:
>   *PrimabeExpr8* **as** *TypeExpression*.

>   *UnprimableCastExpression*:
>   *CompareExpr* **as** *TypeExpression*.

If a type widening expression is used as an operand, it must be enclosed in brackets.

[SC] Type *TypeExpression* must be a type containing the type of *CompareExpr*. That is, it must be a union containing the type, or a type with fewer constraints. If the types are identical a warning is generated.

5.4.8 Type enquiry expression

It is possible to enquire whether a value of a union type (including a union type generated using the **from** keyword) is a member of one of the types from which the union type is formed, or a member of a subset of the union (including a subset expressed using the **from** keyword). The syntax is:

>   *TypeEnquiry*:
>   *CompareExpr* ["~"] **within** *TypeExpression*.

The result has type **bool**.

If a type enquiry expression is used as an operand, it must be enclosed in brackets.

Note that the types of two expressions can be compared using the **like** operator (see section 5.3.5).

5.4.9 Subclass expression

The subclass selector is used to assert that a value whose type is a union belongs to a subclass of that union, or that a value known to be derived from some base class type is actually of a particular derived class, and in either case to yield a value of that subclass or derived class. Its main use is to form operands of the correct type for passing to functions and operators. It can also be used to assert that a particular constraint is satisfied by a value.

> *SubclassExpression*:
>   *CompareExpr* **is** *TypeExpression*.

If a subclass expression is used as an operand, it must be enclosed in brackets.

[SC] In this construct, either *Expression* has a union type and *TypeExpression* is a member of that type or a subset of a member or a union of some (but not all) members of that union, or *Expression* has type "**from** *Classname*" and *TypeExpression* is the name of a class derived from *Classname*. The compiler will report an error if this condition is not satisfied (i.e. it is statically impossible for *Expression* to have type *TypeExpression* at runtime). The compiler will report a warning if the type of *Expression* is statically the same as *TypeExpression*.

5.4.10 Subjunctive expression

The subjunctive expression yields the value which an object would have if a schema which modifies it were to be invoked. It does not actually modify the object concerned (a typical implementation might take a copy of the object, modify the copy and yield the modified copy as the result). The syntax is:

> *SubjunctiveExpression*:
>   *CompareExpr* **after** *Postcondition*.

[SC] *Postcondition* must be a postcondition which modifies the pronoun **it** and nothing else.

The subjunctive expression should only be used where the alternative means of specifying the value required are cumbersome. For example, if a variable *x* is a class with abstract variable members *a*, *b*, *c .. z* and we wish to express a value of the same type whose members all have the same values as the members of *x* except for *a*, we might write:

> x **after it**.a! = newa

instead of explicitly stating the values of all the members. Similarly, if *y* is an object of a class that includes a modifying member schema *normalize*, we could write:

> y **after it**!normalize

The postcondition following **after** is permitted only to modify the object denoted by the pronoun **it** (which is a copy of the expression to the left of **after**). It may not modify objects addressed through references, since such modifications might affect other objects.

The subjunctive expression provides the sole mechanism for specifying functions in terms of schemas. Subjunctive expressions used as operands must be enclosed in brackets.

The full syntax for postconditions is described in <u>section 6.8.2</u>.

5.4.11 Over expression

The over expression "*op* **over** s", where "s" is a non-empty set, sequence or bag, and "*op*" is a binary operator whose operand and return types are all equal to the type of the elements of "s", is defined in the case of a sequence as follows:

> ([#s = 1]: **that** s, []: (*op* **over** s.front) *op* s.last)

In the case of a set or bag the definition is:

> ([#s = 1]: **that** s, []: (**let** tmp ^= **any** s; (*op* **over** s.remove(tmp)) *op* tmp))

[SC] The operator must not have any precondition.

[SC] If "s" is a set or bag, the operator must have been declared associative and commutative (see <u>section 6.6.3</u>).

[PO: The collection "s" is non-empty, unless a left identity has been declared for the operator].

5.4.12 Heap expression

The heap expression creates a value on a named heap and yields a reference to that value.

> *HeapExpression*:
>   **ref** *Expression* **on** *Identifier*.

Here, *Identifier* must be the name of a heap. The type of the expression is a reference to the type of *Expression*.

5.4.13 Value expression

The value selector *UnprimableExpr8* "**.**" **value** returns the value of an object on a heap. The expression must be of type **ref** X **on** H, where X is some type, and H some heap, and returns the object of type X.

5.4.14 Converting between types

When an expression is used as an operand of a function or operator, the only type conversions that may be invoked implicitly are widening conversions from a class to a union which includes that class (which includes converting X to **from** X, X to **from** Y, or **from** X to **from** Y, where Y is an ancestor of X), and conversions between types which are equal once all constraints have been removed (even if this involves adding new constraints, for example **int** to nat). Other type conversions must be performed explicitly using constructors, subclass expressions, operators and functions.

5.4.15 Scope resolution

To specify the class in which a non-member function or enumeration value is to be found, the name of the function or value may be preceded by the name of the class, with the class name and function or value name separated by white space only (i.e. *class-name function-or-value-name*). Enumeration values are treated as non-members of the enumeration class, so must be referred to in this way. No scope resolution is required to refer to non-members declared in the current class or one of its ancestors.

*Note: older versions of Perfect Developer used the syntax **function-or-value-name** @ **class-name**. This syntax is still supported, but deprecated.*

5.4.16 "?" expression

This represents an expression whose value has deliberately not been specified yet. Typically, it may be used in skeletal source files that do not yet need to be compiled but need to be included in other *Perfect* source files.

5.5 Writable, Limited-writable and Non-writable expressions

Any expression belongs to one of three categories: writable, limited-writable and non-writable. The category of an expression is relevant within postconditions and implementations. A writable expression (sometimes called an *lvalue*) may be changed in any way that conforms to its type (for example, it may be re-assigned). A limited-writable expression may only be modified by calling a member schema of the type to which it belongs, and its actual run-time type can never be changed by such an operation. A non-writable expression cannot be written at all.

A writable expression is one of the following:

- An identifier that binds to an implementation variable declaration
- Within a function or operator postcondition, **result**
- Within an implementation of a member method, an identifier that binds to an internal data member
- An identifier that binds to a schema parameter declaration decorated with "!" and not declared **limited**
- A writable or limited-writable expression followed by ".*Identifier*", where the identifier binds to a variable member declaration provided that the expression occurs within the class declaration.
- A writable or limited-writable expression followed by ".*Identifier*" and a parameter list (if required), where the identifier binds to a selector member declaration or redeclaration whose result is not declared **limited**
- A writable or limited-writable expression followed by an expression within square brackets, provided that indexing has been declared as a selector (not an operator) for the class of the first expression and the selector result type not declared **limited**
- A subclass expression whose operand is writable

A limited-writable expression is one of the following:

- Within a constructor or modifying schema postcondition or implementation, **self** (the current object)
- An identifier that binds to a schema parameter declaration decorated with "!" and declared **limited**
- A type-widening expression whose operand is writable or limited-writable
- A subclass expression whose operand is limited-writable

- A writable expression which yields a reference, followed by "**.value**"
- A writable or limited-writable expression followed by "**.***Identifier*" and a parameter list (if required), where the identifier binds to a selector member declaration whose result is declared **limited**
- A writable or limited-writable expression followed by an expression within square brackets, provided that indexing has been declared as a selector (not an operator) for the class of the first expression and the selector result type is declared **limited**

5.6 Primed expressions

To "prime an expression" means to place a prime (single quotation mark) after it. Only writable and limited-writable expressions may be primed, and only in contexts where the expression has potentially a final value that differs from its initial value (e.g. postconditions, schema post-assertions, and implementations). In such contexts, a primed expression refers to the final value of that expression; an un-primed equivalent expression refers to the initial value of the expression.

In the context of a multithreaded environment, the term "initial value" is misleading and an unprimed expression is taken instead to refer to "the value the expression would have had if the current thread had not modified it".

6. Modules and Declarations

6.1 Declarations

There are several sorts of declaration in *Perfect*:

> *Declaration*:
> > *ConstantDeclaration*;
> > *VariableDeclarations*;
> > *HeapDeclaration*;
> > *FunctionDeclaration*;
> > *SelectorDeclaration*;
> > *OperatorDeclaration*;
> > *SchemaDeclaration*;
> > *PropertyDeclaration*;
> > *AxiomDeclaration*;
> > *ClassDeclaration*;
> > *TypeDeclaration*.

Class and type declarations were covered in <u>Chapter 4</u>. This chapter describes the remaining types of declaration.

6.2 Constant declaration

Constants are declared using the syntax:

> *ConstantDeclaration*:
> > [**ghost**] **const** *ConstDeclItem* *["**,**" *ConstDeclItem*].

> *ConstDeclItem*:
> > *Identifier* ["**:**" *TypeExpression*] "**^=**" *Expression* [*NvImplementation*].

The *TypeExpression* and preceding colon are optional if the *Expression* is a literal, constructor expression, Boolean operator expression, quantified expression, type-widening expression or subclass expression; otherwise it is mandatory. If a *TypeExpression* is given, the type of the *Expression* will be widened to match it, if necessary.

If the optional **ghost** keyword is present, no code will be generated for the constants, and they may only be used in 'ghost' contexts (i.e. preconditions, assertions, invariants, variants and implemented specifications). The optional implementation tells the system how to calculate the value of the constant. Implementations are covered in <u>chapter 8</u>.

6.3 Heap declarations

Heaps are declared as follows:

HeapDeclaration:
>   **heap** *Identifier* *["**,**" *Identifier*]*.

A heap may be used to hold values of any number of types.

A heap declaration is visible wherever a class declaration at the same place would be visible.

6.4 Variable declarations

Variables are declared using the following syntax:

>   *VariableDeclarations*:
>   **var** *DataDeclarationList*.

>   *DataDeclarationList*:
>   *DataDeclarations* *["**,**" *DataDeclarations*]*.

>   *DataDeclarations*:
>   *Identifier* *["**,**" *Identifier*]* "**:**" *PossConstrainedTypeExpression*;
>   *Identifier* "**:**" *AbbrevTypeExpression*;
>   **when** *Guard DataDeclarationList* *["**,**" *Guard DataDeclarationList*]* **end**.

>   *AbbrevTypeExpression*:
>   **those** *TypeExpression* "**:-**" *Predicate*.

The abbreviated form of type expression avoids the need to introduce a new bound variable name, since the name of the single variable being declared can fulfil this function.

The **when**...**end** form is only permitted within the **abstract** or **internal** section of an abstract class declaration, and allows the presence of one or more variables to be made conditional on the value of other member variables. The *Predicate* within each *Guard* must be a function of variable members only, and may not contain calls to member functions of the class being declared.

[TBD: we could require all the guards within a **when** clause to be mutually exclusive, then it might be sensible to allow an "else" part as well. This would avoid a proliferation of styles and might improve readability since conditional declarations which were unrelated would have to be in different **when** clauses. However, there are other cases when it might be reasonable to have overlapping guards, e.g. "**when** [x=c]: decls, [x=d]: decls, [x=c | x=d]: decls **end**".]

[Semantic note: declaring a variable **v** of type **T** within **when**...**end** is similar but not equivalent to declaring **v** unconditionally with type "**T** || **void**", adding a class invariant that "**v** = **null**" whenever the corresponding guard is not satisfied, and replacing all non-assigning occurrences of **v** by "(**v is T**)". Declaring the data as guarded implies a stronger condition, since the data must be initialised whenever a postcondition modifies the guard so that it becomes true, even on intermediate instances of **self**, which we allow *not* to satisfy the class invariant. It is also legal to guard data which is of a type including **void**, whereas the previous construction makes no sense in this case.]

6.5 Function declarations

Like traditional programming languages, *Perfect* allows functions to be defined; however, *Perfect* functions are pure in that they have no side-effects. The *Perfect* notion of a schema corresponds to a function with side-effects in other languages.

A non-opaque function must either be a member of a class, or take at least one parameter. A constant declaration (see <u>section 6.2</u>) should be used instead of declaring a deterministic function with no parameters.

6.5.1 Syntax of function declarations

FunctionDeclaration:
　　FunctionHeader [*ExceptionSpecification*] [*RequirePart*] [**pre** *PredicateList*] [**decrease** *RecursionVariant*] *FunctionBody* [*PostAssertion*].

FunctionHeader:
　　[**opaque** | **ghost**] **function** *Identifier* ["(" *FunctionParameterList* ")"] *FunctionType*.

FunctionBody:
　　"^=" *PossMultipleExpression* [*Implementation*];
　　satisfy *PredicateList* [*Implementation*].

PostAssertion:
　　assert "..." ["," *PredicateList*] [*Proof*];
　　Assertion.

FunctionParameterList:
　　FunctionParameters [*Separator* **repeated** *FunctionParameters*];
　　repeated *FunctionParameters*.

FunctionParameters:
　　FunctionParams *[*Separator* FunctionParams*].

FunctionParams:
　　Identifier *[*Separator* Identifier*] ":" *ParameterType*.

ParameterType:
　　class *Identifier*;
　　TypeExpression.

FunctionType:
　　":" *TypeExpression*;
　　FunctionTypeList *["," *FunctionTypeList*].

FunctionTypeList:
　　Identifier *["," *Identifier*] ":" *TypeExpression*.

> *PredicateList*:
>   *Predicate* *[**","** Predicate*].

> *RecursionVariant*:
>   "**...**" [**","** *Variant*];
>   *Variant*.

> *Variant*:
>   *Expression* *[**","** Expression*].

> *ExceptionSpecification*:
>   **throw** *TypeExpression*.

The parameter list comprises declarations of formal parameters, with parameters being separated by any of the four separators ("**,**" and the three forms of arrow). The keyword **repeated** indicates that the remaining formal parameters may be matched by one or more matching sequences of actual parameters. If there is more than one repeated formal parameter, the separators between the repeated formal parameters may not include comma (this rule is imposed for clarity since comma is used to separate multiple sequences of actual parameters matching the repeated parameters). Within the function declaration, the type of each repeated formal parameter is "seq of X" where X is the declared type of the parameter.

The function result type may be declared as either a simple type or a list of simple identifier declarations (similar to the declarations which may follow the keyword **var**). The second form is used for functions which return more than one value; the effect is to declare an anonymous class with the declared identifiers as data members. If this form is used for the declaration of a member function, the identifiers declared must be distinct from all the class member names (including the names of inherited members).

The precondition, if present, comprises the keyword **pre** followed by a comma-separated list of predicates. The predicates describe conditions (normally functions of the parameters and/or current class members) which must all be true whenever the function is evaluated (i.e. all the predicates in the list are combined with the logical "&amp" operator, and the resulting expression must yield true). If no precondition part is given, **true** is assumed (i.e. the function will succeed for all values of its input parameters). The precondition must be well-defined for all values of the parameters (i.e. the precondition for evaluating the precondition is **true**).

A variant part is only needed by recursive functions and serves to guarantee termination. It comprises the keyword **decrease** followed by a list of expressions, each of which is of type **int**, **bool**, **char** or an enumeration. If just one expression is given, this expression is guaranteed to decrease on every recursion but never to become negative. If more than one variant expression is given, it is required that those expressions having type **int** are non-negative and that on each recursion either the first expression decreases, or the first expression remains constant and the variant consisting of the remaining expressions decreases according to the same rule. A boolean variant expression is considered to decrease if its values changes from **true** to **false**. The variant must be well-defined and its integral components non-negative whenever the function precondition is satisfied.

The form of variant beginning "**...**" means that the subsequent variant terms should be appended to an inherited variant. This form may only be used when overriding an inherited method with a variant, and in this case it is compulsory.

The function body defines the result of the function. If the function result is defined by the "^= " symbol and a list of expressions, these expressions must be well-defined whenever the function precondition is satisfied, and the number and types of the expressions must match the declared return types. Alternatively, the function result may be defined using the keyword **satisfy** followed by a predicate list in which each predicate expresses a condition on the predefined identifier **result**.

The optional post-assertion specifies any additional properties of the function result that are expected to hold. These properties should be provable from the function definition. Within the post-assertion, the predefined identifier **result** refers to the result of the function. Each expression in the post-assertion must refer to **result**. The "**...,**" form of post-assertion may be used when the function overrides an inherited function that already has a post-assertion; in this case the total post-assertion is the post-assertion of the overridden function followed by the new post-assertion.

The optional implementation part tells the system on how best to implement the function. Implementations are covered in Chapter 8.

The exception specification declares that any exceptions thrown by the function are within the declared type. This type may be a union type, allowing exceptions conforming to any member type of the union to be thrown. If no exception specification is given, then the function does not throw any exceptions.

Within all expressions in a member function declaration, members of the current class may be used unqualified (they are implicitly prefixed by "**self**").

If the **opaque** keyword is given, the function is nondeterministic (i.e. two different calls to the same function with correspondingly equal parameters may yield results which are not equal to each other); otherwise the result must be precisely defined. Note that if the result includes a reference to a newly-constructed expression on a heap, this guarantees that the function is nondeterministic, therefore the **opaque** keyword must be used in this case.

The **ghost** keyword indicates that the function is used only in preconditions, class invariants, assertions, variants, and in the result expressions and postconditions of a method or constructor for which an implementation is given. It is therefore not necessary for the compiler to generate code for the function (unless run-time checks are being generated).

[SC] The names of the parameters, the names of the returned values (if the multiple-returned-value form is used) and (for a declaration of a member or nonmember function within a class) the names of accessible members of the current class must all be distinct, in order to avoid ambiguity.

[PO: the precondition does not directly or indirectly refer to the function recursively; the precondition can always be evaluated (i.e. its precondition is true); the variant and the function value can be evaluated whenever the precondition is true; if the function result specification is directly or indirectly recursive, the variant decreases on each recursion].

6.5.2 Polymorphic function declarations

In parameter declarations, instead of following the colon by an actual type, it may be followed by the construct "**class** *Identifier*" to denote a class parameter. The same identifier may subsequently be used within the parameter list, result type or function body as a type name (without the **class** prefix).

This mechanism allows the construction of a polymorphic function (one which supports many operand types). Any member methods of the unspecified class that are used by the function specification or body must be declared in the optional **require** part (see section 7.4 for the grammar for *RequirePart*). The presence of these members will be checked when the function is invoked.

It is permissible to have multiple class parameters in a parameter list.

Polymorphic type names cannot occur within unions or after **from**. This restriction is imposed to guarantee that instantiation of polymorphic type names according to the actual parameter types is always unambiguous.

When the same polymorphic type name occurs more than once in a formal parameter list, the types in the corresponding actual parameter list of the call must be equivalent. This requirement is captured by the rules (n1-n4) in the formal definition of nesting. For example, if a function is declared with the signature "f(e: **class** X, s: **seq of** X)" then the call "f(p, q)" where "p" has type "a || **int**" and "q" has type "**seq of int**" is invalid: the types of the actual parameters can not be nested in the types of the formal parameters simultaneously.

Whenever a polymorphic function is declared, an additional condition is implicitly appended to the precondition; namely, that all methods declared in the **require** part must be declared for the actual operand types at the point of call, and their preconditions must be satisfied whenever the function precondition is satisfied.

6.5.3 Function usage

A function is invoked in the conventional way using its name followed by a list of actual parameters in round brackets. Where there are no operands, no brackets are used.

Where a function returns more than one result, the function call may either be followed by "**.**" and the name of a member of the result (in order to select just one part of the result), or the entire result object may be captured in a **let** declaration.

[PO: the function precondition (extended to cover operator definitions in the case of polymorphic functions) is satisfied at the point of invocation.]

6.6 Operator declarations

Perfect also allows many of its own operator symbols to be redefined for user-defined classes. Operators are declared in a similar way as functions except that "**function** *identifier*" is replaced by "**operator** *symbol*". Operators may not have multiple result values (except by explicitly declaring a class having the required members) and must be class members. Polymorphic operators may be declared.

The unary operator symbols that may be defined are: **# + - < >**

The binary operator symbols that may be defined are: **^ ** ++ -- # + - * / % << <<= in [] ..**

The binary operator symbols that may be re-implemented (see section 7.1.5) but not redefined are: **= &lt <=**

6.6.1 Syntax of operator declarations

OperatorDeclaration:
>   *OperatorHeader* [*ExceptionSpecification*] [*RequirePart*] *[*OperatorProperty*] [**pre** *PredicateList*] [**decrease** *RecursionVariant*] *OperatorBody* [*PostAssertion*].

OperatorHeader:
>   [**opaque** | **ghost**] **operator** *UnaryRedefinableOp* "**:**" *TypeExpression*;
>   [**opaque** | **ghost**] **operator** *BinaryRedefinableOp OperatorParameter* "**:**" *TypeExpression*;
>   [**opaque** | **ghost**] **operator** *OperatorParameter RevRedefinableOp* "**:**" *TypeExpression*.

OperatorBody:
>   "**^=**" *Expression* [*Implementation*];
>   **satisfy** *PredicateList* [*Implementation*].

OperatorParameter:
>   "**(**" *Identifier* "**:**" *ParameterType* "**)**".

UnaryRedefinableOp:
>   "<"; ">";
>   " +"; "**-**"; "*"; "/"; "%"; "#"
>   "^".

BinaryRedefinableOp:
>   "**[**" "**]**";
>   *RevRedefinableOp*.

RevRedefinableOp:
>   "<"; "<="; "<< "; "<<=";
>   ">";
>   "**in**";
>   " +"; "-"; "++"; "--";
>   "*"; "/"; "**"; "%"; "%%"; "#" "##";
>   "**..**"; "^".

OperatorProperty:
>   **associative**;
>   **commutative**;
>   **idempotent**;
>   **identity** *Expression*.

All operators must be declared as members, and there must be the correct number of parameters (zero or one) for the operator symbol being defined (i.e. one less than the arity of the operator). If a parameter is declared, it may be declared either before the operator keyword (in which case the current object will take the place of the right operand), or after the operator symbol (in which case the current object will take the place of the left operand).

6.6.2 Comparison operator declarations

When declaring comparison operators, additional properties normally associated with the operator symbol being used must be satisfied, as follows:

- The result type must be **bool**.
- Operators which are already defined for the types concerned may not be declared. Note that since the equality operator is defined for all types, it may not be redefined (although it may be declared as **ghost** or non-**ghost** and its implementation may be redefined within a class).
- Operators consisting of "~" followed by another operator symbol may not be declared (they are generated automatically from the corresponding operator symbol).
- The binary operators ">" and ">=" may not be declared; they are automatically generated from the operators "<" and "<=" respectively by reversing the operands. Similarly, ">>" and ">>=" are automatically generated from "<<" and "<<=".
- The binary operators "<" and "<=" may not be declared, though may be reimplemented. They are generated automatically from "~~" ("<=" is only generated when "~~" is declared as defining a **total** ordering on the type).

When declaring the comparison operator "~~", a special form of operator declaration is used where the parameter type and return type are not specified (they are automatically the type of the current class and **rank** respectively), there may not be any preconditions, and the specification must satisfy the properties for "~~" detailed in section 7.1.7.

6.6.3 Declaring operator properties

Where a binary operator is declared with no precondition, the operator may be declared any or all of associative, commutative and idempotent; in addition, a left identity expression can be declared (i.e. an expression *E* such that *E op X = X* for all expressions *X*). Verification conditions will be generated to verify that the stated properties hold.

6.7 Selector declarations

Selectors may only be declared as members of abstract classes (see Chapter 7) but selector declarations will be described here due to their similarity with function and operator declarations.

A selector declaration has similar syntax to a function declaration, except that the **function** keyword is replaced by **selector**, the selector name may either be an identifier or the "**[]**" symbol, the result type may not be a declaration list, and the result must be defined by "^=" and a writable or limited writable expression. The grammar is:

6.7.1 Syntax of selector declarations

> *SelectorDeclaration*:
>   *SelectorHeader* [*ExceptionSpecification*] [*RequirePart*] [**pre** *PredicateList*] [**decrease**
> *RecursionVariant*] "^=" *Expression* [*Implementation*] [*PostAssertion*].

SelectorHeader:
> [**opaque** | **ghost**] **selector** *SelectorName* ["(" *FunctionParameterList* ")"] "**:**" [**limited**] *TypeExpression*.

SelectorName:
> *Identifier*;
> "**[**" "**]**".

The purpose of a selector is to identify a sub-part of the current object. Unlike a function, the result of a selector can be modified.

If the selector result type is declared **limited**, the access provided does not allow clients to reassign the result (but does allow member schemas and further selectors to be invoked on the result) and the result expression that follows "**^=**" must be writable or limited writable.

If the selector is not declared **limited**, clients may use the selector to re-assign the returned sub-object (and in doing so, to change its actual type, if its declared type is a union). The result expression that follows "**^=**" must be writable.

A classic example of a selector is the indexing operator "**[]**" on sequences.

6.8 Schema declarations

A schema is a description of a change of state (with optional implementation detail). Schemas correspond to procedures in conventional programming languages. Every schema has a defined postcondition; schemas may also have parameters, preconditions, variants and implementations. The syntax is:

6.8.1 Syntax of schema declarations

SchemaDeclaration:
> *SchemaHeader* [*ExceptionSpecification*] [*RequirePart*] [**pre** *PredicateList*] [**decrease** *RecursionVariant*] **post** *Postcondition* [*Implementation*] [*PostAssertion*].

SchemaHeader:
> [**opaque** | **ghost**] **schema** ["**!**"] *Identifier* ["(" *SchemaParameterList* ")"].

SchemaParameterList:
> *SchemaParameters* [*Separator* **repeated** *SchemaParameters*];
> **repeated** *SchemaParameters*.

SchemaParameters:
> *SchemaParams* *[*Separator* *SchemaParams*].

SchemaParams:
> *SchemaParamIdentifier* *[*Separator* *SchemaParamIdentifier*]
>   "**:**" [**limited** | **out**] *ParameterType*.

    SchemaParamIdentifier:
        Identifier;
        Identifier "**!**".

The "**!**" symbol before the schema name may only appear in class member schemas; it indicates that the schema modifies the current object. The parameter list is similar to a function parameter list except that the names of the parameters may each be followed by an exclamation mark, and the parameter types may be preceded by the keywords **limited** and **out**. An exclamation mark indicates that the actual parameter may be modified by the schema. Parameters declared after **repeated** may not be decorated.

The optional keywords **limited** and **out** before the parameter type may only be used when all the parameters to which they apply are decorated with "**!**". The keyword **out** indicates that the parameter's initial value is of no interest to the schema. The keyword **limited** indicates that the parameter is to be limited writable (instead of fully writable), i.e. the schema is not permitted to reassign the parameter (and may therefore not change its actual type, if it is a union).

Polymorphic parameters are permitted in the same way as for functions and lead to the same implicit additional precondition.

The precondition and variant parts are exactly as already described for functions.

The postcondition (**post** part) describes one or more conditions that must be satisfied after execution of the schema. As with the precondition, a list of postconditions may be given; all must be satisfied. Postconditions are described in more detail in the <u>following section</u>.

Within the schema declaration, parameters which were listed in the parameter list without the keyword **out** may be referred to unprimed (but need not be referred to at all). Parameters which were listed with an exclamation mark may appear primed within the postcondition list. Parameters which were declared as **out** must be assigned by the postcondition.

A schema which is not a class member may only modify parameters which appear followed by "**!**". A schema which is a class member may be declared with the "**!**" symbol before the schema name, in which case it may modify members of the current object as well.

A schema post-assertion specifies additional conditions that are expected to hold when the schema completes. These conditions should be provable from the postcondition. Each expression in the post-assertion must refer to at least one primed entity (i.e. a primed instance of a modified parameter, or a primed member variable or primed **self** if it is a member schema that modifies the current object). As with function post-assertions, a schema post-assertion may start with "**... ,**" to indicate that this post-assertion adds to rather then overrrides the post-assertion of the overridden inherited schema.

[SC] If the **opaque** keyword is used, the schema is nondeterministic (i.e. the postcondition does not uniquely determine the final state, in consequence multiple calls with the same parameters may give rise to different states); otherwise it is deterministic.

[SC] The **nonmember** keyword indicates that the schema is not a member of the class and does not have the "current object" parameter. It is only valid within in a class declaration.

[SC] A variant is specified if the postcondition refers to the schema under definition, i.e. if the schema specification is recursive.

[PO: the precondition can always be evaluated (i.e. its precondition is true); the postcondition can be evaluated whenever the precondition is true; if the postcondition is directly or indirectly recursive, the variant decreases on each recursion; the variant can never be negative.]

6.8.2 Postconditions

The full grammar for postconditions is as follows:

>*Postcondition*:
>>**change** *ExpressionList* **satisfy** *PredicateList*;
>>*PostconditionList*;
>>"**?**".

>*PostconditionList*:
>>*PostconditionElement* *["**,**" *PostconditionElement*].

>*PostconditionElement*:
>>**forall** *BoundVariableDeclarations* "**:-**" *PostconditionElement*;
>>*Postcondition0*.

>*Postcondition0*:
>>*Postcondition0* **then** *Postcondition1*;
>>*Postcondition1*.

>*Postcondition1*:
>>*Postcondition1* "**&**" *Postcondition2*;
>>*Postcondition2*.

>*Postcondition2*:
>>*PrimableExpr8* "**!**" "=" *Expression*;
>>*PrimableExpr8* "**!**" *AssignableOp Expression*;
>>"**(**" *[*LetDeclarationOrVariableDeclarationsOrAssertion*] *PostconditionOrChoices* ")";
>>*SchemaCall*;
>>**pass**.

