4.900

flex Reference

by Dave Westwood

WIN-PROLOG *flex* Technical Reference

Written by Dave Westwood

Logic Programming Associates Ltd
Studio 30
The Royal Victoria Patriotic Building
Trinity Road
London SW18 3SX England

phone: +44 (0) 20 8871 2016
fax: +44 (0) 20 8874 0449
email: support@lpa.co.uk
web: http://www.lpa.co.uk

flex Expert System Toolkit

Version 1.2

Edition 4

Written by	*Phil Vasey*	*- April 1989*
Revised by	*David Westwood*	*- February 1990*
Updated by	*David Westwood*	*- August 1992*
Revised by	*Nicky Johns*	*- January 1996*

Logic Programming Associates Ltd
Studio 30
The Royal Victoria Patriotic Building
Trinity Road
London SW18 3SX
ENGLAND

The contents of this manual describe the product, *flex™ 1.2*, and are believed correct at 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.

flex was designed and written by Phil Vasey, Logic Programming Associates Ltd.

Welcome to the *flex* expert system toolkit.

Flex was originally developed by LPA in 1988 as a Prolog-based toolkit for LPA Prolog programmers using MS-DOS. Since then it has matured into a general purpose expert systems toolkit, and has been ported to many different hardware and software environments.

Flex has been widely used in industry and research in various applications such as modelling and simulation, legal reasoning, expert advisory systems, scheduling, planning, and diagnostics.

Flex is licensed to ICL/TeamWare as part of the TeamWare/ProcessWise portfolio of Business Process Management range of software products, and has been used in various re-engineering projects in the UK and overseas.

Flex has been licensed to the Open University in the UK for re-distribution to students enrolled on their course, "T396: Artificial Intelligence for Technology".

Contents

1. Introduction

Welcome to the flex expert system toolkit.

We wish you success with generating your expert system applications using flex.

What is flex ?

flex is an expressive and powerful expert system toolkit which supports frame-based reasoning with inheritance, rule-based programming and data-driven procedures fully integrated within a logic programming environment, and contains its own English-like Knowledge Specification Language (KSL).

flex goes beyond most expert system shells in that it employs an open architecture and allows you to access, augment and modify its behaviour through a layer of access functions. Because of this, *flex* is often referred to as an AI toolkit. The combination of *flex* and Prolog, i.e a hybrid expert system toolkit with a powerful general-purpose AI programming language, results in a functionally rich and versatile expert system development environment where developers can fine tune and enhance the built-in behaviour mechanisms to suit their own specific requirements.

flex appeals to various groups of developers; expert systems developers who want to deliver readable and maintainable knowledge-bases, advanced expert system builders who want to incorporate their own controls, AI programmers who want access to a high-level language-based product and Prolog programmers who require extra functionality and structures.

What are expert systems?

Expert systems (or knowledge-based systems) allow the scarce and expensive knowledge of experts to be *explicitly* stored into computer programs and made available to others who may be less experienced. They range in scale from simple rule-based systems with flat data to very large scale, integrated developments taking many person-years to develop. They typically have a set of **`if-then`** rules which forms the *knowledge base*, and a dedicated *inference engine*, which provides the execution mechanism. This contrasts with conventional programs where domain knowledge and execution control are closely intertwined such that the knowledge is *implicitly* stored in the program. This explicit separation of the knowledge from the control mechanism makes it easier to examine knowledge, incorporate new knowledge and modify existing knowledge.

Forward Chaining

Forward chaining *production rules* in *flex* follow the classical **`if-then`** rule format. Forward chaining is data-driven and is very suitable for problems which involve too many possible outcomes to check by backward chaining, or where the final outcome is not known.

The forward chaining inference engine cycles through the current rule agenda looking for rules whose **`if`** conditions can be satisfied, and selects a rule to use or *fire* by executing its **`then`** part. This typically side affects data values, which means that a different set of rules now have their conditions satisfiable.

Flex extends the classical production rule with an optional explanation facility and dynamic scoring mechanism for resolving conflicts during rule selection. Rules can have multiple conclusions or actions (either positive or negative) in their **`then`** part .

The rule selection and agenda update algorithms of the forward chaining engine are flexible, with many built-in algorithms and the option of applying user-defined algorithms.

Backward Chaining

Backward chaining rules, which correspond closely to Prolog predicates, are called *relations* in *flex*. They have a single conclusion that is true, if all the conditions can be proven. Backward chaining is often referred to as goal-driven, and is closely linked to the notion of provability.

Search

Search is one of the key characteristics of expert systems. There are normally many ways of combining or chaining rules together with data to infer new conclusions. How to examine only the relevant part of this *search space* is a serious consideration with regard to efficiency. The ordering of rules, the provision of meta-rules (rules about which rules to use) and conflict-resolution schemes are all ways of helping us produce a sensible *search tree* which we can investigate. Prolog-based systems tend to use a depth-first strategy, whereby a certain path is fully explored by checking related paths, combined with *backtracking* to go back and explore other possibilities when a dead-end is reached.

Frames and Inheritance

Frame hierarchies are similar to object-oriented hierarchies. They allow data to be stored in an abstract manner within a nested hierarchy with common properties automatically inherited through the hierarchy. This avoids the unnecessary duplication of information, simplifies code and provides a more readable and maintainable system.

Each **frame** or **instance** has a set of slots that contain attributes describing the frame's characteristics. These slots are analogous to fields within records (using database terminology) except that their expressive power is greatly extended.

Frames **inherit** attribute-values from other frames according to their position in the frame hierarchy. This inheritance of characteristics is automatic, but can be controlled using different built-in algorithms.

Questions and Answers

Flex has a built-in **question** and **answer** sub-system that allows final applications to query the user for additional input via interactive dialogs. These screens can be simple pre-defined ones, or complex, sophisticated screens constructed using Prolog's own screen handling facilities and then attached to the question and answer sub-system.

Explanations

Flex has a built-in explanation system which supports both *how* and *why* explanations. Explanations can be attached to both rules and questions using simple **because** clauses.

Data-driven Programming

Flex offers special procedures which can be attached to collections of frames, individual frames or slots within frames. These procedures remain dormant until activated by the accessing or updating of the particular structure to which they have been attached. There are four different types of data-driven procedures available within *flex*: **launches**, **demons**, **watchdogs**, and **constraints**.

Knowledge Specification Language

Flex has its own expressive English-like Knowledge Specification Language (KSL) for defining rules, frames and procedures. The KSL enables developers to write simple and concise statements about the expert's world and produce virtually self-documenting knowledge-bases which can be understood and maintained by non-programmers. The KSL supports mathematical, boolean and conditional expressions and functions along with set abstractions; furthermore, the KSL is extendable through synonyms and templates. By supporting both logical and global *variables* in rules, *flex* avoids unnecessary rule duplication and requires fewer rules than most other expert systems.

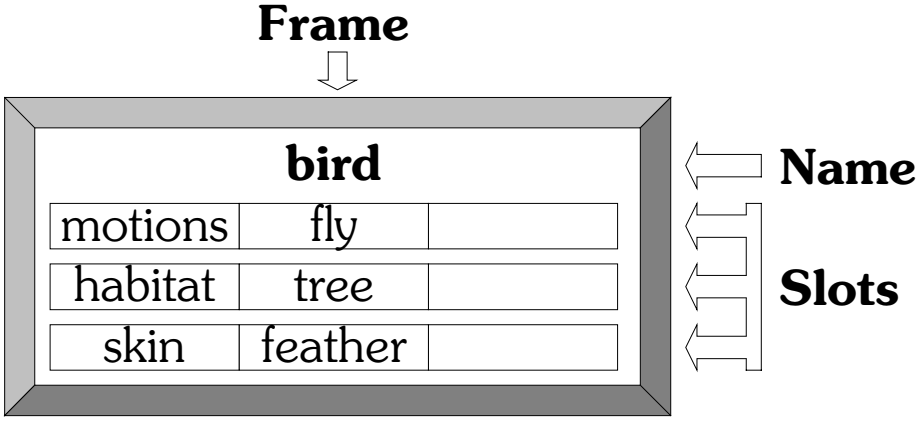
2. Frames and Inheritance

In this chapter we describe the frame sub-system of *flex*. This includes the form and content of individual frames, how frames are linked together to form a frame hierarchy, and how values are inherited through that hierarchy.

What is a Frame ?

A frame is similar to an object and is a complex data structure which provides a useful way of modelling real-world data objects.

Frames are analogous to records within a database but are far more powerful and expressive. Each individual frame has a name by which it is referred, details of its parent(s) frame, and a collection of slots or attributes (similar to fields within records) which will contain values or pointers to values. Slot values can be explicitly defined locally, or implicitly *inherited* from an ancestor frame further up the hierarchy.

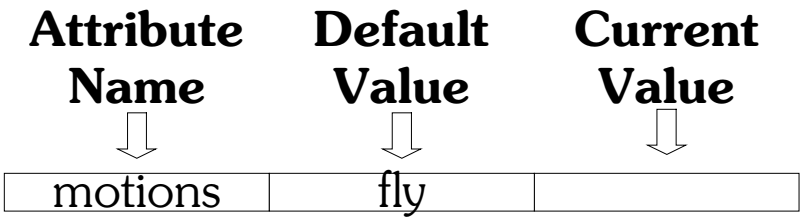


Flex has its own language (for representing frames and other constructs) called the Knowledge Specification Language. For example, the KSL code for the above frame could be:

```
frame bird
   default skin is feather and
   default habitat is a tree and
   default motions are { fly } .
```

Each slot has three principal components:

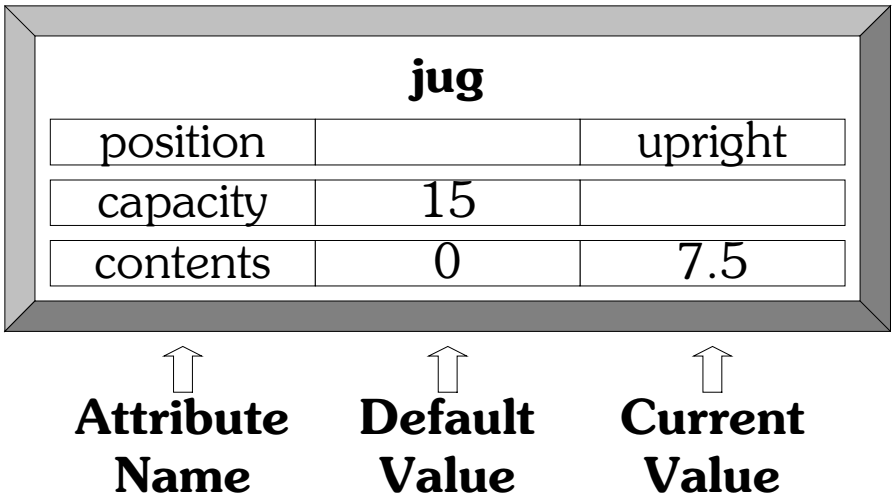
- attribute name - such as `habitat`, describing the concept
- default value - the default value, to be used when there is no current value
- current value - the current value for the attribute



A frame can be viewed as a dynamic array having three columns (*Attribute Name*, *Default Value* and *Current Value*) and an arbitrary number of rows, one for each slot.

It is important to note the difference between *default* and *current* values, since some *flex* operations work on only on *current* values.

The following example is an illustration of a frame representing the concept of a jug.



Slots may or may not have values. For example, there is a current but not a default value for the `position` of the `jug`, a default but not a current value for the `capacity` of the `jug`, and both a default and a current value for the `contents` of the `jug`. The default value for a slot is used *only* in the absence of a current value for that slot.

When a frame is declared in the KSL, the initial default values of its attributes may be declared, as in the above example of the frame `bird`. However, additional slots may be added dynamically simply by referring to them

and giving them a value. For example, the above `jug` frame may be declared in KSL as

```
frame jug
   default capacity is 15 and
   default contents is 0 .
```

Its `position` slot may then be created and its `contents` updated as follows, using a KSL `action` (described later).

```
action jug_update ;
   do the contents of the jug becomes 7.5 and
   the position of the jug becomes upright .
```

There are no restrictions on what terms can be used as the default or current values of slots. They can be any valid Prolog term. They can be calculations (or *access functions*), which are performed whenever you need the slot value.

```
frame box
   default width is 10 and
   default depth is  5 and
   default volume is its width times its depth .
```

Default Values

Default values are usually associated with general objects or classes, rather than specific instances. They are only used when a specific (current) value is not available.

The default values of slots normally remain throughout the lifespan of the frame: they are not intended to change dynamically (although *flex* does allow the creation at run-time of dynamic frames, instances, slots, default and current values).

Current Values

Current values are usually associated with specific instances rather than general classes. A current value for a slot *overrides* any default value which that particular slot may have.

The current value of a slot usually changes dynamically as further information is gathered at run-time, maybe as a result of a question being asked or some look-up in a database.

For example, if we are monitoring the environment within a mine, we can reflect changes in the physical environment by updating the appropriate current values. This is often as the consequence of some rule being fired or some action being executed.

```
rule methane_update
   if the temperature of the mine is above 66
   then the methane_level of the mine becomes slight .
```

So, whenever this rule is fired, the attribute `methane_level` of the frame `mine` is given the current value `slight`.

Linking Frames

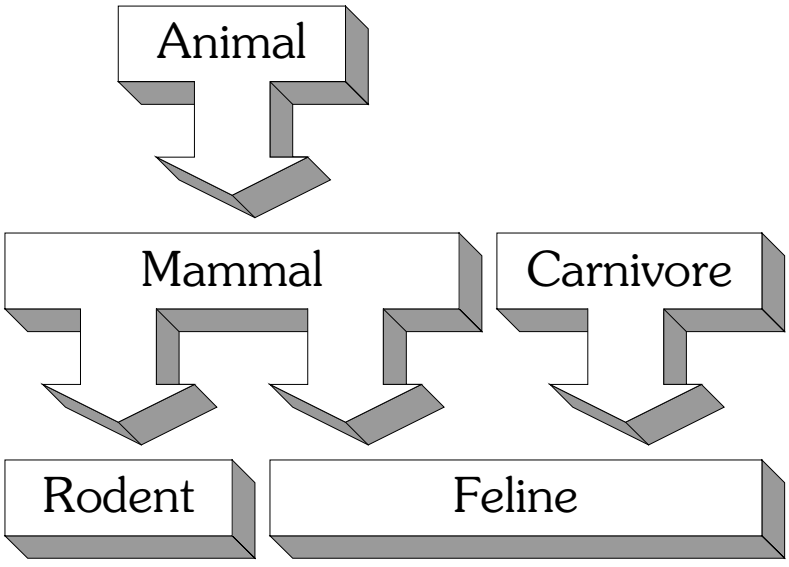
In the previous section we looked at the form and content of frames, which are structures for representing objects. The frame system, however, provides more than a data storage mechanism: by allowing objects to be linked to each other, it enables a *frame hierarchy* to be established.

The links between frames determine the overall structure of the frame hierarchy. Each link links a parent-frame to a child-frame. The child-frame can be thought of as a specialisation of the parent-frame, or, the parent-frame as a generalisation of the child-frame.

A child-frame can *inherit* values (both default and current) from any of its parent-frames, which in turn can inherit values from their parent-frames, and so on. In this way, information filters down from the frames at the top of the hierarchy to those leaf nodes at the bottom of the hierarchy.

This allows the distribution of information without duplication.

Let us consider a small section of the animal kingdom as below.



An arrow pointing from one frame to another indicates a parent-frame to child-frame link in the hierarchy.

The KSL frame declarations for the above diagram are as follows:

```
frame animal .

frame carnivore .

frame mammal is an animal
   default blood is warm and
   default habitat is land .

frame rodent is a kind of mammal
   default habitat is sewer .

frame feline is a mammal, carnivore .
```

There are three kinds of *isa* links used in this example hierarchy.

A single connection between the parent-frame and the child-frame, i.e. *one-to-one*. An example of this is the link between `animal` and `mammal`, and is the usual kind of link to be found in hierarchies.

Links from a parent-frame to more than one child-frame, i.e. *multiple children*. An example of this is the link between `mammal` and its two sub-classes, `rodent` and `feline`. In this case both `rodent` and `feline` are types of `mammal,` and share some characteristics (like warm blood), but not necessarily all characteristics.

Links from a child-frame to more than one parent-frame, i.e. *multiple parents*. An example of this is the linkage from `mammal` and `carnivore` to the class `feline`. In this case a `feline` has some of the characteristics of a `mammal` and some of the characteristics of a `carnivore`. Here, the ordering of the parenthood is important, as it affects the order for which inherited values are returned by the inheritance search algorithm. This capability to have more than one parent is sometimes referred to as *multiple inheritance*.

By inheritance from mammals, all felines and all rodents are warm-blooded. However, whilst all felines live on land, the locally declared default for a rodent's habitat will override the inherited default from mammals, with the result that rodents are deemed to live in sewers. If we now defined, say, a squirrel as a kind of rodent, then the default value for their habitat attribute would be a sewer.

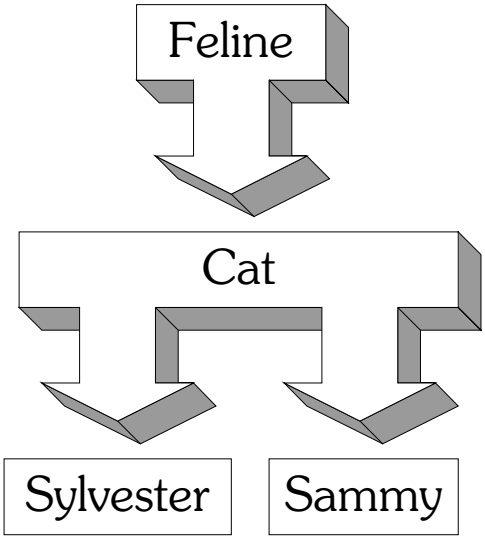
Creating an Instance of a Frame

So far we have discussed the use of frames to represent general (static) objects such as mammals, felines and rodents. However, frames may also represent specific (dynamic) instances of objects such as `Sylvester` (a well-known cat) or `Sammy` (my cat).

In formal terms there is very little to distinguish a *frame* representing a class of objects from an *instance* representing a specific instance of the frame. Instances appear as leaf nodes in the frame hierarchy and can have only one

single parent-frame. In addition, instances may only contain *current* values in their slots; they may not have default values declared.

Example



The instances are represented by a box without a shadow.

The KSL representation of the above is as follows.

```
frame feline is a mammal, carnivore
   default legs are 4 .

frame cat is a feline
   default habitat is house and
   default meal is kit_e_kat .

instance sylvester is a kind of cat .

instance sammy is an instance of cat .
```

Here, by default, both `sylvester` and `sammy` will live in a `house`, eat `kit_e_kat` and have 4 legs.

Overriding Inheritance

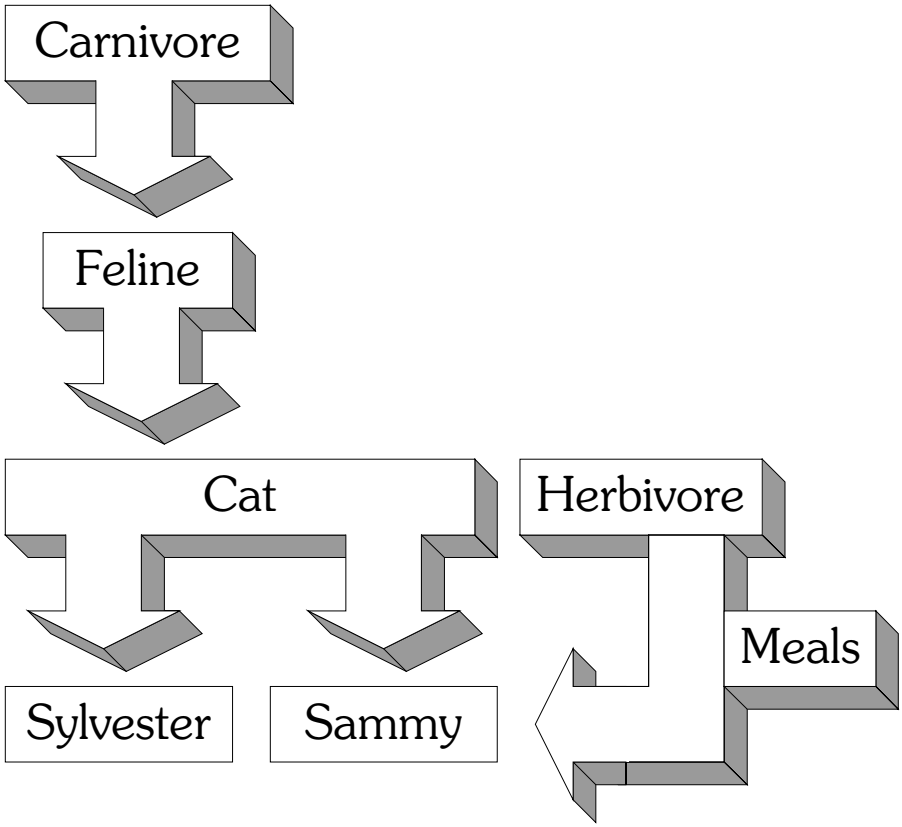
In our examples so far, a child-frame will automatically inherit from its parent-frames. We may wish, however, for a particular attribute to be inherited from a frame outside the hierarchy, or from a particular frame within the hierarchy, or not inherited at all.

Specialised Inheritance

In *flex* a special inheritance link may be defined that allows a specific attribute to be inherited from a specific frame.

For example, if we had a vegetarian instance of `cat` called `sammy`, we could

define the hierarchy such that the `meal` of `sammy` is specially inherited from `herbivore` and not by normal inheritance from `carnivore` (via `cat` and `feline`). Note that this only affects the `meal` attribute.



The corresponding KSL code would be:

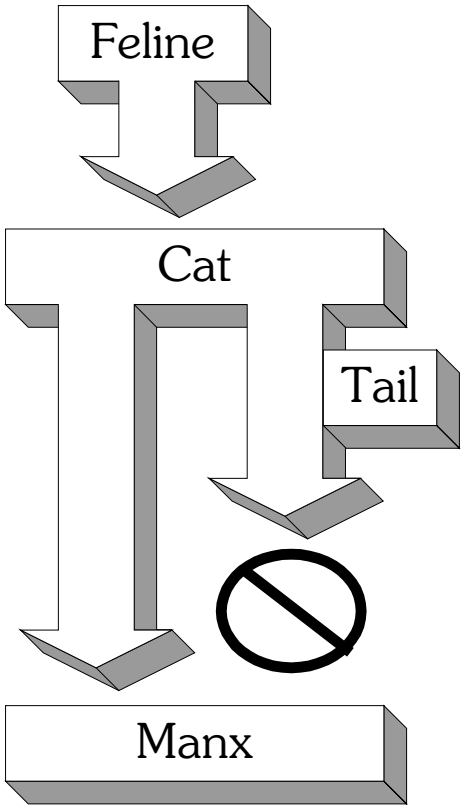
```
instance sammy is an instance of cat ;
   inherit meal from herbivore .
```

Note that `herbivore` is not a parent of `sammy`: it only contributes the `meal` attribute.

Negative Inheritance

In *flex* the inheritance of a particular attribute for a particular frame may be suppressed.

For example, `manx` cats do not have tails, if as part of our hierarchy we have defined the `tail` attribute in the `feline` frame. We could define the `manx` frame such that the inheritance of the `tail` attribute is suppressed.



The KSL code for this is as follows.

```
frame cat
   default tail is furry .

      frame manx is a cat
         do not inherit tail .
```

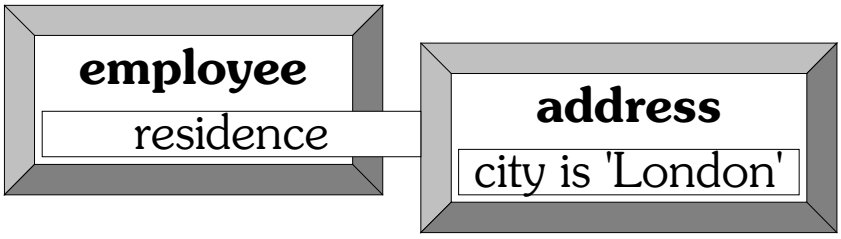
Attribute Chaining

Sometimes it may be convenient for an attribute to have its own *set* of values, and in this case slots may contain pointers to other frames rather than simple values.

Example

```
frame address
   default city is 'London' .

frame employee
   default residence is an address .
```



In this example, the value attached to the `residence` attribute of the `employee` frame is a pointer to another frame, namely `address`.

If we want to know the city of residence of an employee, we can refer to this in three different ways:

```
X is the residence of employee
and Y is the city of X
```

or

```
Y is the city of the residence of employee
```

or, using the operator **`s** as shorthand

```
Y is employee`s residence`s city
```

all of which make `London` the value of the variable Y.

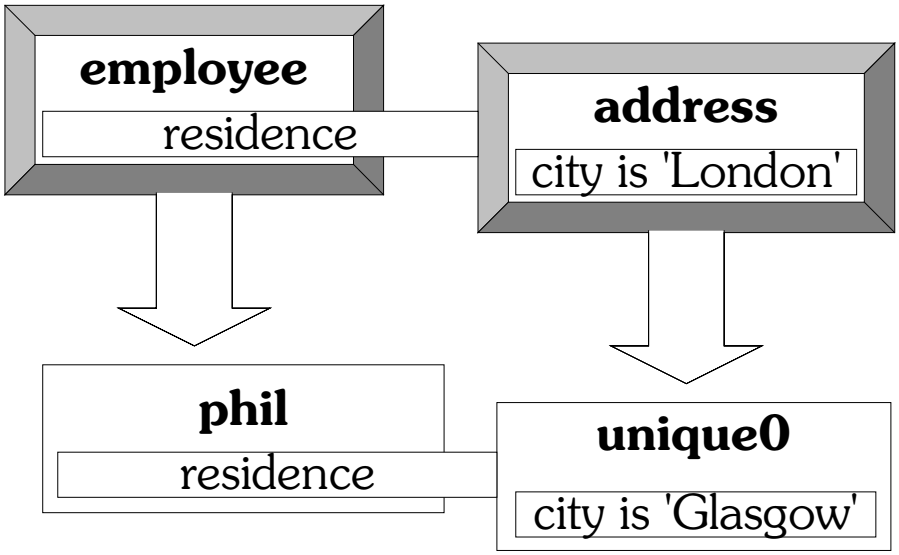
For example, if we create a new employee instance called `phil`, then it will be assumed that `phil` lives in `London`.

```
instance phil is an employee .
```

If, however, Phil does not live in London, but in Glasgow, then we can reflect this with the following directive.

```
do the city of residence of phil becomes 'Glasgow'
```

This has actually set up the following structure.



Global Variables

One special use of frames is to store global variables. These are defined as attributes of a special `frame` called `global`.

Example

```
frame global
   default current_interest_rate is 10.3 .
```

This creates a global variable called `current_interest_rate` which may then be referred to by any KSL statement.

The values of global variables may be updated at run-time.

Global variables are also used to store the response to a *flex* `question` - see the chapter on Questions.

Inheriting Values through the Frame Hierarchy

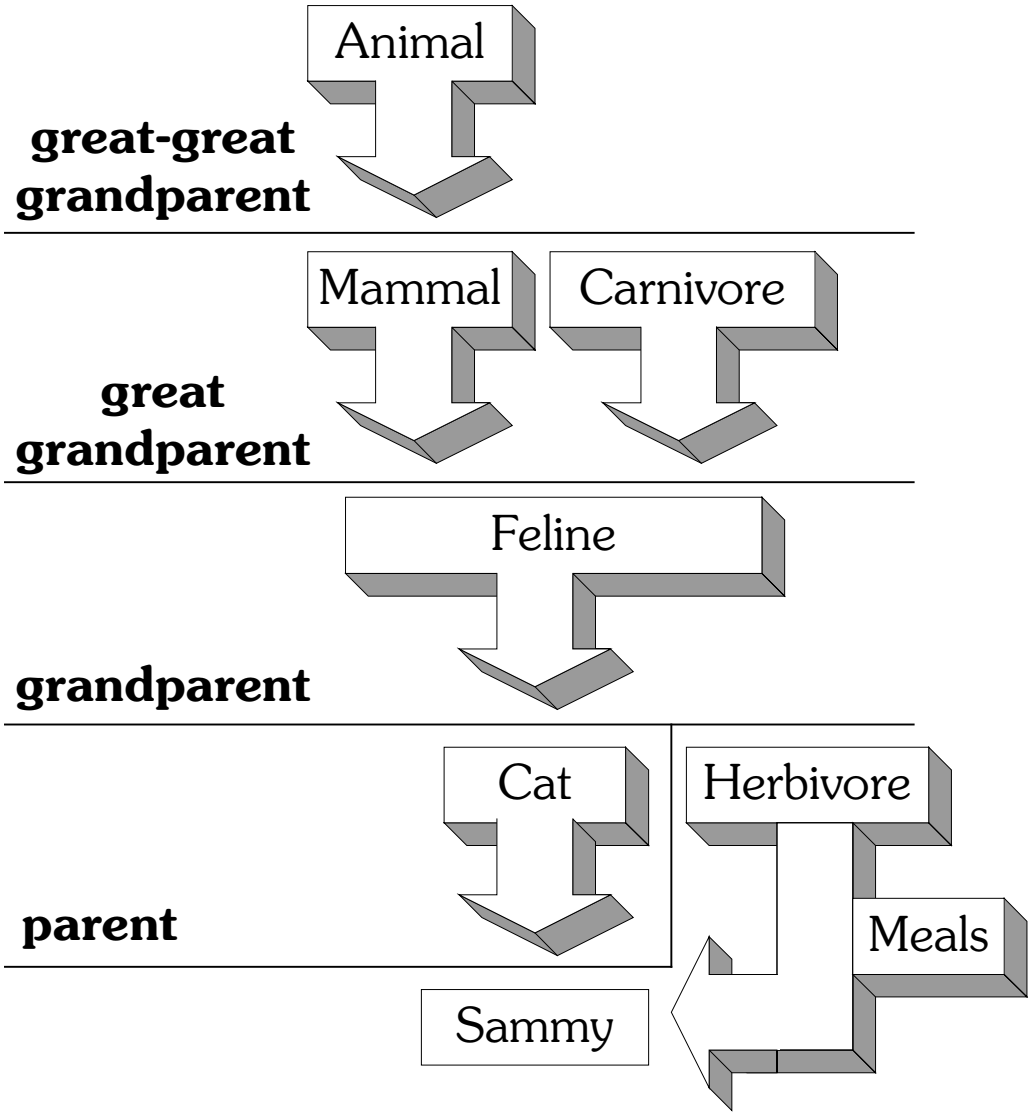
In earlier sections we discussed frames and their possibility for distributing information. In this section we shall discuss in more detail exactly how the distributed information passes from one frame to another.

In general, information flows down the frame hierarchy from those at the top to those at the tips. This is accomplished by *inheritance*.

Whenever there is a request for the some slot value, the inheritance algorithm is automatically invoked. The most obvious place to look first is in the original frame itself, since it may have either a current or a default value for the required attribute. Only if such a value does not exist locally will it be necessary to look elsewhere.

Where to look, in which order to look and when to stop looking are the subject matter of the following sections. We shall use the following example hierarchy to illustrate the different methods.

Example


Animal
great-great
grandparent
Mammal
Carnivore
great
grandparent
Feline
grandparent
Cat
Herbivore
parent
Meals
Sammy

Depth-First versus Breadth-First Search ?

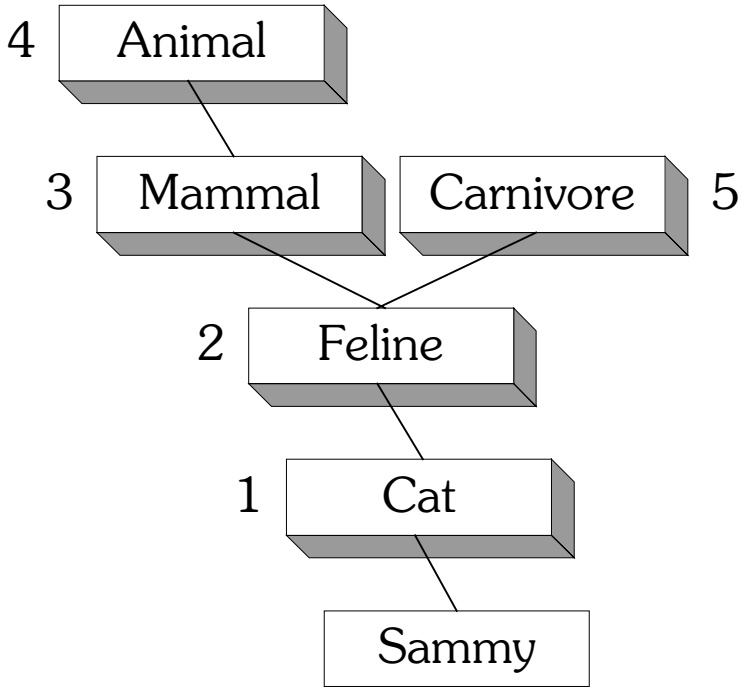
The frames to be considered, when looking for a potential place to inherit from, are determined by the frame hierarchy and the attribute in question.

Whenever there are alternative places to search, there are inevitably alternative ways of looking.

In both of the following methods, the search is left-to-right over the ordering of parent-frames.

A *depth-first* search of the frame hierarchy will investigate a complete ancestor branch before considering alternative ancestor branches.

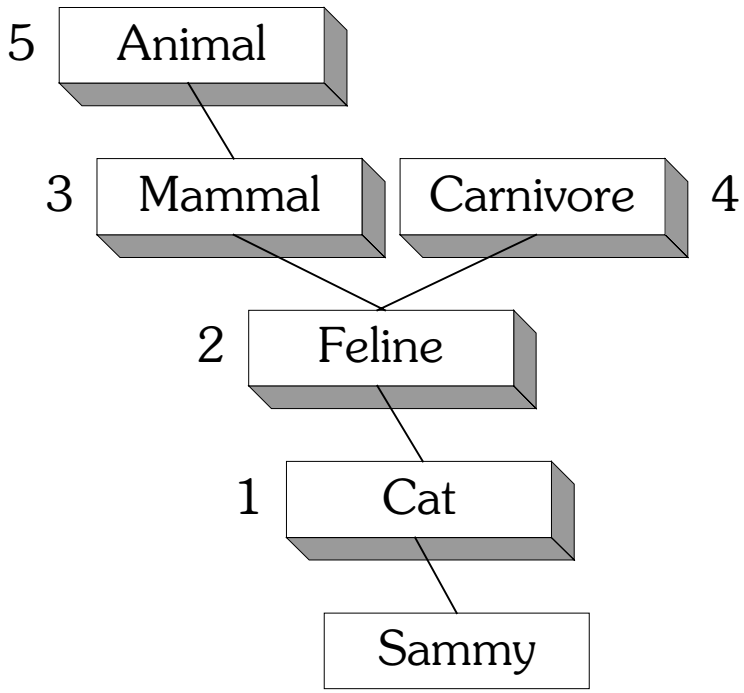
For example, a depth first search of the example hierarchy, starting at Sammy, will visit the following frames in order :



1 *Cat* being the parent-frame of Sammy
2 *Feline* being a grandparent-frame
3 *Mammal* being the first great-grandparent-frame
4 *Animal* being a great-great-grandparent-frame
5 *Carnivore* being the second great grandparent-frame

A *breadth-first* search, however, will visit all frames at a particular ancestor level before considering any at a higher level. It will visit all parent-frames first, then all grandparent-frames, all great-grandparent-frames, and so on.

For example, a breadth first search of the example hierarchy, starting at Sammy, will visit the following frames in order :



1	*Cat*	being the parent-frame of Sammy
2	*Feline*	being a grandparent-frame
3	*Mammal*	being the first great-grandparent-frame
4	*Carnivore*	being the second great grandparent-frame
5	*Animal*	being a great-great-grandparent-frame

An important characteristic of a breadth-first search is that the ultimate value returned will be from an ancestor-frame as close as possible to the original frame.

Depth-first is the default search strategy, and the more efficient, as it maps closely onto Prolog's own built-in search mechanism.

How Far Should the Search Go ?

