Getting Started with Perfect

Version 3.0, December 2004

Contents

Preface

1 Basics of building reliable software systems
   1.1 Introduction
   1.2 Component level specification
   1.3 Modeling the data
   1.4 Describing the interfaces
   1.5 Describing the behavior
   1.6 An example specification

2 The type system
   2.1 Built-in types, collections and structures
   2.2 Enumerations
   2.3 Constrained types
   2.4 Abstract classes
   2.5 Unions

3 Expressions in Perfect
   3.1 Operator expressions
   3.2 Brackets
   3.3 Conditional expressions
   3.4 Function and constructor calls
   3.5 Operations on collections
   3.6 'after' expressions
   3.7 Type checks and conversions

4 Postconditions in Perfect
   4.1 Assignment postconditions
   4.2 Brackets and conditional postconditions
   4.3 Schema calls
   4.4 Combining postconditions
   4.5 Iterating over collections
   4.6 Doing nothing

4.7 The general form of postcondition

5 Abstract classes and methods
   5.1 The abstract section
   5.2 The interface section
   5.3 Constructors
   5.4 Using post-assertions
   5.5 Bringing it all together
   5.6 Recursion

6 Class inheritance
   6.1 Principles
   6.2 Specifying inheritance
   6.3 The confined section
   6.4 Redefining inherited methods
   6.5 Using polymorphism
   6.6 Using post-assertions with inheritance
   6.7 Preventing inheritance and overriding
   6.8 Deferred methods and classes
   6.9 Deferring a precondition
   6.10 Using ghost methods to represent abstract properties
   6.11 Calling methods from class invariants and deferred class constructs

7 Copying, equality and ordering
   7.1 Semantics of assignment and parameter passing
   7.2 Equality
   7.3 Ordering relations

8 Implementations
   8.1 The internal section
   8.2 Re-implementing confined and interface methods
   8.3 Re-implementing abstract methods, equality and ordering operators

9 Statements
   9.1 Declarations, let-statements and assertions
   9.2 Postcondition statements
   9.3 Conditional statements
   9.4 Completers
   9.5 Loops
10 Summary

Preface

This text is intended to help you to get started with the Perfect language. It introduces the only the most commonly-used features of the Perfect language and does not attempt to be comprehensive. When you require a more complete description of a particular construct or facility, please refer to the Language Reference Manual. To learn how to run the Perfect Developer project manager, compiler and verifier, refer to the User Guide.

The name Perfect is not intended to claim that the language itself is perfect, rather that the Perfect language makes it far easier to construct perfect programs than conventional programming languages and tools. What’s more, Perfect is productive and fun to use. Enjoy!

1 Basics of building reliable software systems

1.1 Introduction

Managers of software projects are faced with the task of delivering a product that meets the requirements on-time and within budget. Frequently, a product will need to be reworked to some degree to fix bugs. The manager’s dilemma is whether to delay product release until the rework is complete, or deliver a buggy product in the hope that customer dissatisfaction can be kept at bay until the updated version can be delivered. Many customers are wary of purchasing a major new version of a product, preferring instead to wait at least until the first maintenance release is available.

It has long been recognized that a higher quality product is produced if complete and detailed specifications are produced at the start of the development process. In theory, this means that the form and exact functionality of the final product is known from the start. Unfortunately, traditional methods of handling specifications leave the following questions largely unanswered:

- How can I be sure that the specification is complete, unambiguous and takes account of all combinations of circumstances that may occur?

1.2 Component level specification

A specification may be produced for an individual component or class, a subsystem, an entire software system, or even an entire system comprising hardware as well as software.

We will start by considering the specification for an individual component or class. A class specification is essentially a clients-eye view of the class. Such a specification must describe the following:

- A model of the data maintained by the component;
- The interfaces between the component and the outside world and the data that may be sent and received through each interface;
- The behavior of the component in response to allowable inputs (in terms of the changes made to the data model and the data returned through the interface).

We will deal with each of these aspects in turn.

1.3 Modelling the data
Most components and classes contain data. Using object-oriented development practice, the data is not directly exposed by the interface; instead, it can only be accessed and modified via the interface.

Even though a developer using the component does not need to know the details of how the data is stored, it is still necessary to have a mental model of the stored information in order to understand what the interface methods do. Perfect requires this model to be declared explicitly by way of “abstract variable” declarations, optionally with constraints (called “class invariants”) to restrict the allowed values.

When declaring abstract data, always use the simplest possible abstract model. Do not worry about how you will actually store the data efficiently; such implementation detail has no place in a specification. Take care to avoid describing redundant data (i.e. data elements whose value is determined by the values of other data elements); these sorts of properties should be described as functions of the remaining data, even if it is expected that an efficient implementation will want to store them.

1.4 Describing the interfaces

Interfaces are the member functions and other methods that are provided to access and modify the stored data. Perfect allows the following sorts of method:

- A function returns a result that depends on the stored data and on the (optional) parameter list and has no side-effects;
- An operator is similar to a function but its name is a symbol instead of an identifier;
- A schema may modify both the stored data and its parameters;
- A selector provides direct read/write access to a subcomponent of the class instance;
- A constructor is a class method that returns an instance of the class.

Perfect also allows abstract variables to be redeclared as interface functions or selectors, making them directly accessible with read-only or read-write access respectively. This facility should be used with caution as it makes it impossible to change the stored form of the corresponding data without changing the interface.

1.5 Describing the behavior

The behavior of an interface method comprises two parts:

- The conditions that must be satisfied before the method may be invoked (the precondition)
  - The result value returned (if any) and the changes (if any) made to the abstract data and parameters (these changes are known as the postcondition)

Taken together, the precondition and result value or postcondition form a contract:

Whenever the method is invoked, then provided the precondition is satisfied, the method guarantees to preserve the class invariant and to satisfy the specified postcondition or deliver the specified result.

This style of programming is known as Programming by Contract. Whereas some earlier programming languages merely allow contracts to be expressed, Perfect provides the means to guarantee that a method will honor its contract.

1.6 An example specification

We will take as our example a component to check the spelling of text in an application (such a component might be used by a word processor, spreadsheet, email package etc.). The core of the spell checker is a dictionary of known words and we will focus on this item.

We shall assume the following requirements for the dictionary:

- We submit alleged words to the dictionary one at a time; for each word submitted, the dictionary responds by indicating whether it recognizes the word
- If a word is not already in the dictionary, we can add it to the dictionary
- A word already in the dictionary can be removed from it
- We intend that if the dictionary does not recognize a word, then after adding the word it will be recognized
- Adding a word to the dictionary does not change the status of whether another, different word is in the dictionary
- We expect that if a word is not in the dictionary, then if we add it and then remove it, the dictionary we end up with is the same as the one we started with

1.6.1 An interface for the dictionary

We first need to define what a word is. We shall define a word as a string that is at least one character long and contains only letters. Since Perfect already has a type string (which is defined as seq of char), we have the option of defining a word as a constrained string rather than defining a new abstract
class to represent a word. For simplicity we will choose this option. So our component definition starts like this:

```scala
class Word "= those x: string :: #x -> 0 & (forall c::x :: c in \"a\"..\"z\"\"A\"..\"Z\")
```