>*AssignableOp*:
>>*BooleanImplicationOperator*;
>>"**&**"; "**|**";
>>*AddingOperator*;
>>*MultiplyingOperator*;
>>"**^**"; "**..**".

>*LetDeclarationOrVariableDeclarationsOrAssertion*:
>>*LetDeclarationOrAssertion*;
>>*VariableDeclarations* "**;**".

PostconditionOrChoices:
    PostconditionList;
    Guard Postcondition *["**,**" *Guard Postcondition*] ["**,**" *LastChoice*];
    opaque *Guard Postcondition* "**,**" *Guard Postcondition* *["**,**" *Guard Postcondition*].

LastChoice:
    EmptyGuard Postcondition;
    "**[**" "**]**".

SchemaCall:
    [*PrimableExpr8*] "**!**" *IdentifierOrSuper* [*SchemaActualParameterList*];
    PrimableExpr8 "**.**" *IdentifierOrSuper SchemaActualParameterList*;
    IdentifierOrSuper SchemaActualParameterList.

SchemaActualParameterList:
    "**(**" *SchemaActualParameter* *[*Separator SchemaActualParameter*] "**)**".

SchemaActualParameter:
    PrimableExpr8 "**!**";
    Expression.

6.8.2.1 The "change...satisfy" form

In this form of postcondition, a list of expressions to be changed and a list of predicates to be satisfied is given. Each expression in the list must be a primable expression (but should not be primed in the list). The code generator will attempt to modify the specified variables to achieve the desired conditions.

Although this is the most general form of postcondition, it has two disadvantages; it is verbose (because subexpressions whose values change may have to be written twice - once in the expression list and once in the predicate) - and the code generator may not be able satisfy the predicate, in which case an implementation will have to be provided.

6.8.2.2 The "then" form

This form allows two postconditions to be joined in a strictly sequential manner. In each of the two component postconditions, a primed subexpression refers to the final value after execution of that postcondition (which, for the first postcondition, is not necessarily the final value after execution of the complete compound postcondition), and an unprimed subexpression refers to the value before execution of that postcondition (which, for the second postcondition, is not necessarily the same as the value before execution of the complete compound postcondition).

A **then**-postcondition may not appear in a postcondition list containing more than one postcondition, nor may it be combined with another postcondition using "&".

6.8.2.3 The "&" form

This form allows two postconditions to be joined in a parallel manner. This form is only valid if the sets of subexpressions modified in each of the component postconditions do not overlap. So if the component

postconditions can be represented by "**change** *exprlist1* **satisfy** *cond1*" and "**change** *exprlist2* **satisfy** *cond1*" respectively, the combination is equivalent to "**change** *exprlist1*, *exprlist2* **satisfy** *cond1* & *cond2*".

The individual postconditions conjoined by "&" may not be or contain the **then**-form of postcondition.

6.8.2.4 The assignment form

The "*x!* = *y*" form of postcondition assigns the value of *y* to *x*. It is equivalent to the postcondition "**change** *x* **satisfy** *x'* = *y*".

The form "*x!* *AssignableOp* *y*" is shorthand for "*x!* = *x AssignableOp y*". The operator must be a binary operator accepting parameters of the types of *x* and *y* and returning a value of the type of *x* (exactly as if the full version had been used). Using this form also serves as a hint to the code generator that if possible the condition should be satisfied without copying *x*.

6.8.2.5 The bracketed form

The bracketed form parallels the syntax and semantics of bracketed expressions: **let** declarations can be used to define temporary values (whose names are in scope until the closing bracket), assertions may be introduced, and in schema postconditions (but not in postconditions attached to constructors or **after** expressions) new temporary variables may be declared. Finally, either a single postcondition, a comma-separated list of postconditions, or a guarded list of postconditions may be provided.

If a guarded list is provided, at least one guard must be satisfied (unless an empty guard is provided); the guards are evaluated in the given order, and the postconditions of the first guard found to be **true** will be selected for execution, defaulting to the postcondition with the empty guard if no guard is satisfied. If the empty guard is of the form with no postcondition then no action is taken if no guard is satisfied. There is no restriction on the relationships between the sets of subexpressions modified by the postconditions.

If the guarded list is preceded by **opaque**, then the semantics is that of nondeterministic choice between those postconditions whose guards are true. No empty guard is permitted in this case.

6.8.2.6 Schema calls

The form "*SchemaCall*" indicates that the designated schema should be invoked with the parameters supplied.

The name of the schema to be invoked is followed by a parameter list in brackets (see the grammar for *Expression*). Each actual parameter in the brackets must be followed an exclamation mark if the corresponding formal parameter was in the declaration (this makes it possible to tell at the point of schema invocation which parameters are modified, but it is not possible to tell without reference to the original declaration if a modified parameter is read or not). When invoking a schema which has no parameters, the brackets which would have enclosed the actual parameters are omitted.

When invoking a schema which is a class member, the schema member name must follow an expression yielding an object of the class concerned and be separated from that expression by an exclamation mark if the schema modifies the object, or a period otherwise (therefore it is possible to tell at the point of schema invocation whether the current object is modified). If exclamation mark is used to separate the expression from the member name, the expression must be writable or limited writable.

Within the declaration of the class concerned, a member schema name may appear without a preceding expression, in which case the current object is understood, but if the schema modifies the object, a leading exclamation mark is required and the current object must be writable or limited writable.

In the case of a formal parameter which is decorated by "**!**", the corresponding actual parameter must be writable (limited writable will suffice if the formal parameter's type is declared **limited**).

Where the current class declares a schema that overrides an inherited schema, the overridden schema can be accessed by prefixing its name with the keyword **super**. If the circumstances require the schema name to be preceded by an exclamation mark or a period, then the exclamation mark or period must be placed just before the **super** keyword.

For the purposes of applying the rules concerning the combining of postconditions using "&", a schema call is assumed to modify all parts of any subexpressions that appear followed by "**!**" in its parameter list; and if the schema name is preceded by "**!**", the entire expression preceding "**!**" if there is one, otherwise the entire current object. This means that schema calls that modify an object can only be combined with other postconditions that modify the same object using "**then**".

[PO: The schema precondition is satisfied.]

6.8.2.7 The "forall" form

The form "**forall** *BoundVariableDeclaration* **:-** *Postcondition*" instantiates and satisfies *Postcondition* for every value of the bound variable in the bound. The instantiations are considered to be executed in parallel. The subexpressions changed by each instantiation must be distinct.

[PO: For any value of the bound variable in the bound, the postcondition is well defined. For distinct values of the bound variable the postconditions produced are independent. If the bound is a sequence or bag, the elements are distinct.]

[TBD: The uniqueness condition above is not necessary - a weaker sufficient condition is that for any element appearing more than once in the collection, the postcondition does nothing. However, in this case the postcondition can be rewritten as '**forall** x::**those** y:: ...', so we use the simpler, stronger condition. A shorthand for writing this form of postcondition may be added at a later date.]

6.8.2.8 The "pass" form

The postcondition **pass** can be trivially satisfied by doing nothing.

6.8.2.9 The "?" form

This form indicates that the postcondition has deliberately not been specified yet. Typically, it may be used in skeletal source files that do not yet need to be compiled but need to be included in other *Perfect* source files.

[TBD: whether to insist that postconditions are deterministic unless they are specifically written in a nondeterministic manner using the **any** keyword.]

6.9 Property declarations

Properties come in two forms: with and without postconditions. A property *without* a postcondition is essentially an assertion with optional parameters and, in the case of a member property, an implied current object. A property *with* a postcondition describes conditions that are expected to hold after the postcondition is satisfied. Typically, the postcondition will comprise one or more schema calls.

Properties serve to verify that the system will satisfy its requirements, and to give hints to the verifier for proving verification conditions (including other properties).

6.9.1 Syntax of property declarations

Properties are declared using the syntax:

> *PropertyDeclaration*:
> > [**nonmember**] **property** [*Identifier*] ["(" *FunctionParameterList* ")"] [**pre** *PredicateList*]
> *Assertion*;
> > [**nonmember** | "**!**"] **property** [*Identifier*] ["(" *SchemaParameterList* ")"] [**pre**
> *PredicateList*] **post** *Postcondition PostAssertion*.

A property declaration *without* a postcondition is like a ghost function with no result, such that if it is called in a state satisfying the precondition and all applicable class invariants, all the expressions in the assertion should yield **true**. Mathematically, a property declaration can be turned into a traditional theorem by universally quantifying over each parameter (together with **self**, if it is a member property), and putting an implication operator between the precondition and the assertion.

A property declaration *with* a postcondition is like a ghost schema, although it cannot be called (even though it may be given a name), and it is not required to honour history invariants. Just as in the case of a real schema with a postassertion, if it is called in a state satisfying the precondition, then the postassertion should be **true**. If a member property declaration is prefixed with "**!**" then the postconditon may modify the current object; otherwise it may not.

The proof list is a hint to the theorem prover, needed if the property is valid but the prover is unable to prove it unaided.

Polymorphic properties are permitted. Property declarations may be overloaded in the same way as function declarations.

Member properties are inherited by derived classes, however they may not be deferred or redefined.

[SC] The **nonmember** keyword indicates that the property is not a member of the class and does not have the "current object" parameter. It is only valid within in a class declaration.

[SC] Neither the precondition nor the assertion list may directly or indirectly refer to the property.]

[PO: the precondition can always be evaluated; the assertion is satisfied whenever the precondition is satisfied and all the parameters and **self** (if applicable) satisfy the class invariants for their types.]

6.10 Axiom declaration

An axiom is similar to a property except that it asserts a truth which it is not possible for the theorem prover to prove but which is nevertheless to be assumed true. An axiom declaration has the syntax:

> *AxiomDeclaration*:
>> [**nonmember**] **axiom** [*Identifier*] ["(" *FunctionParameterList* ")"] [**pre** *PredicateList*]
>> **assert** *PredicateList*.

[SC] The **nonmember** keyword indicates that the axiom is not a member of the class and does not have the "current object" parameter. It is only valid within in a class declaration.

[PO: The precondition can always be evaluated; the predicate list which follows **assert** can be evaluated whenever the precondition is satisfied.]

6.11 Type compatibility of parameters and results

This section makes use of the relations "same" and "nested" defined in section 4.15. Type constraints take no part in determining type compatibility but do give rise to additional proof obligations. The exception to this is where a constrained type is used to instantiate a templated type, in which case the constraints must be identical for the types to match.

6.11.1 Type compatibility of undecorated parameters

An expression used as an actual parameter is type-compatible with a corresponding undecorated formal parameter if the expression type is nested within the formal parameter type.

6.11.2 Type compatibility of parameters decorated with "!"

An expression used as an actual parameter is type-compatible with a corresponding formal parameter decorated with "**!**" if the expression type is the same as the formal parameter type, or the expression type is nested within the formal parameter type and the formal parameter type is declared **limited**..

6.11.3 Type compatibility of repeated parameter groups

Where the formal parameter list includes the keyword **repeated** then the formal parameters that follow may be matched by one or more comma-separated groups of actual parameters, each of which must match all of those formal parameters and the intervening separators. For example, the formal parameter list:

> (a: **bool**, **repeated** b: **int** -> **string**)

could be matched by either of:

> (**true**, 1 -> "one")
> (**false**, 1 -> "one", 2 -> "two")

but not by either of:

> (**true**)
> (**false**, 1 -> "one", 2)

(the first violates the requirement to match the repeated group at least once, the second violates the requirement to match the entire repeated group a whole number of times).

6.11.4 Type compatibility of result values

A result expression in a function or operator declaration, or in a selector declaration with a return type declared **limited**, is type-compatible with the declared result type if the expression type is nested within the declared result type.

A result expression in a selector declaration with a return type not declared **limited** is type-compatible with the declared result type if the expression type is the same as the declared result type.

6.12 Function, Operator, Selector and Schema Overloading

Overloading the same name or operator symbol is permissible provided it is not possible to construct a call that could match more than one declaration, regardless of whether such a call actually occurs in the program. "Could" means that we assume all undefined type relations are actually true (see <u>section 4.15.2</u> for the definition of when type relations are undefined). For example, functions identical except for operands of types "**seq of** nat" and "**seq of int**" would not be permitted, since an operand of either type has an undefined match with the other.

Matching a call to a declaration considers only the number of parameters, the separators and the type compatibility rules; no account is taken of the decoration and writability of actual parameters or any constraints in the types of formal parameter.

6.13 Modules

A *Perfect* module comprises an import list followed by a sequence of declarations:

>   *Module*:
>   *[*ImportList*] *DeclarationList* ["**;**"].

>   *ImportList*:
>   **import** *NonEmptyStringLiteral* *["**,**" *NonEmptyStringLiteral*] "**;**".

>   *DeclarationList*:
>   *Declaration* *["**;**" *Declaration*].

The import list identifies other *Perfect* modules containing declarations referred to in the current module. Each *NonEmptyStringLiteral* names a file containing such a module.

[TBD: define precise rules for the search path for imported modules],

The declaration list for a module cannot include variable declarations (i.e. there are no global variables). The purpose of this restriction is to ensure that functions cannot depend on anything other than their parameters and the current object, and that schemas cannot modify anything not flagged as modifiable in their signatures.

7. Abstract Classes

7.1 Abstract class declaration

7.1.1 Syntax

ClassDeclaration
  [**deferred** | **final**] **class** *Identifier* [**of** *FormalTemplateParameters*] [*RequirePart*] "^="
[**inherits** *TypeExpr5* | **storable**] *ClassBody* **end**.

ClassBody:
  AbstractPart [*InternalPart*] [*ConfinedPart*] [*InterfacePart*];
  ConfinedPart [*InterfacePart*];
  InterfacePart.

AbstractPart:
  abstract [*AbstractMember* *["**;**" *AbstractMember*] ["**;**"]].

InternalPart:
  internal [*InternalMember* *["**;**" *InternalMember*] ["**;**"]].

ConfinedPart:
  confined [*ConfinedMember* *["**;**" *ConfinedMember*] ["**;**"]].

InterfacePart:
  interface [*InterfaceMember* *["**;**" *InterfaceMember*] ["**;**"]].

AbstractMember:
  MemberDeclaration;
  ClassInvariant;
  HistoryInvariant.

InternalMember:
  MemberDeclaration;
  InternalRedeclaration ;
  InternalReimplementation;
  ClassInvariant.

ConfinedMember:
  [**define** | **redefine**] *MemberFunctionEtcDeclaration* ;
  DeferredDeclaration;
  AbsurdDeclaration;
  ConfinedOrInterfaceRedeclaration.

InterfaceMember:
  ConfinedMember;
  [**ghost**] **operator** "=" "(" *Identifier* ")";

[**total**] **operator** "~~" "(" *Identifier* ")" [**decrease** *RecursionVariant*] *OperatorBody*.

MemberDeclaration:
    ConstantDeclaration;
    ConstructorDeclaration ;
    VariableDeclarations ;
    HeapDeclaration ;
    MemberFunctionEtcDeclaration;
    PropertyDeclaration;
    AxiomDeclaration;
    ClassDeclaration;
    TypeDeclaration.

ClassInvariant:
    invariant *PredicateList*.

HistoryInvariant:
    "**!**" **invariant** *PredicateList* [**exempt** IdentifierList].

InternalRedeclaration:
    function *Identifier* "**^=**" *Expression* .

InternalReimplementation:
    function *Identifier* ["(" *FunctionParameterList* ")"] *Implementation*;
    operator *RedefinableOp* [*OperatorParameter*] *Implementation*;
    operator *OperatorParameter RevRedefinableOp Implementation*;
    operator "=" "(" *Identifier* ")" *Implementation*;
    selector *SelectorName* ["(" *FunctionParameterList* ")"] *Implementation*;
    schema ["**!**"] *Identifier* ["(" *SchemaParameterList* ")"] *Implementation*;
    schema "**!**" *RedefinableOp OperatorParameter Implementation* ;
    build "**{**" [*ConstructorParameterList*] "**}**" *Implementation*.

ConfinedOrInterfaceRedeclaration:
    function *IdentifierList*;
    selector *IdentifierList*.

MemberFunctionEtcDeclaration:
    [**final** | **early**] *FunctionEtcDeclaration*;
    nonmember *FunctionDeclaration*;
    nonmember *SchemaDeclaration*.

FunctionEtcDeclaration:
    FunctionDeclaration;
    SelectorDeclaration ;
    OperatorDeclaration ;
    SchemaDeclaration .

DeferredDeclaration:
> **deferred** *FunctionEtcHeader* [**pre** *PredicateList*] [**decrease** *Variant*] [*Assertion*].

AbsurdDeclaration:
> **absurd** *FunctionEtcHeader*.

FunctionEtcHeader:
> *FunctionHeader*;
> *SelectorHeader*;
> *OperatorHeader*;
> *SchemaHeader*.

7.1.2 Class specification

The class specification comprises the optional inherit list, abstract, confined and interface member declarations. The internal declarations are implementation detail.

Whenever a function, operator, schema, property or axiom class member is declared, there is an implied "current object" parameter. The definition of the member may refer to unqualified members of the current object, and may also refer to the entire current object using the **self** keyword. However, if the **nonmember** modifier is used, the definition is not declaring a member and there is no "current object". Properties and axioms involving more than one value of the class type should normally be declared as nonmembers for clarity.

7.1.3 Inherits part

The optional **inherits** part specifies a class or class template instantiation whose attributes are to be inherited. It is forbidden to inherit from a **final** class or instantiation of a **final** class template.

7.1.4 Abstract members

The abstract member declarations and invariants describe the model of the class as seen by its clients. The list of declarations is prefixed by the keyword **abstract**. The list may include constant, variable, function, selector, operator, schema, property, axiom, constructor and class declarations (function and schema declarations are not common in this context and are only used as an aid to expressing the interface specification). Implementations are not permitted within abstract function, selector, operator, schema and constructor declarations. Member variables may be made conditional using the **when**...**end** construct.

Any invariants in the abstract section specify relationships between abstract members which must be true for any object of the class; it is an implicit part of the postcondition for all constructor and schema members in all sections of the class following the invariant.

7.1.5 Internal members

The optional internal member declarations describe how the class is implemented. The internal member declarations may include constant, variable, function, selector, operator, schema, property, axiom and class declarations.

Internal members serve the following purposes:

- To allow additional redundant data to be stored (e.g. to save having to recalculate a function of the abstract data each time it is needed)
- To allow the abstract data to be represented in a more efficient but more complex manner
- To allow additional functions, schemas etc. to be declared for use in the implementations of confined and interface members

[SC] Within the internal section, abstract variable members of the class may be redeclared as function internal members. In such redeclarations, the usual "**:** *TypeExpression*" is omitted since the type has already been specified; no parameters are permitted in the declaration. Redeclaration as a function indicates that the abstract member may be computed from internal variable members according to the function body. The implied precondition of such a function comprises the following elements:

- All invariants inherited from the parent class
- All invariants in the abstract section
- Those invariants in the internal section preceding the redeclaration
- If the original abstract variable declaration was inside a **when** clause, the corresponding guard

Abstract member redeclarations may not be recursive (i.e. internal members referred to in an abstract member redeclaration may not be described directly or indirectly in terms of that abstract member).

Abstract members which are functions, operators, selectors, schemas or constructors may be reimplemented. In such a reimplementation, no result type, precondition, postcondition, result value or specification variant is given; however, a parameter list is given if required (the parameter names must match the ones in the original declaration).

In a final class, interface and confined functions and operators whose return type is the type of the class may be reimplemented as schemas. The implied specification for the schema is that **self** has changed to be equal to the result value of the function or operator. These reimplementations may be used for more efficient implementation of postconditions which require this state change.

Any abstract variables which are not redeclared as internal functions are represented directly. If the **internal** section is absent all abstract variables will be represented directly.

Any invariants in the internal section describe restrictions on the values of the internal members.

[SC: each *Identifier* within an *InternalRedeclaration* is the name of an abstract variable member.]

7.1.6 Confined and Interface members

The confined and interface declarations comprise the published interface to the class. A confined or interface declaration may be a function, selector, operator, schema, property, axiom or constructor declaration, or a redeclaration of an abstract variable or constant member.

Confined members are only accessible to derived classes; interface members are available to the public at large.

The specifications of confined and interface members may only refer to abstract members and to other interface and confined members (including inherited confined and interface members). The implementations of confined and interface members may refer to internal members as well.

Redeclaring the name of an abstract variable or constant member as an interface function member makes the member accessible but "read-only" to clients of the class. Such a redeclaration consists of the keyword **function** and the abstract member name; no type is specified (since the type was given in the abstract member list), no parameters are declared and no body is given (if the abstract member was redeclared as an internal function member, then the body has already been given, otherwise the abstract member is being implemented as a variable member and a body is constructed which just returns its value). If the abstract variable is declared in a **when** clause, the corresponding guard becomes the implied precondition of the interface function.

Redeclaring the name of an abstract variable member as an interface selector member makes the member completely accessible to clients of the class. Such a redeclaration consists of the keyword **selector** and the abstract member name; again, no type is specified and there are no parameters or body. The abstract data member concerned may not have been redeclared in the internal section as a function, nor may any class invariant in the current class or any derived class directly or indirectly depend on it. If the abstract variable is declared in a **when** clause, the corresponding guard becomes the implied precondition of the interface selector.

Redeclaring the name of an abstract variable as a confined function or selector has a similar effect except that accessibility is limited to members of the current class and derived classes.

A class declared in the abstract member declaration section may not be used as the type of a parameter or the result of a confined interface member (because the class declaration is not visible to clients of the class).

7.1.7 Rank and equality declarations

The interface section may also include declarations for the rank and equality operators. In a declaration of rank or equality, neither the parameter type or return type are stated (these are always implicitly defined as the type of the class and **rank** or **bool** respectively).

Equality is always implicitly defined as equality of the abstract data members; so no result value may be declared in an equality operator declaration. The explicit declaration of an equality operator serves to specify whether equality can actually be evaluated at run-time (i.e. whether equality is **ghost**).

If equality is declared as a non-**ghost** operator in some class C, then equality in C and all its descendents is not **ghost**. Similarly, if equality is declared **ghost** in class C, then equality in C and all its descendents is **ghost**.

If a class C contains no equality operator declaration and neither do any of the ancestors of C, then C has non-**ghost** equality by default. However, decendents of C are still permitted to declare **ghost** equality. Therefore, equality on objects of type **from** C is treated as **ghost** to allow for this possibility. If you require the type **from** C to have non-**ghost** equality, then you must declare a non-**ghost** equality operator explicitly in C or in one of its ancestors.

A rank (i.e. comparison) operator may be declared and defined in any way that, for all values x, y, z of the class in which the rank operator is being defined, satisfies the following properties:

- $x = y \Longrightarrow x \sim\sim y = $ **rank same**
- $x \sim\sim y = $ **rank same** $\Longleftrightarrow y \sim\sim x = $ **rank same**
- $x \sim\sim y = $ **rank below** $\Longleftrightarrow y \sim\sim x = $ **rank above**
- $x \sim\sim y = $ **rank same** $\Longrightarrow z \sim\sim x = z \sim\sim y$
- $x \sim\sim y = $ **rank below** $\& y \sim\sim z = $ **rank below** $\Longrightarrow x \sim\sim z = $ **rank below**

If the operator is declared **total** it must also satisfy:

- $x \sim\sim y = $ **rank same** $\Longrightarrow x = y$

If a rank operator has already been declared in one of the class's parents then it is also required that the new rank operator does not change any of the existing ordering, it merely refines it. If we use $\sim\sim_c$ to represent the rank operator in the child class and $\sim\sim_p$ to represent the rank operator in the parent class, then the condition to be satisfied is:

- $(x \sim\sim_c y = x \sim\sim_p y) \mid (x \sim\sim_p y = $ **rank same**$)$

Note that the system is at liberty to further refine any user definition of rank that is not already **total**. In particular, where the user-defined rank would cause objects of different classes to **rank same**, the system *will* refine the definition so that this is not the case.

In the absence of an explicit declaration of the rank operator in a class, the system will generate one. The definition may be trivial (e.g. all instances of the class **rank same** with each other), or more complex; but it is guaranteed to respect any inherited rank operator.

7.1.8 Nonmember declarations

A function, schema or property declaration within the abstract, internal, confined or interface section of a class declarations may be declared **nonmember** to indicate that the declaration is not a member of the class and does not have the "current object" parameter. Nonmember declarations are not considered to be class members even though they are declared within the body of a class. The **nonmember** keyword may be not be applied to an operator or selector declaration, neither may it be applied to a method that is declared within an implementation or is declared at file scope.

For each confined or interface nonmember function or schema declared, either the result type (if a function member) or at least one parameter should be of the class type, or a union involving the class type, or a template instantiation involving the class type in its parameters; otherwise the declaration does not belong inside the class definition.

An interface nonmember function or schema may be called from outside the class by following its name with the "@" symbol and the class name, then the parameter list, if any, as in *myNonmemberMethod@MyClass*(*aParameter*).

An abstract constant may be redeclared as a nonmember confined or interface function using the same syntax as for the redeclaration of an abstract variable, save that the **nonmember** keyword must be used.

7.1.9 Recursive class declarations

It is permissible for a class declaration to be recursive, i.e. it may have data members (abstract and/or internal) which involve its own type, provided such recursion is finite. In practice this means that in a declaration of class **X** it is permissible to declare members of types **X** || **Y** (where **Y** does not refer to **X**), **seq of X**, **set of X** and **bag of X** (provided an empty collection is permitted), and other types which have conditional members of type **X** (provided the condition is not always true).

7.1.10 Class invariants

A class invariant is a predicate (i.e. a Boolean expression) declared in the **abstract** or **internal** section of a class, whose purpose is to constrain the values of the member variables to combinations for which the invariant yields **true**. To this end, a class invariant is:

- An implicit precondition of every member method whose declaration it logically precedes;
- An implicit post-assertion of every constructor and member schema whose declaration it logically precedes.

An invariant logically precedes a declaration *D* if and only if *D* is declared later in the text than the invariant but within the same **abstract** or **internal** section, or *D* is declared within the **confined** or **interface** section of the same class.

A class invariant may not refer to any declaration that it logically precedes; neither may an invariant declared in the **abstract** section refer to a declaration in the **internal** section.

A class invariant may not depend on any variable whose value can be accessed (in whole or in part) by means of a call to an interface selector and thereby altered such that the invariant is violated. This restriction ensures that any modification of an object other than **self** by means of selector access is bound to preserve the invariant.

The only explicit or implicit references to **self** permitted in a class invariant are references of the form "**self**.*x*" where *x* is the name of a member variable, or one of "**self** .*f* (...)", "**self** *op* ..." or "... *op* **self**" where *f* or *op* is the name of a member function or operator whose declaration logically precedes the class invariant or which is inherited from an ancestor class in which it is declared **early**. This ensures that an invariant cannot be defined in terms of members that assume the invariant is satisfied.

In the **abstract** section, constructors can only be declared after all invariants in that section. Similarly, if an abstract constructor is re-implemented in the **internal** section, then that re-implementation must be after any and all invariants have been declared in the **internal** section.

Multiple comma-separate invariants may be declared following the **invariant** keyword.

A method or constructor need not respect any invariant that does not logically precede its own declaration. A method or constructor postcondition or body may temporarily break a class invariant that does logically precede the method or constructor declaration, provided that the invariant is satisfied both on completion of the postcondition or body and at the point of every call to a method whose declaration the invariant logically precedes.

7.1.11 History invariants

Whereas a class invariant places a constraint on the values that the abstract variables of a class may take, a history invariant places constraints on the ways in which the abstract variables may be changed. Declaring a history invariant in a class is equivalent to declaring a corresponding postassertion on every member schema that modifies the current object and is declared textually later than the history invariant.

A history invariant is distinguished from a class invariant by the initial "**!**" symbol before the keyword **invariant**. Each expression in a history invariant must refer to at least one instance of **self '** or to a primed abstract member variable. Unliike class invariants, history invariants may only appear in the **abstract** of a class and not in the **internal** section.

The optional **exempt** clause lists the names of schemas that are exempted from having to comply with the history invariant. This provides a means to specify that some sorts of changes (i.e. those that do not satisfy a history invariant) may only be made by certain named schemas.

When class inheritance is used, any history invariants declared in a parent class apply not only to schemas declared textually later in the parent class, but also to all modifying schemas declared in any descencent class, apart from schemas whos names appear in the exemption list.

7.1.12 Storable classes

A class or class template declaration is termed storable if it has no **inherits** part but includes the keyword **storable** after the "**^=**" symbol, or if it inherits a storable class. If the declaration for class *X* is storable, all objects of types *X*, **from** *X*, **ref** *X* and **ref from** *X* are storable, meaning they may be written to and read from external storage or communication channels.

Within the declaration of a storable class, all variable members of the class (other than abstract variables redeclared as internal functions) must have storable types.

Objects of type *X* **of** (*A*, *B*...), **from** *X* **of** (*A*, *B* ...) and **ref** *X* **of** (*A*, *B*...) are storable if and only if the declaration of class template *X* **of** (*Y*, *Z* ...) is storable and objects of types *A*, *B*... are all storable. A united type is storable if and only if all the types in the union are storable. Adding a constraint to a type does not affect its storability.

Predefined types **bool**, **byte**, **char**, **int**, **real** and **void**, and template types **seq**, **set**, **bag**, **pair**, **triple** and **map** are storable, as are all enumeration and tag types.