For large frame hierarchies it may be desirable to limit the amount of *effort* used to search for inherited values. For this reason, *flex* provides a means whereby the maximum depth of search (a non-negative integer) can be imposed. This throttle is irrespective of whether the search procedure is depth first or breadth first.

For example, if the limit is 2 then only parent-frames and grandparent-frames will be considered.

In particular, if the limit is 0 then there will be no inheritance at all. The default value for effort is 9 levels.

Universal Defaults

Flex supports the notion of *universal default values*. These are default values which may be inherited by any frame in the hierarchy.

For example, in a system representing physical objects the notion of weight is universal. It is the product of the object's volume and density.

Rather than have a weight slot in each frame of the hierarchy, it would be more sensible to have a single global definition of weight which is universally accessible.

This is accomplished by having a special frame called **`root`**, which is always considered when inheriting values. The inheritance algorithm can be directed to visit the **`root`** frame either before or after visiting any ancestor frames.

Singular Versus Multiple Inheritance ?

Whenever a frame system allows for values to be inherited, there is the possibility for alternative answers according to where the inheritance comes from. This is in many ways similar to Prolog itself, which allows for alternative answers to the same query.

The *plurality* of inheritance within *flex* can either be *singular* (the default) or *multiple*. In both cases the search stops as soon as the first value is found.

For singular inheritance there is a commitment to this first value, and no others will ever be considered.

For multiple inheritance, however, alternative values can be obtained by *backtracking* using the inheritance algorithm, which is a Prolog program.

In reality, the singular inheritance algorithm is the same as the multiple inheritance algorithm except that it terminates with a cut (!) to avoid any potential backtracking.

Frame Relationships

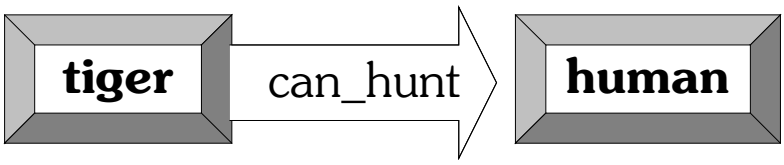
In its default setting, the only relationship between frames is the AKO (a-kind-of) hierarchy which defines how values are to be inherited. In general, though, it would be of great benefit to be able to define other relationships between frames, such as all tigers can hunt humans.

Example

```
frame tiger .

frame human .

relation can_hunt( tiger, human ) .
```



In its present form, the extension of the `can_hunt/2` relation contains only a single tuple, namely the pair `<tiger,human>`. If we were to pose the Prolog query …

```
?- prove( can_hunt( X, Y ) ) .
```

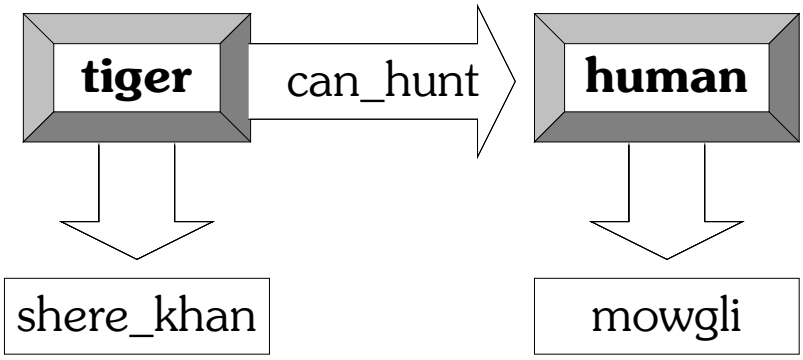
there would be a single solution which binds the indentifier `tiger` to the variable X, and binds the indentifier `human` to the variable Y. (`prove/1` is a built-in *flex* predicate, described later.)

Now consider two particular instances of tiger and human.

Example

```
instance shere_khan is a tiger .

instance mowgli is a human .
```



The answers to our query above will remain the same, namely only a single solution. This is because the underlying logic only allows unification between objects which have the same name (i.e. pattern-matching). The query

```
?- prove( can_hunt( shere_khan, mowgli ) ) .
```

would fail because `shere_khan` does not match `tiger`, and furthermore because `mowgli` does not match `human`.

The *flex* system allows the underlying logic to be changed from one involving unification to one involving inheritance.

That is, although ...

```
tiger     does not match     shere_khan
human     does not match     mowgli
```

with an inheritance logic we can show that ...

```
tiger     is an ancestor of     shere_khan     in the frame hierarchy
human     is an ancestor of     mowgli         in the frame hierarchy
```

and as such we can conclude that the `tiger shere_khan` can hunt the `human mowgli`. The underlying logic can be changed by issuing the Prolog command ...

```
?- new_logic( inherit ) .
```

It should be noted at this point that there is a general overhead involved in changing from a unification-based logic to an inheritance-based logic. That is, every procedure invocation will involve data lookup rather than direct pattern-matching.

It should also be noted that the inheritance-based logic can only be used for *checking* given values, and not for generating instances of relationships. For example, it can check that `shere_khan` can hunt `mowgli`, but will not be able to generate it.

3. Forward Chaining and Rules

In this chapter we shall describe the rules and inference engine of the forward chaining system and contrast it with backward chaining. Rules are really the life-blood of expert systems technology, and provide elegant, expressive and intuitive means of expressing knowledge.

A simple *rule* may be, '*if it is raining then carry an umbrella*'. Then given the *fact* that '*it is raining*', we can infer or derive that we should '*carry an umbrella*'. Facts can be thought of as degenerate cases of rules, i.e. rules without any pre-conditions, or rules whose pre-condition part is always true. Facts may be stored in a local database, retrieved from some external database, obtained through user question and answer interaction, stored as global variables, or derived from other rules.

Another rule may be '*if it is raining then the ground is wet*'. Now, given the same fact that '*it is raining*', we can also infer or derive that '*the ground is wet*', and store this as a fact, or, if `ground` is an attribute of some frame, set its value to `wet`.

The collection of all facts is often referred to as the *fact base*. In rule-based programming we use an inference engine to match rules against facts to produce new facts which means we can then use new rules, until we reach some final state.

Trying to prove the pre-conditions of forward chaining rules may well involve some backward chaining evaluation of sub-goals, i.e. there may be a backward chaining program for determining the current state of the weather. The integration of the two inference mechanisms can be as simple or as complicated as required. Notice, in the absence of information, execution fails and conditions in effect are deemed false. This is known as the *closed world assumption*.

We shall use the following propositional example to show the difference between backward chaining inferences and forward chaining inferences.

A if B and C .
B if D .
C .
D .

The first two statements are rules, and the second two statements are axioms (or facts).

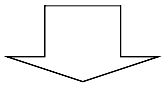
Forward Chaining

Forward chaining inference is very much akin to the way in which mathematical proofs are presented (but not necessarily arrived at). Using

the axioms as a starting point, further conclusions (lemmas in a mathematical proof) can be added to what is known by the bottom-up application of the rules. For example, since we already know D (it is an axiom), we can use the second rule (B if D) to conclude B. From the lemma B and the axiom C we can then use the first rule to conclude A. This is often referred to as *data-driven reasoning*, because we are using the data to drive forward the chain of inferences.

Within a forward chaining system, the rules above can be interpreted as:

given D, B may be inferred



given B and C, A may be inferred

What this style of reasoning suffers from is a lack of a sense of direction, or purpose. When deciding which rule to apply next, it is very difficult to decide which conclusion will get us closer to the desired goal. In this mode, you are likely to generate some valid configuration, with which you may or may not be happy.

Forward chaining has proved itself very suitable to configuration problems, where you do not know what the final configuration will be, but you know how to combine bits of data together according to some combining rules, and know that if you just keep applying rules, then eventually you will have combined everything accordingly.

A production rule system, as implemented in *flex*, can be viewed as a kind of bottom-up reasoning system where you push the data up through the rules to get more *inferred* data. Although the propositional rules here only have one action per rule (i.e. add the conclusion to what is known), a production rule may contain multiple actions and conclusions for updating the state of the system.

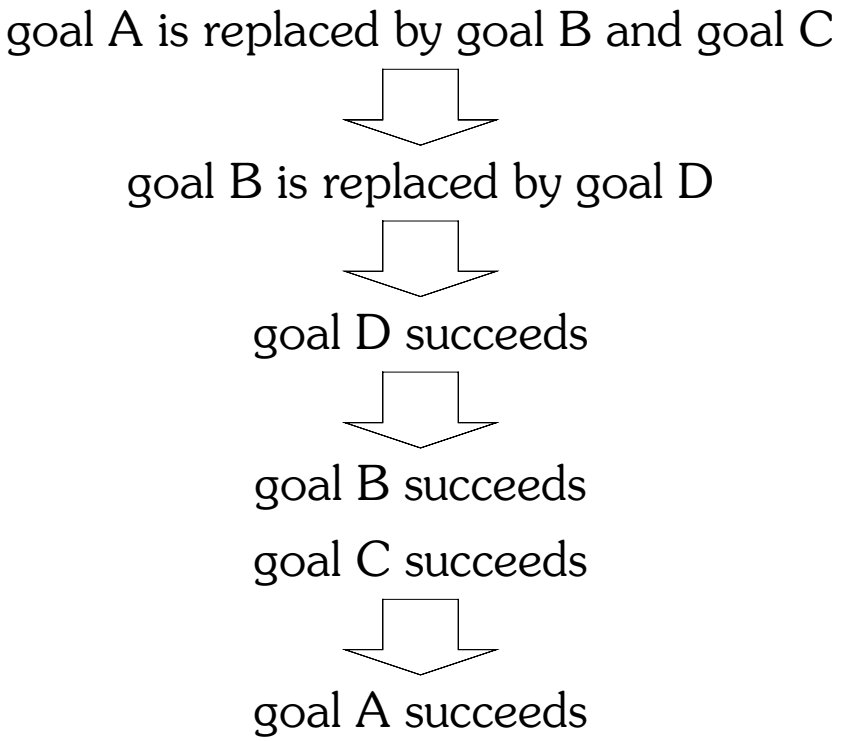
Backward Chaining

Backward chaining inference begins with a conclusion that needs to be established, say A. At each point during the inference, the rules are used to reduce the current goal or goals to a collection of sub-goals. For example, the first rule suggests that the goal of showing A can be reduced to the sub-goals of showing B and C. This process continues until all of the remaining sub-goals correspond to axioms, such as C and D. This is often referred to as *goal-driven reasoning*, because we are constantly trying to prove goals by proving their sub-goals.

This style of reasoning is very effective when we wish to determine whether or not a certain condition holds. As in Prolog, goals are replaced by sub-

goals which are then exploded using a left-to-right, depth-first search algorithm combined with backtracking.

Within a backward chaining system the rules above can be interpreted as:



Backward chaining has proved itself very suitable to provability and validation applications, but often requires a generate-and-test approach which itself can involve the checking of many, often redundant, combinations. The ideal solution is to mix and match, i.e. *interleave* both types of chaining. This leads to the main thrust of the computation being forwards, with frame-attributes being constantly updated by rules being fired, but with local provability computations being initiated by the backward chaining engine as it tries to establish which rules can be triggered.

Rules and Relations

In order to distinguish between rules which are intended to be used in a forward chaining manner from those which are to be used in a backward chaining manner, *flex* uses different rule formats and different KSL keywords.

Forward chaining rules are indicated by the keyword **`rule`**; backward chaining rules are indicated by the keyword **`relation`**.

Both formats fit the classical *if-then* style, but whereas backward chaining relations allow only for a single, positive conclusion in the *then* part, there is no such restriction in forward chaining rules which may have multiple

conclusions, any of which may be either positive or negative in nature.

Forward chaining rules often contain an action as part of their conclusion. This action usually updates various data (slot) values, which means that different rules will or will not now fire next time round the forward chaining cycle.

Backward chaining, on the other hand, generally seeks to establish a logical sequence of rules and facts in order to prove a clause or goal. This does not involve rule firing or executing actions, but is of a more deductive nature.

Weighting of Rules

In rule-based systems there are always choice points where one rule is preferred to another. Attaching weights to rules is an option which can assist in making those decisions.

The weight of a rule reflects its relative importance with respect to the other rules in the system. Whenever two or more rules are simultaneously applicable, their relative weights can be compared to decide which one to use.

Most weighting systems will be static, with each rule being assigned a specific score. The more important the rule, the higher its score should be. So, in the example below, if we have plenty of beer in the fridge, and the weather is hot, and it's late, then we will always drink beer regardless of the order of the rules.

Example

```
rule drink_tea
   if the hour is late
   then drink_a_cup_of_tea
   score 5 .

rule drink_beer
   if the fridge includes some beer
   and  the weather is hot
   then drink_a_can_of_beer
   score 10 .
```

In addition, *flex* allows for dynamic weighting systems whereby the score attached to a rule is not fixed when the rule is defined, but is dependent upon some changing information.

For example, suppose we have a rule-based system for controlling the flow of materials between different storage vessels in a chemical plant, which contains a rule `empty_master_into_slave`. It is not easy to give an exact score for this rule which is accurate in all circumstances. At certain times this may be the most appropriate rule, but at other times there may be another rule, say emptying the master vessel completely, that is more important. To reflect this dependence upon the current circumstances, we

can give the rule a dynamic score.

```
rule empty_master_into_slave
   if the master is not empty
   and  the slave`s contents
                        > the master`s spare_capacity
   then fill_from( master, slave )
   score master`s contents plus slave`s contents .
```

As the contents of the two vessels change, so the weighting of this rule also changes. We could even give the system a sense of chance by using random-valued scores.

Attaching Explanations to Rules

The final part of a production rule in *flex* involves the attachment of an optional explanation to rules. This is used to explain why a rule was fired.

The explanation itself can either be some canned text or a pointer to a disk file accessed at run-time through a *file browser*.

When explaining the rule either the canned text is displayed on screen or the user is allowed to browse through the file.

```
rule 'check status'
   if the applicant`s job is officer
   and the applicant`s age is greater than 65
   then ask_for_grading
   because The job grading affects the pension .
```

The above rule will cause the single line of text `The job grading affects the pension` to be displayed on the screen whenever an explanation is requested.

The latter alternative allows for a more flexible explanation facility. Indeed, the only purpose of the rule-based system might be as an intelligent link into the file browsing system. Such applications include the reviewing of technical manuals, especially medical journals.

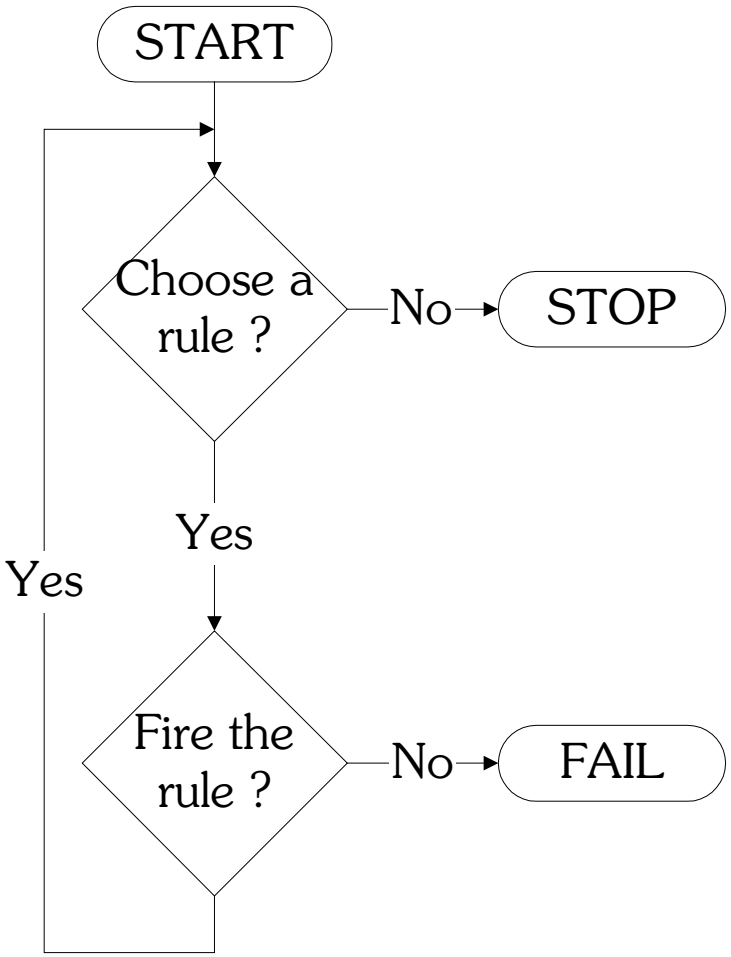
```
rule 'check status'
   if the applicant`s job is principal
   and the applicant`s age is greater than 65
   then check_principal_function
   browse file status .
```

The Forward Chaining Engine

The forward chaining engine is implemented as a Prolog program. The emphasis of the implementation is placed upon simplicity coupled with great flexibility, rather than extracting the highest possible performance.

A Simple Model

All production rule systems are based upon the same, very simple model. This is illustrated by the following flowchart.

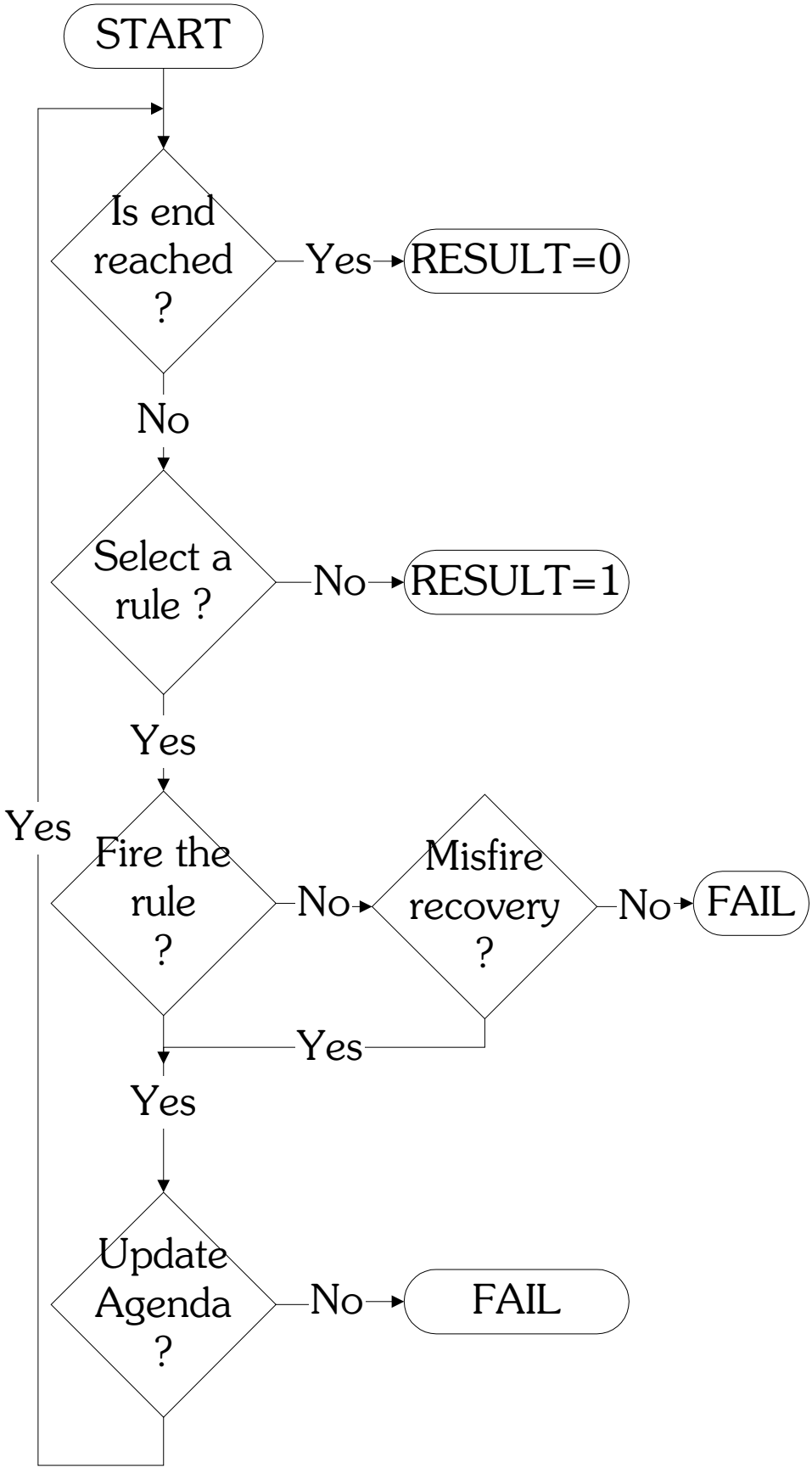


The basic cycle of all forward chaining engines involves the selection of a rule (checking the IF part), followed by the firing of that rule (executing the THEN part).

When implemented as a Prolog program, the individual parts of the engine can either succeed or fail, as indicated by the alternative yes and no branches.

The Implemented Model

Forward chaining in *flex* is an elaboration of the simple model given above. The implemented model is closer to the following flowchart:


START
Is end reached ?
Yes
RESULT=0
No
Select a rule ?
No
RESULT=1
Yes
Yes
Fire the rule ?
No
Misfire recovery ?
No
FAIL
Yes
Yes
Update Agenda ?
No
FAIL

The basic cycle of chosing a rule and then firing that rule remains, but with some elaborations.

1. In the simple model, the engine fires the first rule whose **if** part is satisfiable. A more sophisticated approach is to go through all the currently available rules, and check which *ones* are satisfiable. This forms a *conflict set*. Some algorithm is then used to *select* a rule to fire from this conflict set (this is called *conflict resolution*).

2. In the simple model, the engine terminates only when no more rules can be selected. In the extended model the user can specify an alternative criterion (a Prolog call) for *early termination*. The engine will terminate as soon as the call succeeds, regardless of whether there are more rules which could be fired.

3. The extended model also allows for the handling of the misfiring of rules. A rule is said to misfire if its conditions are satisfied but its actions fail for some reason. A recovery corresponds to the specified *misfire* program succeeding, in which case the cycle completes and the engine carries on. Otherwise an error condition is raised.

4. The final extension to the simple model concerns the rule agenda, from which rule selections are made during each and every cycle of the engine. This agenda can be *updated* at the end of each cycle according to which rule was fired during that cycle. For example, removing that rule from the agenda means that rules can only be fired at most once during that particular forward chaining session. Other possibilities include re-ordering the list of rules thus making it more likely that certain rules will be fired next time.

Ruleset

Rules are grouped together into *rulesets*, and forward chaining is started using the KSL keywords **invoke ruleset**. This provides a construct for forming rules into stratified rule-bases and governing the forward-chaining engine in terms of rule selection method and agenda updating.

For example, to move a piece in a game we might have a ruleset called `make_move` containing rules for possible moves.

```
ruleset make_move
   contains corner_move, edge_move, centre_move .

action move ;
   invoke ruleset make_move .
```

The Rule Agenda

The rule agenda determines the rules currently available to the inference engine during forward chaining (initially specified by a ruleset). It might include all of the rules in the system, or only a subset.

The rule agenda may contain duplicates, and there are no restrictions whatsoever on which names can go into the list.

After a rule has been selected and fired, the rule agenda may be updated, and it is this revised agenda that the inference engine uses as the basis for the next cycle.

Setting The Initial Rule Agenda

The initial rule agenda is the set of rules from which the forward chaining engine makes its first selection. The initial rule agenda may be specified as containing all rules, a list of rules or a rule group.

The only mandatory part of a given **ruleset** is the definition of the initial rule agenda.

```
ruleset everything
   contains all rules .
```

This KSL sentence sets the initial rule agenda to contain all the rules currently defined.

Selecting Rules

A vital part of any engine, whether forward or backward chaining, is the method by which rules are selected. *flex* provides three built-in methods for selecting rules:

First come first served

Conflict resolution

Conflict resolution with a threshold.

In addition, there is a facility for hooking in user-defined selection algorithms.

First Come First Served

The **`first come first served`** selection algorithm simply chooses the first rule in the agenda whose conditions (the **`if`** part) are all satisfied. This means that the *order* of rules in the agenda (actually represented as a list of rule names) is very important. The major benefit of this simple scheme is efficiency: firstly, during the cycle not all of the rules in the agenda are considered, and secondly, no stacks or heaps need be set up to store temporary sets of satisfiable rules.

For example, if the agenda contained 1000 rules we might strike lucky and find that rule 5 can be fired.

The major drawback is a lack of control when choosing rules, since the only control option available is to re-order the agenda. This, however, cannot always reflect the relative importance of rules.

First come first served is the default selection algorithm for the ruleset.

For example:

```
ruleset mover
   contains push, pull, lift .

ruleset make_tea
   contains all rules ;
   select rule using first come first served .
```

The selection algorithm used in both of these examples will be first come first served.

Conflict Resolution

Conflict resolution is a more sophisticated and computationally expensive selection scheme whereby the "best" rule is always selected. A conflict occurs when more than one rule can be fired (i.e. the **`if`** conditions of more than one rule are satisfied). This conflict is then resolved by choosing the rule with the highest score. In the event of a tie the first is chosen.

This scheme certainly allows far more control over the selection phase, but at a high cost. The "best" rule can only be chosen if every rule is considered.

For example, if the agenda contains 1000 rules then all of them need to be tested, even though the best rule may be the 5^{th} rule in the sequence. We do not know for certain that it is the best rule until the other 995 have been considered.

The following example assesses the scores of the rules whose conditions are satisfied and fires the highest found.

```
ruleset mover
   contains push, pull, lift
```

```
select rule using conflict resolution .
```

Conflict Resolution with Threshold Values

A compromise between these two contrasting selection schemes, (first come first served and conflict resolution), is to introduce *threshold values* into the latter. This offers a conflict resolution scheme in principle, but which stops as soon as a candidate is found whose score is greater than the threshold value.

For example, if the agenda contained 1000 rules we might strike lucky and find that the 5*th* rule can not only be fired, but also that its score is above the threshold value that has been set. Thus this rule will be fired with no further searching.

```
ruleset make_tea
   contains all rules ;
   select rule using conflict resolution
                with threshold 6 .
```

This example assesses the scores of the rules whose conditions are satisfied and fires the first rule found whose score is above 6.

In general, the higher the threshold value, the more rules that need to be considered when searching.

The two selection strategies, first come first served and full conflict resolution, can be seen as extreme cases of conflict resolution with threshold values: the first come first served algorithm represents a threshold value of minus infinity, and full conflict resolution represents a threshold value of plus infinity.

Updating the Agenda

At the end of each cycle of the engine, the agenda can be updated according to the name of the rule which was fired. The default is to leave the set of rules exactly as it is, so that the rules are always considered in the same order every time.

There are various built-in procedures for updating the agenda; alternatively the *flex* programmer can provide a user-defined algorithm.

The options for updating the ruleset are as follows.

Each time a rule is fired it can be removed from the ruleset. This effectively means that each rule is fired just once.

Each time a rule is fired it can be moved to the top or the bottom of the current ruleset. Moving it to the top means it will be the first rule to be considered next time round, or moving it to the bottom means it will be the last rule to be

considered.

The rule agenda may be viewed as a cyclic queue, whereby the rule *following* the selected rule is used as the top of the agenda for the next cycle.

The ruleset may be updated by removing from the rule agenda any rules whose conditions were not satisfied in the last cycle of the forward chaining engine.

Finally a rule "network" may be specified enabling the agenda to switch between different groups of rules for each cycle. Each rule group may be defined using the KSL `group` construct, and after each rule is fired, the rule agenda becomes the set of rules specified by the `group` with the same name as the rule which was just fired.

In addition to these built-in update methods you may define your own.

It is also possible to disable and enable certain rules. See the built-in *flex* predicates `disable_rules/1 and enable_rules/1`.

4. Data-Driven Programming

The frame system of *flex* can be used for the representation of data and knowledge. In this chapter we describe data-driven programming, where rather than program the control-flow or logic of a system, procedures are attached to individual frames or groups of frames.

These procedures lie dormant and are activated whenever there is a request to update, access or create an instance of the frame or slot to which they are attached. This concept of attached procedures, sometimes referred to as *active values*, is also found in object-oriented programming.

Data-Driven Procedures

There are four types of data-driven procedures:

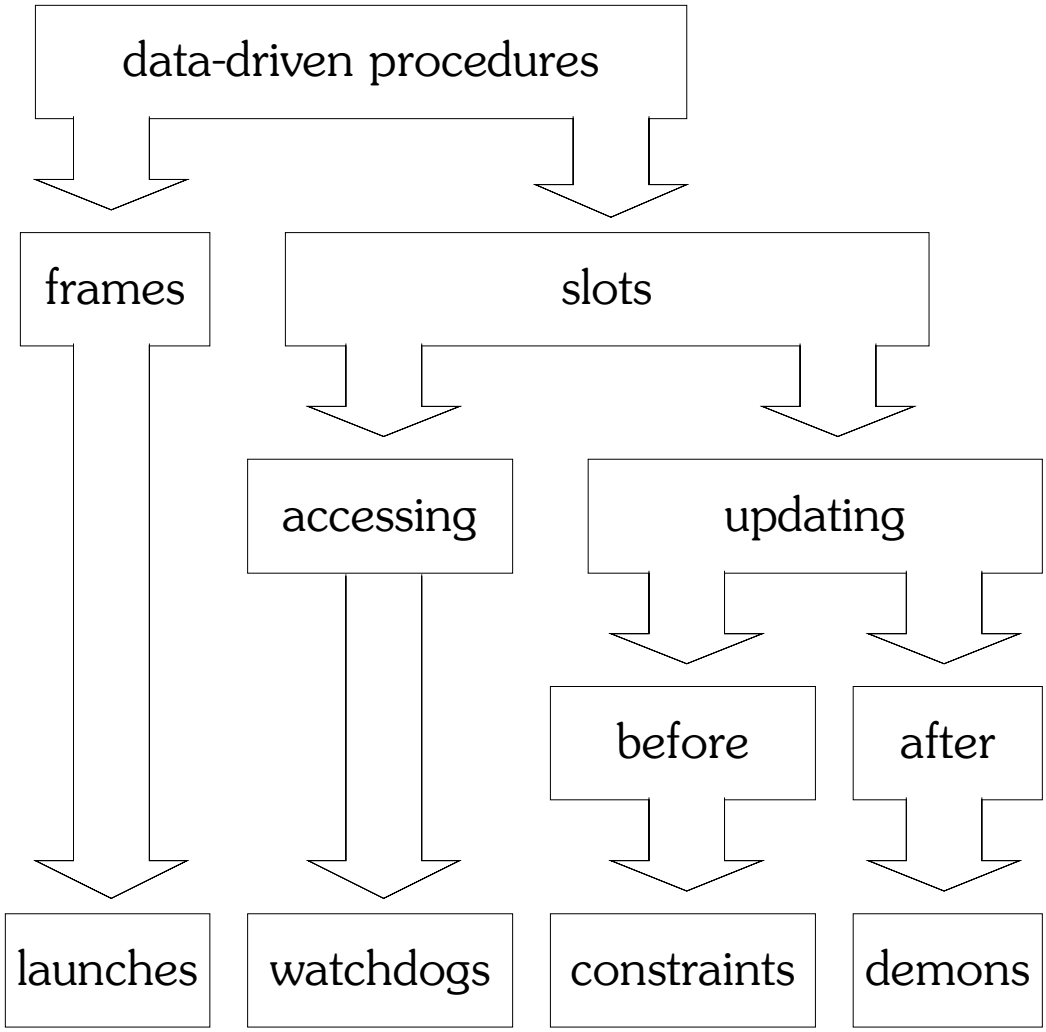
Launches

Watchdogs

Constraints

Demons

The following diagram shows when and where the various procedures are invoked.



Launches

A **launch** procedure is activated whenever there is a request to create an instance of the frame to which it is attached.

A **launch** procedure has three main parts:

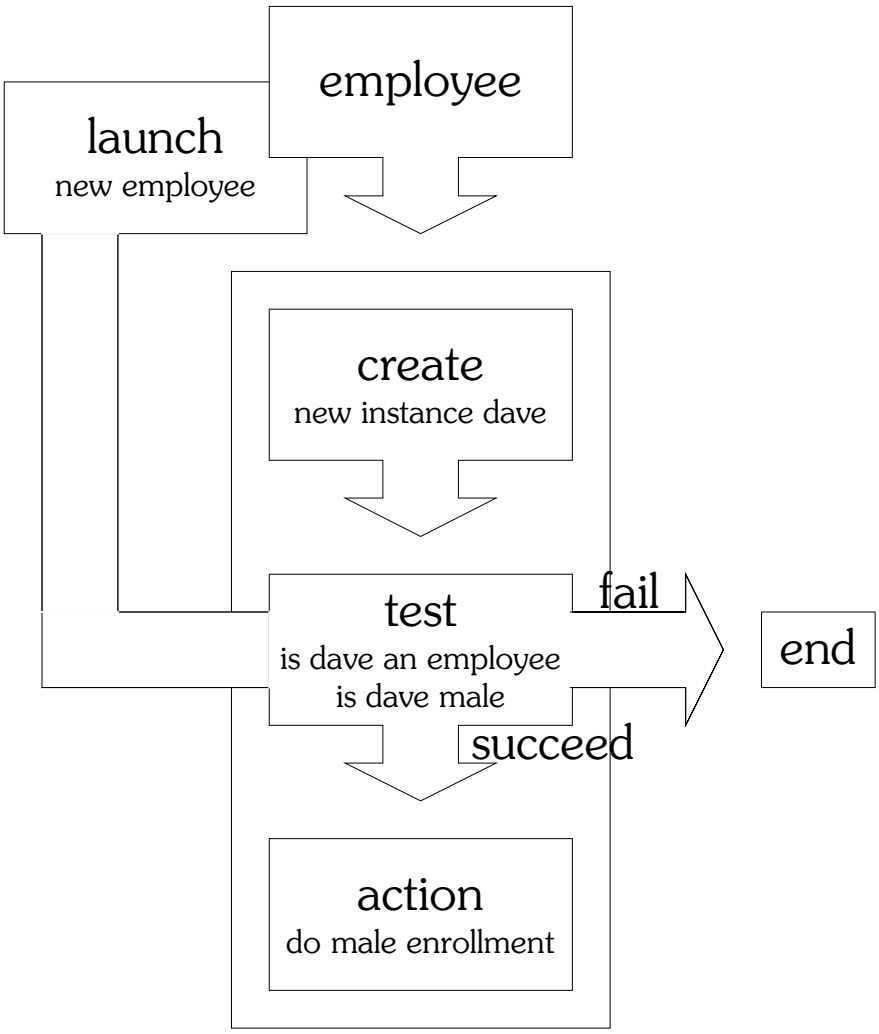
- context A test to see whether certain conditions hold.
- test A test to see whether certain conditions hold.
- action A series of commands to be performed.

The action will only be invoked if the test succeeds. The launch is invoked *after* the instance has been created.

Example

Suppose we have a frame system representing a company's personnel structure, and that a new employee `dave` is to be added.

The launch will automatically be invoked whenever a new instance of `employee` is requested, and provided the conditions hold, the actions will be performed.



The launch procedure in this example, attached to the `employee` frame, is set up to collect the personal details about new male employees.

The KSL code for this example would be written as follows:

```
frame employee
   default sex is male .

launch new_employee
   when Person is a new employee
   and  sex of Person is male
   then male_enrolment_questions(Person) .

instance dave is an employee .
```

The `male_enrolment_questions` will be defined elsewhere.

Constraining the Values of Slots

A **`constraint`** is attached to an individual slot, and is designed to constrain the contents of a slot to valid values. It is activated whenever the *current* value of that slot is updated, and the activation occurs immediately *before* the update.

