4.900

Prolog++ Reference

by Dave Westwood

The contents of this manual describe the product, *Prolog++* version 2.0, and are believed correct at the time of going to press. They do not embody a commitment on the part of Logic Programming Associates (LPA), who may from time to time make changes to the specification of the product in line with their policy of continual improvement. No part of this manual may be reproduced or transmitted in any form, electronic or mechanical, for any purpose other than the licensee's personal use without the prior written agreement of LPA.

Edition 2

13 July, 2004

Contents

This page is intentionally blank

Welcome

Welcome to this latest version of Prolog++, a state-of-the-art development system based on the best features from two advanced technologies, artificial intelligence (AI) and object-oriented programming systems (OOPS).

The relative strengths of Prolog and object systems are complimentary, which gives rise to a "dovetail" join rather than a "bolt-on". The underlying philosophy of the two technologies have a similarly good fit; the ideas of type-free programming and polymorphism fit well as do the ideas of dynamic data structures and run-time generation of new objects.

All in all this gives rise to an exciting and powerful product, Prolog++. We hope you enjoy using it and wish you much success in building your applications.

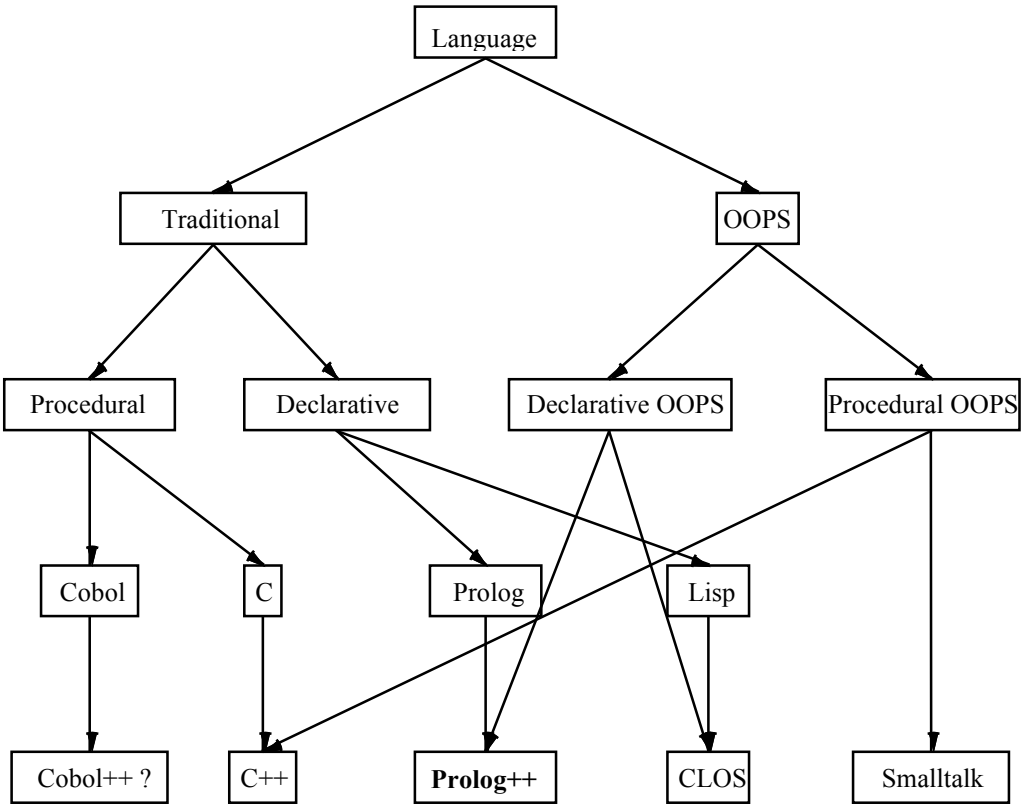
We at LPA are always keen to hear from our customers and users of our products, so if you have any criticisms, bug reports, comments on the manual, grumbles, congratulations, etc. you know what to do.

Clive Spenser,
LPA Product Development Team,
Studio 4,
The Royal Victoria Patriotic Building (RVPB),
Trinity Road,
London, SW18 3SX

Tel	+44 181-871-2016
Fax	+44 181-874-0449
email	support@lpa.co.uk

An Object-Oriented View Of Prolog++

The following diagram illustrates the relationship between Prolog++ and other programming languages.



Programming languages can be roughly split between the traditional approach which describes a program as a collection of algorithms and procedures which manipulate data, and the object-oriented approach which maps data directly onto objects which have the capability of communicating with each other.

The traditional approach can be further classified into procedural languages, such as Cobol, C, etc,. and declarative languages such as Prolog. Each approach has its benefits and its drawbacks.

One of the first object-oriented languages to be developed was Smalltalk, in which everything (including all integers, reals and identifiers) is considered to be an object. The emerging languages, such as C++, combine features of traditional languages with the object- oriented approach. One might say that C++ inherits characteristics of both approaches.

Prolog++ follows the lead of C++, but whereas C++ combines a procedural language with OOPS, Prolog++ combines a declarative language with OOPS. One might say that Prolog++ inherits the problem solving and database characteristics of Prolog, and also inherits the sound methodology of an OOPS approach.

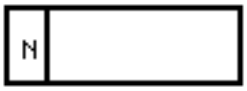
How To Use This Manual

The basic layout of this manual has 4 parts :-

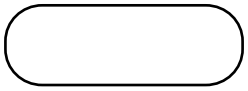
Part 1	Chapters 1-3	A general discussion about OOPS
Part 2	Chapters 4-8	Case studies using Prolog++
Part 3	Chapters 9-11	Technical specification of Prolog++
Part 4	Chapters 12-17	Operational features of Prolog++

This manual is intended to have a dual rôle, both as a general introduction to OOPS and Prolog++, and as a reference manual for the Prolog++ implementation.

The following sections outline how to read each of the major divisions. The accompanying diagrams illustrate the inter-dependency of chapters, using a pictorial convention :-



Chapter *N* of this manual.



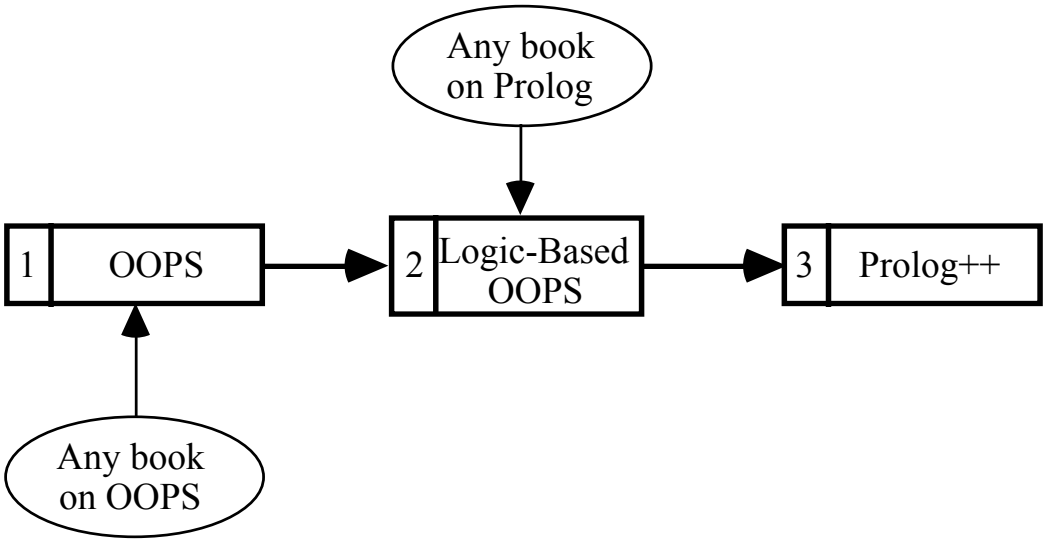
Additional reading material.

Advised route through the chapters.

Recommended material before reading a chapter of the manual.

Supplementary material.

Part 1 Object-Oriented Programming Systems

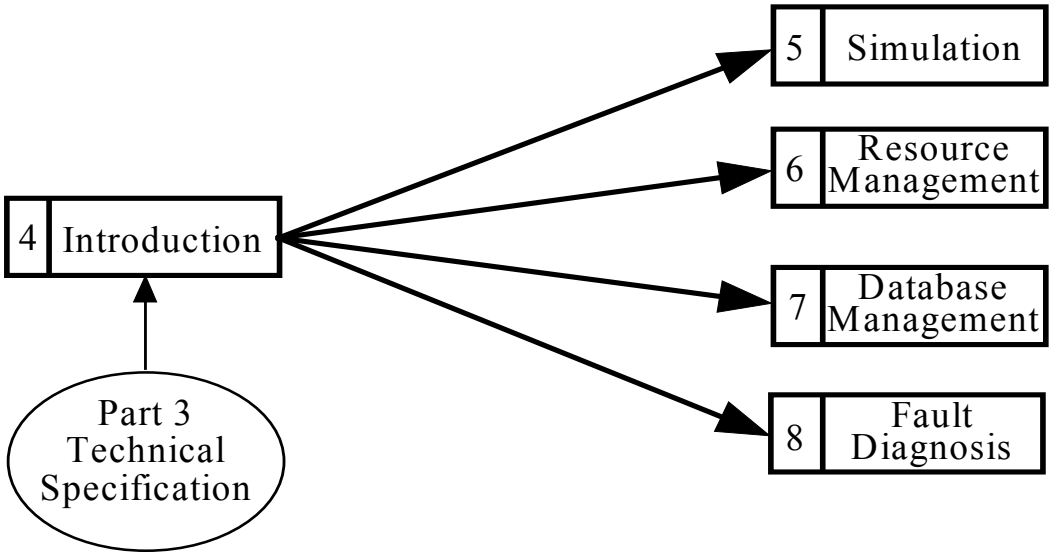


The chapters of this part are intended to be read sequentially, starting at chapter 1. Those readers who are already familiar with the general concepts of OOPS can begin at chapter 2, and those who have an understanding of both Prolog and OOPS might like to go straight to chapter 3.

The reader is advised to accompany this manual with any general book on OOPS, although this is not strictly necessary. Before reading chapter 2, however, a basic understanding of logic-based languages, and Prolog in particular, is strongly recommended.

Each chapter can be thought of as a specialization of the previous chapter, starting at general concepts (chapter 1), through more specific, logic-based concepts (chapter 2) and finally to Prolog++ (chapter 3), an actual instance of OOPS.

Part 2 Case Studies



The case studies of part 2 cover four different topics. It is not necessary to be familiar with any of these topics, although a basic understanding will assist the reader in the formulation of each problem.

At this stage, no details about the technical specification of Prolog++ have been given. The reader may either accept faithfully the solutions given, or refer to the technical part which follows for clarification of particular points.

Part 3 Technical Specification

The technical part consists of three chapters, consisting of a specification of the language, the two pre-defined classes and the interface between Prolog and Prolog++.

Part 4 Operational Features

The chapters of this part can be read in any order whatsoever, since they cover completely different topics. This part should only be read after the technical specification of Prolog++ given in part 3.

The chapters on optimization, saving and loading reference certain aspects of the underlying Prolog implementation.

Part 1 - Object-Oriented Programming Systems

Chapter 1 - General Concepts Of OOPS

The object-oriented approach to computer programming has its own terminology. In this chapter we list the key concepts, and explain the ideas they represent.

Definitions

We start by looking at what the terminology surrounding OOPS actually means. These definitions, whilst generally true of most OOPS environments, are not intended as the only interpretation.

Abstraction

Abstraction is the process of generating an abstract or outline of a complicated entity.

Encapsulation

Encapsulation is the process of gathering together and isolating, within its outline, all aspects of an entity.

Class

In OOPS, the process of encapsulation results in the formation of classes. All code and data representing an entity is found within a unique structure, namely its class.

Class Hierarchy

A class hierarchy refers to the connecting together of classes, in which more general classes appear towards the top of the hierarchy and more specific classes towards the bottom.

Inheritance

Inheritance is the ability of a class to receive some of its characteristics from more general classes which appear higher up the class hierarchy.

Instance

An instance refers to an individual incarnation of a class. A class instance is sometimes called an object.

Message

A message is the means by which instances communicate with each other.

Abstract

A class abstract defines the range of messages which an instance of that class can handle.

Method

A method is a procedure whose purpose is to handle messages sent to instances of the class in which it is defined, or a more specific class lower down the class hierarchy.

Attribute

A class attribute corresponds to a specific characteristic of the entity which the class represents.

Public

A public attribute or method is one to which all other classes have access.

Private

A private attribute or method is one that may only be accessed by the class in which it is defined.

Polymorphism

Polymorphism is the ability to define the same method in different classes.

OOPS

OOPS is based on the use of separate entities which communicate with each other to form a system.

There are two main aspects to OOPS: organisation and communication.

Organisation is accompliched through class hierarchies and communication is carried out through the sending of messages between instances of those classes.

Conventional programming generally concentrates on procedures and procedure calls. Data is somewhat secondary and is passed around as arguments to procedures. In OOPS classes have both the procedures (methods) and the data (attributes) held locally. These have equal status within the class.

Classes and Instances

A class is a template which defines the general characteristics of its instances and those of any of its sub-classes, sub-sub-classes, etc..

A class contains methods for handling messages sent to instances of that class and attributes for storing data. The methods of a class handle the range of messages an instance of that class is capable of responding to.

A class does not have access to the internal mechanisms of other classes which can only be accessed through the sending of messages. Classes do not need to have all of their methods available publicly. Some methods and attributes can be private. The abstract of an class contains all the names of its public methods and attributes. This provides a complete list of all the external messages that an instance can handle.

Abstraction

Abstraction in OOPS refers to the process by which the definition of a class is summarised into a few key concepts.

Data abstraction is a definition of the interface to a data structure.

When using classes it is not necessary for us to know their mechanics, only their key concepts and intended usage. In this way the key concepts can be used as an interface to the classes allowing them to be re-used, like building blocks, in different programs.

To illustrate this, consider a knight and a bishop from the game of chess.

We know that they both have similar characteristics, such as how they move, what shape they are, what their initial positions are, etc.. The differences between these two chess pieces lie in the actual *description* of how they move, what shape they are, what their initial positions are and so on.

For example, to define the moves of a knight and a bishop, we could state:

> *To move a knight, move it two squares horizontally or vertically in either direction then one square vertically or horizontally in either direction as long as the resultant position is legal.*

and:

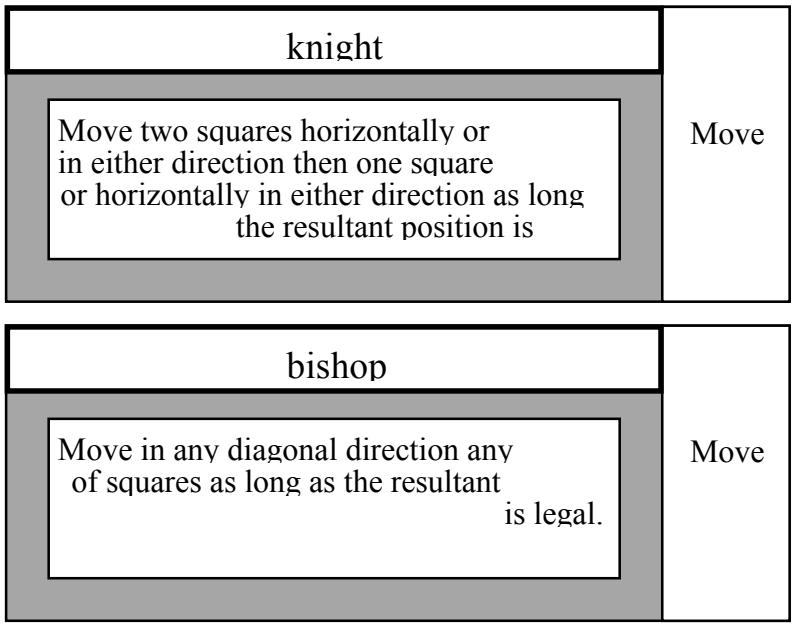
> *To move a bishop, move it in any diagonal direction any number of squares as long as the resultant position is legal.*

When moving one of the pieces it is natural for us to want to say:

> "Move the knight or the bishop (*I do not care how you do it*)"

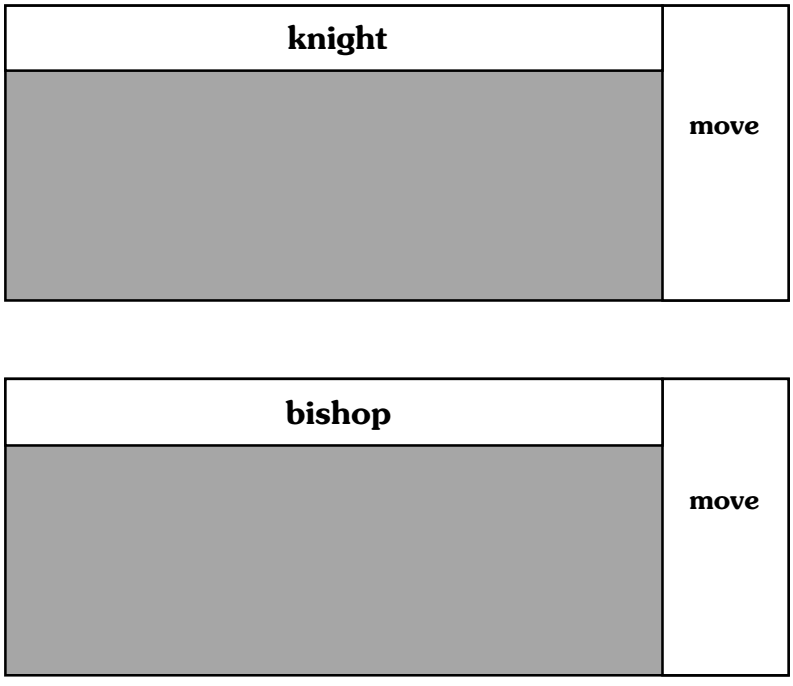
as opposed to having to say:

"Move the knight two squares horizontally or vertically in any direction then one square vertically or horizontally in any direction...".



The diagram above shows the knight and bishop pieces, with the procedure for moving each contained in the box of text. On the right of the diagram are the abstracts of the pieces, which define the range of messages to which the piece may react.

The pieces are actually moved by simply telling them to *move*. The capability of being able to move the knight, without having to be concerned about the actual mechanics of how a knight moves, allows the use of a more modular or distributed approach to programming.



We can think of the inner mechanisms of the pieces as 'black boxes', which allows us to use the objects by just referring to their abstract.

Another facet of abstraction is in the construction of class hierarchies.

Within the definitions of *move* for both the knight and the bishop we referred to the concept of a legal position. We can abstract this concept of a legal position, and define it in a more general class called chesspiece. In the chesspiece class we place all the concepts common to any chesspiece, not just knights and bishops.

We would then say that both the knight and bishop were specialisations of the chesspiece class and these would then share the definition of a legal position.

One advantage of abstraction is that it allows classes to be easily re-used in different programs. This is because classes are self-contained entities, and all that needs to be known about a class are the abstractions of its internal mechanisms.

For example, the knight may be used both in a chess program and in a 'knights tour' program. We only need to know the names of the methods the knight uses, such as *move* etc.

Encapsulation

Encapsulation isolates all the aspects of a class within its outline. This protects the internal routines of the class and thus guarantees its functionality. Through encapsulation the interface between one class and another is unambiguous.

Message Passing

Class instances communicate in OOPS by sending messages. When a message is received by an instance it must have the appropriate message handler (method) to deal with the message.

Methods generally take the form of a set of procedures, defined at the class level, which are invoked by the name of the message.

We talked earlier about telling a knight or a bishop to move. Such a directive would be implemented in an object-oriented system by sending the message *move* to either a specific knight or a specific bishop.

Polymorphism

Polymorphism literally means 'many forms' and refers to the ability to use a single name for different methods in different classes. This allows us to send the same message to several different objects, each having their own set of methods for handling the message.

As an example, we saw earlier that the same message, *move*, could be used for two different chesspieces the *knight* and *bishop*, to trigger two different reactions.

Polymorphism is an extremely powerful tool for generalising the name of a task across many different types of class, each having their own definition of that task.

Inheritance

We have seen that a class may be defined as part of a class hierarchy, and that attributes and methods may filter down to other classes through the hierarchy.

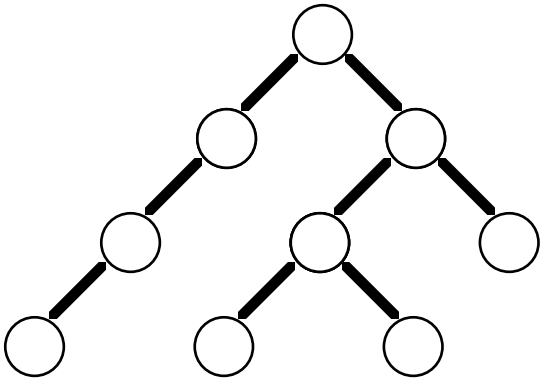
The mechanism for this distribution of information is inheritance.

Inheritance has two advantages. Firstly, it avoids unnecessary duplication of information. Secondly, because the code for a class is re-used by its sub-classes, the integrity of the system is easier to maintain. Once a method has been shown to be reliable that reliability is propagated down through the class hierarchy.

An example of inheritance is the shared definitions of the knight and the bishop, such as the definition of legal position, which they both inherit from the chesspiece class.

Singular Inheritance

Singular inheritance corresponds to a property of the class hierarchy in which each class is limited to at most one parent class. This property guarantees that the structure is a strict hierarchy, with a unique path (or branch) between any two connected instances.



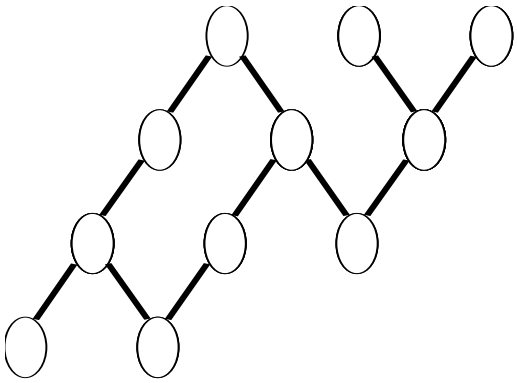
This diagram shows a singular inheritance class hierarchy, where each instance is represented by a circle and inheritance links are shown as dark lines.

Each class has at most one parent class.

When a message is sent to an instance whose class definition does not contain the appropriate message handler, a search is made of the hierarchy. The search begins at the instance's class where the message was originally sent and proceeds upwards until a class is found that can handle the message.

Multiple Inheritance

Multiple inheritance relaxes the parenthood constraint to allow for multiple parents of a class. Multiple inheritance leads to a tangled hierarchy (or directed, acyclic graph) rather than a strict hierarchy.



This diagram shows multiple inheritance, where a given class may have one or more parents.

Binding

Binding is the mechanism by which messages are associated with their handlers. In traditional programming languages a procedure call is associated with a procedure definition simply by name, and the link between them forged by the compiler. In an OOPS world with polymorphism there is a binding process to establishes the links since the same message can potentially be handled by several alternative handlers.

Early Binding

The early binding of a message to a particular handler is performed by the OOPS compiler, and corresponds exactly to the compiling techniques of traditional languages.

Late Binding

The late binding of a message to a particular handler is performed at run-time, and is determined by the exact context in which the message is sent.

Data-Driven Programming

The paradigm of data-driven programming is a relatively new phenomenon. Traditional languages such as C, Pascal and even Prolog view a program as a collection of procedures and statements. The flow of control can be completely determined by the way procedures, and to some extent program statements, call one another. With data-driven programming, however, this scenario is turned on its head.

A collection of data-driven procedures, called daemons, may be written which interrupt the main process. These are not connected to each other and, as with interrupts, there is no direct flow of control between them. Instead, daemons lie dormant and are only activated by certain events occurring. Of course, the execution of a daemon may itself cause

another event which triggers another daemon, possibly resulting in a cascade of daemons being triggered.

For example, suppose we have developed an object system which describes the logical relationship between chess pieces and the chess board. It will involve such things as the initial configuration of the board, which moves are legal and perhaps some game playing skills. This logical view of the game of chess can then be integrated into a graphics system by introducing daemons which react to the movement of pieces. The daemons will be triggered when a logical move takes place, and will drive graphical operations to reflect such moves on a screen.

Chapter 2 - A Logical Approach To OOPS

Having looked at the terminology that surrounds OOPS, we will now discuss how a logical approach to OOPS is beneficial.

Prolog as a Logic-Based Language

Prolog is a declarative language. This means that the formal definition or declaration of a problem may be used as a functional program to solve that problem. In Prolog we declare the logical relationships of a given problem domain rather than state a step-by-step procedural recipe for solving the problem. This is useful as often we know various aspects of the problem but very little about how to find a solution. Prolog will attempt to find a solution for us given the information about the problem.

The way Prolog does this is by using a built-in inference engine, which automatically infers the solution to a given query using the facts and rules as defined in the program.

Prolog programs consist of facts and rules, which are referred to as clauses. A fact consists of a single assertion with no conditions: a fact is always true. A rule consists of a goal, whose truth is dependent upon a set of conditions or sub-goals. The goal of a rule is referred to as its head and the sub-goals as its body.

Prolog attempts to prove a goal by using its backward chaining inference engine to match the initial goal with either a known fact or the head of a rule. If the goal is matched with a known fact the goal is proven. If the goal is matched with the head of a rule the original goal is then replaced with the sub-goals which form the body of the rule. Prolog then attempts to prove, in the same way, each of these sub-goals in turn.

Although Prolog programs are thought of as declarative they can also have a procedural reading. Prolog is versatile: there is a style of Prolog programming which mimics the conventional procedural approach, but with less emphasis on the actual assignment statements.

Prolog as a logic programming language has a sound theoretical basis, being modelled on the first-order predicate calculus. This means that theoretically a Prolog program, which may be read as a set of axioms and theorems, contains the possibility of being proven consistent or inconsistent, regardless of the implementation. In this way Prolog can be thought of as an automated theorem prover.

Variable Types

A feature of Prolog is its use of type free variables which can represent widely different data structures. This allows any variable in a clause to represent a number, a string, a list of numbers, a list of strings, a list of strings and numbers etc. Prolog variables can even be used to represent clauses. This last use of variables leads to a technique called meta-programming, where the clauses in one part of a program can manipulate other clauses as data. This 'meta-level' ability is one reason why Prolog has been widely used in the realm of expert systems.

