
4.900

# flex Examples

by Dave Westwood

# Flex Examples

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



**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

**web site:** http://www.lpa.co.uk/



14 July, 2004

# A Guide to the Flex Expert System Toolkit

This document is intended as a guide for learning and using the flex expert system toolkit. It shows where flex fits into the LPA software suite and how flex integrates with the underlying Prolog language. This document also gives a quick guide table showing the relative difficulties of learning the flex constructs. Finally a suite of examples are presented. A brief description is given for each example and the major points that it illustrates.

This guide should be used in conjunction with the 'flex Technical Reference' manual.

## Where Does Flex Fit In?

The following diagram shows some of the LPA suite of programming tools and how they work together. The products flex, FLINT and Prolog++ are all implemented in Prolog. The flex toolkit can be used on its own or in conjunction with FLINT, which provides support for uncertainty handling. Prolog also provides the means to access ODBC compliant databases and facilities for communicating with other programming languages.

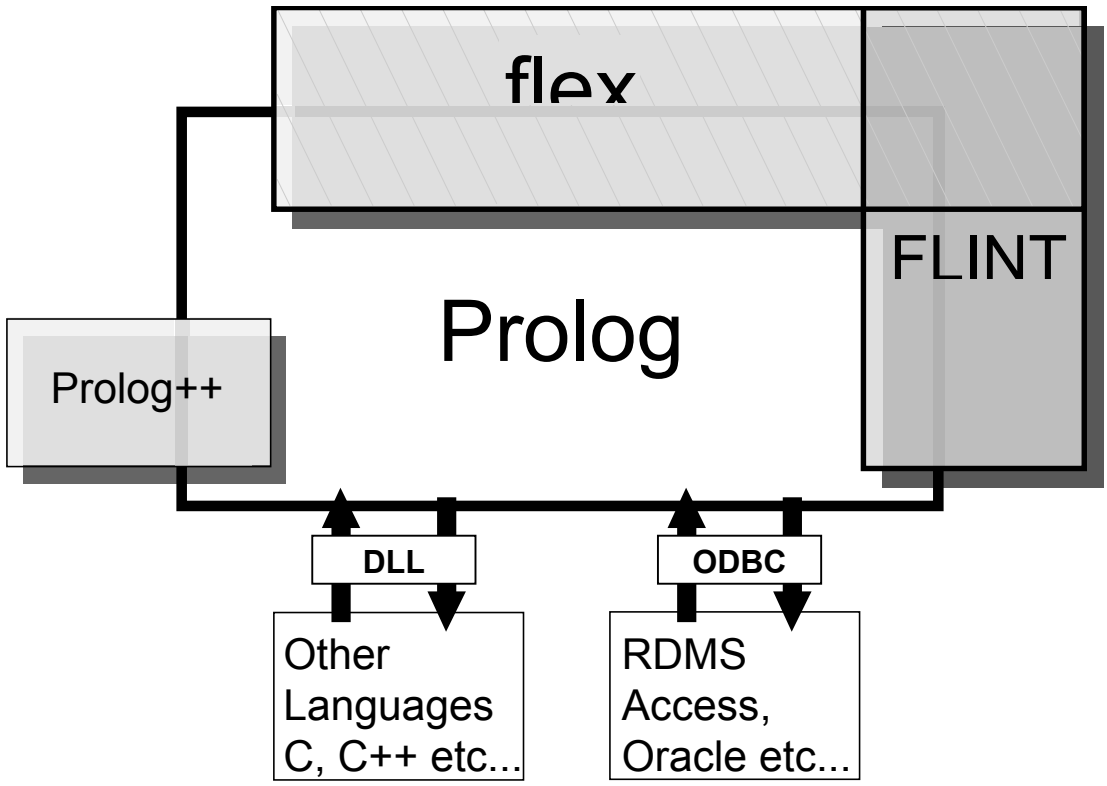


*Figure 1 - The LPA suite of products*

## What is flex

Flex is completely implemented in Prolog and has been designed specifically to enable the easy development of expert systems. The main features of flex are:

- Fully functional expert system toolkit with explicit support for many constructs such as: frames, rules, procedures etc..
- English-like Knowledge Specification Language (KSL) for defining these constructs.
- Extendible and configurable through a layer of Prolog access functions.

In addition to these main features, flex also contains:

- Access to Prolog and other languages.
- Portability to most hardware and software platforms.
- Optional support for uncertainty handling.
- Optional ODBC interface.
- High-level access to graphical user interface functions.

## Flex and Prolog

Flex has a three-level architecture: the KSL, the flex support predicates and the flex engine. The KSL sits on top of Prolog and the flex support predicates and flex engine are integrated directly into Prolog. At the top-most level is the English-like KSL that enables you to write expert systems in a form that can be easily read by non-programmers. This language is composed of sentences that are used to describe the constructs of the expert system. The flex support predicates in the next level are Prolog access functions that support these constructs. If you find that the KSL does not provide exactly what you need, you can extend and configure the behaviour of the constructs through this layer of flex support predicates. Finally, the flex engine provides the mechanism for implementing the run-time behaviour of the flex components. This is necessary to handle attribute inheritance, the forward-chaining process, data-driven programs etc..

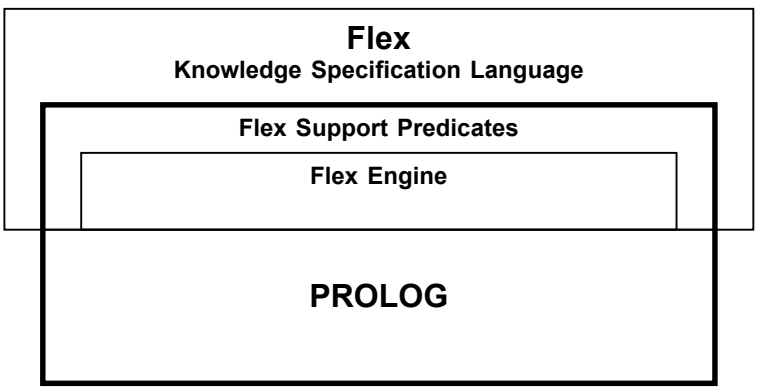


*Figure 2 - The flex architecture*

## A Closer Look at Flex

Flex is a fully functional expert system toolkit supporting a whole range of constructs commonly used in this area of programming. These include the following:

- Frame hierarchies, where the data for an expert system can be represented as a number of linked frames, attributes and values. This enables the data to be defined in a structured way, where common attributes are inherited.

- Forward-chaining, this is an iterative process whereby at each stage a single rule is selected from a set of rules according to the current state of the data. The selected rule then generates a new state for the data and the whole process then repeats.

- Backward-chaining, this is a process whereby the truth of an overall goal is established by proving the truth of its sub-goals. This process may provide alternate solutions on request.

- Questions and answers, where flex programs can query the user by asking a question. The answer to the question is then stored for later reference.

- Data-driven programs, where programs are triggered automatically by creating, accessing or updating specified types of data.

- Templates and synonyms, where the KSL program for an expert system may be tailored to suit the language appropriate to that expert system's domain.

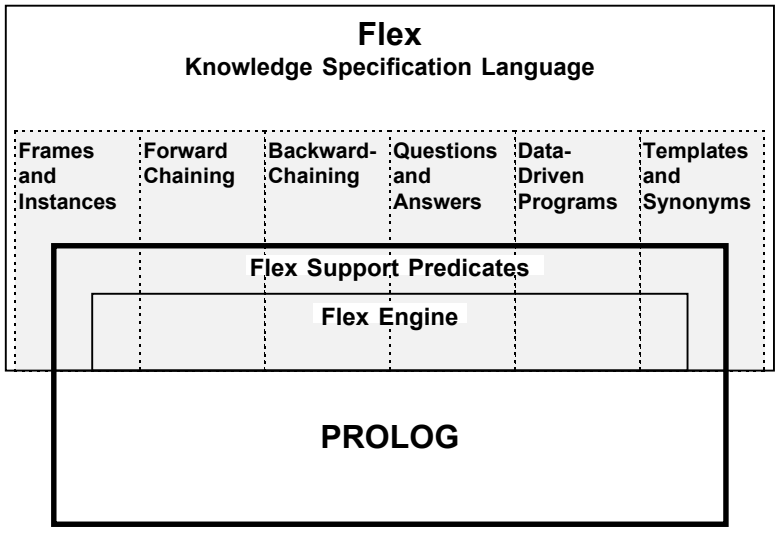


*Figure 3 - Zooming in on flex*

Figure 3 shows the flex constructs mapped onto the flex architecture shown previously. The KSL is usually sufficient to create and handle definitions for each of these constructs. If you find however, that your requirements have grown beyond the capabilities of the KSL, flex gives you a major advantage over most other expert system shells as the flex support predicates allow you to create, modify and interrogate flex constructs programmatically using Prolog. These support predicates allow you to tailor the features of flex to your own needs, whilst the flex engine handles

the underlying complexities of implementing the constructs. This makes flex extremely powerful in terms of adapting it to the requirements of any expert system domain.

## A Closer Look at Prolog

As well as being an expert system development environment, flex has seamless access to the high-level Prolog programming language. This is a major advantage if your expert system needs to go beyond the normal requirements. For example you may want to implement your own forward-chaining rule agenda update algorithm or frame inheritance strategy.

In addition to the standard facilities of Prolog (such as: pattern matching, list processing, backward chaining rules, definite clause grammars (DCGs), meta-level reasoning and recursion) *LPA-PROLOG* has specifically designed predicates for handling the following features, these features are also shown in Figure 4:

- High-level access to the Windows or Macintosh graphical user interface. Enabling you to create and handle dialogs, fonts and graphics.
- Random access file handling. Giving you the ability to interrogate structured disk-based data files.
- Formatted I/O. Enabling you to easily create formatted reports from your expert system results.
- Interfacing on the Windows platform to databases via ODBC. Using an additional add-on toolkit, you can access any database supporting ODBC, as if the records contained in the database were Prolog clauses.
- Interfacing to other languages. On the Windows platform, using the built-in DLL interface predicates, you can access the functions in a DLL to integrate code based in other languages such as C/C++, Delphi and Visual Basic. On the Macintosh there is support for code resources and apple events.