A **`constraint`** has three main parts:

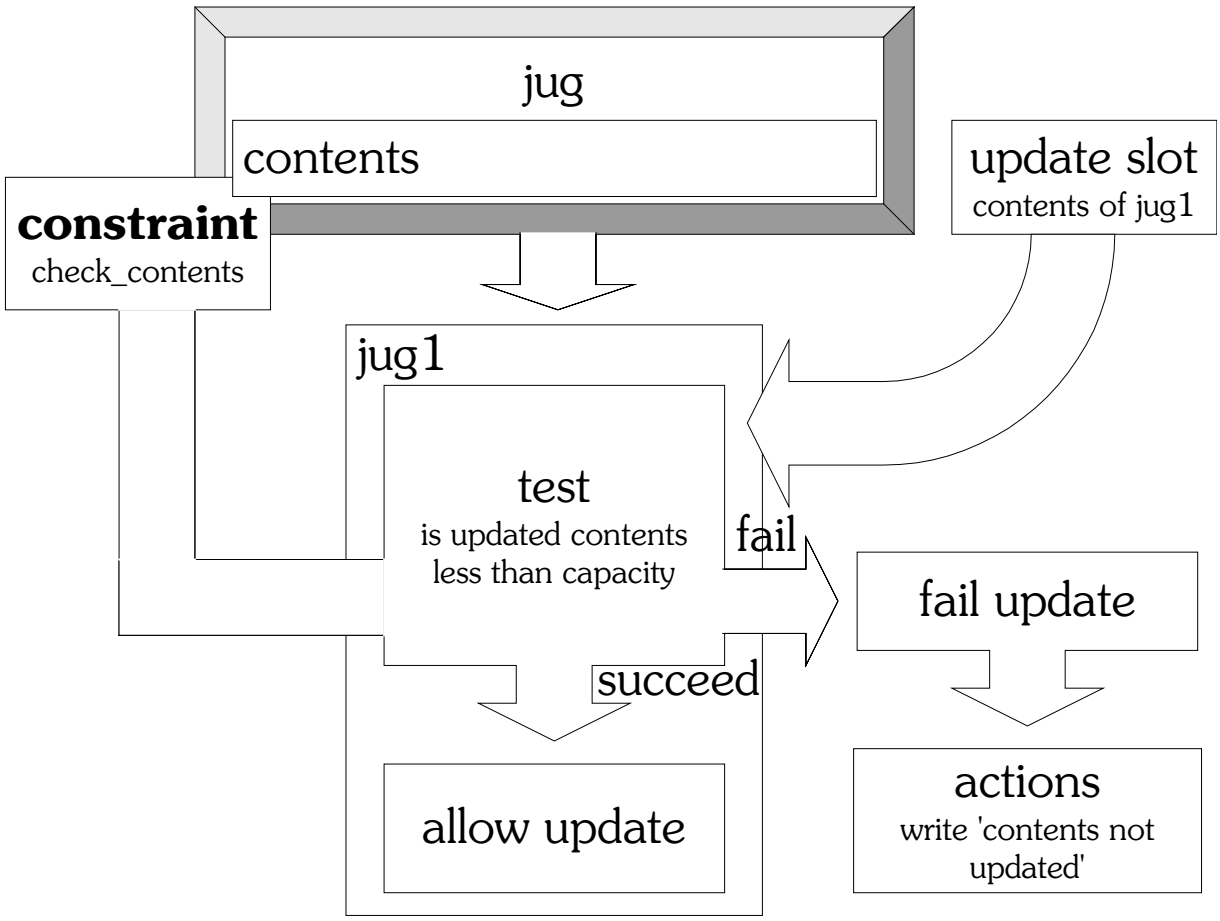
- context A test to see whether certain conditions hold.
- check A test to see whether the update is valid.
- error A series of commands to be performed for an invalid update.

The check will only be made if the context holds. If the check is successful then the update is allowed, otherwise the error commands are invoked and the update is not allowed.

Example

If we had a frame system representing instances of water containers, we could put a constraint on the `contents` slot of any `jug`, such that when the value for the particular slot is being updated, a test is performed making sure that the new value is less than the value of the `jug`s capacity`, thus ensuring the jug does not overflow!

In the example the constraint is activated if the `contents` attribute of any jug changes. The prospective new value for the slot is then tested to see if it is less than that jug's capacity. If the test succeeds the update is allowed. If the test fails the update is not allowed and a message is displayed.



The code for this example could be written as follows:

```
frame jug
   default contents are 0 and
   default capacity is 7 .

instance jug1 is a jug .

constraint check_contents
   when the contents of Jug changes to X
   and Jug is some jug
   then check that number( X )
   and  X =< Jug`s capacity
   otherwise write( 'contents not updated' )
   and nl .
```

Note the use of `Jug`, a local variable, which will unify with any frame or instance which has, in this case, a `contents` attribute.

Attaching Demons to Slot Updates

A **demon** is attached to an individual slot. It is activated whenever the *current* value of that slot is updated, and the activation occurs immediately *after* the update.

A **demon** has two main parts:

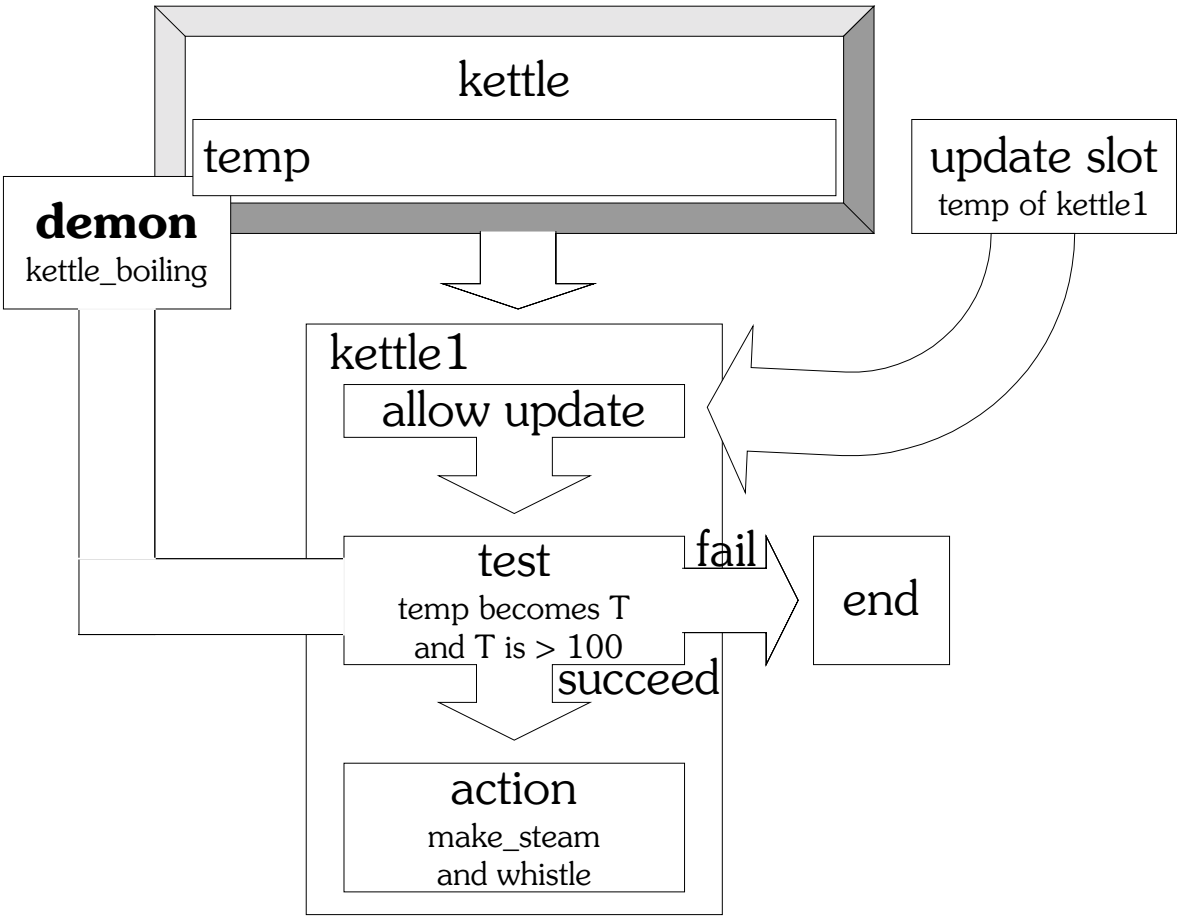
- context A test to see whether certain conditions hold.
- action A series of commands to be performed.

The slot is updated and then, given that the context holds, the actions will be invoked.

A **demon** can be tailored such that it fires only for given values and/or only under certain circumstances.

Example

If we are modelling the action of kettles, we could attach a demon to the `temp` slot of any instance of the frame `kettle`.



Whenever the `temp` slot is updated, a check will be made on the new value, such that, if it is greater than `100`, then the actions `make_steam` and `whistle` are performed.

The code for this example could be written as follows:

```
frame kettle
   default temp is 0 .

instance kettle1 is a kettle .
```

```
demon kettle_boiling
   when the temp changes to T
   and T is greater than 100
   then make_steam
   and  nl
   and  whistle .
```

Restricting the Access to Slots

A **watchdog** is attached to an individual slot. It is activated whenever the *current* value of the slot is accessed.

A watchdog has three main parts:

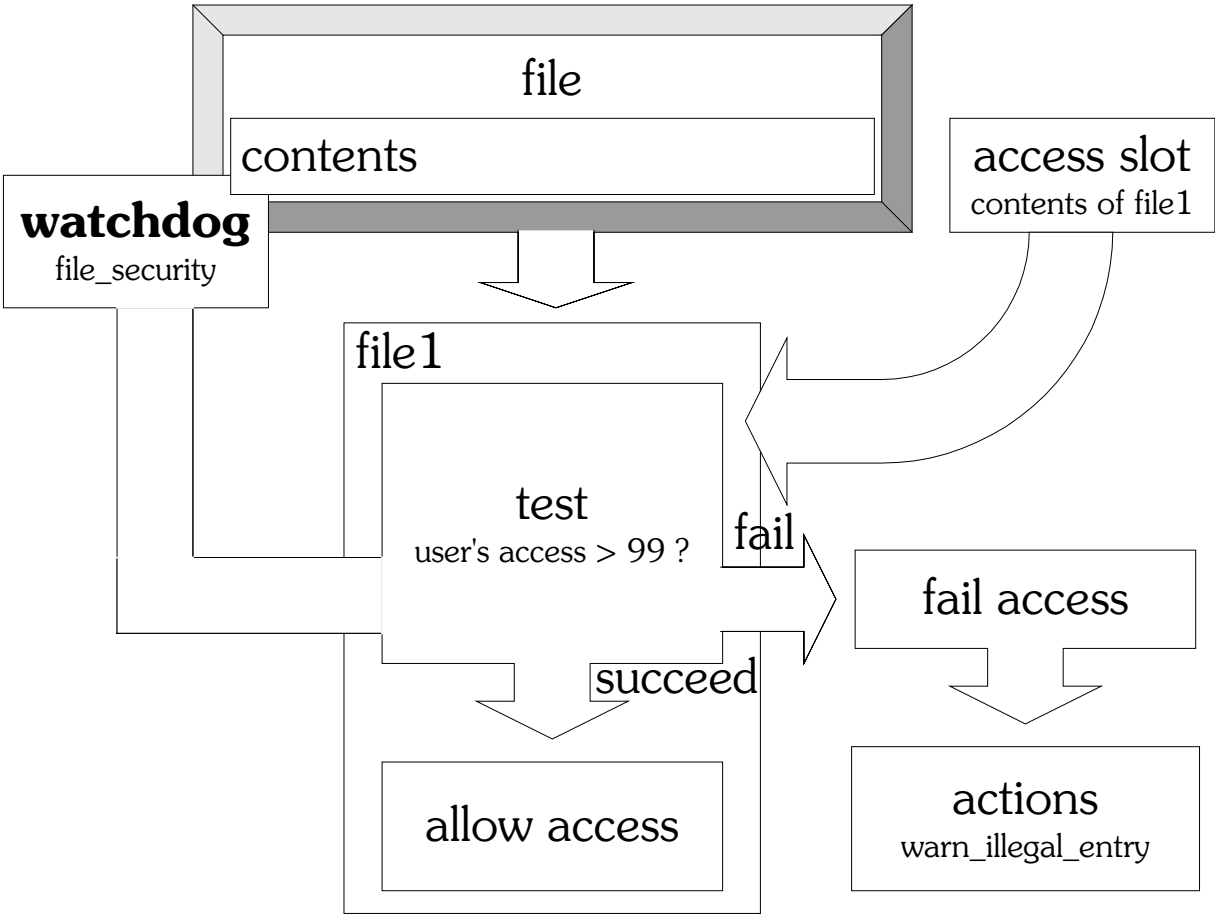
- context A test to see whether certain conditions hold.
- check A test to see whether the access is valid.
- error A series of commands to be performed if access is invalid.

The check will only be made if the context holds. If the check is successful then the access is allowed. Otherwise, the error commands are invoked and the access is denied.

A watchdog can be used to check the access rights to an attribute of a frame. It is invoked whenever there is a request for the *current* value (*not* the default value) of that slot (attribute-frame pair).

Example

In the example shown below, the watchdog is activated when the contents of a file are requested. A check on the user's classification is then made, and if the check succeeds the access is allowed. If the check fails the access is denied and a warning message is displayed.



The KSL for a similar example is shown below. In this case, only users with sufficiently high access priority may find out the balance in a bank account. The `current_user` frame stores the access code of the current user, which is checked in the `account_security` watchdog.

```
frame bank_account
  default balance is 0.

frame current_user
  default name is '' and
  default access is 0.

watchdog account_security
  when the balance of Account is requested
  and Account is some bank_account
  then check current_user`s access is above 99
  otherwise write( 'Balance access denied to user ' )
  and
  write( current_user`s name ).
```

5. Questions and Answers

Most expert system applications will involve some communication with the user. In *flex*, this is achieved by invoking pre-defined questions. These questions may involve making a selection from a menu, typing information at a keyboard, or indeed any set of operations which require a reaction by the user.

Defining Questions

Questions are defined within a KSL program by use of the keyword **`question`**. The main part of each question definition states how to obtain an answer, and what form it should take. *Flex* provides certain built-in constructs for obtaining answers, including single and multiple choice menus and data input screens. More sophisticated user interactions can be defined using the GUI facilities of the underlying Prolog system and can then easily integrated with the *flex* question and answer sub-system using relations or actions. Once a question is defined, it may be invoked using the KSL directives **`ask`** or **`answer`**, described later.

Menu Selection

In a menu selection format question, the user is presented with a collection of options, and is offered the choice of making either a single selection or multiple selections. The attraction of this is that the values obtained are implicitly validated, since they come from a fixed set of alternatives which have either been hard-wired into the question or programmatically generated.

Consider the problem of putting together a menu for some meal. The user is allowed to choose various combinations of dishes to make up that meal. For the main course, there is a straightforward choice between various meat dishes and fish dishes, only *one* of which can be selected.

The KSL for describing this question would be as follows.

```
question main_course
   Please select a dish for your main course ;
   choose one of steak,'lamb chops',trout,'dover sole'
.
```

Whenever the question `main_course` is asked, the user is presented with a menu containing the items `steak`, `'lamb chops'`, `trout` and `'dover sole'`. One, and only one, item can be selected from this menu.

Rather than state the items explicitly within the definition of the question, we can collect the items together and store them within a **group**. We can then refer to this name within the question.

```
group main_courses
   steak, 'lamb chops', trout, 'dover sole' .

question main_course
   Please select a dish for your main course ;
   choose one of main_courses .
```

Now, to accompany the main course there are various vegetables which can be selected in any configuration. Using the KSL keywords **choose some of** we can allow the user to select any number of vegetables.

```
question vegetables
   Please select vegetables to accompany your main
course ;
   choose some of potatoes, leeks, carrots, peas .
```

Storing Answers

Whenever a question is invoked, the "answer" to the question is stored as the value of a global variable of the same name as the question. So, in the above two examples, the answers to the questions are stored in the global variables named `main_course` and `vegetables` respectively.

To test the above two questions we may define an action as follows. (Here we use of the KSL construct **check that** to "retrieve" the value of a global variable.)

```
action get_main_course( Main, Vegs ) ;
   do ask main_course and
   ask vegetables and
   check that Main is main_course and
   check that Vegs is vegetables .
```

Then `get_main_course/2` may be called from a Prolog program, or a *flex* procedure, or run from the Prolog command line, e.g.

```
?- get_main_course( M, V ).
M = steak, Vegs = [leeks, carrots, peas]
```

Alternatively we may define an action which simply prints out the values.

```
action show_main_course ;
   do ask main_course and
   ask vegetables and
   write( main_course ) and
   nl and
```

```
  write( vegetables ) .


?- show_main_course .
steak
[leeks, carrots, peas]
```

Keyboard Input

The second pre-defined question mechanism is through single field keyboard **input**. The data entered can easily be constrained to be either a text item (name), a floating-point number, an integer, or a set of such items.

Examples

```
question name_of_applicant
  Please enter your name ;
  input name .

question height_of_applicant
  Please enter your height (in metres) ;
  input number .

question address1_of_applicant
  Please enter your house number ;
  input integer .

question address5_of_applicant
  Please enter your city and post code ;
  input set .
```

Constrained Input

You can constrain the standard keyboard input to be something other than a name, number, integer or set of such objects, by nominating a Prolog program or *flex* relation to be used to validate the answer. This is indicated by the keywords **such that**.

```
question yes_or_no
  Please answer yes or no ;
  input K such that yes_or_no_answer( K ) .

relation yes_or_no_answer( yes ) .
relation yes_or_no_answer( no ) .
```

This will present a standard dialog (which will depend on your implementation of *flex*), and the user's response must satisfy the `yes_or_no_answer` relationship.

Customized Input

The range of standard questions provided will inevitably not cover all possible situations. For this reason, *flex* allows customized questions in which the programmer can specify both how to obtain an answer, and what form that answer should take. The onus is totally on the programmer to present the question to the user (for example create a dialog) and to return the appropriate answer. This is indicated by the KSL keyword **answer**.

```
question my_question
   answer is K such that ask_my_question( K ) .
```

In this case no predefined dialog will be presented, but a call will be made to `ask_my_question/1`: this may be defined as a *flex* action, a *flex* relation or as a Prolog predicate. It should ask the question, creating any necessary dialogs, and return a value for the variable K.

Default Questions

When developing an application, it is often useful to delay the exact implementation of questions until some later stage. During this development process, *flex* allows you to declare a default question which is used in the absence of a specific definition. The name of the default question is `catchall`.

```
question catchall
   Please enter data ;
   input name .
```

Whenever a question is asked for which there is no definition, the `catchall` definition is used instead. In this case, the default answer will be of type `name`.

Explaining Questions

In addition to the form of a question, you can optionally attach an explanation to any of the standard question types (as you can with rules) using a **because** clause.

The explanation itself can either be some canned text to be displayed, or it can be the name of a file to be browsed over. The explanations are presented whenever the end-user requests them (usually there is an *Explain* button in the built-in question dialogs).

```
question main_course
   Please select a dish for your main course ;
   choose from steak, 'lamb chops', trout, 'dover sole'
;
   because The main course is an integral part of a
meal .

question headache
   Have you got a headache ? ;
   answer is K such that yes_or_no_answer( K ) ;
   browse file medical_symptoms .
```

Whenever there is a request to explain the `headache` question, the user begins browsing the file `medical_symptoms` starting at the `headache` topic.

If explanations are attached to customized questions, then the onus is on the programmer to reflect any explanations to the end-user.

(Note that the following example will not work in MacProlog since there is no support for byte-level keyboard input.)

Example

```
question headache
   answer is K such that
        write( 'Have you got a headache ?' ) and
        yes_no_question( K, medical_symptoms ) .

action yes_no_question( K, File ) ;
   do write( 'Please type Y or N or ESC' )
   and  repeat
   and  get0( Byte )
   and  yes_or_no_check( Byte, K, File )
   and  ! .

relation yes_or_no_check(  89, yes, File ) .
relation yes_or_no_check( 121, yes, File ) .
relation yes_or_no_check(  78,  no, File ) .
relation yes_or_no_check( 110,  no, File ) .
relation yes_or_no_check(  26,   _, File )
   if browse( File )
   and  fail .
```

Invoking Questions

There are two underlying procedures, `ask/1` and `answer/2`, for invoking questions. These are reflected in the KSL as the directive

```
ask <name of question>
```

and as the term

```
answer to <name of question>
```

respectively. These built-in procedures behave quite differently at run-time. Whenever there is a request to **ask** a question, that question is always asked immediately. However, a request for the **answer to** a question will only invoke that question if it has not previously been asked.

If the behaviour of an application is such that the same question should be asked once, and only once, then use the **answer to** construct.

```
action decide_meal ;
   do ask main_course
   and ask vegetables .
```

In this case, using **ask**, the question dialogs will be displayed *every time* the `decide_meal` action is executed. The global values of `main_course` and `vegetables` will therefore change for each invocation of the `decide_meal` action.

```
rule prescription1
   if the answer to headache is yes
   and the answer to pregnant is no
   then prescribe( paracetamol ) .

rule prescription2
   if the answer to headache is yes
   and the answer to pregnant is yes
   then prescribe( nothing ) .
```

In this example, the actual questions `headache` and `pregnant` will only be asked if they haven't previously been asked. Once there is a value for each of `headache` and `pregnant`, this value will be used for the rest of the session. The questions `headache` and `pregnant` will therefore only be asked once.

6. The Anatomy of a *flex* Program

This chapter describes the basic composition of a *flex* program using the Knowledge Specification Language (KSL). The KSL's structures and syntax are described in more detail in the next chapter.

A *flex* program comprises a series of sentences written in the KSL (Knowledge Specification Language). Each sentence starts with a KSL keyword and ends with a full stop. These sentences are compiled into Prolog clauses by the *flex* compiler. (See the *Runtime Interpretation of KSL* chapter).

A KSL sentence begins with one of the following keywords:

```
action
constraint
data
demon
do
frame
function
group
instance
launch
question
relation
rule
ruleset
synonym
template
watchdog
```

These are all described in detail in the next chapter.

Note that a KSL program does not have to use forward chaining, and may consist entirely of **relation**s and **action**s, which are the equivalent of backward chaining Prolog programs. For example, the following Prolog program.

```
sibling( X, Y ):-
   parent( Z, X ),
   parent( Z, Y ).
```

may be written using the KSL as follows.

```
relation sibling( X, Y )
   if parent( Z, X )
   and parent( Z, Y ).
```

Either of these may be called from the Prolog command line, e.g.

```
?- sibling( harriet, S ) .
```

However, to use the *flex* forward chaining engine a minimal *flex* program contains at least one of each of the following:

frame
rule
ruleset
action

The **frame**s describe the data structures. The frames have *slots* (sometimes called *attributes*), which are like fields in a conventional record structure. The rules defined in a *flex* program manipulate the data contained in these frames.

A *flex* **rule** consists of a set of conditions and some actions to be performed if the conditions are satisfied. A rule is said to *fire* if its conditions are satisfied.

A *flex* **ruleset** declares the names of the rules to be used for the current problem.

The basic mechanism of the *flex* forward chaining engine is to go through the current set of rules, testing the conditions, until a rule is found whose conditions are satisfied, and its actions are then performed. This cycle repeats until no more rules can be fired, i.e. there are no rules whose conditions can be satisfied.

It is possible to specify other termination criteria, and to specify exactly which rules should be considered, in what order, and how they should be re-ordered for each cycle of the forward chaining engine. See the description of **ruleset** in the next chapter.

The *flex* forward chaining engine may be started by defining an **action**, which is similar to a Prolog program and may be run as a Prolog query.

A Simple *flex* Program

We may write a very simple *flex* program to sell cinema seats to viewers. This will demonstrate the essential components of a forward chaining *flex* program, and give a flavour of the KSL. (The constructs used here are explained in more detail in the next chapter.)

Note that because this is KSL code it should be compiled in a file or window with the extension **.KSL**. If you compile the KSL code contained in an 'untitled' window, it will be compiled as if it were Prolog code and errors will be generated.

The frames

First we define the data structures, which means declaring a `frame` for a cinema and a `frame` for a cinema-goer. Each will have a single slot, or attribute, which contains the number of seats for a cinema and the number of tickets required by a viewer.

```
frame cinema
   default seats is 500.

frame viewer
   default tickets_required is 3.
```

The rules

We will define one rule, which describes how the cinema seats will be allocated to the viewer. We will call this rule `allocate_tickets`. The condition under which the rule will be fired is simply that the viewer requires some tickets!

```
rule allocate_tickets
   if the tickets_required of viewer is greater than 0
   then the seats of the cinema becomes
      the seats of the cinema
         minus the tickets_required of viewer and
      the tickets_required of viewer becomes 0.
```

The ruleset

Next, a `ruleset` must be defined to say what rules are to be considered. We will call our ruleset `seating`; in this case we only have one rule, called `allocate_tickets`.

```
ruleset seating
   contains allocate_tickets.
```

The action

Finally, to set this going we *invoke* these rules, defining an **`action`** to do so.

```
action go ;
   do invoke ruleset seating.
```

Starting The Forward Chaining Engine

We now have a complete *flex* program. To start the *flex* forward chaining engine, we simply run the Prolog query `go` (type `go` at the Prolog command line).

```
?- go.
```

This should succeed, with the given rule firing once only (because after that the viewer's tickets required will be zero).

Displaying Results

Unfortunately, at the moment we have no way of knowing if it ran correctly, i.e. if 3 seats were subtracted from the cinema's total seats. We will add another `action` to simply write out some relevant values, called `write_values`.

Here we will use the KSL operator **`s** instead of **of** for accessing the slots of frames.

```
action write_values ;
   write( 'Cinema seats: ' ) and
   write( cinema`s seats ) and
   tab( 2 ) and
   write( 'Viewer tickets required: ' ) and
   write( viewer`s tickets_required ) and
   nl.
```

Note the use of the built-in Prolog predicates `write/1`, `tab/1` and `nl/0` to write text, spaces and a new line. Any Prolog predicate may be called in this way from *flex*; its arguments will be dereferenced by the *flex* interpreter before executing (so that, for example, the term `viewer`s tickets_required` will be dereferenced to the current value of the slot `tickets_required` of the frame `viewer`).

By writing out the values before and after running *flex*, we may see that the operation has been done correctly.

```
?- restart, write_values, go, write_values.
Cinema seats: 500  Viewer tickets required: 3
Cinema seats: 497  Viewer tickets required: 0
```

Note that `restart/0` is a built-in *flex* predicate which resets slot values back to their original values.

Extending the Program

In reality there would be more than one cinema and more than one viewer, (so we would probably uses `instance`s), and the forward chaining engine would continue until there were no more `viewer`s with `tickets_required` values greater than 0. There would also need to be a check that the number of tickets required was less than the number of seats in the cinema.

```
rule allocate_tickets
   if the viewer`s tickets_required is greater than 0
   and the cinema`s seats is greater than or equal to
      the viewer`s tickets_required
 then the seats of the cinema becomes
      the seats of the cinema
```

```
      minus the tickets_required of viewer and
    the tickets_required of viewer becomes 0 .
```

A second rule could be added to inform the viewer that no seats were available.

```
rule refuse_tickets
   if the viewer`s tickets_required is greater than 0
   and the cinema`s seats is less than
      the viewer`s tickets_required
   then write( 'Sorry - no seats left' ) and nl and
      the tickets_required of viewer becomes 0 .
```

It would also be better if there was some user interaction so that at runtime we could ask how many tickets the viewer wanted - for this we could use the *flex* **question** construct.

flex and Prolog

flex is built on top of Prolog and all the functionality of Prolog is also available to the *flex* programmer. Any Prolog predicate (either built-in or user-defined) may be called from within *flex*, and its arguments will be dereferenced before being called. Conversely, any `action` or `relation` defined in the KSL may be called from Prolog as if it were a Prolog program.

Applications may therefore be easily written as a mix of Prolog and *flex*.

However, we also provide full access to the *flex* system from Prolog alone: you do not have to use the KSL at all. You may write *flex* programs entirely in Prolog, using the *flex* predicates listed in the *Flex Predicates* chapter of this manual. For each KSL sentence there is an equivalent set of Prolog predicates. Alternatively you may write part of your code using the Prolog *flex* predicates provided, and write part of your code using the KSL. Prolog and KSL code may be freely intermixed.

Note that you will normally have to compile KSL and Prolog code separately; KSL code should be stored in a file or window with the extension **.KSL**. If you compile the KSL code contained in an 'untitled' window, it will be compiled as if it were Prolog code and errors will be generated.

Components of the KSL

The KSL contains *terms*, *formulae*, and *sentences*.

The *terms* describe the objects in the world being defined.

The *formulae* are used to describe the relationships between different objects of the KSL.

The *sentences* of the KSL are valid statements which relate the formulae and terms.

A KSL program comprises a series of sentences.

7. The KSL

This chapter describes all the elements of the KSL. See also the chapter on *flex Predicates* for the equivalent Prolog predicates, and the chapter on the *Run-Time Interpretation of KSL* for a description of how the KSL is compiled into Prolog.

KSL Terms

This section describes the valid tokens and terms of the KSL. Essentially they follow the tokenisation of Prolog's Edinburgh syntax, with a few enhancements.

There are five types of Edinburgh token: *punctuation*, *number*, *atom*, *string* and *variable*. Built on top of Edinburgh tokens are the concepts of a KSL *name* and a KSL *value*.

Comments

Any text between the symbols `/*` and `*/` is treated as a comment and is ignored by the *flex* compiler.

The `%` symbol indicates the start of a comment up to the end of the line on which it occurs.

Punctuation

A *punctuation* mark is always considered as a separate token (unless it lies within quotes), and is one of the following:

```
(   )   [   ]   {   }   |   !   ;   ,
```

Numbers

Numbers are either integers or floating-point numbers.

Examples

```
   211327    -32768       0    2.34      10.3e99      -
0.81
```

Atoms

Atoms are of three types: *alphanumeric*, *symbolic* and *quoted*.

An *alphanumeric* atom is a lowercase letter (a-z) followed by a sequence of zero or more alphabetic characters (a-z, A-Z or _) or digits (0-9).

Examples

```
apple        aPPLE        h45j        apple_cart
```

A *symbolic* atom is a contiguous sequence of symbols such as *, >, or #. The symbolic characters are all those characters other than digits (0-9), alphabetic (a-z, A-Z and _) and punctuation marks.

Examples

```
&            >=          #@&           **/
```

A *quoted* atom is any sequence of characters delimited by single quotes (use two quote characters to denote a single quote within a quoted atom).

Examples

```
'Apple'      'The green ***'        '^h''ht'
```

Byte Lists

A *Prolog byte list* is any sequence of characters delimited by double quotes. It is an abbreviated notation for the list of decimal ASCII codes of the characters in the sequence.

Examples

`"A boy"`	is shorthand for	`[65,32,98,111,121]`
`""`	is shorthand for	`[]`

Variables

A *variable* is an uppercase letter (A-Z) or an underscore (_), followed by a sequence of zero or more alphabetic characters (a-z, A-Z or _) or digits (0-9). An underscore on its own is an anonymous (don't care) variable.

Examples

```
X       Apple        X_Apple _23
```

Names

A *name* is any *quoted* atom, or any *atom* which is not a reserved word (i.e. an atom which does not appear in the glossary of the KSL).

Examples

```
brick     brick32     'The'     'the brick'
```

Values

A *value* is any *number*, any *string* or any *name*.

Examples

```
9821    -0.81     "the empty string follows"    ""
   'the'
```

KSL Objects

This section describes the actual objects of KSL (i.e. constructs which correspond to entities in your particular domain). They range from variants which can change in value through time (by assignment) to set abstractions which portray a collection of objects.

Variants

A *variant* can change in value over time (i.e. like a variable). There are essentially two forms of *variant*: a straightforward global variable, or a slot. Both of these have values associated with them.

A *global variable* is any name whatsoever, optionally be prefixed by the determiner **the**.

Examples

```
staff
the staff
'today''s temperature'
```

A simple *slot* is specified by the name of an attribute and the name of the frame (or frame instance) to which the attribute belongs, using the KSL keyword **of** or the operator **`s**. (Note the use of the backwards quote character `.) The attribute or frame may be optionally prefixed by the determiner **the**. The general formats of slots are as follows.

```
frame`s attribute
attribute of frame
the attribute of the frame
```

Examples

```
size of the collar
the collar`s size
the colour of money
money`s colour
```

Variant Schema

The variant schema is a generalisation of slots by which the name of the frame is only indirectly referred to. That is, the variant is a specific attribute of some under-specified frame, rather than any particular frame.

The first kind of variant schema does not mention any particular frame at all. These are *don't care* constructs, in which the name of the frame is irrelevant, only that one exists. The general format is as follows.

```
attribute of anything
attribute of something
```

```
attribute of anybody
attribute of somebody
```

Examples

```
the colour of anything
a product of something
anybody`s address
```

The second type of variant schema is similar in that the name of the actual frame is irrelevant, but the class is specified - the slot must belong to a frame or instance of the class. The general format is as follows.

```
attribute of some frame
attribute of any frame
attribute of some instance of frame
attribute of any instance of frame
```

Examples

```
the skin of any reptile
the capacity of some instance of cinema
some brick`s shape
any fruit`s taste
```

Complex Variants

The previous two sections described variants referring to an attribute of a frame. This concept may be extended with arbitrary levels of indirection, also referred to as *attribute-chaining*.

The value assigned to an attribute of a frame can itself be the name of another attribute or frame. For example, the `employee_address` attribute of the `employee` frame can have the value `address`, which is the name of another frame. We can then refer to other attributes of the `address` frame.

Examples

```
the city of ( the employee_address of the employee )
city of employee_address of some employee
some employee`s employee_address`s city
( a1 of frame1 ) of ( ( a2 of frame2 ) of frame3 )
```

where the value of `a1` of `frame1` and the value of `a2` of `frame2` are the names of other attributes, say `a11` and `a22`. Then, the value of `a22` of `frame3` must be the name of another frame, say `frame33`. Finally, the term reduces to the value of `a11` of `frame33`.

Sets

There are two forms of sets in KSL; an *explicit* collection of objects or a set abstraction which *implicitly* defines the collection.

An *explicit* set is a description of each and every individual element in the set. Set elements may be separated by a comma or the KSL keywords **or** or **and**.

Examples

```
{ fred, angela, john and mary }
{ the staff or fred }
{ the staff or fred and mary }
{ 10, 20, 30, 40, 50 }
```

NOTE An explicit set in *flex* corresponds to a list in Prolog. The KSL uses curly brackets **{}** for sets, whereas Prolog uses square brackets **[]**.

An *implicit* set states how to find or compute each individual element without necessarily mentioning each one.

Examples

```
{ X such that generate_number(X)
             and X >= 0 and X < 10 }
{ X : bird(X) or reptile(X) and can_fly(X) }
```

Alternatively, an implicit set can be all the members of a particular group, specified by the keywords **all**, **every**, or **each**.

Examples

```
all colours
every dessert
each flower
```

And finally, an implicit set can be a collection of frame instances which have particular attributes. The keyword **whose** may be used to refer to attributes of a frame.

Examples

```
every instance of animal
every reptile
   whose skin is smooth
   and whose ears are not large
   and whose legs > 2
   and whose habitat includes { river or ocean }
   and whose skin is included in { fur, scale } .
```

General Terms

General terms include names, values, variables, variants, variant schemas and sets. In addition, there are some special constructs described here.

A *compound* term is, like in Prolog, the name of the principal functor followed by a sequence of arguments within parentheses.

Examples

```
factorial( 45 )
foo( alpha , beta , gamma )
foo( the staff , { john and mary } )
```

A *conditional* term consists of a test and two alternative sub-terms. Which sub-term is chosen depends upon the success or failure of the test. The general format is as follows.

if *condition(s)* **then** *expression1* **else** *expression2*

Examples

```
if positive(X) then 1 else 0
if test1 or test2 and test3 then volume else density
```

Another kind of conditional term refers to instances of some formulae.

Examples

```
X such that number(X) and prime(X)
X : X is -1 or X is 0 or X is 1
```

A term can refer to an *individual* member of a group, without stating explicitly who that member is, using **some**, **any**, **some instance of** or **any instance of**.

Examples

```
any colour
some dessert
any instance of jug
```

A term can refer to an *instance* of a frame which has certain attributes. Again, such terms do not explicitly state which instances. Once again the keyword **whose** may be used to refer to attributes of a frame.

Examples

```
some instance of animal
some reptile
   whose skin is smooth
   and whose ears are not large
   and whose legs > 2
   and whose skin is included in { fur, scale } .
```

A term can *self-reference* a particular attribute without naming the frame (or frame instance) involved. This construct only has a meaning when used as part of the definition of a slot (i.e. frame-attribute pair). KSL keywords which may be used are **whose** (as in the above example), **its** or **their**.

Examples

```
its density
their colours
some box whose width is less than its depth
```

A term can refer to the *answer* to a particular question which may have already been asked. If not, the question is asked when the answer is first requested.