7.2 Constructors

Constructor declarations provide the means to build values of the class. The syntax is:

> *ConstructorDeclaration*:
>   [**opaque** | **ghost**] **build** "**{** " [*ConstructorParameterList*] "**}**" [*ExceptionSpecification*]
> [*RequirePart*] [**pre** *PredicateList*] [**decrease** *Variant*] [*ConstructorBody*] [*PostAssertion*].

ConstructorParameterList:
    ConstructorParameters [**repeated** *FunctionParameters*];
    repeated *FunctionParameters*.

ConstructorParameters:
    ConstructorParams *[*Separator ConstructorParams*].

ConstructorParams:
    ["**!**"] *Identifier* *[*Separator* ["**!**"] *Identifier*] "**:**" *TypeExpression*;
    ["**!**"] *Identifier* "**:**" *AbbrevTypeExpression*.

ConstructorBody:
    "**^=**" *Expression* [*Implementation*];
    inherits *Expression*;
    [**inherits** *Expression*] **post** *Postcondition* [*Implementation*].

The first form of *ConstructorBody* is used to define the result of a constructor in terms of an expression yielding a value belonging to the class (typically the expression invokes another constructor for the class). The second form defines the result in terms of the value of the parent object (the expression following the **inherits** keyword) and the values of the abstract variable members (which must be defined in the postcondition).

Where a formal parameter name is prefixed by the symbol "**!** ", the parameter name must match the name of an abstract data member of the class and the body (if present) must have the " **post** *Postcondition*" form (not the "**^=** *Expression*" form). The type of the parameter must be equal to, or a subtype of, the type of the abstract data member. *Postcondition* is implicitly expanded to include the condition that the final value of the abstract data member is equal to the value of the actual parameter. Thus, using the form "**!**name" in the parameter list is a shorthand for using instead the form "name2" in the parameter list and appending ", name! = name2" to the postcondition which follows **post** (where "name" is an abstract data member and "name2" is not).

Where the constructor has an implementation, assignments are also added to the start of the implementation for every "**!** " parameter corresponding to a data member which has not been re-implemented as an internal function. Where the constructor itself appears in the internal section, the same rules apply, but the " **!**" parameter name must match either an internal data member or an abstract data member which is not re-implemented.

The optional post-assertion specifies additional properties of the returned object that are expected to hold. These properties should be provable from the postcondition or result expression. Within the post-assertion of a constructor declared using "**^=**" and a result expression, the predefined identifier **result** refers to the returned object, and each expression in the post-assertion must refer to **result**. Within the post-assertion of a constructor declared with a postcondition, the returned object is denoted by **self '**, and each expression in the post-assertion must refer to **self '** or to a primed member variable.

[SC] Every abstract class declaration containing member declarations must include at least one constructor declaration that does not use the "**^=** *Expression*" form of body (otherwise it would be impossible to construct objects of the class).

[SC] A constructor for a derived class must have a body, using either the "**^=** *Expression* " syntax or including an "**inherits** *Expression*" part.

[SC] A constructor that does not use the "**^=** *Expression*" syntax must define the values of all its data members, except for members declared within a **when** clause if the corresponding guard is false (if the "**^=** *Expression* " syntax is used, all the data members are bound to be defined anyway).

[SC] If the **opaque** keyword is used, the constructor does not uniquely define the object. Note that if the class includes one or more reference members and the constructor creates new values on the heap for the reference member(s) to address, the object cannot be uniquely defined so this is one instance where **opaque** must be used.

7.3 Derived abstract classes

7.3.1 Inheriting another abstract class

When an abstract class declaration inherits from another abstract class then all the confined and interface members of that class are inherited by the new class. The abstract and internal members are inherited at an implementation level but are not directly accessible by the inheriting class. Additional abstract, internal, confined and interface members may be declared.

It is not permitted to inherit from a class that was declared **final**, neither is the type **from T** permitted if **T** is a class declared **final**.

7.3.2 Overriding inherited declarations

Provided certain restrictions are obeyed, confined and interface function, selector, operator and schema members of the inherited class may be overridden by new declarations. The new declaration must be in the same section (confined or interface) as the old, have the same parameter and result types as the old and must be preceded by the keyword **redefine** to indicate that the overriding is deliberate. The specification of the overriding declaration must conform to the specification of the overridden declaration as follows:

- If a precondition is given in the overriding declaration, then the precondition of the overridden declaration must imply this new precondition whenever the class invariants are satisfied. If no precondition is given in the overriding declaration, the precondition of the overridden declaration is assumed.
- If an assertion is given in the overriding declaration, then this new assertion must imply the assertion of the overridden declaration whenever the class invariants are satisfied. If no assertion is given in the overriding declaration, the assertion of the overridden declaration is assumed. An overriding assertion may begin " **...**", in which case the stated assertions are added to those of the overridden declaration.
- The validity of any inherited member properties must be preserved.

Declarations that were declared **final** or **early** may not be overridden. Overriding declarations may also be declared **final** or **early**.

7.3.3 Accessing overridden members

When a class overrides a non-deferred member, the corresponding member in the parent class can be accessed by prefixing its name with the keyword **super**. It is not permissible to apply more than one **super** to a member name. If the member being invoked is a schema that modifies the current object, the "**!**" symbol precedes **super**.

7.3.4 Deferred abstract classes

It sometimes happens that we wish to have a client invoke some operation on a class without understanding the effect of the operation on the class. Typically, the operation will correspond to some real-world action which may vary greatly between objects of different classes derived from a base class. In this situation we may use a base class which includes one or more named and typed but undefined members.

A deferred base class is declared in exactly the same way as a normal non-inheriting abstract class except that the entire declaration is prefixed with **deferred** and one or more of its interface function, operator, selector or schema member declarations may be a deferred declaration.

No variable, parameter or result may be declared as having a type which is a deferred class. A deferred class name may only appear within an **inherits** part in an abstract class declaration or after the **from** keyword in a type expression.

Whenever a class declaration inherits a deferred class without defining all of the deferred members, or has one or more declarations of deferred members, it is itself deferred and its declaration must be preceded by **deferred**.

Whenever a class member declaration overrides an inherited member declaration that was originally declared **deferred** and has not already been overridden, the keyword **define** must be used in place of **redefine** .

The constructors for a deferred class may only be invoked by the **inherits** parts of the constructors for classes derived from it.

Constructors for deferred classes may not call any member functions, operators, selectors or schemas on the current object apart from such members that were declared **early**. Member functions, operators, selectors and schemas that are declared **early** may not call any other member functions, operators, selectors or schemas unless they too are declared **early**. The purpose of these rules and the **early** keyword is to prevent the possibility of calling a deferred member.

Those abstract and internal functions, operators, selectors and schemas which precede one or more invariants are automatically considered **early**, and in addition are not allowed to call members declared in the current class with fewer following invariants. It is not permitted to apply the **early** keyword to their declarations.

7.4 Class templates

A class may be declared with one or more template parameters, representing unspecified types that will be instantiated whenever an object of the class is created. By default the only methods available on objects of template type are the rank operator and a **ghost** equality; the optional *RequirePart* specifies other methods that

should also form part of the interface of any class with which we instantiate the template.

The full grammar for the **require** part is as follows:

> *RequirePart*
>> *RequireItem* *["," *RequireItem*].
>
> *RequireItem*:
>> *Identifier* **within** *TypeExpression*;
>> *Identifier* **has** *PrototypeList* **end**.
>
> *PrototypeList*:
>> *PrototypeItem* *[";" *PrototypeItem*] [";"].
>
> *PrototypeItem*:
>> *FunctionEtcHeader* [**pre** *PredicateList*] [**assert** *Assertion*];
>> **build** "{ " [*ConstructorParameterList*] "}" [**pre** *PredicateList*] [**assert** *Assertion*];
>> **operator** "=" "("*Identifier* ") ";
>> **total operator** "~~" "(" *Identifier* ") ".

The *Identifier* in each *RequireItem* must bind to one of the class's template parameters. The **within** item specifies a common base class that all types used to instantiate that parameter must inherit from. The **has** item specifies a list of methods together with preconditions and assertions for the verifier.

The requirements are statically checked whenever the template is instantiated.

8. Implementations and Proof Lists

8.1 Overview

Implementations are used to tell the compiler what algorithm to use.

There are two types of implementation: value implementation and state implementation. A value implementation specifies how a value is to be computed; a state implementation specifies how a predicate is to be satisfied.

An implementation that directly follows "^=" and an expression is a value implementation, as are implementations for functions specified with a **satisfy** condition. All other implementations are state implementations.

8.2 Syntax of implementations

Implementations are built up from implementation items, separated by semicolons. A variant may be given at the start if the implementation is recursive. The grammar is:

> *Implementation*:
>   **via** [**decrease** *Variant* "**;**"] *ImplList* ["**;**"] **end**.

> *NvImplementation*:
>   **via** *ImplList* ["**;**"] **end**.

> *ImplList*:
>   *ImplItem* *["**;**" *ImplItem*].

> *ImplItem*:
>   *LocalDeclaration*;
>   *LetStatement*;
>   *PostCondition* [*NvImplementation*];
>   *ConditionalStatement*;
>   *ValueCompletor*;
>   *StateCompletor*;
>   *Assertion*;
>   *Label*;
>   *Jump*;
>   *BlockStatement*;
>   *Loop*;
>   *ThrowStatement*;   *TryStatement*.

The initial *Variant* is only needed if the implementation is recursive and either the specification was not recursive (so no variant was given in the specification) or a the specification variant is insufficient for the implementation.

[SC] A discontinuous statement is defined as any statement which is a *Jump*, *ValueCompletor*, *StateCompletor*, *ThrowStatement*, or a conditional statement each of whose branches ends in a discontinuous statement, or a try-statement whose body and catch-part are both discontinuous. A discontinuous statement may only appear as the last item in an implementation, the last item in a branch of a conditional statement, the last item in a block statement, the last item in a try-block or catch-block, or immediately before a label (because the following statements would otherwise be unreachable).

[SC] The last *ImplItem* in an implementation of a function, operator or selector must be a discontinuous statement.

8.2.1 Declarations

> *LocalDeclaration*:
>   *LocalVarDecls*;
>   *FunctionDeclaration*;
>   *SchemaDeclaration*;
>   *PropertyDeclaration*;
>   *AxiomDeclaration*;
>   *ClassDeclaration*;
>   *TypeDeclaration*;
>   *HeapDeclaration*.

> *LocalVarDecls*:
>   **var** *LocalVarDeclGroup* *["**,**" *LocalVarDeclGroup*].

> *LocalVarDeclGroup*:
>   *Identifier* *["**,**" *Identifier*] "**:**" *PossConstrainedTypeExpression*;
>   *Identifier* "**:**" *AbbrevTypeExpression*;
>   *Identifier* "**:**" *TypeExpression* "**!**" "**=**" *Expression*;
>   *Identifier* "**:**" "**(**" (*ConstrainedTypeExpression* | *AbbrevTypeExpression*) "**)**" "**!**" "**=**"
> *Expression*.

Variables, types, classes, functions, schemas, properties and axioms may be declared. The forms of declarations in an implementation are similar to the forms of global declarations. Declarations within implementations are referred to as local declarations and are not member declarations irrespective of whether the implementation is for a member or nonmember entity.

Local variables may be initialized at the time of their declaration using the "**!=**" forms of declaration.

8.2.2 Let-statement

> *LetStatement*:
>   **let** *Identifier* "**^=**" *Expression*.

The let-statement is used to evaluate an expression and save the resulting value. It is similar to a constant declaration except that the expression need not be a compile-time constant.

8.2.3 Postcondition statement

The postcondition statement declares a postcondition to be satisfied. The rules defining what may be changed are exactly as for schema postconditions. The postcondition may itself be implemented.

8.2.4 Labels

Labels serve as the targets for jumps. The syntax for a label is:

> *Label*:
> > *Identifier* "**:**" **pre** *PredicateList*.

The scope of a label is determined in the same way as the scope of a declaration (which means that it is not possible to jump into an implementation from outside).

The label precondition must be satisfied at the point immediately prior to the label unless that point is immediately after a jump. It must also be satisfied at all jumps to the label. Following the label, the only knowledge available to the validation is the following:

- The predicates in *PredicateList*
- Type constraints on variables and parameters
- All class invariants and preconditions on the current object (if the implementation is not permitted to change it) and on all parameters (including those that can be modified) and local variables.

Therefore, the predicate list must encapsulate all assumptions required to satisfy the proof obligations of the following statements up to and including the next jump, label or completor (if any), or otherwise to the end of the statement list.

[PO: the precondition is true on the fall-through.]

8.2.5 Jumps

A jump may be made to any label which is defined at a later point in the current implementation and is in scope at the point of the jump, provided that the label precondition is satisfied at that point. The syntax is:

> *Jump*:
> > **goto** *Identifier.*

Labels obey the same scope rules as other declarations. Therefore, a label declared in an inner block cannot be referred to from outside the block; so jumping into a block is not possible.

Jumps are permitted only if there are no declarations or let-statements (other than within block statements, conditional statements, loops and nested implementations) between the jump and the target label.

[Note: it might be thought that jumps are anachronistic and have no place in a modern programming language; however, there are some situations where their use can simplify the code substantially. By insisting on a specification for the target of every jump, we tame the label/jump combination.]

[SC: the identifier after **goto** corresponds to a label in scope and within the current implementation. The jump is to a later point in the code.]

[PO: the precondition of the label which is the target of a jump is satisfied at the point of the jump.]

8.2.6 Loops

Loops have the following syntax:

> *Loop*:
> > **loop**
> > > *LoopChangedVars*
> > > **keep** *PredicateList*
> > > [**until** *PredicateList*]
> > > **decrease** *Variant* "**;**"
> > > *ImplList* ["**;**"]
> >
> > **end**.
>
> *LoopChangedVars*:
> > [*LocalVarDecls* "**;**"] **change** *ExpressionList*;
> > *LocalVarDecls* "**;**".

The loop begins with a set of local variable declarations (which will also usually initialize the variables), together with, following the word **change**, a list of variables from the enclosing scope which the loop is permitted to change. All variables other than those declared and listed in this section must remain constant.

The predicate list which follows **keep** is the loop invariant, a set of conditions which must be true at the start of the loop and remain true throughout its execution. Together with the **until** part, the loop invariant forms the postcondition for the loop.

The predicate list after **until** may be viewed both as a termination condition and as a partial postcondition for the loop (if multiple comma-separated predicates are given, they are &ed together as usual). If no **until** part is specified, the condition that the variant can no longer decrease is used (for example, if the variant consists of a single integer value, the loop will terminate when it is zero)

The variant following **decrease** follows the same rules as for variants in recursive functions and schemas (each iteration of the loop must decrease the variant).

Within the invariant, until-part and variant, primed expressions refer to the value of the expression at the start of each iteration; unprimed subexpressions refer to the value before any part of the loop is executed. Only objects which the loop is permitted to change may appear primed. Local variables must *always* appear primed as they have no meaning before the loop is executed (since they are declared within the loop).

The final implementation comprises a loop body which decreases the variant while preserving the invariant. This implementation may only modify local variables and terms which were declared in the **change** list. Within the loop body primed and unprimed expressions have their usual meanings.

[PO: The loop invariant is true on loop entry (where local variables have their initial values). The variant is valid on loop entry. If the until part is false, the loop body preserves the loop invariant, and either decreases the variant or causes the until part to become true.]

8.2.7 Assertions

Assertions declare a set of predicates to be true at a point in an implementation without further work being necessary to achieve them, provided only that the original precondition is satisfied. They may serve as hints to the theorem prover as well as additional validation checks. Even if an assertion is unproven, it is still assumed satisfied during validation of the statements that follow. The syntax for assertions was given in <u>section 5.4.2</u>.

8.2.8 Conditional statement

> *ConditionalStatement*:
>   **if** *Guard ImplList* *["**;**"*Guard ImplList*] ["**;**" *LastImplGuard*] **fi**.

> *LastImplGuard*:
>   *EmptyGuard ImplList* ["**;**"];
>   "**;**";
>   "**[**" "**]**".

The conditional statement is structurally similar to the conditional expression except that the enclosing brackets are replaced by **if fi** and each guard is followed by an implementation item list instead of an expression. If no *EmptyGuard* part is present, at least one of the guards must be satisfied.

Each *ImplList* within a conditional statement defines a new scope; therefore any declarations within a branch of a conditional statement are not visible in other branches, nor are they visible outside the conditional statement.

8.2.9 Block statements

> *BlockStatement*:
>   **begin** *ImplList* ["**;**"] **end**;
>   **par** *ImplList* ["**;**"] **end**.

The **begin**...**end** block statement puts the enclosed implementation in its own scope. The **par**...**end** statement is similar, except all the statements in the list are notionally executed in parallel, and so must modify disjoint objects.

8.2.10 Value completors

A value completor has the syntax:

> *ValueCompletor*:
>   **value** *ExpressionList*.

Value completors are only allowed in value implementations (not in state implementations). The number of expressions in *ExpressionList* must be one, except for implementations of functions that return multiple results, in which case the number of expressions must either match the number of returned results or there must be only one expression and this must take the form of a recursive call to the function. The expression(s) must be provably equivalent to the required value(s) as defined in the specification of the function, operator, selector or constructor which is being implemented. The effect of a value completor is to exit the entire implementation, yielding the specified result value.

[PO: all variables and function calls in *Expression* which are not present in the original specification of the value of the function can be eliminated using information about the state at this point and the result is equivalent to the value of the function in the original specification.]

8.2.11 State completors

A state completor comprises the keyword **done**.

> *StateCompletor*:
> **done**.

State completors are only allowed in state implementations. There is an implicit assertion that the schema or constructor postcondition which is being implemented has been satisfied at the point of a state completor. The effect of a state completor is to exit the implementation.

There is an implicit state completor at the end of any state implementation whose final statement is not discontinuous.

[PO: the required postcondition is satisfied at the point of a state completor.]

8.2.12 Throw statements

Throw statements raise or re-raise exceptions:

> *ThrowStatement*:
> **throw** *Expression*;
> **throw**.

The first form raises a new exception, abandoning the containing implementation list. The second form, which is valid only in the catch-part of a try-statement, re-throws the exception caught in the catch-part.

The exception that is thrown or re-thrown must either be caught in the catch-part of an enclosing try-statement within the containing function, selector, operator, schema or constructor, or else it must be of a type contained in the exception signature of the containing function, selector, operator, schema or constructor. This condition is checked by the *Perfect* compiler.

Important note: The semantics of throw statements are not fully defined in this edition. In particular, there is currently no requirement or verification condition that class invariants are preserved at the point a throw-statement is executed. Therefore, if throw-statements are used in implementations that mutate objects, then when the exception is caught, object invariants may have been broken. We strongly recommend that you

use throw-statements only in contexts where no modifications to the state of any object have been made between the start of the try-statement whose catch list catches the exception, and the throw-statement; except that modifications to objects that have passed out of scope when the catch list is reached are safe. It is expected that critical software will typically not use exceptions.

8.2.13 Try statements

A try statement executes a contained implementation, catching some or all exceptions that it may throw:

>*TryStatement*:
>>**try** *ImplList* [**";"**] **catch** *CatchItem* *[';' *CatchItem*] [**";"**] **end**.

>*CatchItem*:
>>'[' *Identifier* ':' *TypeExpression* ']' ':' *ImplList*.

The implementation list following the **try** keyword is executed. If that implementation list throws an exception, then control is transferred to the catch list. Each catch item declares a type to be caught, an identifier to name the value of the caught exception (so that it can be used in the following implementation list), and an implementation list to be executed when catching exceptions of that type. The catch item that is executed will be the first one (lexically) whose type is or includes the actual type of the exception that was thrown. At most one catch item will be executed, even if the types declared in the catch items are not disjoint. If the exception was of a type that is not contained in any of the types of the catch items, then it is not caught in that catch list and continues to propagate out of the context surrounding the try statement.

8.3 Proof lists

Proof lists are used to give hints to the theorem prover about how an assertion or property might be proved. The grammar is:

>*Proof*:
>>**proof** *ProofList* [**";"**] **end**.

>*ProofList*:
>>*ProofItem* *[**";"** *ProofItem*]**;**
>>*[*ProofItem* **";"**] **if** *Guard ProofList* *[**";"** *Guard ProofList*] [**";"**] **fi**.

>*ProofItem*:
>>*Assertion*;
>>*LetStatement*.

A proof list contains a sequence of assertions and temporary name definitions separated by semicolons. The last element in a proof may be a conditional proof.

The theorem prover should attempt to prove the predicate lists in the order in which they are given, using each assertion as an assumption when proving later assertions in the proof list. The prover will attempt to prove the assertion to which the proof list is attached by assuming all assertions in the proof list are satisfied.

9. Scopes, Overloading and Binding

9.1 Overview

This section describes the scope rules; i.e. the rules which determine whether it is legal to redefine a particular identifier and how the compiler binds each use of an identifier to a declaration. The principles employed are:

- An identifier may be used to represent multiple entities in the same region provided that it is not possible to write a construct in which the use of the identifier is ambiguous.
- A declaration is never permitted to hide another declaration of the same identifier inherited from an outer block.
- The order of declarations within each block plays no part in determining the binding of the uses of identifiers to the corresponding declarations. However, forward-referencing is prohibited where there is no good reason to allow it.

Within this chapter, the term "function" should be taken to encompass operators and selectors also.

9.2 Name spaces

Identifiers in *Perfect* fall into the following categories:

- Class, type and class template parameter names
- Constant, variable, parameter, function, selector and schema names
- Heap names
- Label names
- Property and axiom names

The *Perfect* language is designed such that it can always be determined from context which category an identifier belongs to. Each category therefore has its own name space (meaning that at any point in the program, the same name may stand for one entity of each category).

9.3 Definition of the various declaration contexts

- A declaration is global if it occurs within the global declaration list.
- A declaration is local if it occurs within an implementation. All label declarations are local.
- A declaration is member if occurs within a class abstract, internal, confined or interface section and is not declared **nonmember**.
- A declaration is nonmember if occurs within a class abstract, internal, confined or interface section and is declared **nonmember**.

9.4 Overloading class and type names

Where an identifier is declared as the name of more than one class or type, each declaration must have a distinct signature. The signature of a class or type declaration is defined as the number of template parameters in the declaration together with the sequence of separators used to separate them.

Within a class or type template declaration, the declaration of an identifier as a parameter is considered to declare that identifier as a type name with no template parameters. Within a polymorphic function or schema declaration, an identifier which follows the keyword **class** in the parameter list is considered to declare that identifier as a type name with no template parameters.

It is not permitted to hide a class or class template declaration by means of a conflicting class or class template declaration in an inner block.

[Note: we could permit such hiding, but then we would have to define the rules under which it is permitted, which gets quite complicated when we consider inheritance. Or, to avoid having to make class template parameter names distinct from all visible class names and template parameter names, we could use a different syntax when referring to them, e.g. by prefixing template parameter names with **class** always, thereby giving template parameter names a different namespace.]

9.5 Overloading variable and function names

At any point in the program, an identifier may represent any number of constants, variables, let-names, functions, selectors, schemas and formal parameters, subject to the following rules provided that all of the corresponding declarations have distinct signatures.

The signature of a declaration of a variable, function schema etc. comprises two elements:

- A sequence of classes, which are the classes of its parameters (the implicit **self** parameter for class members does not count). In evaluating this sequence, constraints are discarded and type names are replaced by their definitions.
- A sequence of separators, which are the separators used to separate the parameters.

In the case of a variable, constant, parameter or let-declaration, both sequences are empty.

In the case of a polymorphic declaration, the sequence of classes will include polymorphic parameters and class or type template instantiations that depend on one or more polymorphic parameters.

A declaration with a parameter list having a **repeated** section is considered as having an infinite family of signatures. The members of this family are obtained by removing the keyword **repeated** and the following group of parameters and separators, substituting instead one or more repetitions of that group, with a comma inserted between repetitions.

Two signatures are distinct if any of the following is true:

- They have different lengths
- They have different separator sequences
- There is an index into the two sequences of classes for which the corresponding items are disjoint classes and neither is polymorphic or is an instantiation depending on a polymorphic parameter
- There are two indices into the two sequences of classes that have the same polymorphic class name in one of the signatures and disjoint non-polymorphic classes in the other

9.6 Overloading operator and selector symbols

At any point in the program, a symbol may represent any number of operators and, in the case of indexing, selectors; provided that all of the corresponding declarations have distinct signatures.

The signature of an operator declaration (or an indexing selector declaration) is a sequence of classes. For declarations that are not members, this sequence comprises the parameter types. For member declarations with no parameters, this sequence has a single entry which is the type of **self**. For member declarations with one parameter, this sequence comprises the type of **self** followed by the type of the parameter if the formal parameter follows the operator symbol in the declaration, or vice versa if the formal parameter declaration precedes the operator symbol in the declaration.

Parameter declarations involving type names, constraints or polymorphic class names are handled in the same way as for signatures of functions.

9.7 How binding is defined in Perfect

Binding in *Perfect* is defined in terms of dictionaries of available identifier definitions.

In each region in which identifiers may be used, there is a dictionary called the general dictionary. Every class also has two dictionaries, called the member dictionary and the non-member dictionary.

Whenever an identifier or operator symbol is mentioned other than for the purposes of declaring it, the appropriate dictionary is used to determine its meaning. If the identifier is preceded by an expression and then "**.**" or "**!**" then the member dictionary for the class corresponding to the type of the expression is used; if the identifier is followed by "@" and then a class name then the non-member dictionary of this class is used; otherwise, the general dictionary is used.

Binding of an identifier name is successful if the identifier is matched with an entry in the dictionary and access restrictions do not prohibit the binding.

9.8 Uniting and Core

In describing the available dictionary, we sometimes define a new dictionary by uniting an old dictionary with a set of new declarations. The resulting dictionary contains all definitions in the old dictionary together with definitions corresponding to the new declarations.

The core of a dictionary is defined as all its class or type declarations, global constant declarations, heap declarations, and local or nonmember function, operator, selector, schema, property and axiom declarations (i.e. the original dictionary less its variable, parameter, non-global constant, and member declarations other than heaps).

9.9 Definition of the general dictionary for various regions

9.9.1 Global declaration list

The dictionary for the global declaration list is the imported environment dictionary united with the declarations in the list.

9.9.2 General dictionary for declarations of functions, operators and selectors

The basic general dictionary comprises the core of the dictionary for the region in which the declaration occurs, united with:

- the declarations of the parameters
- for a member declaration, a declaration for **self** and the declarations of all members and non-members of the class (including all inherited members) except constructors
- for a non-member declaration, all the non-members of the class

The dictionary for parameter lists, preconditions and result values is the basic dictionary.

The dictionary for postconditions and assertions is the basic dictionary united with a declaration for **result** .

9.9.3 General dictionary for declarations of schemas

The basic general dictionary comprises the core of the dictionary for the region in which the declaration occurs, united with:

- the declarations of the parameters
- for a member declaration, a declaration for **self** and the declarations of all members of the class (including all inherited members) except constructors

The dictionary for parameter lists, preconditions, postconditions and assertions is the basic dictionary.

9.9.4 General dictionary for declarations of constructors defined using a result expression

The basic general dictionary comprises the core of the dictionary for the region in which the declaration occurs, united with the declarations of the parameters.

The dictionary for the parameter list, precondition and result expression is the basic dictionary.

The dictionary for the post-assertion is the basic dictionary united with a declaration for **result**.

9.9.5 General dictionary for declarations of constructors defined without using a result expression

The basic general dictionary comprises the core of the dictionary for the region in which the declaration occurs, united with the declarations of the parameters.

The dictionary for the parameter list, preconditions and **inherits**-part is the basic dictionary.

The dictionary for the postcondition and post-assertion is the basic dictionary united with a declaration for **self** and the declarations of all members of the class (including all inherited members) except constructors.

9.9.6 General dictionary for implementations

The dictionary for an implementation is the dictionary for the expression or postcondition it is implementing united with the declarations in the implementation (excluding declarations within loops, conditional statements and further implementations).

Within a loop statement, the dictionary for the variable declarations following the **with** keyword is the dictionary for the implementation in which it occurs; the dictionary for all other regions of the statement is that dictionary united with those variable declarations.

Within a conditional statement, the dictionary for the guards is the dictionary for the implementation in which it occurs. The dictionary for each statement lost that follows a guard is that dictionary united with the declarations in the statement list (excluding declarations within loops, further conditional statements and implementations).

9.9.7 General dictionary for the inherits-part of a class declaration

The dictionary within a class inherits-part comprises the core of the dictionary for the region in which the class declaration occurs united with the parameter names (if it is a template).

The dictionary for the optional class invariant directly following the inherits-part comprises that dictionary united with the non-internal members of the inherited class (including inherited members) and a declaration for **self**.

9.9.8 General dictionary for member declaration regions of a class declaration

The dictionary for member declaration regions of a class declaration is the dictionary for the inherits-part, united with all the declarations in the region (including nonmember declarations) and all non-internal inherited members.

The dictionary for class invariants (other than those directly following an inherits-part) comprises that dictionary united with a declaration for **self**.

9.9.9 Bracketed expressions

The dictionary within a bracketed expression is the dictionary for the region in which the expression occurs united with its let-declarations.