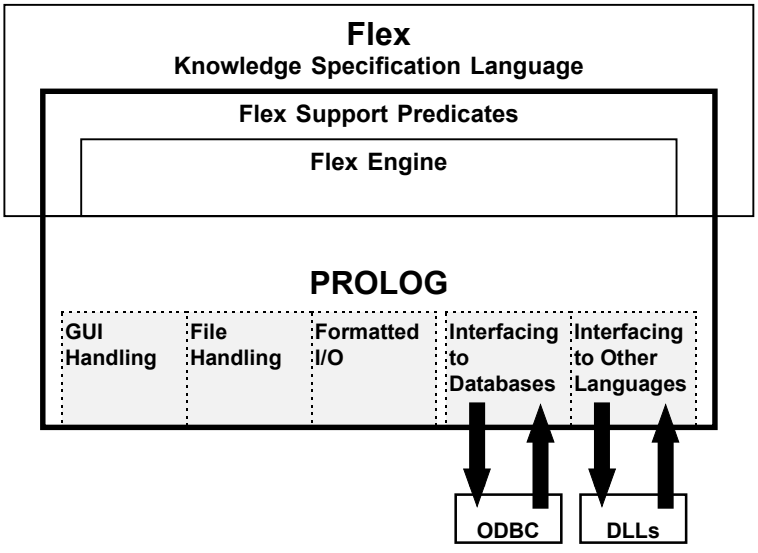


*Figure 4 - Zooming in on Prolog*

# Flex Constructs

In the previous section we looked at how flex integrates with Prolog and looked briefly at some of the constructs available in flex. In this section we outline all the constructs available in flex and group these into three categories, fundamental, intermediate and advanced. This table can also be used as a quick structured guide to accessing the '*flex Technical Reference*' manual.

| Level | Constructs |
|---|---|
| Fundamental | frames and instances<br>backward-chaining rules<br>forward-chaining rules<br>questions<br>storing and removing facts<br>groups<br>explanations<br>global variables<br>logical variables |
| Intermediate | control structures<br>data-driven programming<br>conflict resolution<br>compiler directives<br>data setup<br>templates/synonyms<br>sets<br>multiple inheritance<br>user-defined questions<br>de-referencing attributes |
| Advanced | functions<br>flex support predicates<br>user-defined ruleset agenda updates<br>attribute chaining<br>inherited logic |

# The Flex Example Suite

The examples provided with the flex system have been reviewed, revised, added to and commented, to enable an easier path to learning the flex expert system toolkit. The following table extends the flex construct overview to include, where appropriate, some flex examples that illustrate the use and programming of the construct. Flex keywords are shown in **bold**.

| Level | Constructs | Examples |
|---|---|---|
| Fundamental | **frames** and **instances** | ANIMAL.KSL, ROBBIE.KSL |
| | backward-chaining<br><br>**relations** and **actions** | SOLVENT.KSL, YIELD.KSL, SPECIES.KSL, TIMTABLE.KSL |
| | **questions** | QUESTION.KSL, QUESTION.PL, SPECIES.KSL, ROBBIE.KSL |
| | forward-chaining<br><br>**rules** and **rulesets** | TIMTABLE.KSL, WATER.KSL, ROBBIE.KSL, BLOCKS.KSL |
| | **groups** | QUESTION.KSL, ROBBIE.KSL, WATER.KSL |
| | explanations<br><br>**questions** and **rules** | QUESTION.KSL, WATER.KSL |
| | global variables | YIELD.KSL |
| | logical variables | YIELD.KSL |
| | storing and removing facts<br><br>**remember** and **forget** | BLOCKS.KSL, TIMTABLE.KSL |

| Level | Constructs | Examples |
|---|---|---|
| Intermediate | **templates** | YIELD.KSL |
| | **synonyms** | ROBBIE.KSL |
| | data-driven programming<br><br>**launches**, **demons**, **constraints** and **watchdogs** | ROBBIE.KSL, |
| | compiler directives<br><br>**do** | QUESTION.KSL |
| | data setup<br><br>**data** | ANIMAL.KSL |
| | multiple inheritance | ANIMAL.KSL |
| | sets | |
| | program control<br><br>if-then-else, repeat-until, while-do, for loops | FUNCTION.KSL, ANIMAL.KSL, |
| | user-defined questions | QUESTION.KSL, QUESTION.PL |
| | de-referencing | QUESTION.KSL |
| Advanced | flex support predicates | ANIMAL.KSL |
| | user-defined ruleset agenda updates | |
| | **functions** | FUNCTION.KSL |
| | attribute chaining | ADDRESS.KSL |
| | inherited logic | |

The following descriptions provide a quick guide into which technical points are covered by each example.

## SOLVENT.KSL

This example recommends a solvent to be used on some equipment. The solvent recommended depends on the equipment class, the ventilation of the site, the main material of the equipment and whether the equipment contains rubber compounds.

The following additional flex technical points are demonstrated in this example:

- The use of a single backward chaining **relation** to find the solvent.
- The automatic asking of a **question** when its value is tested.
- The use of square brackets to limit "**or**" disjunctions.

## BLOCKS.KSL

This example implements the blocks world problem, as defined in 'Logic for Problem Solving' by Professor Robert Kowalski.

The following additional flex technical points are demonstrated in this example:

- The use of a single forward chaining **rule** to generate valid moves.
- The use of square brackets in conjunction with "**or**".

## SPECIES.KSL

The objective of this example is to show two contrasting methods, forward and backward chaining, for identifying a species of animal given a set of attributes.

The following additional flex technical points are demonstrated in this example:

- Disjunctions ("**or**") used in the conditions of a forward-chaining **rule**.
- The constraining of a backward-chaining **relation** to a single solution.

## QUESTION.KSL

This simple example shows the range of different built-in questions that can be asked.

The following additional flex technical points are demonstrated in this example:

- The "**do**" directive for automatically running goals on compilation.
- User-defined questions implemented in Prolog.
- The **catchall** question that handles any undefined question asked.
- The use of relations to constrain the answers to questions.

- The use of a flex support predicate to find all the currently defined questions.
- The control of attribute de-referencing.

## ROBBIE.KSL

This example demonstrates the use of forward-chaining in a configuration and resource allocation problem.

The following additional flex technical points are demonstrated in this example:

- Data-driven **launch** procedures.
- The use of **synonyms** and **templates**.
- The use of nested **groups**.

## TIMTABLE.KSL

This example contrasts forward and backward-chaining methods for creating a timetable.

The following additional flex technical points are demonstrated in this example:

- The use of **remember** and **forget** to store values. It is important to note that remembered facts are automatically forgotten on backtracking.
- The use of backward chaining **relations** in the forward chaining process.

## WATER.KSL

This example implements the water containers problem, as defined in 'Logic for Problem Solving' by Professor Robert Kowalski.

The following additional flex technical points are demonstrated in this example:

- The use of dynamically updated scores in forward chaining rules.
- The linking of forward chaining rules into a network using groups.
- The use of rule explanations.

## YIELD.KSL

This example implements a simple expert system that recommends a species of tree seed, where to get the seed from, the normal yield of the seed and how that yield should be varied according to a description of the location, soil and conditions that hold where the seed is to be planted.

The following additional flex technical points are demonstrated in this example:

- The automatic asking of questions when an attribute is tested.

- The use of group ordering in testing the answer to questions
- The difference between setting values for logical and global variables.

## ANIMAL.KSL

This example demonstrates the frame hierarchy system and a mechanism for identifying frames according to positive and negative attribute clues.

The following additional flex technical points are demonstrated in this example:

- The use of a flex support predicate to find a currently defined frame.
- Specific inheritance links.
- Suppressing inheritance.
- Specifying more than one parent frame.
- Data declarations that are automatically run on compilation.

## FUNCTION.KSL

This example demonstrates the function construct by implementing a fibonacci and a factorial function.

The following additional flex technical points are demonstrated in this example:

- The automatic invocation of functions when requested.

# Flex Example Listings

This section gives a listing of each of the examples mentioned previously:

## SOLVENT.KSL Listing

```
/*
    The solvent example - From an example by James Stephenson
    ========================================================

    This example recommends a solvent to be used on some equipment.
    Depending on the equipment class, the ventilation of the site,
    the main material of the equipment and whether the equipment
    contains rubber compounds.

    Description
    -----------
    The main body of the example is the solvent/1 relation which contains
    the conditions appropriate for the various solvents.

    Each solvent is tried in turn. The conditions are also the names of
    questions. If the question related to the condition has not yet been
    asked it gets asked automatically when the condition is tested.
    The solvent is recommended or rejected depending on the response to
    these questions.

    Once a question has been asked the answer is stored so that subsequent
    tests for the condition does not invoke the question again.

    Running The Example
    -------------------
    The following goal reports the recommended solvent according to the
    answers to the questions:

    ?- find_solvent.

    Flex Technical Points
    ---------------------
    The following flex technical points are demonstrated in this example:

    1. The use of a single backward chaining relation to find the solvent.

    2. The automatic asking of questions when they are tested.

    3. The use of square brackets to limit "or" disjunctions.

*/

% this section contains the questions for the example

question equipment_class
   Which class does the equipment to be cleaned belong to? ;
   choose one of equipment_class_types
   because I need to know the classification of what is being cleaned .

group equipment_class_types
```

```
      class_1, class_2, class_3, class_4.

   question main_material
      Which of the following materials are primary materials in the equipment
    to be cleaned? ;
      choose some of main_material_types
      because I need to know what is being cleaned .

   group main_material_types
      stainless_steel, other_metal, plastic .

   question ventilation
      How good is the ventilation at the cleaning site? ;
      choose one of ventilation_types
      because some cleaning fumes are dangerous .

   group ventilation_types
      poor, fair, good .

   question rubber
      Does the equipment to be cleaned contain any rubber compounds? ;
      choose one of rubber_types
      because some cleaning solvents damage rubber .

   group rubber_types
      yes, no .

   % The solvent relation
   relation solvent('galaxy')
      if equipment_class is class_1
      and [ ventilation is poor or ventilation is fair ]
      and rubber is no .

   relation solvent('pulverizer' )
      if equipment_class is class_1
      and ventilation is good
      and rubber is no .

   relation solvent('polysol')
      if equipment_class is class_1
      and [ ventilation is poor or ventilation is fair ]
      and rubber is yes .

   relation solvent('polysol_plus')
      if equipment_class is class_1
      and ventilation is good
      and rubber is yes .

   relation solvent('cloripro_1')
      if equipment_class is class_2
      and [ ventilation is poor or ventilation is fair ]
      and rubber is no  .

   relation solvent('cloripro_2')
      if equipment_class is class_2
      and ventilation is good
      and rubber is no.
```

```
relation solvent('mtz_80')
   if equipment_class is class_2
   and [ ventilation is poor or ventilation is fair ]
   and rubber is yes  .

relation solvent('machine_saver' )
   if equipment_class is class_2
   and ventilation is good
   and rubber is yes.

relation solvent('acd_100')
   if equipment_class is class_3
   and [ ventilation is poor or ventilation is fair ]
   and rubber is no .

relation solvent('banish')
   if equipment_class is class_3
   and ventilation is good
   and rubber is no .

relation solvent('d_grease' )
   if equipment_class is class_3
   and [ ventilation is poor or ventilation is fair ]
   and rubber is yes.

relation solvent('grime_stopper' )
   if equipment_class is class_3
   and ventilation is good
   and rubber is yes.

relation solvent(not_known1)
   if main_material includes other_metal
   and [ main_material includes stainless_steel
         or main_material includes plastic ]
   and equipment_class is class_4 .

relation solvent(not_known2)
   if [ main_material includes other_metal
        or main_material includes plastic ]
   and main_material does not include stainless_steel
   and equipment_class is class_4 .

relation solvent(not_known3) .

% This action gets the solvent and then prints it to the screen.

action find_solvent ;
        do  restart
        and solvent(BestSolvent)
        and write( 'The recommended solvent is ' )
        and print( BestSolvent )
        and nl .
```

## BLOCKS.KSL Listing

```
/*
```

```
    The blocks world example - Adapted by - Phil Vasey
    ==================================================

    This example implements the blocks world problem, as defined in
    'Logic for Problem Solving' by Robert Kowalski.

    Description
    -----------
    Essentially the problem is to move three blocks, one at a time,
    from the initial state to the final state shown below:

    Initial  State                      Final  State

     ╔═══╗
     ╚═══╝
     ╔═══╗
     ╚═══╝
     ╔═══╗                                ╔═══╗    ╔═══╗    ╔═══╗
     ╚═══╝                                ╚═══╝    ╚═══╝    ╚═══╝
    _______   _______  _______          _______  _______  _______
    Table p   Table q  Table r          Table p  Table q  Table r

    Running The Example
    -------------------
    The following goal reports each move as it is made
    will run this example:

    ?- find_blocks_solution.

    Flex Technical Points
    ---------------------
    The following flex technical points are demonstrated in this example:

    1. The use of a single forward chaining rule to generate valid moves.

    2. The use of square brackets in conjunction with "or".

*/

group table
  p, q, r.

group block
  a, b, c.

template clear
  there is nothing on top of ^ .

/*
   Templates also allow the definition of the inverse relationship,
   the second alternative for the ontop template shows this.
*/

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

```
template justmoved
  block ^ has just been moved ;
  block ^ has not just been moved .

template hasmoved
  block ^ has been moved from ^ to ^ ;
  block ^ has never been moved from ^ to ^ .

template mover
  move block ^ from ^ to ^ .

template checker
  block ^ can be moved from ^ to ^ .

relation block X can be moved from Y to Z
  if   block X is on top of Y
  and  block X has not just been moved
  and  there is nothing on top of X
  and  there is nothing on top of Z
  and  X is different from Z
  and  block X has never been moved from Y to Z .

/*
  The square brackets in the following relation are necessary to indicate
  which conditions are part of the disjunction. If the brackets were not
  there a completely different disjunction would be defined.
*/

relation there is nothing on top of X
  if   [ X is some table or X is some block ]
  and  block anything is not on top of X .

action move block X from Y to Z ;
  do   forget   that block X is on top of Y
  and  remember that block X is on top of Z
  and  [ forget that block something has just been moved
         or     true ]
  and  remember that block X has just been moved
  and  remember that block X has been moved from Y to Z .

/*
   The following ruleset sets the conditions for forward chaining.
   The ruleset contains a single rule move_a_block.
   Forward chaining starts after the initial state has been set
   and only terminates when the final conditions have been met.
*/

ruleset blocks_world
  contains  move_a_block ;
  initiate  by doing restart
            and remember block a is on top of b
            and remember block b  is on top of c
            and remember block c is on top of p
            and display_blocks ;
  terminate when block a is on top of p
            and  block b is on top of q
            and  block c is on top of r .
```

```
rule move_a_block
  if    block X can be moved from Y to Z
  then  move block X from Y to Z
  and   display_move( X, Y, Z )
  and   display_blocks .

/*
   The following action is the top-level program. It runs the forward
   chaining engine, and then reports when the solution has been found.
*/

action find_blocks_solution ;
  do  invoke ruleset blocks_world
  and nl
  and write( 'the final state has now been reached')
  and nl .

action display_move( X, Y, Z )
  do    nl
  and   write( 'move block ' ) and write( X )
  and   write(' from ') and write( Y )
  and   write(' to ' )  and write( Z )
  and   nl.

relation display_blocks
  if  above_row( P1,Q1,R1,p,q,r )
  and above_row( P2,Q2,R2,P1,Q1,R1 )
  and above_row( P3,Q3,R3,P2,Q2,R2 )
  and display_line( P3, Q3, R3 )
  and display_line( P2, Q2, R2 )
  and display_line( P1, Q1, R1 )
  and display_base
  and nl.

relation  above_row( AboveP, AboveQ, AboveR , P, Q, R )
  if  if block AP is on top of P
      then  AboveP = AP
      else AboveP = noblock
      end if
  and if block AQ is on top of Q
      then AboveQ = AQ
      else AboveQ = noblock
      end if
  and if block AR is on top of R
      then AboveR = AR
      else AboveR = noblock
      end if .

relation display_line( Name1, Name2, Name3 )
  if  tab(1)
  and display_block_top( Name1 )
  and tab(8)
  and display_block_top( Name2 )
  and tab(8)
  and display_block_top( Name3 )
  and nl
  and tab(1)
  and display_block_name( Name1 )
```

```
  and tab(8)
  and display_block_name( Name2 )
  and tab(8)
  and display_block_name( Name3 )
  and nl
  and tab(1)
  and display_block_base( Name1 )
  and tab(8)
  and display_block_base( Name2 )
  and tab(8)
  and display_block_base( Name3 )
  and nl .

relation display_base
  if  write('_____')
  and tab(6)
  and write('_____')
  and tab(6)
  and write('_____')
  and nl
  and tab(2)
  and write(p)
  and tab(10)
  and write(q)
  and tab(10)
  and write(r) .

relation display_block_top( noblock )
  if  write('   ') .
relation display_block_top( Name )
  if  write('╔═╗') .

relation display_block_name( noblock )
  if  write('   ') .
relation display_block_name( Name )
  if write('║') and write( Name ) and write('║') .

relation display_block_base( noblock )
  if  write('   ') .
relation display_block_base( Name )
  if write('╚═╝') .
```

## SPECIES.KSL Listing

```
/*  The Species Identification Example -
    Ported from Mike to WIN-PROLOG by Clive Spenser and Dave Westwood
    ================================================================

    The objective of this example is to show two contrasting methods,
    forward and backward chaining, for identifying a species of animal
    given a set of attributes.

    Description
    -----------
    The backward-chaining method looks at a given species and attempts
    to prove, by asking questions related to the attributes of the
    species, if this species is the correct one.
```

```
        The forward-chaining method looks at the attributes provided prior
        to starting the forward-chaining process and attempts to build on
        these attributes until a full description of a species is given.

        Running The Example
        -------------------
        The species identification example reports a species appropriate to
        the answers to some discriminating questions.
        To run the forward-chaining method enter the following goal at the
        command line:

        ?- find_species_forward .

        To run the backward-chaining method enter the following goal at the
        command line:

        Flex Technical Points
        ---------------------
        The following flex technical points are demonstrated in this example:

        1. Disjunctions ("or") used in the conditions of forward-chaining
           rules.

        2. The constraining of backward-chaining relations to a single
           solution.

    */

    % The attribute question is used in the forward-chaining method.
    % It gives a list of attributes which the user can provide before
    % forward-chaining starts.

    question attributes
       Which attributes do you know to start with? ;
       choose some of attribute_types  .

    group attribute_types
       body_covering, colour, eats, eyes, feeds_young_on, feet,
       legs_and_neck, marking, motion, reproduction, teeth .