Examples

```
the answer to entree
the answer to dessert
```

Here `entree` and `dessert` will have been defined as **question** constructs.

There are three special names for denoting the *empty* set:

```
empty
nobody
nothing
```

There are four special *anonymous* names to be used when you do not need to state who or what the individual is.

```
somebody
something
anybody
anything
```

Arithmetic Expressions

Arithmetic expressions are formed by binary operators *****, **+**, **-**, **/** and **^** and unary operator **-** over terms. The words **plus**, **minus**, **times**, **divided by** and **to the power of** may also be used.

The usual operator precedence apply, with **^** more binding than ***** and **/** which are in turn more binding than **+** and **-**.

Operator associativity is to the left whenever more than one operator of the same precedence appears in the expression.

Examples

```
1 + 2 * 3 - 4 / 5 ^ 6
the number of managers plus the number of secretaries
X`s mileage times 30
the contents of jug1 minus the contents of jug2
the box`s size to the power of 3
```

The precedence of operators can be overridden by enclosing the expression within brackets.

Examples

```
( 1 - 2 ) * ( 3 + 4 )
( its temp times its volume ) to the power of 2
```

Dereferencing

Unlike Prolog, expressions in *flex* are automatically dereferenced and evaluated when required. For example, any reference to the name of a slot or a global variable will automatically cause that slot or variable's current value to be accessed and used.

This dereferencing can be suppressed by prefixing the term with the **$** symbol. Any term prefixed with **$** will be taken literally and no attempt will be made to evaluate it as a *flex* entity. This means that the name of a frame or attribute when prefixed by **$** will be treated simply as text with no reference to its *flex* status.

For instance, consider the following code.

```
action writer ;
   do ask toys
   and write( $toys )
   and nl
   and write( toys ) .

question toys
   Which toys would you like? ;
   choose some of doll, teddy, train, jigsaw, paints .
```

If we now execute the `writer` action on the Prolog command line, we will see that we get back two written values. The first is the absolute value of `toys`, namely the atom itself (because of the use of the $ symbol), and the second is the current value that `toys` evaluates to, which in this case, is the list of items selected from the `toys` question.

```
?- writer.
toys
[teddy, train, paints]
```

KSL Formulae

The formulae of KSL are used to establish relationships between the objects of KSL. These fall into two distinct areas.

Conditions test whether or not something is currently true.

Directives change the current state of an object to some new state.

Conditions

A *condition formula* is used to test the current state (for example of global variables, frames or facts).

Conditions either test for the existence of a variant or compare the value of two expressions; a condition may also be a procedure call.

Equality Comparison

The simplest comparison of two terms is a straightforward equality (or inequality) test. To test for equality use the arithmetic operator **=** or the KSL keywords

```
is
are
is equal to
```

Any of these may be used with **not**.

Examples

```
alpha = beta / 2
jugA`s contents are jugA`s capacity
the size of some brick is equal to 4
the employee`s name is phil

not alpha = beta
not the pupil`s mark is below 70

X is an elephant
X is a kind of animal whose ears are small
```

To test for inequality, use the arithmetic operator **\=**, or the KSL keywords

```
not equal to
different from.
```

Examples

```
the temperature \= the 'freezing point' of water
the staff are not equal to nothing
the capacity of jugA is different from jugA`s volume
```

Existence Test

A test for the existence of a variant is to decide whether or not it has a current value. For this use either of the KSL keywords

known
unknown.

Examples

```
the starter of the meal is unknown
the temperature is known
```

Direct Comparison

A direct *comparison* of two terms uses the built-in *typographical* ordering of Prolog terms. For comparison, use the arithmetic operators **>**, **<**, **=<**, **>=**, or the KSL keywords

greater than
greater than or equal to
above
at or above
less than
less than or equal to
below
at or below

Any of these may be prefixed by **not**.

Examples

```
alpha > beta / 2
the temperature =< the 'freezing point' of water
the pupil`s mark is not below 50
the temperature is at or above boiling_point
the likelihood of frost is less than probable
the food`s calories is less than or equal to 400
```

Relative Comparison

The *relative comparison* of two terms is determined by their relative positions within a defined `group` construct. The group can be specified explicitly as an ordered set, using the KSL keywords **according to**, or implicitly by name (i.e. a group previously defined).

Any of the above direct comparison operators may be used to define the type of the comparison.

Examples

```
its colour is at or above the colour of money
   according to { red , blue , white , green }

group fuzzy_ordering
   certain, probable, possible, unlikely, impossible .

the likelihood of frost is less than probable
   according to fuzzy_ordering
```

Set Membership

To test set membership, use the KSL keywords

```
include
includes
included in
does not include
do not include.
```

Examples

```
the staff include { john and mary }
a surprise is included in the contents of the box
the Rodent`s tail does not include bushy
```

Procedure Calls

A condition can be a call to some procedure, either a *flex* `relation`, a *flex* `action`, or a Prolog call (either built-in or user-defined) . It is the means by which conditions (and thus rules and attached procedures) link into Prolog's backward chaining execution mode.

Conjunctions and Disjunctions

Conditions may be logically combined using **and** and **or**. The *precedence* associated with negations, conjunctions and disjunctions (**not** binds more tightly than **and**, which in turn binds more tightly than **or**) can be overridden by the introduction of square brackets.

Examples

```
C is some cat and M is C`s meal
test1 and [ test2( X ) or alpha > 10 ]
not [ test1 and test2 ]
```

Context Switching

If you wish to use a condition, but the context expects a directive, then the context can be *switched* by the use of the word **check**, optionally followed by the word **that**. For example, an `action` requires *directives* but a `relation` requires *conditions*.

Examples

```
relation emp_name( Emp, Name )
   if Name is Emp`s name .

action emp_name( Emp, Name ) ;
   do check that Name is Emp`s name .
```

Directives

Directives are used to change the current state to some new state, where a state consists of the global variables, frames, facts and exceptions.

The changing of global variables and frames are *assignments*, whereas the addition and removal of facts and exceptions are classed as *database maintenance*.

Direct Assignments

An *assignment* consists of a variant on the left hand side and an expression on the right hand side. An assignment will replace any existing value for the variant with the value of the expression. The general formats of a direct assignment are as follows.

```
Variant := Expression
Variant become Expression
Variant becomes Expression
```

Examples

```
methane_level := high
the kettle`s temperature becomes 45
the cinema`s films become { 'Pulp Fiction', 'Apollo 13'
}
```

Incrementing and Decrementing

Arithmetic values can be *incremented* or *decremented*, using the following KSL formats.

```
add Expression to Variant
subtract Expression from Variant
```

Examples

```
add 100 to the car`s mileage
add X`s contents to Y`s contents
subtract X`s spare_capacity from the jug`s contents
subtract 1 from the total
```

Important Note

Incrementing and decrementing work on the *current* value of an attribute. If an attribute has no current value, the effect of adding to it will be to *set* the current value, i.e. a direct assignment. An attempt to *subtract* from a slot

with no current value will fail.

Set Membership

In a similar way, sets may be incremented or decremented using the following KSL formats.

```
include Item(s) in Set
remove Item(s) from Set
```

Note that if there is no existing current value then it is equivalent to direct assignment, as above. An attempt to remove an item when there is no current value will fail.

Examples

```
include lemon_sole in the entree of set_meal
include { fred and mary } in the prizewinners
remove land from the whale`s habitat
remove { david and liz } from the staff
```

New Instances

Directives can dynamically create new instances of frames with local attributes. All other attributes of the parent frame will automatically be inherited by the instance. The general KSL format is as follows.

```
Instance is a new Frame
Instance is another Frame
```

The **whose** keyword may optionally be used to create or assign values to local attributes.

Examples

```
brick8 is another brick
'Tiddles' is another cat whose owner is alexander
plant33 is a new plant whose size is medium
house_2 is a new house whose bedrooms is 4
     and whose floors is 2
     and whose extras are { garden, garage }
```

Database Maintenance

Database maintenance is accomplished by directives which add assertions to, or delete assertions from both the positive (*facts*) and the negative (*exceptions*) databases. Facts may be added and removed using the KSL keywords

```
remember
remember that
forget
forget that
```

Examples

```
remember that pregnant( P )
remember likes( alexander, harvey )
remember not raining
forget danger_level( red )
forget that not boiling
```

Questions

A very specific directive is the asking of a question, using the KSL keyword **ask**.

```
ask Question
```

Examples

```
ask starter
ask password
```

The questions `starter` and `password` will be defined elsewhere (using the **question** keyword).

Procedure Calls

Finally, a directive can be a call to some procedure. It is the means by which directives (and thus rules and attached procedures) link into Prolog's backward chaining execution mode.

KSL Control Structures

The control structures in *flex* fall into two categories: conditional statements, and control loops.

If-Then-Else

Directives and conditions can be combined to form **if-then-else** statements.

if-then-else statements test a given condition, and if the condition holds the directives indicated by **then** are performed, otherwise the directives indicated by **else** are performed.

The basic format of an **if-then-else** loop is:

```
if condition(s)
then directive(s)
else directive(s)
end if
```

There may be multiple conditions combined with **and** or **or**, and there may also be multiple directives combined with **and**.

Examples

```
action print_result ;
do if the result of test is greater than 50
   then write( 'You have passed' )
   else write( 'You have failed' )
end if .

action test_mileage ;
do if   the mileage of the car is greater than 20000
        or the age of the car is greater than 2
   then subtract 300 from the car`s value
         and add 20 to the car`s premium
   else add 30 to the car`s premium
end if .
```

Repeat-Until Loops

A **repeat-until** loop repeats the given directive until a condition holds. The directive is performed *before* the test is made.

The basic format of a **repeat-until** loop is:

```
repeat directive(s)
until condition(s)
end repeat
```

Example

The following **action** repeatedly adds one to the current value of the up attribute of count and subtracts one from the current value of the down attribute of count until the up value is greater than or equal to the down value.

```
action balance ;
   do repeat add 1 to the up of count
                 and subtract 1 from the down of count
      until the up of count >= down of count
      end repeat
   and write( 'Balance ' )
   and write( up of count ) and tab( 1 )
   and write( down of count ) and nl .
```

Note the use of the built-in Prolog predicates write/1, tab/1 and nl/0.

Recall that the increment and decrement operators, add and subtract, only increment and decrement the *current* values of attributes. If there is no current value the directive subtract will fail. In the case of the add directive, if there is no current value the increment will be *assigned* to the current value of the slot.

While-Do Loops

While loops repeat the given directive while a condition holds. The directive is performed after the test is made. The basic format of a **while-do** loop is as follows.

```
while condition(s)
do directive(s)
end while
```

Example

The following action would be used to remove from the customer loan records of a bank all instances of customer whose loan has been repaid.

```
action remove_paid_customer ;
   do   while Customer is some customer
```

```
      and the loan of Customer is paid
do remove_instance( Customer )
end while .
```

For Loops

Simple **for** loops repeat the given directives for every solution found to a given condition.

The basic format of a **for** loop is:

```
for condition(s)
do directive(s)
end for
```

Examples

The following `action` could be used to print the names of all the instances of `employee`, adding a new line between each (`nl/0` is a built-in Prolog predicate).

```
action print_employees ;
   do   for every Name is some instance of employee
        do write( Name ) and nl
        end for .
```

Extended For Loops

Extended **for** loops repeat the given directives while an index steps from one limit to another.

The basic format of a **for-from-to-step** loop is:

```
for  counter
from expression1
to   expression2
step expression3
do   directive(s)
end for
```

If the **step** *expression3* is omitted, the *counter* will be incremented in steps of 1.

Example

The following action steps through values for X from -20 to 20 in steps of 5, and for every value of X steps through the values of Y from -20 to 20 in steps of 1, and for each Y value prints the value of X times Y.

```
action multiply ;
  do   for X from -20 to 20 step 5
       do   for Y from -20 to 20
            do write( X * Y ) and nl
            end for
       end for .
```

KSL Sentences

The previous sections have described the valid KSL representations for objects and the relationships between them.

In this section we describe the valid sentences in which such objects and formulae can occur. They constitute what can and cannot be stated in a KSL program.

A KSL *sentence* begins with one of the following KSL keywords.

```
action
constraint
data
demon
do
frame
function
group
instance
launch
question
relation
rule
ruleset
synonym
template
watchdog
```

Each sentence is terminated by a space and a full-stop.

Frames

There are three parts to the definition of a frame:

1. The first part of a frame specifies whereabouts in the hierarchy of frames it exists, by specifying its parent frames (if any). The frame hierarchy determines how attributes are inherited.

2. The second part of a frame is optional and specifies what the default attributes of a frame are. The defaults remain throughout the life-span of the frame, and are used in the absence of any current value being assigned.

3. The third and final part to a frame is also optional and represents a refinement of the inheritance hierarchy. Alternative parent frames can be nominated for the inheritance of specific attributes, or specific attributes may be declared *not* to be inherited.

The general format of a frame is as follows.

```
frame Name
default Attribute1 is Value ;
inherit Attribute2 from Frame .
```

The simplest example of a frame is a declaration of the frame's name only.

Example

```
frame toy .
```

To this may be added some default values for its slots.

Examples

```
frame picture
   default colour is red and
   default shape is oval and
   default pen_size is thin and
   default link is arrow.

frame text_spec
   default size is 12 and
   default font is 'Courier' and
   default style is normal.

frame computer
   default memory is 24 and
   default disk is 500 and
   default accessories are
             { monitor and keyboard and mouse } and
   default speed is 66.
```

To specify the parents of a frame, use the keywords

```
   is a
```

```
   is a kind of.
```

Examples

```
frame macintosh is a kind of computer .

frame green_picture is a picture ;
   default colour is green .
```

To specify inheritance of particular attributes, use the keywords

```
   inherit ... from
   do not inherit.
```

Examples

```
frame wedge is a kind of block , toy ;
   do not inherit shape and
   inherit volume from block and
   inherit density from pyramid, toy .

frame block
   default volume is its size to the power of 3 and
   default shape is regular ;
   inherit usage from toy  .
```

Instances

An *instance* is a particular instance of a frame, differing from the frame itself in that an instance can have only one parent frame, can have no children, and all its values are deemed to be *current* values (so it may not have `default` values declared).

Instances may have inheritance declared as for frames.

For *instances*, use the KSL format

```
   instance Instance is a Frame ;
      Attribute is Value .
```

or

```
   instance Instance is an Frame
   instance Instance is a kind of Frame
   instance Instance is an instance of Frame
```

Examples

```
instance brick7 is an instance of brick ;
   shape is cuboid .

instance tweety_pie is a kind of bird ;
   habitat is a cage and
   predator is tom ;
   do not inherit motions .

instance smudge is a cat ;
   meal is rabbit .
```

```
instance silky is a cat ;
   meal is chicken and
   drink is milk .
```

Rules

A *rule* is a forward chaining rule and is the construct which is used during each cycle of the forward chaining engine.

The major parts of a rule are the conditions which must be satisfied before the rule can fire, and the directives which constitute the firing mechanism.

In addition there may be an explanation associated with the rule.

Finally, a score may be attached to the rule to indicate its priority (the higher the score the more likely it is to be fired). Such scores are used to resolve conflicts whenever more than one rule can be fired at any one time.

The general format of a rule is as follows.

```
rule rule_name
   if condition(s)
   then directive(s) ;
   because explanation ;
   score score .
```

The **because** and **score** parts are optional.

Examples

```
rule prescribe_lomotil
   if the patient`s complaint is diarrhoea and
   the patient`s disorders
        do not include liver_disorder and
   not pregnant( patient )
   then advise( patient, lomotil ) .

rule feed_cat
   if C is some cat
     and C`s condition is hungry
   then feed( C, C`s food )
   and C`s condition becomes fed ;
   because We need to feed hungry cats ;
   score 5 .
```

Rulesets

The *ruleset* is the construct which governs the running of the forward chaining engine. Defining the `ruleset` determines which rules will be considered for firing (i.e. the initial rule agenda), in what order they will be considered, when the forward chaining will terminate etc.

Recall that each cycle of the forward chaining engine fires one rule, that rule being the first rule found whose conditions are satisfied (or the rule with the highest score out of all rules whose conditions are satisfied, if conflict resolution is being used). The search for the rule to fire always starts at the beginning of the list of rules in the current rule agenda. Rules may be removed, added or re-ordered after each cycle.

A *flex* program is run by starting the forward-chaining engine, using the KSL directive **`invoke ruleset`**; normally this will be as part of an `action`, which can then be run from the Prolog command line as a normal Prolog query.

Within a ruleset you can specify:

- The *initial rule agenda* (this is mandatory).
- The *initiation directives* to be performed prior to starting the engine.
- The conditions which can *terminate* the engine.
- The rule *selection* algorithm to be used.
- The rule agenda *update* algorithm to be used.
- The procedure to be used when a rule *misfires*.

Any combination of the above specifications may be included in the definition of a ruleset. The only specification which is mandatory is the initial rule agenda; all other specifications are optional.

The most general format of a `ruleset` is as follows.

```
ruleset Name
   contains rule(s) ;
   initiate by doing directive(s) ;
   terminate when conditions(s) ;
   select rule using rule_selection ;
   update ruleset agenda_update ;
   when a rule misfires do directive(s) .
```

Specifying The Initial Rule Agenda

The simplest declaration of a `ruleset` is just to state which rules are to be used, using the KSL keyword

```
contains
```

In this case *flex* will use the rules in the given set, starting at the top of the list for each cycle, and terminating when no more rules can be fired (i.e. when there are no rules whose conditions can be satisfied).

If all currently defined rules are to be included, use the KSL keywords

```
all rules.
```

Examples

```
ruleset example_set1
   contains all rules .
```

This declares that the `ruleset` should contain all the rules currently defined.

```
ruleset mover
   contains push, pull, lift .
```

The initial rule agenda will contain the rules `push`, `pull` and `lift`.

```
ruleset example_set2
   contains packing_rules .
```

Sets the initial rule agenda to the rules defined by the group `packing_rules` (this will have been declared as a KSL `group` construct).

Initialisation Routines

Some initial actions may optionally be specified which will be performed *before* the forward chaining engine is started. For this use the KSL keywords

```
initiate by doing
```

followed by one or more directives.

Examples

```
ruleset example_set2
   contains all rules ;
   initiate by doing write( 'Starting' ) and nl .
```

This allows a message to be output before the forward chaining begins .

```
ruleset my_rules
   contains my_rule1, my_rule2, my_rule3 ;
   initiate by doing action1 and action2 and action3 .
```

This declares that the three actions `action1`, `action2` and `action3` should be performed before the forward chaining engine starts.

Terminating The Engine

Termination conditions may be set for a particular ruleset using the KSL keywords

terminate when

followed by some conditions. This provides an alternative way of stopping the *flex* program: without termination conditions the forward chaining engine stops when no more rules can be fired.

Examples

```
ruleset example_set4
   contains all rules ;
   terminate when condition1 and condition2 .
```

This specifies that the forward chaining engine will stop when both `condition1` and `condition2` are satisfied (even if there are still some rules which could be fired).

```
ruleset pack_shopping
   contains pack_small_items, pack_large_items ;
   terminate when the shopping is empty .
```

This specifies that the forward chaining engine will stop when the value of `shopping` becomes empty.

Selecting Rules

Normally *flex* will consider each rule in the order specified in the ruleset. However, a number of alternative rule selection criteria may be applied instead, using the KSL keywords

select rule using.

The default rule selection is **first come first served**, which considers rules in the order in which they are given and fires the first rule whose conditions are satisfied.

```
ruleset example_set5
   contains all rules ;
   select rule using first come first served .
```

The rule selected for each cycle is the first rule in the agenda whose conditions are satisfied. This is the default for selecting rules and is equivalent to omitting any rule selection criteria.

Rules may be selected using **conflict resolution**. In this case an attempt is always made to use the "best" rule according to the scoring system built into the rules. Each rule should have a **score** associated with it, and the conflict resolution system then fires the rule whose conditions are

satisfied with the highest score.

```
ruleset example_set6
   contains all rules ;
   select rule using conflict resolution .
```

Note that there is a high cost to using conflict resolution since *every* rule must be tested before the highest score is known.

An adaptation of conflict resolution is to set a threshold value, so that the first rule whose conditions are satisfied and whose score is greater than the threshold value will be fired. This reduces the search somewhat.

```
ruleset example_set7
   contains all rules ;
   select rule using conflict resolution
        with threshold 7 .
```

The rules are selected using the conflict resolution scoring system with a threshold value (in this example the threshold value is 7).

Finally, the *flex* programmer may specify a routine which makes a rule selection from the current agenda.

```
ruleset example_set8
   contains all rules ;
   select rule using my_selector .
```

Here *flex* will look for a program called `my_selector/3` which should be defined with three arguments:

```
my_selector( RuleAgenda, RuleConditions, Actions )
```

where at the time of the call `RuleAgenda` will be instantiated to a list of the names of the rules currently in the rule agenda, and `RuleConditions` and `Actions` are variables which `my_selector` should bind to the conditions and actions respectively of the required rule. For more information see the description of the predicates `new_rule/5`, `isa_rule/5` and `forward_chain/5` in the "*flex* Predicates" chapter of this manual.

Updating The Rule Agenda

Each time a rule is fired, there is the option of updating the rule agenda for the next cycle of the forward chaining engine. The default is to leave the set of rules exactly as it is, so that the rules are always considered in the same order every time. Other options may be specified using the KSL keywords

```
update ruleset
```

The options are as follows.

Each time a rule is fired it can be removed from the ruleset. This effectively

means that each rule is fired just once.

```
ruleset example_set9
   contains all rules ;
   update ruleset by removing each selected rule.
```

Each time a rule is fired it can be moved to the top or the bottom of the current ruleset. Moving it to the top means it will be the first rule to be considered next time round, or moving it to the bottom means it will be the last rule to be considered. These two options are specified as follows.

```
ruleset example_set10
   contains all rules ;
   update ruleset by promoting each selected rule.

ruleset example_set11
   contains all rules ;
   update ruleset by demoting each selected rule.
```

The rule agenda may be viewed as a cyclic queue, whereby the rule *following* the selected rule is used as the top of the agenda for the next cycle.

```
ruleset example_set12
   contains all rules ;
   update ruleset by cyclic rotation of rules .
```

The ruleset may be updated by removing from the rule agenda any rules whose conditions were not satisfied in the last cycle of the forward chaining engine.

```
ruleset example_set13
   contains all rules ;
   update ruleset by removing any unsatisfied rule.
```

A rule "network" may be specified enabling the agenda to switch between different groups of rules for each cycle. Each group may be defined using the KSL `group` construct, and after each rule is fired, the rule agenda becomes the set of rules specified by the `group` with the same name as the rule which was just fired.

```
ruleset example_set14
   contains all rules ;
   update ruleset using rule transition network.
```

Finally, in addition to these built-in update methods you may define your own.

```
ruleset example_set15
   contains all rules ;
   update ruleset using user_update .
```

In this case *flex* will look for a definition of `user_update/3` (which may be

a *flex* `action` or `relation` or a Prolog program) which should be defined as follows:

```
user_update( SelRule, Rules, NewRules )
```

where at the time of the call, `SelRule` will be the name of the rule just fired, `Rules` will be a list of the names of the other rules currently in the agenda, and `NewRules` will be a variable which `user_update` should bind to a list of the names of the rules to be used as the next rule agenda.

Specifying Misfire Procedures

Sometimes rules may *misfire*, meaning the conditions of a rule may be satisfied but the actions may fail or cause an error for some reason. You may specify what action is to be taken if this happens, using the KSL keywords

```
when a rule misfires do
```

followed by one or more directives.

You can choose to ignore the misfiring of rules by giving `true` as the misfire procedure. This means the forward chaining engine will continue with its next cycle as if nothing happened.

```
ruleset example_set16
   contains all rules ;
   when a rule misfires do true .
```

You can choose to fail the forward chaining engine when a rule misfires by giving `fail` as the misfire procedure.

```
ruleset example_set17
   contains all rules ;
   when a rule misfires do fail .
```

Finally you can choose to define your own procedure to process the misfire.

```
ruleset example_set18
   contains all rules ;
   when a rule misfires do my_misfire .
```

In this case *flex* will look for a definition of `my_misfire/1` (which may be a *flex* `action` or a Prolog program) which should be defined as follows:

```
my_misfire( Rule )
```

where at the time of the call, `Rule` will be the name of the rule which just misfired.

Actions

An **`action`** is a collection of directives to perform. It is similar to the

definition of a Prolog predicate, except only one definition of each action is allowed. An action may have any number of arguments.

An action may be executed either from the Prolog command line, or from within a *flex* `relation`, a *flex* `rule` or another `action`, or from within a Prolog program.

It is often helpful to use actions to access or display the values of slots, or to set up testing data etc.

The basic format of an **`action`** is as follows.

```
action action_name ;
   do directive(s) .
```

Examples

```
action demo ;
   do  invoke ruleset demo_rules .

action empty_into( X, Y ) ;
   do   Y`s contents := Y`s contents + X`s contents
   and  X`s contents := 0 .

action write_data( Customer ) ;
   do write( 'Customer : ' ) and
      write( Customer`s name ) and
      nl and
      write( 'ID : ' ) and
      write( Customer`s id ) and
      nl .
```

Note that Prolog built-in predicates (such as `write/1` and `nl/0` above) may be included in the directives of an **`action`**.

Any of the above actions may be executed from the Prolog command line as queries, e.g.

```
?- demo .

?- empty_into( jug1, jug2 ).

?- write_data( cust23 ).
```

Relations

A *relation* is defined by a collection of clauses. It is a logical relationship over entities, similar to a Prolog predicate.

Unlike `action`s, there may be several different ways of defining a relationship.

A `relation` may be executed either from the Prolog command line, or from within a *flex* `action`, a *flex* `rule` or another `relation`, or from within a Prolog program.

The basic format of a relation is as follows.

```
relation relation_name
  if condition(s) .
```

The *`conditions`* may be other relations, actions or Prolog programs. If the conditions are not satisfied any further definitions of the relation will be tried.

Examples

```
relation is_leaf( Node )
  if the links_out of Node is empty.

relation yes_or_no( yes ).
relation yes_or_no( no ).
```

In the following example, the relation `bank_balance` retrieves the named customer's balance if the customer exists, otherwise it creates a new customer and sets the customer's balance to 0.

```
relation bank_balance( Customer, Balance )
  if C is some customer and
    Customer is C`s cname and
    Balance is C`s balance.

relation bank_balance( Customer, 0 )
  if gensym( cust, NewCust ) and
    NewCust is a new customer and
    NewCust`s cname becomes Customer and
    NewCust`s balance becomes 0.
```

The `fibonacci` relation will generate the `N`th number in the Fibonacci sequence (where each number in the sequence is the sum of the two preceding numbers).

```
relation fibonacci( 1 , 1 ) .
relation fibonacci( 2 , 1 ) .
relation fibonacci( N , Z )
  if   fibonacci( N-1 , X )
  and  fibonacci( N-2 , Y )
  and  Z is X + Y .
```

A relation can optionally be prefixed by the symbol **$** to indicate that none of its constituent parts should be dereferenced. The relation is interpreted exactly as a Prolog clause, with each term taken literally rather than having any *flex* interpretation imposed.

Example

```
relation $ fibonacci( N , Z )
   if N1 is N-1
   and  N2 is N-2
   and  fibonacci( N1 , X )
   and  fibonacci( N2 , Y )
   and  Z is X + Y .
```

Note that in this example we cannot use the expressions `N-1` and `N-2` as arguments (as in the previous `fibonacci` example), because the $ symbol means that no dereferencing of arguments will occur.

Functions

A *function* is evaluated at run-time by replacing the left-hand-side of the equality symbol with the right-hand-side. A function may contain conditional statements which will be evaluated before the replacement takes place.

The basic formats of a `function` are as follows.

```
function function_name = expression .

function function_name = Variable
  where condition(s) .

function function_name =
  if condition(s)
  then expression1
  else expression2 .
```

Examples

```
function access_code = current_user`s access .

function father( X ) = Y
   where   parent( Y, X )
   and     male( Y ) .

function taxed( Amount ) = T
  where T is Amount * 1.17 .

function maximum( A, B ) =
  if A > B then A
  else B .

function fibonacci( N ) =
   if N > 1
   then fibonacci( N-1 ) + fibonacci( N-2 )
   else 1 .
```

Launches

A *launch* is a procedure which can be attached to frames and is automatically invoked whenever a new instance of that frame is created. Its primary use is in setting up the initial characteristics of frame instances.

The directives associated with a launch are executed immediately *after* the instance is created.

A launch can be tailored such that it fires only under certain circumstances.

The general format of a `launch` is as follows.

```
launch launch_name
   when Instance is a new Frame
   and condition(s)
   then directive(s) .

launch launch_name
   when Instance is a new instance of Frame
   and condition(s)
   then directive(s) .
```

The *conditions* are optional.

Examples

This first example will simply display a message each time a new instance of `carrier` is created.

```
launch pick_up_new_carrier_bag
   when Bag is a new instance of carrier
   then write( 'I need another carrier bag ' )
   and write( Bag )
   and nl .
```

The next example executes the `female_enrolment_questions` procedure each time a new instance of `student` is created who is female.

```
launch female_enrolment
   when Person is a new student
   and female( Person )
   then female_enrolment_questions( Person ) .
```

Constraints

A *constraint* is a validity check which can be attached to an attribute of a frame. It is automatically invoked whenever the value for that slot changes.

The checks associated with a constraint are executed immediately *before* the value of the slot is to be changed, and the value only changes if this

check succeeds. If the check fails then the slot is *not* updated, and the update itself will fail.

Note that a `demon` may be used to perform checks *after* a slot value has changed.

The general format of a `constraint` is as follows.

```
constraint constraint_name
  when Attribute changes
  from Expression1
  to Expression2
  and condition1(s)
  then check that condition2(s)
  otherwise directive(s) .
```

The **from**, **to**, the *conditions1* and the **otherwise** are all optional.

Example

This example checks that a `vessel` is never filled to beyond its capacity. If the proposed value is invalid, an error message is written (the **otherwise** part of the constraint) and the slot update does not take place.

```
constraint maximum_contents_of_vessel
   when the contents of Vessel changes to X
   and Vessel is some vessel
   then check that number( X )
   and  X =< the Vessel `s capacity
   otherwise write( 'Illegal contents of vessel ' )
   and write( Vessel )
   and write( ' Contents ' )
   and write( X )
   and  nl .
```

This next example makes sure that the number of `widgets` in a `container` never falls below zero.

```
constraint minimum_widgets
   when the widgets of Container changes to N
   and Container is some container
   then check that N > 0
   otherwise Container`s widgets becomes 0.
```

Demons

A *demon* is a procedure which can be attached to an attribute of a frame. It is automatically invoked whenever the value for that slot changes.

The directives associated with a `demon` are executed immediately *after* the slot value changes.

A demon can be tailored such that it fires only for given values and/or only

under certain circumstances.

Note that a `constraint` may be used to perform checks *before* a slot value is changed.

The general format of a `demon` is as follows.

```
demon demon_name
   when Attribute changes
   from Expression1
   to Expression2
   and condition(s)
   then directive(s) .
```

The **`from`** and **`to`** and the *conditions* are all optional.

Examples

The first example displays information about the `contents` of any `jug` when its value changes.

```
demon spy_the_contents
   when the contents of any jug changes from X to Y
   then write( 'jug change ... ' )
   and write( Y - X )
   and nl .
```

The second example checks the changes in `temperature` value (in any frame), and takes necessary actions when it rises above boiling point.

```
demon check_for_melt_down_of_core
   when the temperature changes to T
   and T is above boiling_point
   then remember that danger_level( red )
   and shut_down .
```

Watchdogs

A *watchdog* checks the access rights to an attribute of a frame. It is automatically invoked whenever there is a request for the *current* value (not the default value) of that slot.

The checks associated with a watchdog are executed immediately *before* the value is accessed. If the check fails then the access call also fails.

The general format of a `watchdog` is as follows.

```
watchdog watchdog_name
   when Attribute is requested
   and condition1(s)
   then check that condition2(s)
   otherwise directive(s) .
```

The *conditions1* and the **otherwise** are optional.

Examples

```
watchdog account_security
   when the contents of account is requested
   and  outside_office_hours
   then check that the user`s classification is above
99
   otherwise report_illegal_entry .

watchdog gift_security
   when the surprise of the box is requested
   then check that the date is 'Christmas Day' .
```

Data

A *data* entry is a series of directives which assign values to slots or modify the facts and exceptions in the database.

Its use is in establishing the initial state of some problem, since the `data` sentences will be executed at compile time, and again each time the *flex* routine `restart/0` is called. This initial state (i.e. as if all *flex* code has just been compiled but not run) can always be retrieved by calling `restart/0`.

The general format of a `data` sentence is as follows.

```
 data data_name
   do directive(s) .
```

Examples

```
data start_up_configuration
   do the contents of jugA becomes 2
   and  jugB`s contents becomes jugB`s capacity - 2
   and  remember that danger_level( yellow ) .

data initialise_cinemas
   do c1 is a new cinema whose film is 'Frankenstein'
        and whose seats is 50 and
      c2 is a new cinema whose film is 'The Third Man'
        and whose seats is 75.
```

Do Statements

A *do* statement is, like `data` above, a series of directives which assign values to slots or modify the facts / exceptions in the database.

Its use is similar to a `data` statement in establishing the initial state of some problem, but, unlike `data`, anything executed by a **do** statement in this way is not re-established by the *flex* routine `restart/1`. A **do** statement is for executing directives once at compile time, and once only.

The general format of a `do` statement is as follows.

do *directive(s)* .

Examples

```
do the contents of jugA become 2
and  jugB`s contents become jugB`s capacity - 2
and  remember that danger_level( yellow ) .

do c1 is a new car whose engine_size is 1100
   and c1`s model := 'Whizzo'
   and c1`s colour := metallic_blue .
```

To illustrate the difference between `do` and `data`, consider the following example.

```
frame widget
   default screws is 0 and
   default bolts is 0.

relation get_widget( W, S, B )
   if W is some widget
      whose screws is S and whose bolts is B.

do w1 is a new widget
      whose screws is 50 and whose bolts is 10.

data new_widgets
   do w2 is a new widget
      whose screws is 20 and whose bolts is 30.
```

We may find the current instances of `widget` using the query `get_widget/3`. Initially after compilation, there will be two widgets, `w1` and `w2`, as declared by the `data` and `do` statements above.

```
?- get_widget( W, S, B ).
W = w1, S = 50, B = 10
W = w2, S = 20, B = 30
```

However, if we precede the query with the *flex* call to `restart/0`, which reinitialises the *flex* workspace, we will see that the widget `w2` is restored (as in the `data` statement) but the widget `w1` is not restored (because it was only declared in a `do` statement).

```
?- restart, get_widget( W, S, B ).
W = w2, S = 20, B = 30
```

Questions

A *question* describes the manner in which some information is to be extracted from the user. The response is obtained either through one of *flex*'s standard answer mechanisms or a customized answer.

A standard **question** comprises a question name, followed by some text to be displayed, followed by the type of question and an optional explanation indicated by the KSL keyword **because**.

Menu Questions

In a menu question, the user is presented with a collection of options, and is offered the choice of making either a single selection or multiple selections.

The general formats of a menu question are as follows.

```
question question_name
   text of question ;
   choose from menu items ;
   because explanation .

question question_name
   text of question ;
   choose some of menu items .

question question_name
   text of question ;
   choose one of menu items .
```

An *explanation* (the `because` part) is optional in each case.

The KSL keywords **choose one of** are for single selection menus, where the user must select just one item; the others are for multiple selection menus where the user can select any number of items.

The user's response to the question is stored in a global variable called *question_name*.

Examples

```
question choose_a_colour
   Select a colour ;
   choose one of  red, green, yellow, blue, black,
white .
```

The text `Select a colour` will be displayed when the question is asked. In this case the user may select *just one* of the items displayed, and this selection will be stored as the current value of the global variable `choose_a_colour`.

In the following example the user may select *any number* of the items displayed.

```
question choose_colours
   Select your colours ;
   choose some of red, green, yellow, blue, black,
white.
```

This is exactly equivalent to:

```
question choose_colours
   Select your colours ;
   choose from red, green, yellow, blue, black, white.
```

Alternatively, a *group* name may be given for the menu items, e.g.

```
group colours
   red, green, yellow, blue, black, white.

question choose_a_colour
   Select a colour ;
   choose one of  colours.
```

To actually invoke a question, use the KSL keyword **ask** in a directive.