9.9.10 Expressions involving bound variables

The dictionary for a predicate following "**:-**" and for the expression following **yield** is the dictionary for the region in which the construct occurs united with the bound variable declarations. Where a quantified expression declares more than one bound variable declaration, the dictionary for each bound variable declaration is the dictionary for the region united with the preceding bound variable declarations.

9.10 Class member dictionaries

A class member dictionary comprises all declarations (other than nonmember declarations) of functions, operators, selectors and schemas in its abstract, internal, confined and interface sections, united with the dictionary of its parent class (if any) less the parent class internal members. Declarations in the class override declarations with identical signatures in the parent.

Each class also has a dictionary of constructors, comprising all constructor declarations in the class declaration.

9.11 Access restrictions

Even if the use of an identifier matches an entry in the dictionary for the region in which it occurs, there may be an access restriction prohibiting binding.

There are no access restrictions to class, heap or label declarations, nor to global, local or interface declarations.

Within class member declarations, full access is permitted to members of **self** and **it** in a subjunctive expression whose initial value is **self**, which are declared in the same class as the current declaration, or which are defined in the confined and interface sections of ancestor classes, with the following restrictions:

- Within specification of abstract, confined and interface members, access to internal members is prohibited
- Within abstract members defined before at least one abstract invariant, access to members with fewer following invariants (including all confined and interface members) is prohibited
- Within internal members defined before at least one internal invariant, access to members with fewer following invariants (including all confined and interface members) is prohibited

Within class member declarations, access to members of objects of the same class other than **self** or **it** as above is subject to the following additional restrictions:

- Write access to abstract data members other than those declared as interface selectors is prohibited (including those declared as confined selectors)
- Access to **early** schemas is prohibited

Access to class members from outside the class declaration is subject to the following restriction:

- Access to abstract, internal and confined members is prohibited

Where a member is declared in one section and redeclared or reimplemented in one or more other sections, it is considered as belonging to all of these sections and access to it is permitted, provided that it belongs to at least one section to which access is not prohibited.

9.12 Forward referencing and references to declarations in imported files

A reference to an entity declared in an imported file is treated as a forward reference, since the order in which the compiler processes a collection of files is not defined. In practice this has little effect because the only declarations in imported files that can be referenced are global declarations.

Forward referencing of class and type declarations (other than class template parameter names) is permitted, however infinite recursion is prohibited. In particular:

- A type declaration may not define a new type in terms of itself, nor may a circular chain of type declarations be constructed that has a similar effect
- A class declaration may not inherit from itself or from a template instantiation using itself as an actual parameter, nor may a circular chain of inheritance relationships be constructed
- If a class is referred to within a type expression in one of its own member declarations, the reference must be conditional either by the use of a **when**-clause or a union of types or a suitable template (e.g. a sequence), such that it is possible to define a constructor for the class which is not infinitely recursive; and similarly for any circular chain of such relationships

Forward referencing of global and member constant and variable declarations is permitted; forward referencing of local constant and variable declarations is prohibited.

Forward referencing of function, selector, operator and schema declarations is always permitted; so long as for all circular chains of references thereby arising, an appropriate variant is declared for each declaration in the chain. However, a class invariant may not forward reference any other declaration in the same class.

Forward referencing of let-declarations and parameters is prohibited.

Forward referencing of heap and label declarations is permitted.

10. Interface to other languages

10.1 Overview

The *Perfect* compiler produces implementation code from the *Perfect* source. This code may be expressed either in object module format or as source code in a target programming language.

It is often required to mix program elements written in *Perfect* with program elements written in a traditional programming language. This section describes some facilities available to achieve this. The exact behavior may be implementation-dependent and may depend on the target language.

10.2 Pragmas

Where an identifier or operator declared in the *Perfect* source causes a corresponding identifier to be declared in the output file, the compiler has default rules for generating the name of this external identifier from the original *Perfect* identifier. A developer may override these rules by placing a *Pragma* after the identifier or operator symbol being declared, or after the **build** keyword in a constructor declaration. The syntax is:

> *Pragma*:
>   **pragma** '(' **name** '=' *NonEmptyStringLiteral* ')'.

A pragma may be attached to any identifier or operator being declared. The effect is that for all instances of the identifier which bind to the declaration, the corresponding external name will be as specified in the string literal.

Normally, the *Perfect* compiler performs any name-mangling necessary to avoid *Perfect* identifiers clashing with reserved words in the output language or external names used in the run-time system or environment interface. It is the developer's responsibility to avoid clashes when using pragmas to define the names used in the generated code.

Pragmas can also be used to change the generated code in other ways, for example to suppress reference counting when generating C++.

11. Library overview

Library classes and methods are documented alphabetically in Appendix A. This section provides an overview of commonly-used operations by category.

11.1 Order and sorting

The built-in class `seq of X` has a number of members that relate to the ordering of elements. Many of these methods take a parameter of type from `Comparator of X`. The user may define the ordering required by declaring a class that inherits from `Comparator of T` (where `T` is the type of elements concerned) and defines the method compare. Predefined classes inheriting from `Comparator of X` are `SimpleComparator of X` (which compares according to the ~~ operator) and `ReverseComparator of X` (which defines the inverse ordering).

The sequence member `isOrdered` tests whether a sequence is ordered with respect to a comparator. Functions `isndec` and `isninc` are specializations of `isOrdered` using `SimpleComparator` and `ReverseComparator` respectively (so `isndec` means "is non-decreasing" according to normal element comparison using the "~~" or "<" operator).

Schema `sort` sorts the sequence so that it is ordered with respect to the given comparator. Functions `permndec` and `permninc` return new sequences which are sorted into nondecreasing and nonincreasing order respectively. These functions are also provided as members of classes `set of X` and `bag of X`.

11.2 Input/output

The `Environment` class is the interface through which *Perfect* programs may interact with the outside world. It is expected that in a future release of *Perfect Developer*, the environment class will be supplied as a separate library.

All methods referred to in this section are members of class `Environment` unless otherwise stated.

11.2.1 Console input / output

The `print` schemas of class `Environment` output characters to the standard output stream (usually the screen). The `printStdErr` schema outputs the given string to the standard error stream.

The `readLine` schema reads a line of text from the standard input stream (usually the keyboard); the value of `ret` will be either `success@FileError`, `attribError@FileError` or `otherError@FileError`. Note that the final line feed or carriage return + line feed are *not* returned.

The `stdIn`, `stdOut` and `stdErr` functions return the `FileRef` objects representing the standard input, output and error streams. In a future version of the library, these methods may return streams instead.

11.2.2 File operations

The `open` schema attempts to open a file named `filename` in the mode specified by the `mode` parameter as follows:

- `create@FileModeType` means a new file is created, and any existing file is deleted
- `append@FileModeType` means the file is opened and data written will be appended to the existing data
- `read@FileModeType` means the file may be read
- `text@FileModeType` means the file is opened as a text file in the underlying operating system, otherwise it is opened as a binary file

The modes `create@FileModeType` and `append@FileModeType` are mutually exclusive. At least one of `create@FileModeType`, `append@FileModeType` and `read@FileModeType` must be present.

If the open succeeds a FileRef is returned in the **out** parameter, otherwise a FileError is returned as follows:

- `fileNotFound@FileError` means the file was not present, and neither `append@FileModeType` nor `create@FileModeType` were specified
- `attribError@FileError` means the file was not accessible in the requested mode.

The `close` schema will attempt to close the given file. `ret'` will be set to either `success@FileError` is the close was successful, or `writeError@FileError` if it failed in some way.

The `print` schemas output character data to a given file. If the write fails `ret'` will be equal to `writeError@FileError`, otherwise it will be `success@FileError`.

The `write` schemas output binary data to a given file. The schema taking two **int** parameters will output the integer n as a sequence of `numBytes` bytes, most significant byte first. If the integer to be written is too large to fit into the specified number of bytes, the higher bytes will be lost. If the write fails, `ret'` will be equal to `writeError@FileError`, otherwise it will be `success@FileError`.

The `scan` schemas read character data from a file. If the read succeeds `ret'` will be equal to `success@FileError`, otherwise it will be either `readError@FileError` or `endOfFileError@FileError`.

The readLine schema attempts to read a line of text from the given file. If successful `line'` contains the line of text (including the linefeed and/or carriage return if present) and `ret'` = `success@FileError`. If the read failed `line'` is the empty string and `ret'` = `readError@FileError`.

The `read` schemas read binary data from a file. If the read succeeds `ret'` will be equal to `success@FileError`, otherwise it will be either `readError@FileError` or `endOfFileError@FileError`.

The `seek`, `fastForward` and `rewind` schemas attempt to position the file pointer at a specified position in the file. `ret'` will be set to either `success@FileError` is this was successful, or `seekError@FileError` if it failed in some way.

The `fileSize` schema returns the size of the data contained in a given file. The schema may change the file pointer of the queried file only if the call fails, in which case `res'` will be `seekError@FileError`. If the call succeeds, `res'` will be `success@FileError`, and the file pointer will be unchanged.

The `tell` function returns the position of the file pointer if it succeeds. If the call fails `seekError@FileError` is returned.

The `flush` schema attempts to flush any buffer associated with the given file. `ret'` will be set to `success@FileError` is the flush was successful, or to `flushError@FileError` if it failed in some way.

The ghost functions `gIsOpen`, `gFileData`, `gFilePtr`, `gFileAtEof` and ghost schema `gSetFilePtr` can be used to reason about the file system.

11.2.3 Disk operations

The `makeDirectory` schema attempts to create the given directory; all directories below the specified one will also be created if they are not already present. The returned `FileError` has the following meanings:

- `success@FileError`: the directory was successfully created
- `permError@FileError`: the process did not have permission to create the directory
- `createError@FileError`: a file with the same name as the given directory already existed
- `diskFull@FileError`: insufficient disk space
- `otherError@FileError`: the directory could not be created for some other reason.

The `move` schema attempts to move and rename the file given by `oldPath` to that given by `newPath`. If parameter `overwrite` is **true** then if the file `newPath` already exists it will be replaced by the moved file. The `res'` parameter will be set to one of the following values:

- `success@FileError`: the move succeeded
- `fileSpecError@FileError`: one or both file names were illegal, or both pointed to the same file
- `attribError@FileError`: the target file already exists, and `overwrite` is **false**
- `deleteError@FileError`: the target file already exists and could not be deleted
- `fileNotFound@FileError`: the source file did not exist
- `otherError@FileError`: the move failed in some other way.

The `delete` schema will attempt to delete the given file. `ret'` will be set to one of the following values:

- `success@FileError`: the delete succeeded
- `fileNotFound@FileError`: the specified file did not exist
- `deleteError@FileError`: the file could not be deleted.

The `fileValid` function returns **true** if the given file and pathname is legal and refers to an existing file or directory.

The `fileStatus` function returns information on the given file or directory. If the pathname passed ends in the `pathSepChar` then the call will only succeed if the path represents a directory. If the returned value is of

type FileStats, the call succeeded and this is the information. Otherwise the result will be:

- `fileNotFound@FileError`: the file did not exist
- `directoryNotFound@FileError`: the path existed as a file, but the pathname ended in `pathSepChar`
- `otherError@FileError`: no information could be found for some other reason.

The `forceFileTime` schema sets the last modified and accessed times on the given file. If either `modified` or `accessed` is **null**, that attribute is not affected. `res'` will be one of the following values:

- `success@FileError`: the dates were changed successfully
- `fileNotFound@FileError`: the file did not exist
- `otherError@FileError`: failed to change the dates for some other reason

The `setMode` schema (*not implemented in the Java version of the runtime library*) attempts to change the file mode flags on the named file. `res'` will be one of the following values:

- `success@FileError`: the mode was changed successfully
- `fileNotFound@FileError`: the file did not exist
- `permError@FileError`: permission was denied to change mode
- `otherError@FileError`: failed to change mode for some other reason

The `getImagePath` function returns the full path and file name of the current executable.

The `normalizeFile` function takes a path and file, and will return a `FilePath` object corresponding to this file. If the file is not valid, the result is **null**.

The function `getCurrentDirectory` returns the current directory in the underlying file system. If the call fails, the result is **null**. The schema `setCurrentDirectory` attempts to set the current directory to the specified string; `ret` will be set to one of the following values:

- `success@FileError`: the directory was successfully changed
- `directoryNotFound@FileError`: the directory did not exist
- `permError@FileError`: the process did not have permission the access the directory
- `otherError@FileError`: the call failed in some other way

The constant `caseSensitiveFileNames` indicates whether the file system distinguishes between upper and lower case letters. The constant `pathSeparator` is the character used to separate components of a file name (e.g. `\` in Windows, `/` in Unix).

11.2.4 Opening sockets

The two socket `open` schemas attempt to open a socket connected to the specified port of the host given either as a string or as an IP address. Returns either the socket if successful, or a socket error if not. Sockets facilitate communication with other machines over a network or the internet.

11.2.5 Other environment methods

```
schema !(priority: int)
     pre priority in 1..20
```

Schema `setCurrentThreadPriority` sets the priority of the current thread, where 1 is the lowest priority and 20 the highest.

Schema `execute` *(not currently implemented in the Java runtime library)* attempts to run the command `command` on the underlying operating system, passing arguments `args`, and optionally redirecting standard in, out and error to files. The schema returns when the command has completed, and `res` will be set as follows:

- `success@FileError`: the command was successfully executed
- `fileNotFound@FileError`: the command did not exist
- `attribError@FileError`: a file of the stated name was found, but was not executable
- `otherError@FileError`: execution failed in some other way

The `clock` function returns the number of clock ticks since the current executable was started; clocksPerSecond returns the number of clock ticks in a second. Thus, for example, the time in seconds since the current executable started is `clock / clocksPerSecond`.

Function `getCurrentDateTime` returns the current date and time as a `Time` object.

Function `getEnvironmentVar` returns the value of a given environment variable in the underlying operating system, or `null` if the variable is not set.

The `getImageVersion` functions return the version information for the current executable, or named module.

Function `getMemoryUsed` returns the amount of memory being used by the current executable.

Schema `garbageCollect` causes the process to return unused memory to the operating system.

Function `getOsInfo` returns the type and version of the underlying operating system.

11.2.6 Runtime checks and profiling

The `setRuntimeOptionState`, `setRuntimeOption` and `clrRuntimeOption` schemas can be used to change the amount of checking performed at runtime (if runtime checks are being performed at all). Schema `setMaxCheckNestLevel` determines how deeply checks will be performed. For example, if the setting is 2, runtime check points reached while running the program will be evaluated, also any check points reached whilst evaluating these check points will be evaluated; but runtime check points reached during this second-level evaluation will be skipped.

To perform run-time profiling (only available in certain versions of the C++ runtime library), call schema to begin collecting timing information. Calling schema `stopProfiling` will suspend the collection of timing information. To write the timing information to file, call schema `profile` (which implicitly calls

stopProfiling) having already created the file. In order to use this mechanism, the macro _dProfile must have been defined as 1 when compiling the generated C++ code.

11.3 Debugging functions

The global function debugPrint is provided to aid debugging, allowing diagnostic information to be output to the console even where no Environment object is available. The return value serves no purpose. Typically, this function will be invoked within a let-statement like this:

```
let dummy ^= debugPrint("Value of foo is " ++ foo.toString ++
"\n");
```

The function debugHalt also outputs the given string, but then causes execution to stop with a runtime error.

11.4 Streams

Stream-based input/output is provided by the classes InputStream, OutputStream and their descendents (*ByteInputStream*, *FileInputStream*, *StandardInputStream* and the corresponding output streams).

11.5 Serialization

Declaring a class **storable** implicitly means it inherits from Storable. This is the only way a class should inherit from Storable, and directly declaring **inherits** Storable is not allowed.

Storable objects may be stored and re-loaded using the global storeObject and loadObject methods. Note that there is no check that a loaded object satisfies any type constraints or class invariants defined for its class.

The serialization formats used by the C++, C# and Java versions of the runtime system are not compatible with each other.

11.6 Character encoding and decoding

An interface is provided to encode the *Perfect* **char** class into a sequence of bytes using a chosen scheme, and to perform to corresponding decoding. It is expected that these will be integrated into the environment print and scan methods in a future release of *Perfect Developer*.

The base class CharEncoderDecoder serves as an interface for generating encoders and decoders for particular schemes. These encoders and decoders are returned as descendents of CharEncoder and CharDecoder respectively.

An encoder simply contains a function which returns the encoding for any given character, plus a function to return a preamble for the encoding scheme used.

A decoder contains a `process` schema which is passed bytes until a complete character has been built, at which point the method `charReady` will return **true**. The character may then be extracted using `getCompletedCharacter`, the decoder may then be reset and further characters decoded.

12. Application Startup and Initialization

An application may be written entirely in *Perfect*, or it may be written partly in Perfect and partly in a traditional programming language. In the latter case, the program entry point may be in the portion of the system written in *Perfect*, or in the portion written in the programming language. This chapter explains what you need to do in these scenarios to ensure correct application startup and initialization.

12.1 Program entry point written in Perfect

The entry point of a program written in *Perfect* shall have the following signature:

```
schema main(args: seq of string,
            context!: limited Environment, ret!: out int)
   pre #args >= 1
   post ... ;
```

The first element of the *args* parameter contains the name by which the program was executed, if available, or the empty string if not. The remaining elements of *args* are the command-line parameters.

The *context* parameter provides access to the *Environment* in which the program was executed, so that input/output and similar operations with effects outside the program can be performed. Note that the *stdin*, *stdout* and *stderr* streams are not guaranteed to be open, since on many platforms it is possible to close these streams prior to executing a program.

The *ret* parameter provides means of returning a return code to the calling program. The program must assign *ret* before terminating. By convention, a zero return code indicates successful execution.

The postcondition defines what effects the program has. For very simple programs, these affects can be defined directly in the postcondition; for example:

```
schema main(args: seq of string,
            context!: limited Environment, ret!: out int)
   pre #args >= 1
   post context!print("Hello, world!\n") & ret! = 0;
```

More usually, you will need to maintain some state in the program. This state is best represented by the abstract variables of some class. If this class is called *Application* then the main program could look like this:

```
final class Application ^=
abstract
   var context: Environment,
       ... ;          // declare other state variables here
interface
   build{!context: Environment}
     post ... ;       // initialise other state variables here

   schema !run(args: seq of string, ret!: out int)
```

```
        post ... ;
end;

schema main(args: seq of string,
            context!: limited Environment, ret!: out int)
    pre #args >= 1
    post (var myApp: Application! = Application{context};
          myApp!run(args, ret!)
         );
```

In this example, we have assumed that the application requires prolonged access to the context, but that it only needs access to the arguments at the start ot a run. Hence we passed the context as a parameter to the constructor, which stored it as part of the state; but we passed the arguments as parameters to the *run* schema.

12.2 Program entry point not written in Perfect

If the program entry point is not written in *Perfect*, and the part of the application that is written in *Perfect* does not perform I/O or use other members of the *Environment* class, then no special initialization is needed.

If the part of the application that is written in *Perfect* does need access to an *Environment*, then it will be necessary to write code in the target programming language to create a single instance of *Environment*, which can then be passed to the *Perfect* part of the application. The *Environment* constructor takes a single argument, which is the path to the directory from which the program was launched (for subsequent use by the *getImagePath* member).

Appendix A: Library Reference

A1. Global methods

The following global methods are defined. Some or all of these may be replaced by nonmember class methods in future versions of the library.

function debugPrint(s: string): bool ^= ?;	Prints the string to standard output as an unspecified side-effect.
function debugHalt(s: string): bool ^= ?;	Prints the string to standard output as an unspecified side-effect, then halts the program (possibly loading a debugger, if available).

function flatten(s: seq of seq of class X): seq of X
 	
  ^= ([s.empty]:
     seq of X{},
    []:
     ++**over** s
  );
	Converts a sequence of sequences to a single sequence by concatenating all the components. Equivalent to using ++ **over** except that the operand is allowed to be empty.

function flatten(s: set of set of class X): set of X
 	
  require X **has operator** =(arg) **end**
  ^= ([s.empty]:
     set of X{},
    []:
     ++ **over** s
  );
	Converts a set of sets to a single set by uniting all the components. Equivalent to using ++ **over** except that the operand is allowed to be empty.

function flatten(b: bag of bag of class X): bag of X
 	
  require X **has operator** =(arg) **end**
  ^= ([b.empty]:
     bag of X{},
    []:
     ++ **over** b
  );
	Converts a bag of bags to a single bag by uniting all the components. Equivalent to using ++ **over** except that the operand is allowed to be empty.

	Similar to *flatten* but inserts the element *t* between elements of *s*. Especially useful for converting a

function interleave(s: seq of seq of class X, t: seq of X): seq of X
 	
^= ([s.empty]:
    seq of X{},
   []:
   s.head ++ flatten(for x::s.tail yield t ++ x)
)
assert #s ~= 0 & #t ~= 0 ==> #**result** ~= 0;
	sequence of strings to a single string for printing, inserting a separator (e.g. comma or newline) between the elements.

function loadObject (env: Environment, strm: from InputStream, minVersion, maxVersion: nat): (from Storable) || SerialError ^= ?;	Attempts to load an object from the stream. Returns the object retrieved if successful, otherwise the reason for the failure.
function max(a, b: class X): X ^= ([a ~~ b = **rank** below]: b, []: a);	If the two parameters rank same, returns the first one.
function max(a, b: class X, repeated c: X): X ^= max(max(a, b), c.max);	
function min(a, b: class X): X ^= ([a ~~ b = **rank** above]: b, []: a);	If the two parameters rank same, returns the first one.
function min(a, b: class X, repeated c: X): X ^= min(min(a,b), c.min);	
schema storeObject (obj: from Storable, env!: limited Environment, strm: from OutputStream, version: nat, err!: out SerialError || void) **post** ?;	Stores the specified object to the stream. The err parameter is set to **null** if successful, otherwise it holds the reason for the failure.
	Swaps the values of the two parameters. May be more efficient for some data types on some platforms than using $x! = y,\ y! = x$ directly.

schema swap(x!, y!: class X)	
post x!= y, y!= x;	

A2. Classes

All of these classes inherit from class **anything**, except for classes **anything** and **void**.

Variables are only mentioned if they are public or are referred to in the specifications.

The ancestors given in the **inherits** parts are not necessarily the direct ancestors of the classes concerned, since additional classes may be inserted in the inheritance chain for implementation reasons, or for future extension.

For enumeration classes other than **rank**, additional values may be inserted or appended in future versions of the library. The *toString* method of each enumeration class is redefined in the usual way, but not shown here.

anything	
deferred class anything	This is ancestor of all classes defined in *Perfect* (whether by the user or by the system) unless declared **public** or **external**
Methods	

function toString: **string**
  decrease ?
  ^= ?;
	Return a textual representation of the value of the object. The default returns the string "No output string specified for this type" or similar. It should be overridden in any class for which *toString* is likely to be called. Recursive definitions of *toString* are allowed.

bag of X	
final class bag of X	A bag is an unordered collection of values. Duplicate values are permitted and significant.
Constructors	

build{}
  post ?
  assert result.empty;
	Builds an empty bag

build{**repeated** x: X} **post** ?;	Builds a bag containing the values in the parameter list

Methods

operator (a: X)#: **nat**
  require X **has operator** =(arg) **end**
  ^= #(**those** x::**self** :- x = a);
	Returns the number times the given value occurs in the bag

operator (a: X) in: **bool**
  require X **has operator** =(arg) **end**
  ^= a **in** ran;
	Returns **true** if the bag contains the parameter

operator #: **nat** ^= ?;	Returns the number of elements in the bag

operator **(a: **bag of** X): **bag of** X
  require X **has operator** =(arg) **end**
  satisfy forall x:X :- x # **result** = min(x # self, x # a);
	Returns the intersection of the bag with the parameter

operator ##(a: **bag of** X): **bool**
  require X **has operator** =(arg) **end**
  ^= **forall** x::**self** :- x ~**in** a;
	Returns **true** if the bag is disjoint with the parameter

operator ++(a: **bag of** X): **bag of** X **satisfy forall** x:X :- x # **result** = x # self + x # a;	Returns the union of the bag with the parameter
operator --(a: **bag of** X): **bag of** X **require** X **has operator** =(arg) **end satisfy** **forall** x:X :- x # **result** = max(x # **self** - x # a, 0);	Returns the difference between the bag and the parameter

operator <<=(a: **bag of** X): **bool**
  require X **has operator** =(arg) **end**
  ^= **forall** x::**self** :- x # **self** <= x # a;
	Returns **true** if the bag is a sub-bag of the parameter

operator <<(a: **bag of** X):**bool**
  require X **has operator** =(arg) **end**
  ^= **self** <<= a & #**self** < #a;
	Returns **true** if the bag is a strict sub-bag of the parameter

function append(a: X): **bag of** X
  satisfy result >> **self**,
   #**result** = >#**self**,
   forall x::**result** :- x # **result** = ([x = a]: >(x # self), [x ~= a]: x # self);
	Returns a new bag like the original but with one more instance of the parameter adjoined

function empty: **bool** ^= #**self** = 0;	Returns**true** if the bag is empty
function max: X **require** X **has total operator** ~~(arg) **end**	Returns the highest element in the bag

pre ~empty ^= **that** x::**self** :- **forall** y::**self** :- (x ~~ y) ~= **rank** below;	

function min: X
  require X **has total operator** ~~(arg) **end**
  pre ~empty
  ^= **that** x::**self** :- **forall** y::**self** :- (x ~~ y) ~= **rank** above;
	Returns the lowest element in the bag

opaque function omax: X
  pre ~empty
  ^= **any** x::**self** :- **forall** y::**self** :- (x ~~ y) ~= **rank** below;
	Returns a highest element in the bag. If the element type has a total ordering, you should use *max* instead.

opaque function omin: X
  pre ~empty
  ^= **any** x::**self** :- **forall** y::**self** :- (x ~~ y) ~= **rank** above;
	Returns a lowest element in the bag. If the element type has a total ordering, you should use *min* instead.

opaque function opermndec: **seq of** X
  satisfy result .ranb = **self**,
    result.isndec;
	Returns a sequence comprising the elements of the bag in a nondecreasing order. If the element type has a total ordering, you should use *permndec* instead.

opaque function opermninc: **seq of** X
  satisfy result .ranb = **self**,
    result.isninc;
	Returns a sequence comprising the elements of the bag in a nonincreasing order. If the element type has a total ordering, you should use *permninc* instead.

function permndec: **seq of** X
  require X **has total operator** ~~(arg) **end**
  satisfy result.ranb = **self**,
    result.isndec;
	Returns the sequence comprising the elements of the bag in a nondecreasing order

function permninc: **seq of** X
  require X **has total operator** ~~(arg) **end**
  satisfy result.ranb = **self**,
    result.isninc;
	Returns the sequence comprising the elements of the bag in a nonincreasing order

function ran: **set of** X
  require X **has operator** =(arg) **end**
  satisfy forall x:X :- x **in result** <==> x **in self**;
	Converts the bag to a set by removing duplicates

function remove(a: X): **bag of** X
  require X **has operator** =(arg) **end**
  satisfy ([a **in self**]: a # **result** = <(a # **self**) & **result** .append(a) = **self**, [a **~in self**]: **result** = **self**);
	Returns a new bag like the original except that if there were one or more instances of the parameter in the bag, the result contains one fewer instance

function rep(a: **nat**): **bag of** X
  satisfy result .ran = **self**.ran,
    forall x::**self** :- x # **result** = a * (x # **self**)
	Returns a new bag in which each element of the original is replicated the number of times given by the parameter

redefine function toString: **string** ^= ?;	Returns a textual representation of the bag

function unique: **bool**
  require X **has operator** =(arg) **end**
  ^= **forall** x::**self** :- x # self = 1;
	Returns **true** if the bag contains no repeated values

bool	
final class bool	The Boolean type has values **true** and **false**.

<div align="center">

Methods

</div>

operator &(arg: **bool**)	Equivalent (except for precedence) to: ([**~self**]: **false**, []: arg)
operator | (arg: **bool**)	Equivalent (except for precedence) to: ([**self**]: **true**, []: arg)
operator ==> (arg: **bool**)	Equivalent (except for precedence) to: **~self** | arg
operator <== (arg: **bool**)	Equivalent (except for precedence) to: **self** | ~arg
operator <==> (arg: **bool**)	Equivalent (except for precedence) to: **self** = arg
redefine function toString ^= ([**self**]: "true", []: "false");	

byte	
final class byte	Represents an 8-bit byte. Used primarily for reading and writing streams and files.