    % The following questions and groups are used in both the forward and
    % backward chaining examples.

    question body_covering
       What is the body_covering? ;
       choose one of body_covering_types
       because 'hair indicates a mammal, while feathers a bird' .

    group body_covering_types
       hair, feathers, other.

    question colour
       What is the colour? ;
       choose one of colour_types .

    group colour_types
       tawny, black_and_white, other.
```

```
question eats
   What does it eat? ;
   choose one of eats_types .

group eats_types
   meat, grass, other.

question eyes
   How do its eyes point? ;
   choose one of eyes_types .

group eyes_types
   point_forward, point_sideways.

question feeds_young_on
   What does it feed its young on? ;
   choose one of feeds_young_on_types .

group feeds_young_on_types
   milk, other.

question feet
   What kind of feet does it have? ;
   choose one of feet_types .

group feet_types
   claws, hoofs, other.

question legs_and_neck
   Are its legs and neck long or short? ;
   choose one of legs_and_neck_types .

group legs_and_neck_types
   long, short, other.

question marking
   What kind of marking does it have? ;
   choose one of marking_types .

group marking_types
   dark_spots, black_stripes, other.

question motion
   How does it move? ;
   choose one of motion_types .

group motion_types
   flies, swims, walks, other.

question reproduction
   How does it reproduce? ;
   choose one of reproduction_types .

group reproduction_types
   eggs, other.

question teeth
```

```
      Are its teeth pointed or blunt? ;
      choose one of teeth_types .

   group teeth_types
      pointed, blunt, other.

   % The next section defines the ruleset and rules used in the
   % forward-chaining method.

   ruleset identify
      contains identity_rules;
      update ruleset by removing each selected rule .

   group identity_rules
      mammal, bird, feeding_type_1, feeding_type_2,
      species_1, species_2, species_3, species_4,
      species_5, species_6, species_7 .

   % The following rule has a disjunction in its conditions.
   % It will be fired if either of the conditions hold.

   rule mammal
       if      the  body_covering is hair
       or      the feeds_young_on is milk
       then    the creature`s genus becomes mammal .

   rule bird
       if      the creature`s genus is unknown
       and   [ the body_covering is feathers
       or      the motion is flies
       and     the reproduction is eggs ]
       then    the creature`s genus becomes bird.

   % The "or" in the following rule says that either the first two
   % conditions must be true or the next three conditions must be true
   % before the rule can be fired.

   rule feeding_type_1
       if      the creature`s genus is mammal
       and     the  eats is meat
       or      the teeth is pointed
       and     the  feet is claws
       and     the  eyes is point_forward
       then    the creature`s feeding_type becomes carnivore.

   % The "or" in the following rule is restricted to the last two conditions
   % by the presence of the square brackets. This means that the first two
   % conditions must hold along with either of the last two before the rule
   % can be fired.

   rule feeding_type_2
       if      the creature`s genus is mammal
       and     the creature`s feeding_type is unknown
       and   [ the eats is grass
       or      the feet is hoofs ]
       then    the creature`s feeding_type becomes ungulate.

   rule species_1
```

```
    if      the creature`s feeding_type is carnivore
    and     the colour is tawny
    and     the marking is dark_spots
    then    the creature`s species becomes cheetah.

rule species_2
    if      the creature`s feeding_type is carnivore
    and     the colour is tawny
    and     the marking is black_stripes
    then    the creature`s species becomes tiger.

rule species_3
    if      the creature`s feeding_type is ungulate
    and     the colour is tawny
    and     the marking is dark_spots
    and     the legs_and_neck is long
    then    the creature`s species becomes giraffe.

rule species_4
    if      the creature`s feeding_type is ungulate
    and     the colour is black_and_white
    then    the creature`s species becomes zebra.

rule species_5
    if      the creature`s genus is bird
    and     the motion is walks
    and     the colour is black_and_white
    and     the legs_and_neck is long
    then    the creature`s species becomes ostrich.

rule species_6
    if      the creature`s genus is bird
    and     the motion is swims
    and     the colour is black_and_white
    then    the creature`s species becomes penguin.

rule species_7
    if      the creature`s genus is bird
    and     the motion is flies
    then    the creature`s species becomes albatross.

% This section defines the relations for the backward-chaining method.

% The following relation uses one/1 to constrain the suggest_family/1
% relation to a single solution. So that un-necessary questions are not
% asked. That is, once we have established that the family is mammal
% we don't want to go on to check that the family is not bird

relation check_family( Family )
   if  one( suggest_family( SuggestedFamily ) )
   and Family = SuggestedFamily .

relation suggest_family( mammal )
   if  the body_covering is hair
   or  the body_covering is other
   and the feeds_young_on is milk].

relation suggest_family( bird )
```

```
      if  the body_covering is feathers
      or  the motion is flies
      and the reproduction is eggs].

   relation check_feeding_type( FeedingType )
      if  check_family( mammal )
      and one( suggest_feeding_type( SuggestedFeedingType ) )
      and FeedingType = SuggestedFeedingType .

   relation suggest_feeding_type( carnivore )
      if  eats is meat
      or  teeth is pointed
      and feet is claws
      and eyes is point_forward .

   relation suggest_feeding_type( ungulate )
      if  eats is grass
      or  feet is hoofs.

   relation check_species( Species )
      if one( suggest_species( SuggestedSpecies ) )
      and Species = SuggestedSpecies .

   relation suggest_species( cheetah )
      if  check_feeding_type( carnivore )
      and colour is tawny
      and marking is dark_spots
      and legs_and_neck is short.

   relation suggest_species( tiger )
      if  check_feeding_type( carnivore )
      and colour is tawny
      and marking is black_stripes.

   relation suggest_species( giraffe )
      if  check_feeding_type( ungulate )
      and colour is tawny
      and marking is dark_spots
      and legs_and_neck is long.

   relation suggest_species( zebra )
      if  check_feeding_type( ungulate )
      and colour is black_and_white.

   relation suggest_species( ostrich )
      if  check_family( bird )
      and motion is walks
      and ask colour and colour is black_and_white
      and legs_and_neck is long.

   relation suggest_species( penguin )
      if  check_family( bird )
      and motion is swims
      and ask colour and colour is black_and_white.

   relation suggest_species( albatross )
      if  check_family( bird ).
```

```
% This action is the top-level goal for the backward-chaining solution

action find_species_backward
  do  restart
  and check_species( Species )
  and write( Species )
  and nl .

% This action is the top-level goal for the forward-chaining solution

action find_species_forward ;
   do  restart
   and ask attributes
   and invoke ruleset identify
   and write( creature`s species )
   and nl .
```

## QUESTION.KSL Listing

```
/*
    A Question Example - Phil Vasey and Dave Westwood
    ================================================

    This simple example shows the range of different built-in questions
    that can be asked.


    Running The Example
    -------------------
    An action ask_questions/0 has been defined that asks the questions
    contained in this file. This action can be run with the following
    goal:

    ?- ask_questions.

    A more advanced method of checking the all the flex questions that are
    currently defined is given in the ask_all_questions/0 action.
    This can be run with the following goal:

    ?- ask_all_questions.

    Flex Technical Points
    ---------------------
    The following flex technical points are demonstrated in this example:

    1. the "do" directive for automatically running goals on compilation.

    2. User-defined questions implemented in Prolog.

    3. The catchall question that handles any undefined question asked.

    4. The use of relations to constrain the answers to questions.

    5. The use of a flex support predicate to find all the currently
       defined questions.

    6. The control of de-referencing.
```

```
*/

action ask_questions
  do restart
  and ask firstname
  and ask interests
  and ask height
  and ask age
  and ask company_size
  and ask transport_type
  and ask password
  and ask confirm_password
  and ask any_old_question
  and ask any_other_question
  and output_answers .

action output_answers
  do  write('The user''s first name is: ')
  and write( firstname )
  and nl
  and write('interests are: ')
  and write( interests )
  and nl
  and write('height is: ')
  and write( height )
  and nl
  and write('age is: ')
  and write( age )
  and nl
  and write('company size is: ')
  and write( company_size )
  and nl
  and write('transport types are: ')
  and write( transport_type )
  and nl
  and write('any old question is: ')
  and write( any_old_question )
  and nl
  and write('any other question is: ')
  and write( any_other_question )
  and nl .

% The firstname question asks for typed input from the user

question firstname
  Please enter your first name ;
  input name .

% The interests question uses the default input which returns a set.
% Because the question text is quoted the carriage returns are included
% in the dialog question field.

question interests
  'Please type in your interests.
[Each interest should be separated by a space
 Interests that consist of more than one word should be quoted
 e.g. reading surfing 'table tennis' etc...]' .
```

```
% The height may be returned as a floating point number or integer.

question height
  'Enter your height in metres.

[This may be entered as a floating point number or an integer]';
  input number .

% Age must be returned as an integer.

question age
  'Please enter your age

[This must be entered as an integer]';
  input integer .

% The company_size is selected from the specified list of sizes.

question company_size
  'What is the rough size of the company you work for?

[Select one of the following list:]'  ;
   choose one of small, medium, large .

% This question allows multiple selections from the transport group.

question transport_type
  'Which of the following modes of transport do you use?

[Choose some of the following list:]';
  choose some of transport .

% The transport group is a set of nested groups, which are flattened out
% when the transport_type question is asked.

group transport
  motorised_transport, mechanical_transport, animal_transport .

group motorised_transport
  car, motorcycle, plane .

group mechanical_transport
  bicycle, skateboard, rollerskates .

group animal_transport
  donkey, horse .

do ensure_loaded( examples( flxdlgeg ) ) .

% This question asks for a word to be entered.

question password
  answer is Password
  such that password_dialog( Password ) .

% This question uses the confirm_password/1 relation to ensure that the
% response is identical to the response to the previous question.
```

```
question confirm_password
  Please confirm the password you just entered;
  input X such that confirm_password(X) ;
  because the answer must be the same as the password you entered earlier .

% This relation checks that a given response is equivalent to the answer
% to the password question.

relation confirm_password( P )
  if P is equal to the answer to password .
relation confirm_password( P )
  if echo( P, 'is not the correct password' )
  and fail .

/*
    The ask_all_questions/0 action writes the answer to all the questions
    currently defined in the flex system.