For example, the colours question given above will be invoked by the following *flex* relation.

```
relation get_colour( C )
  if ask choose_a_colour
   and C is choose_a_colour .
```

The Prolog query

```
   ?- get_colour( C ).
```

will display a standard menu dialog (its actual appearance will depend on the particular implementation of *flex* that you are using), and the user's choice will be stored in the *flex* global variable `choose_a_colour` and returned as the value of the Prolog variable `C`.

Note that to define `get_colour/1` as an equivalent *action*, we would need to use the KSL `check that` construct.

```
action get_colour( C ) ;
  do ask choose_a_colour
   and check that C is choose_a_colour .
```

Explanations

The question dialog will normally contain a button labelled *Explain*, which allows for explanations to be attached to questions. The user may click the *Explain* button and the *flex* programmer may provide text to be displayed at this point (using the KSL keyword **because**), as an explanation of why the question is being asked, or perhaps to give some help on answering the question.

For example:

```
question choose_a_colour
```

```
   Select a colour ;
   choose one of  colours ;
   because This colour will be used for text display.
```

Keyboard Input

Users may be asked to enter information from the keyboard using the KSL keyword **input**. The type of the data expected may then be specified, as either `name`, `number`, `integer` or `set`.

The general formats of a keyboard input question are as follows.

```
question question_name
   text of question ;
   input datatype ;
   because explanation .

question question_name
   text of question ;
   input variable
   such that condition(s) ;
   because explanation .
```

As before, the **because** part is optional. The *datatype* may be

```
   name
   number
   integer
   set
```

Examples

```
question get_country
   Which country do you live in? ;
   input name .

question number_of_children
   How many children do you have ? ;
   input integer .

question room_width
   What is the width of the room (in metres)? ;
   input number .

question child_names
   Please enter your children's names ;
   input set .
```

The **input** keyword may also be followed by some conditions which define the type of the answer, using the KSL keywords **such that**.

The following example expects an integer greater than 18 for the age. Error messages will be given if the input does not comply with these conditions.

```
question age_of_applicant
   Please enter your age ;
   input X such that integer( X ) and X > 18 .
```

User Defined Questions

The *flex* programmer may define alternative ways of getting information from the user. The KSL keywords

answer is ... **such that**

may be used to specify another routine which will provide the necessary information. This routine may be a *flex* `relation`, a *flex* `action` or a Prolog program, probably one which presents a customised dialog into which the user will enter data.

The general format of a user-defined question is as follows.

```
question question_name
   answer is Value
   such that directive .
```

Example

```
question wine
   answer is X such that get_wine_from_user( X ) .
```

The `get_wine_from_user/1` routine will be defined elsewhere and will return in X the wine chosen by the user.

Groups

A *group* is a means of gathering several names under one collective heading. The basic format of a **group** is:

```
group group_name
   first_item, second_item, .., last_item .
```

A group may simply be a set of items grouped together for convenience, for use in a question.

Example

```
group wall_colours
   magnolia, coffee, apple_white, barley, buttermilk .

question wall_colour
   Please choose a colour for your room ;
   choose from wall_colours .
```

The items in a group may be the names of rules, and the group may then be referenced in a `ruleset`, or it may be part of a rule network.

Example

```
group food_rules
   ask_for_the_starter, ask_for_the_entree .

ruleset dinner
   contains food_rules .
```

A group may also be an ordered list of items which which may then be used for comparison purposes (using the KSL keywords **according to**).

Examples

```
group fuzzy_ordering
   impossible, improbable, possible, probable,
definite.

relation less_likely( A, B )
   if A is less than B
      according to fuzzy_ordering.
```

Synonyms

A *synonym* can be used to make code easier to read by allowing the replacement of frequently occurring terms or expressions with a mnemonic. The **synonym** is textually replaced by its *expression* when the sentence in which it occurs is being parsed.

One advantage of synonyms is improved readability; another is that when a constant value changes it is easier to change it in the synonym declaration than in every place it might occur in the code.

The basic format of a **synonym** is:

synonym *name_to_replace replacement_expression* **.**

All occurrences of *name_to_replace* in the KSL sentences *following* this declaration will be replaced by the term *replacement_expression*.

Examples

```
synonym pi 3.14159.
```

At compile time all occurrences of **pi** will be replaced by the number 3.14159 .

```
synonym weight_calc
   its volume times its density .
```

All occurrences of weight_calc will be replaced by the term **its** volume **times its** density.

```
synonym interest_rate 8.5 .
```

All occurrences of `interest_rate` will be replaced by the number 8.5. Here, if the interest rate changes, only one change is necessary (to the `synonym` definition) before recompiling source code.

Templates

Like a `synonym`, a *template* assists the readability of KSL statements. A **`template`** can be thought of as a parameterised synonym, allowing textual replacements at compile-time. The `^` symbol is used to indicate an argument position.

A template may have a positive form and an optional negative form. The basic format of a **`template`** is:

```
template replacement
   positive_template ;
   negative_template .
```

At compile time, any terms of the form *`positive_template`* will be replaced by *`replacement`*, and any terms of the form *`negative_template`* will be replaced by **`not`** *`replacement`*, with arguments being replaced appropriately. Often the *`replacement`* term will be defined as a `relation` or an `action`.

For example, suppose we have the following template (which only has a positive form).

```
template empty_out
   empty out ^ .
```

Then in KSL source code the expression

```
   empty out jug1
```

will be converted at compile time to

```
   empty_out( jug1 )
```

We may define the action `empty_out/1`, also making use of the template, as follows.

```
action empty out X ;
   do X`s contents becomes 0 .
```

This will be translated at compile time into:

```
action empty_out( X ) ;
   do X`s contents becomes 0 .
```

Examples

```
template on_top_of
   block ^ is on top of ^ ;
   block ^ is not on top of ^ .
```

Having defined the `on_top_of` template, we can then use it in KSL statements. For example, the following statement

```
   block p is on top of the table
```

will be converted at compile time to the term

```
   on_top_of( p, table )
```

The statement

```
   block q is not on top of p
```

will be converted at compile time to the term

```
   not on_top_of( q, p ).
```

The following `template` defines price comparisons of items. The phrase `is more expensive than` is the positive template, with `is cheaper than` being the negative template.

```
template price_greater
  ^ is more expensive than ^ ;
  ^ is cheaper than ^ .
```

Thus the expression `X is more expensive than Y` will translate at compile time to `price_greater( X, Y )`, and `A is cheaper than B` will translate at compile time to `not price_greater( A, B ).`

We may define the relation `price_greater/2` as follows.

```
relation Item1 is more expensive than Item2
  if the price of Item1
     is greater than the price of Item2.
```

Note that

```
   Item1 is more expensive than Item2
```

translates at compile time to

```
   price_greater( Item1, Item2 ).
```

8. Run-Time Interpretation of KSL

In this chapter we discuss how a *flex* program is interpreted at run-time.

Representation of KSL Objects

Each and every KSL object is represented by a Prolog structure. It is these structures which are interpreted at run-time by the *flex* predicates.

The following table defines the mapping from KSL objects onto Prolog structures.

KSL Object	Prolog Representation
value V	`V`
global variable N	`N`
`nothing`	`[ ]`
`something` / `anything`	`_`
attribute A of frame F	`@( A, F )`
attribute A of any frame	`@( A, _ )`
attribute A of an instance of frame F	`@( A, I ) : some_instance( F, I )`
attribute A1 of (attribute A2 of frame F)	`@( A1 , @( A2, F ))`
(attribute A of frame F1) of frame F2	`@( @( A, F1 ), F2 )`
etc.	
{ E_1 **or** … E_k **}**	`or( [` E_1, … , E_k `] )`
{ E_1 **and** … E_k **}**	`[` E_1, … , E_k `]`
{ X **:** C **}**	`S : setof( X, C, S )`
if C **then** E_1 **else** E_2	`X : (C -> X=`E_1 `; X=`E_2`)`
X **such that** C	`X : C`
everthing in group G	`L : every_instance(G,L)`
every instance of frame F whose …	`S : setof(I , (some_instance(F,I), …),S)`
some instance of group G	`X : some_instance(G,X)`
some instance of frame F whose …	`I : (some_instance(F,I),…)`
self-reference to attribute A	`its(A)`
the answer to question Q	`X : answer(Q,X)`
an expression with binary operator O	`O(`E_l`,`E_r`)`
an expression with unary operator O	`O(E)`
F**(** E_1, … ,E_k **)**	`F(`E_1, … ,E_k`)`

```
$ E                    $(E)
```

Interpretation of KSL Sentences

Under the normal Prolog interpretation of relation calls, arguments are treated as fixed objects which can only pattern-match with each other. This interpretation is used wherever a **$** symbol appears in a KSL program.

Under the *flex* interpretation of relation calls, however, arguments are dereferenced immediately before the call is made. Furthermore, KSL objects can be dereferenced to alternative values. For example, the construct `or([`E_1`,…,`E_k`])` dereferences to some E_i where $1 \le i \le k$.

This interpretation means that backtracking can take place not only over relation calls, but also over the dereferencing mechanism itself. For example, consider the following call to the built-in predicate `member/2`.

```
member( X, [ 123, or([a,b,c]) ] )  .
```

Under a Prolog interpretation the possible solutions are

```
X = 123
X = or([a,b,c]) .
```

But under a *flex* interpretation there are 6 possible solutions :

```
X = 123   X = a
X = 123   X = b
X = 123   X = c
```

This is because the *flex* dereferencing mechanism generates three possible Prolog goals from the original *flex* goal, namely:

```
member( X , [123,a] ) .
member( X , [123,b] ) .
member( X , [123,c] ) .
```

Each of these yields two solutions for X.

Dereferencing of KSL Objects

The following table shows the method by which Prolog terms (and thus KSL objects) are dereferenced.

Term	Dereferenced Value
`N`	Get the current value for global variable `N` and dereference it
`T`	`T`, whenever `T` is an atomic term
`[`E_1`, … ,`E_k`]`	`[`T_1`, … ,`T_k`]` where each E_i dereferences to T_i
`or([`E_1`, … ,`E_k`])`	Choose some E_i and dereference it to T_i
`@(A,F)`	First of all, dereference `A` and `F` to `A1` and `F1`, respectively; then get the current value, the default value or an inherited value for attribute `A1` of frame `F1`, and dereference it; during the subsidiary dereferencing `F1` becomes the original frame
`its(A)`	If there is an original frame `F` then the dereferencing proceeds as if it were `@(A,F)`, otherwise the structure `its(A)` is returned
`X : C`	Execute the condition `C` and dereference the resulting instance of `X`
`O(`E_l`,`E_r`)`	Dereference the operands down to T_l and T_r; if they are both numbers then apply the operator, otherwise return the structure `O(`T_l`,`T_r`)`
`O(E)`	Dereference the operand down to `T`; if it is a number then apply the operator, otherwise return the structure `O(T)`
`F(`E_1`,…,`E_k`)`	Dereference each E_i to some T_i; if it is a defined function then replace with the right-hand-side of the definition and continue dereferencing, otherwise return the structure `F(`T_1`,…,`T_k`)`
`$(E)`	`E`

Representation of KSL Sentences

In the following sections we describe how the sentences of KSL are represented as Prolog programs. When a KSL program has been compiled the specified predicates will exist in the Prolog database. See also the chapter on *flex* Predicates for an explanation of the Prolog predicates which are equivalent to the KSL given here.

Frames

Frames are mapped onto clauses for the relations `frame/2`, `default_value/3` and `link/3`, as follows.

```
frame( Frame, Parents ).

default_value( Attribute, Frame, Value ).

link( Attribute, Frame, InheritFrom ).
```

In addition, for each parent frame PF a clause for `PF/1` is created.

Example

```
frame brick is a kind of block, toy ;
   default weight is its volume times its density ;
   inherit volume from toy, pyramid and
   do not inherit shape .
```

will be represented by the Prolog assertions

```
frame( brick, [block,toy] ).
block( brick ).
toy( brick ).

default_value( weight, brick, its(volume) *
its(density) ).
link( volume, brick, [toy,pyramid]).
link( shape, brick, []).
```

Instances

Instances of frames are asserted as clauses for the relations `instance/2`, `current_value/3` and `link/3`, as follows.

```
instance( Instance, Frame ).

current_value( Attribute, Instance, Value ).

link( Attribute, Instance, InheritFrom ).
```

In addition, for the parent frame PF a clause for PF/1 is created.

Example

```
instance tweety_pie is a bird ;
   habitat is a cage and
   predator is sylvester ;
   do not inherit motions .
```

will be represented by the Prolog assertions

```
instance( tweety_pie, bird ).
bird( tweety_pie ).

current_value( habitat, tweety_pie, cage ).
current_value( predator, tweety_pie, sylvester ).

link( motions, tweety_pie, []).
```

Launches

Launches are mapped onto clauses of the relation launch/5, as follows.

```
launch( Launch,
        Instance,
        Frame,
        Context,
        Action ).
```

Example

```
launch female_enrolment
   when Person is a new student
   and  female( Person )
   then female_enrolment_questions( Person ) .
```

will be represented by the Prolog assertion

```
launch( female_enrolment,
        Person,
        student,
        female(Person),
        female_enrolment_questions(Person) ).
```

Demons

Demons are asserted as clauses for the relation demon/7, as follows.

```
demon(  Demon,
        Attribute,
        Frame,
        OldValue,
        NewValue,
```

```
          Context,
          Action ).
```

Example

```
demon check_for_melt_down_of_core
   when the temperature of vessel changes to T
   and  T is above boiling_point
   then remember that danger_level( red )
   and  shut_down .
```

will be represented by the Prolog assertion

```
demon(  check_for_melt_down_of_core,
        temperature,
        vessel,
        _,
        T,
        comparison(>,T,boiling_point),
        ( remember(danger_level(red)),prove(shut_down)
)
    ).
```

Constraints

Constraints are mapped onto the relation `constraint/8`, as follows.

```
   constraint(  Constraint,
                Attribute,
                Frame,
                OldValue,
                NewValue,
                Context,
                Check,
                Action ).
```

Example

```
constraint maximum_contents_of_jugA
   when the contents of jugA changes to X
   then check that number( X )
   and  X =< jugA`s capacity .
```

will be represented by the Prolog assertion

```
constraint(  maximum_contents_of_jugA,
             contents,
             jugA,
             _,
             X,
             true,
             (  prove(number(X)),
                comparison(=<,X,@(capacity,jugA))
             ),
             fail
          ).
```

Watchdogs

Watchdogs are mapped onto clauses of the relation `watchdog/6`, as follows.

```
watchdog( Watchdog,
          Attribute,
          Frame,
          Context,
          Check,
          Action ).
```

Example

```
watchdog security_check
   when the contents of account are requested
   then check user`s classification is above 7.
```

will be represented by the Prolog assertion

```
watchdog( security_check,
          contents,
          account,
          true,
          comparison( >, @(classification,user), 7)
               fail ).
```

Production Rules

Rules are mapped onto instances of the relation `rule/5`, as follows.

```
rule(   Rule,
     Conditions,
     Action,
     Explanation,
     Score ).
```

Example

```
rule prescribe_lomotil
   if the patient complains of diarrhoea and
   the patient does not suffer from liver_disorder
      and the patient is not pregnant
   then I advise the patient to take lomotil
   browse file medical7 .
```

will be represented by the Prolog assertion

```
rule(     prescribe_lomotil,
        (  prove(complains(patient,diarrhoea)),
   disprove(suffers(patient,liver_complaints)),
        disprove(pregnant(patient)) ),
        take(patient,lomotil),
        browse(medical7),
```

```
        0 ).
```

Actions

Actions are represented directly by the corresponding Prolog clause. In addition, the name of the action is remembered as a clause of `relation/2`.

Example

```
action empty_into( X, Y ) ;
   do   Y`s contents := Y`s contents + X`s contents
   and  X`s contents := 0 .
```

will be represented by the Prolog clauses

```
relation( empty_into, 2 ) .

empty_into( X, Y ) :-
   new_value( @(contents,Y), @(contents,Y) +
@(contents,X) ),
   new_value( @(contents,X), 0 ) .
```

Relations

Relations are represented directly by the corresponding Prolog clause. In addition, the name of the relation is remembered as the relation `relation/2`.

Example

```
relation sister( X, Y )
   if father( X ) = father( Y )
   and  not male( Y )
   and  X is not equal to Y .
```

will be represented by the Prolog clauses

```
relation(sister,2) .

sister(X,Y) :-
   equality(father(X),father(Y)),
   not male(Y),
   not equality(X,Y) .
```

Functions

Functions are represented by instances of the relation `function/3`, as follows.

```
   function( Function, Arguments,Value ).
```

Example

```
function father( X ) = Y
   where   parent( X, Y )
   and     male( Y ) .
```

will be represented by the Prolog assertion

```
function( father,
        [X],
        ( Y : parent(X,Y), male(Y) ) ).
```

Data

Data is mapped onto clauses of the relation `data/2`, as follows.

```
   data( Data, Action ).
```

In addition, the directives of the data statement are immediately executed.

Example

```
data start_up_configuration
   the contents of jugA are 2 and
   the contents of jugB are the capacity of jugB - 2
   and remember that danger_level( yellow ) .
```

will be represented by the Prolog assertion

```
data(     start_up_configuration,
        (  new_slot(contents,jugA,2),
           new_slot(contents,jugB,@(capacity,jugB-2)),
                remember(danger_level(yellow)) ) ).
```

Do Statements

Do statements are not stored, but are executed directly.

Questions

Questions are stored as clauses of the relation `question/4`, as follows.

```
   question( Question,
             Text,
             Answer,
             Explanation ).
```

Example

```
question starter
   Please choose a starter for your meal ;
   choose from pate, soup, melon ;
```

```
   because The starter is part of a meal .
```

will be represented by the Prolog assertion

```
question( starter,
          ['Please',choose,a,starter,for,your,meal],
             single([pate,soup,melon]),
          text(['The',starter,is,part,of,a,meal]) ).
```

Groups

Groups are mapped onto the relation `group/2`, as follows.

```
   group( Group, Elements ).
```

Example

```
group colours
   black, blue, green, cyan, red, magenta, yellow,
white .
```

will be represented by the Prolog assertion

```
group(  colours,
   [black,blue,green,cyan,red,magenta,yellow,white] ).
```

Synonyms

Synonyms are mapped onto the relation `synonym/2`, as follows.

```
   synonym( Synonym, Term ).
```

Example

```
synonym weight_calc
   its volume times its density .
```

will be represented by the Prolog assertion

```
synonym( weight_calc, its(volume)*its(density) ).
```

Templates

Templates are stored as clauses of the relation `template/3`, as follows.

```
   template( Template, PositiveForm, NegativeForm ).
```

Each form of the template is stored as a list of lists which indicate the parameter positions.

Example

```
template ontopof
```

```
block ^ is on top of ^ ;
block ^ is not on top of ^ .
```

will be represented by the Prolog assertion

```
template( ontopof,
        [[block],[is,on,top,of],[]],
        [[block],[is,not,on,top,of],[]] ).
```

9. *flex* Toolkit Predicates

This chapter describes the Prolog predicates provided by the *flex* toolkit. These are provided for accessing the functionality of *flex* from Prolog rather than using the KSL, or you may choose to write some of your application in Prolog using these predicates rather than their KSL equivalents. For some operations, such as list processing, it is sometimes easier and more efficient to use the Prolog *flex* predicates in a Prolog program rather than using KSL constructs. You may freely intermix code written using the *flex* Prolog predicates and code written using the KSL: frames, actions and relations may be accessed from Prolog just as Prolog programs may be accessed from the KSL.

The predicates are listed here in alphabetical order; the *flex* predicate index at the back of this document references the predicates by category.

---

add_value(+Slot,+Term)

`add_value/2` can either be used to augment lists or to increment numbers.

The given *`Term`* is first dereferenced down to some *`Value`*.

The *`Slot`* should either be a compound term of the form `@(`*`Attribute`*`,`*`Frame`*`)` or it is the name of a global variable, in which case the name of the frame to be used is `global`.

If there is an existing slot then it is augmented with *`Value`*. Otherwise, a new slot is created.

Example

```
add_value( @(colour,flag), [red,blue,cyan,white] ).
```

---

all_rules(-Names)

Find the names of all rules in the workspace.

Example

```
?- all_rules( N ).
N = [prescribe_lomotil,fill_B_from_A]
```

---

ancestor(+Frame,-Ancestor)

`Ancestor` is an ancestor of *`Frame`* in the hierarchy, i.e. *`Ancestor`* is the *`Frame`* itself, or a parent frame, or a grandparent frame, or a great-grandparent frame, etc.

Example

```
?- ancestor( moby_dick, A ).
A = moby_dick
A = whale
A = mammal
A = animal
A = ocean_dweller
```

---

answer(+Name,-Value)

Retrieve the answer given to the question *`Name`*. If the question has not yet been asked, then the call to `answer/2` will automatically invoke the corresponding call to `ask/1` to ask the question and obtain an answer.

Examples

```
?- answer( starter, X ).
X = soup
```

---

ask(+Name)

Ask the question ***Name*** and record the answer given as a new value for the global variable ***Name*** (see `new_value/2`). The answer can then be accessed either through the dereferencing mechanism (`dereference/2`) or directly through `answer/2` or `isa_value/2`.

Examples

```
ask( starter ).
ask( dessert ).
```

---

atn(+Name,+Names,-Newnames)

This is a built-in program for re-organising the rule agenda after each cycle of the forward chaining engine according to a rule transition network. It is defined as follows.

```
atn( Name, _, Newnames ) :-
   isa_group( Name, NewNames ).
atn( Name, _, [] ).
```

---

back(+Name,+Names,-Newnames)

This is a built-in program for re-organising the rule agenda after each cycle of the forward chaining engine. It puts the rule ***Name*** to the back of the rule agenda. It is defined by the program:

```
back( Name, Names, Newnames ) :-
   once( Name, Names, Remainder ),
   append( Remainder, [Name], Newnames ).
```

---

comparison(?Relation,+Term1,+Term2)

The two terms are dereferenced (to***Value1*** and ***Value2***) and the values compared. The ordering relation is the built-in typographical ordering of Prolog terms.

If ***Value1*** is greater than ***Value2*** then the ***Relation*** is either > or >=.

If ***Value1*** is less than ***Value2*** then the ***Relation*** is either < or =<.

Otherwise, if the two values are the same then the ***Relation*** is either =, >= or =<.

Examples

```
?- comparison( =<, temperature, freezing_point ).
no

?- comparison( X, @(contents,jugB), @(capacity,jugB) ).
```

```
X = '<'
```

---

comparison(?Relation,+Term1,+Term2,+Group)

`comparison/4` behaves as `comparison/3` except that the supplied ***Group*** name is used as the ordering relation. The two terms are dereferenced and the resulting values (***Value1*** and ***Value2***) compared according to their relative positions in the ***Group*** elements.

If ***Value1*** is after ***Value2*** in the ***Group*** then the ***Relation*** is either > or >=.

If ***Value1*** is before ***Value2*** in the ***Group*** then the ***Relation*** is either < or =<.

Otherwise, if the two values are the same then the ***Relation*** is either =, >= or =<.

Examples

```
?- comparison( X, blue, Y, colours ) .
X = '<', Y = black
X = '=', Y = blue
X = '>', Y = green  etc.
```

---

crss(+Names,-Name,-Action)

This is a built-in program for selecting a rule to fire during each cycle of the forward chaining engine. It is the conflict resolution scoring system algorithm, and is defined by the following program.

```
crss( Names, Name, Action ) :-
   find_the_best( Names, no_bsf, bsf(Name,_,Action) ).

find_the_best( [], Best, Best ).
find_the_best( [Name|Names], Best1, Best3 ) :-
   best_so_far( Name, Best1, Best2 ),
   find_the_best( Names, Best2, Best3 ).

best_so_far( Name, BSF, bsf(Name,Value,Action)) :-
   not isa_disabled_rule( Name ),
   isa_rule( Name, Conditions, Action, _, Score ),
   Conditions,
   dereference( Score, Value ),
   ( BSF=bsf(_,Previous,_) -> Previous < Value ; true
),
   !.
best_so_far( _, BSF, BSF).
```

---

crss(+Names,-Name,-Action,+Threshold)

This is a variation of the conflict-resolution algorithm to incorporate threshold values. The selection stops as soon as a satisfied rule attains the threshold

value.

---

crss(+Names,-Name,+FiringMechanism,-If,-Then,-Vars)

FiringMechanism is one of {repeat,first,last,all} using first will give the same behaviour as crss/3 .

---

crss(+Names,-Name,+FiringMechanism,-If,-Then,-Vars,+Threshold)

Threshold is a value for chosing the rule or rules with the best score.

---

cycle(+Name,+Names,-Newnames)

This is a built-in program for re-organising the rule agenda after each cycle of the forward chaining engine according to cyclic rotation. It is defined by the program:

```
cycle( Name, Names, Newnames ) :-
   append( Front, [Name|OldBack], Names ),
   append( Front, [Name], NewBack ),
   append( OldBack, NewBack, Newnames ).
```

---

dereference(+Term,-Value)

This is the kernel of the *flex* run-time procedures. It is used throughout the system as a means of interpreting *flex* objects. See the earlier chapter 'Interpretation of KSL', for a complete description of the dereferencing mechanism.

---

descendant(+Frame,-Descendant)

Descendant is a descendant of ***Frame*** in the hierarchy. That is, ***Descendant*** is the ***Frame*** itself, or a child frame, or a grandchild frame, or a great-grandchild frame, etc.

Example

```
?- descendant( animal, D ).
D = animal
D = mammal
D = whale
D = moby_dick
```

---

disable_rules(+Names)

All the rules in the list ***Names*** are disabled, and will not be considered for firing. It is as if they have been temporarily removed from the workspace.

Example

```
disable_rules( [prescribe_lomotil,fill_B_from_A] ) .
```

---

disprove(+Goal)

Dereference each of the arguments of the ***Goal*** and then try to disprove it.

If there is a Prolog program for the ***Goal***, then that program is run together with negation-as-failure. Otherwise, the workspace is searched for a matching exception (see `new_exception/1`), or a check is made that there are no matching facts (see `new_fact/1`).

Example

```
?- disprove( X=or([a,b,c]) ).
no

?- disprove( likes( X, Y) ).
X = mary, Y = fred
```

---

enable_rules

Re-enable all rules which have previously been disabled.

---

enable_rules(+Names)

Re-enable all of the rules in ***Names*** which may have been previously disabled (see `disable_rules/1`). They can now be considered as potential rules to fire.

Example

```
enable_rules( [fill_B_from_A] )
```

---

equality(+Term1,+Term2)

The two terms are dereferenced and the resulting terms equated. If the equate fails (or there is backtracking into `equality/2`) then alternative dereferenced values will be equated.

Example

```
?- equality( pressure, high ).
no

?- equality( temperature, 2*50 ).
yes

?- equality( X, or([a,b,c]) ).
X = a
X = b
X = c
```

---

every_instance(+Name,-Elements)

Retrieve (or check) all ***Elements*** which are instances of ***Name***. ***Name*** can be the name of a group, the name of a frame or an instance or the name of a unary relation.

Examples

```
?- every_instance( fuzzy_ordering, E ).
E = [impossible,improbable,possible,probable,definite])

?- every_instance( whale, E ).
E = [moby_dick]
```

---

explain(+Rules)

Initiate a dialog which allows each rule in the list***Rules*** to be explained. The explanation depends upon the rule's explanation parameter as set by `new_rule/5`.

Example

```
explain( [fill_B_from_A, empty_A, empty_B_into_A] )
```

---

fcfs(+Names,-Name,-Action)

This is a built-in program for selecting a rule to fire during each cycle of the forward chaining engine. It is the first come first served selection algorithm, defined as follows:

```
fcfs( Names, Name, Action ) :-
   member( Name, Names ),
   not isa_disabled_rule( Name ),
   isa_rule( Name, Conditions, Action, _, _ ),
   Conditions.
```

---

fcfs(+Names, -Name, +FiringMechanism, -If, -Then, -Vars)

FiringMechanism is one of {repeat,first,last,all} using first will give the same behaviour as crss/3.

---

fire_rule(+Name)

Unconditionally execute the action part associated with the rule ***Name***. The rule is fired regardless of whether ***Name*** is enabled or whether its conditions are satisfied.

Example

```
fire_rule( prescribe_lomotil ) .
```

---

fixed(+Name,+Names,-Newnames)

This is a built-in program for processing the rule agenda after each cycle of the forward chaining engine. It leaves the rule agenda unchanged, and is defined by the program:

```
fixed( _, Names, Names ) .
```

---

flatten_group(+Name,-Elements)

Groups, like frames and instances, can be built up into group hierarchies or networks. This occurs when the member elements of a group are themselves names of other groups.

The `flatten_group/2` predicate is used to flatten such a group hierarchy or network into the constituent member elements.

---

flex_name(?Name)

Retrieve (or test) the ***Name*** of a *flex* predicate.

Example

```
?- flex_name( N ).
N = new_frame
N = isa_frame
etc.
```

---

forget_exception(+Term)

Remove the given ***Term***, if it exists, from the exceptions in the workspace.

If subsequent backtracking happens, then the retracted exception will be re-asserted.

Example

```
forget_exception( likes(X,mary) ).
```

---

forget_fact(+Term)

Remove the given ***Term***, if it exists, from the facts in the workspace.

If subsequent backtracking happens, then the retracted fact will be re-asserted.

Example

```
forget_fact( male(fred) ) .
```

---

forward_chain(+Selection,+Misfire,+Termination,+Update,+Agenda)

Initiate a forward chaining session with the given selection algorithm and rule agenda.

Selection is the name of the algorithm for selecting rules during each cycle of the forward chaining engine. There are three built-in algorithms:

`fcfs`	First come first served.
`crss`	Conflict resolution scoring system.
`crss(T)`	Conflict resolution with a threshold value.

In addition, a user-defined algorithm can be nominated. For example, my_`selector/3` randomly selects a rule from the agenda.

```
my_selector( Agenda, Rule, Action) :-
   length( Agenda, N ),
   Posn is irand( N ) + 1,
   mem( Agenda, [Posn], Rule ),
   isa_rule( Rule, _, Action, _, _ ) .
```

If the *Selection* algorithm fails then the call to `forward_chain/5` will terminate.

Misfire is the name of a program which is executed whenever the action associated with the selected rule fails.

There are three built-in misfire programs :

`true`	Ignore misfires.
`fail`	The original call to forward_chain/5 will also fail.
`misfire`	Report the misfire and abort the process.

In addition, the name of a user-defined program can be nominated, such as the following `mf/1`.

```
mf( Name ) :-
   write( 'Rule has misfired ' - Name ), nl.
```

Termination is a Prolog call which is tested during each and every cycle of the forward chaining engine. As soon as the *Termination* criterion succeeds the call to `forward_chain/5` will terminate.

In particular, if *Termination* is the atom `fail`, then the call to `forward_chain/5` will only terminate when a situation arises in which no rules can be selected.

Agenda and *Update* together specify the rule agenda, and how it changes after each cycle of the forward chaining engine.

The *Agenda* specifies the initial agenda, and may be one of the following:

- the atom `all`, to signify all the rules in the workspace.
- the name of a group whose elements are rule names, or names of sub-groups.
- the name of a rule.
- a list in which each member is one of the above.

Update is the name of a program for changing the agenda, and is one of the following:

`fixed` Do not change the rule agenda at all.

`once` Fire each rule in the agenda at most once. This is accomplished by removing the rule name from the agenda after it is fired.

`front` Bring the rule which has just been fired to the front of the agenda. This makes it even more likely to be fired in the next cycle.

`back` Send the rule which has just been fired to the back of the agenda. This makes it less likely to be fired in the next cycle.

`cycle` View the agenda as a cyclic queue. This is accomplished by starting the selection phase immediately after the rule which has just been fired.

`possibles` Remove from the agenda those rules which were rejected as candidates (i.e. their IF parts were not satisfied) in the selection phase.

`atn` Completely replace the agenda with a new agenda, depending upon the name of the rule (also a group name) which was fired. Use this when the rules are connected into a network through `new_group/2`. The selection of rules at each stage will determine the route taken through the network.

In addition, the name of a user-defined program can be nominated, such as the following `organise/3`. This allows the user to select which rules to retain in the next cycle of the forward-chaining engine.

```
organise( Rule, Rules, Newrules) :-
   some_random_permutation( [Rule|Rules], Newrules ) .
```

Examples

```
forward_chain( fcfs,
          true,
          prescription_made,
          fixed,
          prescription_rule_group ) .

forward_chain( crss,
          misfire,
          fail,
          once,
          all ) .
```

---

forward_chain(+Selection,+Misfire,+Termination,+Update,+Agenda,-Sequence)

`forward_chain/6` behaves exactly as `forward_chain/5`, except that an additional piece of information is returned: ***Sequence*** is the list of rule names which were actually fired during the forward chaining session.

---

forward_chain(+Selection,+Misfire,+Termn,+Update,+Agenda,-Sequence,-Result)

`forward_chain/7` behaves exactly as `forward_chain/6`, except that an additional piece of information, the ***Result***, is returned.

There are two situations in which the forward chaining can terminate, and the ***Result*** parameter reflects this.

0 The ***Termination*** criterion was eventually satisfied.

1 A situation was reached in which no rules could be selected.

---

front(+Name,+Names,-Newnames)

This is a built-in program for re-organising the rule agenda after each cycle of the forward chaining engine. It brings the most recently fired rule to the front of the agenda, and is defined by the program:

```
front( Name, Names, [Name|Newnames]) :-
   once( Name, Names, Newnames ).
```

---

inclusion(+List,+Term)

The ***Term*** and all of the members of the ***List*** are first of all dereferenced.

If ***Term*** is itself a list, then a check is made that all of its members are also members of the dereferenced ***List***. Otherwise, the dereferenced ***Term*** is a member of the dereferenced ***List***.

Examples

```
?- inclusion( [a,b,c,d,e], [c,e,c] ).
```

```
yes

?- inclusion( [a,b,c,d], [c,e,c] ).
no

?- inclusion( [a,b,c], X ).
X = a
X = b
X = c
```

---

inherit(+Attribute,+Frame,-Value)

This procedure will search the frame hierarchy starting at the parents of *Frame*. The search will terminate when an ancestor of *Frame* can be found which has a current or default *Value* for the *Attribute*.

`inherit/3` is the inheritance part of the `lookup/3` procedure.

Examples

```
?- inherit( habitat, moby_dick, V ).
V = ocean

?- inherit( predator, moby_dick, V ).
no
```

---

inherit(+Attribute,+Frame,-Value,-Ancestor)

The `inherit/4` procedure behaves as the `inherit/3` procedure except that an additional piece of information, *Ancestor*, is returned. This indicates where exactly in the frame hierarchy the *Value* came from.

Example

```
?- inherit( habitat, moby_dick, V, A ).
V = ocean, A = moby_dick
```

---

inheritance

Return to the default inheritance settings of

```
Search = depth_first
Root = root_last
Plurality = singular
Effort = 9
```

(See `inheritance/4` below for a description of these parameters).

---

inheritance(?Search,?Root,?Plurality,?Effort)

The `inheritance/4` procedure allows for the inspection and/or alteration of certain characteristics of the inheritance algorithm. These characteristics

are :

Search The method by which the frame hierarchy is searched.

Root Whether to visit the root frame before or after any ancestor frames.

Plurality Whether the inheritance can provide unique or alternative values.

Effort How much effort should be used when searching the frame hierarchy.

The *Search* parameter is one of the following atoms:

`depth_first` A depth-first search explores each branch of the frame hierarchy before considering other branches (the default search strategy).

`breadth_first` A breadth-first search visits all ancestors at a given level before moving upwards to the next level. That is, all parents before any grandparents, all grandparents before any great-grandparents, etc.

The *Root* parameter is one of the following atoms:

`root_last` A root-last search will visit ancestor frames before the special root frame. This is the default search tactic.

`root_first` A root-first search visits the root frame before any ancestor frames.

The *Plurality* parameter is one of the following atoms:

`singular` Singular inheritance corresponds to finding the first, and only the first solution when searching the frame hierarchy for the value of an attribute. This is the default inheritance mode.

`multiple` Multiple inheritance corresponds to finding alternative values for an attribute, enumerated in the normal Prolog fashion by backtracking.