Data Structures

Another feature of Prolog is its ability to allow the dynamic assertion and retraction of rules and facts. This makes the Prolog rule and data base truly dynamic at run-time. This is essential for "learning systems" and other applications involving the introduction of new concepts, rules or facts at run-time.

Memory is dynamically allocated and deallocated at run-time. The deallocation is automatically done by a built-in garbage collector. The programmer does not have to be concerned with the implementation, maintenance and bookkeeping normally associated with the creation and destruction of complex dynamic data structures and is freed to concentrate on stating the logic of the problem.

How OOPS Can Benefit From Prolog

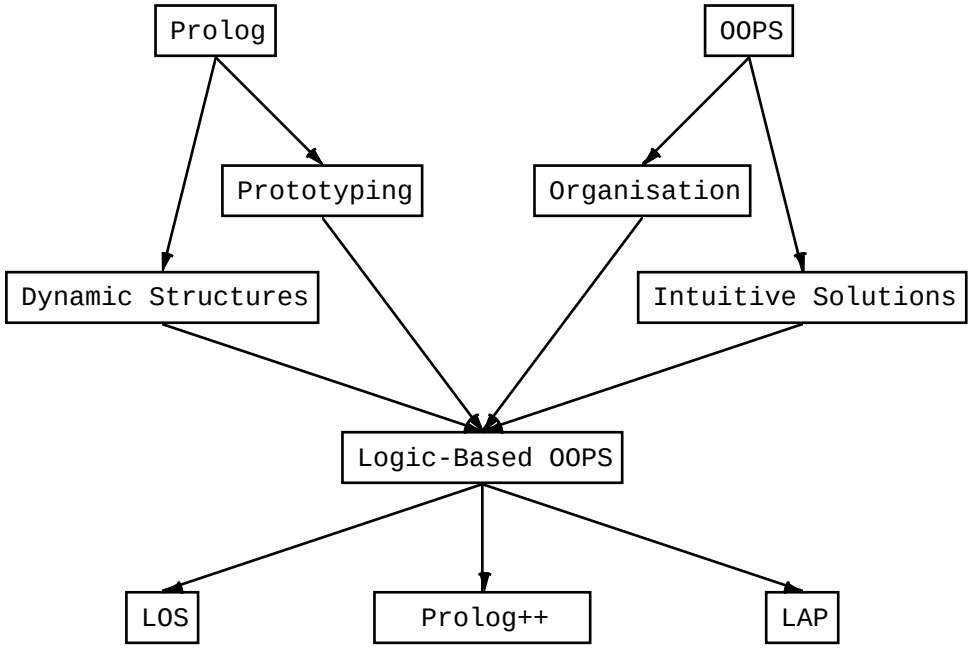
There are a number of advantages gained by using Prolog as a basis for an object-oriented language.

As discussed earlier in this chapter, Prolog is a type free language. This allows greater freedom for the programmer than a typed language, leading to an increase in both productivity and creativity.

All data structures in Prolog are dynamic. The creation and garbage collection of data structures is automatic. A Prolog-based OOPS will reflect this by allowing truly dynamic class hierarchies.

For example, consider a program which establishes animal taxonomies. The common features of animals may be abstracted into general classes and particular animals can then be classified according to those classes. In a Prolog-based OOPS not only can we dynamically add classes to a class hierarchy, we can dynamically augment classes that have already been established. This means that when a new characteristic is found for an existing animal, it can immediately be added to the class structure of that animal at run-time. This is *not* easily achieved in other systems.

The dovetailing of these two technologies leads to a very powerful and expressive system :-



How OOPS Can Enhance Prolog

The introduction of an OOPS methodology to Prolog can be beneficial in the following ways.

An OOPS provides a clear and intuitive structure for programs, in the form of class hierarchies. The one-to-one mapping between entities in the problem domain and classes in the program allows for a more easily understood program structure. In contrast, traditional module systems are a means of organising code for ease of development, and their structure does not necessarily correspond to the problem at hand.

Prolog imposes very few restrictions on the structure and organisation of a program. This is fine for small scale programs, but can lead to problems when constructing large applications. The onus is on the programmer to impose some discipline for organising and managing their code. The imposition of a class hierarchy leads to more efficient and easily manageable program structures.

An class has a well-defined interface which corresponds to its abstract or outline. Since this outline is a direct summary of an entity, knowing the entity allows the interface to be easily understood. This assists in the modularity of programs and the use of classes as "building blocks". Libraries of adaptable classes may then be created which can easily be integrated into larger applications.

Some Prolog-Based OOPS

The integration of Prolog and OOPS has attracted much interest within the academic community over the past few years, and many prototype systems have been developed. In addition, this basic research is now showing itself in the commercial marketplace. Two examples of the integration of Prolog and OOPS are presented here.

LOS

LOS (Logic and Instance System) has been developed at Imperial College, London by F.G. McCabe. It is a very expressive system, incorporating ideas from Prolog, OOPS and Lisp. It differs from Prolog++ in three major areas :-

- There are two, quite different inheritance mechanisms which can be used in LOS. The first is akin to the inheritance mechanism of Prolog++, and is called overriding inheritance. The second mechanism allows methods to be inherited in addition to any local statements. This means that a method is actually defined by the union of all statements in the class hierarchy for that method.

  For example, a class hierarchy for fault diagnosis will implicitly state that the possible faults are the union of all faults at all levels.

- An instance in LOS is not limited to a name alone, but may include parameters which are available to all the methods defined within the instance. These parameterized objects are referred to as class templates.

  For example, a train may have various attributes, such as speed and power rating, which are included as parameters of the *train* class template. A particular train, such as the *Flying_Scotsman*, will be declared as a sub-class of *train* with particular values for speed and power.

- There are no run-time objects in LOS, and no dynamic attribute values. To some extent, the parameters of class templates behave like attribute values.

LAP

LAP is a commercial product from Elsa software which integrates OOPS into Prolog by a suite of built-in routines. There is no additional programming language to learn, as LAP remains completely within the boundaries of Prolog.

The OOPS library of LAP is very extensive, with many routines for manipulating objects, methods and instances. Methods in LAP are divided into prefix statements, body statements and suffix statements, which is a very powerful feature for certain problem domains.

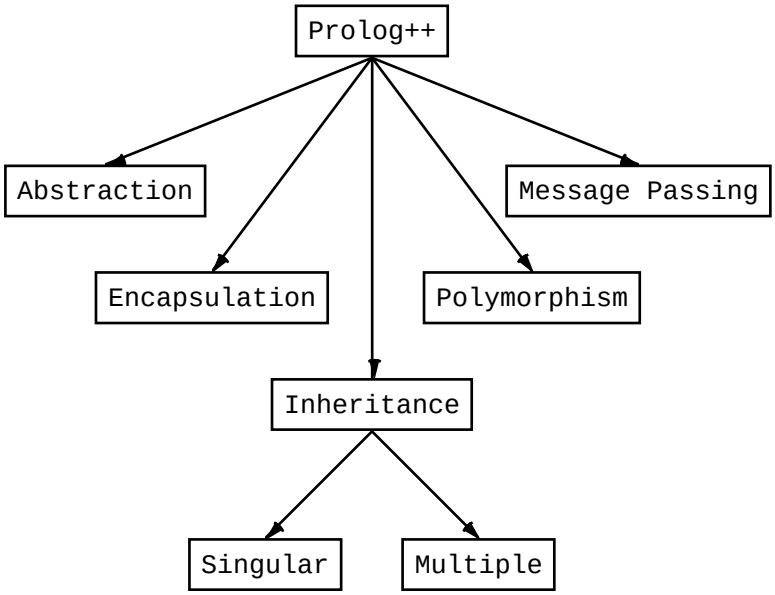
In addition to the basic library, Elsa have built an object-oriented environment for the development of LAP programs.

Chapter 3 - The Prolog++ Approach To OOPS

Here we discuss the way Prolog++ relates to the definition of OOPS found in chapter 1.

OOPS Features Of Prolog++

Prolog++ exhibits all the features of OOPS mentioned in the preceding chapters. They are :-



Class Hierarchies And Inheritance

Class hierarchies are implemented by denoting within an class who its parents are. The inheritance of methods and attributes is governed by this relationship. Prolog++ supports multiple inheritance by multiple declarations of parenthood.

Polymorphism And Message Broadcasting

Prolog++ supports the use of different methods in different classes which can handle the same message. This follows the common view of polymorphism.

Prolog++ provides the ability to broadcast messages in the following ways :-

- A single message to a group of instances.
- A group of messages to a single instance.
- A group of messages to a group of instances.

Encapsulation And Abstraction

Class definitions in Prolog++ are delimited by key words for their beginning and ending. All the methods and attributes within these two statements are encapsulated.

The name of the class and the names of any public methods and attributes form the abstract of that class.

Data-Driven Programming

The paradigm of data-driven programming is expressed in Prolog++ by the use of specific methods. Within Prolog++ there are four kinds of events which may cause a handler to be invoked. These are :-

- create a new class instance
- delete an existing class instance
- assign a new value to a class/instance attribute
- raise a program exception

The Inheritance Mechanism Of Prolog++

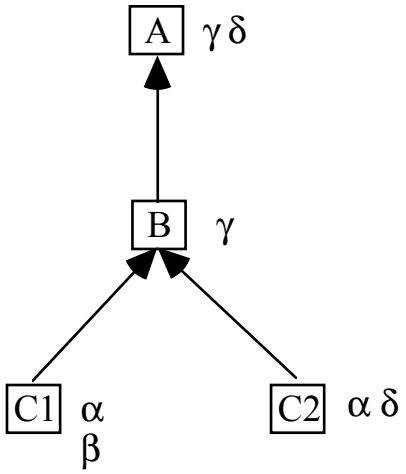
Whenever a message is sent to an instance (or indeed a class itself) the Prolog++ system determines a single location which has the capability of handling that message. Often the location will be the target class itself. Once the location has been determined the system commits itself to sending the message there and nowhere else. The method of determining locations which can actually handle messages is called the inheritance mechanism.

The inheritance mechanism of Prolog++ is based upon the same searching principles as the underlying Prolog system. This search is a branch-first, left-to-right traversal of the class hierarchy starting at the instance's class to which the message was originally sent.

To illustrate the searching involved in the inheritance mechanism we shall consider two example hierarchies. The first is a strict hierarchy representing singular inheritance, and the second is a lattice representing multiple inheritance.

Singular Inheritance

In the following diagram the methods α, β, γ and δ are defined in various objects A, B, C1 and C2.

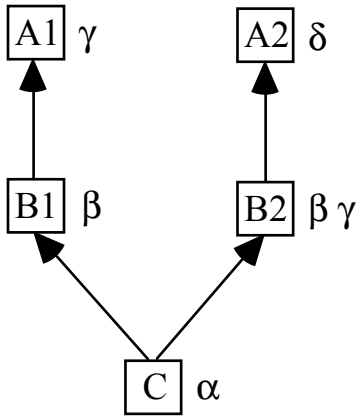


The following table shows where messages will actually be handled. For example, select message γ from the left-hand column and class C1 from the top-most row. Their intersection in the table, namely B, is the class where the message γ will actually be handled. An empty cell in the table indicates that the message cannot be handled when sent to instances of that class.

	A	**B**	**C1**	**C2**
α			C1	C2
β			C1	
γ	A	B	B	B
δ	A	A	A	C2

Multiple Inheritance

In the following diagram the methods α, β, γ and δ are defined in various objects A1, A2, B1, B2 and C. The important links are from B1 and B2 down to C. Class B1 is the first parent of C and class B2 is the second parent of C. This ordering is crucial for determining inheritance.



The following table shows where messages sent will actually be handled. For example, select message γ from the left-hand column and class C from the top-most row. Their intersection in the table, namely A1, is the class where the message γ will actually be handled. An empty cell in the table indicates that the message cannot be handled when sent to any instances of that class.

	A1	**A2**	**B1**	**B2**	**C**
α					C
β			B1	B2	B1
γ	A1		A1	B2	A1
δ		A2		A2	A2

The Binding Process

Prolog++ binds messages to handlers either at compile-time (early) or at run-time (late), depending upon whether or not the message can be handled locally by the class in which it textually occurs.

To illustrate the difference between early and late bindings consider the following definitions for classes *alpha* and *beta*.

```
class alpha .

      print_early :-
            print_header,
            print_contents,
            print_footer.

      print_late :-
            print_header,
            self <- print_contents,
            print_footer.

      print_contents :-
            …

end alpha.
class beta .

      inherit alpha .

      print_contents :-
            …
```

```
end beta
```

Early Binding

Early binding automatically occurs for local messages whenever there is a corresponding local handler. An early binding is forged at compile-time which greatly improves the efficiency of the resulting code by avoiding the search for a handler.

For example, the message *print_contents* in the definition of *print_early* in class *alpha* will be bound to the corresponding method also found in class *alpha*.

Late Binding

Late bindings between a message and a class capable of handling it are forged at run-time whenever no local link can be found and for all messages which are explicitly sent to an object. It takes the form of a search up through the is-a hierarchy of classes starting at the class of the message receiver.

For example, the messages *print_header* and *print_footer* in the definitions of *print_early* and *print_late* in class *alpha* will be bound at run-time according to the context in which they are sent. That is, the binding is determined by the object which originally received the *print_early* or *print_late* message.

Forced Late Binding

If you wish to avoid an early binding (and thus fool the compiler) send the message explicitly to the *self* variable. When this is stated in the code a late binding will always be performed at run-time even if there is a local handler for that message.

For example, the definitions of *print_early* and *print_late* in class *alpha* only differ in the way they send the message *print_contents*. The method *print_late* forces the message to be bound at run-time by explicitly sending it to the *self* variable.

The message *print_early* for an instance of *beta* will use the definition of *print_contents* found in *alpha*, whereas the message *print_late* for a similar instance will use the definition of *print_contents* found in *beta* itself.

If the definition of *print_contents* were removed from class *alpha* then the two calls would be equivalent since the compiler would not be capable of an early binding!

How Prolog++ Benefits From Prolog

Prolog++ inherits from Prolog the ability to use dynamic data structures; this extends to include classes and instances.

The variables used within Prolog++, being type-free, can be used to represent widely different data structures.

In addition to these, Prolog++ may be fully integrated with standard Prolog programs. Each Prolog++ class has access to all Prolog predicates, whether system-defined or user-defined.

This allows Prolog++ to inherit all the capabilities of a fully developed Prolog environment.

How Prolog++ Enhances Prolog

Programs may be developed using Prolog++ which augment standard Prolog in the following ways.

The ability to define a class hierarchy allows the programmer to partition Prolog programs into dynamic modules which closely resemble the structure of the problem domain.

Each class within Prolog++ also has a clearly defined interface formed by the names of its methods. The ability to define classes whose interface corresponds to their function is also of use when trying to model the problem domain.

The ability to have a close resemblance between a class, its interface and the problem domain enables the interfaces to the classes to be easily understood. This coupled with the encapsulation of the classes into packages of code and data allows Prolog++ programs to be re-used like building blocks.

Data-Driven programming is supported by allowing daemons to be attached to crucial events. The mechanism for trapping events is incorporated in the Prolog++ engine itself and does not need to be explicitly handled by the user.

Prolog++ And Smalltalk

This section compares Prolog++ with Smalltalk.

General

Smalltalk is an object-oriented procedural language.

Smalltalk is generally acknowledged as the first major implementation of the OOPS paradigm.

Smalltalk was derived at Xerox's Palo Alto Research Centre, by a process of evolution from Simula-67, a language designed for the implementation of simulations.

Environment

Smalltalk is usually used in a graphical environment tailored specifically for Smalltalk development. This environment usually includes - mouse handling, windows, menus, editors, browsers etc.

Memory management is automatically provided by the system in the form of garbage collection of objects.

Prolog++ provides all of this through the underlying Prolog system.

Classes

A class defines the behaviour of similar objects by specifying their structure and functionality; the data they represent and the methods available for responding to received messages. The class concept in Smalltalk is distinguished from an object in general by virtue of the class definition messages by which a class is defined. These messages are sent to the classes' superclass with its specification as arguments.

Also classes are global structures in that any class can be referenced by name within any instance. In practice this means that objects of any class can be created anywhere.

As classes in Smalltalk are objects, the metaclass concept is used to describe the general class behaviour. Thus every class belongs to its own unique class called a metaclass which determines the messages to which the class can respond. The general behaviour of metaclasses is described in the class MetaClass. All instances of MetaClass (i.e. metaclasses) have the superclass Class. The methods for the class definition messages are defined in Class.

This is called classtrophobia!!!!

Methods and Messages

A message in Smalltalk is simply the identifier for an associated method defined by the class of the instance that recieves the message. A message, in a similar fashion to a function, always returns an instance of some sort. In Smalltalk messages may take arguments.

Messages may be cascaded.

A cascaded message is a series of messages sent to one receiver instance.

Prolog++ can also cascade messages, it can also broadcast messages to several instances.

There are several types of message:

Smalltalk distinguishes three types of message; unary, binary, keyword (n-ary). These are distinguished by arity, precedence and syntax.

Prolog++ does not distinguish between messages according to their arity (unary, binary, ternary etc.). A message may contain any number of arguments.

Objects

In Smalltalk everything is an instance.

An object's internal variables are termed instance variables; these will also contain objects. For example an instance of the class "complex numbers" may have the following values for its instance variables:

real = 1.0

imaginary = 0.4.

The instance variables are defined in the object's class definition.

Prolog++ also has the concept of instance variables. They are called attributes.

Inheritance

Smalltalk does not implement multiple inheritance - everything inherits from the root instance, 'Object'. This form of inheritance is sometimes called 'tree inheritance' for obvious reasons.

Prolog++ supports multiple inheritance, which is a superset of tree inheritance.

Encapsulation

An instance of a class is encapsulated by the data-abstraction for that class. All instance variables are private to that instance and can only be accessed via the class protocol.

In Prolog++ an instance of a class is similarly encapsulated by the data-abstraction for that class.

Polymorphism

Messages and methods are genuinely polymorphic, i.e. the same message for different objects may have different methods depending on the instance.

Data Abstraction

Data abstraction is organised via the class hierarchy.

Prolog++ And C++

The version of C++ referred to here is 2.0 from AT&T.

General

C++ was developed, and is still being developed, by Bjarne Stroustrup at AT&T Bell laboratories. It has become a major systems language as an enhancement of C.

C++ is a more traditional batch compiled language, and to that extent it is a lot less flexible than either Smalltalk or Prolog++.

The C++ language is, like Prolog++, an object-oriented extension of an already existing language. C++ takes the C struct keyword, which is used for user defined data types, and extends and enhances it to provide the basic object-oriented mechanisms. Struct is retained, but an ability to associate functionality with the data type is added. A further more important construct, namely the class, is also provided. The struct is a special case of a class with no protected or private sections.

By this means C++ provides the C language with the various bits and pieces which are generally acknowledged to be object-oriented.

C++ is a strongly typed language. This has important implications for implementation and prototyping techniques. Prolog++ is not strongly typed: this aids the speed of development and prototyping.

C++ is a looser (compared to Smalltalk) object-oriented system in that the programmer is not compelled to use class methodology: She may still write in normal (ANSI) C. Prolog++, via Prolog, also provides this escape from being straight-jacketed into the OOPS paradigm. This is a distinct advantage over Smalltalk as some problems do not naturally fit within the OOPS paradigm.

A system devised in C++ tends to be more rigid than Prolog++ or Smalltalk.

C++ has a major problem in the area of exception handling. It is not defined at all except insofar as inheriting C's primitive approach. These problems come to light most vividly when using dynamic objects and heap manipulation.

Environment

Various commercial products are available for the development of C++ programs.

Memory managment is left to the user.

Classes

The class in C++ is the main vehicle of data-abstraction. It encapsulates and hides the underlying data structure. C++ has within the class construct the notions of private, public and protected, by which it controls the access to data and methods. Private data and methods are only available to member functions defined within the scope of the class. Protected data and methods are available only to the class and its derived classes via inheritance. Public data and methods are available to all. The default is for data and methods to be private. The private access can be overridden by the friend keyword, which can name functions as friends to the class.

Methods and Messages

A message in C++ is the name of a method passed to an instance. A message corresponds to a function defined in the class template of that instance. These are called member functions in C++. Generally definitions within a class are termed class members.

Member functions of a class with the same name as the class (say for class foo - foo()) are termed constructor functions. Member functions with the same name as the class preceded by a tilde (~foo()) are termed destructor functions. Constructors generally correspond to class methods in Smalltalk. While destructors simply destroy objects when they go out of scope.

In C++ there are two types of method, normal functions and in-line functions. In-line functions are intended to replace parameterized macros, which because of their inherent lack of argument typing are considered 'unsafe'. C++ also has two forms of method determination. By default, methods to answer messages are determined by the compiler and hard-wired into the instance code. A virtual function, however, is determined at run-time The run-time system decides which function to field dependent upon the type of the receiving instance. This last is usually called late-binding. Late-binding is fundamental to all object-oriented systems. Both function and in-line methods let the user create more efficient code.

Prolog++ has an advantage in this respect in that the underlying system has late-binding at its core in the form of unification.

Each message evokes one function call only. Cascading must be done manually by the programmer.

Objects

Each instance of a given class has its own copy of all the data defined by the class.

The static declarator overrides this inheritance and provides a means for class variables, as in Smalltalk (i.e. a single variable whose value is inherited by all objects of that class).

Objects may be static - created at compile time; or dynamic - created at runtime. The C++ language extends the C language by providing the functions, new and delete, which allocate and free heap space for the ease of use of dynamic objects (as opposed to malloc etc.).

Inheritance

As of version 2.0 C++ supports multiple inheritance.

Encapsulation

The C++ language provides for encapsulation via the data definition constructs: struct and class. In C++, for efficiencies sake,.this mechanism may be overridden in various ways

Polymorphism

Member functions of a given class can be overloaded. The overloaded member functions are distinguished by the types of their arguments.

Operator overloading

Nearly all the operators in C++ may be overloaded. What this means is that any operator, such as '+' say, can be given a new functionality within a given class. The precedence and associativity is, however, immutable. Prolog++ also provides this convenience via the underlying Prolog system. In Prolog an operator is purely syntactic, being the functor of a relation.

Data Abstraction

Data abstraction in C++ is organised by the class hierarchy.

C++ offers a variety of data protection: public, private or protected. By default , data is private to a class definition. This data protection may be overridden by declaring friend functions.

Part 2 - Case Studies In Prolog++

Chapter 4 - Introduction To The Case Studies

These case studies are intended to introduce both the language of Prolog++ and some of its salient features. The reader may either accept the solutions given or refer to the following technical part for clarification of particular points.

As a brief introduction to some of the notation used within Prolog++ the following table is given. This is not intended to be an exhaustive set of symbols found in this chapter but rather an initial set to bootstrap the reader into the examples.

Method **/** Arity	Methods are referred to by the / symbol, with the name of the method on the left and its arity (number of arguments) on the right.
Function **//** Arity	Functions are referred to by the // symbol, with the name of the function on the left and its arity (number of arguments) on the right.
Attribute	Attributes are referred to by their name alone.
Instance **<-** Message	<- is the principal symbol of Prolog++, and indicates that the message on the right is to be sent to the instance on the left.
Instance **@** Attribute	@ is the symbol used to indicate the value of an instance's attribute.

The case studies cover a range of problems, each of which illustrates different aspects of the language. They are as follows :-

Simulation	A simulation of the activities of a bank which investigates different queueing models.
Resource Scheduling	Constructing a school timetable with respect to both physical and desired constraints.
Database	A stock control system including a report generator for periodic trading results.
Fault Diagnosis	Diagnosing faults in an automobile according to exhibited symptoms and counter symptoms.

Chapter 5 - A Case Study In Simulation

This case study will investigate the application of Prolog++ to event-driven simulation, specifically within a banking environment. It will illustrate various aspects of Prolog++, including polymorphism, message broadcasting, the use of abstract data types and the structuring of a class hierarchy, albeit a small one. More specifically, it will call upon the use of class variables within the definitions of attributes, functions and methods.

A very important characteristic of the overall object-oriented approach is the way in which code is re-utilised for seemingly different situations. Exactly the same routines will be employed for simulating single-queue and multiple-queue scenarios.

The Problem

A bank needs information to rationalise its staffing requirements according to its expected business. An optimum solution, based upon simulating the bank's processes, will minimize the number of bank tellers required together with their idle time, whilst simultaneously reducing how long customers have to wait before being served. Obviously, this is an impossible task as anybody who has tried to use a bank during lunch hours will testify.

Within the banking environment certain events can be recognised; the arrival of customers, their joining of a queue, being served by a teller, and finally leaving the bank. Arrivals and service time are of random durations, and will be modelled accordingly.

The biggest factor which affects customer turnaround time is the number of queues, ranging from a single communal queue for all of the bank tellers to individual queues for each. Hopefully, an object-oriented approach will be able to simulate both with very little specialisation of the code. From the viewpoint of a bank teller, s/he does not care whether the supply queue is private or shared with others. From the viewpoint of a customer, s/he will always choose the smallest queue, however many there are.

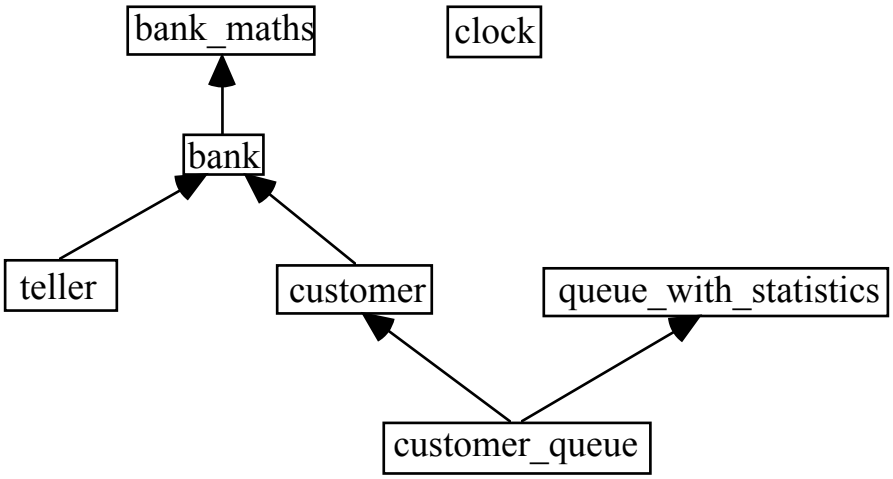
In order to make an informed judgement, various statistics will need to be collected. These will include such values as the total number of customers served, the average size of queues, the average time waiting in each queue, and many more.