The interface is a direct representation of the requirements:

```scala
class Dictionary "= interface
  function check(w: Word): bool
    "= ?;

  schema add(w: Word)
    pre ~check(w) post ?;

  schema remove(w: Word)
    pre check(w) post ?;

Note that we have included a precondition for both the add and remove schemas, in accordance with the requirements we were given. We could, of course, define a more tolerant interface; for example, we could permit a client to attempt to add a word that is already in the dictionary.

We also need at the very least a constructor to build an empty dictionary:

```scala
build() post ?;
```

Now we will express the required behavior, starting with our requirement that if we add a word that check says isn’t in the dictionary, then after using add to add it, check will confirm it is present:

```scala
property(w: Word) pre ~check(w)
  assert (self after it(add(w)).check(w));
```

Now the converse behavior for the remove operation:

```scala
property(w: Word) pre check(w)
  assert ~(self after it(remove(w)).check(w));
```

Now the requirement that adding a word doesn’t affect what other words are in the dictionary:

```scala
property(w1, w2: Word) pre ~check(w1), w1 != w2
  assert (self after it(add(w1)).check(w2) = check(w2));
```

Finally, the requirement that adding a word and then removing it leaves the dictionary unchanged:

```scala
property(w: Word) pre ~check(w)
  assert (self after it(add(w) then it(remove(w))) = self;
```

We close the class declaration with:

```scala
end;
```

### 1.6.2 An abstract data model for the dictionary

The abstract data maintained by the dictionary is obviously a collection of words. Perfect directly supports four types of collection: set, bag, sequence and mapping. In this case, the most appropriate collection is the set because:

- We do not need to store the words in any particular order (remember, we are dealing with an abstract model here) - it matters only whether an alleged word is in the dictionary or not; so a sequence is not appropriate
- It is not meaningful to include a word in the dictionary more than once, so a bag is not appropriate

```scala
class Dictionary "= abstract
  var words: set of Word;

interface ...
end;
```

### 1.6.3 Specification of the interface

We now need to define what the check function, the add and remove schemas and the constructor do with the abstract data. Our choice of the abstract data model and our requirements lead us to the following specification:

```scala
function check(w: Word): bool
  "= w in words;

schema add(w: Word)
  pre ~check(w)
  post words! = words.append(w);

schema remove(w: Word)
  pre check(w)
  post words! = words.remove(w);

build()
  post words! = set of Word{};
```

### 1.6.4 The completed example

Putting it all together and adjusting the layout, our complete dictionary specification is:

```scala
class Word "= those x: string :: #x -> 0 & (forall c::x :: c in \"a\"..\"z\"\"A\"..\"Z\")
```
A specification is not the same as an executable program; however, tools supporting Perfect are capable of generating programs directly from many specifications (including this one). Even if we subsequently choose to refine our dictionary component (e.g. by devising a more efficient way of storing the data), such a generated program serves as a functioning prototype.

## 2 The type system

### 2.1 Built-in types, collections and structures

Just as programming languages provide built-in data types and mechanisms for defining, so does Perfect.

The built-in basic types are `bool`, `char`, `int`, `void`, `real`, `byte` and `rank`. The first five are more or less what a programmer would expect (see the Language Reference Manual for details). The `rank` type represents the result of comparing two values of a type and has values `above`, `same` and `below`.

Perfect also provides built-in collection and structured types. The built-in collection types are `set` of `X`, `bag` of `X`, `seq` of `X` and `map` of `(X -> Y)`, where `X` and `Y` represent any type. The difference between `set`, `bag` and `seq` is:

- A `set` is an unordered collection of values. Any given value is either in the set or not in it (it makes no sense to ask how many times a value occurs in the set). For example, imagine the set of sizes available when buying a shirt of some particular brand and pattern.
- A `bag` is an unordered collection of values, but a value may be in the bag more than once. For example, imagine a bag full of shopping from the local supermarket.
- A `seq` (sequence) is an ordered collection of values - with the order being important. The same value may occur at more than one place in the sequence (unless a constraint is specified to prohibit this). Imagine a list of things to do, in the order in which they must be done.

A `map` (mapping) contains a set of values of one type (this set is called the domain) and a set of values of another type (the range). Each element in the domain is associated with exactly one element in the range (although several elements in the domain may be associated with a single element in the range). We say that each domain element maps to a range element. For example, imagine a mapping from a person’s full name to his/her address. Each person lives at only one address (assuming no two people have the same full name) but
several people may live at a single address.

The built-in structured types are `pair of (X, Y)` and `triple of (X, Y, Z)`. A pair comprises a single pair of values `x` and `y` (one of each of the two types) and a triple comprises one value from each of the three types (the values are named `x`, `y` and `z`). Pairs are often used in constructing mappings, and pairs or triples can be used when it is necessary to temporarily encapsulate two or three values but is not worth defining a new class.

The `Perfect string` class is just a sequence of characters:

```plaintext
class string ~ seq of char;
```

### 2.2 Enumerations

Like most programming languages, `Perfect` allows enumerated types to be defined, for example:

```plaintext
class color ~ enum red, yellow, green, blue end;
```

### 2.3 Constrained types

We have already met constrained types - in the definition of type `Word` in our dictionary example. A constrained type is simply some other type together with a constraint to limit the allowable values. It is usually expressed in the form:

```plaintext
type identifier : type : condition
```

where the condition depends on the identifier. For example, we defined type `Word` as:

```plaintext
type x : string = all l in 'a'..'z' : l is a letter end;
```

In this type expression, "x" and "c" are dummy variable names (we could choose any names we like, as long as they don't clash with other names in the current scope). The expression reads:

"those x of type string such that length of x not-equal-to zero and for all c in x, c is a letter".

Another example is the built-in constrained type `nat` which represents unsigned integers and is defined thus:

```plaintext
class nat = those x : int : x >= 0 end;
```

which reads, "class nat is-defined-as those x of type int such that x greater-than-or-equal-to zero".

When a variable or parameter having a constrained type is used in an expression, it is treated just like a value of the corresponding unconstrained type. When a value is assigned to a variable having a constrained type, `Perfect` requires that for the program to be valid, the value being assigned must satisfy the constraint. Similarly, when passing a value to a function and the parameter was declared in the function with a constrained type, the value must satisfy the constraint.

### 2.4 Abstract classes

The ability to declare abstract classes is at the core of object-oriented technology. We have already seen an example of an abstract class in our dictionary example. Abstract class declarations deserve several chapters to themselves, so we will not elaborate on them here.

### 2.5 Unions

Sometimes it is necessary to allow a variable, parameter or return value to take one of two or more types. Such a combination of types is called a united type or union and is expressed:

```plaintext
type | type | type ...
```

One of the most common forms of united type is the union of a user-defined class with `void` (whose only value is `null`), e.g.:

```plaintext
var optionalColor: color | void;
```

The union of all classes derived from some common ancestor base is a special case of a united type and is written `from base`.

### 3 Expressions in Perfect

`Perfect` has a wide range of expression types, most of which will be familiar to software developers. The main difference is that in `Perfect`, expressions do not have side-effects. As a result, expressions are never ambiguous and the compiler has plenty of scope for rearranging them.

Some types of expression have preconditions (conditions that must be satisfied for the expression to be valid). When validating a project, `Perfect` tools make sure that these conditions are satisfied.
3.1 Operator expressions

*Perfect* defines the usual range operators for integer, real and Boolean types. It also defines operators that act on the built-in collections. If you are coming to *Perfect* from C++ or Java, the most important differences are:

- The symbol for "not" is `!` (the exclamation mark is used to indicate a change in value of a variable or parameter)
- The "and" and "or" operators are `&` and `|` (the `||` symbol means "unite" in *Perfect*)
- It is possible to string multiple comparisons together, for example `a <= x <= b`.
- The symbol `#` is used for various operators that count a number of elements in the built-in collection types (for example, evaluating the length of a sequence)
- The `++` and `--` symbols are binary operators (so is `**`
- Unary `->` and `<` represent successor and predecessor operators (or `+1` and `-1` when applied to an integer operand)
- The binary operator `.' takes two values of integer, character or an enumerated type and yields a sequence of all values of the type in ascending order from the first up to the second

3.2 Brackets

Brackets are used not only to force a particular evaluation order but also to define names for temporary values, introduced by the keyword *let*. For example, to compute the fourth power of a variable `x`, we could use the expression:

```
(let square = x * x; square * square)
```

We can also include assertions within the brackets:

```
(assert x -> 0; x * x)
```

3.3 Conditional expressions

The conditional expression is a special form of bracketed expression. Following any *let*- declarations and assertions, instead of a single expression we provide two or more guarded expressions. A guarded expression takes the form:

```
[condition]; expression
```

so an example of a complete conditional expression (without any embedded *let*-declarations or assertions) is:

```
(total > 10): 'lots'; (total > 5): 'a few'; []: 'hardly any'
```

When evaluating a conditional expression, the guards are evaluated in order until one evaluates to true, then the corresponding expression is evaluated (the other expressions and the remaining guards are not evaluated). As shown here, the last guard may be empty (i.e. no expression is given within the square brackets), which is equivalent to the condition `true` and means "otherwise".

3.4 Function and Constructor calls

The usual syntax is used for calling a function. If the function takes no parameters, an empty pair of brackets is *not* used.

In *Perfect*, a constructor is a function that yields an instance of the class in which it is declared. Constructors are invoked by following the class name with a parameter list in braces. If there are no parameters, an empty set of braces is used.

3.5 Operations on collections

A number of operations are provided on values of the built-in collection types *set*, *bag* and *seq*. In the following, condition and expression depend on `identifier`:

- `forall identifier::collection :: condition` yields `true` if all the elements of *collection* satisfy *condition*, or if *collection* is empty
- `exists identifier::collection :: condition` yields `true` if at least one of the elements of *collection* satisfies *condition*
- `those identifier::collection :: condition` yields a collection of the same type as *collection* comprising those elements of *collection* that satisfy *condition*
- `that identifier::collection :: condition` yields the element of *collection* that satisfies *condition* (there must be exactly one)
- `any identifier::collection :: condition` yields an element of *collection* that satisfies *condition* (there must be at least one)
- `for identifier::collection yield expression` yields a collection similar to *collection* except the element type is the type of *expression*. The output collection comprises the elements in *collection* mapped to the values defined by *expression*
- `for those identifier::collection :: condition yield expression` is similar, but only those elements of *collection* that satisfy *condition* are taken.
3.6 'after' expressions

The expression 'expression after postcondition' yields the value that
expression would have if the changes specified by postcondition were made.
You can imagine that the value of expression is assigned to a temporary
variable called it, postcondition is then satisfied, and the resulting value of it
is returned. For example, given a variable myPair of type pair of (int, int), the
expression:

myPair after it.y! = 0

yields a pair of (int, int) whose x-value is the same as in myPair but whose
y-value is zero.

3.7 Type checks and conversions

Perfect is strongly typed. There are no type conversions that discard any part
of the value being converted.

Values that are not statically typed can be checked and converted at run time.
For example, if variable myVar has type 'string || char || void' then we can use
the following expressions:

- 'myVar within char || void' is true when the current value of myVar is of
type char or void
- 'myVar :within string' is true when the current value of myVar is not of
type string
- 'myVar is int' is only valid when the current value of myVar is of type int,
and yields that value
- '42 as int || void' converts the value 42 to type 'int || void'
- 'myVar like myOtherVar' yields true when the values of myVar and
myOtherVar have exactly the same type

Automatic type conversions are performed only in the following circumstances:

- A value may be automatically converted from a type to another type that
  encompasses the original type (e.g. a union of the original type and other
types)

When determining type compatibility, constraints are ignored. Perfect
compilers will of course check that constraints are satisfied when a value is
required to conform to a constrained type (e.g. during assignment or
parameter passing).

4 Postconditions in Perfect

As expressions in Perfect have no side effects, a separate syntax is required to
describe changes in the state of an object - that is, a syntax for postconditions.
To avoid ambiguity, it is always clear from the form of the postcondition which
objects are being changed - they are usually the ones followed by a "!".

As with expressions, some postconditions have preconditions which are
checked by Perfect tools when validating a project.

4.1 Assignment postconditions

One of the simplest changes that can be made to an object is to change it so
that its value is equal to that of another object. The form of postcondition
stating this has already been used in several example without explanation; the
general form is:

object! = expression

This indicates that object is changing (since it is followed by a "!")), and it
should change to become equal to expression.

A similar form of postcondition can be used to state that an object should be
changed by applying a binary operator whose result and left hand operands
are of the same type. For this form, the "=" is replaced by the desired
operator, for example, both of the following postconditions state that the value
of x should be increased by 5:

\[ x! = x + 5 \]
\[ x! + 5 \]

4.2 Brackets and conditional postconditions

As with expressions, brackets can be used to introduce new temporary objects;
assertions introduced with assert and constant valued expressions which are
introduced with a let are exactly the same as in expressions; we may also
introduce a temporary variable using the keyword var, which may then be
modified by the postcondition.

A bracketed postcondition may also be conditional, the form of which is similar
to that for a conditional expression. A conditional postcondition means that
the postcondition guarded by the first true expression should be satisfied. For
example, the following postcondition states we should add 1 to a variable x
whenever a Boolean flag \( z \) is true:

\[
(b): \; x! = 1; \; [1]
\]

If, as in this example, the final guard and expression is empty (without even the ";"), the meaning is that nothing should be changed at all if all the other guards are false.

4.3 Schema calls

Often the change that we wish to describe is at least partly described be the postcondition of another schema. We can describe this using the usual syntax to "call" the schema, with the difference that every object modified by the schema must be followed by a "!". For example suppose the we are writing a postcondition which occurs in a class that defines a schema \( 
!s \) that takes 2 parameters, and modifies self and the 2nd parameter; we can state that its postcondition should be satisfied as follows:

\[
!s add(x, y!)
\]

The "!" preceding the schema name means that the schema is being called as a member of, and modifies, \( \text{self} \). It can be regarded as following the absent \( \text{self} \) parameter.

4.4 Combining postconditions

There are two ways in which we may combine postconditions - in parallel and in sequence. Postconditions may be combined in parallel using either "&" or "", and "&" has higher precedence and may be used inside conditional postconditions without requiring brackets, otherwise they are identical. Two postconditions may only be combined in parallel if the objects they modify are disjoint. For example we may assign values to two integer variables \( x \) and \( y \) as follows:

\[
x! = 4; \; y! = 5
\]

or like this:

\[
x! = 4 \& y! = 5
\]

Postconditions may also be combined in sequence using then - the meaning is that first one postcondition should be satisfied, and then another. This is frequently of use when a postcondition is defined using other schemas, for example we used this postcondition earlier:

\[
\text{if add}(w) \text{ then remove}(w)
\]

Which first changes \( \text{it} \) by using the add schema, and then changes the result by using the remove schema.

4.5 Iterating over collections

It is often the case that we must satisfy some condition on all objects of a collection. We can state this using a forall postcondition. The syntax is similar to the forall expression, except the condition is replaced by a postcondition. For example, to state that each element of a sequence of integers \( \mathbf{s} \) should have its value increased by one, we could use:

\[
\text{forall } i : \mathbf{s}. \; \mathbf{s}[i]! = i + 1
\]

Each value of the bound variable defines a different postcondition to be satisfied; these conditions are combined in parallel, and so the objects modified by each one must be independent. In the example, \( \mathbf{s}[i] \) and \( \mathbf{s}[j] \) are different objects whenever \( i \rightarrow j \).

4.6 Doing nothing

Sometimes it is desirable to define a postcondition that states that nothing should be changed at all. This can be expressed using the keyword pass.

4.7 The general form of postcondition

In fact, all the forms of postcondition just described are shortcuts for the general form of postcondition:

\[
\text{change objects } \mathbf{objects} \; \text{satisfy condition}
\]

Where objects is a list stating what is permitted to change in the postcondition, and condition is the condition that must become true. Any objects that are listed as changed may appear in the condition either with or without a prime - the unprimed version refers to the object before the postcondition was satisfied, and the primed version is the object after satisfying the condition.

For example, the postcondition:

\[
x! + 1
\]

is a shorthand for:
change \( x \) satisfy \( x' = x - 1 \)

It is not often necessary to use this form of postcondition, as the shortcut forms can express many of the frequently required postconditions.

## 5 Abstract classes and methods

*Perfect* provides a very powerful framework for defining abstract classes. As well as supporting single inheritance, *Perfect* provides facilities for data refinement (re-implementing abstract data in more complex but more efficient ways). The structure of a class declaration is:

```plaintext
class Identifier =>
    optional inherits- or storable-part
    optional abstract-section
    optional internal-section
    optional confined-section
    optional interface-section
end;
```

Although many of the sections are optional, there must be at least a confined section or an interface section.

In this chapter, we will consider only simple classes with abstract and interface sections, so that our class declarations have the following format:

```plaintext
class Identifier =>
    abstract
    abstract declarations
    interface
    interface declarations
end;
```

### 5.1 The abstract section

The abstract section is introduced by the keyword `abstract`. It may contain:

- Variable declarations, describing the abstract model of the class;
- Invariant declarations, describing constraints on the abstract data;
- Abstract method declarations, describing functions and operations on the data.

Note that abstract methods are visible to the client (in that they help to describe the properties of the model), however they are not accessible outside the class (i.e. cannot be directly called).

As an example, consider the declaration of a class representing a bank account. The account has a name, a balance and an overdraft limit. We might begin the class as follows:

```plaintext
class BankAccount =>
    abstract
    var owner: string,
        balance, overdraftLimit: int;
    interface
        ...
end;
```

However, on reflection we may realize that this data model is too general because:

- we don't wish to allow the owner to be empty, otherwise we can't identify whose money it represents;
- as we don't require an account to have a minimum balance, it makes no sense to allow both positive and negative overdraft limits. We will choose to represent the overdraft limit as a positive quantity;

so we change the data model by constraining some of the types:

```plaintext
class BankAccount =>
    abstract
    var owner: those x: string :: #x -> 0,
        balance: int,
        overdraftLimit: nat;
    interface
        ...
end;
```

Now suppose we wish to state that an account can never exceed its overdraft limit. This can't be done using a simple constrained type because the constraint involves the simultaneous values of two data items, so we use a class invariant instead:

```plaintext
class BankAccount =>
    abstract
    var owner: those x: string :: #x -> 0,
        balance: int,
        overdraftLimit: nat;
    invariant balance -> (overdraftLimit);
    interface
        ...
end;
```

When checking whether a transaction is permitted, we may wish to talk about the maximum amount that can be withdrawn from the account. We can define an abstract function to calculate this value:
class BankAccount :=
abstract
  var owner: string := "",
  balance: int,
  overdraftLimit: nat;

invariant balance ≥ 0;

function availableFunds: nat
  := balance + overdraftLimit;

interface
end;

We chose to declare the invariant before the declaration of the 'availableFunds' function. This means that the function may assume the invariant, which in turn means that the result cannot be negative, so we have declared its result type as nat instead of int. Alternatively, we could declare the invariant after the function, in which case the function may not assume the invariant (so its return type must be int) but the invariant may refer to the function:

class BankAccount :=
abstract
  var owner: string := "",
  balance: int,
  overdraftLimit: nat;

function availableFunds: int
  := balance + overdraftLimit;

invariant availableFunds ≥ 0;

interface
end;

In summary, declarations of abstract methods and invariants may be interspersed; but abstract method declarations may only assume invariants already declared, and each invariant may only refer to methods already declared.

5.2 The interface section

The interface section comprises declarations of methods and constructors that are available to clients of the class. Variables may not be declared in the interface section. There are four types of method: function, operator, selector, and schema.

A function may take one or more parameters and yields one or more result values. For example, suppose we wish to enquire whether a given amount can be withdrawn from an account. We could add a function to perform this enquiry:

class BankAccount :=
abstract
  var owner: string := "",
  balance: int,
  overdraftLimit: nat;

function availableFunds: int
  := balance + overdraftLimit;

invariant availableFunds ≥ 0;

interface
  function canWithdraw(amount: nat): bool
    := amount <= availableFunds;
end;

Quite often, we wish to be able to retrieve the value of an abstract variable using a function. In that case, we can just redeclared the variable as a function:

class BankAccount :=
abstract
  var owner: string := "",
  balance: int,

  interface
    function balance;
  end;

A function may return more than one value, in which case names must be given to the returned values (see the Language Reference Manual for details).

An operator declaration is just like a function declaration except:

- Its name is not an identifier, instead it is one of a set of redefinable symbols or a pair of square brackets;
- It must be a class member (unlike functions and schemas, which can also be class nonmembers, or global, or local);
- It must be declared with either no parameters, a single left-hand parameter, or a single right-hand parameter.

A selector declaration is just like a function declaration except:

- Its name is either an identifier or a pair of square brackets;
- It must be a class member (unlike functions and schemas, which can also
be class nonmembers, or global, or local);
- It may only have a single return value;
- The return value must be a modifiable part of the current object (because
  a selector may be used to modify an object).

The classic example of a selector is the indexing selector on sequences (e.g.
the expression "myList[i]" may be used not only to retrieve an element as in
"rslt = myList[i]" but also to change its value as in "myList[i]! = rsLt").

Just as an abstract variable may be redeclared as an interface function if we
wish clients to be able to read its value, so it may be redeclared as an interface
selector if we wish clients to be able to read and write its value (although this
is normally not good practice because it prevents the abstract data being
re-implemented).

A schema is a method that changes the value of the current object and/or one
or more of its parameters. For example, let’s add a schema to withdraw an
amount from our account:

```kotlin
class BankAccount =
abstract
...
interface ...
  function canWithdraw(amount: nat): bool
    := amount <= availableFunds;
  schema withdraw(amount: nat)
    pre canWithdraw(amount)
    post balance! := amount;
end;
```

The specification of ‘withdraw’ states:
- It modified the current object (the exclamation mark before the schema
  name says this);
- It does not modify the parameter ‘amount’ (otherwise there would be an
  exclamation mark after it, before : nat);
- It may only be called when the precondition ‘canWithdraw(amount)’ is
  satisfied;
- On exit, the value of ‘balance’ is the old value minus ‘amount’ (we could
  have written ‘balance! = balance - amount’ instead).

### 5.3 Constructors

Instances of class BankAccount can only be created using a constructor. We
haven’t yet declared any constructors for this class, so we will fix this:

```kotlin
class BankAccount =
abstract
  var owner: those x: string := "0",
  balance: int,
  overdraftLimit: nat;

...;
interface ...
  build {
    owner: string,!
    !overdraftLimit: nat
    pre owner! := 0
    post balance! := 0;
end;
```

The reserved word build indicates a constructor and the parameter list is
enclosed in curly braces. This constructor takes two parameters and the ‘!’
before the parameter name indicates that the corresponding abstract variables
are directly assigned from them. We could have written instead:

```kotlin
build(own: string, limit: nat)
  pre own! := 0
  post owner! := own, overdraftLimit! := limit, balance! := 0;
```

### 5.4 Using post-assertions

A post-assertion may be appended to the declaration of any method or
constructor. Its purpose is to assert that some condition is satisfied as a direct
consequence of how the result (for a function, operator or selector) or
postcondition (for a schema or constructor) is defined. For example, after
making a withdrawal, we expect that the amount available to withdraw has
decreased by the amount withdrawn:

```kotlin
class BankAccount =
abstract
...;
interface ...
  schema withdraw(amount: nat)
    pre canWithdraw(amount)
    post balance! := amount
    assert self .availableFunds = availableFunds - amount;
end;
```

We use the keyword self to refer to the current object and the prime (single-
quote) mark to refer to the final value (self without the prime would refer to
the value before the schema is executed).

Post-assertions can be used as a check that the specification behaves as
expected (as we have done here) but are even more useful in the context of
5.5 Bringing it all together

Our bank account class now looks like this:

class BankAccount :=
abstract
  var owner: string := "",
  balance: int,
  overdraftLimit: nat;
function availableFunds: int
  := balance + overdraftLimit;
invariant availableFunds > 0;
interface
  function balance, overdraftLimit;
function canWithdraw(amount: nat): bool
  := amount <= availableFunds;
schema buildWithdraw(amount: nat)
  pre canWithdraw(amount)
  post balance! - amount
  assert self.availableFunds = availableFunds - amount;
build (owner: string, overdraftLimit: nat)
  pre #owner = "" 
  post balance! = 0;
end;

Because the method and constructor specifications in this example are simple, this can be directly compiled by a Perfect compiler. However, any implementation of Perfect will also be able to validate this class declaration, searching for inconsistencies. In this case there are four things to check:

- Any instance that can be built by the constructor will satisfy the class invariant;
- The 'withdraw' schema always preserves the class invariant;
- The values of 'owner' and 'overdraftLimit' will satisfy their respective type constraints in any instance built by the constructor.

5.6 Recursion

Sometimes the natural way to specify a method is recursively - that is, by defining it in terms of itself. In this case the method must define a variant - this can take several forms, the simplest of which is a non-negative integer value that always decreases when the method's specification refers to itself. The variant is introduced with the keyword decrease immediately before the function definition or postcondition. For example, we can define a factorial function like this:

```plaintext
function factorial(n: nat)
  decrease n
  := [n = 0];
  1;
  n * factorial(n - 1);
end;
```

Notice that when the specification refers to its own method, the variant of the called method is n - 1, which is less than its original value of n. Also, because the parameter is of type nat, the value of the variant can never be negative for any legal call. These two conditions guarantee that recursion will terminate after a finite number of nested calls.

6 Class Inheritance

6.1 Principles

Inheritance is a fundamental tool of object-oriented development. It provides two distinct facilities: re-use and polymorphism.

Re-use is the ability to use a single specification or implementation for more than one purpose. For example, functionality common to two different classes might be written once in a base class, from which the two classes inherit; or a class that is already in use may serve as a base for a new class that is an extension of the original.

Polymorphism is the ability for a specification or some code to work with a variety of different object types at run-time. Associated with polymorphism is dynamic binding, which allows a method call on an object to bind at run-time to a specification that is tailored specifically for the class of that object.

Although re-use and polymorphism are very useful, they both have their dangers:

- Code re-use may fail because the original developer did not foresee the context in which the code is re-used, nor has it been tested in any but the original context; or because the specification of the code is not sufficiently
precise for anyone other than its original developer to judge its suitability for the intended re-use.

- Polymorphism may give rise to errors because when clients call class methods, they necessarily make assumptions about what those methods do. If an object of a derived class is subsequently substituted in place of an object of the original class, it is necessary that the derived class methods satisfy the same assumptions. All too often, these assumptions are not documented; even if they are, the developer of a derived class may not be aware of them and may fail to override an inherited method where it is necessary.

Perfect allows safe re-use and polymorphism:

- Using Perfect, classes and methods are precisely specified, making it easy to determine their suitability for the intended re-use;
- Perfect supports tools that check that methods satisfy their specifications irrespective of the client context;
- Perfect distinguishes between contexts where polymorphism is permitted and contexts where an object of an exact class is required;
- Perfect supports tools not only check methods for correctness in the context in which they are declared, they also check correctness of methods in all classes into which they are inherited and not overridden.

6.2 Specifying inheritance

When declaring an abstract class, inheritance from a base class is specified by adding an inherits clause, for example:

```java
class SavingsAccount ~ inherits BankAccount
abstract ...
interface ...
end;
```

The class SavingsAccount inherits the abstract data model and all interface methods of BankAccount. The inherits clause may be followed by an abstract section, in which the data model may be extended, new invariants defined, and new abstract methods declared. For example, let us define a savings account to have a notice period but never an overdraft facility:

```java
class SavingsAccount ~ inherits BankAccount
abstract
  invariant overdraftLimit = 0;
  var daysNotice: nat;
interface ...
```

In this declaration:

- We have put a constraint on the inherited abstract model (i.e. the overdraft limit must be zero). Note that class SavingsAccount does not have automatic access to the abstract members of BankAccount, however in BankAccount we redeclared overdraftLimit as an interface function, so we can refer to it here.
- The constructor for SavingsAccount must inherit a value of the type of the class it inherits. Typically, this value is provided by a call to a base class constructor, as here. It must satisfy any invariant involving the inherited abstract model (in this example, overdraftLimit must be zero).

6.3 The confined section

It is often useful to declare methods and constructors that are accessible within a class and descendents of that class but inaccessible elsewhere. Perfect allows classes to declare such methods and constructors in an optional confined section that precedes the interface section. Confined methods are inherited by descendent classes.

6.4 Redefining inherited methods

Inherited methods (those in the confined and interface sections) may be overridden in the derived class:

```java
class SavingsAccount ~ inherits BankAccount
abstract ...
interface ...
  redefine function canWithdraw(amount: nat): bool
  ~ super canWithdraw(amount) & noticeGiven(amount);
end;
```

When overriding an inherited method:

- We used the redefine keyword to indicate that this is not a new method but a redefinition of an inherited method;
- The types of the parameters and the result must be the same as in the method we are overriding;
• If we give a precondition, it must be no less restrictive than the precondition of the method we are overriding. If we don’t give a precondition, we automatically inherit the precondition (if any) of the overridden method;
• If we give a post-assertion, it must be at least as restrictive as the post-assertion of the method we are overriding. If we don’t give a post-assertion, we automatically inherit the post-assertion (if any) of the overridden method.

We can prefix a method call with the word super to indicate we are referring to the method as it was defined or inherited in the parent class. This is most often used to refer to the overridden method from within the overriding declaration.

6.5 Using polymorphism

Suppose we wish to declare a variable, parameter or result that may be either a BankAccount or a SavingsAccount. We can accomplish this using a union:

```javascript
var myAccount: BankAccount || SavingsAccount;
```

Because all members of this union have a common base class (i.e. BankAccount), we can invoke all the methods of this common base class on the variable:

```javascript
... myAccount.canWithdraw ...
```

Frequently, we wish to express a type comprising the union of all classes derived (directly or indirectly) from some base. We use the keyword from to denote such types:

```javascript
var myAccount: from BankAccount;
```

Note that if we declared the variable ‘myAccount’ to be of type ‘BankAccount’ instead of from BankAccount, it would not be possible to assign a value of type SavingsAccount to the variable.

6.6 Using post-assertions with inheritance

When we override an inherited function, operator, selector or schema, we are redefining the result value or postcondition. However, in most cases, the original and overriding definitions will have some properties in common. For example, a schema might leave the current object in a state having some particular property.

To express such properties, we need to state them in the base class as post-assertions so that they are automatically inherited in derived classes. Even where a method is overridden and a new post-assertion given, the new post-assertion must imply the old, so correctness is maintained.

Using post-assertions is fundamental to describing the properties of methods in class hierarchies and hence proving programs are correct. Given a variable x of type from A’, if we call a member schema !doSomething on x (using the syntax x/doSomething), the only information we have about what the schema does is given by the post-assertion of the declaration of !doSomething in class A. In contrast x is of type A then we can assume the entire postcondition of !doSomething as declared in A.

6.7 Preventing inheritance and overriding

When declaring a class, we may prevent other class declarations inheriting from it by declaring it final, for example:

```javascript
final class SavingsAccount = ...
```

Declaring a class final has the side-effect of changing the type of self within member methods of the class. Normally, when a method is declared within a class A, the type of self within the method is from A to reflect the fact that the method is also present in all classes derived from A that do not override it. However, if class A is declared final, the type of self within A’s methods is just A.

We may also use final to prevent individual methods from being overridden, for example:

```javascript
final schema withdraw(amount: nat) ...
```

Used correctly, final methods confer the following advantages:

• A person reading the specification knows exactly what the method does without having to allow for the possibility that it is overridden;
• The system can use the full specification of the method when attempting to validate client calls, instead of just the precondition and post-assertion;
• They are a little faster to call at run-time because they are statically bound.

It is possible to override a (non-final) method and make the new definition final:

```javascript
redefine final function canWithdraw(amount: nat) ...
```
The methods of a `final` class are implicitly `final`.

### 6.8 Deferred methods and classes

Sometimes it is desirable to define a class that is not complete in itself but is used only as a base class from which to derive other classes. Such a class is called a deferred class in Perfect. It is declared like a normal class except that the entire class declaration is preceded by the keyword `deferred`.

A deferred class differs from a normal class as follows:

- It may contain deferred method declarations;
- It may not be declared `final`;
- The name of a deferred class is not a valid type name except immediately after the keyword `inherits` in a class declaration. However, its name preceded by `from` is a valid type.

A deferred method declaration has no definition but may have a precondition and a post-assertion.

As an example, suppose we wanted to redesign our bank account example using a deferred base class `Account`. We might proceed as follows:

```plaintext
defered class Account =
  abstract
  var owner: those x:string : x -> 0,
  balance: int;
  interface
  function owner, balance;

  deferred function canWithdraw(amount: nat, when: Time): bool;

  scheme balance(amount: nat, when: Time)
  pre canWithdraw(amount)
  post balance -= amount;

  build(owner: string)
  pre #owner -> 0
  post balance = 0;
end;
```

A class that inherits from a deferred class may define its inherited deferred members. This is just like overriding an inherited non-deferred member except that the keyword `define` is used instead of `redefine`. So we could go on to create a class representing a check account as follows:

```plaintext
class CheckAccount `= inherits Account
  abstract
  var overdraftLimit: nat;
  interface
  function overdraftLimit;
  define function canWithdraw(amount: nat, when: Time): bool
  pre amount <= balance + overdraftLimit;
  build(owner: string, overdraftLimit: nat)
  pre #owner -> 0
  inherits Account(owner);
end;
```

A non-deferred class that inherits from a deferred class must define all the inherited deferred members.

### 6.9 Deferring a precondition

When redefining an inherited method, it sometimes happens that we need to change the precondition in such a way that meeting the requirement that the overridden precondition implies the new precondition is not directly possible.

The solution in these cases is to use a function to represent all or part of the precondition and to declare the method precondition in terms of that function. Usually it will be the case that the function represents some clearly recognizable abstract property of the class. When we override the original method in an inherited class, we can also override the function.

In the bank account example in the previous section, we declared the precondition of the schema "canWithdraw" to be the result of the function "canWithdraw", which can be redefined in descendent classes to take account of any special conditions required to permit withdrawals from particular classes of account.

Sometimes we wish to declare in a schema post-assertion that following application of the schema, the object will have some particular abstract property. Just as for preconditions, the exact nature of the property concerned may vary between classes in the hierarchy. Once again, the property can be represented by a function that is redefined in descendent classes where necessary.

### 6.10 Using ghost methods to represent abstract properties
When defining functions to represent abstract properties, it sometimes happens that the property is only used in declaring preconditions, postconditions, class invariants etc. and code for the function will never be executed. Perfect allows such a function to be declared ghost meaning that no code is to be generated for it. An error will be reported if a ghost function is referred to in a context that requires its evaluation at execution time.

Ghost operators, selectors and schemas are also permitted.

### 6.11 Calling methods from class invariants and deferred class constructors

It is occasionally useful to refer to a confined or interface method in the declaration of an abstract method, an invariant of the same class or in the specification or implementation of a constructor for a deferred class.

Normally, calls to confined and interface methods are not permitted in these contexts because:

- A class invariant must be independent of context, i.e. the same for a standalone instance of the class or for the inherited part of an instance of any descendent class. Therefore, calls to methods that could be overridden are not permitted. Even a method declared final could call a non-final method.

- A constructor for a deferred class is used only in the inherits parts of constructors of its descendent. Since the complete object has not yet been constructed, it cannot be permitted for a constructor to call any method that is deferred or that might call a deferred method.

- Abstract methods are part of the abstract specification and must be independent of inheritance context.

Where a method is needed both as part of the interface and to be available to call in one of the above contexts, an early confined or interface method may be declared. An early method has the following properties:

- It may not be deferred or overridden (i.e. it is implicitly final);

- It may not explicitly or implicitly refer to the current object self other than as the current object in calls to other early methods.

All abstract methods that precede an abstract class invariant declaration are implicitly early.

### 7 Copying, equality and ordering

#### 7.1 Semantics of assignment and parameter passing

Most programming languages define the semantics of variable assignment and parameter passing in one of two ways:

- **value semantics** means that when a value is assigned to a variable or passed as a parameter, the value is copied. If one variable is assigned from another, the values of the two copies are independent of each other, so that changing one does not affect the other.

- **reference semantics** means that when a value is assigned to a variable or passed as a parameter, a reference to the value is copied. If one variable is assigned from another, both variables refer to the same copy of the value, so that changing one causes the values of the other to change also.

Some programming languages (e.g. Java and Eiffel) use value semantics for built-in basic types and reference semantics for user-defined types. Others (e.g. C++) use value semantics but allow explicit reference types to allow reference semantics to be used.

The problem with unconstrained use of reference semantics is the potential for errors caused by aliasing (i.e. two supposedly independent variables referring to the same value so that a change to one variable inadvertently changes the other). It is very hard to show that a program is correct if the underlying language uses reference semantics by default.

The reason that so many languages use reference semantics by default is the perceived high cost of copying complex objects, together with the variable storage requirement of polymorphic variables obeying value semantics. Of course, there are many situations in which deliberate aliasing is required; but there are many more in which value semantics are more appropriate.

Perfect adopts value semantics, with the option of declaring variables and parameters of reference type where reference semantics is needed. So, when one variable is assigned from another, the system behaves as if a copy of the entire value of the variable is made; similarly if a variable is passed as an input parameter.

In reality, a Perfect compiler and its run-time support system will avoid copying where possible; and even when copying is needed, generally it is only necessary to copy part of the complete object.
7.2 Equality

An equality operator is defined automatically for every class declared in Perfect. Two values are deemed to be equal if and only if at run-time:

- They have exactly the same type; and
- All their abstract data variables have correspondingly equal values.

Two values of reference type are equal if and only if they refer to the same object.

7.3 Ordering relations

Also defined for every class is an ordering operator "~" pronounced "rank". The rank operator may be applied between any two objects of the same type and yields a value of type rank, which is an enumeration of the three values above, same and below.

If a user-defined rank operator is not provided in a class declaration, the class inherits the rank operator from its ancestor. If the class has no ancestor, the system will define a rank operator (which may or may not always yield same regardless of the operand value). The definition of the rank operator must define at least a partial ordering; that is to say:

- a = b ==> a ~ b = same
- a ~ b = same ==> b ~ a = same
- a ~ b = same ==> c ~ a = c ~ b
- a ~ b = above <==> b ~ a = below
- a ~ b = above & b ~ c = above ==> a ~ c = above
- a ~ b = below & b ~ c = below ==> a ~ c = below

The rank operator may be used to compare objects of different type by casting both objects to a united type, or from a common base class. In this case rank is guaranteed not to return same for objects of different type. Any user-defined rank operator which does not satisfy this condition will be refined by the system in order that it does.

Comparison operators are defined as follows:

- a > b = a ~ b = above
- a < b = a ~ b = below
- a ~ b = ~a > b (which is the same as: a ~ b = same | a ~ b = below)
- a ~ b = ~(a > b) (which is the same as: a ~ b = same | a ~ b = below)

In addition, if the user declares the rank operator to be total, the operator must only return same for objects with equal value. In this case operators >= and <= are defined, and have the same meanings as ~< and ~> respectively.

In general, if a user class is to be used as the base type for a set or bag or for the domain of a mapping, a rank operator should be defined in order that the system may use sorting to improve efficiency.

8 Implementations

When abstract class declarations were introduced earlier in this text, we stressed the importance of declaring the simplest possible abstract model of the stored data, avoiding complexity and redundancy.

Although it will often be the case that code generated directly from the abstract model is efficient enough for the final product, there are cases where the volume of data is large and the sorts of access required cannot be implemented efficiently using the standard Perfect collection types.

There are also situations where some function of the abstract data needs to be evaluated far more frequently than the data is changed and it would be more efficient to store the result either when the object is created or modified, or the first time it is needed; subsequently the stored result can be retrieved instead of evaluating the function. The stored result is redundant data and belongs not in an abstract model but rather in an implementation.

Another example of redundant data is the creation and maintenance of an index to speed up certain kinds of access to a collection of data.

Perfect allows abstract data variables to be represented by more complex implementations, or to be supplemented by redundant data - all without having to abandon the original simple abstract model.

8.1 The internal section

Perfect allows class declarations to include an additional internal section. The primary purpose of this section is to provide for re-implementation of the abstract model. The internal section is placed after the abstract section but before the confined and interface sections.

The internal section may contain the following types of declarations:
• Variable declarations to describe additional or replacement data;
• Class invariants involving the variable declarations;
• Retrieve functions to describe which abstract variables are replaced and how;
• Methods to assist in the re-implementation of other methods of the class.

Internal variables, class invariants and ordinary methods are declared using the same syntax as for the abstract section.

A retrieve function declaration is a special function declaration with the following properties:

• Its name is the same as the name of an abstract variable;
• It has no parameter list;
• It has no return type;
• It has no modifiers (early, ghost etc.);
• Its result must be explicitly defined (i.e. using "^=" and not satisfy).

The meaning of a retrieve function is that the corresponding abstract variable should not be stored (it has been replaced by internal variables) but its value could in theory be determined by evaluating the result expression.

As an example of data re-implementation, consider our earlier dictionary class:

```java
class Dictionary {
  abstract
  var words: set of Word;

  end;
}
```

Many words are nouns with regular plurals, so instead of storing both a word and its plural, we might decide to store the singular form together with an indication that the regular plural is also a valid word. For the moment we will use this mechanism only for words whose plural form is the singular form with a single letter 's' appended. We can re-implement the data by dividing the words into two sets: words that have this regular plural form, and words that don't. For efficiency reasons, we don't want to represent the same word more than once (e.g. by storing it in both sets), so we will add a class invariant. Here is the same example with data re-implementation and a retrieve function:

```java
class Dictionary {
  abstract
  var words: set of Word;
  internal
  var wordsWithoutPlural, wordsWithPlural: set of Word;

  invariant
    // Conditions to avoid representing the same word twice

  end;
}
```

Also worth noting in this example is the "is Word" cast used in the retrieve function. This is required because Word is a constrained type; the cast asserts that the result of adding "s" to a Word is another Word, and so we may combine these sets of Words.

Retrieve functions and the corresponding abstract variables are automatically ghost. This means that it is necessary to re-implement all non-ghost methods whose specification includes a postcondition or result value that refers to an abstract variable redeclared as a retrieve function.

The system-defined equality operator for a class containing a retrieve function must be declared ghost unless it is re-implemented. We do this as follows:

```java
ghost operator ==(arg);
```

In the declaration of equality we do not need to declare the operand type, the return type, or give a specification, as these are all predefined by the system.

In the above example, declaring equality to be ghost means we cannot compare two dictionaries or declare a set or bag of dictionaries.

### 8.2 Re-implementing confined and interface methods

Confined and interface methods may be re-implemented using a `via ... end` construct after the postcondition or result value (but before the post-assertion, if any). Between `via` and `end` must be one or more statements. We will illustrate some of these statements by re-implementing the interface methods of our dictionary example:

```java
class Dictionary {
  abstract

    // Implementations of interface methods

  end;
}
```
The abstract specification of the result value has not been changed, but since it cannot be directly evaluated because "words" has been defined as a retrieve function and is therefore ghost, we have added an implementation. The implementation here comprises a single value statement (which yields the value of the implementing and therefore behaves like a return statement in this context). The expression following value is a definition of the returned value in terms of the actual stored data.

Turning now to the schema to add a new word, this is somewhat more complicated than before because we may add a word and its plural in either order. We must also allow for words that can not only have "s" appended but also "ss" (for example, "a"). Here is a suitable re-implementation:

```plaintext
var words : set of word;
internal
  var wordsWithPlural : set of word;
...
// rest of internal section as before
interface
  ghost operator += (arg);
function check(w : word) : bool
    := w in words
via
  value w in wordsWithoutPlural
  | w in wordsWithPlural.
end;

end;
```

Again, the postcondition of the schema has been left alone but an implementation has been provided. Two sorts of statement have been used here:

- The conditional statement, comprising the keywords if ... fi surrounding a sequence of guarded statement lists. The expression in the square brackets is called a guard and describes the condition under which the following statement list will be executed. The final guard here is an empty one (i.e. an empty set of brackets) and represents the "else" case.
- The postcondition statement, which is just a postcondition. Here we are only using assignment postconditions.

Similarly, the remove schema and the constructor of our example must be re-implemented to operate on the stored data.

### 8.3 Re-implementing abstract methods, equality and ordering operators

Abstract methods that are not declared ghost may also be re-implemented, however the implementation is placed in the internal section. The system-defined equality operator and the comparison operators ">" and "<" may also be re-implemented. For example, in our dictionary we might want to re-implement equality so that it is not ghost, allowing us to compare dictionaries. The re-implementation must not change the specification (i.e. two dictionaries must compare equal if and only if the abstract data variable "words" is the same in each). So we not only have to consider the case where the dictionaries have identical internal data, we also have to consider cases where the same words are represented differently (for example, the combination of words "a", "as" and "ass" may be stored in two different ways). Here is a possible re-implementation that tries to be fairly efficient by only looking in detail at the differences between the internal data:

```plaintext
class Dictionary =
...
internal
  operator += (arg)
via
    let w1 := wordsWithoutPlural ++ arg.wordsWithoutPlural;
    let w2 := arg.wordsWithoutPlural ++ wordsWithoutPlural;
    let w1p := arg.wordsWithPlural ++ wordsWithPlural;
    let w2p := arg.wordsWithPlural ++ wordsWithoutPlural;
    value (forall x : w1 ++ w1p : arg.check(x))
      & (forall x : w2 ++ w2p : check(x))
      & (forall x : w1p ++ w2p : check(x ++ "ss")
```

Notice that when re-implementing an abstract method, we do not give a result type, precondition or postcondition (since these are unchanged); the parameter list (if present) is followed directly by the implementation.

The equality operator (and the rank operator `~==`) is special in that we do not specify the type of the parameter; it is automatically given the type of the current class.

9 Statements

Statements are used in Perfect to describe implementations. The principle types of statement are as follows:

9.1 Declarations, let-statements and assertions

We have already seen let- and variable declaration statements and assertions, as these may occur in a bracketed expression or postcondition. These statements are also legal in implementations, as are declarations of classes, methods and properties.

Local variables may be given an initial value when they are declared by following the declaration by in initial value, for instance, to declare a local variable i of type nat with an initial value of 0:

```plaintext
var i : nat = 0
```

If this form is not used, all local variables must be initialized before they can be used.

9.2 Postcondition statements

As we have already seen, a statement may take the form of a postcondition to be satisfied. In this case, the postcondition itself may be implemented using a further via ... end.

This is the only form of statement that can actually change the state of objects - all other statements serve to control which statements are executed. That is, the only way to change the state of an object in Perfect is to use a postcondition.

9.3 Conditional statements

We have also already seen the conditional statement. It takes the form of the keyword if followed by a number of guarded statement lists, and is terminated by the keyword fi.

The last guard in the list may take one of two special forms - the usual "else" form consisting of an empty guard followed by a colon and then a statement list means that list is executed if all other guards are false; the other form consists of an empty guard immediately followed by fi (i.e. with no colon), and means "if all other guards are false, do nothing".

9.4 Completers

Perfect has two statements which serve to terminate an implementation: the value completer, which is used when the implementation must return a value (e.g. an implementation of a function), and the done statement which ends an implementation that changes the state of objects (e.g. an implementation of a schema).

The value statement consists of the keyword value followed by an expression representing the value to be returned. All implementations which need to return a value must finish with a value statement, or a conditional statement in which every branch ends with a value statement.

An implementation for a state change (e.g. a schema) implicitly terminates at the final end, so a done statement at this point is not mandatory.

9.5 Loops

The most complex construct in Perfect is the loop. This looks rather different from loops in other languages because not only must it state what is to be performed on each loop iteration and when to terminate the loop, it must also describe the purpose of the loop. The general form of a Perfect loop is as follows:

```plaintext
loop
  var ... ;
  change ... ;
  keep ... ;
  until ... ;
  decrease ... ;
```
The loop is introduced with the keyword **loop**: this is followed by any declarations of variables which are local to the loop (for example loop counts), which will usually be initialized at the point of their declaration, and following the keyword **change** a list of non-local variables that the loop may also change. The loop may not change any non-local variables other than those listed in the **change** section.

The key to writing **Perfect** loops is the expression following the keyword **keep**. This is the **loop invariant** - a condition that must be true when the loop is first entered, remain true on every iteration of the loop, and be true when the loop terminates. It should describe what the loop is trying to achieve.

The expression following the keyword **until** is a Boolean expression which describes the condition for the loop to terminate. The expression following the keyword **decrease** is a variant, similar to the recursion variant which we have already met - it is typically a non-negative integer expression which decreases on every iteration of the loop. The remaining part of the loop comprises the **loop body** - the sequence of statements which are executed in each iteration of the loop.

The best way to understand all this is with a simple example. Recall the recursive factorial function defined earlier - it is more efficient to use a loop instead of recursion, so we might implement the function as follows:

```plaintext
function factorial(n: nat): nat
decrease n
  := { [n = 0]:
        1,
        n * factorial(n - 1)
      }
via
var tot: nat := 1;
loop
  var nn: nat := 0;
  change tot
  keep tot = factorial(nn)
  until nn = n
  decrease n = nn
  nn := nn + 1, tot = nn
end;
value tot
end;
```

Let us look at this implementation in detail. Firstly a variable is declared to hold the currently calculated total, and it is initialized to the value 1. Then comes the loop - we declare a local loop counter variable which is initialized to 0, and declare that the loop is going to change the variable tot.

The loop invariant states what this loop is doing - in this case we are calculating increasingly large factorials. At any given loop iteration the total is equal to the factorial of the current loop counter, thus when the loop counter reaches the input parameter, we have calculated the desired value. Note that both `n` and `nn` appear primed in the loop invariant. This means that we are referring to their values at the beginning of each iteration; if a variable appears unprimed, this refers to its value before entering the loop. Variables local to the loop must always appear primed, as they did not exist before entering the loop.

As would be expected from the invariant, the **until** condition states that we stop when the loop counter is equal to the parameter `n`, and the variant states that the difference between the parameter and the loop counter is decreasing. Again, the loop counter must appear primed in both of these sections.

The loop body the simply states that on each iteration, the loop counter should have its value increased by one and the total should be multiplied by the new value of the loop counter. Here the priming of the loop counter means we are referring to its value after this postcondition has finished, i.e. this value is one more than its unprimed value.

In the final statement of the implementation we return the calculated total.

## 10 Summary

We have covered enough in this introduction to allow the development of fairly complex **Perfect** specifications and code - how to specify new abstract classes and methods, how to inherit specifications from parent classes, and how to write efficient implementations. However, we have of course not covered everything the language has to offer.

The **Language Reference Manual** contains a full specification of the **Perfect** language. Of particular interest will be the description of the **environment**, which describes all the built in methods for displaying output, reading and writing files and other interactions with the outside world.