The *Effort* parameter is any non-negative integer (default is `9`). It indicates how deep the search of the frame hierarchy should go before giving up. Special cases are `0` (no inheritance at all) and `1` (only parent frames are to be considered).

Examples

```
?- inheritance( S, R, P, E ).
S = depth_first, R = root_last, P = singular, E = 9
```

```
?- inheritance( breadth_first, _, multiple, 2 ).

?- inheritance( S, root_first, P, 0 ).
S = breadth_first, P = multiple
```

---

initialise

Clear the workspace completely of all frames, rules, questions, etc.

---

isa_constraint(?Name,?Attribute,?Frame,-Old,-New,-Context,-Check,-Action)

Retrieve the characteristics of a constraint. (See `new_constraint/8` for a description of the arguments.)

Example

```
?- isa_constraint( N, A, F, OV, NV, CX, CK, AT ) .
N = maximum
A = contents
CX = true
CK = ( number(NV), comparison(=<,NV,@(capacity,F)) )
AT = ( write([contents,F,:=,NV]), nl )
```

---

isa_data(?Name,-Action)

Retrieve the ***Name*** and ***Action*** associated with a `data` directive.

Example

```
?- isa_data( N, A ).
N = start_up_configuration
A = ( new_slot(contents,jugA,2),
    new_slot(contents,jugB,@(capacity,jugB)-2),
    remember_fact(danger_level(yellow)) )
```

---

isa_default(?Attribute,?Frame,-Value)

Test for a default value for the attribute of a frame.

A call to `isa_default/3` will backtrack through all known default values and unify ***Attribute*** with the name of the attribute, ***Frame*** with its frame and bind ***Value*** to the default value of the attribute.

The call fails if there are no default values currently defined.

Example

```
?- isa_default( A, F, V ).
A = habitat, F = ocean_dweller, V = ocean
A = habitat, F = mammal, V = land
etc.
```

---

isa_demon(?Name,?Attribute,?Frame,-Old,-New,-Context,-Action)

Retrieve the characteristics of a demon. (See the `new_demon/7` predicate for a description of the arguments.)

Examples

```
?- isa_demon( N, A, F, OV, NV, CX, AT) .
N = kettle_boiling,
A = temp
F = kettle
CX = comparison(>, NV, 100)
AT = (prove(make_steam), prove(nl), prove(whistle))
```

---

isa_disabled_rule(?Name)

Retrieve (or test) the *`Name`* of a rule which is currently disabled. (See the `disable_rules/1` predicate).

Example

```
?- isa_disabled_rule( N ).
N = prescribe_lomotil
```

---

isa_exception(?Term)

Retrieve a known exception from the workspace.

Example

```
?- isa_exception( F ).
F = likes(mary,fred)
```

---

isa_fact(?Term)

Retrieve a known fact from the workspace.

Example

```
?- isa_fact( F ).
F = danger_level(red)
```

---

isa_frame(?Name,?Parents)

Test for the existence of a frame and its parents.

A call to `isa_frame/2` will backtrack through all known frames and unify *`Name`* with the name of a frame and *`Parents`* with a list of the frame's parents.

The call fails if there are no frames currently defined.

Example

```
?- isa_frame( N, P ).
N = animal, P = []
N = mammal, P = [animal]
N = whale, P = [mammal,ocean_dweller]

?- isa_frame( mammal, [P1|Rest] ).
P1 = animal, Rest = []
```

---

isa_function(?Name, ?Arguments, ?Value)

Retrieve the definition of a function. (See the `new_function/3` predicate for a description of the arguments.)

Example

```
?- isa_function( N, A, V ).
N = father, A = [X], V = (Y : (parent(X,Y),male(Y)))
```

---

isa_group(?Name,-Elements)

Retrieve the ***Name*** and ***Elements*** of a group.

Example

```
?- isa_group( N, E) .
N = colours
E = [black,blue,green,cyan,red,magenta,yellow,white])

N = rules1
E = [ask_for_the_starter,ask_for_the_entree])
```

---

isa_instance(?Instance,?Frame)

Test for the existence of an instance of a frame.

A call to `isa_instance/2` will backtrack through all known instances and unify ***Instance*** with the name of an instance and ***Frame*** with the corresponding frame.

The call fails if there are no instances currently defined.

Example

```
?- isa_instance( I, F ).
I = moby_dick, F = whale
```

---

isa_launch(?Name,?Instance,?Frame,-Context,-Action)

Retrieve the characteristics of a launch. (See the `new_launch/5` predicate for a description of the arguments.)

Example

```
?- isa_launch( N, I, F, CX, AT) .
N = new_female_employee
F = employee
CX = female(N)
AT = data_for_female_employee(N)
```

---

isa_link(?Attribute,?Frame,-Parents)

Test for an inheritance link for the attribute of a frame.

A call to `isa_link/3` will backtrack through all known inheritance links and unify ***`Attribute`*** with the name of the attribute, ***`Frame`*** with its frame and ***`Parents`*** with a list of the frames from which the attribute should be inherited.

The call fails if there are no links currently defined.

Example

```
?- isa_link( A, F, P ) .
A = habitat, F = whale, P = [ocean_dweller]
A = colour, F =moby_dick, P = [albino]
A = tail, F = manx, P = []
```

---

isa_logic(?Logic)

Retrieve or check whether a ***`Logic`*** is in force.

Example

```
?- isa_logic( X ).
X = inherit
```

---

isa_question(?Name,-Question,-Answer,-Explanation)

Retrieve a question from the workspace (see the `new_question/4` predicate for a description of the arguments).

Example

```
?- isa_question( N, Q, A, E ).
N = name_of_applicant
Q = ['Please',enter,your,full,name]
A = input(name)
E = text(['No',name,means,no,benefit,!])

N = starter
Q = ['Please',choose,a,starter,for,your,meal]
A = single([pate,soup,melon])
E = none

N = dessert
```

```
Q = ['Please',choose,a,dessert,for,your,meal]
A = multiple(dessert)
E = file(the_complete_irish_cook)
```

---

isa_relation(?Name, ?Arity)

Retrieve or check the ***Name*** and ***Arity*** of a defined relation.

Example

```
?- isa_relation( N, A ).
N = parent
A = 2
```

---

isa_rule(?Name,-Conditions,-Action,-Explanation,-Score)

Retrieve the characteristics of a rule from the workspace. (See the predicate `new_rule/5` for a description of the arguments.)

Example

```
?- isa_rule( prescribe_lomotil, C, A, E, S ).
C = ( prove(complains(patient,diarrhoea)),
      disprove(suffers(patient,liver_complaints)),
      disprove(pregnant(patient)) )
A = prescribe(patient,lomotil)
E = file(medical7)
S = 0
```

---

isa_slot(?Attribute,?Frame,-Value)

Test for the current value for the attribute of a frame.

A call to `isa_slot/3` will backtrack through all known current values and unify ***Attribute*** with the name of the attribute, ***Frame*** with its frame and bind ***Value*** to the current value of the attribute. Before the access is allowed any watchdog procedures (see `watchdog/6`) are first checked.

The call fails if there are no current values defined.

Example

```
?- isa_slot( A, F, V ).
A = habitat, F = moby_dick, V = [atlantic,arctic]
A = predator, F = moby_dick, V = jonah
A = food, F = moby_dick, V = fish

?- isa_slot( food, moby_dick, V ).
V = fish
```

---

isa_synonym(?Name,-Term)

Retrieve the *Name* used as a synonym for the *Term*.

Example

```
?- isa_synonym( N, T ).
N = boiling_point, T = 100
N = weight, T = its(volume)*its(density)
```

---

isa_template(?Name,-Positive,-Negative)

Retrieve both the *Positive* and *Negative* forms of a template. The template forms will be returned as a list of lists, with breaks at the parameter positions.

Example

```
?- isa_template( ontopof, P, N ).
P = [[block],[is,on,top,of,block],[]]
N = [[block],[is,not,on,top,of,block],[]]

P = [[block],[is,on,top,of,table],[]]
N = [[block],[is,not,on,top,of,table],[]]
```

---

isa_value(?Slot,-Value)

Retrieve either the current *Value*, or in its absence the default *Value*, for the *Slot*.

The *Slot* should either be a compound term of the form `@(`*Attribute*`,`*Frame*`)` or it is the name of a global variable, in which case the name of the frame to be used is `global`.

Examples

```
?- isa_value( @(colour,flag), X ).
X = [red,blue,white]

?- isa_value( temperature, X ).
X = 100
```

---

isa_watchdog(?Name,?Attribute,?Frame,-Context,-Check,-Action)

Retrieve the characteristics of a watchdog procedure. (See the predicate `new_watchdog/6` for a description of the arguments.)

Example

```
?- isa_watchdog( N, A, F, CX, CK, AT ) .
N = security
A = balance
F = account
CX = outside_office_hours
CK = comparison(>,@(classification,user),99)
```

```
AT = fail
```

---

is_known(+Slot)

The call succeeds only if there exists a current or default value for the *Slot*.

The *Slot* should either be a compound term of the form @(*Attribute*,*Frame*) or it is the name of a global variable, in which case the name of the frame to be used is `global`.

---

lookup(+Attribute,+Frame,-Value)

Retrieve the current *Value* for an *Attribute* of a *Frame* as in `isa_slot/3`. Only if there is no current value will the default *Value* be returned instead (as in `isa_default/3`).

If there is neither a current nor a default *Value* for the *Attribute* of the *Frame*, then the frame hierarchy is searched. The search finishes whenever an ancestor frame can be found which does have either a current or a default *Value*.

The `lookup/3` procedure is used whenever dereferencing *flex* objects.

Examples

```
?- lookup( colour, moby_dick, V ).
V = white

?- lookup( habitat, moby_dick, V ).
V = [atlantic,arctic]

?- lookup( habitat, whale, V ).
V = land
```

---

lookup(+Attribute,+Frame,-Value,-Ancestor)

The `lookup/4` procedure behaves as `lookup/3` except that an additional piece of information, *Ancestor*, is returned. This indicates where exactly in the frame hierarchy the *Value* came from.

Examples

```
?- lookup( colour, moby_dick, V, A ).
V = white, A = albino

?- lookup( habitat, moby_dick, V, A ).
V = [atlantic,arctic], A = moby_dick

?- lookup( habitat, whale, V, A ).
V = land, A = ocean_dweller
```

---

misfire(+Name)

This is a built-in program for handling misfires during a forward chaining session. The ***Name*** argument is the name of the rule that has misfired. `misfire/1` calls the built-in *flex* error-handler.

---

new_constraint(+Name,?Attr,?Frame,?Old,?New,+Context,+Check,+Action)

Create (or update if the ***Name*** already exists) a constraint which validates slot updates (see `new_slot/3`).

The ***Check*** (a Prolog call) is performed whenever the ***Attribute*** of the ***Frame*** changes from some ***Old*** value (this will be the atom `unknown` if there is no current value) to some ***New*** value. In addition, the ***Context*** (a Prolog call) needs to hold before the constraint is applicable.

The ***Check*** (a Prolog call) occurs immediately *before* the update. Indeed, the update will only be allowed if the ***Check*** succeeds. If the ***Check*** fails, then the ***Action*** (a Prolog call) is initiated and the update (a call to `new_slot/3`) will fail.

Example

```
new_constraint( maximum, contents, F, _, N,
          true,
          ( number(N),
            comparison(=<,N,@(capacity,F)) ),
          write([contents,F,:=,N]) ) .
```

---

new_data(+Name,+Action)

Add (or replace if the ***Name*** already exists) a new data ***Action*** to the workspace. The ***Action*** is automatically invoked whenever there is a call to `run_data/0`, `run_data/1` or `restart/0`, and is generally used to initiate some slot values.

Example

```
new_data( start_up_configuration,
         ( new_slot(contents,jugA,2),
           new_slot(contents,jugB,@(capacity,jugB)-2),
           remember_fact(danger_level(yellow)) ) ) .
```

---

new_default(+Attribute,+Frame,+Value)

Create (or replace) a default ***Value*** for a particular ***Attribute*** of a ***Frame***. The default ***Value*** is used in the absence of any current value (see `new_slot/3`) whenever the ***Attribute*** of the ***Frame*** is accessed.

Examples

```
new_default( habitat, ocean_dweller, ocean ).

new_default( colour, whale, grey ).
```

```
new_default( colour, albino, white ).
```

---

new_demon(+Name,?Attribute,?Frame,?Old,?New,+Context,+Action)

Create (or update if the ***Name*** already exists) a demon which reacts to slot updates (see `new_slot/3`).

The ***Action*** (a Prolog call) is initiated whenever the ***Attribute*** of the ***Frame*** changes from some ***Old*** value (the atom `unknown` if there is no current value) to some ***New*** value.

In addition, the ***Context*** (a Prolog call) must also hold before the demon can be activated.

The ***Action*** (a Prolog call) occurs immediately *after* the update, and will therefore not affect the update itself.

Examples

```
new_demon(spying_slots, Attribute, Frame,
          Old, New, true,
          ( write(Attribute(Frame) = (Old-->New)), nl )
)

new_demon( check_meltdown, global, temperature, _, T,
             comparison(>,T,boiling_point),
          ( remember_fact(danger_level(red)), shut_down
) )
```

---

new_frame(+Name,+Parents)

A new frame called ***Name*** is created whose position in the frame hierarchy is determined by its list of ***Parents***. These ***Parents*** determine from where values can be inherited.

Examples

```
new_frame( animal, [] ) .

new_frame( mammal, [animal] ) .

new_frame( whale, [mammal,ocean_dweller] ) .
```

---

new_function(+Name, +Arguments, +Value)

Record the definition of a function ***Name***. Functions are evaluated at run-time through calls to `dereference/2`.

Example

```
new_function( father, [X], (Y : (parent(X,Y),male(Y)))
) .
```

---

new_group(+Name,+Elements)

Create (or replace if the ***Name*** already exists) a new group containing the given ***Elements***. A group is used in two distinct ways. Firstly, it constitutes a type declaration whereby each element is considered to be of type ***Name*** (see `some_instance/2`). And secondly, it is used as an ordering relation when comparing the relative values of two elements (see `comparison/4`).

Examples

```
new_group( rules1,
   [ask_for_the_starter,ask_for_the_entree]) .

new_group( fuzzy_ordering,
   [impossible,improbable,possible,probable,definite])
```

---

new_instance(+Instance,+Frame)

Create a new ***Instance*** of a specific ***Frame***. All attributes of the ***Frame*** are implicitly inherited by the ***Instance***, unless local slots are created to override the inheritance. Following the creation, any launch procedures associated with the frame (see `new_launch/5`) are activated.

Example

```
new_instance( moby_dick, whale ).
```

---

new_launch(+Name,?Instance,?Frame,+Context,+Action)

Create (or update if the ***Name*** already exists) a launch which reacts to the creation of new instances of frames (see `new_instance/2`).

The ***Action*** (a Prolog call) is initiated whenever an ***Instance*** of the ***Frame*** is created.

In addition, the ***Context*** (a Prolog call) must also hold before the launch can be activated.

Example

```
new_launch(  new_female_employee,
             Name,
             employee,
             female( Name ),
             data_for_female_employee( Name ) ) .
```

---

new_link(+Attribute,+Frame,+Parents)

Establish an inheritance link between a ***Frame*** (and instances of that frame) and a list of ***Parents***, for a particular ***Attribute***. This will override the

default inheritance link established by the original creation of the *Frame* (using `new_frame/2` or `new_instance/2`).

A special case is when the list of *Parents* is empty. This indicates that the *Attribute* should not be inherited at all.

Examples

```
new_link( habitat, whale, [ocean_dweller] ) .

new_link( colour, moby_dick, [albino] ) .

new_link( tail, manx, [] ) .
```

---

new_logic(+Logic)

Change the underlying logic of the system to be *Logic*. The only recognised logic at present is `inherit`.

Example

```
new_logic( inherit ) .
```

---

new_question(+Name,+Question,+Answer,+Explanation)

Add (or replace if the *Name* already exists) a new question to the workspace.

Question is a list of words to be displayed whenever the question is asked (see `ask/1` or `answer/2`).

The *Answer* indicates how an answer is to be obtained. It is one of the following:

`input` Set up a dialog with an edit field into which the user can type words and numbers.

`input(`*T*`)` Set up a dialog with an edit field into which the user can type information. The expected type of the input is determined by *T*, which is one of `set`, `name`, `number`, `integer` or `( X :` *conditions* `)`

`single(`*M*`)` Set up a dialog with a menu *M* from which the user can make a single selection.

`multiple(`*M*`)` Set up a dialog with a menu *M* from which the user can make multiple selections.

Term `:` *Goal* Execute the *Goal* and return the *Term* as the answer to the question.

The *Explanation* parameter is used whenever there is a request to explain the question. This can be invoked from within a standard dialog. It is

one of the following:

`text(M)` Display the canned text given by the list of words *M*.

`file(F)` Browse through the file *F* starting at the topic *Name*.

`none` There is no explanation available for the question.

Examples

```
new_question( name_of_applicant,
        ['Please',enter,your,full,name],
        input(name),
        text(['No',name,means,no,benefit,!]) )

new_question( starter,
        ['Please',choose,a,starter,for,your,meal],
        single([pate,soup,melon]), none )

new_question( dessert,
        ['Please',choose,a,dessert,for,your,meal],
        multiple(dessert),
        file(the_complete_irish_cook) )
```

---

new_relation(+Name,+Arity)

Record the *Name* and *Arity* of a *flex* relation. This information is used when initialising the workspace.

Example

```
new_relation( parent, 2 ) .
```

---

new_rule(+Name,+Conditions,+Action,+Explanation,+Score)

Add a new rule to the current workspace. The rule is accessed through its *Name*.

The *Conditions* (a Prolog call or calls) give the context under which the rule is fireable.

The *Action* (a Prolog call or calls) represent the actual firing mechanism of the rule.

Explanation is used when explaining the rule to a user. It is one of the following:

`text(M)` Echo the list *M* purely as some canned text to be displayed.

`file(F)` Browse file *F* starting at the topic *Name*.

Score is any Prolog term which evaluates via the dereferencing mechanism (see `dereference/2`) to a number. The *Score* parameter is used within

the conflict resolution scoring system (`crss/3`) when deciding which is the best rule to fire.

Examples

```
new_rule( prescribe_lomotil,
        (  prove(complains(patient,diarrhoea)),
           disprove(suffers(patient,liver_complaints)),
           disprove(pregnant(patient)) ),
           prescribe(patient,lomotil),
           file(medical7),
           0 ) .

new_rule( fill_B_from_A
        (
comparison(<,@(contents,jugB),@(capacity,jugB)),
           comparison(>,@(contents,jugA),0) ),
        ( new_value( @(contents,jugA),
                  @(contents,jugA) + @(contents,jugB)),
           new_value(@(contents,jugB),@(capacity,jugB))
),

   text([jugB,was,not,full,and,jugA,was,not,empty]),
           @(contents,jugB)+@(capacity,jugB) ) .
```

---

new_slot(+Attribute,+Frame,+Value)

Create (or replace) a current ***Value*** for a particular ***Attribute*** of a ***Frame*** (or instance of a frame). The current ***Value*** overrides any default value, and is used whenever the ***Attribute*** of the ***Frame*** is accessed.

Before the update is allowed any constraints (see `new_constraint/8`) on the ***Value*** are first checked. Following the update, any associated demons (see `new_demon/7`) are activated.

Examples

```
new_slot( habitat, moby_dick, [atlantic,arctic] ) .

new_slot( predator, moby_dick, jonah ) .

new_slot( food, moby_dick, fish ) .
```

---

new_synonym(+Name,+Term)

Add (or replace if the ***Name*** already exists) a new synonym for the ***Term***.

Whenever a KSL file is reconsulted, all occurrences of ***Name*** will be replaced by ***Term***.

Examples

```
new_synonym( boiling_point, 100 ) .
```

```
new_synonym( weight_calc, its(volume)*its(density) ) .
```

---

new_template(+Name,+Positive,+Negative)

Add the ***Positive*** and ***Negative*** templates for ***Name*** to the workspace. This will not overwrite any existing templates for that ***Name***, but will append onto them.

Templates are used as a method of distributing the predicate ***Name*** amongst the arguments of a relation. This will hopefully assist the readability of KSL sentences.

A template is a non-empty list of partitions, where each partition is a (possibly empty) list of words. The border between one partition and the next indicates an argument position. There will always be one more partition in a template than there are argument positions.

Examples

```
new_template(ontopof,
        [[block],[is,on,top,of,block],[]],
        [[block],[is,not,on,top,of,block],[]] )
```

---

new_value(+Slot,+Term)

The given ***Term*** is first dereferenced down to some ***Value***.

If the ***Slot*** is a compound term of the form `@(`***Attribute***`,`***Frame***`)` then the following call is made :

```
new_slot( Attribute, Frame, Value) .
```

Otherwise, if the ***Slot*** is the name of a global variable then the following call is made:

```
new_slot( Slot, global, Value) .
```

The frame `global` is a special frame reserved for the value of global variables.

Example

```
new_value( @(contents,jugB), @(capacity,jugB) ) .

new_value( temperature, boiling_point) .
```

---

new_watchdog(+Name,?Attribute,?Frame,+Context,+Check,+Action)

Create (or update if the ***Name*** already exists) a watchdog procedure which controls the access to a slot (see `isa_slot/3`).

The ***Check*** (a Prolog call) is performed whenever there is a request for the

Attribute of the *Frame*; in addition, the *Context* (a Prolog call) must also hold before the watchdog can be invoked.

The *Check* occurs immediately *before* the access. Indeed, the access will only be allowed if the *Check* succeeds. If the *Check* fails, then the *Action* (a Prolog call) is initiated and the access (a call to `isa_slot/3`) will also fail.

Example

```
new_watchdog(  security,
               balance,
               account,
               outside_office_hours,
               comparison(>,@(classification,user),99),
               fail ) .
```

---

nospy_chain

Disable the tracing of the forward chaining engine by removing any spypoints.

---

nospy_fact(?Name)

Remove any spypoints that have been set for the fact of the given *Name*.

Example

```
nospy_fact( likes ) .
```

---

nospy_rule(?Name)

Remove any spypoints that have been set on the named rule. If the *Name* is a variable then spypoints are removed from all the rules.

Example

```
nospy_rule( _ ) .
```

---

nospy_slot(?Attribute, ?Frame)

Remove any spypoints which may have been set for the *Attribute* and *Frame*.

Example

```
nospy_slot( colour, _ ) .
```

---

once(+Name,+Names,-Newnames)

This is a built-in program for re-organising the rule agenda after each cycle of the forward chaining engine. The most recently fired rule is removed from

the agenda. It is defined by the program:

```
once( Name, [Name|Names], Names ).
once( Name, [Head|Names], [Head|Newnames]) :-
   once( Name, Names, Newnames ).
```

---

possibles(+Name,+Names,-Newnames)

This is a built-in program for re-organising the rule agenda after each cycle of the forward chaining engine. It removes unsatisfied rules from the agenda, and is defined by the program:

```
possibles( Name, Names, [Name|Newnames] ) :-
   append( _, [Name|NewNames], Names ).
```

---

prove(+Goal)

Dereference each of the arguments of the *`Goal`* and then try to prove it.

If there is a Prolog program for the *`Goal`*, then that program is run, otherwise the workspace is searched for a matching fact (see `new_fact/1`), and in addition a check is made that there are no matching exceptions (see `new_exception/1`).

Example

```
?- prove( likes( X, @(father,peter)) )
X = mary

?- prove( likes( mary, fred) ).
no
```

where `fred` is not the `father` of `peter`.

---

reconsult_rules(+FileName)

Reconsult a flex ksl file.

---

remember_exception(+Term)

Add the given *`Term`* as a new exception in the workspace, which can be accessed through the `prove/1` and `disprove/1` procedures.

If subsequent backtracking happens, then the asserted exception will be removed.

Example

```
remember_exception( likes(mary,fred) ) .
```

---

remember_fact(+Term)

Add the given *Term* as a new fact in the workspace, which can be accessed through the `prove/1` and `disprove/1` procedures.

If subsequent backtracking happens, then the asserted fact will be removed.

Examples

```
remember_fact( male( fred ) ) .

remember_fact( likes( pamela, david ) ) .
```

---

remove_constraints

Clear from the workspace all constraints.

---

remove_constraints(+Attribute,+Frame)

Clear from the workspace of all constraints associated with the *Attribute* of a *Frame*.

Example

```
remove_constraints( contents, _ ) .
```

---

remove_data

Clear the workspace of all data directives.

---

remove_defaults

Clear from the workspace all currently defined default values.

---

remove_defaults(+Frame)

Clear the workspace of all the default values for all the attributes of a particular *Frame*.

Example

```
remove_defaults( whale ) .
```

---

remove_demons

Clear from the workspace all demons.

---

remove_demons(+Attribute,+Frame)

Clear the workspace of all demons associated with a particular *Attribute* of a *Frame*.

Examples

```
remove_demons( _, global ) .
remove_demons( contents, _ ) .
```

---

remove_exceptions

Clear the workspace of all known exceptions.

---

remove_facts

Clear the workspace of all known facts.

---

remove_frame(+Name)

Remove the named frame and all of its associated values (both current and default) and any specialized inheritance links.

Example

```
remove_frame( mammal ) .
```

---

remove_frames

Clear all currently defined frames from the *flex* workspace.

---

remove_function(+Name)

Remove the named function from the workspace.

---

remove_functions

Clear all functions from the workspace.

---

remove_groups

Clear the workspace of all known groups.

---

remove_instance(+Instance)

Clear from the workspace this specific *Instance* of some frame, together with its associated current values, and any specialized inheritance links.

Example

```
remove_instance( moby_dick ) .
```

---

remove_instances

Clear from the workspace all currently defined instances of all frames.

---

remove_instances(+Frame)

Clear the workspace of all known instances of a particular *Frame*. This will not only remove the links between the instances and the *Frame*, but will also remove any existing slots for those instances.

Example

```
remove_instances( whale ) .
```

---

remove_launches

Clear all known launch procedures from the workspace.

---

remove_launches(+Frame)

Clear the workspace of all launches associated with a particular *Frame*.

Example

```
remove_launches( employee ) .
```

---

remove_links

Clear from the workspace *all* currently defined specialized inheritance links.

---

remove_links(+Frame)

Remove the specialized inheritance links for a particular *Frame*. This means that inheritance will revert to the default inheritance of the existing frame hierarchy.

Example

```
remove_links( moby_dick ) .
```

---

remove_logic(+Logic)

Remove the named *Logic* from the workspace.

Example

```
remove_logic( inherit ) .
```

---

remove_logics

Clear the workspace of all specialized logics.

`remove_questions`

Clear the workspace of all known questions.

`remove_relation( +Name, +Arity )`

Abolish the relation *`Name/Arity`* from the workspace.

`remove_relations`

Abolish all known relations from the workspace.

`remove_rules`

Clear the workspace of all known rules.

`remove_slots`

Clear all current values from the workspace.

`remove_slots( +Frame )`

Clear the workspace of all known slots (i.e. current values) for a particular *`Frame`* (and instances of the frame).

Example

```
remove_slots( moby_dick ) .
```

`remove_synonyms`

Clear the workspace of all known synonyms.

`remove_templates`

Clear the workspace of all known templates.

`remove_templates( +Name )`

Clear the workspace of all templates, both positive and negative, for the given *Name*.

Example

```
remove_templates( ontopof ).
```

`remove_watchdogs`

Clear from the workspace all watchdogs.

---

remove_watchdogs(+Attribute,+Frame)

Clear from the workspace all watchdog procedures associated with an *Attribute* of a *Frame*.

Example

```
remove_watchdogs( _, account ) .
```

---

restart

Clear the workspace of all instances, slot values, all facts and all exceptions, re-enable all rules and finally run all data directives. It is defined by the program:

```
restart :-
   remove_instances,
   remove_slots,
   remove_facts,
   remove_exceptions,
   enable_rules,
   run_data.
```

---

run_data

Invoke the actions associated with all data directives. If any action fails then a message to indicate this is written to the current output stream.

---

run_data(+Name)

Invoke the action associated with the data *Name*. If the action fails then a message to indicate this is written to the current output stream.

Example

```
run_data( start_up_configuration ) .
```

---

some_instance(+Name,?Element)

Retrieve (or check) an *Element* which is an instance of *Name*. *Name* can be the name of a group, the name of a frame or an instance or the name of a unary relation (e.g. a type predicate in Prolog).

Examples

```
?- some_instance( colours, E ).
E = black
E = blue
E = green
etc.
```

```
?- some_instance( whale, E ).
E = moby_dick
```

---

spied_chain

Test whether or not the forward chaining engine has a spypoint set on it.

---

spied_fact(?Name)

Test whether or not a spypoint is currently set.

Example

```
?- spied_fact( X ).
X = dislikes
```

---

spied_rule(?Name)

Test whether or not the named rule currently has a spypoint set on it.

Example

```
?- spied_rule( prescribe_lomotil ).
no
```

---

spied_slot(?Attribute, ?Frame)

Check whether or not a there is a spypoint on the ***`Attribute`*** and ***`Frame`***.

A call to `spied_slot/2` will backtrack through all currently spied slots and unify ***`Attribute`*** with the name of the attribute and ***`Frame`*** with its frame.

The call fails if there are no current spypoints set.

Example

```
?- spied_slot( A, F) .
A = habitat, F = moby_dick
etc.
```

---

spy_chain

Set a spypoint on the forward chaining engine. Whenever there is a call to `forward_chain/[5,6,7]` certain information will be written to the current output stream. This information relates to the termination criterion and the selection phase.

Furthermore, if any spypoints have been set on individual rules and those rules are at some stage considered for selection, the conditions of those rules will be traced by the debugger.

---

spy_fact(?Name)

Whenever a fact or an exception whose principal functor is ***Name*** is either added to or removed from the workspace, this information is written to the current output stream.

In particular, if ***Name*** is a variable then all fact and exception transactions will be monitored.

Examples

```
spy_fact( likes ) .
```

```
spy_fact( dislikes ) .
```

---

spy_rule(?Name)

Set a spypoint on the named rule. Whenever the forward chaining engine is being traced (see `spy_chain/0`) and ***Name*** is considered as a potential rule to fire, then the conditions associated with that rule are traced by the debugger.

Example

```
spy_rule( prescribe_lomotil ) .
```

---

spy_slot(?Attribute, ?Frame)

Set a spypoint on an ***Attribute*** of a ***Frame***, such that whenever the slot is updated the relevant information is displayed in the current output stream.

Examples

```
spy_slot( habitat, moby_dick ).
```

```
spy_slot( _, albino ).
```

```
spy_slot( temperature, _ ).
```

```
spy_slot( _, _ ).
```

---

sub_value(+Slot,+Term)

`sub_value/2` can either be used to remove items from lists or to decrement numbers.

The given ***Term*** is first dereferenced down to some ***Value***. The ***Slot*** should either be a compound term of the form `@(`***Attribute***`,`***Frame***`)` or it is the name of a global variable, in which case the name of the frame to be used is `global`.

If the existing slot value and *Value* are both numbers, then*Value* is subtracted from the existing slot to form the new slot.

If the existing slot value and *Value* are both lists, then each member of the *Value* list is removed from the existing slot list.

If the existing slot is a list, but *Value* is not, then the *Value* itself is removed from that list.

Otherwise, the call to `sub_value/2` will fail.

Examples

```
?- sub_value( @(colour,flag), [cyan] ).
yes

?- sub_value( running_total, 3 ).
yes

?- sub_value( @(contents,jugA), 10 ).
yes
```

---

trigger_rule(+Name)

If *Name* is currently not a disabled rule and its conditions are satisfied, then the action part of the rule is fired.

Examples

```
trigger_rule( prescribe_lomotil ).

trigger_rule( fill_B_from_A ) .
```

10. Example - Robbie Goes Shopping

We will now consider a *flex* example, using the KSL, adapted from the book Artificial Intelligence by Patrick Henry Winston (published by Addison Wesley), page 169.

The Problem

Robbie is a robot who bags groceries in the following way:

- ask the user to choose their initial shopping list.
- check that the list is 'complete' according to the requirements :
  - if any snacks are on the list then there is also a drink.
  - if butter is on the list then so is bread, and vice versa.
- pack the items on the shopping according to the requirements :
  - pack any frozen items first.
  - pack the large items, beginning with any large bottles.
  - pack all the medium items.
  - finally, pack any remaining small items.

We can consider the solution to this problem in two phases:

1 The configuration phase - in which the initial shopping list is established.and then checked to see if everything necessary has been included. The shopping list is then adjusted accordingly.

2 The resource allocation phase - in which the items on the final shopping list are packed into bags in terms of their size and type.

The Configuration Section

The initial shopping list is established and then checked, to see if everything necessary has been included. The shopping list is then adjusted accordingly.

The Shopping Question

To obtain the initial shopping list, we ask the user to choose some items from the range of goods available.

It is convenient to group the goods under a single name which may then be referenced by a question.

```
group goods
   bread, butter, coffee, ice_cream, crisps,
   salted_peanuts, beer, lemonade, washing_powder .
```

The following question may then be written.

```
question shopping
   What is on your shopping list today ? ;
   choose some of goods
   because I must know your choice to continue .
```

The first line of the question is the actual text to be displayed when the question is asked.

The second line indicates that the question will take the form of a multiple-choice menu containing all the items in the **group** goods.

The third line contains the text to be displayed if the user enquires *why* the question is being asked.

The Compatibility Rules

Having obtained the initial shopping list from the user, we will use forward chaining rules to check the compatibility of the items on the *initial* shopping list to create the *final* shopping list.

The set of goods needs to be differentiated further in the terms of the compatibility criteria, i.e. if there are snacks on the shopping list then there must also be a drink.

For convenience we define the two groups for snacks and drinks.

```
group snacks
   crisps, salted_peanuts .

group drink
   lemonade, beer .
```

We can now write the rules for the compatibility checks.

We need a rule to ensure that if the shopping includes some snacks there is

also a drink on the list. This can be thought of as follows.

If the shopping includes some snacks and does not include any drinks, then ask the user to choose a drink and include that drink on the shopping list.

The rule can be written in KSL in the following way.

```
rule check_snacks
   if   the shopping includes some snacks
   and  the shopping does not include some drink
   then ask drink
   and  include the drink in the shopping
   and  write( 'You chose ' )
   and  write( drink )
   and  write( ' to have with your snack' ) and nl
   because Snacks make you thirsty .
```

The fourth line of the rule forces the user to answer the `drink` question.

The `drink` question can be defined as follows.

```
question drink
   You must select a drink ! ;
   choose one of drink
   because There are snacks on your shopping list .
```

This time, the user will be offered a single-choice menu comprising all the items in the `drink` group. Note that we have re-used the same label for both the group and the question. This is a matter of style, but is optional.

The second compatibility requirement concerns the inclusive linkage of bread and butter. We can think of this as follows.

If the shopping list contains bread and does not contain butter then include butter on the shopping list.

This can be written in KSL as follows

```
rule check_bread_and_butter
   if the shopping includes bread
   and  the shopping does not include butter
   then include butter in the shopping
      and  write( 'Including butter on shopping list' )
      and nl
   because Bread is very dry by itself .
```

The following rule needs to be written to cover the alternate condition where the shopping list contains butter and not bread.

```
rule check_butter_and_bread
   if the shopping includes butter
   and  the shopping does not include bread
   then include bread in the shopping
```

```
      and  write( 'Including bread on shopping list' )
      and nl
   because What is the use of butter without bread .
```

Having written the compatibility rules, we need to determine how they will be used.

Because there is no conflict between these rules (i.e. they do not compete by having overlapping conditions) the order in which they are used is not important. It is therefore best to select them on a first come first served basis.

Also, once each rule has been selected and fired, its conditions can no longer apply. In this case the most efficient way of updating the ruleset is by removing each selected rule, which is equivalent to firing each rule once only.

The ruleset that governs the checking rules would be written in KSL as follows.

```
ruleset checking_rules
   contains  check_snacks,
             check_bread_and_butter,
             check_butter_and_bread ;
   select rule using first come first served ;
   update ruleset by removing each selected rule .
```

The Resource Allocation Section

In this section the items on the final shopping list are packed into carrier bags in terms of their type, container and size .

The Packing Rules

The packing is to be done according to container type and size. Therefore we can classify the data as follows.

The main difference between each of the goods is their size, which can be either large, medium or small.