<div align="center">

Constructors

</div>

build {bits: **seq of bool**} **pre** #bits = 8;	Constructs a byte from 8 bits (the most significant bit is *theBits[0]*)

build{arg: **nat**}
  pre arg < 256
  ^= **byte**
    { **seq of bool**
	Constructs the byte representing the given number.

```
      { arg >= 128,
       (arg % 128) >= 64,
       (arg % 64) >= 32,
       (arg % 32) >= 16,
       (arg % 16) >= 8,
       (arg % 8) >= 4,
       (arg % 4) >= 2,
       (arg % 2) = 1
      }
     }
   assert +result = arg;
```

Methods	
operator +: **nat** ^= ? **assert result** < 256, **byte**{**result**} = **self**;	Returns the interpretation of the byte as an unsigned integer.
total operator ~~(a) ^= +**self** ~~ +a;	
operator .. (b: **byte**): **seq of byte** ^= ([b >= **self**]: (**self ..** <b).append(b), [b < **self**]:**seq of byte** {}) **assert result**.isndec;	
operator >: **byte** **pre self** ~= **byte** {255 ^= **byte**{>+**self**};	
operator <: **byte** **pre self** ~= **byte**{0 ^= **byte**{<+**self**};	
function and(arg: **byte**): **byte** ^= **byte**{**for** i::0**..**7 **yield** theBits[i] & arg.theBits[i]};	
function or (arg: byte): byte ^= **byte**{**for** i::0**..** 7 **yield** theBits[i] | arg.theBits[i]};	
function compl: **byte** ^= **byte** {**for** i::0**..** 7 **yield** ~theBits[i]};	
function shl(arg: **nat**): **byte** **pre** arg < 8 ^= **byte**{theBits.drop(8) ++ **seq of bool**{**false** }.rep(arg)};	

function shr(arg: **nat**): **byte**
  pre arg < 8
  ^= **byte**{**seq of bool**{**false**}.rep(arg) ++ theBits.take(8-arg)};	

redefine function toString: string ^= (+**self**).toString;	

ByteData	
final class ByteData	A memory buffer that can be used for stream input/output

Data	
var bytes: **seq of byte**	The data held
var index: **nat**	The index of the next byte to be read

Invariants	
index <= #bytes	

Constructors	
build{ } **post** bytes! = **seq of byte**{};	Builds an empty ByteData
build{!bytes: **seq of byte**};	Builds a ByteData from a byte array

Methods	
function eof: **bool** ^= index = #bytes;	Returns **true** if there are no more bytes to be read

schema !get(b!: **out byte**)
  pre ~eof
  post b! = bytes[tell],
    index! + 1;
	Retrieves the next byte and advances the pointer

schema !get(sb!: **out seq of byte**, len: **nat**)
  pre index + len <= #bytes
  post sb! = bytes.slice(streamPos, len), streamPos! + len;
	Retrieves the specified number of bytes

schema !put(b: **byte**) **post** bytes! = bytes.append(b), streamPos! + 1;	

schema !put(sb: **seq of byte**) **post** bytes! ++ sb;	
function size: **nat** ^= #bytes;	

ByteInputStream

class ByteInputStream ^= **inherits** InputStream	An input stream that reads from data in memory

<table>
<tr><td colspan="2" align="center">Constructors</td></tr>
<tr><td>build {!bytes: ref ByteData on StreamHeap}
  inherits InputStream{};</td><td>Builds a ByteInputStream from a reference to a ByteData object</td></tr>
<tr><td colspan="2" align="center">Methods</td></tr>
<tr><td>define schema !close(ret!: out FileError)
  post ?;</td><td>Closes the stream.</td></tr>
<tr><td>define ghost function gStreamData: seq of byte
  ^= bytes.value.bytes;</td><td>The data held by the stream</td></tr>
<tr><td>define ghost function gStreamPtr: nat
  ^= bytes.value .tell;</td><td>The index of the next element to be read</td></tr>
<tr><td>define ghost function gStreamAtEnd: bool
  ^= ?;</td><td>Returns true if there are no more bytes to be read</td></tr>
<tr><td>define schema !read(b!: out byte, ret!: out FileError)
  post ?;</td><td>Reads a byte from the stream</td></tr>
<tr><td>schema !read(s!: out seq of byte, numBytes: nat, ret!: out FileError)
  pre isOpen,
    numBytes ~= 0
  post ?;</td><td>Inherited from class InputStream</td></tr>
<tr><td>schema !read(n!: out int, numBytes: nat, ret!: out FileError)
  pre isOpen,
    numBytes ~= 0
  post ?;</td><td>Inherited from class InputStream</td></tr>
<tr><td>schema !read(c!: out char, decoder!: limited from CharDecoder, ret!: out FileError)
  pre isOpen</td><td>Inherited from class InputStream</td></tr>
</table>

post ?;	

schema !read(r!: **out real**, ret!: **out** FileError)
  pre isOpen
  post ?;
	Inherited from class InputStream

ByteOutputStream	
final class ByteOutputStream ^= **inherits** OutputStream	An output stream that writes to an area of memory
Constructors	
build {!bytes: **ref** ByteData **on** StreamHeap}**inherits** OutputStream{};	Builds a ByteOutputStream from a reference to a ByteData object
Methods	
define schema !close(ret!: **out** FileError) **post** ?;	Closes the stream.
define schema !flush **post pass**;	
define schema !write(b: **byte**, ret!: **out** FileError) **post** ?;	Appends the byte to the memory buffer
define ghost function gStreamData: **seq of byte** ^= bytes.**value**.bytes;	The data in the memory buffer

schema !write(s: **seq of byte**, ret!: **out** FileError)
  pre isOpen
  post ?;
	Inherited from class OutputStream

schema !write(i: **int**, numBytes: **nat**, ret!: **out**
FileError)
  pre isOpen,
    numBytes ~= 0
  post ?;
	Inherited from class OutputStream

schema !write(c: **char**, encoder: **from**
CharEncoder, ret!: **out** FileError)
  pre isOpen
  post ?;
	Inherited from class OutputStream

schema !write(r: **real**, ret!: **out** FileError)
  pre isOpen
  post ?;
	Inherited from class OutputStream

char	
final class char	Represents a single character. When compiling to Java, the character set is Unicode; when compiling to C++ it may be a variant of ASCII, or (depending on code generation options chosen) Unicode.

Constructors

build{n: **nat**} **post** ? **assert** +**self**' = n;	Constructs the character whose numeric value in the supported character set is equal to *n*.

Methods

total operator ~~(a) ^= +**self** ~~ +a;	Defines a total ordering of between characters
operator .. (b: **char**) :**seq of char** ^= ([b >= **self**]: (**self ..**<b).append(b), [b < **self**]:**seq of char** {}) **assert result**.isndec;	
operator >: **char** ^= **char** {>+**self**};	
operator <: **char** ^= **char** {<+**self**};	
operator +: **nat** ^= ? **assert char**{**result** } = **self**;	Returns the numeric value of the character in the supported character set
function isLetter: **bool** ^= ? **assert result** ==> isPrintable, isDigit ==> **~result**, **self in** " !\"$%^&*()-_=+[]{};:'@#~,<.>/?\\|`" ==> ~ **result**, **self in** "abcdefghijklmnopqrstuvwxyz" ++ "ABCDEFGHIJKLMNOPQRSTUVWXYZ" ==> **result**;	Returns **true** if the character is a letter.
function isLowerCase: **bool** ^= ? **assert result** ==> isLetter, **result** ==> ~isUpperCase, isDigit ==> ~**result**,	Returns **true** if the character is a lower case letter.

self in " !\"$%^&*()-_=+[]{};:'@#~,<.>/?\\|`'" ==> ~ **result**,
  self in "ABCDEFGHIJKLMNOPQRSTUVWXYZ" ==> ~**result**,
  self in "abcdefghijklmnopqrstuvwxyz" ==> result;	

function isUpperCase: **bool** ^= ?
 assert result ==> isLetter,
  result ==> ~isLowerCase,
  isDigit ==> ~**result**,
  self in " !\"$%^&*()-_=+[]{};:'@#~,<.>/?\\|`'" ==> ~ **result**,
  self in "ABCDEFGHIJKLMNOPQRSTUVWXYZ" ==> **result**,
  self in "abcdefghijklmnopqrstuvwxyz" ==> ~**result**;
	Returns **true** if the character is an upper case letter.

function isDigit: **bool** ^= **self in** "0123456789";	Returns **true** if the character is a digit.

function isPrintable: **bool**
 ^= ?
 assert isLetter ==> **result**,
  isDigit ==> **result**,
  self in " !\"$%^&*()-_=+[]{};:'@#~,<.>/?\\|`'" ==> **result**;
	Returns true if the character is printable or a normal space (i.e. not a control character)..

function digit: **int**
 pre isDigit
 ^= ?
 assert 0 <= **result** < 10,
  self = `0` <==> **result** = 0,
  self = `1` <==> **result** = 1,
  self = `2` <==> **result** = 2,
  self = `3` <==> **result** = 3,
  self = `4` <==> **result** = 4,
  self = `5` <==> **result** = 5,
  self = `6` <==> **result** = 6,
  self = `7` <==> **result** = 7,
  self = `8` <==> **result** = 8,
  self = `9` <==> **result** = 9;
	Converts the character to the number it represents.

function toLowerCase: **char** ^= ? **assert self** .isUpperCase ==> **result**.isLowerCase,	Returns the lower case version of the character, it it is an upper case character; otherwise returns the character unchanged.

~self .isUpperCase ==> **result** = **self**;	
redefine function toString: **string** ^= **string**{**self** };	
function toUpperCase: **char** ^= ? **assert self** .isLowerCase ==> **result**.isUpperCase, **~self** .isLowerCase ==> **result** = **self**;	Returns the upper case version of the character, it it is a lower case character; otherwise returns the character unchanged.

CharDecoder	
deferred class CharDecoder ^= **inherits** CharEncoderDecoder	
Constructors	
build{ } **inherits** CharEncoderDecoder{ };	
Methods	
deferred function charReady: **bool**;	Returns **true** if the decoder has a completed character.

final ghost function decode(input: **seq of byte**):**string**
 decrease #input
 ^= (**let** temp ^= getFirstChar(input);
  [temp = null]:
   string{ },
  []:
   decode((temp **is pair of** (**char**, **seq of byte**)).y).prepend((temp **is** pair **of** (**char**, **seq of byte**)).x)
  );
	Expresses the result of decoding a sequence of bytes.

deferred function getCompletedCharacter: **char** **pre** charReady;	Retrieves the completed character.

final ghost function getFirstChar(input: **seq of byte**): **pair of** (**char**, **seq of byte**) || **void**
 decrease #input
 ^= ([#input = 0]:
   null as pair of (**char**, **seq of byte**) || **void**,
  []:
   (**let** temp ^= **self after**
	Decodes the next character from a byte string. Returns **null** if the end of the string is reached first, else the character and the rest of the input byte string.

it!process(input.head); 　　　[temp.charReady]: 　　　　**pair of** (**char**, **seq of byte**) {temp.getCompletedCharacter, input.tail}, 　　　[]: 　　　　temp.getFirstChar(input.tail) 　　　) 　　) 　**assert result** ~= **null** ==> #(**result is** pair **of** (**char**, **seq of byte**)).y < #input;	
deferred schema !process(bb: **byte**);	Consumes the byte, possibly giving rise to a completed character
final ghost schema !process(sb: **seq of byte**) **decrease** #sb **post** ([~sb.empty]: 　　　!process(sb.head) **then** !process(sb.tail), 　　[] 　　);	Expresses the effect of consuming a sequence of bytes
deferred schema !reset;	Resets the decoder, ready to decode the next encoded character. This saves having to build a new decoder every time another character is to be decoded.

CharEncoder	
deferred class CharEncoder ^= **inherits** CharEncoderDecoder	
Constructors	
build{ } 　**inherits** CharEncoderDecoder{ };	
Methods	
deferred function encode(c: **char**): **seq of byte**;	Encodes a character into a byte sequence
final ghost function encode(input: **string**): **seq of byte** **decrease** #input 　^= ([input.empty]: 　　　**seq of byte**{ }, 　　[]: 　　　encode(input.head) ++ encode(input.tail)	Expresses the byte sequence generated if an entire character string is encoded.

);	
deferred function preamble: **seq of byte**;	Returns the Byte Order Mark (if any) to generate at the start of a text file or stream.

CharEncoderDecoder	
class CharEncoderDecoder	
Data	
const encodingTable: **map of** (**string** -> pair **of** (**from** CharEncoder, **from** CharDecoder)) ^= ?;	A table of character set names and the corresponding encoders and decoders.
Constructors	
build{};	
Methods	

nonmember function getDecoder(s: **string**): **from** CharDecoder || **void**
 ^= ([isRecognisedEncoding(s)]:
     encodingTable[s].y**as from** CharDecoder || **void**,
    []:
     null
)
  assert isRecognisedEncoding(s) ==> **result** ~= **null**;
	Returns the decoder for the specified character set, or **null** if the character set name is not recognized.

nonmember function getEncoder(s: **string**): **from** CharEncoder || **void**
 ^= ([isRecognisedEncoding(s)]:
     encodingTable[s].x**as from** CharEncoder || **void**,
    []:
     null
)
  assert isRecognisedEncoding(s) ==> **result** ~= **null**;
	Returns the encoder for the specified character set, or **null** if the character set name is not recognized.

nonmember function isRecognisedEncoding(s: **string**):**bool** ^= s **in** encodingTable **assert** s **in set of string**{"ascii", "utf8", ""} ==> result;	Indicates whether the character set name is recognized. ASCII (7-bit) and UFT8 character sets are always recognized. [Note: support for UTF8 may be incomplete on platforms that use an 8-bit character set.]

Comparator of X

deferred class Comparator **of** X	Base class for constructing an object to compare two values of type X. Used as a parameter to some of the ordering and sorting methods for the built-in collection classes.

Constructors

build{};	

Methods

deferred function compare(a, b: X): **rank**;	This must be defined in descendent classes to specify the ordering required. The definition must satisfy the symmetry and transitivity properties.
final function notLessThan(a, b: X): **bool** ^= compare(a, b) ~= **rank** below;	

Properties

property (a, b: X)
  assert compare(a, b) = **rank** same <==>
compare(b, a) = **rank** same,
  compare(a, b) = **rank** below <==> compare(b, a) = **rank** above;
	These symmetry properties must be obeyed by the definition of 'compare' in a descendent class

property(a, b, c: X)
  assert compare(a, b) = **rank** same ==>
compare(a, c) = compare(b, c),
  compare(a, b) = **rank** below & compare(b, c) = **rank** below ==> compare(a, c) = **rank** below;
	These transitivity properties must be obeyed by the definition of 'compare' in a descendent class

DebugType

class DebugType ^= **enum** preConditions, postConditions, loopInvariants, loopVariants, specVariants, impVariants, embAsserts, postAsserts, lastChoices, classInvariants, constraints **end**	Enumeration for control of runtime debug checks. Each of these will only be effective when running a debug build that had the relevant check enabled at build time.

Environment

final class Environment **external**	Class to describe the interface with the operating system

Data

class File ^=
abstract
  var idata: **seq of byte**,
   iptr: **nat**;
  invariant iptr <= #idata;
interface
  ghost selector data: **seq of byte** ^= idata;
  ghost selector fileptr: **nat** ^= iptr;
  ghost function atEof: **bool** ^= fileptr = #data;
end;
 	
var openFiles: map of (FileHandle -> File),
  stdInStream: StandardInputStream,
  stdOutStream, stdErrStream: StandardOutputStream;
 	
const pathSeparator: **char** ^= ?;
const caseSensitiveFileNames: **bool** ^= ?;
function pathSeparator, caseSensitiveFileNames;	

Methods

function clock: **nat** ^= ?;	Returns the number of clock ticks since the program started. Depending on platform, this may be either CPU time consumed or elapsed time.
function clocksPerSecond: **nat** ^= ? **assert** result >= 1;	Returns the number of clock ticks per second.

schema !close(f: FileRef, ret!: **out** FileError)
pre gIsOpen(f)
post change openFiles, ret
  satisfy (ret' = FileError success ==> openFiles' = openFiles.remove(f.handle)),
  (ret' ~= FileError success) ==> openFiles' = openFiles;
	Closed the file identified by the given file reference

schema !clrRuntimeOption(debOpt: DebugType) **post** !setRuntimeOptionState(debOpt, **false**);	Disables debug checks of the specified type
	Disables debug checks of the specified types

schema !clrRuntimeOptions(debOpts: **set of** DebugType) **post** !setRuntimeOptionsState(debOpts, false);	
schema !delete(pathname: **string**, ret!: **out** FileError) **post** ?;	Deletes the file with the specified name
schema !execute(command: **string**, args: **seq of string**, stdin, stdout, stderr: FileRef || **void**, res!: **out** FileError) **post** ?;	Executes the specified command with the specified command line arguments and the specified standard input, output and error streams. [This method is not yet available in Java. In a future revision, the standard input, output and error parameters will be streams instead of file references.]
schema !fastForward(f: FileRef, ret!: **out** FileError) **pre** gIsOpen(f) **post change** openFiles[f.handle].fileptr, ret **satisfy** ret' = FileError success ==> openFiles[f.handle].fileptr' = #openFiles[f.handle].data;	Moves the file pointer to the end of the file

schema !fileSize(f: FileRef, size!: **out nat**, ret!: **out** FileError)
pre gIsOpen(f)
post change size, ret, openFiles[f.handle].fileptr
  satisfy ret' = FileError success ==> size' = #openFiles[f.handle].data,
   openFiles[f.handle].fileptr' = openFiles[f.handle].fileptr;
	Gets the size of an open file. This is a modifying schema because it may change the file pointer if it fails.

function fileStatus(pathname: **string**): FileStats || FileError ^= ?;	Returns status information about a file, including the times it was created, last modified and last accessed; its size; and its attributes, i.e. read-only, archive, system, directory (size will be 0 if this attribute is returned). Returns the FileStats object containing the information described, or a FileError describing the problem. If a path ending with the path separator character is passed, this is taken as an assertion that the path refers to a directory, and so if the path actually refers to a file instead of a directory, the error 'directoryNotFound' is returned.
function fileValid(pathname: **string**): **bool** ^= ?;	Returns **true** if the file specified is accessible on the local file system (i.e. whether an open in read mode or a file status operation would succeed).
schema !flush(f: FileRef, ret!: **out** FileError) **pre** gIsOpen(f)	Flushes any data held in buffers to the disk, console or printer concerned

post ?;	
schema !forceFileTime(filename: **string**, modified, accessed: Time || **void**, res!: **out** FileError) **post** ?;	Forces the time attributes on a file. The schema will fail if the file does not exist or is currently open.
schema !garbageCollect **post pass**;	Invokes the garbage collector of the run-time system provided by the implementation language, or attempts to reduce memory fragmentation in other ways. In a typical C++ implementation, memory held in free-lists in the *Perfect* run-time system is returned to the C++ memory manager.
function getCurrentDateTime: Time ^= ?;	Returns the system date and time.
function getCurrentDirectory: **string** || **void** ^= ?;	Returns the current default directory.
function getEnvironmentVar(envVar: **string**): **string** || **void** ^= ?;	Returns the value of the specified environment variable, or **null** if no variable of that name is defined.

function getImagePath: **string**
  ^= ?
  assert #**result** = 0 | **result**.last = pathSeparator;
	Returns the directory from which the program was loaded, if available.

function getImageVersion: **seq of int**
  ^= getImageVersion("")
  assert #**result** > 0;
	Returns version information for the main module of the current program. On Windows platforms, a sequence of four integers is returned in this order: Major version, Minor version, Revision, Build number. If no version information is available, a single zero is returned.

function getImageVersion(moduleName: **string**): **seq of int**
  ^= ?
  assert #**result** > 0;
	Returns version information for the specified module of the current program, or the main module if *moduleName* is the empty string. On Windows platforms, a sequence of four integers is returned in this order: Major version, Minor version, Revision, Build number. If no version information is available, a single zero is returned.

function getMemoryUsed: **nat** ^= ?;	Returns the total number of bytes of memory allocated to the program (up to a maximum of 2GB on a 32-bit platform).

function getOsInfo: OsInfo ^= ?;	Returns operating system information. Note that windows2000 will be detected as 'windowsNT' with the major version as 5.

ghost function gFileAtEof(f: FileRef): **bool**
  pre gIsOpen(f)
  ^= openFiles[f.handle].atEof;
	Expresses whether the file pointer is at the end of the file.

ghost selector gFileData(f: FileRef): **seq of byte**
  pre gIsOpen(f)
  ^= openFiles[f.handle].data;
	Expresses the contents of the file.

ghost function gFilePtr(f: FileRef): **nat**
  pre gIsOpen(f)
  ^= openFiles[f.handle].fileptr;
	Expresses the value of the file pointer.

ghost function gIsOpen(f: FileRef): **bool** ^= f.handle **in** openFiles.dom;	Expresses whether the specified file is open.

ghost schema !gSetFilePtr(f: FileRef, pos: **nat**)
  pre gIsOpen(f),
    pos <= gFilePtr(f)
post openFiles[f.handle].fileptr! = pos;
	Expresses the concept of setting the file pointer.

schema !makeDirectory(pathname: **string**, err!: **out** FileError) **post** ?;	Creates the directory specified by 'pathname', including all components that do not already exist.
schema !move(oldPath, newPath: **string**, overwrite: **bool**, res!: **out** FileError) **post** ?;	Moves the file specified by *oldPath* to *newPath*, overwriting any existing file if *overwrite* is true. Directories cannot be moved by this method. If the call fails, the error return *res* is set to one of the following values: • *FileNotFound* if 'oldPath' does not specify an existing file • *AttribError* if 'newPath' specifies an existing file and the *overwrite* argument was false • *DeleteError* if we failed to delete the existing target when trying to overwrite it • *FileSpecError* if *oldPath* and n*ewPath* specify the same file or if either are invalid • *OtherError* for any other problem (for example, if *oldPath* specifies a directory).

	Only the 'last-accessed' attributes on the moved file is changed; all others are preserved.

function normalizeFile(path: **string**, fileName:**string**): FilePath || **void**
  pre #path = 0 | path.last = pathSeparator
  ^= ?;
	Splits *fileName* into path and filename components, assuming that the default directory is *path*.

schema !open(filename: **string**, mode: **set of** FileModeType, f!: **out** FileRef || FileError)
  pre #(**set of** FileModeType{FileModeType read, FileModeType create, FileModeType append} ** mode) ~= 0,
  ~(**set of** FileModeType{FileModeType create, FileModeType append} <<= mode)
  post change openFiles, f
   satisfy (f' within FileRef) ==>
    (**let** fileHandle ^= (f' is FileRef).handle;
     openFiles'.dom= openFiles.dom.append(fileHandle)
     & openFiles'[fileHandle].fileptr = 0
     & (forall x::openFiles.dom :- openFiles'[x]= openFiles[x])
     & (FileModeType create **in** mode ==> #openFiles[fileHandle].data = 0)
     & ((FileModeType create **in** mode | FileModeType append **in** mode) <==> (f' is FileRef).fileWritable)
    ),
    (f' within FileError) ==> openFiles'= openFiles;
	Opens the specified file in the specified mode. The mode must include at least one of append, read or create; but may not include both append and create.

schema !open(hostname: **string**, port: **nat**, mode: SocketMode, s!: **out** Socket || SocketError) **post** ?;	Opens a socket at the specified port at the specified named host.

schema !open(ipAddress: **seq of byte**, port: **nat**, mode: SocketMode, s!: **out** Socket || SocketError)
  pre #ipAddress = 4
  post ?;
	Opens a socket on the specified port at the specified IP address.

schema !print(c: **char**)
  post change stdOutStream.charData **satisfy** ?
  assert self'.stdOutStream.isOpen = stdOutStream.isOpen,
   stdOutStream.isOpen ==> **self**'.stdOutStream.charData = stdOutStream.charData.append(c);
	Writes the character to standard output.

schema !print(s: **string**)
  post change stdOutStream.charData **satisfy** ?
  assert self'.stdOutStream.isOpen = stdOutStream.isOpen,
   stdOutStream.isOpen ==> **self**'.stdOutStream.charData
	Writes the string to standard output.

= stdOutStream.charData ++ s;	

schema !print(f: FileRef, c: **char**, ret!: **out** FileError)
  pre gIsOpen(f), f.fileWritable
  post !write(f, **seq of byte**{**byte**{+c}}, ret!);
	Writes the character to the specified file.

schema !print(f: FileRef, s: **string**, ret!: **out** FileError)
  pre gIsOpen(f), f.fileWritable
  post !write(f, **for** x::s **yield byte**{+x}, ret!);
	Writes the string to the specified file.

schema !printStdErr(s: **string**)
  post change stdErrStream.charData **satisfy** ?
  assert self'.stdErrStream.isOpen = stdErrStream.isOpen,
   stdErrStream.isOpen ==> **self'**.stdErrStream.charData =
stdErrStream.charData ++ s;
	Writes the string to the standard error stream.

schema !profile(f: FileRef, ret!: **out** FileError)
  pre gIsOpen(f), f.fileWritable
  post ?
	Stops collecting profile data and writes profiling information to the specified file. The data will be empty unless *startProfiling* has been called previously.

schema !read(f: FileRef, b!: **out byte**, ret!:**out** FileError)
  pre gIsOpen(f)
  post change openFiles[f.handle].fileptr, b, ret
  satisfy ret' = FileError success ==>
   openFiles[f.handle].fileptr < #openFiles[f.handle].data
   & b' =
openFiles[f.handle].data[openFiles[f.handle].fileptr]
   & openFiles[f.handle].fileptr' =
>openFiles[f.handle].fileptr;
	Reads a byte from the file.

schema !read(f: FileRef, s!: **out seq of byte**, len: **nat**, ret!: **out** FileError)
pre gIsOpen(f)
post change openFiles[f.handle].fileptr, s, ret
  satisfy (**let** ptr ^= openFiles[f.handle].fileptr;
   let remainingData ^= openFiles[f.handle].data.drop(ptr);
    (ret' = FileError success ==>
     len <= #remainingData
     & s' = remainingData.take(len)
     & openFiles[f.handle].fileptr' = ptr + len
)
  & (ret' = FileError success ==>
     len > #remainingData
     & s' = remainingData
     & openFiles[f.handle].fileptr' =
#openFiles[f.handle].data
)
  );
	Reads up to *len* bytes from the file.

schema !read(f: FileRef, n!: **out int**, numBytes:**nat**, ret!: **out** FileError)
  pre gIsOpen(f), numBytes > 0
  post ?;
	Reads *numBytes* bytes from the file and interprets them as an integer in big-endian format.

schema !read(f: FileRef, r!: **out real**, ret!:**out** FileError)
  pre gIsOpen(f)
  post ?;
	Reads a number of bytes (typically 8) from file and interprets them as a floating-point number (typically assuming IEEE double precision format).

schema !readLine(line!:**out string**, ret!: **out** FileError)
  post change stdInStream.charData, line, ret **satisfy** ?
  assert `\n` ~**in** line',
   line' ++ "\n" ++ self'.stdInStream.charData = stdInStream.charData
   | (line' = stdInStream.charData & self'.stdInStream.charData.empty)
   | (line' = "" & ret' ~= FileError success);
	Reads a line of text from standard input. The final line-feed or carriage-return and line-feed are discarded and not returned.

schema !readLine(f: FileRef, line!: **out string**, ret!: **out** FileError)
  pre gIsOpen(f)
  post ?;
	Reads a line of text (i.e. a data block ending in either linefeed or end-of-file) from the specified file. The final line-feed or carriage-return and line-feed are discarded and not returned.

schema !rewind(f: FileRef, ret!: **out** FileError)
  pre gIsOpen(f)
  post !seek(f, 0, ret!);
	Moves the file-pointer to the start of the file.

schema !scan(f: FileRef, s!: **out string**, len:**nat**, ret!: **out** FileError)
  pre gIsOpen(f)
  post ?;
	Attempts to read up to *len* characters from the file.

schema !scan(f: FileRef, c!: **out char**, ret!:**out** FileError)
  pre gIsOpen(f)
  post ?;
	Attempts to read a single character from the file.

schema !seek(f: FileRef, pos: **nat**, ret!: **out** FileError)
  pre gIsOpen(f)
  post change openFiles[f.handle].fileptr, ret
   satisfy ret' = FileError success ==> openFiles[f.handle].fileptr' = pos;
	Sets the file pointer to the given file position as a byte offset from the start of the file.

schema !setCurrentDirectory(path: **string**, ret!: **out** FileError) **post** ?;	Sets the current directory.
schema !setCurrentThreadPriority(priority: **int**) **pre** priority **in** 1..20	Sets the priority of the current thread, where 1 is the lowest priority and 20 is the highest.

post ?;	
schema !setMaxCheckNestLevel(level: **nat**) **post** ?;	Sets the nesting level to which debug checks are performed (i.e. when debug checks occur while evaluating debug checks).

schema !setMode(fname: **string**, atts: **set of** FileAttribute, res!:**out** FileError)
  pre atts <<= set of FileAttribute{FileAttribute read, FileAttribute write, FileAttribute execute}
  post ?;
	Sets the attributes of the file with the specified name.

schema !setRuntimeOption(debOpt: DebugType) **post** !setRuntimeOptionState(debOpt, **true**);	Enables debug checks of the specified type.
schema !setRuntimeOptions(debOpts: **set of** DebugType) **post** !setRuntimeOptionsState(debOpts, **true**);	Enables debug checks of the specified types.
schema !setRuntimeOptionState(debOpt: DebugType, state: **bool**) **post** !setRuntimeOptionsState(set of DebugType{debOpt}, **state**);	Enables (if *state* is **true**) or disables (if *state* is **false**) debug checks of the specified type.
schema !setRuntimeOptionsState(debOpt: **set of** DebugType, state: **bool**) **post** ?;	Enables (if *state* is **true**) or disables (if *state* is **false**) debug checks of the specified types.
schema !startProfiling **post** ?	Starts collecting profiling data, if the program has been built with profiling enabled; otherwise ignored.
function stdErr: StandardOutputStream ^= stdErrStream;	Returns a stream corresponding to standard error output.
function stdIn: StandardInputStream ^= stdInStream;	Returns a stream corresponding to standard input.
function stdOut: StandardOutputStream ^= stdOutStream;	Returns a stream corresponding to standard output.

function tell(f: FileRef): **nat** || FileError
  pre gIsOpen(f)
  satisfy (**result within nat**) ==> **result** = openFiles[f.handle].fileptr;
	Returns the current file position as a byte offset from the start of the file.

schema !write(f: FileRef, b: **byte**, ret!: **out** FileError)
  pre gIsOpen(f), f.fileWritable
  post !write(f, seq of byte{b}, ret!);
	Writes a single byte to the file.

schema !write(f: FileRef, s: **seq of byte**, ret!: **out** FileError)	Writes a sequence of bytes to the file.

pre gIsOpen(f), f.fileWritable
post change openFiles[f.handle].data,
openFiles[f.handle].fileptr, ret
  satisfy ret' = FileError success ==>
    (**let** ptr ^= openFiles[f.handle].fileptr;
     let oldData ^= openFiles[f.handle].data;
     openFiles[f.handle].data' = oldData.take(ptr) ++ s ++
([ptr + #s < #oldData]: oldData.drop(ptr + #s), []: seq of
byte{})
    & openFiles[f.handle].fileptr' = ptr + #s
    ),
    ret' ~= FileError success ==>
openFiles[f.handle].data'.begins(openFiles[f.handle].data);	

schema !write(f: FileRef, i, numBytes: **int**, ret!: **out**
FileError)
  pre gIsOpen(f), f.fileWritable, numBytes > 0
  post ?;
	Writes an integer of the specified size (number of bytes) to the file.

schema !write(f: FileRef, r: **real**, ret!: **out** FileError)
  pre gIsOpen(f), f.fileWritable
  post ?;
	Write a real number in binary format to a file (typically 8 bytes in IEEE double-precision format).

