

Flex Tutorial

August 2002

Description:

This tutorial is aimed at providing a start to learning Flex programs on the PC under Windows. Both the code and explanations are applicable to Flex on other platforms.

Modified: Clive Spenser, LPA July 1st 1999
Patricia Hughes, GL7 September 16th 1999
Clive Spenser, LPA December 1st 1999
Patricia Hughes, GL7 December 20th 1999
Clive Spenser, LPA January 19th 2000
Patricia Hughes, GL7 February 11th 2000
Clive Spenser, LPA February 22nd 2000
Clive Spenser, LPA August 9th 2000
Clive Spenser, LPA September 26th 2000
Clive Spenser, LPA December 5th 2000
Clive Spenser, LPA March 8th 2001
Clive Spenser, LPA October 3rd 2001
Clive Spenser, LPA August 2002

1	INTRODUCTION TO FLEX	8
1.1	What is Flex?	8
1.1.1	What are Flex programs?	8
1.1.2	How does Flex relate to Prolog?	8
1.1.3	Constructs in Flex	8
1.1.4	How extendable is Flex?	8
1.1.5	Naming conventions in Flex	8
1.2	Basic Flex Constructs	9
1.2.1	KSL Sentences	9
1.2.2	Conditions and Directives	9
1.3	Installing and starting Flex	9
1.4	Important Tips	9
1.4.1	The Console window	9
1.4.2	Create a new file	9
1.4.3	The Run Menu	10
1.4.4	Analysing syntax errors	10
1.4.5	How are Flex programs compiled?	10
1.4.6	How are Flex programs executed?	10
1.4.7	Restarting Flex?	10
1.4.8	How do I stop Flex (or Prolog) running?	10
1.4.9	Use of semicolons	10
1.4.10	Use of full stop	11
1.4.11	Use of spaces	11
1.4.12	Use of quotes and apostrophes	11
1.4.13	Use of the \$ character	11
1.4.14	Debugging in Flex	11
1.5	Useful Prolog routines	11
1.5.1	Prolog built-in predicates	11
2	STARTING PROGRAMMING	13
2.1	First Program	13
2.2	Questions and answers	14
2.3	Compiling and running queries	14
2.4	Expanding the code	15
2.5	Expanding the code further	15
3	FRAMES AND INSTANCES	17
3.1	Frames and Instances	17
3.2	Attribute Values	19
4	MORE ON ACTIONS	20
4.1	Using the student information base	20

4.2	Scheme for changing a student's status and checking both the old and the new value	20
4.3	A special frame called 'global'	21
4.4	Some aids to formatting	21
5	SOME TECHNICAL NOTES	23
5.1	Comments and punctuation	23
5.2	Numbers	23
5.3	Atoms	23
5.4	Byte Lists	24
5.5	Strings	24
5.6	Variables	24
5.7	Names	24
5.8	Values	24
6	FORWARD-CHAINING PRODUCTION RULES	26
6.1	Rules26	
6.2	Rulesets	26
6.3	Facts and exceptions	29
6.4	Groups (static)	29
6.5	Groups (dynamic)	29
6.5	Questions (dynamic)	30
6.6	Relations	31
6.7	Multiple Relations	33
6.8	Templates	35
7	PROJECTS	36
7.1	Saving a changed database	36
7.2	Changing the database at run-time	37
7.3	Saving the modified database	38
7.4	Retrieving the modified database	38
8	DATA DRIVEN PROGRAMMING	40
8.1	Launches	41

8.2	Constraints	41
8.3	Demons	42
8.4	Watchdogs	43
9	DIRECTIVES AND CONDITIONS	44
9.1	Directives	44
9.1.1	Assignments	44
9.1.1.1	Direct Assignments	44
9.1.1.2	New instances	44
9.1.2	Database Maintenance	45
9.2	Conditions	45
9.2.1	Equality Comparison	45
9.2.2	Direct Comparison	46
9.2.3	Relative Comparison	46
9.2.4	Set Membership	46
9.2.5	Procedure Calls	47
9.3	Conjunctions and Disjunctions	47
9.4	Context Switching	47
10	MISCELLANEOUS	48
10.1	Functions	48
10.2	Importing Records	49
11	TRAVERSING THE FRAME HEIRARCHY	50
11.1	Inherited Values	50
11.2	Identification algorithm	50
12	TROUBLESHOOTING	51
13	USEFUL PROLOG ROUTINES	53
13.1	Listing code	53
13.2	Displaying values	53
13.3	Membership	55
13.4	Misc	55

INTRODUCTION TO THIS TUTORIAL

Flex is a very powerful and versatile expert systems toolkit. In contrast to simple expert system shells, Flex offers an open-ended knowledge-based solution to business problems. Flex is implemented in Prolog, a high-level rules-based language, and has unlimited access to the power of that underlying technology. Flex also has access to whatever the Prolog has access to, and in the case of LPA Prolog for Windows, that includes other applications and processes using various industry standards such as DLLs, DDE, OLE, ODBC and much, much more.

Flex employs a 'Natural Language' style approach to defining knowledge through the provision of a dedicated Knowledge Specification Language, KSL. As with many quasi-NL systems, this can lull developers into a false sense of expectation where rules which look as though they should compile and behave in a certain way, don't. Then starts the painful process of debugging and tracing. Two similar looking KSL statements can map on to totally different underlying structures and behave very differently.

Flex provides an interactive question and answer mechanism, which can be configured and extended by the developer without limit. Again, the underlying Prolog offers various high-level features for extending the User Interface.

Flex stores data within a frame hierarchy with multiple inheritance.

Flex provides a myriad of inferencing technologies, including both brittle and fuzzy rules, and both forward and backward chaining. Forward and backward chaining can be interleaved, so getting the best of both worlds. Flint provides the basis for the uncertainty handling features, and supports Bayesian Updating, Certainty factors and Fuzzy Logic.

A principal aim of this tutorial is to start you off on the right foot, and help you get your syntax right first time, and help you start appreciate the potential of Flex.

1 Introduction to Flex

1.1 What is Flex?

Flex is a software system specially designed to aid the development and delivery of Expert Systems. It is implemented in Prolog but looks much more like standard English than a programming language. This is a feature of its Knowledge Specification Language (KSL) which is very easy to read. Flex is very functional and can carry out most of the procedures needed to build knowledge-based systems.

1.1.1 What are Flex programs?

A Flex program consists of any number of KSL statements. Flex programs are stored in standard ASCII text files, normally with a .KSL extension, and can be edited using either the Flex development environment's internal editor or an external text editor or word-processor. Prolog programs are stored in text files with a .PL extension.