The packing requirements are that only three medium and small items are allowed in a bag and only one large item is allowed in any bag.

```
frame item
   default maximum_allowed is 3 .

frame large_item is an item
   default size is large
   and default maximum_allowed is 1 .
```

```
frame medium_item is an item
   default size is medium .

frame small_item is an item
   default size is small .
```

The `medium_item` and `small_item` frames automatically inherit 3 as the value for `maximum_allowed`, whereas the `large_item` frame overrides this value with 1.

The following KSL sentences define the characteristics of each shopping item.

```
frame bread is a medium_item
   default container is a plastic_bag and
   default condition is fresh .

frame butter is a small_item
   default container is a plastic_carton and
   default condition is fresh .

frame coffee is a medium_item
   default container is a jar and
   default condition is freeze_dried .

frame ice_cream is a medium_item
   default container is a cardboard_carton and
   default condition is frozen .

frame crisps is a small_item
   default container is a plastic_bag and
   default condition is fragile .

frame salted_peanuts is a small_item
   default container is a plastic_bag and
   default condition is salted .

frame beer is a large_item
   default container is a bottle and
   default condition is liquid .

frame lemonade is a large_item
   default container is a bottle and
   default condition is liquid .

frame washing_powder is a large_item
   default container is a cardboard_carton and
   default condition is powder .
```

The KSL rules for packing the items into bags are hopefully self explanatory.

```
rule pack_frozen_item
   if the shopping includes an Item
   and  the Item`s condition is frozen
   then pack_item( Item ) .

rule pack_large_bottle
   if the shopping includes an Item
   and  the Item`s container is a bottle
   and  the Item`s size is large
   then pack_item( Item ) .

rule pack_other_large_item
   if the shopping includes an Item
   and  the Item`s size is large
   then pack_item( Item ) .

rule pack_medium_item
   if the shopping includes an Item
   and  the Item`s size is medium
   then pack_item( Item ) .

rule pack_small_item
   if the shopping includes an Item
   and  the Item`s size is small
   then pack_item( Item ) .
```

The initial rule agenda should be structured to reflect the order in which the rules are meant to be fired (i.e. check for frozen items first, then large bottles etc.).

We note that if the conditions of a rule are not satisfied, they will not become satisfied at any later stage. In this case the most efficient way to update the rule agenda is to remove any unsatisfied rules.

The ruleset which governs this use of the forward chaining rules is defined in the following way. The order and contents of the initial rule agenda is determined in the ruleset, by the part indicated by the KSL keyword `contains`.

```
ruleset packing_rules
   contains  pack_frozen_item,
             pack_large_bottle,
             pack_other_large_item,
             pack_medium_item,
             pack_small_item ;
   select rule using first come first served ;
   update ruleset by removing any unsatisfied rules .
```

Packing The Items

We do not know in advance how many carrier bags will be needed when packing the groceries. So, each carrier bag must be generated as and when

required. The following frame is used as a mould for generating new carrier bags.

```
frame carrier
   default contents are empty .
   default contents_count is 0 .

demon contents_counter
   when the contents of Bag changes
   and Bag is an instance of carrier
   then add 1 to the contents_count of Bag .
```

This demon will be activated whenever we change the `contents` of any `carrier`.

Packing an item can be thought of as follows.

To pack an item, first choose a carrier bag for the item, then remove the item from the shopping list and add the item to the contents of the bag.

```
action pack_item( Item );
   if choose_bag( Bag, Item )
   and  remove the Item from the shopping
   and  include the Item in the contents of the Bag
   and  write( 'Packing ' ) and write( Item )
   and write( ' into ' ) and write( Bag ) and nl .
```

We now need to establish how to choose a carrier bag for an item. This may be done using a backward chaining relation, which can be thought of as follows.

To choose a bag for an item, get an existing carrier bag, and check that the number of items already in the bag is less than the maximum allowed for the item about to be packed.

The first clause of the `choose_bag` relation is:

```
relation choose_bag( Bag, Item )
   if the Bag is some carrier
      whose contents_count is less than
      the maximum_allowed of Item .
```

Whenever a carrier bag is needed it is always better to check existing bags (indicated by the keywords `Bag is some carrier`) if at all possible.

The second clause of the `choose_bag` relation is

```
relation choose_bag( Bag, Item )
   if  gensym( bag_number_ , Bag )
   and  Bag is a new carrier .
```

Note that `gensym/2` is a built-in predicate of the underlying Prolog system for generating unique names from a given root symbol (`bag_number_` in

this case).

The Initial Goal

Having defined all the parts of the program, we now need to put them together into an initial goal that starts the whole process going.

```
action start_robot ;
   do  restart_robot
   and get_shopping
   and check_shopping
   and pack_shopping .

action restart_robot ;
   do restart
   and init_gensym(bag_number_).
```

The relation `restart/0` is a built-in *flex* predicate for removing all current slot values, and thus re-instating any default slot values.

The relation `init_gensym/1` is expected to be a predicate from the underlying Prolog system for resetting the given root symbol (`bag_number_` in this case) to zero.

```
action get_shopping ;
   do ask shopping
   and write( 'Initial shopping list... ' )
   and write( shopping ) and nl.
```

The action `get_shopping` asks the shopping question and then echoes to the screen the user's choice.

The actions `check_shopping` and `pack_shopping` initiate the appropriate forward chaining cycles and tell the user what is going on at each stage of the problem.

```
action check_shopping ;
   do write( 'Compatibility checking phase' )
   and invoke ruleset checking_rules
   and write( 'Final shopping list... ' )
   and write( shopping ) and nl.

action pack_shopping ;
   do write( 'Packing phase' )
   and invoke ruleset packing_rules
   and for every Bag is some carrier
         do write( 'Contents of ' )
         and write( Bag )
         and write( the contents of Bag ) and nl
   end for .
```

Templates

Having finished the example, you may feel the `choose_bag( Bag, Item )` statements are awkward to use within the code. We can now define templates to replace them with more elegant statements.

Example

```
template choose_bag
   choose a carrier ^ for ^ .
```

This template definition will allow us to replace *every* occurrence of

```
   choose_bag( Bag, Item )
```

with the statement

```
   choose a carrier Bag for Item
```

so making the code easier to read.

For example, we could rewrite the `pack_item` action using this template as follows.

```
action pack_item( Item );
   if choose a carrier Bag for Item
   and remove the Item from the shopping
   and include the Item in the contents of the Bag
   and write( 'Packing ' ) and write( Item )
   and write( ' into ' ) and write( Bag ) and nl .
```

Appendix A - Examples

This appendix includes two examples used to illustrate some of the techniques involved in writing *flex* programs.

Example 1 is an animal taxonomy, which is used to demonstrate some of the features of the frame system of *flex*.

Example 2 is a forward chaining solution to the Water Jugs problem, which demonstrates the use of rule transition networks to update the rule agenda used in forward chaining.

Example 1 - Analysing a Taxonomy

The purpose of this example is to illustrate most of the concepts within the frame sub-system of *flex*. This involves the representation (using constructs from the Knowledge Specification Language) of a taxonomy for the animal kingdom, together with an algorithm for identifying species. We then give a couple of example questions that can be posed.

As in the rest of this manual, in the following examples the KSL keywords are emboldened for clarity (they should not be distinguished in source texts).

The Animal Kingdom

The animal kingdom can be split into three main groups, namely mammal, fish and bird. Each of these main groups has its own set of characteristics which separate it from the others. The following KSL statements describe this frame hierarchy, together with the associated default values.

```
frame animal
   default blood is warm .

frame mammal is an animal
   default skin is fur and
   default habitat is the land and
   default motions are walk .

frame fish is an animal
   default skin is scale and
   default habitat is water and
   default motions are { swim } and
```

```
   default blood is cold .

frame bird is an animal
   default skin is feather and
   default habitat is a tree and
   default motions are { walk or fly } .
```

This is the most basic form of taxonomy, whereby each sub-class (`mammal`, `bird` and `fish`) has a single parent class (`animal`). Furthermore, the `blood` characteristic of an `animal` can be inherited by all three sub-classes.

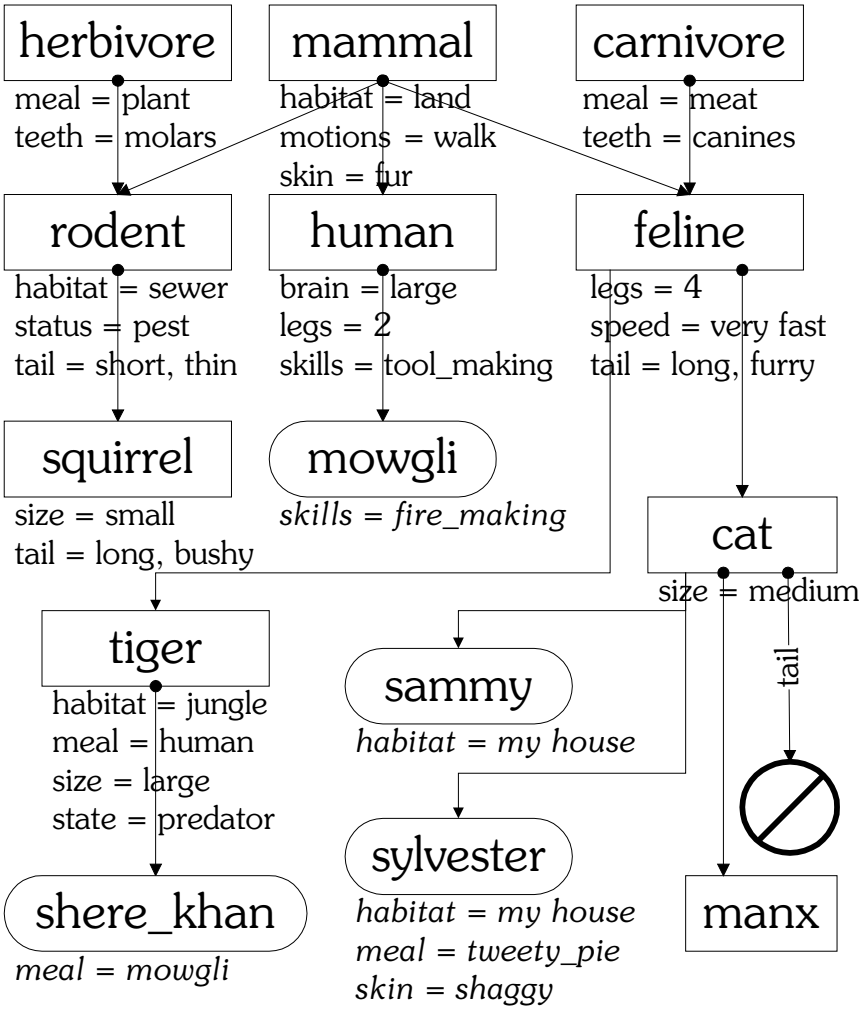
Fish can inherit the characteristic that their `blood` is warm even though there is a local default to the contrary. In effect, there are two solutions to the query :

Q: What temperature is the blood of a fish ?

A: Firstly, the blood of a fish is cold (from fish itself)
A: Secondly, the blood of a fish is warm (from animal)

Representing Mammals

The following diagram illustrates a few of the different types of mammals, together with some of their characteristics.



The following KSL statements describe the frame hierarchy above, together with the associated default values.

```
frame carnivore
   default meal is meat and
   default teeth are canines .

frame herbivore
   default meal is plant and
   default teeth are molars .
```

Note that carnivores and herbivores have no parent frames. Their role is merely to define a set of default characteristics.

```
frame feline is a mammal , carnivore
   default legs are 4 and
   default speed is 'very fast' and
   default tail is { long and furry } .

frame rodent is a mammal , herbivore
   default habitat is sewer and
   default status is pest and
   default tail is { short and thin } .

frame human is a mammal
   default brain is large and
   default legs are 2 and
   default skills are { tool_making } .

frame cat is a feline
   default size is medium .

frame tiger is a feline
   default habitat is jungle and
   default meal is human and
   default state is predator and
   default size is large .

frame squirrel is a rodent
   default size is small and
   default tail is { long and bushy } .
```

Unlike most other rodents, whose tails are generally short and thin, the `tail` of a squirrel is `long` and `bushy`.

```
frame manx is a cat ;
   do not inherit tail .
```

Unlike other cats, a manx does not have a tail. This is represented by not inheriting the `tail` attribute.

```
instance sammy is a cat ;
   habitat is my_house ;
   inherit meal from herbivore .
```

The instance called `sammy` illustrates the technique of overriding the frame hierarchy for a particular attribute. `sammy` will inherit its meal type from `herbivore`.

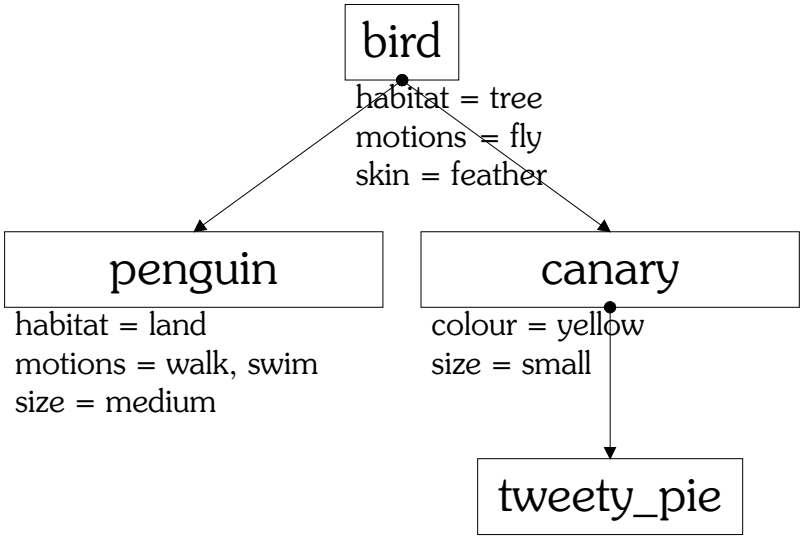
```
instance sylvester is a cat ;
   habitat is my_house ;
   meal is tweety_pie ;
   skin is shaggy .

instance shere_khan is a tiger ;
   meal is mowgli .

instance mowgli is a human ;
   skills are { fire_making } .
```

Representing Birds

The following diagram illustrates the bird taxonomy.



This can be represented by the KSL statements :-

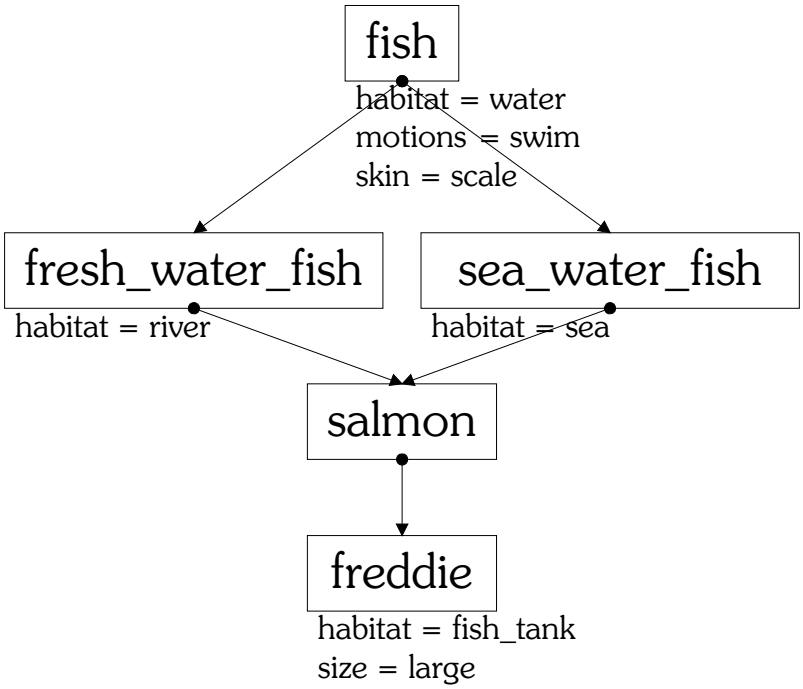
```
frame penguin is a bird
   default habitat is the land and
   default motions are { walk and swim } and
   default size is medium .

frame canary is a bird
   default colour is yellow and
   default size is small .

instance tweety_pie is a bird .
```

Representing Fish

The following diagram illustrates the two major sub-classes of fish, namely sea water fish and fresh water fish.



which is represented by the following KSL sentences :

```
frame 'sea water fish' is a kind of fish
   default habitat is the sea .

frame 'fresh water fish' is a kind of fish
   default habitat is a river .

frame salmon is a sea_water_fish, fresh_water_fish .

instance freddie is a salmon ;
   habitat is fish_tank and
   size is large .
```

The order of the parents for the `salmon` frame is important, as this determines the order in which values are potentially inherited. For example, the habitat of a `salmon` is firstly the `sea` (from `'sea water fish'`), secondly a `river` (from `'fresh water fish'`) and thirdly `water` (from `fish` itself).

The Identification Algorithm

The previous sections have described a small section of the animal kingdom, including the inter-relationships of different species (the frame hierarchy) and some of their characteristics (default values).

The following KSL program can be used to identify a particular species within the example. Note, though, that the algorithm itself is not specific to the example, but could equally be used for identifying classes within any taxonomy.

```
action run(Frame,PosAttribs,NegAttribs)
   do [ isa_frame( Frame, _ ) or isa_instance(Frame,_)
]
   and for every PosAttr-PosVal is included in
PosAttribs
        do check [ the PosAttr of Frame includes PosVal
             or the PosAttr of Frame is equal to PosVal
                  ]
        end for
  and for every NegAttr-NegVal is included in
NegAttribs
        do check [ the NegAttr of Frame does not
include
                  NegVal and the
                  NegAttr of Frame is not equal to
NegVal
                  ]
  end for .
```

Given a set of positive clues and a set of negative clues, the algorithm will identify any class which has each of the positive attributes but none of the negative attributes.

`isa_frame/2` and `lookup/3` are built-in *flex* predicates for retrieving the names of frames and for looking up values of frame attributes, respectively. The `lookup/3` procedure will automatically take into account any inheritance through the frame hierarchy.

Some Example Questions

Using the identification algorithm above we can ask, for example, the following two questions :

- Which type of animal eats meat ?

- Which type of animal lives on land, is medium in size but does not eat meat ?

These questions are posed by the following Prolog queries.

```
?- identify( Class, [meal-meat], [] ).
Class = carnivore
Class = feline
Class = cat
Class = tiger
Class = manx

?- identify( Class,
             [habitat-land,size-medium],
             [meal-meat] )
Class = penguin
```

Example 2 - The Water Containers

The purpose of this example is to illustrate the conflict resolution algorithm for selecting rules from agendas.

The Problem

There are two water containers of differing sizes, one of which is called the master (capacity 7 litres) and the other is called the slave (capacity 4 litres). Water, or indeed any liquid, can be transferred between the two by emptying one into the other, or by filling one from the other. In addition, the containers can be either filled completely from a reservoir, or emptied completely into a sink.

The problem is to find a sequence of operations which will result in the master containing exactly 6 litres.

The Containers

The three principal attributes of a container are its contents, its capacity and its spare capacity (being the difference between capacity and contents). We shall use a template to assist the readability of the attribute `spare` capacity.

```
template spare
   spare capacity .
```

The default size of a container is 7 litres, and its initial contents are empty.

```
frame container
   default contents are 0 and
   default capacity is  7 and
   default spare capacity is
             its capacity minus its contents .
```

The master is a regular container, but the capacity of the slave is only 4 litres.

```
instance master is a container .

instance slave is a container ;
   capacity is 4 .
```

The Tests

We shall declare some templates to assist the readability of the tests.

```
template non_full
   ^ is not full .

template non_empty
   ^ is not empty .

template enough
   ^ contains more than ^ ;
   ^ does not contain more than ^ .
```

which are defined by the following relationships.

```
relation X is not full
   if X`s contents are below X`s capacity .

relation X is not empty
   if X`s contents are above 0 .

relation X contains more than Z
   if X`s contents are above Z .
```

The Operations

Again, we shall declare some templates to assist the readability of the operations.

```
template full_up
   fill up ^ .

template empty_out
   empty out ^ .

template fill_from
   fill ^ from ^ .

template empty_into
   empty ^ into ^ .
```

The two operations for transferring water from one container to the other are to completely empty the contents of one into the other, and to completely fill one from the other.

```
action fill up X ;
   do X`s contents become X`s capacity .

action empty out X ;
   do X`s contents become 0 .
```

```
action fill X from Y ;
   do subtract X`s spare capacity from Y`s contents
   and fill up X .

action empty X into Y ;
   do add X`s contents to Y`s contents
   and empty out X .
```

The Rules

There are 4 rules governing the circumstances when the master container can be changed, and 4 similar rules for the slave container.

The scores attached to each rule are not arbitrary, but were reached through a phase of trial and error. By keeping the rules unchanged, but adjusting their relative scores, different behaviours were observed. The final scoring system is designed to reach a state in which the master contains 6 litres.

```
rule fill_master
   if the master is not full
   then fill up the master
   score master`s spare capacity .

rule empty_master
   if the master is not empty
   then empty out the master
   score master`s contents .

rule fill_master_from_slave
   if  the master is not full
   and the slave contains more than
        the master`s spare capacity
   then fill the master from the slave
   score master`s contents + slave`s contents .

rule empty_master_into_slave
   if the master is not empty
   and the master does not contain more than
        the slave`s spare capacity
   then empty the master into the slave
   score  master`s capacity - slave`s contents .


rule fill_slave
   if the slave is not full
   then fill up the slave
   score slave`s spare capacity .

rule empty_slave
   if the slave is not empty
   then empty out the slave
   score slave`s contents .
```

```
rule fill_slave_from_master
   if the slave is not full
   and the master contains more than
          the slave`s spare capacity
   then fill the slave from the master
   score slave`s contents + master`s contents .

rule empty_slave_into_master
   if the slave is not empty
   and the slave does not contain more than
          the master`s spare capacity
   then empty the slave into the master
   score slave`s capacity - master`s contents .
```

The Rule Network

The problem can be further refined by noting that, following each operation, it only makes sense for a subset of the other operations to be considered. For example, having filled a container it does not make sense to then empty it, as you may as well have emptied it in the first place.

The following definitions state which operations can follow which other operations. Note that each `group` name is also the name of a rule, and the group contains the names of the rules which can follow it.

```
group fill_master
   empty_master_into_slave, fill_slave_from_master,
   fill_slave, empty_slave .

group empty_master
   fill_master_from_slave, empty_slave_into_master,
   fill_slave, empty_slave .

group fill_master_from_slave
   empty_master, fill_slave, empty_slave .

group empty_master_into_slave
   fill_master, fill_slave, empty_slave .

group fill_slave
   empty_slave_into_master, fill_master_from_slave,
   fill_master, empty_master .

group empty_slave
   fill_slave_from_master, empty_master_into_slave,
   fill_master, empty_master .

group fill_slave_from_master
   empty_slave, fill_master, empty_master .

group empty_slave_into_master
   fill_slave, fill_master, empty_master .
```

```
group start_rules
   fill_master, fill_slave .
```

The Ruleset

The problem starts with both jugs being empty, we can therefore limit the initial rule agenda to two rules: the rule that fills the master jug and the rule that fills the slave jug. For convenience we have put these rules in a group called `start_rules`. The rule agenda will subsequently be updated using the rule transition network that we defined above.

We have attached scores to the rules to give them different chances of being selected according to the situation. The rules will then be selected using the conflict resolution scoring system with a threshold of 6. This means that the *first* rule found whose score is above 6 will be fired.

The solution to the problem will be found (so we want the forward chaining engine to terminate) if at any point the contents of any container are 6.

The ruleset that defines this method of selecting rules, updating the rule agenda and terminating the forward chaining is written as follows.

```
ruleset water_jugs
   contains start_rules ;
   select rule using
             conflict resolution with threshold 6 ;
   update ruleset using rule transition network ;
   initiate by doing restart ;
   terminate when
             the contents of any container are 6 .
```

The action that triggers the whole program can then be written as follows. (The `spy_chain/0` built-in *flex* predicate causes execution information to be written to the current output stream.)

```
action demo ;
     do spy_chain
     and invoke ruleset water_jugs .
```

Appendix B - Formal Definition of KSL

In this appendix we formally describe the syntax of the *K*nowledge *S*pecification *L*anguage. We use a Definite Clause Grammar (DCG) as its definition. Reserved words of the language are emboldened.

The three principal components of any language are *terms* (descriptions of objects of the world being defined), *formulae* (concepts and operations over such objects) and *sentences* (valid statements which relate the concepts and operations). The DCG below defines the sentences, formulae and terms of the KSL. This is preceded by some general grammar structures used throughout those definitions.

Grammatical Structures

Optional Structures

```
optional( Grammar_Rule ) -->
        Grammar_Rule .

optional( Grammar_Rule ) -->
        [] .
```

Disjunction & Conjunction

```
disjunction( Grammar_Rule ) -->
        sequence_separator( conjunction( Grammar_Rule ), [or] ) .

conjunction( Grammar_Rule ) -->
        sequence_separator( Grammar_Rule, [and] ) .
```

Sequences

```
sequence( Grammar_Rule ) -->
        Grammar_Rule,
        sequence( Grammar_Rule ) .

sequence( Grammar_Rule ) -->
        [] .

sequence_separator( Grammar_Rule, Separator ) -->
        Grammar_Rule,
        separator_continuation( Grammar_Rule, Separator ) .

separator_continuation( Grammar_Rule, Separator ) -->
        Separator,
    Grammar_Rule,
```

```
        separator_continuation( Grammar_Rule, Separator ) .

separator_continuation( Grammar_Rule, Separator ) -->
        [] .
```

KSL Sentences

In this_section we describe the valid sentences in which KSL objects and formulae can occur. They constitute what can and cannot be stated in a KSL program.

Note that each sentence begins with a keyword to identify its category, is usually followed by a unique identifier within that category (note that the same identifier can be used in different categories), and is always terminated by a full-stop.

The categories of KSL sentences are :

- Frame
- Instance
- Launch
- Demon_
- Constraint_
- Watchdog_
- Rule_
- Ruleset_
- Action_
- Relation_
- Function_
- Command_
- Data_
- Question_
- Group_
- Synonym_
- Template

Frame

```
frame -->
        [frame],
        name( frame ),
        optional( parent_frames ),
        optional( default_values ),
        optional( inheritance_links ),
        [.] .

    parent_frames -->
        ako,
```

```
        sequence_separator( name( parent ), [,] ) .

    ako -->
        [is,a,kind,of]          |
        [is,an,instance,of]     |
        [is,a]                  |
        [is,an] .

    default_values -->
        [;],
        sequence_separator( default_value, [and] ) .

    default_value -->
        [default],
        name( attribute ),
        equate,
        expression .

    equate -->
        [is]        |
        [are] .

    inheritance_links -->
        [;],
        sequence_separator( inheritance_link, [and] ) .

    inheritance_link -->
        [inherit],
        name( attribute ),
        [from],
        sequence_separator( name( frame ), [,] ) .
```

Instance

```
instance -->
        [instance],
        name( instance ),
        ako,
        name( parent_frame ),
        optional( initial_values ),
        optional( inheritance_links ),
        [.] .

    ako -->
        [is,a,kind,of]          |
        [is,an,instance,of]     |
        [is,a]                  |
        [is,an] .

    initial_values -->
    [;],
```

```
        sequence_separator( initial_value, [and] ) .

    initial_value -->
        name( attribute ),
        equate,
        expression .

    equate -->
        [is]        |
        [are] .

    inheritance_links -->
        [;],
        sequence_separator( inheritance_link, [and] ) .

    inheritance_link -->
        [inherit],
        name( attribute ),
        [from],
        sequence_separator( name( frame ), [,] ) .
```

Launch

```
launch -->
        [launch],
        name( launch ),
        [when],
        variable( instance ),
        isa_new,
        name( parent_frame ),
        optional( context ),
        [then],
        conjunction( directive ),
        [.] .

    isa_new -->
        [is,a,new,instance,of]      |
        [is,a,new] .

    context -->
        [and],
        disjunction( condition ) .
```

Demon

```
demon -->
        [demon],
        name( demon ),
        [when],
        trigger,
        change,
        optional( from_value ),
        optional( to_value ),
        optional( context ),
        [then],
        conjunction( directive ),
        [.] .

    trigger -->
        variant( schema ) .

    change -->
        [changes]   |
        [change] .

    from_value -->
        [from],
        expression .

    to_value -->
        [to],
        expression .

    context -->
        [and],
        disjunction( condition ) .
```

Constraint

```
constraint -->
        [constraint],
        name( constraint ),
        [when],
        trigger,
        change,
        optional( from_value ),
        optional( to_value ),
        optional( context ),
        [then,check],
        disjunction( condition ),
        optional( otherwise ),
        [.] .

     trigger -->
```

variant(schema) .

change -->
[**changes**] |
[**change**] .

from_value -->
[**from**],
expression .

to_value -->
[**to**],
expression .

context -->
[**and**],
disjunction(condition) .

otherwise -->
[**otherwise**],
conjunction(directive) .

Watchdog

watchdog -->
[**watchdog**],
name(watchdog),
[**when**],
trigger,
request,
optional(context),
[**then**,**check**],
disjunction(condition),
optional(otherwise), [.] .

trigger -->
variant(schema) .

request -->
[**is**,**requested**] |
[**are**,**requested**] .

context -->
[**and**],
disjunction(condition) .

otherwise -->
[**otherwise**],
conjunction(directive) .

Production Rule

```
production_rule -->
        [rule],
        name( rule ),
        production_conditions,
        conjunction( directive ),
        optional( explanation ),
        optional( score ),
        [.] .

    production_conditions -->
        [if],
        disjunction( condition ),
        [then] .

    production_conditions -->
        [always,do] .

    explanation -->
        [;],
        [browse,file],
        name( file ) .

    explanation -->
        [;],
        [because],
        sequence( token ) .

    score -->
        [;],
        [score],
        expression .
```

Ruleset

```
ruleset-->
        [ruleset],
        name( ruleset ),
        [contains],
        initial_rule_agenda,
        optional( initiate ),
        optional( terminate ),
        optional( select_rule ),
        optional( update_ruleset ),
        optional( misfires ),
        [.] .

    initial_rule_agenda -->
     [all, rules] .
```

```
initial_rule_agenda -->
        sequence_separator( name(rule), [,] ) .

initiate-->
        [;],
        [initiate, by, doing],
        conjunction( directive ) .

terminate -->
        [;],
        [terminate,  when],
        conjunction( conditions ) .

select_rule -->
        [;],
        [select, rule, using],
        selection_method .

selection_method -->
        [conflict, resolution, with, threshold],
        expression .

selection_method -->
        [conflict, resolution] .

selection_method -->
        [first, come, first, served] .

selection_method-->
        name( user_method ) .

update_ruleset -->
        [;],
        [update, ruleset],
        update_method .

update_method-->
        [by, removing, each, selected, rule]   |
        [by, promoting, each, selected, rule]  |
        [by, demoting, each, selected, rule]   |
        [by, cyclic, rotation, of, rules]      |
        [by, removing, any, unsatisfied, rules]     |
        [using, rule, transition, network] .

update_method-->
        [using],
        name( user_update ) .

misfires -->
        [;],    [when, a, rule, misfires, do],
```

```
        name( user_misfire ) .
```

Action

```
action -->
        [action],
        procedure,
        optional( [;] ),
        conjunction( directive ),
        [.] .
```

Relation

```
relation -->
        [relation],
        procedure,
        optional( relation_body ),
        [.] .

    relation_body -->
        [if],
        disjunction( condition ) .
```

Function

```
function -->
        [function],
        name( function ),
        ['('],
        sequence_separator( expression, [','] ),
        [')'],
        [=],
        expression,
        optional( where_clause ),
        [.] .

    where_clause -->
        where,
        disjunction( condition ) .

    where -->
        [where]      |
        [such,that] |
        [:] .
```

Command

```
    command  -->
    [do],
    conjunction( directive ), [.]
```

Data

```
data -->
        [data],
        name( data ),
        conjunction( directive ),
        [.] .
```

Question

```
question -->
        [question],
        name( question ),
        question_form,
        [.] .

    question_form -->
        [answer,is],
        variable( answer ),
        where,
        disjunction( condition ) .

    question_form -->
        sequence( tokens ),                % this forms the text of the question
        [;],
        answer_form,
        optional( explanation ) .

    answer_form -->
        menu_form,
        menu_items .

    answer_form -->
        [input],
        optional( [is] ),
        input_form .

    menu_form -->
        [choose,from]      |
        [choose,one,of]    |
        [choose,some,of] .

    menu_items -->
        name( group ).

    menu_items -->
        sequence_separator( token, [','] ) .

    input_form -->
        [set]          |
        [name]         |
```

```
        [number] |
        [integer] .

input_form -->
        variable( answer ),
        where,
        disjunction( condition ) .

where -->
        [where]          |
        [such,that] |
        [:] .

explanation -->
        [;],
        [browse,file],
        name( file ) .

explanation -->
        [;],
        [because],
        sequence( token ) .
```

Group

```
group -->
        [group],
        name( group ),
        sequence_separator( token, [','] ),
        [.] .
```

Synonym

```
synonym -->
        [synonym],
        name( synonym ),
        expression,
        [.] .
```

Template

```
template -->
        [template],
        name( template ),
        template_form,                  % this forms the positive part of the template
        optional( negative_form ),      % this forms the negative part of the
        template
        [.] .

negative_form -->
 [;],
```

```
        template_form .

    template_form -->
        sequence_separator( sequence( token ), ['^'] ) .
```

KSL Formulae

The formulae of KSL are used to establish relationships between the objects of KSL. This falls into two distinct areas; conditions which test whether or not something is currently true, and directives which change the current state to some new state.

Condition

```
condition -->
        ['['],
        disjunction( condition ),
        [']'] .

condition -->
        [not],
        condition .

condition -->
        control_statement .

condition -->
        procedure .

condition -->
        variable( frame/instance ),
        ako,
        name( frame ),
        optional( conjunction( slot_test ) ) .

    ako -->
        [is,an,instance,of]  |
        [is,a,kind,of]       |
        [is,an]              |
        [is,a] .

    slot_test  -->
        [whose],
        name( attribute ),
        comparison .

condition -->
        expression,
        comparison .
```

Comparison

```
comparison -->
        equate,
        expression .

    equate -->
        [=]     |
        [\=] .

    equate -->
        is,
        optional( [not] ),
        equal .

    equal -->
        [equal,to]          |
        [different,from] .

    is -->
        [is]    |
        [are] .

comparison -->
        compare,
        expression,
        optional( ordering ) .

    compare -->
        [>]     |
        [>=]    |
        [<]     |
        [=<] .



    compare -->
        is,
        optional( [not] ),
        compared .

    compared -->
        [above]                         |
        [at,or,above]                   |
        [below]                         |
        [at,or,below]                   |
        [greater,than]                  |
        [greater,than,or,equal,to]      |
        [less,than]                     |
    [less,than,or,equal,to] .
```

```
    ordering -->
            [according,to],
            order .

    order -->
            name( group ).

    order -->
            ['{'],
            sequence_separator( token, [','] ),
            ['}'] .

comparison -->
            includer,
            expression .

    includer -->
            [includes]          |
            [include]                   |
            [does,not,include]  |
            [do,not,include] .

    includer -->
            is,
            optional( [not] ),
            [included,in] .

    is -->
            [is]    |
            [are] .
```

Directive

```
directive -->
            ['['],
            conjunction( directive ),
            [']'] .

directive -->
            control_statement .

directive -->
            procedure .

directive -->
            variable( instance ),
            new,
            name( frame ),
            optional( conjunction( new_slot_value ) ) .

    new -->
```

```
            [is,a,new]      |
            [is,another] .

    new_slot_value -->
            [whose],
            name( attribute ),
            is,
            expression .

    is -->
            [is]      |
            [are] .

directive -->
            variant( simple ),
            assign,
            expression .

    assign -->
            [becomes]   |
            [become] |
            [:=] .

directive -->
            [add],
            expression,
            [to],
            variant( simple ) .

directive -->
            [subtract],
            expression,
            [from],
            variant( simple ) .

directive -->
            [include],
            expression,
            [in],
            variant( simple ) .

directive -->
            [remove],
            expression,
            [from],
            variant( simple ) .

directive -->
            assertion,
            optional( [that] ),
     optional( [not] ),
```

```
            procedure .

    assertion -->
            [remember] |
            [forget] .
```

Control Statement

```
control_statement -->
            [do],
            directive .


control_statement -->
            [check],
            optional( [that] ),
            condition .

control_statement -->
            [if],
            disjunction( condition ),
            [then],
            conjunction( directive ),
            [else],
            conjunction( directive ),
            [end,if] .

control_statement -->
            [repeat],
            conjunction( directive ),
            [until],
            disjunction( condition ),
            [end,repeat] .

control_statement -->
            [while],
            disjunction( condition ),
            [do],
            conjunction( directive ),
            [end,while] .

control_statement -->
            [for],
            universal,
            disjunction( condition ),
            [do],
            conjunction( directive ),
            [end,for] .

    universal -->
            [all]      |
```

```
            [every]  |
            [each] .

control_statement -->
            [for],
            variable( counter ),
            [from],
            expression,
            [to],
            expression,
            optional( step ),
            [do],
            conjunction( directive ),
            [end,for] .

    step -->
            [step],
            expression .
```

Procedure

```
procedure -->
            [$],
            procedure .

procedure -->
            "an instance of a template declaration".

procedure -->
            [ask],
            name( question ) .

procedure -->
            name( procedure ),
            ['('],
            sequence_separator( expression, [','] ),
            [')'] .

procedure -->
            name( procedure ),
            optional( proposition ) .

    proposition -->
            [is,true]          |
            [is,false]         |
            [is,not,true]      |
            [is,not,false] .
```

KSL Objects

This section describes the actual objects of KSL (i.e. those constructs which correspond to entities in your particular domain). They range from variants which can change in value through time (by assignment) to set abstractions which portray a (possibly infinite) collection of objects.