FileAttribute

class FileAttribute ^= **enum** read, write, execute, archive, system, hidden, directory **end**;	Enumeration class to describe the various attributes of a file.

FileError

class FileError ^= **enum** success, endOfFile, fileNotFound, directoryNotFound, fileNotOpen, diskFull, readError, writeError, flushError, seekError, deleteError, attribError, fileSpecError,	Error codes returned by the various file system methods. The value 'success' indicates no error. The *directoryNotFound* value is generated by the *filestatus* schema of class *Environment* if a path that ends with a path separator is passed and the path refers to a file instead of a directory; and by *setCurrentDirectory* if the path specified does not exist.

```
    permError,
    createError,
    closeError,
    otherError
end
```

FileHandle

class FileHandle ^= **tag**	Class used internally to refer to open files in an Environment

FileInputStream

final class FileInputStream ^= **inherits** InputStream	

<table>
<tr><td colspan="2" align="center"><h3>Constructors</h3></td></tr>
<tr><td>build {!fref: FileRef, !env: Environment}
  inherits InputStream{};</td><td></td></tr>
<tr><td colspan="2" align="center"><h3>Methods</h3></td></tr>
<tr><td>define schema !close(ret!: out FileError)
  post ?;</td><td>Closes the stream and its associated file.</td></tr>
<tr><td>define ghost function gStreamAtEnd: bool
  ^= ?;</td><td></td></tr>
<tr><td>define ghost function gStreamData: seq of byte
  ^= env.gFileData(fref);</td><td></td></tr>
<tr><td>define ghost function gStreamPtr: nat
  ^= env.gFilePtr(fref);</td><td></td></tr>
<tr><td>define schema !read(b!: out byte, ret!: out FileError)
  post ?;</td><td></td></tr>
<tr><td>schema !read(s!: out seq of byte, numBytes: nat, ret!: out FileError)
  pre isOpen,
   numBytes ~= 0
  post ?;</td><td>Inherited from class InputStream.</td></tr>
<tr><td></td><td>Inherited from class InputStream.</td></tr>
</table>

schema !read(n!: **out int**, numBytes: **nat**, ret!: **out** FileError)
  pre isOpen,
   numBytes ~= 0
  post ?;	

schema !read(c!: **out char**, decoder!: **limited from** CharDecoder, ret!: **out** FileError)
  pre isOpen
  post ?;
	Inherited from class InputStream.

schema !read(r!: **out real**, ret!: **out** FileError)
  pre isOpen
  post ?;
	Inherited from class InputStream.

FileMode

class FileMode	This is a namespace class that provides some constant definitions

<table>
<tr><td colspan="2" align="center">Data</td></tr>
<tr>
<td>const read ^= set of FileModeType{FileModeType read};
const create ^= set of FileModeType{FileModeType create};
const append ^= set of FileModeType{FileModeType append};
const readText ^= set of FileModeType{FileModeType read, FileModeType text};
const createText ^= set of FileModeType{FileModeType create, FileModeType text};
const appendText ^= set of FileModeType{FileModeType append, FileModeType text};
function read, create, append, readText, createText, appendText;</td>
<td>These constants represents commonly-used modes.</td>
</tr>
</table>

FileModeType

class FileModeType ^= **enum** read, create, append, text **end**	Enumeration describing the modes in which a file may be opened.

FileOutputStream

final class FileOutputStream ^= **inherits** OutputStream	
Constructors	
build {!fref: FileRef, !env: Environment} **inherits** OutputStream{};	
Methods	
define schema !close(ret!: **out** FileError) **post** ?;	Closes the stream and its associated file, flushing any buffered data to the file.
define schema !flush **post** ?;	Flushes any buffered data to the file.
define ghost function gStreamData: **seq of byte** ^= ?;	
define schema !write(b: **byte**, ret!: **out** FileError) **post** ?;	Writes a byte to the file.
schema !write(s: **seq of byte**, ret!: **out** FileError) **pre** isOpen **post** ?;	Inherited from class OutputStream.
schema !write(i: **int**, numBytes: **nat**, ret!: **out** FileError) **pre** isOpen, numBytes ~= 0 **post** ?;	Inherited from class OutputStream.
schema !write(c: **char**, encoder: **from** CharEncoder, ret!: **out** FileError) **pre** isOpen **post** ?;	Inherited from class OutputStream.
schema !write(r: **real**, ret!: **out** FileError) **pre** isOpen **post** ?;	Inherited from class OutputStream.

FilePath

final class FilePath	Type returned by *Environment* function *normalizeFile* to hold the split components, path and file.

Data

var pathName, fileName: **string**; **function** pathName, fileName;	

Invariants

#pathName > 0,
pathName.last = Environment pathSeparator,
Environment pathSeparator ~**in** fileName;	

Constructors

build {!pathName, !fileName: string}
  pre #pathName > 0,
    pathName.last = Environment pathSeparator,
    Environment pathSeparator ~**in** fileName;	

Methods

redefine function toString: **string**
  ^= ?
  assert result ~= "";	

FileRef

final class FileRef	Encapsulates a FileHandle object. Returned by members of class Environment that create or open files.

Methods

function fileWritable: **bool** ^= ?;	
ghost function handle: FileHandle ^= ?;	

FileStats	
final class FileStats	Hold status information for a file as returned from the environment function *fileStatus*.
Data	
var created, lastModified, lastAccessed: Time, attribs: **set of** FileAttribute, size: **nat** ; **function** created, lastModified, lastAccessed, attribs, size;	
Constructors	
build {!created, !lastModified, !lastAccessed: Time, !attribs:**set of** FileAttribute, !size: **nat**};	
Methods	
redefine function toString: **string** ^= ?;	

GuardedObject of X	
final class GuardedObject **of** X	
Data	
var guard: **bool**, **when** [guard]: object: X **end**; **function** guard, object;	
Constructors	
build{ } **post** guard! = **false**;	
build {!object: X} **post** guard! = **true**;	

InputStream

deferred class InputStream	

Constructors

build{ } **inherits** Stream{ };	

Methods

operator =(other);	

deferred schema !close(ret!: **out** FileError)
   pre isOpen
   assert ~**self**'.isOpen;
	Closes the stream, making it unavailable for subsequent input and releasing any associated resources.

deferred ghost function gStreamAtEnd: **bool**;	
deferred ghost function gStreamPtr: **nat** **assert** result <= #gStreamData;	

deferred schema !read(b!: **out byte**, ret!: **out** FileError)
   pre isOpen
   assert self'.isOpen,
     self'.gStreamData = gStreamData,
     self'.gStreamPtr = ([ret'= FileError success]: gStreamPtr + 1, []: gStreamPtr),
     ret' = FileError success ==> gStreamPtr < #gStreamData & b' = gStreamData[gStreamPtr];
	Basic schema for reading a byte from a stream.

schema !read(s!: **out seq of byte**, numBytes: **nat**, ret!: **out** FileError)
   pre isOpen,
    numBytes ~= 0
   post ?   **assert** self'.isOpen,
    self'.gStreamData = gStreamData,
   ([ret' = FileError success]:
     1 <= #s' <= numBytes
     & self'.gStreamPtr = gStreamPtr + #s'
     & s' = gStreamData.drop(gStreamPtr).take(#s'),
     []:
     s' = **seq of byte**{ }
   );	

schema !read(n!: **out int**, numBytes: **nat**, ret!: **out** FileError)
  pre isOpen,
   numBytes ~= 0
  post ?
  assert self'.isOpen, **self'**.gStreamData = gStreamData;	

schema !read(c!: **out char**, decoder!: **limited from** CharDecoder, ret!: **out** FileError)
  pre isOpen
  post ?
  assert self'.isOpen, **self'**.gStreamData = gStreamData;	

schema !read(r!: **out real**, ret!: **out** FileError)
  pre isOpen
  post ?
  assert self'.isOpen, **self'**.gStreamData = gStreamData;	

int	
final class int ^= storable	A class representing zero and the positive and negative integers
<td colspan="2" align="center">**Constructors**</td>

build {s: **string**}
  pre #s > 0,
   forall x::s :- x.isDigit
  ^= ([#s > 1]:
     10 * s.head.digit + **nat** {s.tail},
   []:
     s.head.digit
)
  assert result >= 0;
	Builds a non-negative integer from its string representation in decimal.

<td colspan="2" align="center">**Methods**</td>
total operator ~~(a) ^= ?;	The usual ordering between integers (the successor of *n* ranks above *n*)
operator +(a: **int**): **int** ^= ?;	Addition.
	Addition.

operator +(a: **real**): **real** ^= **real**{**self**} + a;	
operator -: **int** ^= ?;	Negation.
operator -(a: **int**):**int** ^= ?;	Subtraction.
operator -(a: **real**):**real** ^= **real**{**self**} - a;	Subtraction.
operator *(a: **int**): **int** ^= ?;	Multiplication.
operator *(a: **real**): **real** ^= **real**{**self**} * a;	Multiplication.

operator /(a: **int**): **int**
  pre a > 0
  ^= ?;
	Division, rounding towards minus infinity.

operator /(a: **real**): **real**
  pre a ~= 0.0
  ^= **real**{**self**} / a;
	Division.

operator %(a: **int**): **int**
  pre a > 0
  ^= ?
  assert result in 0**..**<a;
	Modulo (remainder).

operator ^(a: **nat**): **int** ^= ?;	Exponentiation.

operator ^(a: **real**): **real**
  pre a > 0.0
  ^= **real**{**self**} ^ a;
	Exponentiation.

operator >: **int** ^= **self** + 1;	Successor.
operator <:**int** ^= **self** - 1;	Predecessor.

operator .. (b: **int**): **seq of int**
  decrease b
  ^= ([b < **self**]: **seq of int**{ }, [b >= **self**]: (**self ..** <b).append(b))
  assert result.isndec;
	Returns the sequence of all integers in the given range (inclusive) in ascending order.

function abs: **nat**	Returns the absolute value.

^= ([**self** >= 0]: **self**, [**self** <= 0]: -**self**);	

function intln: **nat**
  pre self >= 0
  ^= ([**self** = 0]: 0, [**self** > 0]: **that** i::0..**self** :- 2 ^ i <= **self** & 2 ^ (i + 1) > **self**);
	Returns the logarithm to base 2, rounded down.

redefine function toString: **string**
  ^= ?
  assert result ~= "";
	Returns the integer as a string in minimum-width format, i.e. "0" or "###" or "-###" where ### is a nonempty digit string with no leading zeros.

map of (X -> Y)

final class map of (X->Y) **require** X **has operator** =(arg) **end**	This class represents a mapping from elements of type X to elements of type Y.

<table>
<tr><td colspan="2" align="center"><h3>Constructors</h3></td></tr>
<tr><td>

build{ }
  post ?
  assert self'.pairs.empty;

</td><td></td></tr>
<tr><td>

build{**repeated** a:X -> b:Y}
  pre forall i::0..(#a-2) :- **forall** j::>i**..**<#a :- a[i] = a[j] ==> b[i] = b[j]
  post ?
  assert self'.pairs = (**for** i::0**..**<#a **yield pair of** (X , Y){a[i] , b[i]}).ran,
    forall i::0**..**<#a :- **self**'[a[i]] = b[i];

</td><td></td></tr>
<tr><td>

build{p: **set of pair of** (X,Y)}
  require Y **has operator** =(arg) **end**
  pre forall x, y::p :- x = y | x.x ~= y.x
  post ?
  assert self '.pairs = p;

</td><td></td></tr>
<tr><td>

build {a: **seq of pair of** (X,Y)}
  require Y **has operator** =(arg) **end**
  pre forall x, y::a :- x = y | x.x ~= y.x
  post ?
  assert self '.pairs = a.ran;

</td><td></td></tr>
<tr><td colspan="2" align="center"><h3>Methods</h3></td></tr>
<tr><td>

operator #: **nat**
  ^= #pairs;

</td><td></td></tr>
<tr><td></td><td></td></tr>
</table>

int   128

operator ++(a: **map of** (X->Y)): **map of** (X->Y)
  require Y **has operator** =(arg) **end**
  pre forall x::dom**a.dom :- **self**[x] = a[x]
  ^= **map of** (X->Y){pairs ++ a.pairs}
  assert result.dom = dom ++ a.dom,
    forall x::dom :- **result** [x] = **self**[x],
    forall x::a.dom :- **result**[x] = a[x];	

operator --(a: **set of** X): **map of** (X->Y) ^= **map of** (X->Y){**those** x::pairs :- x.x ~**in** a};	
operator **(a: **set of** X): **map of** (X->Y) ^= **map of** (X->Y){**those** x::pairs :- x.x **in** a};	
operator ##(a: **map of** (X->Y)): **bool** ^= dom ## a.dom;	
operator ##(a: **set of** X): **bool** ^= dom ## a;	

selector [](a: X): Y
  pre a **in** dom
  ^= ?
  assert result = (**that** x::pairs :- x.x = a).y;	

operator (a: X) **in**: **bool** ^= **exists** x::pairs :- x.x = a;	

function append(a: **pair of** (X,Y)): **map of** (X->Y)
  require Y **has operator** =(arg) **end**
  pre a.x **in self** ==> a.y = **self**[a.x]
  ^= **map of** (X->Y){pairs.append(a)};	

function append(a: X -> b: Y): **map of** (X->Y)
  require Y **has operator** =(arg) **end**
  pre a **in self** ==> b = **self**[a]
  ^= **map of** (X->Y){pairs.append(pair of (X,Y){a , b})};	

function dom: **set of** X ^= **for** p::pairs **yield** p.x;	
function empty: **bool** ^= #self = 0;	

function pairs: **set of** pair **of** (X,Y)
  ^= ?
  assert forall a, b::**result** :- a.x = b.x ==> a = b;
	This returns the contents of the mapping viewed as a set of (domain element, range element) pairs.

function ran: **set of** Y **require** Y **has operator** =(arg) **end**	

^= **for** x::pairs **yield** x.y;	
function ranb: **bag of** Y ^= **for** p::pairs.rep(1) **yield** p.y;	
function remove(a: X): **map of** (X->Y) ^= **map of** (X->Y){**those** x::pairs :- x.x ~= a};	

function testIndex(a: X): GuardedObject **of** Y
 ^= ([a **in self**]:
    GuardedObject **of** Y{**self** [a]},
   []:
    GuardedObject **of** Y{}
  );	

redefine function toString: **string** ^= ? **assert result** ~= "";	

nat	
class nat ^= **int** >= 0;	The naturals comprise the non-negative integers.

OsInfo	
class OsInfo ^= **storable**	This is the type returned by *Environment* function *getOsInfo* to hold the operating system type and version information.
Data	

var type: OsType,
  osMajorVersion: **nat**,
  osMinorVersion: **nat**,
  spMajorVersion: **nat**,
  spMinorVersion: **nat**;
 function type, osMajorVersion, osMinorVersion, spMajorVersion, spMinorVersion;
	The operating system type, its major and minor version numbers and its service pack major and minor version numbers.

Constructors	
build {!type: OsType, !osMajorVersion, !osMinorVersion, !spMajorVersion, !spMinorVersion: **nat**};	Constructor to set operating system type, operating system major & minor versions and service pack major & minor versions (for Microsoft operating systems).

build{!type: OsType, !osMajorVersion, !osMinorVersion: **nat**} **post** spMajorVersion! = 0, spMinorVersion! = 0;	Constructor to set operating system type and operating system major & minor versions only.

Methods

redefine function toString: **string**
  ^= ?
  assert result ~= "";	

OsType	
class OsType ^= **enum** unknown, windows95, windows98, windowsNT, linux **end**	Note that Windows 2000 reports itself as Windows NT version 5.

OutputStream	
deferred class OutputStream	

Constructors

build{} **inherits** Stream{};	

Methods

operator =(other);	

deferred schema !close(ret!: **out** FileError)
  pre isOpen
  assert ~**self**'.isOpen;
	Closes the stream, making it unavailable for subsequent output and releasing any associated resources.

deferred schema !flush(ret!: **out** FileError)
  pre isOpen
  assert self'.isOpen, **self**'.gStreamData = gStreamData;
	Flushes any buffered data to the device concerned.

deferred schema !write(b: **byte**, ret!: **out** FileError)
  pre isOpen
  assert self'.isOpen,
    ret' = FileError success ==> **self**'.gStreamData = gStreamData.append(b);	

schema !write(s: **seq of byte**, ret!: **out** FileError) **pre** isOpen **post** ? **assert self'**.isOpen, ret' = FileError success ==> **self'**.gStreamData = gStreamData ++ s;	
schema !write(i: **int**, numBytes: **nat**, ret!: **out** FileError) **pre** isOpen, numBytes ~= 0 **post** ? **assert self'**.isOpen;	
schema !write(c: **char**, encoder: **from** CharEncoder, ret!: **out** FileError) **pre** isOpen **post** ? **assert self'**.isOpen;	
schema !write(r: **real**, ret!: **out** FileError) **pre** isOpen **post** ? **assert self'**.isOpen;	

pair of (X, Y)	
final class pair of (X, Y) ^= **storable**	
Data	
var x: X, y: Y; **selector** x, y;	
Constructors	
build {!x: X , !y: Y};	
Methods	

operator ~~(a)
 ^= (**let** t1 ^= x ~~ a.x;
   [t1 = **rank** same]:
     y ~~ a.y,
   []:
     t1
   );	

redefine function toString: **string** ^= ? **assert result** ~= "";	

rank	
class rank ^= **enum** below, same, above **end**	The class used to express the result of a comparison between two values.

real	
final class real ^= **storable**	A class representing real numbers stored to some limited precision (typically IEEE double-precision format).
Constructors	
build{a: **int**} ^= ?;	Constructs a real from the given integer.
Methods	
total operator ~~(a) ^= ?;	Higher numbers rank above lower numbers.
operator +(a: **real**): **real** ^= ?;	Addition.
operator +(a: **int**): **real** ^= **self** + **real** {a};	Addition.
operator -: **real** ^= ?;	Negation.
operator -(a: **real**): **real** ^= ?;	Subtraction.
operator -(a: **int**): **real** ^= **self** - **real** {a};	Subtraction.
operator *(a: **real**): **real** ^= ?;	Multiplication.
operator *(a: **int**): **real** ^= **self** * **real**{a};	Multiplication.

operator /(a: **real**): **real**
  pre a ~= 0.0
  ^= ?;
	Division.

operator /(a: **int**): **real**
  pre a ~= 0
  ^= **self** / **real**{a};
	Division.

operator ^(a: **real**): **real**
  pre a > 0.0
  ^= ?;
	Exponentiation.

operator ^(a: **int**): **real**
  pre self ~= 0.0 | a ~= 0
  ^= **self** ^ **real**{a};
	Exponentiation.

function abs: **real**
  ^= ([**self** >= 0.0]: **self**, [**self** <= 0.0]: -**self**)
  assert result >= 0.0;
	Absolute value.

function isInfinite: **bool** ^= ?;	Returns true if the given value represents of positive or negative infinity.
function isNaN: **bool** ^= ?;	Returns true if the given value represents a not-a-number.
function rounddn: **int** **satisfy real**{**result**} <= **self** & **real**{>**result**} > **self**;	Return integral part, rounding towards minus infinity.
function roundin: **int** ^= ([**self** >= 0.0]: rounddn, [**self** <= 0.0]: roundup);	Return integral part, rounding towards zero.
function roundout: **int** ^= ([**self** >= 0.0]: roundup, [**self** <= 0.0]: rounddn);	Return integral part, rounding away from zero.
function roundup: **int** **satisfy real**{**result**} >= **self** & **real**{<**result**} < **self**;	Return integral part, rounding towards plus infinity.

redefine function toString: **string**
  ^= ?
  assert result ~= "";
	Returns a string representation of the real number.

ReverseComparator of X

final class ReverseComparator **of** X ^= **inherits** Comparator **of** X	This comparator compares objects according to the reverse of their normal rank ordering.

Constructors

build{ } **inherits** Comparator **of** X{ };	

Methods

define function compare(a, b: X): **rank** ^= b ~~ a;	
final function notLessThan(a, b: X): **bool** ^= compare(a, b) ~= **rank** below;	Inherited from class Comparator of X.

seq of X

final class seq of X	An ordered list of elements.

Constructors

build { }
  post ?
  assert #**self**';=0;
	Builds an empty sequence.

build{**repeated** x: X} **post** ?;	Builds a sequence containing the elements in the parameter list.

Methods

total operator ~~(b)
 ^= (**let** len ^= #**self**;
   let lenb ^= #b;
  [len = 0 = lenb]:
    rank same,
  [len = 0 ~= lenb]:
    rank below,
  [len ~= 0 = lenb]:
    rank above,
  [len ~= 0 ~= lenb]:
  (**let** temp ^= **self**.head ~~ b.head;
   [temp = **rank** same]:
     self.tail ~~ b.tail,
   []:
	Ordering operator. Compares the lengths first; only compares the elements if the lengths are the same.

```
    temp
  )
);
```

operator #: **nat** ^= ?;	Returns the number of elements in **self**.

operator (elem: X) #: **nat**
  require X **has operator** =(arg) **end**
  ^= #(**those** i::dom :- elem = **self**[i]);
	Returns the number of times *elem* occurs in **self**.

operator ++(other: **seq of** X): **seq of** X
  satisfy #**result** = #**self** + #other,
    forall x::dom :-**result**[x] = **self**[x],
    forall x::other.dom :- **result**[x + #**self**] = other[x];
	Returns a new sequence comprising **self** followed by *other*.

operator (elem: X) **in**: **bool**
  require X **has operator** =(arg) **end**
  ^= **exists** i::dom :- **self** [i] = elem;
	Returns **true** if and only if **self** has the element *elem*.

operator (other: **seq of** X)<<=: **bool**
  require X **has operator** =(arg) **end**
  ^= **exists** i::0..(#**self** -#other) :- slice(i, #other) = other;
	Returns **true** if and only if *other* is a subsequence of **self**. See also the **begins** and **ends** methods.

operator (other: **seq of** X)<<: **bool**
  require X **has operator** =(arg) **end**
  ^= a <<= **self** & **self** ~= other;
	Returns **true** if and only if *other* is a strict subsequence of **self**.

selector [](index: **nat**): X
  pre index < #**self**
  ^= ?;
	Accesses the element at position *index*. The first element has index zero.

function append(elem: X): **seq of** X
  satisfy #**result** = >#**self**,
    result.front = **self**,
    result.last = elem;
	Returns a new sequence comprising **self** with *elem* appended.

function begins(other: **seq of** X): **bool**
  require X **has operator** =(arg) **end**
  ^= #other <= #**self** & **self**.take(#other) = other;
	Returns **true** if and only if *other* is a leading subsequence of **self**.

function dom: **set of nat** ^= (**for** i::0.. <#self **yield** i **is nat**).ran;	Returns the set of all the valid indices of **self**.

function drop(howMany: **nat**): **seq of** X
  pre howMany <= #**self**
  satisfy #**result** = #**self** - howMany,
    forall i::**result**.dom :- **result**[i] =**self**[i + howMany];
	Returns a copy of **self** with *howMany* leading elements removed.

function empty: **bool** ^= #**self** = 0;	Returns **true** if and only if **self** contains no elements.

function ends(other: **seq of** X): **bool**
  require X **has operator** =(arg) **end**
  ^= #other <= #**self** & **self**.drop(#self - #other) = other;
	Returns **true** if and only if *other* is a trailing subsequence of **self**.

function findFirst(token: X): **int**
  ^= ([token **in self**]:
    (**those** i::**self**.dom :- **self**[i] = token).min,
   []:
    -1
  );
	Returns the index of the first occurrence of *token* in **self**, or -1 if it does not occur.

function findFirst(arg: seq of X): **int**
  ^= ([arg <<= **self**]:
    (**those** i::0..#**self** :- **self**.drop(i).begins(arg)).min,
   []:
    -1
  );
	Returns the index of the first occurrence of *arg* in **self**, or -1 if it does not occur.

function findLast(token: X): **int**
  ^= ([token **in self**]:
    (**those** i::**self**.dom :- **self**[i] = token).max,
   []:
    -1
  );
	Returns the index of the last occurrence of *token* in **self**, or -1 if it does not occur.

function findLast(arg: seq of X): **int**
  ^= ([arg <<= **self**]:
    (**those** i::0..#**self** :- **self**.drop(i).begins(arg)).max,
   []:
    -1
  );
	Returns the index of the last occurrence of *arg* in **self**, or -1 if it does not occur.

function front: **seq of** X
  pre #**self** ~= 0
  satisfy #**result** = <#**self**,
    forall x::**result** .dom :- **result**[x] =**self**[x];
	Returns a copy of **self** with the last element removed.

selector head: X
  pre ~empty
  ^= **self**[0];
	Returns the first element of **self**.

function insert(index: **nat**, a: X): **seq of** X
  pre index <= #self
  ^= take(index).append(a)++drop(index);
	Returns a new sequence comprising the original with the given element inserted before the element at *index*.

function insert(index: **nat**, a: **seq of** X):**seq of** X
  pre index <= #**self**
  ^= take(index)++ a ++drop(index);
	Returns a new sequence comprising the original with the given sequence inserted before the element at *index*.

function insertndec(x: X): **seq of** X
  pre isndec
  ^= orderedInsert(x, SimpleComparator **of** X{})
  assert result.isndec;
	Returns a new sequence comprising the original nondecreasing sequence with the given element inserted at the latest possible position that preserves the ordering.

function insertninc(x: X): **seq of** X
  pre isninc
  ^= orderedInsert(x, ReverseComparator **of** X{})
  assert result.isninc;
	Returns a new sequence comprising the original nonincreasing sequence with the given element inserted at the latest possible position that preserves the ordering.

function isndec: **bool** ^= #**self** <= 1 | (**forall** i::1**..**<#**self** :- (**self**[i] ~~ **self**[<i]) ~= **rank** below);	Returns **true** if **self** is nondecreasing according to the standard ordering relation on its elements.
function isninc: **bool** ^= #**self** <= 1 | (**forall** i::1**..**<#**self** :- (**self**[i] ~~ **self**[<i]) ~= rank above);	Returns **true** if **self** is nonincreasing according to the standard ordering relation on its elements.
function isOrdered(cp: **from** Comparator **of** X): **bool** ^= **forall** i::1..<#**self** :- cp.notLessThan(self[i],**self**[<i]);	Returns **true** if **self** is nondecreasing according to the ordering defined by *cp* .

selector last: X
  pre ~empty
  ^= **self**[<#**self**];
	Returns the last element of **self**.

function max: X
  pre ~empty
  ^= **self**[**that** x::0..<#**self** :- (**forall** y::0..<#**self** :- self [y] ~~ **self**[x] ~= **rank** above) & (**forall** y::0..<x :- **self**[y] ~~ **self**[x] ~= **rank** same)];
	Returns the highest value from **self**. If the element type does not have a total ordering and there are several different elements in **self** that rank same with each other but above all other elements, returns the earliest one.

function mergendec(other: **seq of** X): **seq of** X
  pre isndec, other.isndec
  ^= orderedMerge(other, SimpleComparator **of** X{})
  assert result.isndec;
	Merges *other* into **self**. Elements of *other* appear in the result after elements of **self** with which they rank same.

function mergeninc(other: **seq of** X): **seq of** X
  pre isninc, other.isninc
  ^= orderedMerge(other, ReverseComparator **of** X{})
  assert result.isninc;
	Merges *other* into **self**. Elements of *other* appear in the result after elements of **self** with which they rank same.

function min: X
  pre ~empty
  ^= **self**[**that** x::0..<#**self** :- (**forall** y::0..<#**self** :- self [y] ~~ self[x] ~= **rank** below) & (**forall** y::0..<x :- **self**[y] ~~ self [x] ~= **rank** same)];
	Returns the lowest value from **self**. If the element type does not have a total ordering and there are several different elements in **self** that rank same with each other but below all other elements, returns the earliest one.