1.1.2 How does Flex relate to Prolog?

Flex is implemented in Prolog. Flex programs can call any user-defined or system-defined Prolog program. The Flex development environment is an extension of the Prolog environment, with complete access to the underlying Prolog. You do not need to know Prolog to use Flex, though being familiar with Prolog syntax and basic Prolog commands can only help. There is a section on useful Prolog routines in this tutorial. Flex (KSL) programs are mapped down onto an internal Prolog-based representation by the Flex parser. By using the `listing` command, which is a standard Prolog routine, you can see the internal (Prolog) representation of your Flex (KSL) code appear in the Console window. You can have both .KSL and .PL files present at the same time. In fact, this is often the case in real applications.

1.1.3 Constructs in Flex

Flex contains many constructs ideal for building knowledge-based systems (frames, instances, rules, relations, groups, questions, answers, demons, actions, functions). You may wish to use some or all of these in building your Knowledge-Based Systems (KBS). You do not have to learn to use all of them at once.

1.1.4 How extendable is Flex?

Flex can recognize and compile both Flex (.KSL files) and Prolog (.PL files). If you find you cannot do what you want in Flex, then it can probably be accomplished in Prolog. If it cannot be done in Prolog, then you can always use the C interface and do it in C, or Java or some other 'low-level' language. LPA Prolog for Windows comes with an Intelligence Server options which maps the built-in Flex GUI components onto a variety of languages including VB, Delphi, Java, C/C++.

1.1.5 Naming conventions in Flex

Often Flex lets you re-use the same name to define different constructs; e.g. you may have a question named `drink`, and a group of the same name. You do not have to make use of this facility and may wish to employ a more explicit naming convention,

for instance, where the question is named `drink_question` or `drink_q` and the group named `drink_group` or `drink_g`.

1.2 Basic Flex Constructs

1.2.1 KSL Sentences

The basic unit of compilation in Flex is the sentence. Sentences are made up of one or more KSL statements containing conditions and/or directives, and are terminated with a full stop.

1.2.2 Conditions and Directives

Conditions test whether or not something is currently true.

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

The context will determine which of these Flex will expect.

1.3 Installing and starting Flex

You should install Flex as per your User Guide. Once installed, you should have a Flex shortcut on your screen. (If not, you can always load the Flex system code up from within the Prolog environment).

Load Flex with a double click on the Flex shortcut. Flex loads in a few seconds.

If you have used WinProlog before, you will recognize the Console window but should notice an extra menu named "Flex". The "File" and "Edit" menus are very similar to other Windows applications. You can open the menus to confirm this.

1.4 Important Tips

1.4.1 The Console window

You should now see the Console window with the standard Prolog prompt:

```
?-
```

You can run your programs by typing in the name of a Prolog or Flex query here. Remember, to put a full-stop (or period) at the end of the query. The Console is used for most all communication between you and the 'system'. It is designed for maximum ease-of-use. You can scroll up and down it, re-execute previous queries, cut-and-paste to and from it and much more.

1.4.2 Create a new file

From the "File" menu, select "New" to open a new file. On modern implementations, v4.3 onwards, you will be asked to chose which kind of file you wish to create, one option being `.KSL`. A new window will be opened called "Untitled".

On older implementations, you are advised to immediately save the file with a .KSL extension. This helps ensure that the system recognises it as a flex file.

The file extension determines which compiler to use. KSL stands for Knowledge Specification Language. Most all other extension are interpreted as Prolog files and use the default Prolog (.PL) compiler. This will result in a syntax error message (Error 42) for Flex KSL code.

1.4.3 The Run Menu

The “Run” menu is used to compile and run your code.

1.4.4 Analysing syntax errors

Pull down the Flex menu and make sure the “Analyse Syntax Errors” item is enabled (ticked) before you enter code. If the system detects any mistakes during compilation, it will make suggestions as to what is wrong. If “Analyse Syntax Errors” is not ticked, you will just get a “Cannot Parse Sentence...” error without any explanations.

Note: the “Check Syntax” option on the “Run” menu refers to standard Prolog (.PL) programs and not to Flex (.KSL) programs (and is disabled for .KSL windows on modern implementations).

1.4.5 How are Flex programs compiled?

The KSL compiler translates Flex programs into an internal Prolog-based representation for use by the Flex run-time system. In the case of a KSL syntax error, the Flex parser attempts to identify where the error occurred and what alternative words would have let the parser continue.

1.4.6 How are Flex programs executed?

There is usually a top-level `action` associated with a Flex program to start it off.

1.4.7 Restarting Flex?

Closing a Flex file does not automatically remove all associated definitions. You can clear memory of all Flex code by typing into the Console window:

```
?- initialise.
```

Alternatively, you can use the “Initialise Workspace” item on the “Flex” menu.

1.4.8 How do I stop Flex (or Prolog) running?

In the event of your program going into a loop (which can easily happen in recursive languages), press the <Control> <Break> at the same time. The <Break> key is normally at the top right of your keyboard. This should activate the abort handler which, in turn, should give you the option to abort the process.

1.4.9 Use of semicolons

Often, semicolons are used as delimiters, for instance within actions, frames etc.

1.4.10 Use of full stop

There must be a full stop at the end of each sentence. It is advisable to precede the full stop with at least one space.

1.4.11 Use of spaces

In general, spaces are not significant. When calling built-in Prolog predicates (like `write/1` etc), extra spaces are often placed after the first and before the last bracket for stylistic reasons. It makes it easier to see the arguments, and does not affect the behaviour of the program. However, spaces within the scope of quotation marks of a quoted argument, say within a write statement, are significant, and will have an effect.

You should always leave a space between a number and the final full-stop of a definition to prevent the final full-stop being interpreted as a decimal point.

1.4.12 Use of quotes and apostrophes

You must be very careful with your quotes in Flex:

```
'this is a quoted atom'  
frame`s slot
```

Prolog files can also include:

```
`this is an LPA string`  
"this is a traditional byte-list Prolog string"
```

1.4.13 Use of the \$ character

One important difference between the Flex and Prolog execution models is that Flex will attempt to `prove` goals rather than directly execute them. This involves a layer of interpretation such that Flex can de-reference (evaluate) any arguments located within the scope of the goal statement, unless told not to by the `$` symbol. Flex de-references its arguments before a call is made. The `$` character can be used to inhibit this and force the use of conventional Prolog pattern matching (called unification).

1.4.14 Debugging in Flex

The trace facility in Flex works at the Prolog level, you may find it useful to familiarise yourself with Prolog syntax. There are a number of Flex directives for setting spy points on various flex constructs, such as: `spy_chain`, `spy_rule(Rule)`, `spy_fact(Fact)`, `spy_slot(Attribute, Frame)`.

In addition, you may always insert write statements within your Flex code at the appropriate points, often followed by a new-line.