Variant

```
variant( Type ) -->                     % Type can be simple or schema
        ['('],
        variant( Type ),
        [')'] .

variant( Type ) -->
        variant( Type ),                % attribute
        [of],                           % 'of' is right associative
        variant( Type ) .               % frame or instance

variant( Type ) -->
        variant( Type ),                % frame or instance
        [`,s],                          % '`s' is left associative
        variant( Type ) .               % attribute

variant( Type ) -->
        optional( determiner ),
        name( variant ) .

    determiner -->
        [the]     |
        [an]      |
        [a] .

variant( schema ) -->
        existential,
        name( variant ) .

    existential -->
        [some,instance,of]  |
        [any,instance,of]   |
        [some]                      |
        [any] .

variant( schema ) -->
        thing .
```

```
    thing -->
            [something]       |
            [anything]    |
            [somebody]    |
            [anybody] .
```

Set

```
implicit_set -->
            ['{'],
            variable( member ),
            where,
            disjunction( condition ),
            ['}'] .

    where -->
            [where]          |
            [such,that] |
            [:] .

implicit_set -->
            ['{'],
            disjunction( explicit_set ),
            ['}'] .

    explicit_set -->
            sequence_separator( expression, set_join ) .

    set_join -->
            [',']        |
            [and] .

implicit_set -->
            universal,
            name( group/frame/instance ),
            optional( conjunction( slot_test ) ) .

    universal -->
            [all]        |
            [every]      |
            [each] .

    slot_test -->
            [whose],
            name( attribute ),
            comparison .
```

General Term

```
term -->
    [$],
```

```
            term .

term -->
            ['('],
            expression,
            [')'] .

term -->
            variant( schema ) .

term -->
            implicit_set .

term -->
            name( functor ),
            ['('],
            sequence_separator( expression, [','] ),
            [')'] .

term -->
            [if],
            disjunction( condition ),
            [then],
            expression,
            [else],
            expression .

term -->
            variable( answer ),
            where,
            disjunction( condition ) .

    where -->
            [where]          |
            [such,that]      |
            [:] .

term -->
            existential,
            name( group/frame/instance ),
            optional( conjunction( comparison ) ) .

    existential -->
            [some,instance,of]   |
            [any,instance,of]    |
            [some]               |
            [any] .

    slot_test -->
            [whose],
            name( attribute ),
```

```
            comparison .

term -->
            owner,
            name( attribute ) .

    owner -->
            [its]       |
            [their] .

term -->
            optional( [the] ),
            [answer,to],
            name( question ) .

term -->
            [something]   |
            [somebody]    |
            [anything]    |
            [anybody]     |
            [nothing]     |
            [nobody] |
            [empty] .

term -->
            optional( determiner ),
            name( identifier ) .

    determiner -->
            [the]     |
            [an]      |
            [a] .

term -->
            number .
```

Arithmetic Expression

```
expression -->
            term .

expression -->
            term,
            binaryop,
            expression .

    binaryop -->
            [+]     |
            [-]     |
            [*]     |
    [/]         |
```

```
        [^] .

    binaryop -->
        [plus]              |
        [minus]             |
        [times]             |
        [divided,by]        |
        [to,the,power,of] .

expression -->
        unaryop,
        expression .

    unaryop -->
        [-] .

    unaryop -->
        [minus] .
```

Appendix C - KSL Keyword Glossary

This glossary defines the reserved words of the Knowledge Specification Language. The KSL context for each entry determines whether it occurs as part of an object, as part of a formula about objects, or whether it is used directly within a sentence.

---

a / an

KSL Context: Sentence
Description: Used in defining the parents of a frame.
Used with: **is**
Example:
```
frame bird is an animal
```

KSL Context: Object
Description: A determiner which is an optional prefix to nouns.
Example:
```
if the favourite_food of X is a banana
then ...
```

---

above

KSL Context: Formula
Description: Checks that the 1st object is greater than the 2nd.
Example:
```
if the temperature is above freezing_point then ...
```

---

according

KSL Context: Formula
Description: Indicates a non-typographical comparison of two objects. Instead, the group which follows is used to determine the relative values of the objects.
Used with: **to**
Example:
```
if the likelihood of frost is less than probable according to fuzzy_ordering and ...
```

---

action

KSL Context: Sentence
Description: Collect together a set of directives as an action.
Used with: **;**
Example:
```
action print_table ;
for every C is some class
do ...
```

add

KSL Context	Formula
Description	Arithmetic addition.
Used with	**to**
Example	`if ...` `then` **`add`** `1 to running_total`

all

KSL Context	Object
Description	Universal quantifier over groups (refers to all the members of a group) and frames (refers to all instances or sub-classes of a frame).
Example	`if the colour of the flag includes` **`all`** `colours then ...`

always

KSL Context	Sentence
Description	Indicates an unconditional forward chaining rule.
Used with	**do**
Example	`rule catchall` **`always`** `do write('No more rules!') .`

and

KSL Context	Formula
Description	Joins together conditions in the IF part of a rule, or directives in the THEN part of a rule.

Example	`if the door is open`
	`and` `the time_of_day is evening`
	`then shut_the_door`
	`and` `lock_the_door`

KSL Context	Object
Description	Enumerates the individuals in a set.
Used with	**{ }**
Example	`if the staff include { john` **`and`** `mary }` `then ...`

another

see **new**.

answer

KSL Context: Object
Description: Retrieves the answer to a question. If the question has not previously been asked, then it is automatically asked when the answer is requested.
Used with: **to**
Example:
```
if the answer to entree is any meat
and the answer to wine is not some
red_wine
then ...
```

any

KSL Context: Object
Description: An existential quantifier over groups (refers to a member of the group) and frames (refers to an instance or sub-class of the frame).
Example:
```
if the colour of any brick is red
then ...
```

anybody / anything

KSL Context: Object
Description: An anonymous variable which can refer to any object whatsoever.
Example:
```
if the age of anybody is at or below
18 then ...
```

are

KSL Context: Formula
Description: Used in the comparison of two objects, one of which is plural.
Example:
```
if the jug`s contents are not equal
to the jug`s capacity then ...
```

KSL Context: Formula
Description: Assignment of a value to a plural slot or global variable.
Example:
```
if ...
then jugA`s contents are jugA`s
capacity
```

ask

KSL Context: Formula
Description: Forces the named question to be asked.
Example:
```
if ...
then ask entree
```

at

KSL Context — Formula
Description — Used in the comparison of two objects to include possible equality.
Used with — **or**
Example —
```
if the temperature is at or above freezing_point
```

because

KSL Context — Sentence
Description — Attaches an optional explanation (some canned text) to a rule or a question, which can then be used to explain how the rule was fired or why the question is being asked.
Example —
```
question starter
Please choose a starter for the meal
;
choose from pate, soup, melon ;
because The starter is an integral part of a meal .
```

become / becomes

KSL Context — Formula
Description — Assignment of a value to a slot or global variable.
Example —
```
if ...
then jugA`s contents becomes 0
```

below

KSL Context — Formula
Description — Checks that the 1st object is less than the 2nd.
Example —
```
if the temp is below boiling then ...
```

browse

KSL Context — Sentence
Description — Attaches an optional explanation (name of a specially formatted disk file) to a rule or a question, which can then be used to explain how the rule was fired or why the question is being asked.
Used with — **file**
Example —
```
question entree
Please choose an entree for the meal
;
choose from entree ;
browse file cookbook .
```

by

KSL Context	Formula
Description	Arithmetic division operator.
Used with	**divided**
Example	`if the payload of the lorry divided` **`by`** `2 is less than 9 then ...`

change / changes

KSL Context	Sentence
Description	Used within demons and constraints to indicate that the value of a slot or global variable has changed.

Example	`demon melt_down`
	`when the temperature` **`changes`** `to T`
	`and ...`

check

KSL Context	Sentence
Description	Used within constraints and watchdogs to indicate the test that will be carried out.
Used with	**then, that**

Example	`watchdog account_security`
	`when ...`
	`then` **`check`** `that the user`s access is above 9`

choose

KSL Context	Sentence
Description	Used within questions to indicate a single-choice or multiple-choice menu
Used with	**from, one, some of**

Example	`question wine`
	`Please choose some wines to go with your meal ;`
	`choose` `some of chablis, claret, champagne, plonk .`

come

KSL Context	Sentence
Description	Used in ruleset to select rules using the built-in first-come first-served rule selection algorithm.
Used with	**select rule using first, served**

Example	`ruleset timetable`
	`contains all rules ;`
	`select rule using first` **`come`** `first served .`

conflict

KSL Context — Sentence
Description — Used in ruleset to select rules using the built-in conflict resolution algorithm.
Used with — **select rule using, resolution**
Example —
```
ruleset stock_take
contains all rules ;
select rule using conflict resolution.
```

constraint

KSL Context — Sentence
Description — Data-driven procedure used to constrain the possible values for a slot or global variable.
Example —
```
constraint maximum_size
when the contents of any jug changes
to X
then check that X is at or below a
jug`s capacity .
```

contains

KSL Context — Sentence
Description — Used within a ruleset to define the initial rule agenda.
Example —
```
ruleset water_jugs
contains all rules .
```

cyclic

KSL Context — Sentence
Description — Used in ruleset to start the next rule agenda at the rule following the selected rule.
Used with — **update ruleset by, rotation of rules**
Example —
```
ruleset moving_blocks
contains all rules ;
update ruleset by cyclic rotation of
rules .
```

data

KSL Context — Sentence
Description — Defines some data initialisation procedures.
Example —
```
data start_up_configuration
the contents of jugA are 2
and ...
```

default

KSL Context — Sentence
Description — Declares a default value for an attribute of a frame.
Example —
```
frame squirrel is a rodent
default tail is { long and bushy }
```

demon

KSL Context — Sentence
Description — Data-driven procedure which reacts to changes in the values of slots or global variables.
Example

```
demon spy_contents
when the contents of X changes from Y
to Z
then write( contents(X,Y,Z) )
and nl .
```

demoting

KSL Context — Sentence
Description — Used in ruleset to demote each selected rule from the rule agenda.
Used with — **update ruleset by, each selected rule**
Example

```
ruleset spelling_checker
contains all rules ;
update ruleset by demoting each
selected rule .
```

different

KSL Context — Formula
Description — Checks that two objects have different values.
Used with — **from**
Example

```
if car`s mileage is different from 0
then ...
```

divided

KSL Context — Formula
Description — Arithmetic division operator.
Used with — **by**
Example

```
if the payload of the lorry divided
by 2 is less than 9
then ...
```

---

do

see also **include** and **while**

KSL Context	Sentence
Description	Prefix to a directive.
Used with	**action**

Example	`action jug_update ;`
	`do the contents of the jug := 7.5 and`
	`the position of the jug := upright .`

KSL Context	Formula
Description	An optional prefix to a directive.
Example	`if ... then do danger_level is red`

---

does

see **include**.

---

doing

KSL Context	Sentence
Description	Used in ruleset to define a set of actions to be performed prior to forward chaining.
Used with	**initiate by**

Example	`ruleset start`
	`contains all rules ;`
	`initiate by doing restart .`

---

each

KSL Context	Sentence
Description	Used in ruleset to remove, promote or demote each selected rule from the rule agenda.
Used with	**update ruleset by [removing, promoting, demoting], selected rule**

Example	`ruleset spelling_checker`
	`contains all rules ;`
	`update ruleset by demoting each selected rule .`

---

else

KSL Context	Object
Description	Indicates the alternative object within an IF-THEN-ELSE construct.
Used with	**if, then**
Example	`the result is if parameter > 0 then 1 else 0`

---

empty

see **nothing / nobody**.

end

KSL Context	Formula
Description	Indicates the end of a loop statement.
Used with	**for, repeat, while, if**

Example	`for X from -100 to 100`
	`do plot(X)`
	`end for`

equal

KSL Context	Formula
Description	Checks that two objects have the same value.
Used with	**to**
Example	`if jugA`s contents are equal to 0` `then ...`

every

see **all**.

false

KSL Context	Formula
Description	Tests whether a proposition does not hold.
Used with	**is**
Example	`if proposition is false` `then ...`

file

see **browse**.

first

KSL Context	Sentence
Description	Used in ruleset to select rules using the built-in first-come first-served rule selection algorithm.
Used with	**select rule using, come, served**

Example	`ruleset timetable`
	`contains all rules ;`
	`select rule using first come first served .`

for

KSL Context	Formula
Description	For loop.
Used with	**from, to, step, do, end**

Example	`for X from -100 to 100 step 5`
	`do plot(X)`
	`end for`

forget

KSL Context	Formula
Description	Removes assertions from either the positive (facts) or negative (exceptions) database.
Used with	**that**
Example	**forget** that danger_level(anything)

frame

KSL Context	Sentence
Description	Declares a new frame, its position in the frame hierarchy, any default values and any inheritance links.
Example	**frame** feline is a mammal ; default tail is { long and furry } ; inherit meal from carnivore .

from

see **inherit**, **remove** or **change** / **changes**.

function

KSL Context	Sentence
Description	Define an evaluable function.
Used with	**=**
Example	**function** factorial(N) = if N>0 then N*factorial(N-1) else 1 .

greater

KSL Context	Formula
Description	Checks that the 1st object is greater than the 2nd.
Used with	**than**
Example	if jugA`s contents are **greater** than 0 then ...

group

KSL Context	Sentence
Description	Defines the membership of a group.
Example	**group** fish salmon, cod, mullet, tuna, plaice .

---

if

KSL Context — Sentence
Description — Used with rules and clauses to discriminate between the conditions and the directives or conclusion.
Example —
```
rule prescribe_lomotil
if complains_of( patient, diarrhoea )
and ...
then ...
```

KSL Context — Object
Description — The start of the test within an IF-THEN-ELSE construct. If the test succeeds the 1st object is chosen, otherwise the 2nd object is chosen.
Used with — **then, else**
Example —
```
the result is if parameter > 0 then 1
else 0
```

---

in

see **include**.

---

include

KSL Context — Formula
Description — A directive for extending the current value of a set with some new objects.
Used with — **in**
Example —
```
if ...
then include lemon_sole in the entree
of meal
```

---

include / includes

KSL Context — Formula
Description — Tests whether a set includes some objects.
Example —
```
if the staff includes a secretary
then ...
```

---

included

KSL Context — Formula
Description — Tests whether a set includes some objects.
Used with — **in**
Example —
```
if a surprise is included in the
contents of the box then ...
```

---

inherit

KSL Context — Sentence
Description — Declares inheritance links for attributes of frames.
Used with — **from**
Example —
```
frame feline is a mammal ;
inherit meal from carnivore .
```

initiate

KSL Context — Sentence
Description — Used in ruleset to define a set of actions to be performed prior to forward chaining.
Used with — **by doing**
Example

```
ruleset start
contains all rules ;
initiate by doing restart .
```

input

KSL Context — Sentence
Description — Define a question as being keyboard input.
Used with — **name, number, integer, set**
Example

```
question age_of_applicant ..... ;
input integer .
```

instance

KSL Context — Sentence
Description — Declares a specific instance of a frame.
Used with — **is a**
Example

```
instance moby_dick is a whale ;
colour is white .
```

integer

see **input**

is

KSL Context — Formula
Description — Checks that two objects have the same value.
Example

```
if the blood`s count is low
then ...
```

KSL Context — Formula
Description — An assignment of a value to a slot or global variable.
Example

```
if ...
then the tail of the squirrel is
     { long and bushy }
```

its

KSL Context — Object
Description — When used to describe an attribute of a frame, it refers to another attribute of the same frame.
Example

```
frame root
default weight is its density times
its volume
```

kind

KSL Context	Sentence
Description	Used in defining the parents of a frame.
Used with	**is a, of**
Example	`frame bird is a` **`kind`** `of animal`

launch

KSL Context	Sentence
Description	A data-driven procedure for initialising new instances of frames.

Example	**`launch`** `female_enrolment`
	`when Person is a new student`
	`and female( Person ) then`
	`female_enrolment_questions( Person )`
	`.`

less

KSL Context	Formula
Description	Checks that the 1st object is less than the 2nd.
Used with	**than**
Example	`if the temperature is` **`less`** `than january`s average`

minus

KSL Context	Object
Description	Arithmetic operator.
Example	`if the number of managers` **`minus`** `the number of clerks > 0 then ...`

name

see **input**

network

KSL Context	Sentence
Description	Used in ruleset to update the rule agenda update according to a rule transition network.
Used with	**update ruleset using rule transition**

Example	`ruleset fault_diagnosis`
	`contains all rules ;`
	`update ruleset using rule transition` **`network`** `.`

---

new

KSL Context	Formula
Description	Creates a new instance of a frame, possibly with its own individual characteristics.
Used with	**is a**
Example	`if ... then X is a` **`new`** `carrier_bag`

---

nobody / nothing

KSL Context	Object
Description	The empty set.
Example	`if shopping_list is` **`nothing`** `then ...`

---

not

KSL Context	Sentence
Description	Negation (by failure).
Example	`if` **`not`** `[ test1 is true or test2 is false ] then ...`

KSL Context	Object
Description	Invert the comparison of two objects.
Example	`if the temperature is` **`not`** `less than january`s average then ...`

---

number

see **input**

---

of

see also **`s**.

KSL Context	Object
Description	Relates an attribute to a frame.
Example	`if the habitat` **`of`** `some animal is land then ...`

---

one

see **choose**.

---

or

KSL Context	Formula
Description	The disjunction operator for alternative conditions.
Example	`if the jug`s contents are 0` **`or`** `the jug`s contents are the jug`s capacity then ...`

KSL Context	Object
Description	The disjunction operator for alternative objects.

Used with **{ }**
Example
```
if the jug`s contents are { 0 or the
jug`s capacity } then ...
```

---

otherwise

KSL Context Sentence
Description Indicates the action to take when an error arises in a data-driven procedure.
Example
```
watchdog account_security
when the contents of account are
requested
then check the user`s access > 99
otherwise report_illegal_entry .
```

---

plus

KSL Context Object
Description Arithmetic operator.
Example
```
if the contents of jug1 plus the
contents of jug2 is above 10 then ...
```

---

power

KSL Context Object
Description Arithmetic operator.
Used with **to the, of**
Example
```
if ...
then its volume is its size to the
power of 3
```

---

promoting

KSL Context Sentence
Description Used in ruleset to promote each selected rule from the rule agenda.
Used with **update ruleset by, each selected rule**
Example
```
ruleset ordering_rules
contains all rules ;
update ruleset by promoting each
selected rule .
```

---

question

KSL Context Sentence
Description Defines the form and content of a set question.
Example
```
question cheeses
Select a cheese after the meal ;
choose some of cheese .
```

relation

KSL Context	Sentence
Description	Define a relationship between objects.
Used with	**if**
Example	**relation** father(Pop, Child) if parent(Pop, Child) and male(Pop) .

remember

KSL Context	Formula
Description	Record new assertions in either the positive (facts) or negative (exceptions) database.
Used with	**that**
Example	if ... then **remember** that danger_level(red)

remove

KSL Context	Formula
Description	Remove elements from a set.
Used with	**from**
Example	**remove** cod from the entree of meal

removing

KSL Context	Sentence
Description	Used in ruleset to remove each selected rule from the rule agenda.
Used with	**update ruleset by, each selected rule**
Example	ruleset checking_rules contains all rules ; update ruleset by **removing** each selected rule .

KSL Context	Sentence
Description	Used in ruleset to update the rule agenda by removing any rule whose conditions were unsatisfied.
Used with	**update ruleset by, any unsatisfied rules**
Example	ruleset state_checker contains all rules ; update ruleset by **removing** any unsatisfied rules .

repeat

KSL Context	Formula
Description	Repeat-Until loop.
Used with	**until, end**
Example	**repeat** ask question until valid_answer end **repeat**

requested

KSL Context	Sentence
Description	Used in the definition of a watchdog to indicate the slot which it is attached to.
Example	watchdog account_security when the balance of account is **requested** then check the user`s access > 99 .

resolution

KSL Context	Sentence
Description	Used in ruleset to select rules using the built-in conflict resolution algorithm.
Used with	**select rule using conflict**
Example	ruleset stock_take contains all rules ; select rule using conflict **resolution** with threshold 9 .

rotation

KSL Context	Sentence
Description	Used in ruleset to start the next rule agenda at the rule following the selected rule.
Used with	**update ruleset by cyclic, of rules**
Example	ruleset moving_blocks contains all rules ; update ruleset by cyclic **rotation** of rules .

rule

KSL Context	Sentence
Description	Defines a production rule within the forward chaining sub-system.
Example	**rule** prescribe_lomotil if complains(patient, diarrhoea) and not suffers(patient,liver_complaints) and not pregnant(patient) then prescribe(patient, lomotil) .

---

ruleset

KSL Context — Sentence
Description — Defines parameters for the forward chaining of a particular set of rules .
Example

```
ruleset water_jugs
contains all rules ;
select rule using conflict resolution
with threshold 6 ;
update ruleset using rule transition
network .
```

`s

see also **of**.

KSL Context — Object
Description — Used to relate an attribute to a frame.
Example

```
if the cinema`s films includes the
viewer`s film_choice
```

---

score

KSL Context — Sentence
Description — Attaches a score to a production rule.
Example

```
rule empty_jugA
if jugA`s contents are above 0
then jugA`s contents := 0
score jugA`s contents .
```

---

select

KSL Context — Sentence
Description — Used in ruleset to define the rule selection algorithm.
Used with — **rule using**
Example

```
ruleset stock_take
contains all rules ;
select rule using conflict resolution
with threshold 30 .
```

---

selected

KSL Context — Sentence
Description — Used in ruleset to remove, promote or demote each selected rule from the rule agenda.
Used with — **update ruleset by [removing, promoting, demoting] each, rule**
Example

```
ruleset ordering_rules
contains all rules ;
update ruleset by promoting each
selected rule .
```

---

served

KSL Context	Sentence
Description	Used in ruleset to select rules using the built-in first-come first-served rule selection algorithm.
Used with	**select rule using first come first**

Example	`ruleset timetable`
	`contains all rules ;`
	`select rule using first come first` **`served .`**

---

set

see **input.**

---

some

see **any**.

---

somebody / something

see **anybody / anything**.

---

step

see **for**

---

subtract

KSL Context	Formula
Description	Arithmetic subtraction.
Used with	**from**
Example	`if ...` `then` **`subtract`** `1 from running_total`

---

such

KSL Context	Object
Description	Qualifies a variable with a given condition.
Used with	**that**

Example	`if prime( X` **`such`** `that [ number( X )`
	`and X < 100 ] )`
	`then ...`

---

synonym

KSL Context	Sentence
Description	Defines a synonym for a frequently occurring term.
Example	**`synonym`** `maximum_number 999999 .`

---

template

KSL Context	Sentence
Description	Defines the positive and negative forms of a relation.
Used with	**^**
Example	**template** ontopof block ^ is on top of ^ ; block ^ is not on top of ^ .

---

terminate

KSL Context	Sentence
Description	Used in ruleset to define a set of conditions, which, if fulfilled, terminate forward chaining successfully.
Used with	**when**
Example	ruleset heat_kettle contains all rules ; **terminate** when the temperature of the water is 100 .

---

than

see **greater** or **less**.

---

that

see **such**.

---

the

see also **a / an**.

KSL Context	Object
Description	A determiner which is an optional prefix to nouns.
Example	if **the** contents of **the** jug are above **the** capacity of **the** jug then ...

---

their

plural form of **its**.

then

KSL Context: Sentence
Description: Indicates the action part of a production rule.
Example:
```
rule prescribe_paracetamol
if complains( patient, headache )
then prescribe(patient,paracetamol) .
```

KSL Context: Object
Description: Indicates the object to be used within an IF-THEN-ELSE construct if the test succeeds.
Used with: **if, else**
Example:
```
the result is if parameter > 0 then 1
else 0
```

threshold

KSL Context: Sentence
Description: Used in ruleset to determine a threshold for the conflict resolution selection algorithm.
Used with: **select rule using conflict resolution with**
Example:
```
ruleset stock_take
contains all rules ;
select rule using conflict resolution
with threshold 10 .
```

times

KSL Context: Object
Description: Arithmetic operator.
Example:
```
if the number of customers times 2 is
greater than the number of staff
then ...
```

to

see **according** or **change / changes**.

transition

KSL Context: Sentence
Description: Used in ruleset to update the rule agenda update according to a rule transition network.
Used with: **update ruleset using rule, network**
Example:
```
ruleset fault_diagnosis
contains all rules ;
update ruleset using rule transition
network .
```

true

KSL Context	Formula
Description	Checks that a proposition holds.
Used with	**is**
Example	`if proposition is` **`true`** `then ...`

unsatisfied

KSL Context	Sentence
Description	Used in ruleset to update the rule agenda by removing any rule whose conditions were unsatisfied.
Used with	**update ruleset by removing any, rules**

Example	`ruleset state_checker`
	`contains all rules ;`
	`update ruleset by removing any` **`unsatisfied`** `rules .`

until

see **repeat**.

update

KSL Context	Sentence
Description	Used in ruleset to define the rule agenda update algorithm.
Used with	**ruleset**

Example	`ruleset fault_diagnosis`
	`contains all rules ;`
	`update` `ruleset using rule transition network .`

using

KSL Context	Sentence
Description	Used in ruleset to define the rule selection algorithm.
Used with	**select rule**

Example	`ruleset stock_take`
	`contains all rules ;`
	`select rule` **`using`** `conflict resolution .`

KSL Context	Sentence
Description	Used in ruleset to define the rule agenda update algorithm.
Used with	**update ruleset**

Example	`ruleset fault_diagnosis`
	`contains all rules ;`
	`update ruleset` **`using`** `rule transition network .`

watchdog

KSL Context	Sentence
Description	Defines a check on a slot before access is allowed.
Example	**watchdog** salary_security when the salary of employee is requested then check the user`s job is supervisor .

when

KSL Context	Sentence
Description	Indicates the slot to which a data-driven procedure is to be attached.
Example	demon spy_all **when** the Attribute of a Frame changes from X to Y then write(update(Attribute,Frame,X,Y)) and nl .

where

see **such**.

while

KSL Context	Formula
Description	While loop.
Used with	**do, end**
Example	**while** invalid_answer do ask question end while

whose

KSL Context	Object
Description	Identifies a frame with particular characteristics.
Used with	**is, are**
Example	if ... then B is a new bag **whose** contents are empty .

'{' , '}'

KSL Context: Object
Description: Set constructors.
Example:
```
{ X : manager(X) } includes { john or mary }
```

'[' , ']'

KSL Context: Formula
Description: Collect together conditions into a single formula.
Example:
```
if test1
and [ test2 or test3 ]
and not [ test4 and test5 ]
then ...
```

$

KSL Context: Object / Formula
Description: Indicates that the following object, or all of the objects in the following formula, should not be dereferenced during evaluation.
Example:
```
if ...
then write( $ temperature )
and write( ' is ' )
and write( temperature )
and nl
```

:

see **such**.

:=

same as **becomes**

=

same as **equal**.

\=

same as **different from**.

<

same as **less**.

=<

same as **equal or less than**.

>

same as **greater**.

>=

same as **greater or equal to**.

+

same as **plus**.

-

same as **minus**.

---

*

same as **times**.

/

same as **divided by**.

---

^

see **power** or **template**.

Appendix D - Dealing with Uncertainty

This document aims to serve as a basic introduction to the various techniques that Flex now offers to support uncertainty. For a more detailed explanation, you are referred to:

"Knowledge-Based Systems for Engineers and Scientists"

(Hopgood, CRC Press, ISBN: 0-8493-8616-0)

Traditional expert systems work on the basis that everything is either true or false, and that any rule whose conditions are satisfiable is useable, i.e. its conclusion(s) are true. This is rather simplistic and can lead to quite brittle expert systems. Flex now offers support for where the domain knowledge is not so clearcut.

Given a rule:

rule1: if A & B then C

there are 3 potential areas for uncertainty.

- Uncertainty in data (how true are A and B)

- Uncertainty in the rule (how often does A and B imply C)

- Impreciseness in general

The first 2 can be handled using probabilities and the third using fuzzy logic.

Uncertainty in Data

Combining Probabilities

A probabilistic rule in flex can look like a production rule:

```
uncertainty_rule r33
 if the temperature is high
 and the water_level is not low
 then the pressure is high .
```

Later on we shall see that this is equivalent to:

```
uncertainty_rule r33
 if the temperature is high ( affirms 1.00 ; denies 1.00 )
 and the water_level is not low ( affirms 1.00 ; denies 1.00 )
 then the pressure is high .
```

with a prior probablity for `'pressure is high' of 0.5`

Probabilistic rules in flex can be invoked via:

```
relation simple_boiler_probability( P )
 if  trace propagation
 and reset all probability values
 and the probability that the water_level is low  = 0.03
 and the probability that the temperature is high  = 0.98
 and propagate simple_boiler_control probability rules
 and the probability that the pressure is high  = P .
```

Bayes' theorem states:

$$P(H/E) = P(H) * P(E/H) / P(E)$$

This states the probability of a hypothesis given some evidence in terms of the probablity of the evidence given the hypothesis. This is useful as it is generally easier to estimate the probablity of evidence given a hypothesis than the reverse.

Affirms and denies

We can attach weights to update our confidence in a hypothesis given new evidence. The larger the affirmed weight, the more confidant we can be in an hypothesis.

A Bayesian rule in flex looks like:

```
uncertainty_rule r44
 if the temperature is high ( affirms 18.00 ; denies 0.11 )
 and the water_level is not low ( affirms  1.90 ; denies 0.10 )
 then the pressure is high .
```

You can invoke the probability engine with something like:

```
relation boiler_prob( P )
 if  trace propagation
 and reset all probability values
 and the probability that the water_level is low  = 0
 and the probability that the temperature is high  = 1
 and the probability that the pressure is high  = 0.099
 and propagate boiler_control probability rules
```

and the probability that the release_valve is need_cleaning = P .

An answer, such as, P = 0.69, indicates that there is a probablity of 0.69 that the valve needs cleaning.

Advantages of Bayesian updating are:

1) technique is based on a proven statistical theorem

2) likelyhood is expressed as a propability (or odd)

3) weightings are based upon the probality of evidence

Disadvantages of Bayesian updating are:

1) the prior probability of an assertion must be known or estimated

2) dependant probabilities must be measured or estimated

3) the probability value tells us nothing about its accuracy

4) adding new rules often requires alterations to prior probablities and weightings in other rules

Odds and Probability

For the purpose of updating probabilities in a rule-based system, it is often more convenient to deal with the odds of an event arising rather than the probability. The odds of an hypothesis, O(H), are related to its probability, P(H), by the following relations.

O(H) = P(H) / P(~H) = P(H) / (1-P(H))

and

P(H) = O(H) / (O(H)+1)

Thus a hypothesis with a probability of 0.2 (1 chance in 5) has odds of 0.25 (or "4 to 1" against).

Similarly, a hypothesis with a probability of 0.8 (4 in 5 chance) has odds of 4 (or "4 to 1" on).

If you' rather use odds than probabilities for the above example, then you can have the following:

```
relation boiler_odds( 0 )
  if  trace propagation
  and reset all odds values
  and the odds that the water_level is low  = 0
  and the odds that the temperature is high  = 65535
  and the odds that the pressure is high  = 0.110
  and propagate boiler_control odds rules
  and the odds that the release_valve is need_cleaning = 0 .
```

An answer, such as, O = 2.18, indicates that the odds that the valve needs cleaning are 2.18.

Evidence based probabilities

The standard formula for updating the odds of a hypothesis, H, given that evidence, E, is observed is:

O(H/E) = A * O(H)

where O(H/E) is the odds of H given E, and A is the affirms weigth of E. The definition of A is:

A = P(E/H) / P(E/~H)

This is also sometimes referred to as the 'Likelihood Ratio' and is the ratio of probabilities that the evidence is there when the hypothesis is true to when the hypothesis is false; i.e. given the evidence exists, how likely is it that the

hypothesis is true.

Absence of Evidence

The absence of evidence is different from not knowing if the evidence is present or not, and can be used to reduce the probability of a hypothesis. The standard formula for updating the odds of a hypothesis, H, given that evidence, E, is absent is:

O(H/~E) = D * O(H)

where O(H/~E) is the odds of H given the absence of E, and D is the denies weigth of E. The definition of D is:

D = P(~E/H) / P(~E/~H)

or

D = (1 - P(E/H)) / (1 - P(E/~H))

This is the ratio of probabilities that the evidence is not there when the hypothesis is true to when the hypothesis is false; i.e. given the evidence is absent, how likely is is that the hypothesis is true.

Uncertain Evidence

To refelect uncertainty in E, we scale both A and D to A' and D' respectively using linear interpolation. The expressions used to calculate interpolated values are:

A' = [2(A-1) * P(E)] + 2 - A

D' = [2(1-D) * P(E)] + D

While P(E) is greater than 0.5 we use the *affirms* weigth, and when P(E) is less than 0.5, we use the *denies* weigth.

Certainty Theory

Certainty theory, as used in MYCIN, represents an attempt to overcome some of the shortcomings of Bayesian updating. Instead of using probablities, each assertion has a certainty value between 1 and -1 associated with it, as do rules.

The updating procedure for certainty values consists of adding a +ve or -ve value to the current certainty of a hypothesis. This contrasts with Bayesian updating where the odds of a hypothesis are always multiplied by the appropriate weighting. The basic formulae are:

CF' = CF' x C(E)

a) if C(H) >= 0 and CF' >= 0

C(H/E) = C(H) + [CF' x (1 - C(H))]

b) if C(H) <= 0 and CF' <= 0

C(H/E) = C(H) + [CF' x (1 + C(H))]

c) if C(H) and CF' have opposite signs

C(H/E) = C(H) + CF'/ (1 - min(|C(H)|, |CF|))

where:

C(H/E) is the certainty of H updated in the light of E

C(H) is the initial certainty of H

```
uncertainty_rule r41
 if the release_valve is stuck
 then the release_valve is need_cleaning
 with certainty factor 1.0 .

relation boiler_cf( CF1, CF2, CF3 )
 if trace propagation
 and reset all certainty_factor values
 and the certainty_factor that the water_level is low  = -1
 and the certainty_factor that the warning_light is on  = 1
 and propagate boiler_control certainty_factor rules
 and the certainty_factor that the release_valve is stuck = CF2
 and the certainty_factor that the release_valve is need_cleaning = CF3
```

— GENERAL INDEX —

— KSL KEYWORDS INDEX —

See also Appendix C - KSL Keyword Glossary

—*flex* PREDICATES INDEX —

Listed by category

Constraints

Data

Demons

Facts

Forward chaining

Frames

Functions

General

Groups

Inheritance

Initialisation