The input parameters for each simulation will include such values as how long the bank is open for, how many tellers there are, the number of queues, and others.

The Classes

From the discussion above, various entities in the problem domain can be identified. These are bank, teller, customer and customer queue. Of course, since it is a simulation problem it will also involve the entity time.

In addition, a customer queue can be thought of as a general queue, showing the same behaviour as queues but also incorporating some of the special features of customers.



The following sections define the various classes in the bank simulation, except *queue_with_statistics* which can be found in the appendix on library classes.

The "clock" Class

The *clock* class represents the simulation time, and is defined by the following attributes and methods.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% CLASS  : clock
% COMMENT: The clock used in the simulation of the bank
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

class clock .


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                              DECLARATIONS                                    %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

category
  user ,
  bank .                    % A class in the bank simulation

public class attributes
  time_now     is 0  ,      % The clock starts at time 0
  time_to_stop is 50 .      % The clock ends at time 50

public methods
  stop       /  1 ,   % Reset the stop time
  start      /  0 ,   % Restart the clock
  tick       /  0 ,   % A tick of the clock
  still_open /  0 ,   % The clock has not yet reached the stop time
  closed     /  0 ,   % The clock has passed the stop time
  print      /  0 .   % Print details about the clock
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                              DEFINITIONS                                    %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% METHOD : stop/1
% COMMENT: Reset the stop time
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

stop( Time ) :-
  time_to_stop := Time.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% METHOD : start/0
% COMMENT: Restart the clock
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

start :-
  time_now := 0.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% METHOD : tick/0
% COMMENT: A tick of the clock
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

tick :-
  time_now += 1,
  nl,
  write( 'Time: ' ),
  write( @time_now ),
  nl.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% METHOD : still_open/0
% COMMENT: Succeeds when clock has not yet reached the stop time
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

still_open :-
  time_now < time_to_stop.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% METHOD : closed/0
% COMMENT: Succeeds when the clock is at or past the stop time
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

closed :-
  time_now >= time_to_stop.
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% METHOD : print/0
% COMMENT: Print details about the clock
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

print :-
  write( 'Closing time                              : ' ),
  write( @time_to_stop ),
  nl,
  write( 'Last customer served at                   : ' ),
  write( @time_now ),
  nl,
  nl.

end clock.
```

The "bank" Class

The declarations include a statement of its only parent class (*bank_maths*), various categories to which the *bank* class belongs and, finally, the methods which are publicly available to all other classes (*setup/1/4*, *simulation/0* and *print/0*).

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% The main class in the bank simulation
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

class bank .

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                              DECLARATIONS                                    %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

inherits
  bank_maths .           % Inherit some mathematical methods

category
  user ,
  bank .                 % A class in the bank simulation

public methods
  setup       /  1 ,     % Setup with personal or communal queues
  setup       /  4 ,     % Setup the bank configuration
  simulation  /  0 ,     % Start the simulation process
  print       /  0 .     % Print the statistics from the simulation
```

The *setup* methods handle various ways of setting up the bank in preparation for the simulation. The mode and service times control the behaviour of the bank tellers, the arrival distribution determines the frequency with which customers arrive at the bank and the stop time determines when the bank closes.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% METHOD : setup/4
% COMMENT: Setup the bank configuration
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

setup( Mode ) :-
  setup(
         Mode,
         [(4,1),(3,2),(3,1),(2,2),(2,1)],
         [0,0,0,1,1,1,1,1,2,2,3],
         20
        ).

setup( Mode, ServiceTimes, ArrivalDistribution, Stop ) :-
  teller   <- setup( Mode, ServiceTimes ),
  customer <- setup( ArrivalDistribution ),
  clock    <- stop( Stop ).
```

The *simulation* method is used to simulate the bank's operations according to how it was set up. In the first phase customers arrive and are served at periodic intervals while the bank is still open for business, whereas in the second phase only the remaining customers are served.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% METHOD : simulation/0
% COMMENT: The simulation consists of two phases
%          1) A period when the bank is open
%          2) The bank is closed, but customers are still being served
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

simulation :-
  (
  clock,
  customer,
  all instance customer_queue,
  all instance teller
  ) <- start,
  while clock <- still_open
  do    (
        clock <- tick,
         (
         customer,                    % Customers arrive when open
         all instance customer_queue, % Update queue statistics
         all instance teller          % Customers served when open
         ) <- after_tick
        ),
  while still_busy
  do    (
        clock <- tick,
```

```
         (
         all instance customer_queue, % Update queue statistics
         all instance teller          % Customers served when busy
         ) <- after_tick
        ).

still_busy :-
  instance teller <- busy.

still_busy :-
  instance customer_queue <- non_empty.
```

Once the simulation has been performed the *print* method is used to report all of the statistics which have been gathered. This is primarily done by broadcasting the *print* message to all parts of the simulation model.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% METHOD : print/0
% COMMENT: Print the statistics resulting from the simulation
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

print :-
  nl,
  write( 'Bank Simulation Statistics' ),
  nl,
  write( '==========================' ),
  nl,
  nl,
  ( clock, customer, teller, customer_queue ) <- print.

end bank.
```

The "bank_maths" Class

The methods of this class are available to all the other classes of the bank simulation, except the *clock* class, through the inheritance mechanism of Prolog++.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Some of the mathematics involved
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

class bank_maths .

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                              DECLARATIONS                                   %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

categories
  user ,
  bank .                        % A class in the bank simulation
```

```
public methods
  nearest_number // 1 ,          % Nearest whole number
  variation      // 2 ,          % Mean variation
  random_member  // 1 .          % Random member of a list

private methods
  variation_sequence // 1 ,      % Auxilliary function
  member             // 2 .      % Auxilliary function

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% METHOD : nearest_number//1
% COMMENT: Nearest whole number
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

nearest_number( Float ) is
  int( ( Float + 0.5 ) * 10 ) // 10.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% METHOD : variation//2
% COMMENT: Normal variation from the mean
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

variation( Mean, Variation ) is
  Mean + random_member( variation_sequence(Variation) ).

variation_sequence( 0 ) = [0].
variation_sequence( 1 ) = [-1,0,0,0,1].
variation_sequence( 2 ) = [-2,-1,-1,-1,0,0,0,0,0,1,1,1,2].
variation_sequence( 3 ) = [-3,-2,-2,-1,-1,-1,-1,
                           0,0,0,0,0,0,0,0,1,1,1,1,2,2,3].

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% METHOD : random_member//1
% COMMENT: Random member of a sequence
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

random_member( Sequence ) is
  member( int(irand(length(Sequence))), Sequence ).

member( 0, [Item|_] ) = Item :-
  !.

member( Jth, [_|Sequence] ) is member( Jth-1, Sequence ).

end bank_maths.
```

The "teller" Class

This class simulates the behaviour of bank tellers. In particular, it introduces the concept of a class attribute, *queueing*, which governs the queueing model used when creating individual tellers.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% A simulation of the operations carried out by a bank teller
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

class teller .

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                              DECLARATIONS                                    %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

categories
  user ,
  bank .                      % A class in the bank simulation

inherit
  bank .                      % Inherit the bank methods

public class attributes
  queueing = personal .       % Which queueing model to use?

public instance attributes
  teller_number .             % Unique teller #

private instance attributes
  mean_service_time         , % A measure of a teller's efficiency
  service_time_variation    , % Variation in efficiency
  served_by_queue           , % Which queue is the teller served by
  serving_customer          , % Which customer they are serving
  aggregate_when_busy  is 0 , % Aggregate time when teller is busy
  aggregate_customers  is 0 , % Aggregate total of customers served
  time_until_next_free is 0 . % Time until teller next becomes free

public methods
  start           /  0 ,      % Reset dynamics
  setup           /  2 ,      % Setup teller characteristics
  when_created    /  0 ,      % React to creation of a new teller
  after_tick      /  0 ,      % What happens after a tick
  busy            /  0 ,      % Is the teller busy ?
  idle            /  0 ,      % Is the teller idle ?
  about_to_finish /  0 ,      % Is the teller about to finish ?
  print           /  0 .      % Print statistics about the tellers
```

```
private methods
  start_serving_customer / 0, % Start serving the next customer
  finish_serving_customer/ 0, % Finish serving a customer
  print_me               / 0, % Print statistics about a teller
  percentage_when_busy   //0. % Percentage time when teller is busy
```

The *start* method re-initialises all of the statistics which are gathered about tellers.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% METHOD : start/0
% COMMENT: Reset dynamic attributes
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

start :-
  reset( serving_customer     ),
  reset( aggregate_when_busy  ),
  reset( aggregate_customers  ),
  reset( time_until_next_free ).
```

The *setup* method creates a number of tellers, each with their own average service time. Each teller is either linked to a communal queue or to their own personal queue.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% METHOD : setup/2
% COMMENT: Setup characteristics of the tellers & then create them
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

setup( Mode, ServiceTimes ) :-
  customer_queue <- delete_all,
  teller         <- delete_all,
  queueing := Mode,
  ( Mode = communal -> create_communal( ServiceTimes )
  ; Mode = personal -> create_personal( ServiceTimes ) ).

create_communal( ServiceTimes ) :-
  customer_queue <- create( Queue, [queue_number = 1] ),
  create_communal( ServiceTimes, 1, Queue ).

create_communal( [], _, _ ).

create_communal(
                 [(MeanServiceTime,ServiceVariation)|ServiceTimes],
                 Number,
                 Queue
               ) :-
  super <- create(
                   _,
                   [
                   served_by_queue        = Queue,
```

```
                  mean_service_time      = MeanServiceTime,
                  service_time_variation = ServiceVariation,
                  teller_number          = Number
                  ]
                ),
  create_communal( ServiceTimes, +Number+1, Queue ).

create_personal( ServiceTimes ) :-
  create_personal( ServiceTimes, 1 ).

create_personal( [], _ ).

create_personal(
                [(MeanServiceTime,ServiceVariation)|ServiceTimes],
                Number
              ) :-
  customer_queue <- create( Queue, [queue_number = Number] ),
  super <- create(
                  _,
                  [
                  served_by_queue        = Queue,
                  mean_service_time      = MeanServiceTime,
                  service_time_variation = ServiceVariation,
                  teller_number          = Number
                  ]
                ),
  create_personal( ServiceTimes, +Number+1 ).
```

The following is an example of data-driven programming. The *when_created* method is not actually called by the program itself but reacts whenever a new teller is created.

```
when_created :-
  write( 'Teller #' ),
  write( @teller_number ),
  write( ' draws customers from queue #' ),
  write( @served_by_queue@queue_number ),
  nl.
```

At each tick of the clock the tellers need to assess what they are doing, whether or not they have finished serving their current customer and whether they can start serving the next customer.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% METHOD : after_tick/0
% COMMENT: What do tellers do after each tick of the clock
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Start serving the next customer in the teller's queue

after_tick :-
  idle,
  start_serving_customer,
  !,
  time_until_next_free := @serving_customer@work_load
                        + variation(
                                     mean_service_time,
                                     service_time_variation
                                   ),
  aggregate_when_busy  += 1,
  aggregate_customers  += 1.


% Finish serving a customer at this tick of the clock

after_tick :-
  about_to_finish,
  !,
  aggregate_when_busy  += 1,
  time_until_next_free := 0,
  finish_serving_customer.


% Still serving a customer

after_tick :-
  busy,
  !,
  aggregate_when_busy  += 1,
  time_until_next_free -= 1.


% Idle time when there are no customers in the teller's queue

after_tick.
```

The notion of a busy teller revolves around the time when s/he next becomes free, which is a function of both their own efficiency and the work-load of the customer.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% METHOD : busy/0
% METHOD : idle/0
% METHOD : about_to_finish/0
% COMMENT: Is the teller busy ?
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

busy :-
  @time_until_next_free > 0.

idle :-
  @time_until_next_free = 0.

about_to_finish :-
  @time_until_next_free = 1.
```

These methods handle the serving of a customer.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% METHOD : start_serving_customer/0
% METHOD : finish_serving_customer/0
% COMMENT: Start and finish serving a customer
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

start_serving_customer :-
  serving_customer := @served_by_queue@next_customer,
  write( 'Teller #' ),
  write( @teller_number ),
  write( ' starts serving customer #' ),
  write( @serving_customer@customer_number ),
  nl.

finish_serving_customer :-
  write( 'Teller #' ),
  write( @teller_number ),
  write( ' finish serving customer #' ),
  write( @serving_customer@customer_number ),
  nl,
  @served_by_queue <- exit_customer,
  @serving_customer <- delete,
  reset( serving_customer ).
```

At the end of the simulation the *print* message is broadcast to all parts of the model. Here the statistics of each teller are printed.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% METHOD : print/0
% COMMENT: Print statistics about the tellers
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

print :-
  write( 'Number of bank tellers                    : ' ),
  write( @length( all instance teller ) ),
  nl,
  write( 'Mode of operation                         : ' ),
  write( @queueing ),
  write( ' queue(s)' ),
  nl,
  nl,
  all instance teller <- print_me.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% METHOD : print_me/0
% COMMENT: Print statistics about a particular teller
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

print_me :-
  write( 'Teller # ' ),
  write( @teller_number ),
  nl,
  write( 'Total number of customers served          : ' ),
  write( @aggregate_customers ),
  nl,
  write( 'Percentage time spent serving customers   : ' ),
  write( @percentage_when_busy ), write( '%' ),
  nl,
  write( 'Mean time to serve a customer             : ' ),
  write( @mean_service_time ),
  nl,
  write( 'Variation in service time                 : ' ),
  write( @service_time_variation ),
  nl,
  nl.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% METHOD : percentage_when_busy//0
% COMMENT: Percentage time when teller is busy serving customers
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

percentage_when_busy is nearest_number(
                                        (
                                           aggregate_when_busy /
                                           clock@time_now
```

```
                                        ) * 100
                                      ).

end teller.
```

The "customer" Class

The primary purpose of this class is to model the arrival distribution of the bank's customers.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Typical customers arriving at the bank have
% different service requirements and so different
% work-loads
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

class customer .

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                     DECLARATIONS                                %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

category
  user ,
  bank .                    % A class in the bank simulation

inherit
  bank .                    % Inherit the bank methods

public class attributes
  arrival_distribution     , % Distribution of customer arrivals
  mean_work_load       is 2, % Mean work-load of each customer
  work_load_variation  is 1, % Variation in the work-loads
  aggregate_customers  is 0, % The number of customers so far
  aggregate_work_load  is 0. % Aggregate work-load of all customers

public instance attributes
  customer_number ,         % Unique customer #
  work_load       .         % The work-load of a specific customer

public methods
  setup      /  1 ,         % Setup characteristics of the customers
  start      /  0 ,         % Delete any old customers
  after_tick /  0 ,         % What happens after a tick of the clock
  print      /  0 .         % Print statistics about the customers

private methods
  create       /  1 ,       % Create N new customers
  when_created /  0 .       % Calculate work-load for next customer
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% METHOD : setup/1
% COMMENT: Setup characteristics of the customers
%          arriving
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

setup( ArrivalDistribution ) :-
  arrival_distribution := ArrivalDistribution.
```

The *start* method re-initialises all of the statistics which are gathered about customers.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% METHOD : start/0
% COMMENT: Delete any old customers !!!
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

start :-
  delete_all,
  aggregate_customers := 0,
  aggregate_work_load := 0.
```

At each tick of the clock a number of customers will arrive. This is determined by the arrival distribution when the simulation is set up.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% METHOD : after_tick/0
% COMMENT: After each tick calculate how many
%          customers will arrive and send them to
%          the shortest queues (at the time of
%          their creation)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

after_tick :-
  create( @random_member( @arrival_distribution ) ).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% METHOD : create/1
% COMMENT: Create N customers with his/her work load
%          & increment counter
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

create( 0 ) :-
  !.

create( N ) :-
  super <- create( Customer ),
  customer_queue@smallest <- new_customer( Customer ),
  create( +N-1 ).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% METHOD : when_created/0
% COMMENT: Calculate the work-load of a customer who
%          has just arrived, and set his/her customer #
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

when_created :-
  aggregate_customers += 1,
  customer_number     := aggregate_customers,
  work_load           := variation(
                                   mean_work_load,
                                   work_load_variation
                                  ),
  aggregate_work_load += work_load,
  write( 'Next customer #' ),
  write( @customer_number ),
  write( ' with work-load ' ),
  write( @work_load ),
  write( ' joins queue #' ),
  write( (customer_queue@smallest)@queue_number ),
  write( ' (length = ' ),
  write( (customer_queue@smallest)@real_size ),
  write( ')' ),
  nl.
```

At the end of the simulation the *print* message is broadcast to all parts of the model. Here the statistics of each customer are printed.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% METHOD : print/0
% COMMENT: Print statistics about the customers
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

print :-
  write( 'Distribution of customer arrivals        : ' ),
  write( @arrival_distribution ), nl,
  write( 'Mean work-load for customers arriving    : ' ),
  write( @mean_work_load ), nl,
  write( 'Variation in the work-load of customers  : ' ),
  write( @work_load_variation ), nl,
  write( 'Total number of customers which arrived  : ' ),
  write( @aggregate_customers ), nl,
  write( 'Total work-load of all customers         : ' ),
  write( @aggregate_work_load ),  nl,
  nl.
```

```
end customer.
```

The "customer_queue" Class

The *customer_queue* class is a specialisation of a general library class for queues which has been adapted for bank customers. A special attribute of this class identifies which is the smallest queue.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% A queue of customers with statistics
% about average size and wait time
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

class customer_queue .

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                      DECLARATIONS                                     %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

category
  bank ,                    % A class in the bank simulation
  user .

inherits
  customer ,                % Inherit the customer & bank methods
  queue_with_statistics .   % Also inherit from statistical queues

public class attributes
  smallest = @smallest_now . % The smallest queue

public instance attributes
  queue_number   ,          % Number of the queue
  real_size is 0 .          % Real queue size includes busy tellers

private instance attributes
  cumulative_size is 0 .    % The cumulative size over time

public methods
  start         /  0 , % Reset dynamic attributes
  when_created  /  0 , % Definitely changes the smallest queue
  new_customer  /  1 , % May change the smallest queue
  next_customer /  1 , % Pop customer before starting service
  exit_customer /  0 , % Reduce size at end of service
  after_tick    /  0 , % What to do after a tick
  smallest_now  // 0 , % Re-compute the smallest queue now
  print         /  0 . % Print statistics about the customer queues

private methods
  smallest_now  // 1 , % Auxilliary function
```

```
  smallest_now  // 2 , % Auxilliary function
  smaller       // 2 , % Auxilliary function
  print_me      /  0 , % Print statistics about a particular queue
  average_size  // 0 , % Average size over time
  average_wait  // 0 . % Average waiting time in the queue
```

The *start* method re-initialises all of the statistics which are gathered about customer queues.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% METHOD : start/0
% COMMENT: Reset dynamic attributes
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

start :-
  Number is queue_number,
  reset,
  queue_number := Number.
```

Whenever a new queue is created it automatically becomes the smallest queue. Whenever a customer joins the smallest queue then it may no longer be the smallest. The smallest is calculated by comparing the "real" sizes of each of the queues.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% METHOD : when_created/0
% COMMENT: Self becomes the smallest queue
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

when_created :-
  smallest := self.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% METHOD : new_customer/1
% COMMENT: Push a customer onto the back of this
%          queue.
%          Increment the "real size" of the queue
%          Be careful if it was the smallest queue!
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

new_customer( Item ) :-
  super <- push( Item ),
  real_size += 1,
  ( smallest = self -> reset( smallest ) ; true ).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% METHOD : smallest_now//0
% METHOD : smallest_now//1
% METHOD : smallest_now//2
% COMMENT: Compute and return the smallest customer
%          queue
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

smallest_now = @smallest_now( all instance class self ).

smallest_now( [Queue|Queues] ) = @smallest_now( Queues, Queue ).

smallest_now( [], Smallest ) = Smallest.

smallest_now( [Queue|Queues], SmallestSoFar ) =
  @smallest_now( Queues, @smaller(Queue,SmallestSoFar) ).

smaller( Queue1, Queue2 ) = Queue1 :-
  Queue1 @ real_size < Queue2 @ real_size,
  !.

smaller( _, Queue2 ) = Queue2.
```

Whenever a teller becomes free the next customer is removed from her/his queue. However, that customer is still considered as if they were still in the queue until after they have been served as this influences which queue new arrivals will join. When the customer eventually exits the bank their queue might then become the smallest.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% METHOD : next_customer/1
% COMMENT: Pop a customer from the front of queue
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

next_customer( Item ) :-
  super <- pop( Item ).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% METHOD : exit_customer/0
% COMMENT: Now we can reduce the "real size" of the
%          queue !
%          Does it become the smallest queue ?
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

exit_customer :-
  real_size -= 1,
  smallest := smaller( self, smallest ).
```

At each tick of the clock the "real" size of the queue is added to its cumulative size over time.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% METHOD : after_tick/0
% COMMENT: What happens to queues after a tick of
%          the clock
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
after_tick :-
  cumulative_size += real_size .
```

At the end of the simulation the *print* message is broadcast to all parts of the model. Here the statistics of each queue are printed.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% METHOD : print/0
% COMMENT: Print statistics about the customer queues
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

print :-
  write( 'Number of customer queues                 : ' ),
  write( @length( all instance customer_queue ) ),
  nl,
  nl,
  all instance customer_queue <- print_me.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% METHOD : print_me/0
% STYLE  : PRIVATE PROCEDURE
% COMMENT: Print statistics about a particular queue
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

print_me :-
  write( 'Queue #' ),
  write( @queue_number ),
  nl,
  write( 'Total number of customers in queue        : ' ),
  write( @aggregate_size ),
  nl,
  write( 'Maximum number of customers in queue      : ' ),
  write( @peak_size ),
  nl,
  write( 'Average number of customers in queue      : ' ),
  write( @average_size ),
  nl,
  write( 'Average time for each customer in queue   : ' ),
  write( @average_wait ),
  nl,
  nl.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% METHOD : average_size//0
% COMMENT: The average size of the queue over time
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

average_size is nearest_number( cumulative_size / clock@time_now ).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% METHOD : average_wait//0
% COMMENT: The average waiting time each customer
%          spends in the queue
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

average_wait is 0 :-
  @aggregate_size = 0,
  !.

average_wait is nearest_number( cumulative_size / aggregate_size ).

end customer_queue.
```

The Output

The following listings were cut from two typical runs of the simulation. In the first run a single communal queue was used and in the second run individual queues were used for each of 5 bank tellers.

Single, Communal Queue

The bank is set up with a communal model where all of the tellers draw customers from a single queue.

```
?- bank <- setup( communal ) .

Teller #1 draws customers from queue #1
Teller #2 draws customers from queue #1
Teller #3 draws customers from queue #1
Teller #4 draws customers from queue #1
Teller #5 draws customers from queue #1
```

A simulation run with the output at each tick of the clock.

```
?- bank<-simulation .

Time: 1
Next customer #1 with work-load 2 joins queue #1 (length = 0)
Teller #1 starts serving customer #1

Time: 2
Next customer #2 with work-load 2 joins queue #1 (length = 1)
Teller #2 starts serving customer #2

Time: 3
Next customer #3 with work-load 3 joins queue #1 (length = 2)
Teller #3 starts serving customer #3

Time: 4
```

```
Next customer #4 with work-load 1 joins queue #1 (length = 3)
Teller #4 starts serving customer #4

Time: 5

Time: 6
Next customer #5 with work-load 1 joins queue #1 (length = 4)
Next customer #6 with work-load 2 joins queue #1 (length = 5)
Next customer #7 with work-load 1 joins queue #1 (length = 6)
Teller #1 finish serving customer #1
Teller #2 finish serving customer #2
Teller #5 starts serving customer #5

Time: 7
Next customer #8 with work-load 2 joins queue #1 (length = 5)
Teller #1 starts serving customer #6
Teller #2 starts serving customer #7

Time: 8
Next customer #9 with work-load 2 joins queue #1 (length = 6)
Teller #3 finish serving customer #3
Teller #4 finish serving customer #4

Time: 9
Next customer #10 with work-load 2 joins queue #1 (length = 5)
Teller #3 starts serving customer #8
Teller #4 starts serving customer #9

Time: 10
Next customer #11 with work-load 2 joins queue #1 (length = 6)
Teller #1 finish serving customer #6
Teller #5 finish serving customer #5

Time: 11
Next customer #12 with work-load 3 joins queue #1 (length = 5)
Teller #1 starts serving customer #10
Teller #5 starts serving customer #11

Time: 12
Next customer #13 with work-load 2 joins queue #1 (length = 6)
Teller #2 finish serving customer #7

Time: 13
Next customer #14 with work-load 2 joins queue #1 (length = 6)
Teller #2 starts serving customer #12
Teller #3 finish serving customer #8
Teller #4 finish serving customer #9

Time: 14
```

```
Next customer #15 with work-load 2 joins queue #1 (length = 5)
Next customer #16 with work-load 2 joins queue #1 (length = 6)
Teller #3 starts serving customer #13
Teller #4 starts serving customer #14

Time: 15
Next customer #17 with work-load 1 joins queue #1 (length = 7)
Next customer #18 with work-load 3 joins queue #1 (length = 8)
Teller #1 finish serving customer #10

Time: 16
Teller #1 starts serving customer #15

Time: 17
Next customer #19 with work-load 1 joins queue #1 (length = 8)
Teller #3 finish serving customer #13

Time: 18
Teller #3 starts serving customer #16
Teller #5 finish serving customer #11

Time: 19
Next customer #20 with work-load 2 joins queue #1 (length = 7)
Teller #2 finish serving customer #12
Teller #5 starts serving customer #17

Time: 20
Next customer #21 with work-load 3 joins queue #1 (length = 7)
Teller #1 finish serving customer #15
Teller #2 starts serving customer #18
Teller #4 finish serving customer #14

Time: 21
Teller #1 starts serving customer #19
Teller #3 finish serving customer #16
Teller #4 starts serving customer #20

Time: 22
Teller #3 starts serving customer #21

Time: 23
Teller #5 finish serving customer #17

Time: 24
Teller #1 finish serving customer #19

Time: 25
```

```
Time: 26

Time: 27
Teller #2 finish serving customer #18
Teller #3 finish serving customer #21
Teller #4 finish serving customer #20
```

A print of the statistics which have been gathered.