1.5 Useful Prolog routines

1.5.1 Prolog built-in predicates

Although you do not need to learn Prolog to use Flex, there are many useful Prolog commands (known as predicates or built-ins) which are likely to be useful. Examples include `write/1` and `nl/0` where `write` and `nl` are the names of the predicates and `/1` and `/0` denote the number of arguments.

Others include: listing/[0,1], member/3, remove/3, writeq/1, tell/1, told, append/3, length/2, ttyflush/0, findall/3, forall/2, nl/0.

2 Starting Programming

2.1 First Program

A minimal Flex program will have at least one action and probably some questions:

In the following code, Flex keywords appear in **bold type** to emphasise them but you just type them in normally. Courier font is used to denote program code. Comments are prefixed with a % (per cent) symbol. We will start by defining a **question** and an **action**. The outline structure for these are:

```
question question_name
  prompt text for question ;
  input datatype ;
  [ because explanation ] .

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

Items in square brackets are optional.

Now enter the following code:

```
question your_name
  Please enter your name ;      % question text to be displayed
  input name .                  % forces the input to be treated
                                % as a character string

action hullo ;
  do ask your_name
  and write( 'hi there ' )
  and write( your_name )
  and nl .                      % notice 'and' is used as a delimiter
```

`write/1` and `nl/0` are built-in Prolog routines, called predicates, which you can use in Flex. When using Prolog predicates, their arguments must be placed in round brackets:

```
rel(Arg1, Arg2 ) .
```

and no space must be present between the predicate name and the opening bracket.

Learn the KSL keywords: `question`, `action`, `and`, `because`, `do`, `input`, `name`,
and the
Prolog predicates: `nl/0`, `write/1`.

2.2 Questions and answers

Flex has a powerful built-in question and answer sub-system. You can simply generate basic questions using reserved words such as **choose one** (indicating that you want a single choice menu to be displayed), **choose some** (indicating that you want a multiple choice menu to be displayed), **input** (indicating that you want a simple dialog box prompt), etc.

Additionally, you can use the ‘Customized Input’ mechanism provided by the **answer is** construct to build more sophisticated dialogs either in Prolog, or using the Intelligence Server, in VB or Delphi, or using ProWeb, in HTML.

Asking a **question** generates a dialog box which, as in the example above, awaits keyboard input. Once the question is answered, the reply is stored in a global variable which has the same name as the question, in this case `your_name`. You can then use `your_name` to access whatever was entered.

Flex also supports ‘Constrained Input’ denoted by the keywords, **input** and **such that** which activate data validation routines at the dialog end.

The keyword **name** denotes a specific Flex data type (character string).

Flex programs often have an **action** to start them off (as above).

2.3 Compiling and running queries

Having made sure you do have a .KSL extension for your window, go to the “Run” menu and choose the “Compile” option to compile your code. You should see:

```
question your_name
action hullo
```

appear in the Console window. If `action hullo` fails to appear, then you may need a carriage return following the final `'and nl .'.`

Now, in the Console window, type `hullo.` at the `?-` prompt and press `<return>`. Be sure to use lower case and end with a full stop. A dialog box should appear ready for you to type in an answer. Type in your name and click the OK button. You can try clicking on the “Explain...” button to see what happens. This links into the `because` clause in your question definition.

You can invoke this action again, by typing it in again at the `?-` prompt, or by scrolling back up the Console window, clicking on the line where you originally typed it in, and then hitting the `<Return>` key. You can also use the “Query” item on the “Run” menu. In the Query dialog, enter the name of the action you wish to execute (you do not need to enter the final full stop here, though having one will not cause any harm), and click on the Run button. The query will be recorded for later re-execution.

You can even execute the question directly by typing in `ask(your_name).` at the `?-` prompt (remember to put the full stop at the end). This can be useful when you just want to develop and test questions on their own. You can display the answer by then writing `prove(write(your_name)).` at the `?-` prompt.

2.4 Expanding the code

Now expand the question to include an explanation for the user:

```
question your_name
  Please enter your name ;
  input name ;
  because I would like to call you personally.
```

Try clicking on the “Explain...” button now to see what happens.

An alternative method of expressing the hullo action above is the hullo1 action below:

```
action hullo1 ;
  do ask your_name
  and write( 'hi there ' - your_name )
  and nl.
```

Notice that `write` can take multiple arguments as long as they are separated by a Prolog ‘operator’ such as `-` or `+` or `*` etc. These are just separators which can allow you to write out multiple arguments. For a full list of Prolog operators consult the on-line help file.

Recompile and try this new code. Don’t forget to click on the “Explain...” button.

2.5 Expanding the code further

Add some code to ask someone their age using `integer`, with something appropriate for the `because` clause, and then expand the definition of `hullo` as follows:

```
question your_age
  Please enter your age ;
  input integer ;                % only accepts integers
  because I would like to know how old you are .

action hullo ;
  do ask your_name
  and ask your_age
  and write( ' Hi ' )
  and write( your_name )        % picks up the name entered
  and write( ' I think ' )
  and write( your_age )         % picks up the age entered
  and write( ' is cool! ' )
  and nl .                      % nl outputs a new line
```

Repeat this code replacing `write` with `echo`. (In some versions of Flex, `echo` does not work with integers, so watch out!). You can replace your five calls to `echo` with just one call:

```
and echo( 'Hi', your_name, 'I think ', your_age, ' is cool' )
```

Notice that `echo` can take multiple arguments. In this case, they are separated by a comma.

<p>Learn the keyword: integer and the Prolog predicate: echo()</p>
--

3 Frames and instances

3.1 Frames and Instances

In Flex, you store knowledge about the data you are trying to model in a frame hierarchy using the keywords **frame** and **instance**. Frames represent classes or outlines of items and instances the actual individuals within those classes. Information is inherited downwards through the hierarchy. The outline structure of a frame is:

```
frame frame_name [ is a Parent_frame_name ] ;
  [ default Attribute_Name is Value ]
  [ and default Attribute_Name1 is Value1 ] ;
  [ inherit Attribute_Name2 from Frame ] .
```

Notice the new keywords **frame**, **default**, **inherit** and **is a** and the use of semi-colon again as a delimiter. The space before the semi-colon is optional.

The following code defines the general notion of students. Each individual student can then be an *instance* of the frame 'student'. Here is what a **frame** looks like:

```
frame student ;
  default nationality is american and
  default nature is studious and
  default discipline is computing and
  default residence is texas and
  default major is undecided .
```

Frames include a number of slots for attributes. The basic attributes of all students are declared to be nationality, nature, discipline, residence and major and we have given these default values - values they possess unless declared otherwise at the individual student instance level. Notice that in Flex, we can introduce new attributes at both the sub-frame (frames whose parents are frames) and instance level.

The outline structure of an **instance** is:

```
instance instance_name [ is a Parent_frame_name ] ;
  [ Attribute_Name is Value ]
  [ and Attribute_Name1 is Value1 ] .
```

Here are some sample instances:

```
instance maria is a student ;           % parent frame is student
  nature is cheerful and
  nationality is spanish and
  discipline is engineering and
  status is sophomore .

instance anton is a student ;
  nature is frivolous and
  nationality is french and
  discipline is mathematics and
  status is freshman and
  interests are {tennis, computing, maria} and
  residence is paris .
```

Notice the use of {} to denote sets in Flex (prefixed with **are** rather than **is**). These map on to Prolog lists which are denoted by [].

Notice, too, that all the values begin with a lowercase letter. Words beginning with a capital letter are taken by Flex (and Prolog) to be variables.

Notice, finally, that **a** (and **an**) are often used just to make the code more readable and the compiler usually ignores them. However they must be used when defining the parents of an instance or a frame as above, i.e. **instance** anton **is a** student. Also, **the** is an optional prefix for use before nouns, see later.

The textual ordering of the attribute-value pairs is irrelevant.

The order in which frames and instances are written affects the order in which they are accessed. Instances are accessed in the same order in which they were created. Flex allows you to create instances with the same name, however, the last created instance will take preference and the earlier one(s) ignored.

The value of an attribute can be obtained by either of the following statements:

```
maria`s nationality      % notice the direction of the apostrophe
the nationality of maria
```

If you have any trouble using the former, you can always use the latter.

You can include these in an action with **do echo** or **do write** to see how they work.

```
action test ;
  do for every S is an instance of student
  do write( S`s discipline ) and nl
end for .
```

In this example, we have used a local ‘logical’ variable, **S**, to link the current instance to the nested **do** directive within the **for** loop. (You’ll find more on control loops in Flex later).

Enter and compile the `student` frame, the instances and `test` action. You should see:

```
frame student
instance maria
instance anton
action test
```

appear in the Console window. Execute `test`. <return> from the `?-` prompt.

You can use **check that** to check that a certain named student exists by:

```
action test1 ;
  do check that R is some student whose name is anton .
```

or use **fail** (a Prolog predicate) to force the action to go back and look for other students and so display all the students:

```
action test2 ;
  do check that R is some student
  and write( R ) and nl and fail .
```

Learn the keywords: frame, instance, default, is, inherit, from, the, of, default, a, an, do, check that, fail, `s.

3.2 Attribute Values

You can think of frames and instances as data structures with a name, three columns and an unspecified number of rows:

Frame: student

attribute name	default value	current value
nationality	american	
nature	studious	
discipline	computing	
residence	texas	
major	undecided	

Instance: maria

attribute name	default value	current value
nationality	american	spanish
nature	studious	cheerful
discipline	computing	engineering
residence	texas	
major	undecided	history
status		sophomore

Enter and compile code for three more students.

Note: Instances inherit their slot values from their parent frames **unless** the values are over-ridden with local (current) definitions.

Note: Frame/instance slot values, both default and current, can be pre-defined in KSL files and/or dynamically created and updated at run-time. Be careful if you use `restart` to clear instances from your workspace - it will clear dynamically created instances but not those pre-defined in your KSL files. `initialise`, on the other hand, clears everything. These can be typed into the Console window (as commands), or included within your programs.

4 More on Actions

4.1 Using the student information base

Prepare a file with the student frame and the 5 instances you have already entered. We are going to alter one student's status from `sophomore` to `junior`. Since there is a fixed set of values that status can have (`freshman`, `sophomore`, `junior` and `senior`), we can put these values into a 'choice box' using **choose from**. We can do the same thing with our 5 students.

Add the following questions to your code, together with an action to call it. Notice that currently `student_q` only contains `anton` and `maria`. Edit the question to include the names of the 3 students you added in Section 3.2 Attribute Values..

```
question student_q
  please choose a student ;
  choose from anton, maria.

question status_q
  please select the new status ;
  choose from freshman, sophomore, junior, senior.
```

Note that the semi-colon tells Flex where the question text ends.

The **choose from** (or **choose one of**) option forces a single choice menu to be created which only allows the user to choose *one* item from the list. To choose several items on a list use **choose some of**. Remember that `status_q` will contain the answer to the question and we can assign it to the student's status attribute using the keyword **becomes**.

Remember, you can test a question by typing, `ask(status_q)`. at the `?-` prompt.

4.2 Scheme for changing a student's status and checking both the old and the new value

1. Use the `student` frame and the five instances you have prepared
2. Choose a student's name using `student_q` question.
3. Show the current status with **echo** or **write** (just to check)
4. Query for the new status, using the question `status_q`
5. Use **becomes** to reset `student_q`'s status attribute
6. **echo** or **write** the new status to screen.

This is the `action` part of the code. You will need to supply the appropriate questions.

```
action change_status ;
  do ask student_q
  and echo( 'Status of', student_q, was, student_q`s status )
  and ask status_q
  and student_q`s status becomes status_q
  and echo( 'Status of', student_q, 'is now', student_q`s status ) .
```

Because some versions of Flex, do not allow you to use question names in conjunction with attributes, you may need the following:

```
action change_status ;
do ask student_q
and check that S is student_q % copies the answer into S
and echo( 'Status of', S, was, S`s status )
and ask status_q
and S`s status becomes status_q
and echo( 'Status of', S, 'is now', S`s status ) .
```

Notice the use of `check that` in conjunction with a local variable, `s`, to capture and pass the name of the student instance that has been entered to the assignment clause later.

Compile and run the program with action: `change_status`.

Learn the keywords: becomes, choose, from, some

4.3 A special frame called 'global'

You sometimes need to store values to be used anywhere in the program. These can be put into a special frame called 'global'. This is what the built-in question and answer mechanism uses.