    The isa_question/4 predicate can be used to find all the currently
    defined questions.

    The "$" sign shows that the value of N is not to be de-referenced. In
    this case we want to write out the name of the question and not its
    answer.
*/

action ask_all_questions ;
  for every isa_question(N,_,_,_)
      do  write( $ N )
      and write( ' is ' )
      and writeq( the answer to N )
      and nl
  end for .


% The catchall question can be used to catch any undefined question.

question catchall
  This question catches any undefined question.
Please input some text;
  input name .
```

## ROBBIE.KSL Listing

```
/*
    Robbie the Robot Example - Phil Vasey
    =====================================

    This example demonstrates the use of forward-chaining in a
    configuration and resource allocation problem.

    Description
    -----------
    This example implements a shopping expert system. The problem is in
    three distinct stages:
```

```
      1. Query the user to get the initial shopping list.
      2. Check the compatibility of the items on the shopping list.
      3. Pack the items on the shopping list into bags.

    Running the Example
    -------------------
    The top-level goal to run the expert system is start_robot/0. This can
    be invoked by entering the following goal:

    ?- start_robot.

    Flex Technical Points
    ---------------------
    The following flex technical points are demonstrated in this example:

    1. Data-driven launch procedure.

    2. The use of synonyms and templates

    3. The use of nested groups

*/

% the following action is the backbone of the expert system
% the three stages are clearly defined

action start_robot ;
  do  restart

  and write_nl(         'Asking for the initial shopping list' )
  and ask shopping
  and display_shopping( initial )

  and write_nl(         'Checking the compatibility of items'  )
  and invoke ruleset checking_rules
  and display_shopping( final )

  and write_nl(         'Packing the items into bags'          )
  and invoke ruleset packing_rules
  and display_bags .

% the following actions display information of various kinds

action write_nl( Message )
  do  nl
  and write( Message )
  and nl .

action display_shopping( Stage )
  do write( 'The shopping list at the ' )
  and write( Stage )
  and write( ' stage is... ' )
  and nl
  and print( shopping )
  and nl
  and nl .

% the nested for loops in the following action display every bag
```

```
% and every item in each bag

action display_bags
  do nl
  and for every Bag is some carrier
     do write( 'The contents of carrier ' )
     and write( Bag )
     and write( ' is ' )
     and nl
     and for every Item is included in the contents of Bag
         do write( Item )
         and write( ' ' )
         end for
     and nl
     and nl
     end for
  and nl
  and nl .

action display_new_bag( Bag )
  do   write( 'Need a new carrier : ' )
  and  write( Bag )
  and  nl .

action display_packed_item( Item, Bag )
  do write( 'Packing ' )
  and write( Item )
  and write( ' into ' )
  and write( Bag )
  and nl .

% the word large_maximum is replaced by 1 wherever it occurs in the file

synonym large_maximum 1 .

synonym items_maximum 3 .

% the text "choose a carrier X for Y" is mapped on to choose_bag( X, Y )

template choose_bag
  choose a carrier ^ for ^ .

template pack_item
  pack ^ .

% a frame and a launch procedure define the carrier bags

frame carrier
  default contents are empty .

% the launch procedure creates and then reports a new carrier bag

launch new_carrier
  when Bag is a new carrier
  then display_new_bag( Bag ) .

% this action makes use of the choose_bag and pack_item templates to make
% the action easier to read
```

```
action pack an Item ;
  do  choose a carrier Bag for the Item
  and remove the Item from the shopping
  and include the Item in the Bag`s contents
  and display_packed_item( Item, Bag ) .

% this relation either selects a currently used bag or creates a new one

relation choose a carrier Bag for an Item
  if  the Bag is some carrier
  and length( the Bag`s contents, Count )
  and if   the Item`s size is large
      then Count is less than the large_maximum
      else Count is less than the items_maximum
      end if
  and ! .

relation choose a carrier Bag for an Item
  if  gensym( bag_number_, Bag )
  and Bag is a new carrier .

% the goods group also includes the drink and nibbles groups

group goods
  bread, butter, coffee, ice_cream, nibbles, drink, washing_powder .

group drink
  beer, lemonade .

group nibbles
  crisps, salted_peanuts .

% this section defines the characteristics of the items on the shopping

frame item .

frame large_item is an item ;
  default size is large .

frame medium_item is an item ;
  default size is medium .

frame small_item is an item ;
  default size is small .

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 .

% this section defines the questions used in the expert system

question shopping
  What is on your shopping list today ? ;
  choose some of goods
  because I need to check your shopping and then pack it into bags .

question drink
  You must select a drink ! ;
  choose one of drink
  because There are nibbles on your shopping list .

% this ruleset defines the forward chaining checking phase
% successful rules are removed after they are fired

ruleset checking_rules contains
    check_nibbles,
    check_bread_and_butter,
    check_butter_and_bread ;
  update ruleset by removing each selected rule .

rule check_nibbles
  if the shopping includes X and X is some nibbles
  and  the shopping does not include some drink
  then ask drink
  and  include the drink in the shopping
  because Nibbles make you thirsty .

rule check_bread_and_butter
  if   the shopping includes bread
  and  the shopping does not include butter
  then include butter in the shopping
  and  flash( 'You', forgot, the, butter )
  because Bread is very dry by itself .
```

```
rule check_butter_and_bread
  if   the shopping includes butter
  and  the shopping does not include bread
  then include bread in the shopping
  and  flash( 'You', forgot, the, bread )
  because WhatUs the use of butter without bread .

% this ruleset defines the forward chaining packing phase
% rules that do not apply are removed

ruleset packing_rules contains
    pack_frozen_item,
    pack_large_bottle,
    pack_other_large_item,
    pack_medium_item,
    pack_small_item ;
  update ruleset by removing any unsatisfied rules .

rule pack_frozen_item
  if   the shopping includes an Item
  and  the Item`s condition is frozen
  then pack the Item .

rule pack_large_bottle
  if   the shopping includes an Item
  and  the Item`s container is bottle
  and  the Item`s size is large
  then pack the Item .

rule pack_other_large_item
  if   the shopping includes an Item
  and  the Item`s size is large
  then pack the Item .

rule pack_medium_item
  if   the shopping includes an Item
  and  the Item`s size is medium
  then pack the Item .

rule pack_small_item
  if   the shopping includes an Item
  and  the Item`s size is small
  then pack the Item .
```

## TIMTABLE.KSL Listing

```
/*
    Timetable Example - Phil Vasey
    ==============================

    This example contrasts forward and backward-chaining methods for
    creating a timetable.

    Description
    -----------
    Timetabling is a classical AI problem which involves both search and
```

```
strategy components. It has similar charactersistics to resource
allocation, scheduling and planning in general.

The timetable example is presented as a choice between two contrasting
forward and backward chaining solutions. To run the example enter the
following goal at the command line:

?- timetable.

In this example we have some data that represents a number of
teachers, classes, subjects and periods. The teachers can teach
certain subjects and classes and have periods of freetime where they
are unavailable to teach.

Given this data, the basic requirement is to schedule a daily teaching
rota such that:

i.  A teacher can only one class during a given period.

ii. A class is taught a subject once only per day.


This example compares two contrasting methods for implementing a
timetable given the above constraints. Both methods attempt to fill
the timetable as much as possible by suggesting teachers that can
teach a subject to a given class during a given period. This often
leads to a situation where there is a partially filled timetable,
where no more teachers and subjects can be given for the remaining
classes and periods without conflicts arising.

The two methods differ in how they deal with this deadlock.

Forward-chaining
----------------
This is the most efficient method, though there is no guarantee that
a potentially existing solution will always be found.

In this approach conflicts are resolved by either swapping previously
assigned teachers or previously assigned subjects. In this way
conflicts are resolved directly when they arise.

Backward-chaining
-----------------
This is less efficient than forward-chaining but if a solution exists
then this approach will eventually get there.

In this approach conflicts are resolved by blindly undoing the last
teacher and subject allocated, replacing them with some alternative,
and then attempting to fill out the rest of the timetable.
If this new allocation does not lead to a solution then this process
is repeated until all teacher - subject combinations have been
exhausted.

One problem with this method is that the very first teacher, class,
subject and period allocated may prevent a solution from being found,
yet this method will try all the possible solutions based on this
first allocation before rejecting it and trying an alternative.
```

```
    Flex Technical Points
    =====================

    The following flex technical points are demonstrated in this example:

    1. The use of remember and forget to store values. It is important to
       note that remembered facts are automatically forgotten on
       backtracking.

    2. The use of backward chaining relations in the forward chaining
       process.

*/

% First we set up the data for the timetable
% the following groups define the classes and periods

group class
   alpha, beta, gamma .

group period
   1, 2, 3, 4, 5 .

% the following frame hierarchy defines the teachers, their subjects
% which classes they may teach and their freetime

frame teacher ;
   default freetime is nothing and
   default classes are every class .

instance john is a teacher ;
   subjects are { maths and physics } and
   freetime is { 3 } .

instance mary is a teacher ;
   subjects are { french } and
   freetime is { 4 } and
   classes are { alpha and gamma } .

instance matthew is a teacher ;
   subjects are { history and geography } and
   freetime is { 1 } .

instance helen is a teacher ;
   subjects are { chemistry } and
   freetime is { 5 } .

instance henry is a teacher ;
   subjects are { english } and
   freetime is { 2 } .

% the following templates enhance the readability of the rules and
% relations.

% The first template outlines the following types of phrase:
% e.g. class alpha is taught maths by john during period 1
%      class gamma is not taught french by mary during period 2
```

```
template taught
   class ^ is taught ^ by ^ during period ^ ;
   class ^ is not taught ^ by ^ during period ^ .

template not_yet_timetabled
   class ^ needs timetabling for period ^ .

template potential_teacher
   ^ can potentially teach ^ to class ^ during period ^ .

template definite_teacher
   ^ can definitely teach ^ to class ^ during period ^ .

template bad_teacher
   ^ already teaches class ^ during ^ so cannot teach class ^ ;
   ^ does not teach any ^ during ^ so is available to teach class ^ .

template bad_subject
   ^ is already taught to ^ during ^ which conflicts with period ^ ;
   ^ is not taught to ^ at any other ^ so is available for period ^ .