```
?- bank <- print .

Bank Simulation Statistics
==========================

Closing time                              : 20
Last customer served at                   : 27

Distribution of customer arrivals         : [0, 0, 0, 0, 1, 1, 1,
                                             1, 1, 1, 2, 2, 3]
Mean work-load for customers arriving     : 2
Variation in the work-load of customers   : 1
Total number of customers which arrived   : 21
Total work-load of all customers          : 41

Number of bank tellers                    : 5
Mode of operation                         : communal queue(s)

Teller # 1
Total number of customers served          : 5
Percentage time spent serving customers   : 89%
Mean time to serve a customer             : 2
Variation in service time                 : 2

Teller # 2
Total number of customers served          : 4
Percentage time spent serving customers   : 96%
Mean time to serve a customer             : 3
Variation in service time                 : 2

Teller # 3
Total number of customers served          : 5
Percentage time spent serving customers   : 93%
Mean time to serve a customer             : 2
Variation in service time                 : 1

Teller # 4
Total number of customers served          : 4
Percentage time spent serving customers   : 89%
Mean time to serve a customer             : 3
Variation in service time                 : 1
```

```
Teller # 5
Total number of customers served          : 3
Percentage time spent serving customers   : 67%
Mean time to serve a customer             : 3
Variation in service time                 : 2

Number of customer queues                 : 1

Queue #1
Total number of customers in queue        : 21
Maximum number of customers in queue      : 4
Average number of customers in queue      : 6
Average time for each customer in queue   : 7
```

Multiple, Personal Queues

The bank is set up with a personal model where each of the tellers will draw customers from their own individual queue.

```
?- bank <- setup( personal ) .

Teller #1 draws customers from queue #1
Teller #2 draws customers from queue #2
Teller #3 draws customers from queue #3
Teller #4 draws customers from queue #4
Teller #5 draws customers from queue #5
```

A simulation run with the output at each tick of the clock.

```
?- bank <- simulation .

Time: 1
Next customer #1 with work-load 2 joins queue #2 (length = 0)
Next customer #2 with work-load 2 joins queue #1 (length = 0)
Teller #1 starts serving customer #2
Teller #2 starts serving customer #1

Time: 2
Next customer #3 with work-load 1 joins queue #3 (length = 0)
Teller #3 starts serving customer #3

Time: 3
Next customer #4 with work-load 2 joins queue #4 (length = 0)
Teller #4 starts serving customer #4

Time: 4
Next customer #5 with work-load 2 joins queue #5 (length = 0)
Teller #1 finish serving customer #2
```

```
Teller #5 starts serving customer #5

Time: 5
Next customer #6 with work-load 2 joins queue #1 (length = 0)
Teller #1 starts serving customer #6
Teller #2 finish serving customer #1

Time: 6
Next customer #7 with work-load 2 joins queue #2 (length = 0)
Next customer #8 with work-load 2 joins queue #1 (length = 1)
Teller #2 starts serving customer #7
Teller #3 finish serving customer #3

Time: 7

Time: 8
Next customer #9 with work-load 2 joins queue #3 (length = 0)
Teller #3 starts serving customer #9
Teller #4 finish serving customer #4
Teller #5 finish serving customer #5

Time: 9
Next customer #10 with work-load 2 joins queue #4 (length = 0)
Teller #1 finish serving customer #6
Teller #4 starts serving customer #10

Time: 10
Next customer #11 with work-load 2 joins queue #5 (length = 0)
Teller #1 starts serving customer #8
Teller #5 starts serving customer #11

Time: 11
Next customer #12 with work-load 3 joins queue #1 (length = 1)
Teller #3 finish serving customer #9

Time: 12
Next customer #13 with work-load 2 joins queue #3 (length = 0)
Next customer #14 with work-load 3 joins queue #2 (length = 1)
Teller #2 finish serving customer #7
Teller #3 starts serving customer #13

Time: 13
Next customer #15 with work-load 2 joins queue #2 (length = 1)
Teller #2 starts serving customer #14

Time: 14
Next customer #16 with work-load 2 joins queue #3 (length = 1)
Next customer #17 with work-load 2 joins queue #4 (length = 1)
Teller #1 finish serving customer #8
```

```
Teller #4 finish serving customer #10

Time: 15
Teller #1 starts serving customer #12
Teller #4 starts serving customer #17
Teller #5 finish serving customer #11

Time: 16
Next customer #18 with work-load 3 joins queue #5 (length = 0)
Teller #3 finish serving customer #13
Teller #5 starts serving customer #18

Time: 17
Next customer #19 with work-load 3 joins queue #1 (length = 1)
Teller #3 starts serving customer #16

Time: 18
Next customer #20 with work-load 2 joins queue #3 (length = 1)

Time: 19
Teller #2 finish serving customer #14
Teller #4 finish serving customer #17

Time: 20
Next customer #21 with work-load 2 joins queue #4 (length = 0)
Teller #2 starts serving customer #15
Teller #3 finish serving customer #16
Teller #4 starts serving customer #21

Time: 21
Teller #1 finish serving customer #12
Teller #3 starts serving customer #20

Time: 22
Teller #1 starts serving customer #19
Teller #5 finish serving customer #18

Time: 23

Time: 24
Teller #3 finish serving customer #20

Time: 25
Teller #4 finish serving customer #21

Time: 26
Teller #2 finish serving customer #15
```

```
Time: 27

Time: 28
Teller #1 finish serving customer #19
```

A print of the statistics which have been gathered.

```
?- bank <- print .

Bank Simulation Statistics
==========================

Closing time                              : 20
Last customer served at                   : 28

Distribution of customer arrivals         : [0, 0, 0, 0, 1, 1, 1,
                                             1, 1, 1, 2, 2, 3]
Mean work-load for customers arriving     : 2
Variation in the work-load of customers   : 1
Total number of customers which arrived   : 21
Total work-load of all customers          : 45

Number of bank tellers                    : 5
Mode of operation                         : personal queue(s)

Teller # 1
Total number of customers served          : 5
Percentage time spent serving customers   : 100%
Mean time to serve a customer             : 2
Variation in service time                 : 2

Teller # 2
Total number of customers served          : 4
Percentage time spent serving customers   : 93%
Mean time to serve a customer             : 3
Variation in service time                 : 2

Teller # 3
Total number of customers served          : 5
Percentage time spent serving customers   : 79%
Mean time to serve a customer             : 2
Variation in service time                 : 1

Teller # 4
Total number of customers served          : 4
Percentage time spent serving customers   : 82%
Mean time to serve a customer             : 3
Variation in service time                 : 1

Teller # 5
```

```
Total number of customers served          : 3
Percentage time spent serving customers   : 64%
Mean time to serve a customer             : 3
Variation in service time                 : 2

Number of customer queues                 : 5

Queue #1
Total number of customers in queue        : 5
Maximum number of customers in queue      : 1
Average number of customers in queue      : 1
Average time for each customer in queue   : 8

Queue #2
Total number of customers in queue        : 4
Maximum number of customers in queue      : 2
Average number of customers in queue      : 1
Average time for each customer in queue   : 9

Queue #3
Total number of customers in queue        : 5
Maximum number of customers in queue      : 1
Average number of customers in queue      : 1
Average time for each customer in queue   : 6

Queue #4
Total number of customers in queue        : 4
Maximum number of customers in queue      : 1
Average number of customers in queue      : 1
Average time for each customer in queue   : 6

Queue #5
Total number of customers in queue        : 3
Maximum number of customers in queue      : 1
Average number of customers in queue      : 1
Average time for each customer in queue   : 6
```

Chapter 6 - A Case Study In Resource Management

This case study will apply Prolog++ to the area of resource management. The resources will be represented as instance hierarchies, utilising default values and inheritance to classify them. A high degree of polymorphism is incorporated with variations of the same method being defined in related classes.

An important feature of Prolog++ exhibited by this study is the ability to store database records locally, rather than having a global repository. This can be extended by having several local databases distributed throughout the instance hierarchy.

The example will also illustrate the use of daemons to monitor the progress made by the scheduling algorithm. This use of daemons can be thought of as a debugging tool that is tailored to the specific requirements of the problem.

The Problem

Every school has the perennial task of scheduling its most important resource, its teachers. The problem varies from year to year, with different classes, the arrival and departure of teachers and perhaps even fundamental changes which put different constraints on the timetable.

Each teacher and each class has its own requirements which must be fulfilled by the timetable. Classes should be taught a broad range of subjects; a timetable that continually scheduled mathematics might be desired by some members of the class, but certainly not by the majority. Teachers usually have expertise in a small range of subjects, and desire certain free periods when they can pursue other activities.

In this prototype two specific constraints will be applied to the timetable. The first constraint is that teachers cannot physically be in two places at the same time, however hard they try. The second constraint is that each class must be taught a different subject for each period of a particular day.

The information about each teacher's subjects and desired free time will be represented as an instance hierarchy, where default information is put at the most general level.

The Classes

The main entities in this problem are the generic teacher, form, subject and period. These are then specialized with particular teachers, particular forms, particular subjects and particular periods. The set of periods is considered to represent a single day in the school timetable.

The scheduling algorithm will be implemented in a separate class called timetable, which gathers information by sending messages to and asking values of the other objects. The result is stored as a local database of 4-tuples, where each tuple consists of the form, period, teacher and subject.

The "timetable" Class

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Set up & create a timetable satisfying all of the
% constraints
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

class timetable .

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                      DECLARATIONS                                      %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

category
  school ,       % A class in the school timetable
  user .         % Also, a user class

inherit
  db .           % Inherit the database stuff!

public methods
  setup /  0 ,   % Set up the teachers, subjects,
                 % forms & periods for this school
  make  /  0 ,   % M_ake the timetable according to
                 % the school setup
  make  /  1 ,   % Make with max. depth of swaps
  print /  0 .   % Print from different perspectives

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                      DEFINITIONS                                       %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% METHOD : setup/0
% COMMENT: Print from different perspectives
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

print :-
  [form,period,teacher,subject] <- print.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% METHOD : setup/0
% COMMENT: Set up the teachers, subjects, forms &
%          periods for this school
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

setup :-
  [form,period,teacher,subject] <- delete_all,
```

```
  form
  <- (
     create( _, [], first_year  ),
     create( _, [], second_year ),
     create( _, [], third_year  ),
     create( _, [], fourth_year )
     ),
  period
  <- (
     create( _, [], 1 ),
     create( _, [], 2 ),
     create( _, [], 3 ),
     create( _, [], 4 ),
     create( _, [], 5 )
     ),
  teacher
  <- (
     create( _, [ subjects=[french,biology],
                  freetime=[1,4]          ], nicky ),
     create( _, [ subjects=[maths,music],
                  freetime=[]            ], brian ),
     create( _, [ subjects=[maths],
                  freetime=[]             ], dave  ),
     create( _, [ subjects=[french,prolog],
                  freetime=[2,3,5]        ], clive ),
     create( _, [ subjects=[accountancy],
                  freetime=[2,4]          ], diane ),
     create( _, [ subjects=[maths,'prolog++'],
                  freetime=[3]            ], phil  )
     ),
  subject
  <- (
     create( _, [], maths       ),
     create( _, [], music       ),
     create( _, [], french      ),
     create( _, [], prolog      ),
     create( _, [], biology     ),
     create( _, [], 'prolog++'  ),
     create( _, [], accountancy )
     ).
when_created :-
  write( 'Create ' ),
  write( class ),
  write( ' : ' ),
  write( mnemonic ),
  nl.
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% METHOD : make/0/1
% COMMENT: Make the timetable according to how the
%          school was setup
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

make :-
  make( 3 ).

make( Effort ) :-
  length( E, Effort ),
  dynamic( entry / 4 ),
  forall
    (
      Form   = instance form,
      Period = instance period
    )
  do
    fill_entry( E, Form, Period, Teacher, Subject ).


% An (un)filled timetable entry

unfilled_entry( Form, Period ) :-
  % select a form
    Form = instance form,
  % select a period
    Period = instance period,
  % which has not yet been assigned
    \+ filled_entry( Form, Period, _, _ ).

filled_entry( Form, Period, Teacher, Subject ) :-
  call( entry( Form, Period, Teacher, Subject ) ).


% Fill an entry by finding one & asserting it

fill_entry( E, Form, Period, Teacher, Subject ) :-
  find_entry( E, Form, Period, Teacher, Subject ),
  !,
  assert( Form, Period, Teacher, Subject ).

fill_entry( _, _, _, _, _ ).
```

```
% A good entry in the timetable

find_entry( _, Form, Period, Teacher, Subject ) :-
  % select a teacher
    Teacher = instance teacher,
  % who can teach during this period
    Teacher <- teach_period( mnemonic Period ),
  % and who is not already elsewhere
    \+ filled_entry( _, Period, Teacher, _ ),
  % select a subject
    Subject = instance subject,
  % which the teacher can teach
    Teacher <- teach_subject( mnemonic Subject ),
  % and which is not already taught to the form
    \+ filled_entry( Form, _, _, Subject ).


% Swap a teacher from another form to this form

find_entry( [_|E], FormA, Period, TeacherA, SubjectA ) :-
  % select a teacher
    TeacherA = instance teacher,
  % who can teach during this period
    TeacherA <- teach_period( mnemonic Period ),
  % but is already teaching some other form
    filled_entry( FormB, Period, TeacherA, _ ),
  % select a subject
    SubjectA = instance subject,
  % which the teacher can teach
    TeacherA <- teach_subject( mnemonic SubjectA ),
  % and which is not already taught to the form
    \+ filled_entry( FormA, _, _, SubjectA ),
  % re-fill the entry for that other form ?
    find_entry( E, FormB, Period, TeacherB, SubjectB ),
  % with a different teacher
    TeacherB \= TeacherA,
  % un-assign our teacher from that other form
  % and re-assign the different teacher
    write( 'Swap teacher ...' ), nl,
    retract( FormB, Period, TeacherA, _ ),
    assert(  FormB, Period, TeacherB, SubjectB ).
```

```
% Swap a subject from another period to this period

find_entry( [_|E], Form, PeriodA, TeacherA, SubjectA ) :-
  % select a teacher
    TeacherA = instance teacher,
  % who can teach during this period
    TeacherA <- teach_period( mnemonic PeriodA ),
  % and who is not already elsewhere
    \+ filled_entry( _, PeriodA, TeacherA, _ ),
  % select a subject
    SubjectA = instance subject,
  % which the teacher can teach
    TeacherA <- teach_subject( mnemonic SubjectA ),
  % and which is already taught to the form
    filled_entry( Form, PeriodB, _, SubjectA ),
  % re-fill the entry for that other period ?
    find_entry( E, Form, PeriodB, TeacherB, SubjectB ),
  % with a different subject
    SubjectA \= SubjectB,
  % un-assign the subject from that other period
  % and re-assign the different subject
    write( 'Swap subject ...' ), nl,
    retract( Form, PeriodB, _, SubjectA ),
    assert( Form, PeriodB, TeacherB, SubjectB ).


% Local database management with echoing

assert( Form, Period, Teacher, Subject ) :-
  super <- assert( entry(Form,Period,Teacher,Subject) ),
  write( '+  ' ),
  write( mnemonic Form    ), write( ' - ' ),
  write( mnemonic Period  ), write( ' - ' ),
  write( mnemonic Teacher ), write( ' - ' ),
  write( mnemonic Subject ), nl.

retract( Form, Period, Teacher, Subject ) :-
  super <- retract( entry(Form,Period,Teacher,Subject) ),
  write( '-  ' ),
  write( mnemonic Form    ), write( ' - ' ),
  write( mnemonic Period  ), write( ' - ' ),
  write( mnemonic Teacher ), write( ' - ' ),
  write( mnemonic Subject ), nl.


end timetable.
```

The form Class

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% General attributes & methods for all forms
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

class form .

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                      DECLARATIONS                                      %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

category
  school ,       % A class in the school timetable
  user .         % Also, a user class

inherit
  timetable .    % Inherit from the timetable class

public methods
  print        / 0 ,  % Print the complete timetable
                      % from the pupil's viewpoint
  print_period / 1 .  % Print the pupil's timetable
                      % for a specific period

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                      DEFINITIONS                                       %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% METHOD : print/0
% COMMENT: Print the complete timetable from the
%          pupil's viewpoint
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

print :-
  isa_class,
  !,
  nl, write( 'FORM TIMETABLE ...' ), nl, nl,
  all instance self <- print, nl.

print :-
  isa_instance,
  write( 'FORM: ' ), write( mnemonic ), nl,
  all instance period <- print_form( self ), nl.



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% METHOD : print_period/1
% COMMENT: Print the pupil's timetable for a
```

```
%          specific period
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

print_period( Period ) :-
  timetable <- filled_entry( self, Period, Teacher, Subject ),
  !,
  write( mnemonic self    ), write( ': ' ),
  write( mnemonic Teacher ), write( ' teaches ' ),
  write( mnemonic Subject ), nl.

print_period( _ ) :-
  write( mnemonic self    ), write( ':' ), nl.

end form.
```

The "period" Class

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% General attributes & methods for all periods
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

class period .

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                     DECLARATIONS                     %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

category
  school ,     % A class in the school timetable
  user .       % Also, a user class

inherit
  timetable .    % Inherit from the timetable class

public methods
  print         /  0 ,  % Print complete timetable
                        % from the period viewpoint
  print_teacher /  1 ,  % Print entry for a specific
                        % teacher in this period
  print_form    /  1 ,  % Print entry for a specific
                        % form    in this period
  print_subject /  1 .  % Print entry for a specific
                        % subject in this period



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                     DEFINITIONS                      %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% METHOD : print/0
% COMMENT: Print the complete timetable from the
%          period viewpoint
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

print :-
  isa_class,
  !,
  nl, write( 'PERIOD TIMETABLE ...' ), nl, nl,
  all instance self <- print, nl.

print :-
  isa_instance,
  write( 'PERIOD: ' ), write( mnemonic ), nl,
  all instance form <- print_period( self ), nl.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% METHOD : print_teacher/1
% COMMENT: Print the timetable entry for a specific
%          teacher in this period
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

print_teacher( Teacher ) :-
  timetable <- filled_entry( Form, self, Teacher, Subject ),
  !,
  write( mnemonic self    ), write( ': teach ' ),
  write( mnemonic Subject ), write( ' to ' ),
  write( mnemonic Form    ), nl.

print_teacher( _ ) :-
  write( mnemonic self    ), write( ':' ), nl.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% METHOD : print_form/1
% COMMENT: Print the timetable entry for a specific
%          form in this period
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

print_form( Form ) :-
  timetable <- filled_entry( Form, self, Teacher, Subject ),
  !,
  write( mnemonic self    ), write( ': ' ),
  write( mnemonic Teacher ), write( ' teaches ' ),
  write( mnemonic Subject ), nl.
print_form( _ ) :-
  write( mnemonic self    ), write( ':' ), nl.
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% METHOD : print_subject/1
% COMMENT: Print the timetable entry for a specific
%          subject in this period
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

print_subject( Subject ) :-
  timetable <- filled_entry( Form, self, Teacher, Subject ),
  write( mnemonic self    ), write( ': ' ),
  write( mnemonic Form    ), write( ' taught by ' ),
  write( mnemonic Teacher ), nl,
  fail.

print_subject( _ ).

end period.
```

The "teacher" Class

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% General attributes & methods for all teachers
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

class teacher .

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                     DECLARATIONS                    %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

category
  school ,        % A class in the school timetable
  user .          % Also, a user class

inherit
  timetable .     % Inherit from the timetable class

public instance attributes
  freetime = [] , % Teachers normally work very hard!
  subjects.

public methods
  teach_period  // 0 , % A period for which the
                       % teacher can be assigned
  teach_subject // 0 , % A subject which the teacher can teach
  print         /  0 . % Print the complete timetable
                       % from the teacher viewpoint
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                     DEFINITIONS                     %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% METHOD : teach_period//0
% COMMENT: A period for which the teacher can be
%          assigned
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

teach_period = P :-
  \+ member( P, @freetime ).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% METHOD : teach_subject//0
% COMMENT: A subject which the teacher can teach
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

teach_subject = S :-
  member( S, @subjects ).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% METHOD : print/0
% COMMENT: Print the complete timetable from the
%          teacher viewpoint
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

print :-
  isa_class,
  !,
  nl, write( 'TEACHER TIMETABLE ...' ), nl, nl,
  all instance self <- print,
  nl.

print :-
  isa_instance,
  write( 'TEACHER: ' ), write( mnemonic ), nl,
  all instance period <- print_teacher( self ), nl.

end teacher.
```

The "subject" Class

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% General attributes & methods for all subjects
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

class subject .

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                      DECLARATIONS                               %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
category
  school ,          % A class in the school timetable
  user .            % Also, a user class

inherit
  timetable .       % Inherit from the timetable class

public methods
  print / 0 .       % Print the complete timetable
                    % from the subjects' viewpoints

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                    DEFINITIONS                      %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% METHOD : print/0
% COMMENT: Print the complete timetable from the
%          subject viewpoint
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

print :-
  isa_class,
  !,
  nl, write( 'SUBJECT TIMETABLE ...' ), nl, nl,
  all instance self <- print, nl.

print :-
  isa_instance,
  write( 'SUBJECT: ' ), write( mnemonic ), nl,
  all instance period <- print_subject( self ), nl.

end subject.
```

The Output

The same timetable was constructed twice with different degrees of effort when trying to resolve conflicts. By increasing the effort it was possible to fully complete the timetable.

Setting Up The Timetable

The setup phase creates the timetable objects with their individual characteristics.

```
:- timetable <- setup .


Create form : first_year
Create form : second_year
Create form : third_year
```

```
Create form : fourth_year
Create period : 1
Create period : 2
Create period : 3
Create period : 4
Create period : 5
Create teacher : nicky
Create teacher : brian
Create teacher : dave
Create teacher : clive
Create teacher : diane
Create teacher : phil
Create subject : maths
Create subject : music
Create subject : french
Create subject : prolog
Create subject : biology
Create subject : prolog++
Create subject : accountancy
```

Less Effort

In this run the algorithm searches up to 2 levels deep when trying to resolve conflicts. As a result of this restriction it was not possible to complete the timetable.

```
?- timetable <- make( 2 ) .

+  first_year - 1 - brian - maths
+  first_year - 2 - nicky - french
+  first_year - 3 - nicky - biology
+  first_year - 4 - brian - music
+  first_year - 5 - diane - accountancy
+  second_year - 1 - dave - maths
+  second_year - 2 - brian - music
+  second_year - 3 - diane - accountancy
+  second_year - 4 - clive - french
+  second_year - 5 - nicky - biology
+  third_year - 1 - clive - french
+  third_year - 2 - dave - maths
+  third_year - 3 - brian - music
+  third_year - 4 - phil - prolog++
Swap teacher ...
-  second_year - 5 - nicky - biology
+  second_year - 5 - phil - prolog++
+  third_year - 5 - nicky - biology
+  fourth_year - 1 - diane - accountancy
+  fourth_year - 2 - phil - maths
Swap subject ...
-  first_year - 1 - brian - maths
+  first_year - 1 - phil - prolog++
```

```
Swap teacher ...
-  first_year - 3 - nicky - biology
+  first_year - 3 - dave - maths
+  fourth_year - 3 - nicky - french
+  fourth_year - 5 - brian - music
```

The partially completed timetable is …

```
:- timetable <- print .

FORM TIMETABLE ...

FORM: first_year
1: phil teaches prolog++
2: nicky teaches french
3: dave teaches maths
4: brian teaches music
5: diane teaches accountancy

FORM: second_year
1: dave teaches maths
2: brian teaches music
3: diane teaches accountancy
4: clive teaches french
5: phil teaches prolog++

FORM: third_year
1: clive teaches french
2: dave teaches maths
3: brian teaches music
4: phil teaches prolog++
5: nicky teaches biology

FORM: fourth_year
1: diane teaches accountancy
2: phil teaches maths
3: nicky teaches french
4:
5: brian teaches music
```

etc.