```
frame global;
default current_year is 1996 and
default college_name is 'Texas College' .
```

4.4 Some aids to formatting

You have already met `nl` for a new line. You can use `tab(N)` to move text across the screen:

```
action action_name;
do write( 'something' )
and tab( 10 ) % tab 10 spaces
and write( 'Something else' ) .
```

Try this. Change the number of spaces and try again.

The `for` loop is very much like that found in any other programming language. Try to explain the following code:

```
action half_screen;
do for N from 0 to 10
do nl
end for .
```

Modify the hullo program of section '2.5 Expanding the code further' to use `half_screen` twice:

```
action hullo ;
do half_screen
and ask your_name
and echo( 'Hi', your_name )
and ask your_age
and write( 'I think ' - your_age - 'is cool!' )
and half_screen .
```

The **for/end for** loop, is a useful control structure. Flex also offers other control constructs such as **if then else**, **repeat until**, **while do**. We can use a for loop and local variable, `Student`, to visit all the students as below:

```
action student_status ;
  do for every Student is some instance of student
    do write( Student )
    and write( ' record is ' )
    and nl
    and tab( 5 )
    and write( 'Nationality is ' )
    and write( Student`s nationality )
    and
      % etc.
  end for
```

You can fill in the rest, compile and test. Note the order in which you do things is important. For example:

```
action do_something ;
  do check that the nationality of X is Y
  and check that X is a kind of student
  and write( X-Y ) .
```

This will generate an 'Unbound frame given for attribute ...' error because when Flex tries to look at `nationality of X` it doesn't know what `X` is. If you reorder the code as below, then Flex will know to first bind `X` to an actual student, and then look for that student's nationality. It will use the first student instance created:

```
action do_something ;
  do check that X is a kind of student
  and check that the nationality of X is Y
  and write( X-Y ) .
```

Note: if you type `do_something`, fail at the `?-` prompt you will see all the students' nationalities and then the word 'no'.

You can avoid the repetition of `check that`, by using square brackets, `[]`, and writing:

```
action do_something_else ;
  do check that [ X is a kind of student
  and nationality of X is Y ]
  and write( X-Y ) .
```

Learn the keywords: `global`, `for`, `to`, `end`, `every`
and
Prolog predicate: `tab()`.

5 Some technical notes

In order to be able to talk intelligently about aspects of programming in a language (whether in Flex or in any other language) we all need to understand and be able to name the basic units of the language.

The basic unit in Flex is a token. Tokens are treated as single items by the compiler. Flex uses various types of Prolog token: punctuation, number, atom, string, byte list and variable, and built on top of these Prolog tokens are the concepts of a KSL name and a KSL value.

5.1 Comments and punctuation

`/*` is a token - it indicates the start of a block comment

`*/` is a token - it indicates the end of a block comment.

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

`%` is a token - indicates the start of a single line comment; the end being the carriage return at the end of the line.

`() [] { } | ! ; ,` are all treated as separate items and each one is a token.

5.2 Numbers

Numbers are either integers or floating-point numbers.

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

5.3 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).

```
apple APPLE h45j apple_cart orangesAndApples
```

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.

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

A *quoted* atom is any sequence of characters delimited by single quotes. You need to quote atoms if the atom has a space in or begins with a capital letter. (Note: you must use two quotes to denote an actual quote character within a quoted atom).

```
'Apple' '123' 'The green man' '^h''ht'
```

For the technically minded: atoms are kept in a dictionary with 32-bit pointers. Every time an atom is encountered in input, it is searched for in this dictionary. Atoms are compact but maintaining the dictionary takes time.

5.4 Byte Lists

Characters enclosed in "double quotes" are treated as *lists* of ASCII characters. Most Prolog implementations do not have a dedicated string datatype, have restrictions on the size and number of atoms you can have and rely on byte-lists as the only way of representing non-atoms collections of characters.

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

For the technically minded: LPA is a 32-bit system. Each character occupies 4 bytes but only one of them is used - this is wasteful. In addition, there is a 4 byte pointer to each character - add the type tags and it turns out that each character uses 10 bytes of memory!!

5.5 Strings

LPA offers a special dedicated string data type for text items denoted by the use of backwards quotes. The maximum length of these `strings` is 64K (much bigger than regular atoms). These strings can be manipulated by the various string handling predicates that LPA supplies and do not occupy space in the atom dictionary. They can also be used as input/output buffers. Whilst you can not reference them explicitly in KSL files, you do have programmatic access to them.

For the technically minded: This method of string storage uses approximately 1.3 bytes per character. This is quite efficient. Maximum length 65535 bytes - compare atoms.

5.6 Variables

A (logical) *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).

```
MrSpock Apple APPLE _23 X.
```

The underscore on its own `_` is the 'don't care' variable. Its contents are not stored.

5.7 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).

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

5.8 Values

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

6 Forward-chaining Production Rules

6.1 Rules

Forward-chaining (production) rules are used by the (meta-level) forward-chaining engine in Flex. They have the following format:

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

The IF part contains the *conditions* (tests) and the THEN part contains *directives* (things to do). A rule fires, that is its directives are carried out, when the rule's conditions are satisfiable and the rule is selected. In the case where more than one rule is satisfiable, there are various techniques that can be employed to determine which rule will actually fire (see Section 6.2 Rulesets on Rulesets below).

Examples:

```
rule check_residence1
  if student_q`s residence is not included in
    {london,texas,kensington}
  and student_q`s status is included in {freshman,sophomore}
  then echo( 'need to book rooms in texas for', student_q ) .

rule check_residence2
  if student_q`s residence is included in
    {london,texas,kensington}
  then echo( 'no need to book rooms in texas for', student_q ) .
```

As an exercise, write one equivalent rule for juniors and seniors called `check_residence3` in which the accommodation should be booked at Kensington.

For now, just check that the rules compile and go to the next section. The rules on their own are insufficient, you need a way to invoke them.

6.2 Rulesets

You will appreciate that in a large application there may be many hundreds of rules. It is convenient to group sets of rules which belong together and may be relevant at one particular stage of processing. Such a group is called a **ruleset**. However, if you don't want to have discrete groups, you can inform Flex of this by using the general definition:

```
ruleset set1
  contains all rules .
```

Once you have defined the ruleset, you need an action which will invoke the ruleset.

Don't enter this yet, read the following paragraph.

```
action residence;
  do write( 'starting check' )
  and invoke ruleset set1
  and write( finished ) .
```

The action invokes the *forward chaining inference engine*. This engine cycles round and round checking the rules in the ruleset until some terminating condition is met or no rules fire. The rules we have written so far do not change anything (check this), therefore no terminating condition will ever be met and so as it stands, this code will loop forever, with the engine applying the same rules over and over again.

As mentioned, one possible way for the forward-chaining session to terminate is when there are no rules left. We can achieve this by using the **update** command in the **ruleset** definition to ensure that all successful rules are removed or all unsuccessful rules are removed:

```
update ruleset by removing each selected rule
```

or

```
update ruleset by removing any unsatisfied rules
```

Think carefully about which of these might be suitable for the above 4 rules. (Hint: the engine stops when no rules fire). Now some code which works: (study the code and read the notes at the end).