% the following relations are used by both the timetable methods

% generate a class and a period which are not already allocated

action class C needs timetabling for period P
   if C is some class
   and      P is some period
   and      class C is not taught anything by anybody during period P .

% generate any teacher and subject for a given class and period
% according to the intitial data specifications

relation T can potentially teach S to class C during period P
   if T is an instance of teacher
   whose freetime does not include P and
   whose subjects include S and
   whose classes include C .

% generate a teacher and subject for a given class and period
% and test that they do not conflict with any previous allocations

relation T can definitely teach S to class C during period P
   if  T can potentially teach S to class C during period P
   and T does not teach any C1 during P so is available to teach class C
   and S is not taught to C at any other P1 so is available for period P .

% identify potential teacher conflicts due to teachers already being
% allocated to some class for a given period

relation T already teaches class C1 during P so cannot teach class C
   if  class C1 is taught something by T during period P
   and C1 is different from C .

% identify potential subject conflicts due to subjects already being
% taught to a given class during some period

relation S is already taught to C during P1 which conflicts with period P
```

```
   if class C is taught S by somebody during period P1
   and      P1 is different from P .

% forward chaining method

% the following ruleset first continually fires the extend_timetable rule
% until it can no longer be fired. This rule is then removed from the
% timetable and forward chaining continues with the two conflict
% resolving rules only

ruleset forward_chaining_timetable
   contains extend_timetable,
            resolve_teacher_conflict,
            resolve_subject_conflict  ;
   select rule using first come first served ;
   update ruleset by removing any unsatisfied rules ;
   initiate by doing restart .

% this rule generates a valid class, period, teacher and subject
% and then allocates this to the current timetable

rule extend_timetable
   if   class C needs timetabling for period P
   and      T can definitely teach S to class C during period P
   then     remember that class C is taught S by T during period P .

% this rule resolves potential teacher conflicts

rule resolve_teacher_conflict
   if   class C needs timetabling for period P
   and      T can potentially teach S to class C during period P
   and      S is not taught to C at any other P1 so is available for period
P
   and      T already teaches class C1 during P so cannot teach class C
   and      T1 can definitely teach S1 to class C1 during period P
   and      T1 is different from T
   then     forget       that  class  C1  is  taught  something  by  T  during
period P
   and      remember that class C1 is taught S1 by T1 during period P
   and      remember that class C is taught S by T during period P .

% this rule resolves potential subject conflicts

rule resolve_subject_conflict
   if  class C needs timetabling for period P
   and T can potentially teach S to class C during period P
   and T does not teach any OldC during P so is available to teach class C
   and S is already taught to C during P1 which conflicts with period P
   and T1 can definitely teach S1 to class C during period P1
   and S1 is different from S
   then     forget       that class C is taught S by somebody during period
P1
   and remember that class C is taught S1 by T1 during period P1
   and remember that class C is taught S by T during period P .

% backward chaining method

% in the following relation the remembering of the class, subject,
```

```
% teacher and period is undone when no more teachers can be proposed
% for the remaining classes and periods

relation backward_chain_timetable
   if class C needs timetabling for period P
   and  !
   and      T can definitely teach S to class C during period P
   and      remember that class C is taught S by T during period P
   and      backward_chain_timetable .
relation backward_chain_timetable .

% this action lets you select the forward or backward chaining solution.

action timetable ;
  do  ask timetable_method
  and write( 'Thinking ...' )
  and nl
  and if   the answer to timetable_method is 'forward chaining solution'
      then do forward_chaining_timetable
      else do backward_chaining_timetable
      end if .

% this action initiates the backward chaining solution.

action backward_chaining_timetable ;
   do restart
   and      backward_chain_timetable
   and      print_table .

% this action initiates the forward chaining solution.

action forward_chaining_timetable ;
   do   restart
   and  invoke ruleset forward_chaining_timetable
   and  print_table .

question timetable_method
  Which method ? ;
  choose one of 'forward chaining solution',
                'backward chaining solution' .

% this action prints the final timetable solution

action print_table ;
   for every C is some class
      do    write( class(C) )
      and   nl
      and   for every P is some period
                and     class C is taught S by T during period P
                  do write( period(P,S,T) )
                  and nl
            end for
      and   nl
   end for .
```

## YIELD.KSL Listing

```
/*
    Forest Yield Example - Phil Vasey
    =================================
    This example implements a simple expert system that recommends a
    species of tree seed, where to get the seed from, the normal yield
    of the seed and how that yield should be varied according to a
    description of the location, soil and conditions that hold where the
    seed is to be planted.

    Description
    -----------
    The problem is structured around a backward-chaining relation that
    defines which attributes are needed to recommend a particular seed.
    These attributes include the location of the site, its soil-type,
    elevation, exposure and climate. When the program is testing for the
    existance of a given set of attribute values the questions relating
    to those attributes are asked automatically. After a suitable seed
    has been found, the best provenance for the seed and its expected
    yield, given the location, are calculated.

    The top-level goal ensures that all the possible solutions are
    returned.

    Running The Example
    -------------------
    The forest yield example asks a series of questions and then reports
    the recommended species of tree based on the given answers. To run the
    example enter the following goal at the command line:

    ?- recommend_species .

    Flex Technical Points
    ---------------------
    The following flex technical points are demonstrated in this example:

    1. The automatic asking of questions when an attribute is tested.

    2. The use of group ordering in testing the answer to questions

    3. The difference between setting values for logical and global
       variables.

*/

% The following templates aid the readability of the code

template valid_species_yield
   valid ^ and ^ combination .

template species_yields
   can find ^ with yields ^ and ^ .

template notes1
   obtain seed for ^ from ^ .
```

```
template notes2
   compute ^ for ^ .

template notes3
   ^ returns a variance of ^ .

template output
  output ^, ^, ^ and ^ to the screen .

% The following top-level action gets as many species and yield
% combinations that conform to the answers given.

action  recommend_species ;
   do  restart
   and for every valid Species and Yield combination
                and   obtain seed for Species from Provenance
                and   compute Variance for Provenance
       do output Species, Provenance, Yield and Variance to the screen
   end for .

% The questions are defined in the following section

question site
   Which region is the site located in ? ;
   choose one of site_types .

group site_types
   'North Scotland',
   'East Scotland',
   'West Scotland' .

question soil
   Which type of soil can be found at the site ? ;
   choose one of soil_types .

group soil_types
   'Brown earth',
   'Gley',
   'Peat',
   'Podsol',
   'Ironpan',
   'Intergrade',
   'Bog',
   'Alluvium',
   'Skeletal' .

question elevation
   At which elevation zone is the site located ? ;
   choose one of 'Lower',
                 'Upper' .

question exposure
   In which range does the site exposure lie ? ;
   choose one of exposure_types
   because 'Low' = TOPEX < 30, and 'High' = TOPEX > 60 .

group exposure_types
   'Low', 'Moderate', 'High'.
```

```
question gley
   Which type of gley is it ? ;
   choose one of 'Surface water gley',
                 'Flushed gley',
                 'Unflushed gley',
                 'Peaty gley' .

question status
   What is the status of the peaty gley ? ;
   choose one of 'Good',
                 'Poor' .

question peat
   Which type of peat is it ? ;
   choose one of 'Flushed peat',
                 'Unflushed peat',
                 'Basin peat',
                 'Hill peat' .

question bog
   Which type of bog is it ? ;
   choose one of 'Flushed bog',
                 'Unflushed bog' .

question moisture
   What is the predominant climate of the site  ;
   choose one of 'Wet',
                 'Dry' .

question grass
   Which grass is local to the site ? ;
   choose one of 'Heather',
                 'Blue moor grass' .

question plough
   Which method of ploughing is used ? ;
   choose one of 'Complete',
                 'Spaced line' .

question flush
   Is the flush local or not ? ;
   choose one of 'Local',
                 'Non-local' .

% This section defines the main species and their yield class (both
% lower and upper) according to:
%
% i.   Site location
% ii.  Site elevation
% iii. Soil type
% iv.  Soil sub-type
% v.   Other miscellaneous factors

% Given the site location, the specific yield can be
% chosen from the upper yield or the lower yield.
```

```
relation valid Species and Yield combination
   if  can find Species with yields Yield1 and Yield2
   and if   the elevation is 'Lower'
       then the Yield is Yield1
       else the Yield is Yield2
       end if .

% The species and expected yields (both upper and lower)
% are determined by the following relation

% The relation is split into three main sections:
% 1. The North of Scotland

relation can find 'Sitka spruce' with yields 15 and 12
   if  the site is 'North Scotland'
   and the soil is 'Brown earth' .

relation can find 'Scots pine' with yields 8 and 7
   if  the site is 'North Scotland'
   and the soil is 'Podsol'
   and the moisture is 'Dry' .

relation can find 'Lodgepole pine' with yields 10 and 9
   if  the site is 'North Scotland'
   and the soil is 'Podsol'
   and the moisture is 'Wet' .

relation can find 'Scots pine' with yields 9 and 7
   if  the site is 'North Scotland'
   and the soil is 'Ironpan'
   and the moisture is 'Dry' .

relation can find 'Lodgepole pine' with yields 11 and 9
   if  the site is 'North Scotland'
   and the soil is 'Ironpan'
   and the moisture is 'Wet' .

relation can find 'Sitka spruce' with yields 18 and 12
   if  the site is 'North Scotland'
   and the soil is 'Gley'
   and the gley is 'Surface water gley' .

relation can find 'Sitka spruce' with yields 12 and 11
   if  the site is 'North Scotland'
   and the soil is 'Gley'
   and the gley is 'Flushed gley' .

relation can find 'Lodgepole pine' with yields 10 and 8
   if  the site is 'North Scotland'
   and the soil is 'Gley'
   and the gley is 'Unflushed gley' .

relation can find 'Sitka spruce' with yields 14 and 11
   if  the site is 'North Scotland'
   and the soil is 'Peat'
   and the peat is 'Flushed peat'
   and the grass is 'Blue moor grass' .
```

```
relation can find 'Lodgepole pine' with yields 10 and 8
   if  the site is 'North Scotland'
   and the soil is 'Peat'
   and the peat is 'Flushed peat'
   and the grass is 'Heather' .

relation can find 'Lodgepole pine' with yields 10 and 7
   if  the site is 'North Scotland'
   and the soil is 'Peat'
   and the peat is 'Unflushed peat' .