More Effort

In this run the effort was increased to searching 5 levels deep and as a result it was able to complete the timetable.

```
?- timetable <- make( 5 ) .
```

```
+  first_year - 1 - brian - maths
+  first_year - 2 - nicky - french
+  first_year - 3 - nicky - biology
+  first_year - 4 - brian - music
+  first_year - 5 - diane - accountancy
+  second_year - 1 - dave - maths
+  second_year - 2 - brian - music
+  second_year - 3 - diane - accountancy
+  second_year - 4 - clive - french
+  second_year - 5 - nicky - biology
+  third_year - 1 - clive - french
+  third_year - 2 - dave - maths
+  third_year - 3 - brian - music
+  third_year - 4 - phil - prolog++
Swap teacher ...
-  second_year - 5 - nicky - biology
+  second_year - 5 - phil - prolog++
+  third_year - 5 - nicky - biology
+  fourth_year - 1 - diane - accountancy
+  fourth_year - 2 - phil - maths
Swap subject ...
-  first_year - 1 - brian - maths
+  first_year - 1 - phil - prolog++
Swap teacher ...
-  first_year - 3 - nicky - biology
+  first_year - 3 - dave - maths
+  fourth_year - 3 - nicky - french
Swap teacher ...
-  fourth_year - 1 - diane - accountancy
+  fourth_year - 1 - brian - music
Swap teacher ...
-  third_year - 1 - clive - french
+  third_year - 1 - diane - accountancy
Swap subject ...
-  second_year - 1 - dave - maths
+  second_year - 1 - clive - prolog
Swap teacher ...
-  second_year - 4 - clive - french
+  second_year - 4 - dave - maths
Swap teacher ...
-  first_year - 4 - brian - music
+  first_year - 4 - clive - prolog
+  fourth_year - 4 - brian - music
Swap teacher ...
-  first_year - 5 - diane - accountancy
+  first_year - 5 - brian - music
+  fourth_year - 5 - diane - accountancy
```

The completed timetable is ...

```
:- timetable <- print .

FORM TIMETABLE ...

FORM: first_year
1: phil teaches prolog++
2: nicky teaches french
3: dave teaches maths
4: clive teaches prolog
5: brian teaches music

FORM: second_year
1: clive teaches prolog
2: brian teaches music
3: diane teaches accountancy
4: dave teaches maths
5: phil teaches prolog++

FORM: third_year
1: diane teaches accountancy
2: dave teaches maths
3: brian teaches music
4: phil teaches prolog++
5: nicky teaches biology

FORM: fourth_year
1: brian teaches music
2: phil teaches maths
3: nicky teaches french
4: brian teaches music
5: diane teaches accountancy


PERIOD TIMETABLE ...

PERIOD: 1
first_year: phil teaches prolog++
second_year: clive teaches prolog
third_year: diane teaches accountancy
fourth_year: brian teaches music

PERIOD: 2
first_year: nicky teaches french
second_year: brian teaches music
third_year: dave teaches maths
fourth_year: phil teaches maths

PERIOD: 3
```

```
first_year: dave teaches maths
second_year: diane teaches accountancy
third_year: brian teaches music
fourth_year: nicky teaches french

PERIOD: 4
first_year: clive teaches prolog
second_year: dave teaches maths
third_year: phil teaches prolog++
fourth_year: brian teaches music

PERIOD: 5
first_year: brian teaches music
second_year: phil teaches prolog++
third_year: nicky teaches biology
fourth_year: diane teaches accountancy


TEACHER TIMETABLE ...

TEACHER: nicky
1:
2: teach french to first_year
3: teach french to fourth_year
4:
5: teach biology to third_year

TEACHER: brian
1: teach music to fourth_year
2: teach music to second_year
3: teach music to third_year
4: teach music to fourth_year
5: teach music to first_year

TEACHER: dave
1:
2: teach maths to third_year
3: teach maths to first_year
4: teach maths to second_year
5:

TEACHER: clive
1: teach prolog to second_year
2:
3:
4: teach prolog to first_year
5:

TEACHER: diane
```

```
1: teach accountancy to third_year
2:
3: teach accountancy to second_year
4:
5: teach accountancy to fourth_year

TEACHER: phil
1: teach prolog++ to first_year
2: teach maths to fourth_year
3:
4: teach prolog++ to third_year
5: teach prolog++ to second_year


SUBJECT TIMETABLE ...

SUBJECT: maths
2: third_year taught by dave
2: fourth_year taught by phil
3: first_year taught by dave
4: second_year taught by dave

SUBJECT: music
1: fourth_year taught by brian
2: second_year taught by brian
3: third_year taught by brian
4: fourth_year taught by brian
5: first_year taught by brian

SUBJECT: french
2: first_year taught by nicky
3: fourth_year taught by nicky

SUBJECT: prolog
1: second_year taught by clive
4: first_year taught by clive

SUBJECT: biology
5: third_year taught by nicky

SUBJECT: prolog++
1: first_year taught by phil
4: third_year taught by phil
5: second_year taught by phil

SUBJECT: accountancy
1: third_year taught by diane
```

```
3: second_year taught by diane
5: fourth_year taught by diane
```

Chapter 7 - A Case Study In Database Management

This case study will investigate the application of Prolog++ to small, active databases. Prolog++ inherits all of the database features of Prolog, which can be thought of as a deductive, relational database manager. It augments this with the ability to structure the database as an instance hierarchy, using inheritance to avoid duplication and redundant values.

Since Prolog++, like Prolog, keeps everything resident in memory, it is not recommended for large scale databases. The new generation of DBMS's now emerging, such as Sybase, are incorporating many aspects of OOPS. This is not to say, however, that small databases are not interesting. There are a plethora of small to medium sized businesses that are demanding smart, compact databases. An example of such an application is outlined in the following sections.

The Problem

Bill Hobbs is a stocktaker for the restaurant and public house trade. He has many clients that he visits on a regular basis to count the stock on hand. Using the information gathered, Bill generates a report of the client's trading for that period, including such results as the total actual sales, the expected sales and in particular any shortfall or excess. The service provided by Bill Hobbs is very important to each client, as it keeps a regular check on any pilfering by the staff, and also provides valuable information as to the profitability of the business.

This is a very repetitive and laborious task, and since most restaurants and public houses share the same characteristics it is highly amenable to computerisation. An analysis of the problem showed that it was also suited to an object-oriented representation.

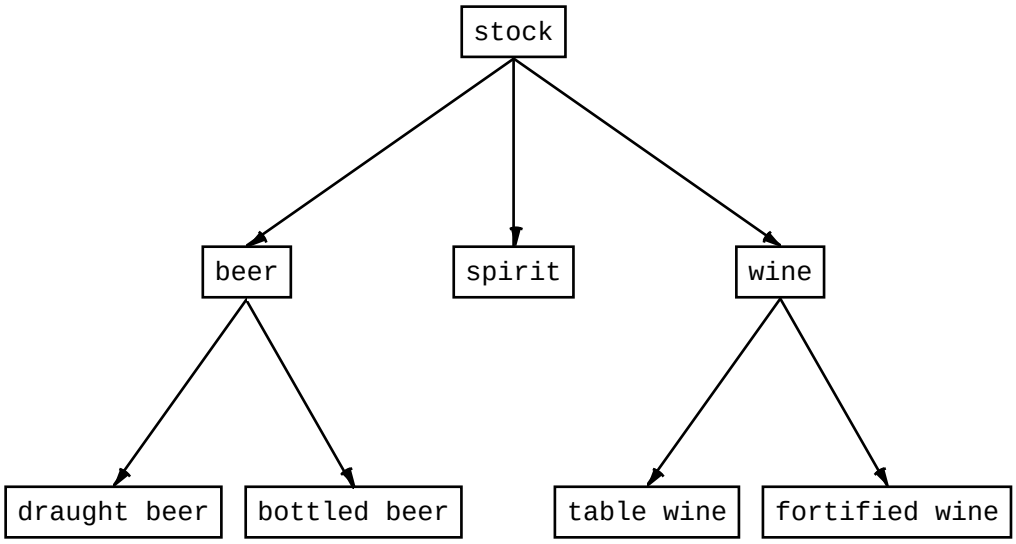
The stock carried by restaurants and public houses falls neatly into a hierarchy of sections, lines and actual stock entries. Broadly, the stock can be divided into beers, wines and sprits (the sections), which are further divided into lines such as draught beers, bottled beers, table wines, fortified wines, brandies, whiskeys, etc.. These demarcations are very important to the client as they give a complete breakdown of his trading patterns.

Each individual item of stock has a fixed set of values, including its cost and selling prices, its current stock level, and others. Each of the lines, sections and overall stock have two important totals, which are its cumulative purchases and its cumulative sales.

Value added tax (VAT) applies to all the business carried out. It is normal practice that cost prices are given exclusive of VAT, and that retail prices are given inclusive of VAT.

The Classes

It was decided to structure the stock as a class hierarchy which completely mirrors the client's view.



The "stock" Class

The stock class represents all the kinds of stock from individual lines such as Guinness to groups such as draughts beers and beers in general.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% CLASS  : stock
% COMMENT: The generic class for all kinds of stock
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

class stock .


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                  DECLARATIONS
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

categories
  user ,
  stock .                    % A class in the stock database


public methods
  setup     /  0 ,   % Set up the stock instances
  report    /  0 ,   % Generate a stock report
  listings  /  0 ,   % Listings of sub-groups & individual lines
  listing   /  0 ,   % Listing of a single line
  summaries /  0 .   % Summaries of all sub-groups
```

```
private methods
  setup              /  1 ,
  listings           /  1 ,
  summaries          /  1 ,
  total_up_classes   /  1 ,
  total_up_instances /  1 ,
  type               // 0 ,
  apply_vat          // 1 .
```

In addition to the primary methods setup/0 and report/0, certain other methods need to be made public so that they are accessible to the group classes. This is because messages such as listings/0 are sent from outside to, for instance, the beer class.

The stock attributes are split into thse which pertain to the group classes and those which pertain to individual instances of lines.

```
public class attributes
  total_costs_excluding_vat is 0 ,
  total_sales_including_vat is 0 .

private class attributes
  total_costs_including_vat is apply_vat(
                                 total_costs_excluding_vat
                                        ) ,
  profit_or_loss            is total_sales_including_vat -
                               total_costs_including_vat .


public instance attributes
  costs_excluding_vat         is bought *
                                 buying_price_excluding_vat ,
  sales_including_vat         is sold   *
                                 selling_price_including_vat .

private instance attributes
  name = self,
  buying_price_excluding_vat  is 0 ,
  selling_price_including_vat is 0 ,
  previous_stock              is 0 ,
  current_stock               is 0 ,
  purchases                   is 0 ,
  credits                     is 0 ,
  bought                      is purchases - credits ,
  sold                        is previous_stock -
                                 current_stock  +
                                 bought ,
  costs_including_vat         is apply_vat( costs_excluding_vat ) .
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                    DEFINITIONS
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% METHOD : setup/0
% COMMENT: Set up the stock instances according to the type of the
%          stock class
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


setup :-
  setup( @type ).


setup( group ) :-
  all sub_class self <- setup .


setup( lines ) :-
  delete_all,
  forall line( Name, Buy, Sell, Old, New, Purchases, Credits )
  do     create(
                _,
                [
                  name = Name,
                  buying_price_excluding_vat  is Buy,
                  selling_price_including_vat is Sell,
                  previous_stock              is Old,
                  current_stock               is New,
                  purchases                   is Purchases,
                  credits                     is Credits
                ]
               ) .
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% METHOD : report/0
% COMMENT: Generate a stock report
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


report :-
  write( '/-------------------------------------------------------------
------\' ), nl,
  write( '|   Buying |  Selling | Previous |  Current |          |
|' ), nl,
  write( '|    Price |    Price |    Stock |    Stock | Purchase |
Credit |' ), nl,
  write( '|-------------------------------------------------------------
------|' ), nl,
  listings,
  write( '\-------------------------------------------------------------
------/' ), nl,
  nl,
  write( '/-----------------------------------------------------\'
), nl,
  write( '|       Costs |       Costs |       Sales | Profit/Loss |'
), nl,
  write( '|  (Excl VAT) |  (Incl VAT) |  (Incl VAT) |  (Incl VAT) |'
), nl,
  write( '|-----------------------------------------------------|'
), nl,
  summaries,
  write( '\-----------------------------------------------------/'
), nl,
  nl.

listings :-
  listings( @type ) .


summaries :-
  summaries( @type ),
  summary .
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Case 1)  This is a general class of stock with further
%          sub-classes
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

listings( group ) :-
  all sub_class self <- listings.


summaries( group ) :-
  all sub_class self <- summaries,
  total_costs_excluding_vat := 0,
  total_sales_including_vat := 0,
  total_up_classes( all sub_class self ).


total_up_classes( [] ).

total_up_classes( [Class|Classes] ) :-
  total_costs_excluding_vat += Class@total_costs_excluding_vat,
  total_sales_including_vat += Class@total_sales_including_vat,
  total_up_classes( Classes ).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Case 2) This is a bottom-level class with actual instances %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


listings( lines ) :-
  write( '|          |          |          |          |          |
|  ' ),
  write( self ), nl,
  all instance self <- listing.


summaries( lines ) :-
  total_costs_excluding_vat := 0,
  total_sales_including_vat := 0,
  total_up_instances( all instance self ).


total_up_instances( [] ).

total_up_instances( [Instance|Instances] ) :-
  total_costs_excluding_vat += Instance@costs_excluding_vat,
  total_sales_including_vat += Instance@sales_including_vat,
  total_up_instances( Instances ).
```

```
listing :-
  write_value( @buying_price_excluding_vat,  9, 2 ),
  write_value( @selling_price_including_vat, 9, 2 ),
  write_value( @previous_stock,              9, 1 ),
  write_value( @current_stock,               9, 1 ),
  write_value( @purchases,                   9, 0 ),
  write_value( @credits,                     9, 0 ),
  write( '|  ' ), write( @name ), nl.


summary :-
  write_value( @total_costs_excluding_vat, 12, 2 ),
  write_value( @total_costs_including_vat, 12, 2 ),
  write_value( @total_sales_including_vat, 12, 2 ),
  write_value( @profit_or_loss,            12, 2 ),
  write( '|  ' ), write( self ), nl.


write_value( Value, Width, Decimals ) :-
  Integer is int( Value ),
  (  Decimals = 0
  -> ( Integer <   10 -> Tab =  6
     ; Integer <  100 -> Tab =  5
     ; Integer < 1000 -> Tab =  4
     ;                   Tab =  3 )
  ;  ( Integer <   10 -> Tab is 5 - Decimals
     ; Integer <  100 -> Tab is 4 - Decimals
     ; Integer < 1000 -> Tab is 3 - Decimals
     ;                   Tab is 2 - Decimals ) ),
  write( '|' ),
  tab( Width+Tab-7 ),
  write( Integer ),
  (  Decimals = 0
  -> true
  ;  write( '.' ),
     Fraction is int( (Value - Integer) * (10^Decimals) + 0.5 ) /
(10^Decimals),
     write_decimals( Decimals, Fraction ) ),
  write( ' ' ).

write_decimals( 0, _ ) :-
  !.
write_decimals( Decimals1, Fraction1 ) :-
  Digit is int( 10 * Fraction1 ),
  write( Digit ),
  Decimals2 is Decimals1 - 1,
  Fraction2 is (10 * Fraction1) - Digit,
  write_decimals( Decimals2, Fraction2 ).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% METHOD : apply_vat//1
% COMMENT: Apply VAT to an exclusive amount to give an inclusive
%          amount
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

apply_vat( Amount ) is Amount * 1.175 .

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% METHOD : type//0
% COMMENT: Determine the type according to whether there are any
%          sub-classes
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

type = group :-
  sub_class self = _,
  !.

type = lines .


end stock .
```

The Sections And Lines

The sections are represented as sub-classes of the generic stock class. The individual lines are represented as instances created through the setup/0 method.

```
class beer .
      categories stock, user .
      inherits   stock .
end   beer .


class beer_draught .
      categories stock, user .
      inherits   beer .
      public method line / 7 .
      line( abbot_ale,  4.20,  8.40, 4.0, 5.0, 11,  0 ).
      line( carlsberg,  4.80,  9.60, 5.0, 2.0, 11,  0 ).
      line( guinness,   5.20, 10.40, 7.0, 4.0, 18,  9 ).
end   beer_draught .

class beer_bottled .
      categories stock, user .
      inherits   beer .
      public method line / 7 .
      line( pale_ale,   8.00, 16.20, 5.0, 3.2, 12,  0 ).
      line( brown_ale,  8.50, 17.40, 7.3, 6.0, 12,  0 ).
```

```
end   beer_bottled .
class wine .
      categories stock, user .
      inherits   stock .
end   wine .

class wine_table .
      categories stock, user .
      inherits   wine .
      public method line / 7 .
      line( claret,     1.95,  4.50,  17,  15, 36,  0 ).
      line( chablis,    5.00, 11.50,  22,  16, 48, 12 ).
end   wine_table .

class wine_fortified .
      categories stock, user .
      inherits   wine .
      public method line / 7 .
      line( martini,    3.25,  9.70, 4.2, 2.7,  0,  0 ).
end   wine_fortified .

class spirit .
      categories stock, user .
      inherits   stock .
      public method line / 7 .
      line( brandy,     8.40, 25.20, 2.4, 1.6,  0,  0 ).
      line( whiskey,    7.50, 22.50, 2.3, 3.2,  6,  0 ).
      line( vodka,      6.95, 20.85, 3.5, 4.2,  6,  0 ).
      line( gin,        7.25, 21.75, 4.1, 2.2,  0,  0 ).
end   spirit .
```

The Output

The stock instances are created through the setup/0 method and then a report generated.

```
?- stock <- setup .
yes

?- stock <- report .
/-------------------------------------------------------------------\
|   Buying |  Selling | Previous |  Current |          |          |
|    Price |    Price |    Stock |    Stock | Purchase |   Credit |
|-------------------------------------------------------------------|
|          |          |          |          |          |          |
beer_draught
|     4.20 |     8.40 |      4.0 |      5.0 |       11 |        0 |
abbot_ale
|     4.80 |     9.60 |      5.0 |      2.0 |       11 |        0 |
carlsberg
```

```
|     5.20 |    10.40 |      7.0 |      4.0 |       18 |        9 |
guinness
|          |          |          |          |          |          |
beer_bottled
|     8.00 |    16.20 |      5.0 |      3.2 |       12 |        0 |
pale_ale
|     8.50 |    17.40 |      7.3 |      6.0 |       12 |        0 |
brown_ale
|          |          |          |          |          |          |
wine_table
|     1.95 |     4.50 |     17.0 |     15.0 |       36 |        0 |
claret
|     5.00 |    11.50 |     22.0 |     16.0 |       48 |       12 |
chablis
|          |          |          |          |          |          |
wine_fortified
|     3.25 |     9.70 |      4.2 |      2.7 |        0 |        0 |
martini
|          |          |          |          |          |          |
spirit
|     8.40 |    25.20 |      2.4 |      1.6 |        0 |        0 |
brandy
|     7.50 |    22.50 |      2.3 |      3.2 |        6 |        0 |
whiskey
|     6.95 |    20.85 |      3.5 |      4.2 |        6 |        0 |
vodka
|     7.25 |    21.75 |      4.1 |      2.2 |        0 |        0 |
gin
\-------------------------------------------------------------------/

/-----------------------------------------------------\
|       Costs |       Costs |       Sales | Profit/Loss |
|  (Excl VAT) |  (Incl VAT) |  (Incl VAT) |  (Incl VAT) |
|-----------------------------------------------------|
|      145.80 |      171.30 |      343.20 |      171.88 |
beer_draught
|      198.00 |      232.65 |      454.98 |      222.33 |
beer_bottled
|      343.80 |      403.97 |      798.18 |      394.20 |  beer
|      250.20 |      293.98 |      654.00 |      360.00 |  wine_table
|        0.00 |        0.00 |       14.55 |       14.55 |
wine_fortified
|      250.20 |      293.98 |      668.55 |      374.56 |  wine
|       86.70 |      101.86 |      286.74 |      184.86 |  spirit
|      680.70 |      799.81 |     1753.46 |      953.65 |  stock
\-----------------------------------------------------/
```

Chapter 8 - A Case Study In Fault Diagnosis

This case study will investigate the application of Prolog++ to the diagnosis of faults. It is intended to illustrate the technique of progressively sending messages through the hierarchy, starting from at a general level and grdually moving down to classes at the lower, more specific levels.

The hierarchy is used to represent the implicit relationships between different classes of faults.

The Problem

We would like to represent the causal-effect relationship between faults and symptoms. The particular domain, that of car maintenance, was chosen to illustrate the techniques in layman terms. Everybody is familiar, to a greater or lesser degree, with the problems that can occur with automobiles.

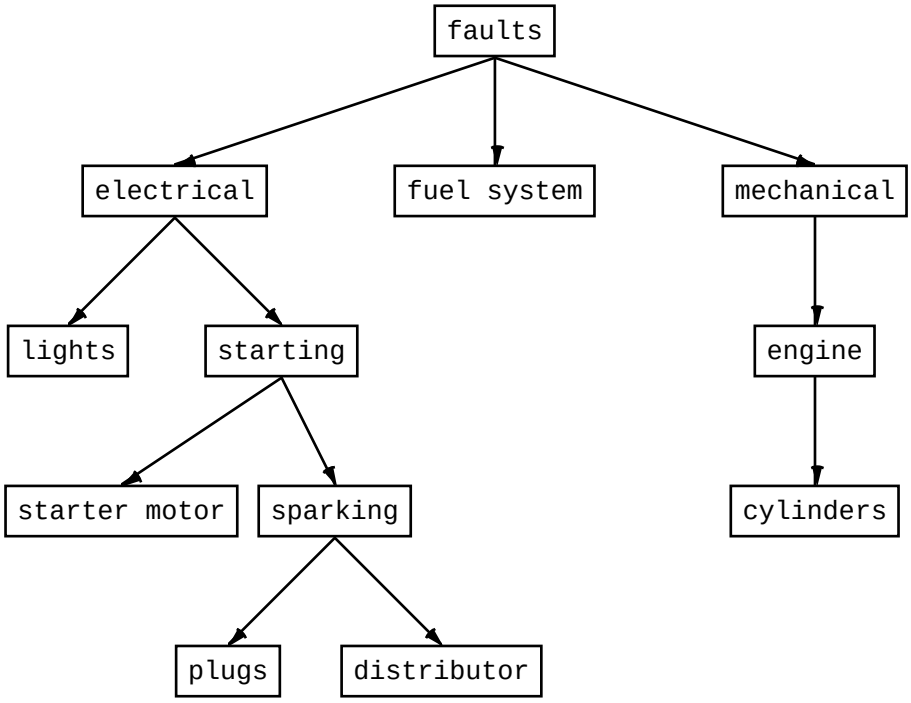
The crucial factor in fault diagnosis is being able to quickly pinpoint the general area of the problem, before focusing in on the root cause. Identifying general aspects of the problem can avoid going down blind alleys, and more importantly avoids asking the user seemingly irrelevant questions.

For the domain of car maintenance, an automobile can be dissected into several problem areas such as the fuel system, mechanical faults and electrical faults. Associated with each area is a collection of faults which may occur, and the symtoms which they cause. Some of the symptoms may be contradictory, in the sense that two symptoms cannot possibly occur simultaneously.

The Classes

From the above discussion two quite separate types of instance can be identified. The first concerns itself with fault diagnosis, including the ability to move from general terms down to specific terms. The second deals with the problem domain, in terms of the causal-effect relationships between actual faults and exhibited symptoms.

The following diagram illustrates a hierarchy for identifying faults in automobiles. At the topmost level is the *faults* class containing the algorithm for finding faults, and which is inherited by all of the other classes in the hierarchy. At any point, however, a class has the option to override the default search algorithm with one that is specialized for its problem area.



The following sections define the protocol (attributes, methods and functions) for the *faults* instance and for the domain specific objects.

The Domain Classes

```
class   mechanical.
inherit fault.
end     mechanical.

class   engine.
inherit mechanical.
end     engine.

class   cylinders.
inherit engine.
end     cylinders.

class   fuel_system.
inherit fault.
end     fuel_system.

class   electrical.
inherit fault.
end     electrical.

class   lights.
inherit electrical.
end     lights.
```

```
class   starting.
inherit electrical.
end     starting.

class   starter_motor.
inherit starting.
end     starter_motor.

class   sparking.
inherit starting.
end     sparking.

class   plugs.
inherit sparking.
end     plugs.

class   distributor.
inherit sparking.
method  fault    // 1.
        fault(   f1001 ) = 'Condensation in the distributor cap'.
        fault(   f1002 ) = 'Faulty distributor arm'.
        fault(   f1003 ) = 'Worn distributor brushes'.
method  symptom  // 1.
        symptom( s1001 ) = 'The starter motor turns but the engine
does not fire'.
        symptom( s1002 ) = 'The engine has difficulty starting'.
        symptom( s1003 ) = 'The engine cuts out shortly after
starting'.
        symptom( s1004 ) = 'The engine cuts out at speed'.
method  effect   // 1.
        effect(  f1001 ) = s1001.
        effect(  f1002 ) = s1001.
        effect(  f1002 ) = s1004.
        effect(  f1003 ) = s1002.
        effect(  f1003 ) = s1003.
method  contrary /  2.
        contrary( s1002, s1001 ).
        contrary( s1003, s1001 ).
end     distributor.
```

The "fault" Class

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% CLASS  : fault
% COMMENT: How to search for and report faults
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

class fault .
```

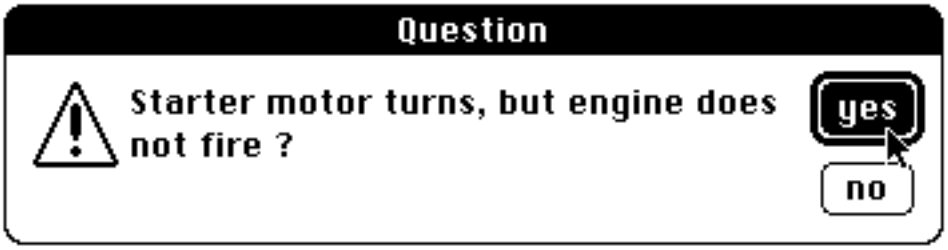
```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                DECLARATIONS                                  %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


category
  fault ,                  % A class in the fault diagnosis
  user .


public methods
  findall         /  0 .   % Find and report all possible faults

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                DEFINITIONS                                   %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% METHOD : findall/0
% COMMENT: Find and print all possible faults
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


findall :-
  dynamic( told_by_user / 2 ),
  forall (
           ( Where = self
           ; Where = descendant_class ),
           Where <- find( Fault )
         )
  do     (
           write( 'Location      : ' ), write( Where ), nl,
           write( 'Possible Fault: ' ), write( Fault ), nl, nl
         ).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% METHOD : find/1
% COMMENT: Find a possible fault for some class
%          Explicit use of 'self' to avoid any early bindings!
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


find( Fault ) :-
  public( fault // 1 ),
  Fault = self@fault( FaultNumber ),
  forall SymptomNumber = self@effect( FaultNumber )
```

```
  do     exhibited( SymptomNumber ).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% METHOD : exhibited/2
% COMMENT: Ask the user whether or not the symptom is
%          exhibited & remember
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


exhibited( SymtomNumber ) :-
  told_by_user( SymtomNumber, Reply ),
  !,
  Reply = yes.

exhibited( SymptomNumber ) :-
  cat( [self@symptom(SymptomNumber),'?'], Question, _ ),
  ( yesno( Question ) -> Reply = yes ; Reply = no ),
  asserta( told_by_user( SymptomNumber, Reply ) ),
  Reply = yes,
  forall ( self <- contrary( SymptomNumber, ContrarySymptom )
         ; self <- contrary( ContrarySymptom, SymptomNumber ) )
  do     asserta( told_by_user( ContrarySymptom, no ) ).


end fault.
```

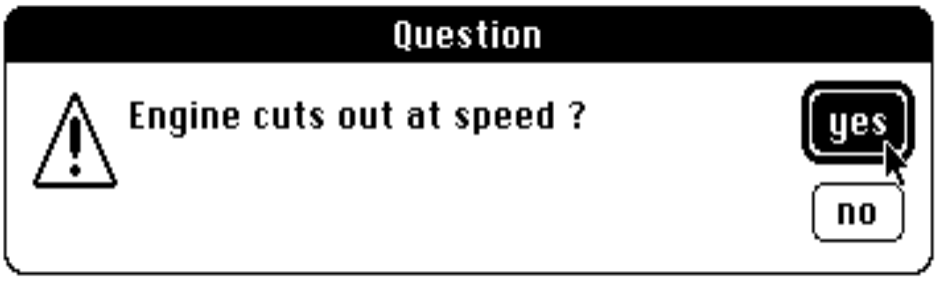
The Output

The following outputs were taken from two fault diagnosis sessions.

```
?- faults <- findall .
```

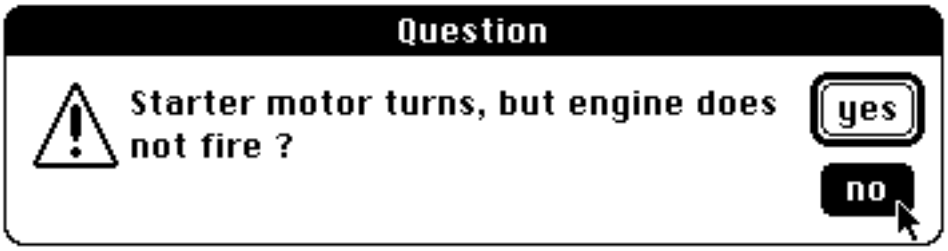


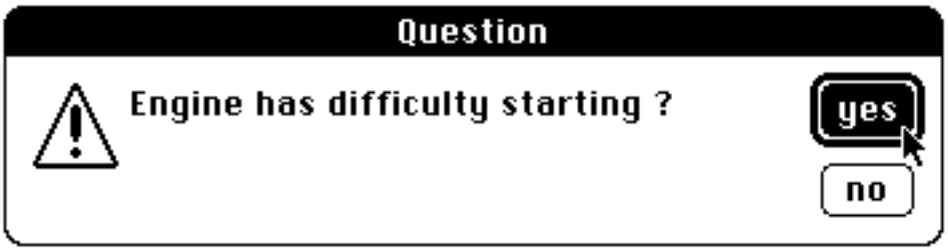

```
Location       : distributor
Possible Fault : Condensation in distributor cap
```

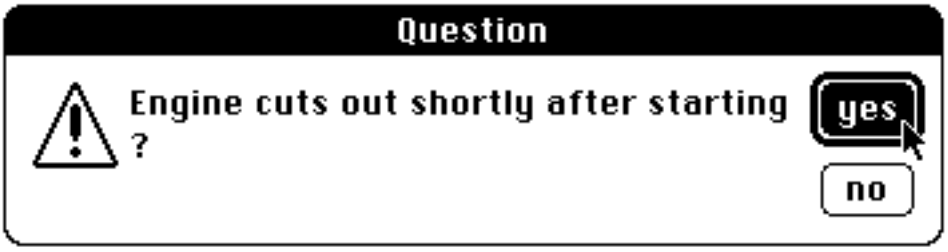



```
Location       : distributor
Possible Fault : Faulty distributor arm

No more solutions
```

?- faults <- findall .







```
Location       : distributor
Possible Fault : Worn distributor brushes

No more solutions
```

Part 3 - Technical Specification Of Prolog++

Chapter 9 - The Prolog++ Language

In this section we present the language of Prolog++. A formal definition of the syntax can be found in the appendices.