```
frame student; % the name of the frame
  default nationality is american and % frame attributes always
  default nature is studious and % have default keyword
  default discipline is computing and
  default residence is texas and
  default major is undecided .

instance maria is a student; % parent frame is student
  nature is cheerful and
  nationality is spanish and
  discipline is engineering and
  status is sophomore and
  residence is madrid .

instance anton is a student;
  nature is frivolous and
  nationality is french and
  discipline is mathematics and
  status is freshman and
  interests are {tennis, computing, maria} and
  residence is paris .

instance margaret is a student;
  nature is sporty and
  nationality is finnish and
  discipline is art and
  status is freshman and
  interests are {tennis, computing, anton} and
  residence is texas .
```

```

rule check_residence1
  if S is an instance of student and not checked( S )
  and S`s residence is not included in {london,texas,kensington}
  and S`s status is included in {freshman,sophomore}
  then echo( 'need to book rooms in Texas for', S )
  and S`s residence becomes texas
  and echo('Residence for', S, 'is now', S`s residence, '- rule1')
  and remember that checked( S ) .

rule check_residence2
  if S is an instance of student and not checked( S )
  and S`s residence is not included in {london,texas,kensington}
  and S`s status is not included in {freshman,sophomore}
  then echo( 'no need to book rooms in kensington for', S )
  and S`s residence becomes kensington
  and echo('Residence for', S, 'is now', S`s residence, '- rule2')
  and remember that checked( S ) .

rule check_residence3
  if S is an instance of student and not checked( S )
  and S`s residence is included in {london,texas,kensington}
  and S`s status is included in {freshman,sophomore,junior,senior}
  then echo( 'no need to book rooms in texas for', S, '- rule3' )
  and remember that checked( S ) .

ruleset set1
  contains all rules ;
  select rule using first come first served .

action residence;
  do restart % this clears any remembered facts
  and echo( starting, check )
  and invoke ruleset set1
  and echo( finished, check ).

```

We need to ensure that the inference engine terminates - remember it keeps cycling through the rules until none fires. The first two students were relatively easy, changing the residence meant that rules 1 and 2 no longer fired. However, all the students now cause rule 3 to fire. For this reason, the fact `checked(S)` is inserted into a database (see Section 6.3 Facts and exceptions on facts and exceptions) and forms a pre-condition for all the rules. When each student has been examined (and checked), no rule now fires and so the forward-chaining cycle terminates.

You'll notice that we've used the `select` keyword in the above ruleset. Flex provides three methods of determining the order in which to select rules: `first come first served`, in which it picks the rules in the order in which they appear in the file; `conflict resolution`, where the best rule is picked according to a score; and `conflict resolution with threshold` which picks the first rule whose score is above a user supplied threshold. This *score* is defined using the `score` clause in rules.

The full outline structure for ruleset is the following:

```

ruleset <ruleset_name
  contains rule1,[rule2,rule3,...];
  [ initiate by doing action ] ;
  [ terminate when conditions ] ;
  [ select rule using selection_criterion ] ;
  [ update ruleset by update_criterion ] ;
  [ when a rule misfires do misfire_criterion ] .

```

6.3 Facts and exceptions

Everything the expert system knows about the current state of its world is contained in frames and instances and in **facts** and **exceptions**.

Flex maintains two databases: one of facts, which are true, and one of exceptions, which are not true. These are entered during run-time and exist only for the duration of the session. They are entered with the keywords `remember that` and removed with the keywords `forget that`. The facts and exceptions will either be obtained as answers to questions or as consequences of applying production rules - where the consequences will contain a line such as `remember that` and/or `forget that...`

Facts can be atoms, strings or predicates. Here are some examples of facts:

```
the sun rises at 5 am
asthma is a disease
danger_level(red)
```

Learn the keywords: rule, ruleset, invoke, remember, forget, that, contains

6.4 Groups (static)

Groups are collections of words. Think of them as type declarations but with a built in ordering. Groups are often used to collect items to feed into a **question**.

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

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

Groups are often used to collect items to feed into a **ruleset**.

```
group initial_rules
    rule1, rule2, rule3 .

ruleset set1
    contains initial_rules .
```

6.5 Groups (dynamic)

Because groups can be dynamically re-computed using `new_group/2`, we can have dynamic questions. We will use this later on to update the database of instances.

```
question student_q
    Choose a student ;
    choose one of student_g .

group student_g
    no_one .

action update_student_group ;
    do check that SL is student_list
    and new_group(student_g, SL) .
```

```

action create_student_list ;
  do for every S is some instance of student
    do include S in the student_list
  end for .

action test ;
  do create_student_list
  and update_student_group
  and ask student_q
  and write( student_q )
  and write( ' ')
  and write ( student_q `s nationality )
  and nl.

```

6.5 Questions (dynamic)

Because questions can also be dynamically computed using `new_question/4`, we can get similar behaviour without using groups.

```

action create_student_list ;
  do for every S is some instance of student
    do include S in the student_list
  end for .

action reset_questions ;
  do remove_frame( global )
  and remove_questions .

action test2 ;
  do reset_questions
  and create_student_list
  and prove( new_question(test2_q, {choose,a,student},
                           single(student_list), none) )

  and write( test2_q )
  and write( ' ')
  and write ( test2_q`s nationality )
  and nl.

```

Note: we need to wrap the call to `new_question/4` within a `prove/1` structure, so as to ensure that the global variable `student_list` is dereferenced correctly; i.e. replaced with the items it contains.

Another example combines groups and questions.

```

question starter_q
  Choose a your question ;
  choose one of alpha, beta, gamma .

group alpha
  aleph, alp, aa .

group beta
  bes, bet, bb .

group gamma
  gimmel, gamm, gg .

action starter( SL ) ;
  do check that SL is starter_q .

```

```

action next ;
  do restart
  and starter( X )
  and new_question( next_q, {choose,an,item}, single(X), none )
  and ask next_q .

```

6.6 Relations

In Flex, relations are used to represent backward-chaining rules and have the following outline. (If you have studied Prolog you may recognise it.)

```

relation relation_name(arg1,arg2,..., argN)
  if condition1
  [ and condition2 ] .

```

Backward-chaining rules are true if each of their sub-goals (i.e. conditions) are also true. As conditions themselves may be named relations, it is possible to have quite a sophisticated execution mechanism.

Relations are like actions. However, whereas actions can only have one definition, relations can have multiple (alternative) definitions.

Relations can be called directly from within an **action** (like the system predicates `echo` and `write`).

Relations can be used in the **if** or **then** part of (forward-chaining) rules.

Relations can call other relations.

Now something for you to try. Check how the following code works:

```

frame student.

instance maria is a student ;
  nature is cheerful and
  nationality is spanish and
  status is freshman and
  residence is madrid.

relation check_residence( S )
  if S`s status is included in {freshman,junior}
  and S`s residence is not included in {texas,london}
  and echo( 'we need to book accommodation for', S, 'at texas' ).