opaque function opermndec: **seq of** X
  satisfy result .ranb = ranb,
    result.isndec;
	Returns a copy of the sequence sorted into a nondecreasing order. If the element type has a total ordering, you should use *permndec* instead.

opaque function opermninc: **seq of** X
  satisfy result .ranb = ranb,
    result.isninc;
	Returns a copy of the sequence sorted into a nonincreasing order. If the element type has a total ordering, you should use *permninc* instead.

function orderedInsert(elem: X, cp: **from** Comparator **of** X): **seq of** X
  pre isOrdered(cp)
  satisfy result.isOrdered(cp) & **result** .ranb =**self**.ranb.append(x);
	Returns a new sequence comprising the original ordered sequence with *elem* inserted at the latest possible position that preserves the ordering.

function orderedMerge(other: **seq of** X, cp: **from** Comparator **of** X): **seq of** X
  pre isOrdered(cp), other.isOrdered(cp)
  satisfy result .isOrdered(cp) &**result**.ranb = **self**.ranb ++ other.ranb;
	Merges *other* into **self**. Elements of *other* appear in the result after elements of **self** with which they rank same.

opaque schema !osort(cp: **from** Comparator **of** X)
  post change self satisfy self'.isOrdered(cp),
    self'.ranb = ranb;
	Sorts the sequence into a nondecreasing order according to the ordering defined by *cp*. If *cp* provides a total ordering on the element type, you should use *sort* instead.

function permndec: **seq of** X
  require X **has total operator** ~~(arg) **end**
  satisfy result.ranb = ranb,
    result.isndec;
	Returns a copy of **self** sorted into nondecreasing order.

function permninc: **seq of** X
  require X **has total operator** ~~(arg) **end**
  satisfy result.ranb = ranb,
    result.isninc;
	Returns a copy of **self** sorted into nonincreasing order.

function prepend(a: X): **seq of** X
  satisfy #result = >#**self**,
    result.head = a,
    result.tail = **self**;
	Returns a new sequence comprising the original with the parameter prepended.

function ran: **set of** X
  require X **has operator** =(arg) **end**
  satisfy forall x:X :- (x **in** result) = (x **in** self);
	Creates a set from the elements by removing duplicates and ordering.

function ranb: **bag of** X
  require X **has operator** =(arg) **end**
  satisfy #result = #**self**,
    forall x::**result** :- x # **result** = x #**self**;
	Creates a bag from the elements by removing the ordering.

function remove(index: **nat**): **seq of** X
  pre index < #self
  ^= take(index) ++ drop(>index);
	Creates a copy of **self** with the element at the given position removed.

function remove(index: **nat**, length: **nat**):**seq of** X
  pre length+index <= #**self**
  satisfy result = take(index) ++ drop(index + length);
	Creates a copy of **self** with *length* elements starting at position *index* removed.

function rep(n: **nat**): **seq of** X
  satisfy #**result** = #**self** * n,
    forall j::0**..** <n, i::dom :- **result**[i + (j * #**self**)] = **self**[i];
	Produces a new sequence by concatenating **self** *n* times.

function rev: **seq of** X
  satisfy #**result** = #**self**,
    forall x::dom :- **result**[#**self**-x-1] =**self**[x];
	Produces the sequence with the elements in reverse order.

function slice(index, length: **nat**): **seq of** X
  pre index+length <= #**self**
  ^= drop(index).take(length);
	Returns *length* elements starting at offset *index*.

schema !sort(cp: **from** Comparator **of** X)
  pre forall x,y: X :- cp.compare(x, y) = **rank** same ==> x = y
  post change self satisfy self'.isOrdered(cp),
    self'.ranb = ranb;
	Sorts the elements into nondecreasing order according to the ordering defined by *cp*.

function split(token: X): **seq of seq of** X
  require X **has operator** =(arg) **end**
  ^= (**let** bounded ^= append(token).prepend(token);
    let token_indices ^= **those** i::0..<#bounded :- bounded[i] = token;
      for i::0**..**(#token_indices-2) **yield** bounded.take(token_indices[i+1]).drop(token_indices[i]+1)
)
  assert #**result** > 0;
	Splits the sequence into a sequence of sequences, using the specified token element to determine the split points.

function tail: **seq of** X
  pre #**self** ~= 0
  satisfy #**result** = <#**self**,
    forall x::**result** .dom :- **result**[x] =**self**[>x];
	Returns a new sequence comprising **self** with the first element removed.

function take(n: **nat**): **seq of** X
  pre n <= #**self**
  satisfy #**result** = n,
    forall i::0**..**<n :- **result**[i] = **self**[i];
	Returns a new sequence comprising the first *n* elements of **self**.

redefine function toString: **string**
  ^= ?
  assert result ~= "";
	Returns a textual representation of **self**.

function unique: **bool**
  require X **has operator** =(arg) **end**
  ^= **forall** x::dom :- ~(**exists** y::>x..<#**self** :- **self**[y] = **self**[x])
  assert result <==> #**self** = #(**self**.ran);
	Returns **true** if no value occurs more than once in **self**.

SerialError

final class SerialError	

<table>
<tr><td colspan="2" align="center">Data</td></tr>
<tr><td>var id: SerialErrorType,
  msg: string;
function id, msg;</td><td></td></tr>
<tr><td colspan="2" align="center">Constructors</td></tr>
<tr><td>build {!id: SerialErrorType, !msg: string};</td><td></td></tr>
</table>

SerialErrorType

class SerialErrorType ^=
enum
  // General system errors ...
  writeError, internalError, readError,
  // Storage-stream system errors ...
  systemVersionError, userVersionError,
streamError, missingInstantiation,
  missingHeap, missingType, unexpectedType,
corruptStream,
  // Catchall system error
  unspecifiedError
end;	

set of X

final class set of X **require** X **has operator** =(arg) **end**	An unordered collection of elements in which duplicates are not permitted

<table>
<tr><td colspan="2" align="center">Constructors</td></tr>
</table>

build { }
   post ?
   assert self'.empty;
	Builds an empty set

build{**repeated** x: X} **post** ?;	Builds a set containing the parameters (any duplicates are removed)

Methods

operator #: **nat** ^= ?;	Cardinality of **self**.
operator ##(a: **set of** X): **bool** ^= **forall** x::**self** :- x ~**in** a;	Disjointness operator.
operator ++(a: **set of** X): **set of** X **satisfy forall** x:X :- x **in result** <==> x **in self** | x **in** a;	Union.
operator --(a: **set of** X): **set of** X **satisfy forall** x:X :- x **in result** <==> x **in self** & x ~**in** a;	Difference.
operator **(a: **set of** X): **set of** X ^= **those** x::**self** :- x **in** a;	Intersection.
operator <<=(a: **set of** X): **bool** ^= **forall** x::**self** :- x **in** a;	Subset.
operator <<(a: **set of** X): **bool** ^= **self** <<= a & #**self** < #a;	Strict subset.
operator (a: X) **in**: **bool** ^= ?;	Membership.

function append(a: X): **set of** X
   satisfy self <<= **result**,
    a **in result**,
    forall x::**result** :- x = a | x **in self**;
	Adds an element to **self**, returning a new set.

function empty: **bool** ^= #**self** = 0;	

function max: X
   require X **has total operator** ~~(arg) **end**
   pre ~empty
   ^= **that** x::**self** :- **forall** y::**self** :- (x ~~ y) ~= **rank** below;
	Return the maximum element from **self**.

function min: X **require** X **has total operator** ~~(arg) **end**	Return the minimum element from **self**.

pre ~empty ^= **that** x::**self** :- **forall** y::**self** :- (x ~~ y) ~= **rank** above;	

opaque function omax: X
  pre ~empty
  ^= **any** x::**self** :- **forall** y::**self** :- (x ~~ y) ~= **rank** below;
	Return a maximum element from **self**. If the element type has a total ordering, you should use *max* instead.

opaque function omin: X
  pre ~empty
  ^= **any** x::**self** :- **forall** y::**self** :- (x ~~ y) ~= **rank** above;
	Return a minimum element from **self**. If the element type has a total ordering, you should use *min* instead.

opaque function opermndec: **seq of** X
  satisfy result .ranb = **self**.rep(1),
    result.isndec;
	Returns a sequence comprising the elements of **self** in nondecreasing order. If the element type has a total ordering, you should use *permndec* instead.

opaque function opermninc: **seq of** X
  satisfy result .ranb = **self**.rep(1),
    result.isninc;
	Returns a sequence comprising the elements of **self** in nonincreasing order. If the element type has a total ordering, you should use *permninc* instead.

function permndec: **seq of** X
  require X **has total operator** ~~(arg) **end**
  satisfy result.ranb =**self**.rep(1),
    result.isndec;
	Returns the sequence comprising the elements of **self** in nondecreasing order

function permninc: **seq of** X
  require X **has total operator** ~~(arg) **end**
  satisfy result.ranb =**self**.rep(1),
    result.isninc;
	Returns the sequence comprising the elements of **self** in nonincreasing order

function remove(a: X): **set of** X
  satisfy result <<= **self**,
  a ~**in result**,
  self = **result** | **self** = **result**.append(a);
	Removes an element to **self**, returning a new set

function rep(a: **nat**): **bag of** X
  satisfy result .ran = **self**,
    forall x::**self** :- x # **result** = a;
	Returns a bag comprising each element of **self** repeated *a* times

redefine function toString: **string**
  ^= ?
  assert result ~= "";	

SimpleComparator of X

final class SimpleComparator **of** X ^= **inherits** Comparator **of** X	This comparator compares objects using their normal rank ordering.

Constructors

build{ } **inherits** Comparator **of** X{ };	

Methods

define function compare(a, b: X): **rank** ^= a ~~ b;	
final function notLessThan(a, b: X): **bool** ^= compare(a, b) ~= **rank** below;	Inherited from class Comparator of X

Socket

final class Socket	Represents a socket for communicating over a network. [A future version of the library is likely to support sockets as streams instead.]

Methods

schema !awaitConnection(res!: **out bool**) **post** ?;	Wait for a connection, blocking until one arrives (for server mode sockets only).
schema !closeSocket(res!: **out bool**) **post** ?;	
function getLastError: SocketError ^= ?;	Returns the last network error recorded.

function getRemoteAddress: **seq of nat**
  pre gIsServerSocket
  ^= ?
  assert #**result** = 4;	

function getRemotePort: **nat**
  pre gIsServerSocket
  ^= ?;	

ghost function gIsServerSocket: **bool** ^= ?;	
schema !read(res!: **out byte** || SocketError)	Reads a single byte from the socket.

post ?;	
schema !read(numBytes: **nat**, res!: **out** (**seq of byte**) ‖ SocketError) **pre** numBytes > 0 **post** ?;	Reads the specified number of 8-bit bytes from the socket specified. Blocks until all bytes read or an error is encountered. Returns the sequence of bytes read, in the order read, or an appropriate socket error.
schema !read(rdata!: **out seq of byte**, res!: **out bool**) **post** ? **assert** #rdata' > 0;	Read currently available data from the socket as a sequence of bytes.
schema !read_noblock(rdata!: **out seq of byte**, res!: **out bool**) **post** ?;	Read data if any is available but don't block if not, just return an empty sequence.
schema !write(data: **byte**, res!: **out** SocketError) **post** ?;	
schema !write(data: **seq of byte**, res!: **out** SocketError) **pre** #data > 0 **post** ?;	

SocketError	
class SocketError ^= **enum** success, unknownHost, ioError, generalError, invalidAddress, initError, invalidSocket, wrongType, connectionNotOpen **end**	

SocketMode	
class SocketMode ^= **enum** client, server **end**	

StandardInputStream	
final class StandardInputStream ^= **inherits** InputStream	Class returned by method *stdIn* of class *Environment*.
Data	

var charData: **seq of char**; **ghost function** charData;	
Methods	
define schema !close(ret!: **out** FileError) **post** ?;	Closes the stream.
define ghost function gStreamAtEnd: **bool** ^= ?;	
define ghost function gStreamData: **seq of byte** ^= env.gFileData(fref);	
define ghost function gStreamPtr: **nat** ^= env.gFilePtr(fref);	
define schema !read(b!: **out byte**, ret!: **out** FileError) **post** ?;	Reads a byte from the stream.

schema !read(s!: **out seq of byte**, numBytes: **nat**,
ret!: **out** FileError)
  pre isOpen,
    numBytes ~= 0
  post ?;
	Inherited from class InputStream.

schema !read(n!: **out int**, numBytes: **nat**, ret!: **out**
FileError)
  pre isOpen,
    numBytes ~= 0
  post ?;
	Inherited from class InputStream.

schema !read(c!: **out char**, decoder!: **limited from**
CharDecoder, ret!: **out** FileError)
  pre isOpen
  post ?;
	Inherited from class InputStream.

schema !read(r!: **out real**, ret!: **out** FileError)
  pre isOpen
  post ?;
	Inherited from class InputStream.

StandardOutputStream	
final class StandardOutputStream ^= **inherits** OutputStream	Class returned by methods *stdOut* and *stdErr* of class *Environment*.
Data	

var charData: **seq of char**; **ghost function** charData;	

Methods	
define schema !close(ret!: **out** FileError) **post** ?;	Closes the stream and its associated file, flushing any buffered data to the device or file.
define schema !flush **post** ?;	Flushes any buffered data to the device or file.
define ghost function gStreamData: **seq of byte** ^= ?;	
define schema !write(b: **byte**, ret!: **out** FileError) **post** ?;	Writes a byte to the file or device.

schema !write(s: **seq of byte**, ret!: **out** FileError)
  pre isOpen
  post ?;
	Inherited from class OutputStream.

schema !write(i: **int**, numBytes: **nat**, ret!: **out** FileError)
  pre isOpen,
    numBytes ~= 0
  post ?;
	Inherited from class OutputStream.

schema !write(c: **char**, encoder: **from** CharEncoder, ret!: **out** FileError)
  pre isOpen
  post ?;
	Inherited from class OutputStream.

schema !write(r: **real**, ret!: **out** FileError)
  pre isOpen
  post ?;
	Inherited from class OutputStream.

Storable	
deferred class Storable	This is the implicit ancestor of any class declared **storable**.

Methods	
final schema store(env!: **limited** Environment, fref: **from** OutputStream, version: **nat**) **post** ?;	Stores the object to the stream.

string	
class string ^= seq of char	

Time	
final class Time ^= **storable**	Represents a date and time.
Data	

var seconds: **nat in** 0..59,
  minutes: **nat in** 0..59,
  hours: **nat in** 0..23,
  day: **nat in** 0..6,
  date: **nat in** 1..31,
  month: **nat in** 1..12,
  year: **nat**;
function seconds, minutes, hours, day, date, month, year;
	Day 0 is Sunday, month 1 is January, and the year is AD.

Constructors	

build {!seconds, !minutes, !hours, !day, !date, !month, !year: **nat**}
  pre seconds **in** 0..59,
    minutes **in** 0..59,
    hours **in** 0..23,
    day **in** 0..6,
    date **in** 1..31,
    month **in** 1..12;	

Methods	

total operator ~~(rhs)
  ^= (**let** yearRank ^= year ~~ rhs.year;
    [yearRank = **rank** same]:
      (let monthRank ^= month ~~ rhs.month;
      [monthRank = **rank** same]:
        (let dateRank ^= date ~~ rhs.date;
        [dateRank = **rank** same]:
          (let dayRank ^= day ~~ rhs.day;
          [dayRank = **rank** same]:
            (let hourRank ^= hours ~~ rhs.hours;
              [hourRank = **rank** same]:
	Later dates/times rank above earlier ones.

<table>
<tr><td>

```
         ( let minuteRank ^= minutes ~~
rhs.minutes;

            [minuteRank = rank same]:
              seconds ~~ rhs.seconds,
            []:
              minuteRank
           ),
          []:
            hourRank
          ),
         []:
           dayRank
         ),
        []:
          dateRank
       ),
     []:
       monthRank
     ),
   []:
     yearRank
 );
```

</td><td></td></tr>
<tr><td>

redefine function toString: **string**
 ^= ?
assert result ~= "";

</td><td>

This is a simple definition of *toString* for diagnostic purposes. In an application, you will need to define a text representation that takes account of the local conventions for printing days, dates and times.

</td></tr>
</table>

triple of (X, Y, Z)

final class triple **of** (X, Y, Z) ^= **storable**	

Data

var x: X, y: Y, z: Z; **selector** x, y, z;	

Constructors

build {!x: X, !y: Y, !z: Z};	

Methods

operator ~~(a)
 ^= (**let** t1 ^= x ~~ a.x;
   [t1 = **rank** same]:
    (**let** t2 ^= y ~~ a.y;	

[t2 = **rank** same]:
     z ~~ a.z,
   []:
    t2
  ),
  []:
   t1
 );	

redefine function toString: **string** ^= ? **assert result** ~= "";	

void	
final class void ^= **storable**	The void type has a single value denoted by the literal **null**. It has no constructors.
Methods	
redefine function toString: **string** ^= "null";	

Appendix B: LALR (1) Grammar

B1. Introduction

In the following grammar, keywords are display in **bold**. Other terminal symbols have UPPERCASE names and the following meanings:

ARROW	-> or <- or <->
DBLCOLON	**::**
DEFAS	^=
DEFINEorREDEFINE	**define** or **redefine**
DOTDOTDOT	**...**
EMPTYSTRINGLITERAL	The empty string literal, i.e. ""
IDENTIFIER	An identifier
ITorSELF	**it** or **self**
LINKAGE	**extern** or **public**
LITERAL	A character, integer or real literal token, or one of **false null true**
NONEMPTYSTRINGLITERAL	Any string literal except the empty string literal
NOT	~
OP0	==> or <== or <==>
OP3	<= or <<= or >= or >>= or << or >>
OP3M	< or >
OP4	++ or --
OP4M	+ or -
OP5	** or %% or ##
OP5M	* or % or / or #
OP6	**..**
OP6M	^
OPAND	&
OPEQUAL	=
OPOR		

OPPROPERTY	**associative**, **commutative** or **idempotent**
OPRANK	~~
PREDEFCLASS	**bag**, **map**, **seq** or **set**
PREDEFTYPE	**bool**, **byte**, **char**, **int**, **rank**, **real** or **void**
SUCHTHAT	:-
THATorANY	**that** or **any**
TYPEOP	**lowest** or **highest**
UNITE	||

The symbol 'Empty' means the empty string and the symbol 'EndOfFile' means the end of the source text. Comments are italicised.

B2. Grammar

Goal:
    OptImportList GeneralDeclarations OptSEMI EndOfFile.

---------------------- *Import lists* ----------------------

OptImportList:
    Empty;
    OptImportList ImportItemList ';'.

ImportItemList:
    import ImportName;
    ImportItemList ',' ImportName.

ImportName:
    NONEMPTYSTRINGLITERAL.

----------------- *Declarations and declaration lists* ----------------

*- The following declarations can all be occur after **nonmember** or in a global or local declaration list*
FunctionEtcDeclaration:
    FunctionDeclaration;
    SchemaDeclaration.

- The following can all occur as class members
MemberFunctionEtcDeclaration:
    early PlainMemberFunctionEtcDeclaration;
    final PlainMemberFunctionEtcDeclaration;
    PlainMemberFunctionEtcDeclaration.

PlainMemberFunctionEtcDeclaration:
   FunctionDeclaration;
   SelectorDeclaration;
   MemberOperatorDeclaration;
   SchemaDeclaration;
   ModifyingSchemaDeclaration.

TheoremOrAxiomDeclaration:
   TheoremDeclaration;
   AxiomDeclaration.

NonmemberFunctionEtcDeclaration:
   nonmember FunctionEtcDeclaration.

NonmemberTheoremOrAxiomDeclaration:
   nonmember TheoremOrAxiomDeclaration.

ModifiedMemberFunctionEtcDeclaration:
   DEFINEorREDEFINE MemberFunctionEtcDeclaration;
   MemberFunctionEtcDeclaration.

- Deferred function and schema declarations are allowed together (i.e. only in class non-internal member sections)
DeferredDeclaration:
   deferred FunctionHeader OptRequire OptPrecondition OptRecursionVariant OptPostAssertion;
   deferred SelectorHeader OptRequire OptPrecondition OptRecursionVariant OptPostAssertion;
   deferred MemberOperatorHeader ':' TypeExpression OptRequire OpProperties OptPrecondition
OptRecursionVariant OptPostAssertion;
   deferred SchemaHeader OptRequire OptPrecondition OptRecursionVariant OptPostAssertion;
   deferred ModifyingSchemaHeader OptRequire OptPrecondition OptRecursionVariant OptPostAssertion.

MemberFSOSPrototype:
   FunctionPrototype;
   SelectorPrototype;
   OperatorPrototype;
   SchemaPrototype.

AbsurdDeclaration:
   AbsurdFunctionDeclaration;
   AbsurdSelectorDeclaration;
   AbsurdOperatorDeclaration;
   AbsurdSchemaDeclaration.

- A general declaration may occur at the global level of the program
GeneralDeclaration:
   ConstantDeclarations;
   FunctionEtcDeclaration;
   TheoremOrAxiomDeclaration;

    ClassOrTypeDeclaration;
    HeapDeclarations.

- A local declaration may not include constant declarations. We treat variable and let-declarations separately.
LocalDeclaration:
    FunctionEtcDeclaration;
    TheoremOrAxiomDeclaration;
    ClassOrTypeDeclaration.

*- Now for an abstract member declaration. We allow **nonmember** declarations*
AbstractMemberDeclaration:
    ConstantDeclarations;
    AbstractVariableDeclarations;
    MemberFunctionEtcDeclaration;
    TheoremOrAxiomDeclaration;
    ModifyingTheoremDeclaration;
    NonmemberFunctionEtcDeclaration;
    NonmemberTheoremOrAxiomDeclaration;
    ConstructorDeclaration;
    ClassOrTypeDeclaration;
    HeapDeclarations;
    ClassInvariant;
    HistoryInvariant.

HeapDeclarations:
    heap IdentifierOptPragmaList.

- Internal members may be anything which may be a general declaration, or a redeclaration of an abstract member
InternalMemberDeclaration:
    ConstantDeclarations;
    VariableDeclarations;
    MemberFunctionEtcDeclaration;
    TheoremOrAxiomDeclaration;
    ModifyingTheoremDeclaration;
    NonmemberFunctionEtcDeclaration;
    NonmemberTheoremOrAxiomDeclaration;
    ConstructorDeclaration;
    ClassOrTypeDeclaration;
    HeapDeclarations;
    ClassInvariant;
    InternalRedeclaration.

InternalRedeclaration:
    - Redeclarations of abstract variables as internal functions
    function SimpIdentifier DEFAS Expression OptImplementation;
    - Reimplementations of other abstract members

   function SimpIdentifier OptParmList Implementation;
   MemberOperatorHeader2 Implementation;
   EqualityOperatorHeader Implementation;
   selector SimpIdentifier OptParmList Implementation;
   selector IndexOp OptParmList Implementation;
   schema SimpIdentifier OptSchemaParmList Implementation;
   schema '!' SimpIdentifier OptSchemaParmList Implementation;
   schema '!' RedefinableOp OperatorParam Implementation;
   build '{' OptConstructorParams '}' Implementation.

ClassParameterList:
   SimpIdentifier;
   ClassParameterList Separator SimpIdentifier.

TwoOrMoreSimpIdentifiers:
   SimpIdentifier ',' SimpIdentifier;
   TwoOrMoreSimpIdentifiers ',' SimpIdentifier.

SimpIdentifierList:
   SimpIdentifier;
   TwoOrMoreSimpIdentifiers.

- Interface members may not be variables, constants, classes, types or templates but may be redeclarations of abstract members
InterfaceOrConfinedDeclaration:
   ModifiedMemberFunctionEtcDeclaration;
   TheoremOrAxiomDeclaration;
   ModifyingTheoremDeclaration;
   NonmemberFunctionEtcDeclaration;
   NonmemberTheoremOrAxiomDeclaration;
   ConstructorDeclaration;
   DeferredDeclaration;
   AbsurdDeclaration;
   InterfaceRedeclarations.

InterfaceRedeclarations:
   function SimpIdentifierList;
   ghost function SimpIdentifierList;
   selector SimpIdentifierList;
   ghost selector SimpIdentifierList.

---------------------- *Declaration lists* ------------------------

GeneralDeclarations:
   GeneralDeclaration;
   GeneralDeclarations ';' GeneralDeclaration.

---------------------- *Forms of declaration* ----------------------

ConstantDeclarations:
  const ConstDeclList;
  ghost const ConstDeclList.

ConstDeclList:
  ConstDeclItem;
  ConstDeclList ',' ConstDeclItem.

ConstDeclItem:
  IdentifierOptPragma ':' TypeExpression DEFAS Expression OptNvImplementation;
  IdentifierOptPragma DEFAS Expression OptNvImplementation.

VariableDeclarations:
  var DataDeclarationList.

AbstractVariableDeclarations:
  VariableDeclarations;
  ghost VariableDeclarations.

DataDeclarationList:
  DataDeclarations;
  DataDeclarationList ',' DataDeclarations.

DataDeclarations:
  TwoOrMoreIdentifiersOptPragma ':' PossConstrainedTypeExpression;
  IdentifierOptPragma ':' PossConstrainedTypeExpression;
  IdentifierOptPragma ':' AbbrevTypeExpr;
  when GuardedDataDeclarationList **end**.

TwoOrMoreIdentifiersOptPragma:
  IdentifierOptPragma ',' IdentifierOptPragma;
  TwoOrMoreIdentifiersOptPragma ',' IdentifierOptPragma.

GuardedDataDeclarationList:
  '[' Expression ']' ':' DataDeclarationList;
  '[' Expression ']' ':' DataDeclarationList ',' GuardedDataDeclarationList.

- Function and schema declarations

FunctionDeclaration:
  FunctionHeader OptExceptionSignature OptRequire OptPrecondition OptRecursionVariant FunctionBody.

FunctionHeader:
  FunctionHeader2;
  opaque FunctionHeader2;
  ghost FunctionHeader2.

FunctionHeader2:
   function SimpIdentifier OptParmList FunctionType;
   function IdWithPragma OptParmList FunctionType.

FunctionBody:
   DEFAS PossiblyMultipleExpression OptImplementation OptPostAssertion;
   satisfy ListOfPredicates OptImplementation OptPostAssertion.

FunctionPrototype:
   FunctionHeader OptExceptionSignature OptRequire OptPrecondition OptPostAssertion.

AbsurdFunctionDeclaration:
   absurd FunctionHeader.

SelectorDeclaration:
   SelectorHeader OptExceptionSignature OptRequire OptPrecondition OptRecursionVariant DEFAS
Expression OptImplementation OptPostAssertion.

SelectorHeader:
   SelectorHeader2;
   opaque SelectorHeader2;
   ghost SelectorHeader2.

SelectorHeader2:
   selector SimpIdentifier OptParmList ':' TypeExpression;
   selector SimpIdentifier OptParmList ':' **limited** TypeExpression;
   selector IdWithPragma OptParmList ':' TypeExpression;
   selector IdWithPragma OptParmList ':' **limited** TypeExpression;
   selector IndexOp OptParmList ':' TypeExpression;
   selector IndexOp OptParmList ':' **limited** TypeExpression;
   selector IndexOpWithPragma OptParmList ':' TypeExpression;
   selector IndexOpWithPragma OptParmList ':' **limited** TypeExpression.

SelectorPrototype:
   SelectorHeader OptExceptionSignature OptRequire OptPrecondition OptPostAssertion.

AbsurdSelectorDeclaration:
   absurd SelectorHeader.

MemberOperatorDeclaration:
   MemberOperatorHeader ':' TypeExpression OptExceptionSignature OptRequire OpProperties
OptPrecondition OptRecursionVariant OperatorBody.

EqualityOrRankDeclaration:
   GhostEqualityOperatorHeader;
   EqualityOperatorHeader;
   RankOperatorHeader OptRecursionVariant OperatorBody.

OperatorBody:
  DEFAS Expression OptImplementation OptPostAssertion;
  satisfy ListOfPredicates OptImplementation OptPostAssertion.

OperatorPrototype:
  MemberOperatorHeader ':' TypeExpression OptExceptionSignature OptRequire OptPrecondition
OptPostAssertion.

EqualityOrRankPrototype:
  EqualityOperatorHeader;
  RankOperatorHeader.

AbsurdOperatorDeclaration:
  absurd MemberOperatorHeader ':' TypeExpression.

MemberOperatorHeader:
  MemberOperatorHeader2;
  opaque MemberOperatorHeader2;
  ghost MemberOperatorHeader2.

MemberOperatorHeader2:
  LeftOperatorHeader;
  RightOperatorHeader;
  NeitherOperatorHeader.

LeftOperatorHeader:
  operator OperatorParam RevRedefinableOp;
  operator OperatorParam RevRedefinableOpWithPragma

RightOperatorHeader:
  operator RedefinableOp OperatorParam;
  operator RedefinableOpWithPragma OperatorParam.

EqualityOperatorHeader:
  operator OPEQUAL TypelessOperatorParam.

GhostEqualityOperatorHeader:
  ghost EqualityOperatorHeader.

RankOperatorHeader:
  RankOperatorHeader2;
  total RankOperatorHeader2.

RankOperatorHeader2:
  operator OPRANK TypelessOperatorParam.