Programs

A Prolog++ program is a collection of Prolog++ classes which communicate with each other by sending messages. It is similar to a Prolog program defined as a collection of modules which communicate by calling each other's procedures.

Each class definition in a Prolog++ program is delimited by open and close statements. The open/close statements are of the form :-

class "Name of Class" **.**

end "Name of Class" **.**

where exactly the same class name is mentioned in both statements. As with Prolog, a full stop (**.**) is used to terminate each and every statement. Indeed, all statements must conform to the same syntactic constraints as Prolog clauses[1].

Program Statements

The program statements occurring between the open and close statements define the structure and contents of a class. They fall into two distinct categories; i) code statements which define how various kinds of messages are handled, and ii) meta statements which talk about the class and its methods.

The code statements can be further divided into procedural and functional method definitions. The meta statements cover the areas of class hierarchies, part-of hierarchies, encapsulation and dynamic data. These statements can occur in any order whatsoever, with code and meta information freely intermixed.

Logical Variables

Prolog++ uses the same terminology for logical variables as Prolog. Unquoted atoms beginning with a capital letter (e.g. Element, Stack) or beginning with an underscore (e.g. _1, _xyz) are named variables. An underscore by itself denotes an anonymous variable whose instantiated value is not important.

[1] See the appendices for the syntactic operators declared by Prolog++.

Instance Variables

Prolog++ provides two instance variables represented by specially reserved words of the language. They can be used in any context whatsoever, having the same syntactic status as any other Prolog term.

The instance variables provided by Prolog++ are :-

self substituted at run-time by the instance, or class, to which the message being handled was originally sent

super substituted at compile-time by the super class of the class in which it textually occurs

---

The Is-A Hierarchy

The is-a hierarchy of Prolog++ relates specific classes with more general classes through the mechanism of inheritance.

Is-A Declarations

The is-a relationship between classes is expressed within each class definition by declaring its super or parent class(es).

```
class stack.
      inherits buffer .      % inherits all characteristics of buffers
…
end stack.

class queue_with_statistics.
      inherits queue,              % inherits from general queues
               buffer_with_statistics.  % as well as statistical
buffers
…
end queue_with_statistics.
```

A strict hierarchy is one in which all classes have either a single super class or none at all. Such a hierarchy is said to exhibit singular inheritance. Hierarchies in which at least one class has several super classes is said to exhibit multiple inheritance.

Accessing The Is-A Hierarchy

The is-a hierarchy can be navigated by referring to super, sub, ancestral and descendant classes. This is expressed by a collection of reserved words.

super_class the/a super or parent class

sub_class a sub or child class

ancestor_class a super class (parent), or a super-super class (grandparent), etc.

descendant_class a sub class (child), or a sub-sub class (grandchild), etc.

The Part-Of Hierarchy

The part-of hierarchy of Prolog++ relates class instances with oneanother such that they can be thought of either collectively or individually.

The creation of a complex object such as a bicycle may involve the creation of several parts, including wheels, a frame and handlebars. These in turn may involve the creation of further sub-parts such as spokes and rims.

Part-Of Declarations

The part-of relationship between class instances is expressed within each class definition by declaring its component parts.

```
class bicycle.
      parts frame, wheel * 2, seat, handle_bars.
end   bicycle.

class unicycle.
      inherit bicycle.
      parts wheel, handle_bars * 0.
end   unicycle.

class tandem.
      inherit bicycle.
      parts seat * 2, handle_bars * 2.
end   tandem.
```

The parts declaration can be inherited. For example, in addition to the 2 seats and 2 sets of handle-bars a tandem will inherit a frame and 2 wheels from the bicycle class. The unicycle forces the handle-bars not to be inherited by explicitly declaring 0 for that part.

Creating Composite Instances

Whenever a new instance of a class is created the parts declaration of that class is inspected and the corresponding instances of the sub-parts are created.

For example, consider the various cycle classes above and assume that the others are all sub-classes of *cycle_component*.

```
class cycle_component.

public method when_created/0.
when_created :-
      write( 'Created: ' ),
      write( self ),
      nl.

end cycle_component.
```

The creation of a new bicycle will involve the creation of a frame, two wheels, a seat and a set of handle-bars.

```
?- bicycle <- create(C).

Created: frame|163844
Created: wheel|163824
Created: wheel|163834
Created: seat|163846
Created: handle_bars|163848

C = (bicycle|163823)
```

whereas the creation of a unicycle will only create a wheel, a frame and a seat in addition to the unicycle itself.

```
?- unicycle <- create(C).

Created: wheel|181366
Created: frame|181376
Created: seat|181378

C = (unicycle|181365)
```

Now consider that a wheel is actually made from a rim and some spokes.

```
class wheel.
      inherit cycle_component.
      parts rim, spoke * @spokes.
      class attribute spokes is 3.
end   wheel.

class rim.
      inherit cycle_component.
end   rim.

class spoke.
      inherit cycle_component.
end   spoke.
```

The creation of a unicycle will now create a wheel, a frame and a seat as before, but in addition the wheel will be further sub-divided into a rim and three spokes.

```
?- unicycle <- create(C).

Created: wheel|198130
Created: rim|198132
Created: spoke|198134
Created: spoke|198136
Created: spoke|198138
```

```
Created: frame|198140
Created: seat|198142

C = (unicycle|198129)
```

Accessing The Part-Of Hierarchy

The part-of hierarchy can be navigated by referring to sub parts, super parts and the top-most ancestral part. This is expressed by a collection of reserved words.

composite_part the top-most instance in this part-of hierarchy.

super_part the super or parent instance which this is a part of.

sub_part a sub or child part of this instance.

a specifically numbered sub part.

For example, the relationship between the various components for the unicycle created above can be described by the following statements.

```
unicycle|198129 is the composite_part of wheel|198130
unicycle|198129 is the composite_part of rim|198132
unicycle|198129 is the composite_part of spoke|198134
unicycle|198129 is the composite_part of spoke|198136
unicycle|198129 is the composite_part of spoke|198138
unicycle|198129 is the composite_part of frame|198140
unicycle|198129 is the composite_part of seat|198142

unicycle|198129 is the super_part of wheel|198130
wheel|198130    is the super_part of rim|198132
wheel|198130    is the super_part of spoke|198134
wheel|198130    is the super_part of spoke|198136
wheel|198130    is the super_part of spoke|198138
unicycle|198129 is the super_part of frame|198140
unicycle|198129 is the super_part of seat|198142

wheel|198130     is a sub_part of unicycle|198129
rim|198132       is a sub_part of wheel|198130
spoke|198134     is a sub_part of wheel|198130
spoke|198136     is a sub_part of wheel|198130
spoke|198138     is a sub_part of wheel|198130
frame|198140     is a sub_part of unicycle|198129
seat|198142      is a sub_part of unicycle|198129

wheel|198130     is wheel#1 of unicycle|198129
rim|198132       is rim#1   of wheel|198130
spoke|198134     is spoke#1 of wheel|198130
spoke|198136     is spoke#2 of wheel|198130
spoke|198138     is spoke#3 of wheel|198130
```

```
frame|198140     is frame#1 of unicycle|198129
seat|198142      is seat#1  of unicycle|198129
```

Attributes

The attributes of a class correspond to the characteristics of the entity which it represents. They may or may not have default values.

Attribute Declarations

Attributes can be split into those which relate to individual instances of the class and those for the overall class itself. Each of these can be declared as private or public attributes.

private instance attributes ...

public instance attributes ...

private class attributes ...

public class attributes ...

A public attribute is one whose value is accessible from outside the class in which it occurs, whereas a private attribute can only be accessed from within. In either case, the attribute's value can only be changed from within the class.

```
public class attribute
     smallest .                  % the smallest instance in
                                 % this class; no default value

private instance attributes
     contents = [] ,             % default contents is empty list
     size is 0 ,                 % the default size is zero
     cumulative_size inherited . % cumulative size is inherited
                                 % from an ancestor class
```

An instance attribute is one which relates to an individual instance of a class. For example, consider a class representing customers of a bank. One attribute is the amount of work necessary to service each individual customer.

A class attribute is one which relates to the overall class itself. For example, an atribute of a class of customer queues may be a pointer to the smallest queue. Each attribute in the declaration list can be assigned an explicit default value (arithmetic or non-arithmetic), an inherited default value or have no default.

Assignment Of Attributes

There are two forms of attribute assignment in Prolog++, one which activates procedures and the other which does not. An assignment which invokes the daemon manager is referred to as a noisy assignment, and one which merely performs the update and does nothing else is referred to as a quiet assignment. They both have the same basic form :

"Name of Attribute" **:=** "New Value"

"Name of Attribute" **:==** "New Value"

where *:=* is an example of a noisy assignment operator and *:==* is an example of a quiet assignment operator.

```
class stack.



contents = [].
size is 0.

push( Element ) :-
      contents := [ Element | contents ],
      size +== 1.
…


end stack.
```

Noisy Assignment

A noisy assignment of a new value to an attribute will invoke the daemon manager in two ways. Before the assignment is performed any constraints will be checked. Immediately after the assignment has been confirmed all reactionary daemons will be invoked[2].

Quiet Assignment

A quiet assignment will not invoke the daemon manager at all, but will merely perform the update.

[2] See the chapter on Prolog++ : Daemons for a full description.

Assignment Operators

There are five different assignment operators :-

Noisy	Quiet	Description
:=	:==	Replace any current value with the incoming value
+=	+=	Add (arithmetic) the incoming value to the current value or default value and assign the result to the attribute
-=	-==	Subtract (arithmetic) the incoming value from the current value or default value and assign the result to the attribute
*=	*==	Multiply (arithmetic) the current value or default value by the incoming value and assign the result to the attribute
/=	/==	Divide (arithmetic) the current value or default value by the incoming value and assign the result to the attribute

Chaining Attributes Together

It is possible that the values of attributes are the names/handles of other objects. That is, attributes can act as pointers to other objects.

```
class employee_record.
public instance attribute address.
when_created :-
  address_record <- create( AddressRecord ),
  address := AddressRecord.
set_city( City ) :-
  @address <- set_city( City ).
get_city( City ) :-
  City = @address@city.
end employee_record.

class address_record.
public instance attribute city='London'.
set_city( City ) :-
  city := City.
end address_record.
```

By chaining together attributes with the @ operator the city where an employee lives can be accessed without directly referring to *address_record*.

```
?-    employee_record <- create(E),
           E <- get_city(C1),
           E <- set_city('Oxford'),
           E <- get_city(C2).

E = ..., C1='London', C2='Oxford'
```

Procedural Methods

A class's methods define how and when messages can be handled, either sent directly to it or passed up the hierarchy by the inheritance mechanism from a descendant class.

Procedural Declarations

The notion of encapsulation in Prolog++ applies not only to data (the attributes of an instance or class) but equally to its algorithms. That is, some of the internal routines used by a class can be hidden from the outside world. This is rather like a module system, which exports a certain collection of visible routines whilst hiding its innermost workings.

Methods can be declared either as public or private, and involve both the name of the method and the number of parameters (its arity).

```
class stack.
public methods
      push / 1,
      pop  / 1.
…
end stack.
```

A public method is one which can handle both local messages and remote messages eminating from outside the class in which it is defined, whereas a private method can only handle local messages originating from within the class itself.

Procedural Definitions

A method is defined in exactly the same way as a Prolog procedure, as a collection of clauses. Each clause takes the general form :-

"Method" **:-** "Prolog++ Formulae" **.**

where the body can optionally be omitted, in which case it takes the form :-

"Method" **.**

```
class stack.

…

push( Element ) :-
      contents := [Element | self::contents],
      size +== 1.

print :-
      write( self = contents ),
      nl.

end stack.
```

A method without any parameters takes the form :-

"Name of Method"

and a method which has N parameters takes the form :-

"Name of Method"**(** "Parameter$_1$" **,** … **,** "Parameter$_N$" **)**

Remote Message Passing

The fundamental communication between objects is by sending messages to one another. This takes the general form :-

"Receiver" **<-** "Message"

where the message receiver is either explicitly named or implicitly referred to by instance/class variables such as *self*, *sub_class*, etc..

```
class bank.

…

simulation :-
      …
      while clock <- still_open
      do    (
            clock <- tick,
             …
            ),
      … .

end bank.
```

Local Message Passing

When sending messages locally (i.e. to the class/instance in which they textually appear), an abbreviated form can be given :-

"Message"

```
class teller.

create_communal( [], _, _ ).

create_communal( [(MeanServiceTime,ServiceVariation)|ServiceTimes],
Number, Queue ) :-
  …,
  create_communal( ServiceTimes, +Number+1, Queue ).

…

end teller.
```

Message Broadcasting

The concept of polymorphism in OOPS means that the same method can be re-defined in several different classes. The idea is that the operation is basically the same (hence the use of the same name), but will be implemented differently according to the desired behaviour of each instance.

The polymorphism of Prolog++ allows the same message to be sent to many different objects. This is done either implicitly through the use of the plural terms like "*all instance teller*" or explicitly by the conjunction operator "**,**".

("Object$_1$" **,** … **,** "Object$_k$" **) <-** "Message"

```
class bank.

…

simulation :-
      …
      while clock <- still_open
      do    (
            clock <- tick,
             (
             customer,                    % Customers arrive when open
             all instance customer_queue, % Update queue statistics
             all instance teller          % Customers served when open
             ) <- after_tick
            ),
      … .

end bank.
```

In addition to sending the same message to different objects, Prolog++ allows multiple messages to be sent to the same class/instance. This broadcasting takes the form :-

"Receiver" **<- (** "Message$_1$" **,** … "Message$_j$" **)**

```
class timetable.

…

setup :-
  …
  form <- (
            create( _, [], first_year  ),
            create( _, [], second_year ),
            create( _, [], third_year  ),
            create( _, [], fourth_year )
```

```
        ),
    …

end timetable.
```

Functional Methods

In addition to defining general procedures, classes can include definitions of evaluable functions which return as values Prolog terms. They are similar to the evaluable functions of languages such as Lisp and Hope.

Functional Declarations

Functional methods, as with procedural methods, can be declared either as public or private.

```
class stack.

public method
    top // 0.

top = Top :-
    size > 0,
    contents = [Top|_].

…

end stack.
```

An evaluable function *F* with arity *A* is referred to by the notation *F//A*.

Functional Definitions

The definition of an evaluable function depends upon whether the value returned is arithmetic or non-arithmetic. The general form of an arithmetic function is :-

"Function" **is** "Arithmetic Expression" **.**

and the general form of a non-arithmetic function is :-

"Function" **=** "Term" **.**

where a parameterized function takes the form :-

"Name of Function"**(** "Parameter$_1$" **,** … **,** "Parameter$_K$" **)**

```
class customer_queue.

…

public method
    smallest_now // 0.
```

```
smallest_now = @smallest_now( all instance class self ).

private methods
      smallest_now // 1,
      smallest_now // 2.

smallest_now( [Queue|Queues] ) = @smallest_now( Queues, Queue ).

smallest_now( [], Smallest ) = Smallest.

smallest_now( [Queue|Queues], SmallestSoFar ) =
  @smallest_now( Queues, @smaller(Queue,SmallestSoFar) ).

public method
      average_size // 0.

average_size is nearest_number( cumulative_size / clock@time_now ).

…

end customer_queue
```

In addition, constraints may be placed upon the values of evaluable functions by attaching conditions. These take the form :-

"Function" **is** "Arithmetic Expression" **:-** "Prolog++ Formulae" **.**

"Function" **=** "Term" **:-** "Prolog++ Formulae" **.**

```
class customer_queue.

…

smaller( Queue1, Queue2 ) = Queue1 :-
  Queue1 @ real_size < Queue2 @ real_size,
  !.

smaller( _, Queue2 ) = Queue2.

…

end customer_queue.
```

Evaluating Functions

A function is evaluated by sending the function call to the relevant class or instance. As with message passing, it can be a remote function call

"Receiver" @ "Function"

or a local function call

```
@ "Function"
```

Arithmetic Functions

In addition to evaluable functions defined within classes, there are certain arithmetic functions inherited from the underlying Prolog. These standard arithmetic functions are either binary or unary, with the normal precedences applying.

For a full description of arithmetic functions see the formal syntax of Prolog++ in the appendices.

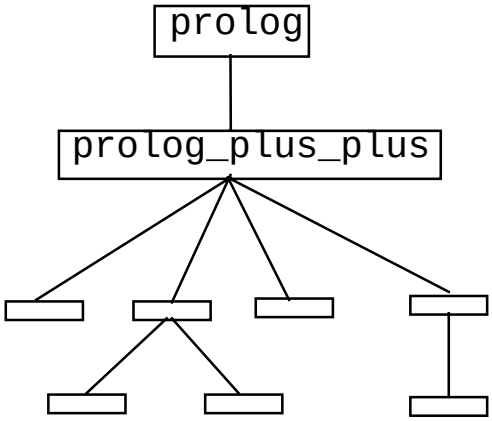
Chapter 10 - The Prolog++ ↔ Prolog Interface

The interface between Prolog and Prolog++ is to a large degree seamless. Calls which are eventually handled by Prolog programs have the same appearance as messages. Messages sent from Prolog to a Prolog++ class or instance have the same appearance as any other procedure call.

The integration is mainly accomplished by the addition of two system-defined classes, one of which makes public the entire set of built-in procedures of the host Prolog engine together with any user-defined predicates.

The Implicit Class Hierarchy

The Prolog++ compiler automatically attaches two system-defined classes to the top-most layer of any class hierarchy. A class at the top-most layer is one which does not have any *inherits* declarations.



As a result of this, all classes will inherit the public methods defined by the *prolog_plus_plus* system class together with all of the built-ins and predicates which are made public by the *prolog* class.

Calling Prolog From Prolog++

One of the benefits of having *prolog* as the top-most class in any hierarchy is that calling a Prolog built-in or user-defined predicate is exactly the same as sending a message to a class or instance.

```
class clock.

...

print :-
  write( 'Closing time                              : ' ),
  write( @time_to_stop ),
  nl,
```

```
  write( 'Last customer served at                    : ' ),
  write( @time_now ),
  nl,
  nl.

…

end clock.
```

In the print method above each of the *write/1* and *nl/0* messages will be handled by Prolog built-in procedures, unless some other class higher up the hierarchy chooses to re-define them as Prolog++ methods.

Alternatively, the messages can be sent explicitly to the *prolog* class, as in the alternative definition below.

```
class clock.

…

print :-
  Stop = @time_to_stop,
  Now  = @time_now,
  prolog <-
    (
      write( 'Closing time                               : ' ),
      write( Stop ),
      nl,
      write( 'Last customer served at                    : ' ),
      write( Now ),
      nl,
      nl
    ).

…

end clock.
```

Calling Prolog++ From Prolog

In addition to the Prolog++ compiler which translates class definitions into collections of Prolog programs, there is a run-time predicate *->/2* for sending messages from Prolog to any Prolog++ class or instance.

```
?-    employee_record <- create(E),
            E <- get_city(C1),
            E <- set_city('Oxford'),
            E <- get_city(C2).

E = ..., C1='London', C2='Oxford'
```

In the example above the predicate *<-/2* is called four times, firstly with the *employee_record* class name as a parameter and then with a system generated instance as a parameter.

Chapter 11 - The "prolog_plus_plus" Class

This chapter details all of the methods made public by the *prolog_plus_plus* class. Since this class is implicitly placed at the top of a class hierarchy these public methods are inherited by all classes.

The public methods of the *prolog_plus_plus* class are given below. Each method indicates whether the message should be sent to a class or an instance of a class. In some cases, for example *compile/2*, it is irrelevant where the message is sent. In other cases, for example *isa_class/0* and *isa_instance/0*, it can be sent to either kind of object. Finally, certain messages such as *create/1* can only be sent to a specific kind of object as indicated in the Self column.

Method	Self	Description
compile/2		Equivalent to the Prolog++ class compiler allows new classes to be incorporated at run-time.
create/1	class	Create a new instance of the class.
create/2	class	Create a new instance of the class with some initial attribute values.
create/3	class	Create a new instance of the class with some initial attribute values and give it a reference name.
delete/0	instance	Delete this instance.
delete_all/0	class	Delete all instances of this class.
dump/1	class	Dump all instances of this class to a file.
dump/2		Dump all instances of the named classes to a file.
duplicate/1	instance	Duplicate instance by creating a copy with the same attribute values.
isa_class/0	class/instance	Test whether the object is a class.
isa_instance/0	class/instance	Test whether the object is an instance of a class.
kill/0	class	Remove the definition of a class.
load/1		Load the previously saved class definitions from a file.
optimize/0	class	Optimize a class's implementation.
public/1	class	Test or retrieve the name and arity of a public method for this class.
reset/0	class/instance	Reset attributes of this class/instance back to their default values.
reset/1	class/instance	Reset named attribute of this class/instance back to its default value.
restore/1		Restore the previously saved class instances from a file.
save/1	class	Save this class's defintion to a file.
save/2		Save the definitions of the named classes to a file.

when_error / 2	class/instance	A catchall method for handling errors which were raised by the system but not handled by the erroneous class.

compile/2

the run-time compiler for incorporating new classes

Syntax compile(Class, Sentences)

+Class	<atom>
+Sentences	<list of class declarations and definitions>

Comments The *compile/2* predicate invokes the Prolog++ compiler so that new classes can be incorporated at run-time. It does not matter which class the message is sent, although for clarity it makes sense to send the message to the "prolog_plus_plus" class itself.

Exceptions The errors raised by this method are those detailed in appendix E for the Prolog++ compiler.

Example The following example creates a new "stack" class with the given categories, inheritance and public methods.

```
?- prolog_plus_plus <- compile(
      stack,
      [
      ( categories
            library ,
            buffer  ),
      ( inherit
            buffer ),         %A stack is a kind of buffer.
      ( public methods
            nil  // 0 ,      %A nil stack
            push // 2 ,      %Push something onto the stack
            pop  // 2 ,      %Pop something off the stack
            top  // 1 ,      %Return the top of the stack
            list // 1),      %Return contents of the stack
      ( nil = [] ),
      ( push( Item, Sta      ck ) = [Item|Stack] ),
      ( pop( Item, [Item|Stack] ) = Stack ),
      ( top( [Item|_] ) = Item ),
      ( list( Stack ) = Stack )
      ] ).
```

create/1

create a new instance of a class

Syntax create(Instance)

-Instance	<variable>

Comments Create a new instance of the class which received the message, returning the unique handle for that instance.

Exceptions

- The message is sent to an existing instance rather than a class.
- *Instance* is already instantiated when the message is sent.

Example The following example creates a new instance of the class "unicycle" and returns its handle.

```
?- unicycle <- create(C).
Created: wheel|198130
Created: rim|198132
Created: spoke|198134
Created: spoke|198136
Created: spoke|198138
Created: frame|198140
Created: seat|198142
C = (unicycle|198129)
```

---

create/2

create a new instance of a class with a collection of attribute values

Syntax create(Instance, AttributeValues)

-Instance	<variable>
+AttributeValues	<(attribute is value) or (attribute = value) list>

Comments Create a new instance of the class which received the message, returning the unique handle for that instance. In addition, assign the designated attributes to their corresponding arithmetic (is) or non-arithmetic (=) values.