/* this definition is used when the first one fails */

relation check_residence( S )
  if echo( S, 'does not need to have accommodation booked' ).

action test;
  do for every S is an instance of student
    do check_residence( S ) and write( S ) and nl
  end for.

```

This is, of course, very similar to the code used for illustrating `rulesets` earlier. Here, instead of using the forward chaining engine of Flex, we are using the backward chaining of the underlying Prolog engine. This is useful when we want to pass a parameter, something we can't do with rules. The trouble is that if the relation does not evaluate to 'true' the whole program will fail, though any write statements encountered will be executed. That is why we add a second definition of `check_residence(S)`. If the first definition does not succeed, then the second definition will be tried.

Program construction note:

Notice that a student's residence is not available to the (calling) action to be used subsequently. We can overcome this by inserting another (logical) variable into the relation's arguments:

```
relation check_residence1( S, Residence )
    if S`s status is included in {freshman,junior}
    and S`s residence is not included in {texas,london}
    and check that Residence is S`s residence
    and echo( 'we need to book accommodation for', S, 'at texas' ).

/* this definition is used when the first one fails */

relation check_residence1( S, Residence )
    if echo( S, ' does not need to have accommodation booked' )
    and check that Residence is S`s residence.
```

and change the action to take account of the extra argument:

Notice the use of `/*` and `*/` to denote enclosed comments.

```
action test1 ;
    do for every S is an instance of student
        do check_residence1( S, Residence )
            and write( S - Residence ) and nl
    end for .
```

At the point when the `check_residence1` relation is called from within the `test1` action, `S` will already contain a real value (a specific student instance) but `Residence` will be unbound (have no value). In working through the conditions of the `check_residence1` relation, `Residence` will acquire a value (become instantiated). This value will be returned to the action and can be used subsequently. (This cannot be done with rules and is one reason why a relation may be used rather than a rule).

6.7 Multiple Relations

Let's look at a typical usage of backward chaining.

```
relation temp_r
  if temp_q is hot and hot_r .
relation temp_r
  if temp_q is cold and cold_r .

relation hot_r
  if time_q is day and write(' maybe you are overdressed') .
relation hot_r
  if time_q is night and write(' maybe you have a fever') .

relation cold_r
  if time_q is day and write(' maybe you need to eat') .
relation cold_r
  if time_q is night and write(' maybe you need a blanket') .

question temp_q
  'do you feel hot or cold' ;
  choose one of hot, cold .
question time_q
  'is it day or night' ;
  choose one of day, night .
```

Now, we can query this program by typing into the Console:

```
?- temp_r.
```

And by answering the questions, we will get some advice displayed. To re-run the example, type:

```
?- restart, temp_r.
```

Note the usage of **restart** to clear previously given answers.

This combination of relations and questions is very common; and can be used to build quite large diagnostic systems. Notice if you do a listing of question, you should get something like:

```
?- listing( question ).

% question/4
question(temp_q, ['do you feel hot or cold'], single([hot, cold]),
true).
question(time_q, ['is it day or night'], single([day, night]), true).
```

And if you do a listing of one of the relations, you should get something like:

```
?- listing( temp_r ).

% temp_r/0
temp_r :-
  equality(temp_q, hot),
  prove(hot_r).

temp_r :-
  equality(temp_q, cold),
  prove(cold_r).
```


6.8 Templates

Flex provides a template facility to make KSL more readable. Templates are replaced at compile time by text substitution. Carets (^) are used to indicate where any variables may be. The outline structure for a `template` is:

```
template label_to_replace
  positive_template ;
  negative_template .
```

Given the following definition:

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

We can define an action `empty_out/1`, and then say

```
... please empty out jug23
```

instead of

```
... empty_out(jug23)
```

Note: template definitions must go in the file BEFORE any usage of them; so it's best to have them right at the beginning of the file.

Now going back to the example in section 6.5, the relation `check_residence1(S, Residence)` is quite simple to understand but by using a template we can turn it into more ordinary English:

```
template check_residence1
  check that ^ is currently resident at ^ .
  % e.g. check that maria is currently resident at texas
```

The carets (^) show where the arguments (atoms or variables) will be in a relation called `check_residence1`. So we can use this in our code instead of:

```
check_residence1( S, Residence )
```

which must still be defined elsewhere as before. This can be even more useful with relations with many arguments. The action can now read:

```
action test
  for every S is an instance of student
    do check that S is currently resident at Residence
      % in here you can fit code, for example, to count the
      % number of students resident at texas...
  end for .
```

Notice that you can use KSL keywords in your templates (e.g., `check that`), but this is not advisable since it can lead to confusion.

Learn the keywords: group, includes, relation, template
--

7 Projects

As programs get longer, it can become more convenient to split them up into several files (windows), bound together in one Project. Also a project can contain a mixture of both Prolog and Flex files.

Now we will start a new project which contains two files, the first, some Prolog code, and the second a set of Flex frames and procedures. The project shows how we can save and retrieve a changed database.

7.1 Saving a changed database

To save a changed database, we will use some of Flex's underlying Prolog predicates. The first procedure, `get_current_values/2`, finds all the slots using `isa_slot/3` (which is how the attributes are located) and then all the instances using `isa_instance/2` (which is how the instances are located). The 'Is' is then the list of instances and the 'As' is a list of attributes. The save option in the second procedure, `save_values/1`, deletes any previous value of 'Is' and 'As', puts the new one into memory and then saves it using `save_predicates/2` as a single predicate containing as its arguments the lists of I's and A's.

The `retrieve_values/1` predicate gets the saved predicates back out of the file, copies the values into a new set of I's and A's and then pulls them apart into separate `new_slots` and instances.

`findall/3`, `forall/2`, `assert/2`, and `abolish/1` are all built-in Prolog predicates.

Type the following code into a new window and save it immediately into a file with the `.pl` extension (for Prolog). To do this, select the "Prolog Files (*.pl)" option on the "Save as type" menu of the "Save AsÖ." dialogue box.