% 2. The East of Scotland

% Note the use of group ordering in testing the answer to the exposure
% question in the following clause

relation can find 'Sitka spruce' with yields 14 and 12
   if  the site is 'East Scotland'
   and the soil is 'Brown earth'
   and the exposure is at or above 'Moderate' according to exposure_types.

relation can find 'Douglas fir' with yields 14 and 12
   if  the site is 'East Scotland'
   and the soil is 'Brown earth'
   and the exposure is below 'Moderate' according to exposure_types .

relation can find 'Grand fir' with yields 16 and 14
   if  the site is 'East Scotland'
   and the soil is 'Brown earth' .

relation can find 'Hybrid larch' with yields 10 and 8
   if  the site is 'East Scotland'
   and the soil is 'Brown earth' .

relation can find 'Broadleaved' with yields 8 and 8
   if  the site is 'East Scotland'
   and the soil is 'Brown earth' .

relation can find 'Sitka spruce' with yields 12 and 10
   if  the site is 'East Scotland'
   and the soil is { 'Podsol' or 'Ironpan' or 'Intergrade' }
   and the plough is 'Complete' .

relation can find 'Lodgepole pine' with yields 10 and 8
   if  the site is 'East Scotland'
   and the soil is { 'Podsol' or 'Ironpan' or 'Intergrade' }
   and the plough is not 'Complete' .

relation can find 'Sitka spruce' with yields 14 and 10
   if  the site is 'East Scotland'
   and [ the soil is 'Gley' and
         the gley is { 'Surface water gley' or 'Flushed gley' }
         or
         the soil is 'Peat' and
         the peat is 'Flushed peat' ]
   and the grass is not 'Heather' .

relation can find 'Lodgepole pine' with yields 10 and 8
```

```
      if  the site is 'East Scotland'
      and [ the soil is 'Gley' and
            the gley is { 'Surface water gley' or 'Flushed gley' }
            or
            the soil is 'Peat' and
            the peat is 'Flushed peat' ]
      and the grass is 'Heather' .

   relation can find 'Sitka spruce' with yields 12 and 10
      if  the site is 'East Scotland'
      and the soil is 'Gley'
      and the gley is 'Unflushed gley'
      and the grass is not 'Heather' .

   relation can find 'Lodgepole pine' with yields 10 and 8
      if  the site is 'East Scotland'
      and the soil is 'Gley'
      and the gley is 'Unflushed gley'
      and the grass is 'Heather' .

   relation can find 'Sitka spruce' with yields 10 and 9
      if  the site is 'East Scotland'
      and the soil is 'Peat'
      and the peat is 'Unflushed peat'
      and the flush is 'Local' .

   relation can find 'Lodgepole pine' with yields 8 and 8
      if  the site is 'East Scotland'
      and the soil is 'Peat'
      and the peat is 'Unflushed peat'
      and the flush is not 'Local' .

   relation can find 'Lodgepole pine' with yields 6 and 6
      if  the site is 'East Scotland'
      and the soil is 'Skeletal' .

   relation can find 'Scots pine' with yields 6 and 6
      if  the site is 'East Scotland'
      and the soil is 'Skeletal' .

   % 4. The West of Scotland

   relation can find 'Sitka spruce' with yields 16 and 14
      if  the site is 'West Scotland'
      and the soil is 'Brown earth' .

   relation can find 'Sitka spruce' with yields 14 and 12
      if  the site is 'West Scotland'
      and the soil is { 'Podsol' or 'Ironpan' or 'Intergrade' } .

   relation can find 'Sitka spruce' with yields 18 and 16
      if  the site is 'West Scotland'
      and [ the soil is 'Gley' and
            the gley is { 'Surface water gley' or 'Flushed gley' }
            or
            the soil is 'Peat' and
            the peat is 'Basin peat'
            or
```

```
          the soil is 'Alluvium' ] .

relation can find 'Sitka spruce' with yields 14 and 12
   if  the site is 'West Scotland'
   and [ the soil is 'Gley' and
         the gley is 'Peaty gley' and
         the status is 'Good'
         or
         the soil is 'Bog' and
         the bog is 'Flushed bog' ] .

relation can find 'Sitka spruce' with yields 12 and 10
   if  the site is 'West Scotland'
   and [ the soil is 'Gley' and
         the gley is 'Peaty gley' and
         the status is 'Poor'
         or
         the soil is 'Bog' and
         the bog is 'Unflushed bog'
         or
         the soil is 'Peat' and
         the peat is 'Hill peat' ]
   and the moisture is 'Wet' .

relation can find 'Lodgepole pine' with yields 8 and 6
   if  the site is 'West Scotland'
   and [ the soil is 'Gley' and
         the gley is 'Peaty gley' and
         the status is 'Poor'
         or
         the soil is 'Bog' and
         the bog is 'Unflushed bog'
         or
         the soil is 'Peat' and
         the peat is 'Hill peat' ]
   and the moisture is 'Dry' .

% This section determines the provenance for a given
% species according to the conservancy and soil type.

relation obtain seed for 'Sitka spruce' from 'Queen Charlotte Island'
   if  the site is not 'South Scotland'
   or  the site is 'South Scotland'
   and the ss_half is not 'Western half' .

relation obtain seed for 'Sitka spruce' from 'Washington'
   if  the site is 'South Scotland'
   and the ss_half is 'Western half' .

relation obtain seed for 'Lodgepole pine' from 'Alaska'
   if  the site is 'North Scotland'
   and the exposure is 'Low' .

relation obtain seed for 'Lodgepole pine' from 'Southern Inland'
   if  the site is 'North Scotland'
   and the soil is 'Peat'
   and the exposure is 'Moderate' .
```

```
relation obtain seed for 'Lodgepole pine' from 'Skeena River'
   if  the site is 'North Scotland'
   and the soil is not 'Peat'
   and the exposure is 'Moderate' .

relation obtain seed for 'Lodgepole pine' from 'South Coastal'
   if  the site is 'North Scotland'
   and the exposure is 'High' .

% Note, the following relation must use "is" to set the value for "P" to
% the given value. This is because "P" is a "logical" variable.
% If instead of "P" we had "p", we would need to use "becomes" to assign
% the value for the variant "p" to the given value.
% Beware! If we used "p" and "is" this is interpreted as a check that the
% value of "p" is the same as the given value.

relation obtain seed for 'Lodgepole pine' from P
   if  the site is 'East Scotland'
   and if   the soil is 'Peat' and
            the peat is 'Unflushed peat' and
            the elevation is 'Upper'
       then P is 'South Coastal'
       else P is { 'Southern Inland' or
                   'Skeena River' }
       end if .

relation obtain seed for 'Lodgepole pine' from P
   if  the site is 'West Scotland'
   and P is { 'South Coastal'   or
              'Southern Inland' or
              'Skeena River'    or
              'North Coastal' } .

% in this program the provenances for seeds other than 'Sitka spruce' or
% 'Lodgepole pine' are unspecified

relation obtain seed for Species from 'Unspecified Location...'
   if the Species is not { 'Sitka spruce' or 'Lodgepole pine' } .

% The following section calculates the variance in the expected
% yield class according to both the exposure of the location and the
% provenance of the seed

relation compute TotalVariance for Provenance
   if  the exposure returns a variance of ExposureVariance
   and Provenance returns a variance of ProvenanceVariance
   and TotalVariance is ExposureVariance + ProvenanceVariance .

relation 'Low'             returns a variance of  2 .

relation 'High'            returns a variance of -2 .

relation 'Southern Inland' returns a variance of -1 .

relation 'Skeena River'    returns a variance of -1 .

relation 'North Coastal'   returns a variance of -2 .
```

```
relation Reason            returns a variance of 0
   if the Reason is not { 'Low'             or
                          'High'            or
                          'Southern Inland' or
                          'Skeena River'    or
                          'North Coastal' } .

% The following actions output a recommended species of pine to the
% current output

action output Species, Provenance, Yield and Variance to the screen ;
   do  nl
   and writenl( 'Recommended species: ', Species )
   and writenl( 'Provenance:          ', Provenance )
   and writenl( 'Normal yield (ñ3):   ', Yield )
   and writenl( 'Yield revision:      ', Variance )
   and nl .

action  writenl( Label, Value ) ;
   do  write( Label )
   and write( Value )
   and nl .
```

## WATER.KSL Listing

```
/*
    Water Containers Example - Adapted by - Phil Vasey
    ==================================================

    This example implements the water containers problem, as defined in
    'Logic for Problem Solving' by Robert Kowalski.

    Description
    -----------
    Given both a seven and a five litre container, initially empty, the
    goal is to find a sequence of actions which leaves four litres of
    liquid in the seven litre container. There are three kinds of actions
    which can alter the state of the containers:

    i.   A container can be filled.
    ii.  A container can be emptied.
    iii. Liquid can be poured from one contained into the other, until
         the first is empty or the second is full.