Exceptions

- The message is sent to an existing instance rather than a class.
- *Instance* is already instantiated when the message is sent.
- The attribute-value pairs are not of the expected form.

Example The following example creates an instance of the "stack" class with the given "contents" and "size" attribute values and returns its handle.

```
?- stack <- create(S,[contents=[a],size is 1]).
S = (stack|201322)
```

---

create/3

create a new instance of a class with a collection of attribute values and assign to it a mnemonic handle by which it can be referred

Syntax create(Instance, AttributeValues, Mnemonic)

-Instance	<variable>
+AttributeValues	<(attribute is value) or (attribute = value) list>
+Mnemonic	<non-variable>

Comments Create a new instance of the class which received the message, returning the unique handle for that instance. In addition, assign the designated attributes to their corresponding arithmetic (is) or non-arithmetic (=) values and record the

association between the instance's handle and its mnemonic name for future reference.

Exceptions

- The message is sent to an existing instance rather than a class.
- *Instance* is already instantiated when the message is sent.
- The attribute-value pairs are not of the expected form.
- *Mnemonic* is not instantiated when the message is sent.
- *Mnemonic* unifies with the mnemonic name of an existing instance.

Example The following example creates an instance of the "stack" class with the given "contents" and "size" attribute values. It also sets a mnemonic "stack1" for the instance and returns its handle.

```
?- stack <- create(S,[contents=[a],size is 1],stack1).
S = (stack|201322)
```

---

delete/0

delete this instance

Comments Delete the class instance to which the message was sent. In addition, if this is the composite part of a complex object then all of its component parts will also be deleted.

Exceptions

- The message is sent to a class rather than an instance of a class.
- The instance to be deleted is a component of some complex object.

Example The following example creates and then deletes an instance of the "unicycle" class.

```
?- unicycle <- create(C), C <- delete.
C = (unicycle|243187)
```

---

delete_all/0

delete all class instances

Comments Delete all instances of the class to which the message was sent. In addition, if any of those are the composite part of a complex object then all of their corresponding component parts will also be deleted.

Exceptions

- The message is sent to an instance rather than a class.
- One or more of the class instances is a component of some other complex object.

Example The following example creates two instances for the "unicycle" class and then deletes them.

```
?- unicycle <- create(C1),
   unicycle <- create(C2),
   unicycle <- delete_all.
C1 = (unicycle|249426)
C2 = (unicycle|249451)
```

---

dump/1

dump all class instances into a file

Syntax dump(File)

+*File*	<prolog_file_specification>

Comments Dump all the instances of the class to which the message is sent. This involves storing their attribute values and part-of structures in the specified file.

Exceptions

- The message is sent to an instance rather than a class.
- *File* is not a Prolog file specification.
- One or more attributes of the instances reference some other instance which is not being dumped to the same file. That is, no dangling references are allowed.
- One or more of the instances is a component in a complex object which by definition is itself not being dumped to the same file. Again, no dangling references are allowed.

Example The following example dumps all the instances of the "unicycle" class into the file "UNICYCLE.PC"

```
?- unicycle <- dump( 'unicycle' ).
```

---

dump/2

dump all class instances of the specified classes into a file

Syntax dump(File, Classes)

+*File*	<prolog_file_specification>
+*Classes*	<list of atom>

Comments Dump all the instances of the specified classes into the file. This involves storing their attribute values and part-of structures.

Exceptions

- *File* is not a Prolog file specification.
- *Classes* is not a list of atoms.

- One or more attributes of the instances reference some other instance which is not being dumped to the same file. That is, no dangling references are allowed.

- One or more of the instances is a component in a complex object which is itself not being dumped to the same file. Again, no dangling references are allowed.

Example The following example dumps all the instances of the class "cycle" to the file CYCLES.PC.

```
?- prolog_plus_plus <- dump('Cycles',(category cycle)).
```

---

duplicate/1

duplicate this instance

Syntax duplicate(Instance)

-Instance <variable>

Comments Duplicate the instance to which the message is sent. This involves creating a new instance of the same class and then copying all of the original's attribute values. In addition, if the original instance has any component parts they too are duplicated and the part-of structure itself is copied.

Exceptions

- The message is sent to a class rather than an instance of a class.

- *Instance* is already instantiated when the message is sent.

Example The following example creates an instance of the "stack" class, then creates a duplicate of that instance and finally tests the "contents" attribute of the duplicate instance to see if it is identical to the original instance.

```
?- stack <- create(S1,[contents=[a],size is 1]),
   S1 <- duplicate(S2),
   S2 <- @contents = C.

S1 = (stack|263188)
S2 = (stack|263201)
C  = [a]
```

---

isa_class/0

has this message been sent to a class?

Comments The message is successful when sent to a class but not when sent to an instance of a class.

Example The following example tests whether the class "stack" is indeed a class.

```
?- stack <- isa_class.
yes
```

The next example creates an instance of the "stack" class and then confirms that this is not a class.

```
?- stack <- create(S), S <- isa_class.
no
```

---

isa_instance/0

has this message been sent to an instance of a class?

Comments The message is successful when sent to an instance of a class but not when sent to a class itself.

Example The following example confirms that the class "stack" is not an instance.

```
?- stack <- isa_instance.
no
```

The next example creates an instance of the "stack" class and then confirms that this is indeed an instance.

```
?- stack <- create(S), S <- isa_instance.
yes
```

---

kill/0

remove the class definition and all instances

Comments Remove the class to which the message is sent. Furthermore, remove all instances of that class together with their component parts.

Exception

- The message is sent to an instance rather than a class.

Example The following example removes the "unicycle" class and all its instances.

```
?- unicycle <- kill.
```

---

load/1

load the class definitions from file

Syntax load(File)

+File	<prolog_file_specification>

Comments Load the class definitions which had previously been saved to the specified file.

Exception

- *File* is not a Prolog file specification.

Example The following example loads all the class definitions in the file "CYCLES.PC".

```
?- prolog_plus_plus <- load( 'CYCLES' ).
```

optimize/0

optimize the class

Comments Optimize the previously compiled class to which the message is sent.

Exception
- The message is sent to an instance rather than a class.

Example The following example optimizes the definition of the "stack" class.

```
?- stack <- optimize.
```

public/1

the method is public

Syntax public(Method)

?Method	<atom> / <integer>

Comments *Method* is instantiated or unified with the structure *MethodName* / *MethodArity* where these are publicly declared methods of the class to which the message is sent.

Exception
- The message is sent to an instance rather than a class.

Example The following example returns the public methods of the "stack" class.

```
?- stack <- public( M ).

M = push / 1 ;

M = pop / 1 ;

M = top / 1 ;

M = empty / 0
```

reset/0

reset attribute values

Comments Reset the attributes back to their default values. If the message is sent to a class then the class attributes are reset, whereas if the message is sent to an instance then the instance attributes are reset.

Example The following example resets the class attributes of the "customer_queue".

```
?- customer_queue <- reset.
```

The next example resets the instance attributes of all the instances of the class "customer_queue".

```
?- all instances customer_queue <- reset.
```

reset/1

reset an attribute value

Syntax reset(Attribute)

+Attribute	<atom>

Comments Reset the specified attribute back to its default value.

Example The following example

```
?- customer_queue <- reset( smallest_now ).

?- all instances customer_queue <- reset( my_neighbour ).
```

restore/1

load the instances from file

Syntax restore(File)

+File	<prolog_file_specification>

Comments Load the instances which had previously been dumped to the specified file. In addition, restore all of their attribute values and part-of hierarchies.

Exception

- *File* is not a Prolog file specification.

Example The following example loads the instances from the CYCLES.PC file.

```
?- prolog_plus_plus <- restore( 'Cycles' ).
```

save/1

save the class definition to a file

Syntax save(File)

+File	<prolog_file_specification>

Comments Save in the specified file the class to which the message is sent.

Exceptions

- The message is sent to an instance rather than a class.
- *File* is not a Prolog file specification.

Example The following example saves the class definition of the *unicycle* class in the file UNICYCLE.PC

```
?- unicycle <- save( 'UNICYCLE' ).
```

save/2

save all classes into the file

Syntax save(File, Classes)

+*File*	<prolog_file_specification>
+*Classes*	<list of atom>

Comments Save the definitions of all the specified classes into the file.

Exceptions

- *File* is not a Prolog file specification.
- *Classes* is not a list of atoms.

Example The following example saves all the currently defined classes in the cycle category in the file CYCLES.PC.

```
?- prolog_plus_plus <- save(
   'CYCLES',
   (category cycle)
   ).
```

when_error/2

react to an exception in the standard Prolog++ manner

Syntax when_error(Exception, Message)

+*Exception*	<any term>
+*Message*	<prolog++ message>

Comments React to the specified *Exception*, which was raised by sending the *Message* to the recipient object, in the standard Prolog++ manner.

Example The following example reacts to the instantiation error when sending the *create/1* message to the *unicycle* class.

```
?- unicycle <- when_error(
   instantiation_error,
   create(fred)
   ).
```

which is equivalent to

```
?- unicycle <- create(fred).
```

providing that the methods *when_error/2* and *create/1* are not made public by the class *unicycle* or any of its ancestors. That is, providing both messages are eventually handled by the *prolog_plus_plus* class.

Part 4 - Operational Features Of Prolog++

Chapter 12 - Reacting To Class Instances

Whenever a new instance of a class is created it is likely that it will need to be specialized in some way with its own set of characteristics.

For example, suppose we have implemented an object-oriented database as part of a company's payroll system. As new employees join the company so new instances of an *employee* class need to be created. The details of the employee, for example name, address, NI number, etc., must be collected and added as attributes of the new instance. To cater for this situation, a *when_created/0* method can be defined which reacts to the creation of new instances.

Similarly, an employee leaving the company may require certain modifications to the database other than the simple deletion of their associated instance. This tidying up process can be implemented by a *when_deleted/0* method.

When A New Instance Is Created

The *when_created/0* method reacts whenever a new instance of a class is created. The message is sent by Prolog++ immediately after it has created the new instance.

For example, a method which reacts to new employees joining the company might automatically assign that employee the next payroll number in sequence.

```
class employee.

public instance attribute payroll_number.
when_created :-
      payroll_number := @next_payroll_number.

private class attribute last_payroll_number is 100000.
private method next_payroll_number//0.
next_payroll_number is @last_payroll_number :-
      last_payroll_number += 1.

end employee.
```

The creation of an instance is implemented as if by the *prolog_plus_plus* method.

```
create( Instance ) :-
      prolog <- create_class_instance( self, Instance ),
      while Instance <- when_created
      do    true.
```

When An Existing Instance Is Deleted

The *when_deleted/0* method reacts whenever an existing instance of a class is deleted. The message is sent by Prolog++ immediately before it is removed.

For example, a method which reacts to old employees leaving the company might automatically issue the appropriate social security notice.

```
class employee.

when_deleted :-
      self <- issue_p45_form.

end employee.
```

The deletion of an instance is implemented as if by the *prolog_plus_plus* method.

```
delete :-
      while self <- when_deleted
      do    true,
      prolog <- delete_class_instance( self ).
```

Chapter 13 - Reacting To Attribute Values

Most procedural languages such as C, Fortran and Pascal, and most OOPS such as C++ and Smalltalk, are typed languages. That is, everything is known at compile-time about the possible values which variables, functions, attributes etc. can take. However, since Prolog++ is fundamentally based upon the typeless language Prolog it has no built-in type checking This is both a positive and a negative feature of the language. It is positive in the sense that not having to make type declarations has been shown to improve the productivity of Prolog (and thus Prolog++) programmers. It is negative in the sense that type declarations allow the compiler to make certain assumptions about the program, and so improve its run-time efficiency. More important, though, type declarations pick up programming errors at compile time and also invalid data at run-time.

In Prolog++ methods can be defined which determine the (in)validity of attribute values. Moreover, such methods need not be restricted to simple type checking along the lines of traditional languages, but may involve comparisons between the incoming data and any existing data. This general constraint mechanism is far more powerful than any static type checking system.

In addition to checking the value of an attribute before it is assigned, methods can be defined which react to assignments after they have been made. For example, the classic rule of stock control management which states that a re-order must be made whenever the stock level of an item falls below some critical level can easily be implemented by one of these methods.

```
public instance attributes
      stock_level is 0,
      re_order_level.

when_assigned( stock_level, _ ) :-
      Level < @re_order_level,
      self <- issue_re_order.
```

Invalid Attribute Values

The general form for this method is

invalid(AttributeName, NewValue)

and is invoked immediately before the *NewValue* is assigned to the *AttributeName*. The message is sent by Prolog++ to the object (class or instance) whose attribute is about to be updated.

For example, all employee NI numbers must follow a fixed format.

```
invalid( ni_number, NI ) :-
      \+ ni_number_format( NI ).
```

The update of an attribute value is partially implemented as if by the *prolog_plus_plus* method.

```
AttributeName := NewValue :-
      self <- invalid( AttributeName, NewValue ),
      !,
      self <- when_error( invalid, (AttributeName := NewValue) ).

AttributeName := NewValue :-
      prolog <- update_attribute_value(self,AttributeName,NewValue).
```

Reacting To New Attribute Values

The general form for this method is

when_assigned(AttributeName, OldValue)

and is invoked immediately after the *OldValue* for the *AttributeName* has been replaced. Thus, the new value is available by inspection of the attribute.

For example, a simple log of attribute updates can be encoded by the following method.

```
when_assigned( AttributeName, OldValue ) :-
      NewValue = @AttributeName,
      write( self ),
      write( ' @ ' ),
      write( AttributeName ),
      write( ' updated from ' ),
      write( OldValue ),
      write( ' to ' ),
      write( NewValue ),
      nl.
```

The update of an attribute value is implemented as if by the *prolog_plus_plus* method.

```
AttributeName := NewValue :-
      self <- invalid( AttributeName, NewValue ),
      !,
      self <- when_error( invalid, (AttributeName := NewValue) ).

AttributeName := NewValue :-
      OldValue = @AttributeName,
      prolog <- update_attribute_value(self,AttributeName,NewValue),
      while self <- when_assigned( AttributeName, OldValue )
      do    true.
```

Chapter 14 - Exception Handling

Many programming errors and invalid data items can be trapped by Prolog++.

For example, sending an unrecognised message to an object (class or instance) will result in that message eventually reaching the *prolog_plus_plus* class. If at this point the message is still not recognised and furthermore it cannot be handled by a Prolog predicate, then a *no_message_handler* exception will be raised.

Exceptions are raised in Prolog++ by sending an exception message back to the object which received the original message.

For example, suppose the instance *(teller|123456)* is sent the unrecognised message *you_cannot_handle_me*. The system will respond by sending the message

```
when_error(
        no_message_handler,
        you_cannot_handle_me
      )
```

back to instance *(teller|123456)*. If this exception message is not handled by the class hierarchy it will eventually find its way back to the *prolog_plus_plus* class for handling in the standard Prolog++ way.

```
class complex.
      public instance attribute valp.
      setp(Rho,Theta) :-
            valp &= polar(Rho,Theta).
      getp(Rho,Theta) :-
            @valp = polar(Rho,Theta).
      invalid(Attribute,Value) :-
            \+ valid(Attribute,Value).
      valid(valp,polar(Rho,Theta)) :-
            number(Rho),
            number(Theta).
      when_error( no_message_handler, _ ) :-
            !,
            fail.
      when_error( Exception, Message ) :-
            super <- when_error( Exception, Message ).
end complex.

class sub_complex.
      inherit complex.
      when_error( invalid, _ ) :-
            !.
      when_error( Exception, Message ) :-
```

```
            super <- when_error( Exception, Message ).
end sub_complex.
```

Note the manner in which *when_error/2* messages are passed up the hierarchy by re-sending those not handled locally up to *super*.

The classes *sub_complex* and *complex* only differ in the way *invalid* exceptions are handled.

```
?- complex <- create(C),
   C <- setp(1,two),
   C <- getp(R,T).
* Prolog++ Run-Time Error
* (complex|437089)<-valp:=polar(1, two)

?- sub_complex <- create(C),
   C <- setp(1,two),
   C <- getp(R,T).
C = (sub_complex|437664),
R = 1,
T = two

?- complex <- can_you_handle_me.
no

?- sub_complex <- can_you_handle_me.
no

?- complex <- create(me).
* Prolog++ Run-Time Error
* complex<-create(me)

?- sub_complex <- create(me).
* Prolog++ Run-Time Error
* sub_complex<-create(me)
```

Part 5

Appendices

Appendix A - Formal Syntax Of Prolog++

This appendix details the formal syntax of Prolog++ programs. It is the definitive specification of all constructs which can appear within a Prolog++ instance.

How To Read The Grammar Rules

The formal syntax of Prolog++ is defined in the following sections by grammar rules having the general form :-

non_terminal
→ $phrase_{1,1}$ $phrase_{1,2}$ ... $phrase_{1,k_1}$
or $phrase_{2,1}$ $phrase_{2,2}$ ... $phrase_{2,k_2}$
or ...
or $phrase_{n,1}$ $phrase_{n,2}$ ... $phrase_{n,k_n}$

where $n \geq 1$ represents the number of different definitions for the *non_terminal* and $k_m \geq 1$ (for every $1 \leq m \leq n$) represents the number of phrases in each definition. Each *phrase*i,j is either itself a grammatical construct or a terminal symbol. To assist the reading of these rules all terminal symbols in the text are in **bold** type.

A few basic grammar rules are presented below which will be used in the more complex grammar rules of the following sections.

The name of something is a Prolog atom (other than *[]*).

name.<type>
→ *prolog_atom*

Meta phrases of a particular type are given by Prolog variables, but they must be instantiated to something of that type at run-time.

meta.<type>
→ *prolog_variable*

Terms of a particular type can be specified by any legal Prolog++ term, but they must evaluate to something of that type at run-time.

term.<type>
→ *term*

An arity is a Prolog integer greater than or equal to zero which represents the number of arguments for the associated method.

arity
→ *non_negative_prolog_integer*

A parameter is any Prolog term which appears in the head (left-hand side) of a definition.

parameter
→ *prolog_term*

Sequences are grammatical constructs separated by commas.

sequence.<type>
→ *<type>* **,** *sequence.<type>*
or *<type>*

Class

A class definition is a sequence of sentences delimited by begin and end statements. Both the begin and end statements must name the class being defined.

class_definition
→ *begin_class sentences end_class*

begin_class
→ ***class*** *name.class* **.**

end_class
→ ***end*** *name.class* **.**

Each individual sentence represents either a declaration or a definition, and is always terminated with a period (.).

sentences
→ *sentence sentences*
or *sentence*

sentence
→ *declaration* **.**
or *definition* **.**

Declarations

A declaration states how classes are related to eachother (through categories, inheritance links or component links), the attributes of the class and the methods used to handle messages.

declaration
→ *category_declaration*
or *inherits_declaration*
or *parts_declaration*
or *attributes_declaration*
or *methods_declaration*

Categories Declaration

Categories are used to cluster classes so that communal operations (such as kill, save, etc.) can be applied without prior knowledge of which classes form the category. A class which does not declare any categories is assumed to be in the **user** category.

For example, when Prolog++ clears its workspace all classes in the **user** category are killed but none of the system or library classes are killed.

category_declaration
→ **categories** *sequence.name.category*

Inherits Declaration

Classes can be is-a related to form inheritance hierarchies. Attributes and methods which are not defined locally are inherited according to such declarations.

inherits_declaration
→ **inherits** *sequence.name.super_class*

Super classes are generalisations of more specific classes.

When more than one super-class is declared the hierarchy in which it exists is said to exhibit multiple inheritance. In such cases the ordering of super-classes in the declaration is very important as this determines the order of search for inherited attributes and methods.

Parts Declaration

Classes can be part-of related to form component hierarchies.

parts_declaration
→ **parts** *sequence.part_specification*

part_specification
→ *name.class* * *part_enumeration*
or *name.class*

part_enumeration
→ *term*

A part-of hierarchy determines how instances of that class are created. If no parts exist then only an instance of the class is created. If parts do exist then instances of the part classes are created in addition to an instance of the class itself. Similarly, when instances of a class are deleted all of its component parts are also deleted.

For example, whenever a specific bicycle is created, a pair of handle-bars, a frame, a seat and two wheels (each consisting of a rim and 12 spokes) might also created with the appropriate links established between those component instances and the bicycle instance.

Parts can be inherited from super-classes. Note that only the part names which are not declared locally will be inherited.

For example, a tandem might have two pairs of handle-bars and two seats in addition to the frame and two wheels inherited from a normal bicycle.

Attributes Declaration

Attribute declarations determine the dynamic characteristics of a class and its instances. They are sometimes referred to as class/instance/object variables in other OO languages. The terminology chosen here is simply to avoid confusion with the notion of a Prolog variable.

The attributes of a class are not only those explicitly declared but also those of any of its super-classes, super-super-classes, etc.. That is, the total set of attributes for a class is the union of its locally declared attributes and those of all its ancestor classes.

There are two distinct types of attributes, those assigned to instances of the class and those assigned to the class itself. For example, an individual instance of a queue may have an attribute called *size* and the queue class itself may have an atrribute called *smallest* which points to that particular instance which has the smallest *size* value.

Class attributes are prefixed with the keyword **class** and instance attributes are prefixed with the keyword **instance**.

Attributes can be declared **public** or **private**. Public attributes can be accesssed from outside the class in which they are textually declared, whereas private attributes can only be accesssed from within. In both cases, however, the assignment of new values must be done locally.

attributes_declaration
→ *attribute_mode*
attribute_type
attributes
sequence.attribute_default

attribute_mode
→ **private**
or **public**

attribute_type
→ **instance**
or **class**

An attribute can be given a default value when it is declared. If it is not then any attempted accesses will fail until the attribute is assigned a current value.

The default value can be an arithmetic expression (using **is**) which is automatically evaluated by the Prolog built-in *is/2*, a non-arithmetic term (using **=**) or its default can be **inherited** from some ancestor class in which that same attribute is declared.

attribute_default
→ *name.attribute* **is** *term*
or *name.attribute* **=** *term*
or *name.attribute* **inherited**
or *name.attribute*

Note that a default value of the form

name.attribute **is** *term*

is equivalent to

name.attribute **= +(** *term* **)**

Methods Declaration

A methods declaration states what kind of messages can be handled by a class. Any other message will be passed up the is-a hierarchy to one of its ancestor classes which is capable of handling it.

Methods can be declared **public** or **private**. Public methods can handle messages originating from outside the class in which they are textually declared as well as from inside, whereas private methods can only handle messages originating from within.

methods_declaration
→ *method_mode* **methods**
sequence.method_arity

method_mode
→ **public**
or **private**

Method declarations are classified as procedural (using **/**) or as functional (using **//**).

method_arity
→ *name.method* **/** *arity* {procedural}
or *name.method* **//** *arity* {functional}

Definitions

Definitions can be given for functional or procedural methods. In either case the definiens can be qualified by a series of statements or not, thus taking the same general form as Prolog clauses.

defininition
→ *definiens* **:-** *statements*
or *definiens*

definiens
→ *functional_definiens*
or *procedural_definiens*

Functional Method Definition

A functional definition specifies the head of a method for handling functional messages. The value of a function can either be arithmetic (using **is**) which is evaluated by the Prolog built-in *is/2*, or non-arithmetic (using **=**).

functional_definiens
→ *functional_head* **is** *term*
or *functional_head* **=** *term*

Note that a functional definition of the form

functional_head **is** *term*

is equivalent to

functional_head **= +(** *term* **)**

The head of a functional method is similar to the head of a Prolog clause.

functional_head
→ *name.method***(** *sequence.parameter* **)** {arity > 0}
or *name.method* {arity = 0}

Procedural Method Definition

A procedural definition specifies a method for handling procedural messages.

Certain special procedures, namely error handlers, constraint handlers, attribute assignments and the creation or deletion of class instances are recognised as having special significance in Prolog++.

procedural_definiens
→ *message_handler_definiens*
or *error_handler_definiens*
or *constraint_handler_definiens*
or *assignment_handler_definiens*
or *create_handler_definiens*
or *delete_handler_definiens*

In all cases, the definiens of a procedural method is like the head of a Prolog clause. The parameters are treated in exactly the same manner as the parameters in the head of a Prolog clause, unifying with the arguments passed when a message is sent.

Message Handler Definiens

A general message handler either has many parameters (arity>0) or no parameters (arity=0).

message_handler_definiens
→ *name.method* **(** *sequence.parameter* **)** {arity > 0}
or *name.method* {arity = 0}

Error Handler Definiens

An error handler is defined in exactly the same manner as general procedural methods. They can be handled locally, inherited or even passed progressively up the class hierarchy through the *super* variable.

Error handlers are given special significance because the underlying Prolog++ engine may raise errors which the programmer then has the capability of trapping.

error_handler_definiens
→ **when_error(** *parameter.term.error* **,** *parameter.message* **)**

The error (1st parameter) is a Prolog term indicating the type of error, such as instantiation faults, domain violations, etc.. A *when_error/2* message is sent to the same *self* variable to which the original erroneous message (2nd parameter) was sent.