NeitherOperatorHeader:
  operator RedefinableOp;

operator RedefinableOpWithPragma.

OpProperties:
   SimpleOpProperties;
   SimpleOpProperties **identity** Expression.

SimpleOpProperties:
   SimpleOpProperties OPPROPERTY;
   Empty.

Precondition:
   pre ListOfPredicates.

OptPrecondition:
   Precondition;
   Empty.

OptParmList:
   '(' FormalParams ')';
   Empty.

FormalParams:
   FormalParameterList;
   RepeatedParams;
   FormalParameterList Separator RepeatedParams.

RepeatedParams:
   repeated FormalParameterList.

FormalParameterList:
   ParameterDeclarations;
   FormalParameterList Separator ParameterDeclarations.

Separator:
   ',';
   ARROW.

OperatorParam:
   '(' SingleParm ')'.

TypelessOperatorParam:
   '(' IdentifierOptPragma ')'.

ParameterDeclarations:
   SingleParm;
   TwoOrMoreParameters ':' ParameterType.

SingleParm:
   IdentifierOptPragma ':' ParameterType.

TwoOrMoreParameters:
   IdentifierOptPragma Separator ParameterNameList.

ParameterNameList:
   IdentifierOptPragma;
   ParameterNameList Separator IdentifierOptPragma.

ModifyingSchemaDeclaration:
   ModifyingSchemaHeader OptExceptionSignature OptRequire OptPrecondition OptRecursionVariant **post**
Postcondition OptImplementation OptPostAssertion.

SchemaDeclaration:
   SchemaHeader OptExceptionSignature OptRequire OptPrecondition OptRecursionVariant **post**
Postcondition OptImplementation OptPostAssertion.

SchemaPrototype:
   SchemaHeader OptExceptionSignature OptRequire OptPrecondition OptPostAssertion;
   ModifyingSchemaHeader OptExceptionSignature OptRequire OptPrecondition OptPostAssertion.

AbsurdSchemaDeclaration:
   absurd SchemaHeader;
   absurd ModifyingSchemaHeader.

SchemaHeader:
   schema SchemaHead;
   OpaqueOrGhost **schema** SchemaHead.

ModifyingSchemaHeader:
   ModifyingSchemaHeader2;
   OpaqueOrGhost ModifyingSchemaHeader2.

ModifyingSchemaHeader2:
   schema '!' SchemaHead.

SchemaHead:
   SimpIdentifier OptSchemaParmList;
   IdWithPragma OptSchemaParmList.

OptSchemaParmList:
   '(' SchemaParams ')';
   Empty.

SchemaParams:
   SchemaParameterList;
   RepeatedParams;

SchemaParameterList Separator RepeatedParams.

SchemaParameterList:
   SchemaParameterDeclarations;
   SchemaParameterList Separator SchemaParameterDeclarations.

SchemaParameterDeclarations:
   DecoratedIdentifierOptPragma ':' ParameterType;
   DecoratedIdentifierOptPragma ':' **limited** ParameterType;
   DecoratedIdentifierOptPragma ':' **out** ParameterType;
   TwoOrMoreSchemaParameters ':' ParameterType;
   TwoOrMoreSchemaParameters ':' **limited** ParameterType;
   TwoOrMoreSchemaParameters ':' **out** ParameterType.

TwoOrMoreSchemaParameters:
   DecoratedIdentifierOptPragma Separator DecoratedIdentifierOptPragma;
   TwoOrMoreSchemaParameters Separator DecoratedIdentifierOptPragma.

DecoratedIdentifierOptPragma:
   IdentifierOptPragma;
   IdentifierOptPragma '!'.

- Theorem and axiom declarations
TheoremDeclaration:
   property OptIdentifier OptSchemaParmList OptPrecondition GeneralAssertion;
   property OptIdentifier OptSchemaParmList OptPrecondition **post** Postcondition GeneralAssertion.

ModifyingTheoremDeclaration:
   '!' **property** OptIdentifier OptSchemaParmList OptPrecondition **post** Postcondition GeneralAssertion.

- An axiom declaration is like a theorem declaration but cannot have a proof
AxiomDeclaration:
   axiom OptIdentifier OptParmList OptPrecondition AssertionWithoutProof.

OptIdentifier:
   SimpIdentifier;
   Empty.

Postcondition:
   Postcondition0;
   '?'.

Postcondition0:
   PostconditionList;
   change Expr8pList **satisfy** ListOfPredicates.

PostconditionList:
   PostconditionList ',' PostconditionElement;

PostconditionElement.

PostconditionElement:
  forall BoundVariableDeclarations SUCHTHAT PostconditionElement;
  PostconditionElem0.

PostconditionElem0:
  PostconditionElem0 **then** PostconditionElem1;
  PostconditionElem1.

PostconditionElem1:
  PostconditionElem1 OPAND PostconditionElem2;
  PostconditionElem2.

PostconditionElem2:
  Expr8lp '!' OPEQUAL Expr3to4;
  Expr8lp '!' AssignableOp Expr3to4;
  Expr8lp '!' BoolAssignableOp Expr3to4;
  '(' InBracketsPostconditionElem ')';
  SchemaCall;
  pass.

SchemaCall:
  - Schema calls that modify the current object
  '!' IdOrMember SchemaActualParameterList;
  '!' IdOrMember OptActualParameterList;
  Expr8lp '!' IdOrMember SchemaActualParameterList;
  Expr8lp '!' IdOrMember OptActualParameterList;
  - Schema calls that do not modify the current object
  Expr8np '.' IdOrMember SchemaActualParameterList;
  Expr8lp '.' IdOrMember SchemaActualParameterList;
  GeneralIdOrMember SchemaActualParameterList.

InBracketsPostconditionElem:
  LetDeclAssertionList InBracketsPostconditionElem2;
  InBracketsPostconditionElem2.

InBracketsPostconditionElem2:
  LocalVarDecls ';' InBracketsPostconditionElem;
  Postcondition0;
  GuardedPostconditionElements.

GuardedPostconditionElements:
  GuardedPostconditionElemsNoElse;
  opaque GuardedPostconditionElemsNoElse;
  GuardedElemsComma EmptyGuard PostconditionList;
  GuardedElemsComma NullGuard.

GuardedPostconditionElemsNoElse:
  '[' Expression ']' ':' PostconditionList;
  GuardedElemsComma '[' Expression ']' ':' PostconditionList.

GuardedElemsComma:
  '[' Expression ']' ':' PostconditionList ',';
  GuardedElemsComma '[' Expression ']' ':' PostconditionList ','.

Expr8pList:
  Expr8pNotCall;
  FuncOrSelecCall;
  Expr8pList ',' Expr8pNotCall;
  Expr8pList ',' FuncOrSelecCall.

FuncOrSelecCall:
  Expr8lp '.' IdOrMember OptActualParameterList;
  GeneralIdOrMember OptActualParameterList.

GeneralAssertion:
  AssertionWithoutProof;
  AssertionWithoutProof **proof** ProofList **end**;
  AssertionWithoutProof **proof** ProofListSEMI **end**.

AssertionWithoutProof:
  assert ListOfPredicates.

OptPostAssertion:
  GeneralAssertion;
  AssertionWithInherit;
  AssertionWithInherit **proof** ProofList **end**;
  AssertionWithInherit **proof** ProofListSEMI **end**;
  Empty.

AssertionWithInherit:
  assert DOTDOTDOT ',' ListOfPredicates;
  assert DOTDOTDOT.

ProofList:
  SimpleProofList;
  ConditionalProof.

ConditionalProof:
  if ConditionalProofList **fi**;
  if ConditionalProofListSEMI **fi**;
  SimpleProofListSEMI **if** ConditionalProofList **fi**;
  SimpleProofListSEMI **if** ConditionalProofListSEMI **fi**.

ProofListSEMI:
  SimpleProofListSEMI;
  ConditionalProof ';'.

SimpleProofList:
  ProofItem;
  SimpleProofListSEMI ProofItem.

SimpleProofListSEMI:
  SimpleProofList ';'.

ProofItem:
  LetDeclaration;
  AssertionStatement.

ConditionalProofList:
  GuardedProofList;
  ConditionalProofListSEMI GuardedProofList.

ConditionalProofListSEMI:
  GuardedProofListSEMI;
  ConditionalProofListSEMI GuardedProofListSEMI.

GuardedProofList:
  '[' Expression ']' ':' ProofList.

GuardedProofListSEMI:
  '[' Expression ']' ':' ProofListSEMI.

------------------ *Class and type declarations* ------------------

ClassOrTypeDeclaration:
  ClassDeclName DEFAS **tag**;
  ClassDeclName DEFAS **tag** '(' Expression ')';
  ClassDeclName DEFAS EnumDefinition;
  PossiblyFinalClassDeclaration;
  ClassDeclName OptClassParamsDefas PossConstrainedTypeExpression.

PossiblyFinalClassDeclaration:
  final ClassDeclaration;
  deferred ClassDeclaration;
  ClassDeclaration.

ClassDeclaration:
  ClassDeclName OptClassParamsDefas ClassBody.

ClassDeclName:
  class IdentifierOptPragma.

OptClassParamsDefas:
  of '(' ClassParameterList ')' OptRequire DEFAS;
  of SimpIdentifier OptRequire DEFAS;
  DEFAS.

ClassBody:
  ClassBody2;
  storable ClassBody2;
  inherits ClassName ClassBody2.

ClassBody2:
  AbstractDeclarations OptInternalDeclarations OptConfinedDeclarations OptInterfaceDeclarations **end**;
  ConfinedDeclarations OptInterfaceDeclarations **end**;
  InterfaceDeclarations **end**.

EnumDefinition:
  enum IdentifierOptPragmaList OptCOMMA **end**.

IdentifierOptPragmaList:
  IdentifierOptPragma;
  IdentifierOptPragmaList ',' IdentifierOptPragma.

AbstractDeclarations:
  abstract AbstractMemberDeclarations OptSEMI;
  abstract .

AbstractMemberDeclarations:
  AbstractMemberDeclaration;
  AbstractMemberDeclarations ';' AbstractMemberDeclaration.

OptCOMMA:
  ',';
  Empty.

ClassInvariant:
  invariant ListOfPredicates.

HistoryInvariant:
  '!' **invariant** ListOfPredicates OptExempt.

OptExempt:
  exempt SimpIdentifierList;
  Empty.

OptInternalDeclarations:
  internal InternalDeclarations OptSEMI;
  internal ;
  Empty.

InternalDeclarations:
  InternalMemberDeclaration;
  InternalDeclarations ';' InternalMemberDeclaration.

InterfaceDeclarations:
  interface InterfaceDecls OptSEMI;
  interface.

OptInterfaceDeclarations:
  InterfaceDeclarations;
  Empty.

ConfinedDeclarations:
  confined ConfinedDecls OptSEMI;
  confined.

OptConfinedDeclarations:
  ConfinedDeclarations;
  Empty.

InterfaceDecls:
  InterfaceOrConfinedDeclaration;
  EqualityOrRankDeclaration;
  InterfaceDecls ';' InterfaceOrConfinedDeclaration;
  InterfaceDecls ';' EqualityOrRankDeclaration.

ConfinedDecls:
  InterfaceOrConfinedDeclaration;
  ConfinedDecls ';' InterfaceOrConfinedDeclaration.

ExpressionList:
  Expression;
  ExpressionList ',' Expression.

ListOfPredicates:
  ExpressionList.

ConstructorDeclaration:
  ConstrDecl2;
  OpaqueOrGhost ConstrDecl2.

ConstrDecl2:
  build '{' OptConstructorParams '}' OptExceptionSignature OptRequire OptPrecondition
OptRecursionVariant OptConstructorBody;
  build name NONEMPTYSTRINGLITERAL '{' OptConstructorParams '}' OptExceptionSignature
OptRequire OptPrecondition OptRecursionVariant OptConstructorBody.

ConstructorPrototype:
  '{' OptConstructorParams '}' OptExceptionSignature OptRequire OptPrecondition OptPostAssertion.

OptConstructorParams:
  ConstructorParameterList;
  RepeatedParams;
  ConstructorParameterList Separator RepeatedParams;
  Empty.

ConstructorParameterList:
  ConstructorParams;
  ConstructorParameterList Separator ConstructorParams.

- Note that constructor parameters cannot be polymorphic.
ConstructorParams:
  ParamOptDecBang ':' TypeExpression;
  TwoOrMoreConstructorParameters ':' TypeExpression.

ParamOptDecBang:
  IdentifierOptPragma;
  '!' SimpIdentifier.

TwoOrMoreConstructorParameters:
  ParamOptDecBang Separator ParamOptDecBang;
  TwoOrMoreConstructorParameters Separator ParamOptDecBang.

OptConstructorBody:
  DEFAS Expression OptImplementation OptPostAssertion;
  inherits Expression OptConstructorPostcondition OptPostAssertion;
  OptConstructorPostcondition OptPostAssertion.

OptConstructorPostcondition:
  post Postcondition OptImplementation;
  Empty.

---------------------- *'require' parts* ----------------------

OptRequire:
  require RequirementList;
  Empty.

RequirementList:
  RequirementItem;
  RequirementList ',' RequirementItem.

RequirementItem:
  identifier within TypeExpression;
  identifier has MemberDeclarationPrototypeList OptSEMI **end**.

MemberDeclarationPrototypeList:
   MemberDeclarationPrototype;
   MemberDeclarationPrototypeList ';' MemberDeclarationPrototype.

MemberDeclarationPrototype:
   MemberFSOSPrototype;
   EqualityOrRankPrototype;
   ConstructorPrototype.

------------------ *Exception signatures* ------------------

OptExceptionSignature:
   throw TypeExpression;
   Empty.

---------------------- *Expressions* ----------------------

PossiblyMultipleExpression:
   Expression;
   TwoOrMoreExpressions.

TwoOrMoreExpressions:
   Expression ',' Expression;
   TwoOrMoreExpressions ',' Expression.

- *Single expressions*

Expression:
   PrimableExpression;
   UnprimableExpression.

UnprimableExpression:
   Expr0np;
   - *The following types of expression involve constructs which span to the end of the expression and have to be treated specially*
   Expr3to4np ASorIS TypeExpression;
   Expr3to4np WithinOrNot TypeExpression;
   Expr8lp WithinOrNot TypeExpression;
   Expr3to4 **after** PostconditionElement;
   THATorANY Expression;
   THATorANY BoundVariableDeclaration SUCHTHAT Expression;
   those BoundVariableDeclaration SUCHTHAT Expression;
   forall BoundVariableDeclarations SUCHTHAT Expression;
   exists BoundVariableDeclarations SUCHTHAT Expression;
   for those BoundVariableDeclaration SUCHTHAT Expression **yield** Expression;
   for BoundVariableDeclaration **yield** Expression.

PrimableExpression:
  Expr8lp ASorIS TypeExpression;
  Expr8lp.

BoundVariableDeclaration:
  SimpIdentifier ':' TypeExpr2;
  SimpIdentifier DBLCOLON Expr4.

BoundVariableDeclarations:
  BoundVariableDecls;
  BoundVariableDeclarations ',' BoundVariableDecls.

BoundVariableDecls:
  SimpIdentifierList ':' TypeExpr2;
  SimpIdentifierList DBLCOLON Expr4.

Expr0np:
  Expr0 Op0 Expr1;   *- Op0 are ==> <==> <==*
  Expr1np.

Expr0:
  Expr0np;
  Expr8lp.

Expr1np:
  Expr1 Op1 Expr2;   *- Op1 is |*
  Expr2np.

Expr1:
  Expr1np;
  Expr8lp.

Expr2np:
  Expr2 Op2 Expr3;   *- Op2 is &*
  Expr3np.

Expr2:
  Expr2np;
  Expr8lp.

Expr3np:
  CompareList Expr3to4;
  Expr3to4np.

Expr3:
  Expr3np;
  Expr8lp.

CompareList:
  Expr3to4np Op3;   *- Op3 are the comparisons and negated comparisons*
  Expr8lp Op3;
  CompareList Expr3to4np Op3;
  CompareList Expr8lp Op3.

Expr3to4np:
  Expr3to4 OPRANK Expr4;
  Expr3to4 **like** Expr4;
  Expr3to4np NOT **like** Expr4;
  Expr8lp NOT **like** Expr4;
  Expr3to4np Op3to4 Expr4;   *- Op3to4 are "in" and "~in"*
  Expr8lp Op3to4 Expr4;
  Expr4np.

Expr3to4:
  Expr3to4np;
  Expr8lp.

Expr4np:
  Expr4 Op4 Expr5;   *- Op4 are + - ++ --*
  Expr5np.

Expr4:
  Expr4np;
  Expr8lp.

Expr5np:
  Expr5 Op5 Expr6;   *- Op5 are * / % ** # %% ##*
  Expr6np.

Expr5:
  Expr5np;
  Expr8lp.

Expr6np:
  Expr6 Op6 Expr7;   *- OP6 are ^ and ..*
  Expr7np.

Expr6:
  Expr6np;
  Expr8lp.

Expr7np:
  NOT Expr7;
  Monop Expr7;
  TYPEOP TypeName;
  OverOp **over** Expr7;

Expr8np.

Expr7:
   Expr7np;
   Expr8lp.

- Unprimable expressions
Expr8np:
   Expr8np IndexingExpression;
   Expr8np '.' **value**;
   Expr8p ''';   *- primed expression*
   Expr8np '.' IdOrMember OptActualParameterList;
   ref Expression **on** IDENTIFIER;
   - Constructor calls
   ClassNameNotPoly '{' OptActConstructorParams '}';
   - Literals
   NONEMPTYSTRINGLITERAL;
   EMPTYSTRINGLITERAL;
   LITERAL;
   - Brackets
   '(' UnprimableInBracketsExpr ')';
   '(' LetDeclAssertionList UnprimableInBracketsExpr ')';
   - Allow '?' to represent an unfinished program
   '?'.

- Primable and limited-primable expressions
Expr8lp:
   Expr8pNotCall;
   - Function and selector calls (we can't tell which, so assume selector which is primable)
   - Also variables and member expressions
   FuncOrSelecCall;
   result;
   - Bracketed expressions are here because expressions of the form "(e is T)" may be primable
   '(' PrimableExpression ')';
   '(' LetDeclAssertionList PrimableExpression ')'.

- Fully Primable expressions
Expr8pNotCall:
   ITorSELF;
   Expr8lp '.' **value**;
   Expr8lp IndexingExpression.

IndexingExpression:
   '[' Expression ']'.

Expr8p:
   Expr8pNotCall;
   - Function and selector calls (we can't tell which, so assume selector which is primable)

- Also variables and member expressions
   FuncOrSelecCall.

IdOrMember:
   IDENTIFIER;
   super IDENTIFIER.

GeneralIdOrMember:
   IDENTIFIER;
   super IDENTIFIER;
   IDENTIFIER '@' ClassName;   *- deprecated syntax*
   ClassName IDENTIFIER.   *- new syntax*

OptActualParameterList:
   '(' ExpressionSepList ')';
   Empty.

ExpressionSepList:
   Expression;
   ExpressionSepList Separator Expression.

- A schema actual parameter list must have at least one "!" actual parameter in it
SchemaActualParameterList:
   '(' SchemaActualParameters ')';
   '(' ExpressionSepList Separator SchemaActualParameters ')'.

SchemaActualParameters:
   Expr8lp '!';
   SchemaActualParameters Separator Expression;
   SchemaActualParameters Separator Expr8lp '!'.

- Various forms of expressions in brackets
UnprimableInBracketsExpr:
   UnprimableExpression;
   TwoOrMoreExpressions;
   ChoicesWithElse;
   opaque ChoicesNoElse;
   ChoicesNoElse.

LetDeclAssertionList:
   LetDeclaration ';';
   AssertionStatement ';';
   TraceStatement ';';
   LetDeclAssertionList LetDeclaration ';';
   LetDeclAssertionList AssertionStatement ';';
   LetDeclAssertionList TraceStatement ';'.

LetDeclaration:
  let IdentifierOptPragma DEFAS Expression OptNvImplementation.

AssertionStatement:
  GeneralAssertion.

TraceStatement:
  trace ExpressionSepList.

ChoicesWithElse:
  ChoicesNoElse ',' EmptyGuard Expression.

ChoicesNoElse:
  Choice;
  ChoicesNoElse ',' Choice.

Choice:
  '[' Expression ']' ':' Expression.

EmptyGuard:
  '[' ']' ':'.

NullGuard:
  '[' ']'.

OptActConstructorParams:
  ExpressionSepList;
  Empty.

--------------------- *Type expressions* ---------------------

ParameterType:
  TypeExpression;
  TemplateParameter.

TemplateParameter:
  class SimpIdentifier.

FunctionType:
  FunctionTypeList;
  ':' TypeExpression.

FunctionTypeList:
  FunctionTypeElements;
  FunctionTypeList ',' FunctionTypeElements.

FunctionTypeElements:
  SimpIdentifierList ':' TypeExpression.

ConstrainedTypeExpression:
  those IDENTIFIER':' TypeExpr2 SUCHTHAT Expression;
  TypeExpr3 Op3 Expr4;
  TypeExpr3 Op3to4 Expr4;
  BracketedConstrainedTypeExpr.

BracketedConstrainedTypeExpr:
  '(' ConstrainedTypeExpression ')'.

PossConstrainedTypeExpression:
  ConstrainedTypeExpression;
  TypeExpression.

TypeExpression:
  TypeExpression UNITE TypeExpr2;
  TypeExpr2.

TypeExpr2:
  RefClassName;
  TypeExpr2a.

TypeExpr2a:
  FromClassName;
  TypeExpr3.

TypeExpr3:
  ClassName;
  '(' TypeExpression ')'.

ClassName:
  IDENTIFIER **of** ClassNameParameters;
  PREDEFCLASS **of** ClassNameParameters;
  IDENTIFIER;
  PREDEFTYPE.

- Template type argument lists
ClassNameParameters:
  TemplateParameter;
  ClassNameAsTypeExpr;
  FromClassName;
  RefClassName;
  '(' TypeExpressionList ')'.

ClassNameAsTypeExpr:
  ClassName.

FromClassName:
  from ClassName.

RefClassName:
  ref limited TypeExpr2a **on** IDENTIFIER;
  ref TypeExpr2a **on** IDENTIFIER.

- ClassNameNotPoly is used in the syntax for constructors.
ClassNameNotPoly:
  IDENTIFIER **of** ClassNameParameters;
  PREDEFCLASS **of** ClassNameParameters;
  TypeName.

TypeExpressionList:
  ParameterType;
  TypeExpressionList Separator ParameterType.

TypeName:
  IDENTIFIER;
  PREDEFTYPE.

- Abbreviated type expressions are permitted after ":" when declaring a single non-bound local variable
AbbrevTypeExpr:
  those TypeExpr2 SUCHTHAT Expression;
  AbbrevTypeExpr1.

AbbrevTypeExpr1:
  '(' AbbrevTypeExpr ')'.

-------------------- *Implementations* ----------------------

OptImplementation:
  Implementation;
  Empty.

Implementation:
  via OptImplVariantSemi ImplList OptSEMI **end**.

- Same as above but no variant permitted
OptNvImplementation:
  via ImplList OptSEMI **end**;
  Empty.

ImplList:
  ImplItem;
  LocalVarDecls;
  HeapDeclarations;
  ImplList ';' ImplItem;
  ImplList ';' LocalVarDecls;
  ImplList ';' HeapDeclarations.

- Implementation items valid in both functions and schemas
ImplItem:
   begin ImplList OptSEMI **end**;
   par ImplList OptSEMI **end**;
   if Conditional **fi**;
   value PossiblyMultipleExpression;
   done;
   LocalDeclaration;
   LetDeclaration;
   Postcondition OptNvImplementation;
   GeneralAssertion;
   IdentifierOptPragma ':' Precondition;
   goto IDENTIFIER;
   - A loop must either have a 'change' part, or some local variable declarations, otherwise it really can't do anything.
   loop OptLocalVars **change** Expr8pList **keep** ListOfPredicates OptLoopUntil LoopVariant ';' ImplList OptSEMI **end**;
   loop LocalVarDecls ';' **keep** ListOfPredicates OptLoopUntil LoopVariant ';' ImplList OptSEMI **end**;
   throw Expression;
   throw;
   try ImplList OptSEMI **catch** CatchList **end**.

LocalVarDecls:
   var LocalVarDeclGroup;
   LocalVarDecls ',' LocalVarDeclGroup.

LocalVarDeclGroup:
   SimpIdentifier ':' PossConstrainedTypeExpression;
   SimpIdentifier ':' TypeExpression '!' OPEQUAL Expression;
   SimpIdentifier ':' BracketedConstrainedTypeExpr '!' OPEQUAL Expression;
   SimpIdentifier ':' AbbrevTypeExpr;
   SimpIdentifier ':' AbbrevTypeExpr1 '!' OPEQUAL Expression;
   TwoOrMoreSimpIdentifiers ':' PossConstrainedTypeExpression.

OptLocalVars:
   LocalVarDecls ';';
   Empty.

OptLoopUntil:
   until ListOfPredicates;
   Empty.

Conditional:
   ConditionalNoElse;
   ConditionalNoElseSemi;
   ConditionalNoElseSemi EmptyGuard ImplList OptSEMI;
   ConditionalNoElseSemi NullGuard.

ConditionalNoElse:
   GuardedImplList;
   ConditionalNoElseSemi GuardedImplList.

ConditionalNoElseSemi:
   GuardedImplList ';';
   ConditionalNoElseSemi GuardedImplList ';'.

GuardedImplList:
   '[' Expression ']' ':' ImplItem;
   '[' Expression ']' ':' LocalVarDecls;
   GuardedImplList ';' ImplItem;
   GuardedImplList ';' LocalVarDecls.

CatchList:
   CatchItem;
   CatchItem ';';
   CatchItem ';' CatchList.

CatchItem:
   '[' IdentifierOptPragma ':' TypeExpression ']' ':' ImplItem;
   '[' IdentifierOptPragma ':' TypeExpression ']' ':' LocalVarDecls;
   CatchItem ';' ImplItem;
   CatchItem ';' LocalVarDecls.

-------------------- *Variants* --------------------

OptRecursionVariant:
   decrease ExpressionList;
   decrease DOTDOTDOT;
   decrease DOTDOTDOT ',' ExpressionList;
   Empty.

OptImplVariantSemi:
   decrease ExpressionList ';';
   Empty.

LoopVariant:
   decrease ExpressionList.

-------------------- *Operators* --------------------

- Some operators can be both unary and binary. The following allows such operators to be both (e.g. OP3M represents the operators ">" and "<").
- Also, all Op3 operators return Boolean results and can be prefixed by "~" to generate the inverse operator

Op0:
   OP0.

Op1:
  OPOR.

Op2:
  OPAND.

Op3:
  PlainOp3;
  NOT PlainOp3.

PlainOp3:
  OP3;   *- binary only, <= <<= >= >>= << >>*
  OPEQUAL;
  OP3M.   *- unary or binary, < >*

Op3to4:
  in;
  NOT **in**.

Op4:
  OP4;   *- binary only, ++ --*
  OP4M.   *- unary or binary, + -*

Op5:
  OP5;   *- binary only, ***
  OP5M.   *- unary or binary, * % /*

Op6:
  OP6;
  OP6M.

Monop:
  OP3M;
  OP4M;
  OP5M;
  OP6M.

IndexOp:
  '[' ']'.

IndexOpWithPragma:
  IndexOp Pragma.

RedefinableOp:
  RevRedefinableOp;
  IndexOp.

RedefinableOpWithPragma:
   RevRedefinableOpWithPragma;
   IndexOpWithPragma.

RevRedefinableOp:
   OP3;
   OP3M;
   OP4;
   OP4M;
   OP5;
   OP5M;
   OP6;
   OP6M;
   in.

RevRedefinableOpWithPragma:
   RevRedefinableOp Pragma.

- All bound binary operators except the comparisons can be used with "over"
OverOp:
   Op4;
   Op5;
   Op6.

- All binary operators except the comparisons can be used after "!"
AssignableOp:
   Op4;
   Op5;
   Op6.

- Boolean operators
BoolAssignableOp:
   Op0;
   Op1;
   Op2.

WithinOrNot:
   within;
   NOT **within**.

------------------ *Miscellaneous* ------------------

OpaqueOrGhost:
   opaque;
   ghost.

SimpIdentifier:
   IDENTIFIER.

- When declaring most names, we allow the identifier to be modified by a pragma
IdentifierOptPragma:
   IDENTIFIER;
   IdWithPragma.

IdWithPragma:
   IDENTIFIER Pragma.

Pragma: **pragma** PragmaMapping;
   - The remaining forms are deprecated
   name NONEMPTYSTRINGLITERAL;
   LINKAGE OptNONEMPTYSTRINGLITERAL;
   LINKAGE OptNONEMPTYSTRINGLITERAL **name** NONEMPTYSTRINGLITERAL.

PragmaMapping:
   '(' PragmaMapList ')'.

PragmaMapList:
   PragmaMapItem;
   PragmaMapList ',' PragmaMapItem.

PragmaMapItem:
   IDENTIFIER OPEQUAL NONEMPTYSTRINGLITERAL;
   IDENTIFIER OPEQUAL IDENTIFIER;
   IDENTIFIER OPEQUAL PragmaMapping.

OptSTRINGLITERAL:
   NONEMPTYSTRINGLITERAL;
   Empty.

OptSEMI:
   ';' ;
   Empty.

----------------- End of grammar ----------------