    Initial State       Goal

    |      |                 |      |
    |      |                 |      |
    |      |    |      |     |------|    |      |
    |      |    |      |     |      |    |      |
    |   7  |    |   5  |     |   4  |    | Don't|
    |Litres|    |Litres|     |Litres|    | Care |
    |______|    |______|     |______|    |______|

        [Both Empty]

    Running The Example
```

```
    ------------------
    The water containers example lets you choose between having or not
    having a presentation of the explanations for the rules that were
    fired during the solution of the problem.
    The state of the containers is reported at each stage of the solution.
    To run the example enter the following goal at the command line:

    ?- find_container_solution .

    Flex Technical Points
    ---------------------
    The following flex technical points are demonstrated in this example:

    1. The use of dynamically updated scores in forward chaining rules.

    2. The linking of forward chaining rules into a network using groups.

    3. The use of rule explanations.

*/

% the following template allows the words spare capacity to be used in
% place of the word spare

template spare
   spare capacity .


% the initial contents and capacity of the two containers is set here

frame container ;
   default contents are 0 and
   default capacity is 7 and
   default spare capacity is its capacity minus its contents .

instance master is a container .

instance slave is a container ;
   capacity is 5 .

template not_full
   ^ is not full .

template not_empty
   ^ is not empty .

template enough
   ^ contains more than ^ ;
   ^ does not contain more than ^ .

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 transfer operations ...

template fill_up
   fill up ^ .

template empty_out
   empty out ^ .

template fill_from
   fill ^ from ^ .

template empty_into
   empty ^ into ^ .

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 for master ...


rule fill_master
   if    the master is not full
   then  fill up the master
   and   write_action( 'fill master' )
   because the emptier the master the better it is to fill it ;
   score master`s spare capacity .

rule empty_master
   if    the master is not empty
   then  empty out the master
   and   write_action( 'empty master' )
   because the fuller the master the better it is to empty it ;
   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
   and   write_action( 'fill master from slave' )
   because the emptier the master and the fuller the slave the better it
           is to fill the master from the slave ;
   score master`s spare capacity + slave`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
   and   write_action( 'empty slave into master' )
   because the emptier the master and the fuller the slave the better it
           is to empty the slave contents into the master ;
   score master`s spare capacity + slave`s contents .

% The rules for slave ...

rule fill_slave
   if    the slave is not full
   then  fill up the slave
   and   write_action( 'fill slave' )
   because the emptier the slave the better it is to fill it ;
   score slave`s spare capacity .

rule empty_slave
   if    the slave is not empty
   then  empty out the slave
   and   write_action( 'empty slave' )
   because the fuller the slave the better it is to empty it ;
   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
   and   write_action( 'fill slave from master' )
   because the emptier the slave and the fuller the master the better
           it is to fill the slave from the master ;
   score slave`s spare capacity + master`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
   and   write_action( 'empty master into slave' )
   because the emptier the slave and the fuller the master the better
           it is to empty the master contents into the slave ;
   score slave`s spare capacity + master`s contents .

ruleset container_rules
   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 the master is 4  .

% The rules are now linked into a network by declaring groups ...

group start_rules
   fill_master, fill_slave .

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 .

% The following action uses the forward_chain/6 flex support predicate
% to return the list of rules that were used during the solution to
% the problem. This list of rules is then passed through to the explain
% facility, which allows you to look at the explanations for the rules
% that were fired.

action find_container_solution ;
  do  if the answer to solution_presentation is explanation
      then forward_chain(crss(6),misfire,
                  termination_criteria,
                  atn, start_rules,
              RulesUsed )
          and explain( RulesUsed )
      else invoke ruleset container_rules
      end if
  and write( 'the goal state has now been reached' )
  and nl .

question solution_presentation
  How would you like the solution presented? ;
  choose one of explanation, 'no explanation' .

relation termination_criteria
  if the contents of the master is 4 .

relation write_container( 0, Capacity, _ )
  if write( '|_____|') .

relation write_container( Line, Capacity, _ )
  if  Line > Capacity .

relation write_container( Line, _, Contents )
  if  Contents < Line
  and write( '|     |') .
```

```
relation write_container( _, _, _ )
  if  write( '|████|') .

action write_containers
   do  nl
   and check MaxLine is equal to the capacity of master
   and for Line from MaxLine to 0 step -1
       do  write_container( Line, the capacity of master, the contents of
master )
       and tab(8)
       and write_container( Line, the capacity of slave, the contents of
slave )
       and nl
   end for
   and write( master ) and tab(10) and write( slave) and nl and nl .

action write_action( Action )
  do  nl
  and write(Action)
  and nl
  and write_containers .
```

## ANIMAL.KSL Listing

```
/*
    A Simple Taxonomy Example - Phil Vasey
    ======================================

    This example shows how to define an animal taxonomy using the frame
    hierarchy system. A relation is provided that will find species
    according to sets of positive and negative clues.

    Description
    -----------
    The identification in this example is done by finding a frame in the
    hierarchy and then checking that frame's attributes against the list
    of positive and negative clues.

    Running the Example
    -------------------
    The identify/3 relation returns on backtracking each frame in the
    frame hierarchy that has the given positive attributes and not the
    given negative attributes. The following query finds all the animals
    whose habitat is land, whose size is medium and whose meal is not
    meat:

    ?- identify( Class, [habitat-land,size-medium],[meal-meat]).

    Flex Technical Points
    ---------------------
    The following flex technical points are demonstrated in this example:

    1. The use of a flex support predicate to find a currently defined
       frame.

    2. Specific inheritance links.
```

```
    3. Suppressing inheritance.

    4. Specifying more than one parent frame.

    5. Data declarations that are automatically run on compilation.


*/


% the flex support predicate isa_frame/2 is used to get a currently
% defined frame to check against the positive and negative attributes

relation identify(Frame,PosAttributes,NegAttributes)
  if  isa_frame( Frame, _ )
  and for every PosAttr-PosVal is included in PosAttributes
      do  [ check the PosAttr of Frame includes PosVal
            or check the PosAttr of Frame is equal to PosVal ]
  end for
  and for every NegAttr-NegVal is included in NegAttributes
      do  check the NegAttr of Frame does not include NegVal
      and check the NegAttr of Frame is not equal to NegVal
  end for .

% the top-level animal frame containing a single attribute

frame animal ;
  default blood is warm .

% the "mammal" and "bird" frames inherit "blood" from the "animal" frame

frame mammal is a kind of animal;
  default skin is fur and
  default habitat is the land and
  default motions are { walk and swim } .

frame bird is a kind of animal;
  default skin is feather and
  default habitat is a tree and
  default motions are { fly } .

% the "fish" frame gives a new value to the "blood" attribute

frame fish is a kind of animal;
  default skin is scale and
  default habitat is the water and
  default motions are { swim } and
  default blood is cold .

% the "carnivore" frame is not linked directly into the hierarchy

frame carnivore;
  default meal is meat .

frame herbivore;
  default meal is plant .

frame penguin is a bird;
```

```
  default habitat is the land and
  default motions are { walk and swim } and
  default size is medium .

frame 'sea water fish' is a fish;
  default habitat is the sea .

frame 'fresh water fish' is a fish;
  default habitat is a river .

% the "salmon" frame has two parents

frame salmon is a 'sea water fish', 'fresh water fish' .

% the "feline" frame inherits the "meal" attribute from "carnivore"

frame feline is a mammal;
  default tail is { long and furry } and
  default speed is 'very fast' and
  default legs are 4 ;
  inherit meal from carnivore .

frame tiger is a feline;
  default size is large and
  default state is predator and
  default habitat is the jungle and
  default meal is human .

frame cat is a feline;
  default size is medium .

% the "manx" frame suppresses the inheritance of the "tail" attribute

frame manx is a cat;
  do not inherit tail .

frame moggy is a cat;
  default legs are 3.5 .

frame rodent is a mammal;
  default tail is { short and thin } and
  default status is pest and
  default habitat is sewer;
  inherit meal from herbivore .

frame squirrel is a rodent;
  default size is small and
  default tail is { long and bushy } ;
  do not inherit habitat .

frame whale is a mammal;
  default habitat is the sea .

% the following definitions show how instances are specified

instance arthur is a cat ;
  skin is shaggy and
  meal is 'kit-e-kat' and
```

```
  habitat is 'my house' .

instance shere_khan is a tiger ;
  speed is 'quite slow' and
  legs are 3.5 .

instance my_other_tig is a tiger ;
  meal is my_pusscat .

instance joey is a bird .

% the following creates a new instance of the salmon frame on compilation

data new_freddie
  do    freddie is a new salmon
        whose size is large and
        whose habitat is fish_tank
  and   echo( 'freddie has been created' )
  and   nl .
```

## FUNCTION.KSL Listing

```
/*
    Two Function Examples - Phil Vasey
    ==================================
    The two functions defined in this example implement the fibonacci and
    factorial values.

    Description
    -----------
    The definition of the factorial function reads:

    If N is greater than 0
    then the factorial of N
         is N times the factorial of N - 1
    else if N is not greater than 0
         the factorial of N is 1 .

    Running The Example
    -------------------
    The series/3 relation can be used to select which type of function to
    be applied across a given range of integer values, as shown in the
    following query:

    ?- series( factorial, 1, 10 ).

    this query generates the factorial value for all the integers between
    1 and 10 inclusive.

    Flex Technical Points
    ---------------------
    The following flex technical points are demonstrated in this example:

    1. The automatic invocation of functions when requested.


*/
```

```
function factorial( N ) =
   if       N > 0
   then     N * factorial( N-1 )
   else 1 .

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

% when the following relation writes out the function fibonacci(N)
% the value for the function is calculated from N

relation series( fibonacci, Lower, Upper )
   if for N from Lower to Upper step 1
      do write( fibonacci(N) )
      and nl
      end for .

relation series( factorial, Lower, Upper )
   if for N from Lower to Upper step 1
      do write( factorial(N) )
      and nl
      end for .
```