The error controller is implemented as if it were a Prolog++ method of the form :-

```
error_controller( Exception, Message ) :-
      self <- when_error( Exception, Message ).

when_error( Exception, Message ) :-
      prolog <- run_time_error_dialog( Exception, Message, self ).
```

Constraint Handler Definiens

A constraint governs the assignment of attribute values. The assignment is not permitted if any of the invalidity constraints governing that particular attribute hold, and is permitted otherwise. In addition, whenever an invalidity constraint is violated an appropriate error is raised.

constraint_handler_definiens
→ **Invalid(** *parameter.name.attribute* **,** *parameter.value* **)**

The constraint checker is implemented as if it were a Prolog++ method of the form :-

```
constraint_checker( Attribute, NewValue ) :-
      self <- invalid( Attribute, NewValue ),
      !,
      self <- when_error( invalid, (Attribute := NewValue) ).

constraint_checker( _, _ ).
```

Assignment Handler Definiens

An assignment handler reacts after the assignment of an attribute.

assignment_handler_definiens
→ ***when_assigned*(**
parameter.name.attribute **,**
parameter.oldvalue
)

Whenever an attribute is assigned a new value all of the handlers governing that attribute are invoked. The assignment controller is implemented as if it were a Prolog++ method of the form :-

```
assignment_controller( Attribute, OldValue ) :-
      self <- when_assigned( Attribute, OldValue ),
      fail.

assignment_controller( _, _ ).
```

Create Handler Definiens

A create procedure reacts to the creation of a new class instance. It is invoked immediately after the actual instance is created.

create_handler_definiens
→ ***when_created***

Whenever an instance is created all of the handlers are invoked. The creation controller is implemented as if it were a Prolog++ method of the form :-

```
creation_controller :-
      self <- when_created,
      fail.

creation_controller.
```

Delete Handler Definiens

A delete procedure reacts to the deletion of an existing class instance. It is invoked immediately before the actual instance is deleted.

delete_handler_definiens
→ ***when_deleted***

Whenever an instance is deleted all of the handlers are invoked. The deletion controller is implemented as if it were a Prolog++ method of the form :-

```
deletion_controller :-
      self <- when_deleted,
      fail.

deletion_controller.
```

Prolog++ Statements

Prolog++ statements are made up of conjunctions, disjunctions, negations, universals, existentials, sets, etc..

statements
→ *meta.statements*
or *statements* **,** *statements*
or *statements* **;** *statements*
or *statements* **->** *statements*
or **forall** *statement* ***do*** *statement*
or **while** *statement* ***do*** *statement*
or **repeat** *statement* **until** *statement*
or **\+** *statement*
or ***not*** *statement*
or *solution_collector***(**
prolog_term **,**
statement **,**
prolog_variable
)
or *prolog_term* **^** *statement*
or *statement*

solution_collector
→ **setof**
or **bagof**
or **findall**

An individual statement is either a simple Prolog control structure, an assignment of some value to an attribute or the sending of a message. An individual statement can also be some compound statements delimited by round brackets.

statement
→ *control*
or *attribute_assignment*
or *send_message*
or **(** *statements* **)**

Control

Simple control statements such as *!*, *true*, *fail* and *repeat* are treated exactly the same as they are in Prolog clauses.

control
→ **!**
or **true**
or **fail**
or **repeat**

Attribute Assignment

The left-hand side of an assignment statement refers to the name of the attribute which is being assigned and the right-hand side is the value which it is given.

The class (if it is a class attribute) or the instance (if it is an instance attribute) which is assigned the new attribute value is determined by the *self* variable.

If the value assigned is an arithmetic expression it will be evaluated using the Prolog built-in predicate *is/2*.

attribute_assignment
→ *name.attribute assign_operator term*

The type of assignment operator determines whether or not constraint checking and assignment handlers will be invoked. These are referred to, respectively, as noisy assignments and quiet assignments.

assign_operator
→ *noisy_assign_operator*
or *quiet_assign_operator*

Noisy Assignment Operators

A noisy assignment will invoke any associated constraint checking and/or assignment handlers. The first assignment operator applies to all symbolic values whereas the others only apply to arithmetic values.

noisy_assign_operator

→	**&=**	{Replace attribute with symbolic value}
or	**:=**	{Replace attribute with arithmetic value}
or	**+=**	{Add to arithmetic attribute value}
or	**-=**	{Subtract from arithmetic attribute value}
or	***=**	{Multiply arithmetic attribute value by …}
or	**/=**	{Divide arithmetic attribute value by …}

Quiet Assignment Operators

A quiet assignment in contrast to a noisy assignment will not invoke any associated constraint checking or any associated assignment handlers. The first assignment operator applies to all symbolic values whereas the others only apply to arithmetic values.

quiet_assign_operator

→	**&==**	{Replace attribute with symbolic value}
or	**:==**	{Replace attribute with arithmetic value}
or	**+==**	{Add to arithmetic attribute value}
or	**-==**	{Subtract from arithmetic attribute value}
or	***==**	{Multiply arithmetic attribute value by …}
or	**/==**	{Divide arithmetic attribute value by …}

Procedural Message Passing

Procedural messages can either be sent explicitly to a remote class/instance or sent implicitly to the local *self* variable.

send_message
→ *send_remote_message*
or *send_local_message*

send_remote_message
→ *sequence.receiver* **<-** *sequence.procedural_message*

send_local_message
→ *procedural_message*

When remote receivers are combined as a sequence the corresponding messages are broadcast to all of them.

When messages are combined as a sequence they are each sent to the receiver(s) in succession.

There is a subtle yet very important difference between local message passing (implictly sent to *self*) and remote messages explicitly sent to the *self* variable and is to do with early versus late binding.

Early binding automatically occurs for local messages whenever there is a corresponding local handler. An early binding is forged at compile-time which greatly improves the efficiency of the resulting code by avoiding the search for a handler at run-time.

Late bindings between a message and a class capable of handling it are forged at run-time whenever no local link can be found and for all messages sent remotely. It takes the form of a search up through the is-a hierarchy of classes starting at the class of the message receiver.

If you wish to avoid an early binding (and thus fool the compiler) send the same message explicitly to the *self* variable. When this is stated in the code a late binding will be performed at run-time.

For example, consider the definitions of *print_early/0* and *print_late/0* below which only differ in their manner of calling *print_contents/0*.

```
class alpha .

      print_early :-
            print_header,
            print_contents,
            print_footer.

      print_late :-
            print_header,
            self <- print_contents,
            print_footer.

      print_contents :-
            …

end alpha


class beta .

      inherits alpha .

      print_contents :-
            …

end beta
```

A call to *print_early/0* for an instance of *beta* will use the definition of *print_contents/0* found in *alpha*, whereas a call to *print_late/0* for a similar instance will use the definition of *print_contents/0* found in *beta* itself.

If the definition of *print_contents/0* were removed from class *alpha* then the two calls would be equivalent since the compiler would not be capable of an early binding!

Procedural Message

A procedural message is handled by a method either defined in or inherited by the receiving class/instance.

procedural_message
→ *meta.procedural_message*
or *name.method*(*sequence.term*) {arity > 0}
or *name.method* {arity = 0}

Remote Message Receiver

Remote receivers of messages are Prolog++ terms which are instantiated at run-time to classes or instances of classes.

receiver
→ *meta.receiver*
or *term.instance*
or *term.name.class*

Prolog++ Terms

A few examples of Prolog++ terms are :-

```
Buffer                                  % Prolog variable

-12345.67890                            % Prolog number

& size                                  % Symbolic context
&( aggregate_size / clock@time )

+( aggregate_size / clock@time )        % Arithmetic context
+Number+1

@ size                                  % Explicit local
@ smallest( all instance class )        % functional messages

clock @ time                            % Remote functional
@serves_teller @ status                 % messages

X suchthat X = instance teller          % Conditional term
X suchthat ( X=1 ; X=2 )
```

```
all instance teller                    % All instantiations

[ Queue | Queues ]                     % Non-empty list constructor

[]                                     % Empty list constructor

self                                   % Instantiates to self

super                                  % Instantiates to super
                                       % of current class


class                                  % Class name of an object
identifier                             % Unique instance identifier
mnemonic                               % Mnemonic name of instance
$ fred                                 % Instance with mnemonic name

category user                          % All classes in the category
super_class teller                     % A super-class
sub_class bank                         % A sub-class
ancestor_class customer_queue          % Any ancestor class
descendant_class buffer                % Any descendant class
instance teller                        % Any instance of the class

composite_part Spoke                   % Composite-part of instance
super_part Spoke                       % Super-part of an instance
sub_part Wheel                         % A sub-part of an instance
wheel # 1                              % A specific sub-part of self

X + Y                                  % Arithmetic operators
-Z

mary  % Symbolic atom
                                       %   non-evaluable context
tree( Left, Node, Right )              % Symbolic functor
                                       %   non-evaluable context


size                                   % Implicit local
                                       % functional messages
smallest( all instance class )         % when context is evaluable
```

The ordering given below is exactly the same ordering adopted by the Prolog++ compiler when parsing terms.

```
term
    →   prolog_variable                          {Prolog variable}
    or  prolog_number                            {Prolog number}
    or  & prolog_term                            {Force it to be a Prolog term}
    or  + term                                   {Evaluate term as an arithmetic expression}
    or  @ functional_message                     {Local functional message}
    or  receiver @ functional_message            {Remote functional message}
    or  term suchthat statement                  {Term such that statement holds}
    or  all term                                 {All instantiations of a term}
    or  [ term | term ]                          {Non-empty list constructor}
    or  []                                       {Empty list constructor}
    or  self                                     {Instantiates to the self variable}
    or  super                                    {Instantiates to a super of the defined class}
    or  link                                     {A structural, is-a or part-of link}
    or  term binary_operator term                {Binary arithmetic operator}
    or  unary_operator term                      {Unary arithmetic operator}
    or  prolog_atom                              {Prolog atom; not evaluable context}
    or  name.functor( sequence.term )            {Constructor; not evaluable context}
    or  functional_message                       {Implicit functional message sent to self}
```

There are three types of links in Prolog++. A structural link about an instance or class, an is-a hierarchy relating classes to eachother and a part-of hierarchy relating instances of classes to eachother.

```
link
    →   structural_link
    or  is_a_link
    or  part_of_link
```

Context

The various contexts in which a term can occur are :-

symbolic Terms are considered purely symbolic, as in Prolog. This context is only attainable within the context switcher **&**.

prolog++ Certain special terms, such as hierarchy links, are evaluated. This is the normal context for terms.

evaluable In addition to the *prolog++* context all other atoms and functors (except arithmetic operators) are considered as implicit local messages. This context is used for the sub-terms of arithmetic expressions.

arithmetic In addition to the *evaluable* context all arithmetic operators are considered evaluable by the Prolog built-in *is/2*. This context is attainable within the explicit context switcher **+**, arithmetic default attribute values (using **is** rather than **=**), arithmetic definitions of functional methods (again using **is**) and calls to arithmetic Prolog built-ins such as *is/2*, *>=/2* and *tab/1*.

Note that the context of a term is preserved for all of its immediate sub-terms. The only exception to this rule is that receivers of remote functional messages are always interpreted in the *Prolog++* context.

Functional Message

A functional message will be handled by a functional method either defined in or inherited by the class of the receiver.

functional_message

→	*name.method*(*sequence.term*)	{arity > 0}
or	*name.method*	{arity = 0}

As with procedural message passing, there is a subtle difference between local functional messages (early binding) and remote functional messages sent to the *self* variable (late binding).

Structural Link

A structural link relates an instance, explicit or implict, with either its class name, its unique identifier or with its mnemonic name assigned to it by the program when it is created.

structural_link

→	explicit_structural_link	
or	implicit_structural_link	{relative to *self* variable}

explicit_structural_link

→	**class**	*term.instance*	{Instance's class name}
or	**identifier**	*term.instance*	{Instance's identifier}
or	**mnemonic**	*term.instance*	{Instance's mnemonic}

implicit_structural_link

→	**class**		{*self's* class name}
or	**identifier**		{*self's* identifier}
or	**mnemonic**		{*self's* mnemonic}
or	**$**	*name*	{Instance with mnemonic name}

Is-A Link

Is-a links refer to classes in the class hierarchy or an instance of a class. They can explicitly refer to a class or instance or they can implicitly refer to the *self* variable.

```
is_a_link
    →   explicit_is_a_link
    or  implicit_is_a_link                          {relative to self variable}
```

Some constructs (**super_class** and **ancestor_class**) reference classes higher up the is-a hierarchy whereas others (**instance**, **sub_class**, and **descendant_class**) reference classes and instances of classes lower down the is-a hierarchy.

The construct **category** refers to all the classes in the given category.

```
explicit_is_a_link
    →   category          term.category      {All classes in the category}
    or  super_class       term.name.class    {A super-class of a class}
    or  sub_class         term.name.class    {A sub-class of a class}
    or  ancestor_class    term.name.class    {An ancestor of a class}
    or  descendant_class  term.name.class    {A descendant of a class}
    or  instance          term.name.class    {An instance of a class}

implicit_is_a_link
    →   category                             {All classes in self category}
    or  super_class                          {A super-class of self}
    or  sub_class                            {A sub-class of self}
    or  ancestor_class                       {An ancestor class of self}
    or  descendant_class                     {A descendant class of self}
    or  instance                             {An instance of self}
```

Part-Of Link

Part-of links refer to instances in the run-time part-of hierarchy. They can explicitly refer to an instance or they can implicitly refer to the *self* variable.

```
part_of_link
    →   explicit_part_of_link
    or  implicit_part_of_link                       {relative to self variable}
```

Note that in contrast to is-a hierarchies which can be lattices (a class may have multiple super-classes) part-of hierarchies are strict (all instances have at most one super-instance). Consequently, the root of a part-of hierarchy is unique and is referred to as the **composite_part**.

Some constructs (**composite_part** and **super_part**) refer to instances higher up the part-of hierarchy, whereas others (**sub_part** and **#**) refer to instances lower down the part-of hierarchy.

The **composite_part** is at the top or root of a part-of hierarchy.

The **super_part** is that unique instance directly above in the part-of hierarchy.

A **sub_part** is any instance directly below in the part-of hierarchy.

A specific sub-part of *self* can be referenced using the **#** construct, where the left argument refers to the name of the sub-part and the right argument refers to its occurrence number.

explicit_part_of_link
→ **composite_part** term.instance
or **super_part** term.instance
or **sub_part** term.instance

implicit_part_of_link
→ **composite_part**
or **super_part**
or **sub_part**
or term.name.part_class **#** term.positive_number

Appendix B - Shorthand Declarations

This appendix summarises the various shorthand statements which can be made within Prolog++ class definitions.

Shorthand Declaration	Full Declaration
category	categories
inherit	inherits
part	parts
attributes	private instance attributes
private attributes	private instance attributes
public attributes	public instance attributes
instance attributes	private instance attributes
class attributes	private class attributes
attribute	private instance attributes
private attribute	private instance attributes
public attribute	public instance attributes
instance attribute	private instance attributes
class attribute	private class attributes
private instance attribute	private instance attributes
private class attribute	private class attributes
public instance attribute	public instance attributes
public class attribute	public class attributes
methods	public methods
public	public methods
private	private methods
method	public methods
public method	public methods
private method	private methods

Appendix C - Prolog++ Operators

This appendix details the syntactic operators which are declared by Prolog++.

Priority	Type	Operator Name	Category
1020	fx	public	Declarations
1020	fx	private	
1010	fx	categories	
1010	fx	category	
1010	fx	inherits	
1010	fx	inherit	
1010	fx	parts	
1010	fx	part	
1010	fx	attributes	
1010	xfx	attributes	
1010	fx	attribute	
1010	xfx	attribute	
1010	fx	methods	
1010	xfx	methods	
1010	fx	method	
1010	xfx	method	
980	fx	inherited	Default Attributes
980	xf	inherited	
970	fx	forall	Control Constructs
970	fx	while	
970	fx	repeat	
960	xfx	do	
960	xfx	until	
950	xfx	<-	Procedural Message Passing
700	xfx	&=	Noisy Attribute Assignment
700	xfx	:=	
700	xfx	+=	
700	xfx	-=	
700	xfx	*=	
700	xfx	/=	
700	xfx	&==	Quiet Attribute Assignment

700	xfx	:==	
Priority	**Type**	**Operator Name**	**Category**
700	xfx	+=	
700	xfx	-==	
700	xfx	*==	
700	xfx	/==	
650	fy	&	Context Switching
650	fy	+	
600	fy	all	Terms Qualifiers
600	xfx	suchthat	
100	fy	class	Program Delimiters
100	fy	end	
100	fy	identifier	Structural Links
100	fy	mnemonic	
100	fy	super_class	Is-A Links
100	fy	sub_class	
100	fy	ancestor_class	
100	fy	descendant_class	
100	fy	instance	
100	fy	composite_part	Part-Of Links
100	fy	super_part	
100	fy	sub_part	
100	yfx	#	
50	fx	$	Mnemonic Name
45	yfx	@	Functional Message Passing
45	fy	@	

Appendix D - Summary Of "prolog_plus_plus"

The table below summarises the public methods of the *prolog_plus_plus* class, and in each case indicates whether the message should be sent to a class or an instance of a class. In some cases, for example *compile/2*, it is irrelevant where the message is sent. In other cases, for example *isa_class/0* and *isa_instance/0*, it can be sent to either kind of object. Finally, certain messages such as *create/1* can only be sent to a specific kind of object as indicated in the self column.

Method	Self	Description
compile / 2		The equivalent of the Prolog++ class compiler which allows new classes to be incorporated at run-time.
create / 1	class	Create a new instance of the class.
create / 2	class	Create a new instance of the class with some initial values for its attributes.
create / 3	class	Create a new instance of the class with some initial values for its attributes and also give it a mnemonic name by which it can be referenced.
delete / 0	instance	Delete this instance.
delete_all / 0	class	Delete all instances of this class.
dump / 1	class	Dump all instances of this class to a file.
dump / 2		Dump all instances of the named classes to a file.
duplicate / 1	instance	Duplicate this instance by creating a copy with the same attribute values.
isa_class / 0	class / instance	Test whether the object is a class.
isa_instance / 0	class / instance	Test whether the object is an instance of a class.
kill / 0	class	Remove the definition of a class.
load / 1		Load the previously saved class definitions from a file.
optimize / 0	class	Optimize a class's implementation.
public / 1	class	Test or retrieve the name and arity of a public method for this class.
reset / 0	class / instance	Reset the attributes of this class/instance back to their default values.

Method	Self	Description
reset / 1	class / instance	Reset the named attribute of this class/instance back to its default value.
restore / 1		Restore the previously saved class instances from a file.
save / 1	class	Save this class's defintion to a file.
save / 2		Save the definitions of the named classes to a file.
when_error / 2	class / instance	A catchall method for handling errors which were raised by the system but not handled by the erroneous class.

Appendix E - Benchmark

This appendix details the relative performance of Prolog++ with respect to the underlying Prolog system. The benchmark used is an adaptation of the classical Prolog benchmark "Naive Reverse".

Naive reverse is a deliberately bad encoding of a program which reverses the elements of a list. It is a test of the LIPS (Logical Inferences Per Second) attained by a Prolog system, using a non-linear equation between the length of the list and the number of recursive calls which are made.

The intention of using this benchmark in a Prolog++ context is to test the performance of both the message passing system and the inheritance mechanism. It is not only a naive encoding of reverse, but also a deliberately bad structuring of the class hierarchy in which it is defined.

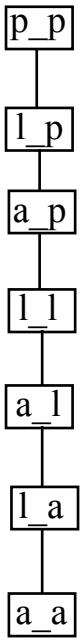
The Prolog Program

Naive reverse is implemented by a recursive program *nrev/2* which calls a second recursive program, *app/3*. These same Prolog definitions will be used when defining the Prolog++ methods.

```
nrev( [], [] ) .
nrev( [U|X], Z ) :-
      nrev( X, Y ),
      app( Y, [U], Z ) .

app( [], Z, Z ) .
app( [U|X], Y, [U|Z] ) :-
      app( X, Y, Z ) .
```

The Prolog++ Classes

The following diagram illustrates the linear class hierarchy within which the various Prolog++ versions of the naive-reverse algorithm are implemented.



The essential difference between Prolog and Prolog++ in terms of performance is the inheritance mechanism of Prolog++. And so the purpose of designing the hierarchy this way is to create classes which define or inherit the two recursive methods, *nrev/2* and *app/3*, in different combinations. This is set out in the following table.

Class	***nrev / 2***	***app / 3***
p_p	Inherit from prolog	Inherit from prolog
l_p	Defined locally	Inherit from prolog
a_p	Inherit from ancestor	Inherit from prolog
l_l	Defined locally	Defined locally
a_l	Inherit from ancestor	Defined locally
l_a	Defined locally	Inherit from ancestor
a_a	Inherit from ancestor	Inherit from ancestor

```
class p_p.
          category benchmark.
end   p_p.

class l_p.
          category benchmark.
          inherit  p_p.
          nrev( [], [] ).
```

```
            nrev( [Head|Tail], Rev ) :-
              nrev( Tail, TailRev ),
              app( TailRev, [Head], Rev ).
end   l_p.

class a_p.
            category benchmark.
            inherit  l_p.
end   a_p.

class l_l.
            category benchmark.
            inherit  a_p.
            nrev( [], [] ).
            nrev( [Head|Tail], Rev ) :-
              nrev( Tail, TailRev ),
              app( TailRev, [Head], Rev ).
            app( [], List, List ).
            app( [Head|Front], Back, [Head|List] ) :-
              app( Front, Back, List ).
end   l_l.



class a_l.
            category benchmark.
            inherit  l_l.
            app( [], List, List ).
            app( [Head|Front], Back, [Head|List] ) :-
              app( Front, Back, List ).
end   a_l.

class l_a.
            category benchmark.
            inherit  a_l.
            nrev( [], [] ).
            nrev( [Head|Tail], Rev ) :-
              nrev( Tail, TailRev ),
              app( TailRev, [Head], Rev ).
end   l_a.

class a_a.
            category benchmark.
            inherit  l_a.
end   a_a.
```

Benchmark Results

The listing below gives the results of the benchmark specified in the previous sections. The test was performed on a Macintosh LC475 with 8Mb of RAM running MacProlog32 version

1.05. The application was assigned 5Mb of real memory, 2Mb of which formed the evaluation space.

The percentage figures compare the speed of Prolog++ with respect to the base speed of Prolog, and thus reflecting the overhead incurred by the inheritance mechanism.

```
/* ***************************  *
*  Machine:     Macintosh LC475  *
*  Memory:      8Mb              *
*  Application: MacProlog32      *
*  Version:     1.05             *
*  Memory:      5Mb              *
*  Heap Space:  2Mb              *
*  *************************** */

?- bench( 500 ).

Interpreted...
DIRECT  nrev/2                      6300ms
Prolog  nrev/2                      6300ms  100%
Local   nrev/2  Prolog  app/3       6433ms   98%
Inherit nrev/2  Prolog  app/3       6416ms   98%
Local   nrev/2  Local   app/3       6300ms  100%
Inherit nrev/2  Local   app/3       6300ms  100%
Local   nrev/2  Inherit app/3       6483ms   97%
Inherit nrev/2  Inherit app/3       6483ms   97%

Optimized...
DIRECT  nrev/2                       416ms
Prolog  nrev/2                       416ms  100%
Local   nrev/2  Prolog  app/3        516ms   81%
Inherit nrev/2  Prolog  app/3        516ms   81%
Local   nrev/2  Local   app/3        416ms  100%
Inherit nrev/2  Local   app/3        416ms  100%
Local   nrev/2  Inherit app/3        416ms  100%
Inherit nrev/2  Inherit app/3        416ms  100%

No.1 : yes
```

Appendix F - Glossary

The following table is a glossary of reserved words used in Prolog++.

Reserved Word	Description
^	Existential term in set statements.
!	Prolog cut.
#	Infix operator relating the class name of a specific sub-part of *self* with that part's occurrence number.
&	Switch to a symbolic context.
&=	Noisy attribute assignment of a symbolic value.
&==	Quiet attribute assignment of a symbolic value.
$	Find an instance with a mnemonic name.
*	Infix operator relating the class name with its number of occurrences when defining a part.
*=	Noisy multiplication of an arithmetic attribute value.
*==	Quiet multiplication of an arithmetic attribute value.
+	Switch to an arithmetic context.
+=	Noisy addition to an arithmetic attribute value.
+=	Quiet addition to an arithmetic attribute value.
,	Separator for a conjunction of statements, messages or receivers, and for a sequence of parameters or arguments.
-=	Noisy subtraction from an arithmetic attribute value.
-==	Quiet subtraction from an arithmetic attribute value.
->	Separator for conditional statements.
/	Declaration of a procedural method's arity.
//	Declaration of a functional method's arity.
/=	Noisy division of an arithmetic attribute value.
/==	Quiet division of an arithmetic attribute value.
:-	Preceeds statements in the definition of a method.
:=	Noisy attribute assignment of an arithmetic value.
:==	Quiet attribute assignment of an arithmetic value.
;	Separator for a disjunction of statements.
<-	Remote message passing symbol.

Reserved Word	Description
=	Regular equality used in the declaration of an attribute's default value or the definition of a functional method.
@	Precedes a local functional message.
@	When used as an infix operator it is a remote functional message.
all	All values of a term.
ancestor_class	An ancestor of a class.
attributes	Declaration of the attributes defined in a class.
bagof	Collect solutions as a bag.
categories	Declaration of the categories in which a class exists.
category	Evaluates to the list of classes forming the named category.
class	Compulsory statement at the beginning of each class definition.
class	Prefix to attributes indicating the declaration of class rather than instance attributes.
class	The class name of an instance.
composite_part	The composite part of *self* at the top (or root) of its part-of hierarchy.
descendant_class	A descendant of a class.
do	Used with forall for universal statements and with while for repetitive statements.
end	Compulsory statement at the end of each class definition.
fail	Forced failure.
findall	Collect solutions as a list.
forall	Used with do for universal statements.
identifier	Unique identifier of an instance.
inherits	Declaration of one or more super-classes.
instance	Prefix to attributes indicating the declaration of instance rather than class attributes.
instance	An instance of a class.
invalid	Method name for contraint checking procedures before an attribute is assigned a new value.
is	Arithmetic equality used in the declaration of an attribute's default value or the definition of a functional method.
methods	Declaration of the procedural and functional methods used to handle messages sent to a class (or any of its instances).
mnemonic	Mnemonic name of an instance assigned to it when created.

not	Sound negation for statements.

Reserved Word	Description
parts	Declaration of one or more component parts. Parts are created whenever an instance of the class is created.
private	Declaration of private attributes and methods not visible outside the class.
public	Declaration of public attributes and methods which are visible outside the class.
repeat	Repeat until failure.
repeat	Used with until for repetitive statements.
self	The instance to which the message being handled was originally sent.
setof	Collect solutions as a set.
sub_class	A sub-class of a class.
sub_part	A sub-part of *self*.
suchthat	Used to impose conditions on a term.
super	A super-class of the class being defined. This is used to force messages up the is-a hierarchy whilst preserving the value of the *self* variable.
super_class	A super-class of a class.
super_part	The super-part which *self* is a direct sub-part of.
true	Vacuous success.
until	Used with repeat for repetitive statements.
when_assigned	Method name for dæmon procedures activated after an attribute is assigned a new value.
when_created	Method name for procedures activated after the creation of a new class instance.
when_deleted	Method name for procedures activated immediately before the deletion of a class instance.
when_error	Method name for procedures which handle errors.
while	Used with do for repetitive statements.
\+	Negation-as-failure for statements.

Index