```
get_current_values( As, Is ):-
    findall( slot(Attr,Frame,Value),
             isa_slot(Attr,Frame,Value),
             As),
    findall( instance(Instance,Frame),
             isa_instance(Instance,Frame),
             Is).

save_values( File ):-
    name
    retractall( my_current_values(_,_) ),
    get_current_values( As, Is ),
    assert( my_current_values(As,Is) ),
    save_predicates( [my_current_values/2], File ).

retrieve_values( File ):-
    abolish( my_current_values/2 ),
    load_files( File, [all_dynamic(true)] ),
    my_current_values( As, Is ),
    restore_values( As, Is ).
```

```

restore_values( As, Is ) :-
    forall( member(instance(Inst,Frame),Is),
            new_instance(Inst,Frame)           % restore the instances
          ),
    forall( member(slot(Attr,Frame,Value),As),
            new_slot(Attr,Frame,Value)        % puts them into slots
          ),
    true.

```

You can check the syntax with the syntax checker and then compile to check that it is OK.

Now you can create your first Project using the “File” menu and “Project” option. This will bring up the "Create Project" dialogue where you should provide a filename for your project – it will be given the file extension .pj. Note that you must have at least one source file open, otherwise the "Project" option will not be available.

When you exit from Flex, you will be asked whether you want to save any changed files. When you restart Flex, use the “Load” option on the “File” menu, select the "Project Files (*.pj)" option on the "Files of type" menu in the "Load" dialogue box. Note that your files will need to be compiled after opening.

This code only saves and retrieves instances and attribute slots. Later, it could be altered to include any links, frames and defaults which have been added at run-time.

7.2 Changing the database at run-time

Open a new file and save it immediately to prevent compilation difficulties - use the extension .KSL as this window will contain only Flex. You should also save the Project again to make sure this new window is included. Use the “File” menu and “Project” option as before to “create” the project with the same name.

Here is some code you can use to try out saving information entered at run-time:

```

frame student .

question s_name
    Enter the student name ;
    input name .

question s_nationality
    Enter the student nationality ;
    input name .

action add_student ;
    do ask s_name
    and ask s_nationality
    and check that S is s_name
    and s_name is a new student
    and the nationality of S becomes s_nationality
    and echo( 'The nationality of', S, is, S`s nationality ) .

% if this echoes correctly then the instance has been created.

```

Compile and query with `add_student` adding one student. (This is ‘run-time’ modification of the database). Note: a more elegant way of expressing the above is:

```

action add_student ;
    do ask s_name
    and ask s_nationality
    and s_name is a new student whose nationality is s_nationality
    and echo( 'The nationality of', s_name, is, s_nationality ) .

```

The next stage modifies the code to enter more than one student, with an exit option.

```

frame student .

question s_name
    Enter the student name, type exit to finish ;
    input name .

question s_nationality
    Enter the student nationality ;
    input name .

action add_students ;                % notice the extra s
    do repeat
        ask s_name and
        check that S is s_name        % same problem as previously
        and if S is not exit
            then [
                new_instance(S,student) % this creates the new instance
                and ask s_nationality
                and S`s nationality becomes s_nationality
                and echo( S, 'is a new student whose nationality is',
                            S`s nationality)
            ]
        else do
        end if
    until S = exit

    end repeat .

```

7.3 Saving the modified database

Now use the modified code of the previous section to enter some students. Type into the Console window:

```

?- save_values( <FileName> ).        % type your filename with
                                     % no extension

```

Go and check that the new file has been saved. It should have the extension .PC for compiled Prolog. Close down Flex.

7.4 Retrieving the modified database

Starting afresh, load and compile your complete project. Now type into the Console window:

```

?- retrieve_values(<FileName> ).      % type your filename with
                                     % no extension

```

This opens the file and retrieves the information that was saved. Now test that it really is there with some code such as:

```
action test ;  
  do for every S is an instance of student  
    do echo('The nationality of', S, is, S`s nationality)  
  end for .
```

If it works, then we can have a program with the following format:

```
action start ;  
  do retrieve_values(<FileName> ).      % at start  
  and add_students  
  and save_values(<FileName> ).      % at end
```

8 Data driven programming

There are four types of procedure which take place automatically when data in frames or instances is added or changed. These are: **launches**, **constraints**, **demons** and **watchdogs**. Sometimes this technique is called procedural attachment and is often found in object oriented systems.

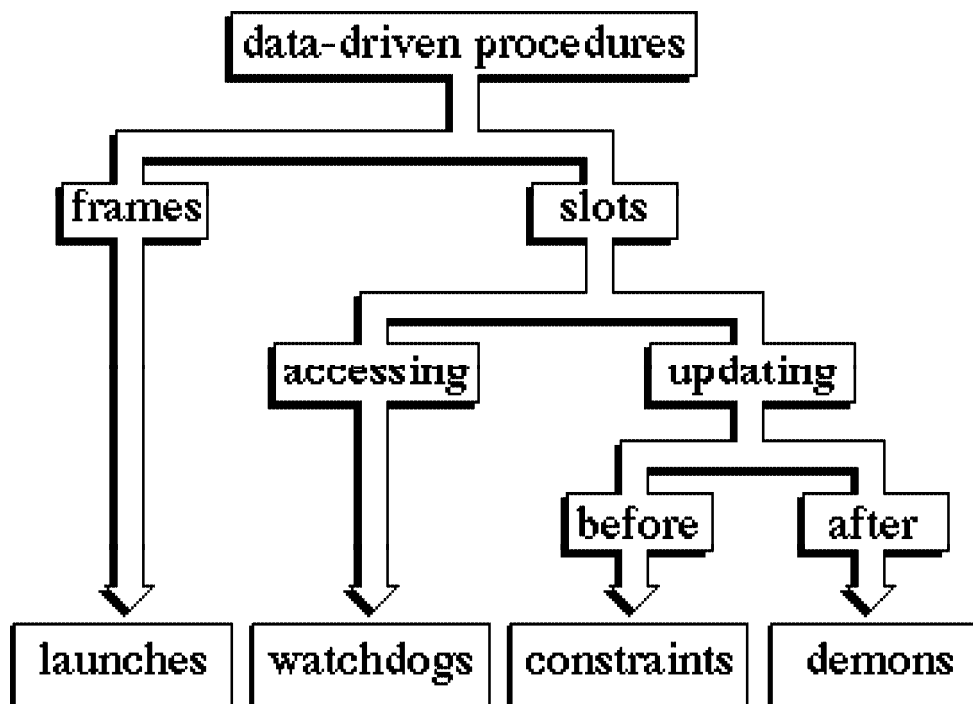
launches keep watch for new instances or frames, carrying out any tests or commands when the new request is received but *before* the instance is created ;

constraints check the validity of any updates to *attributes* and carry out commands when an update is invalid - they check *before* the update is allowed;

demons are called up *after* an attribute has been updated and carry out any commands necessary;

watchdogs are activated when an attempt is made to *access* a slot; consider how you could use this to implement passwords.

These data driven procedures and their relationships to frames/instances and slots are shown in the diagram below as a simple hierarchical tree with nodes and arcs.



8.1 Launches

A `launch` is a procedure which can be attached to a frame 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 outline structure for a `launch` is:

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

Add the following `launch` to the code you developed in Chapter 7 Projects.

```
launch new_student
  when S is a new student
  and S is not exit
  then ask s_nationality .
```

(line/s will need to be deleted from the main code so as not to ask twice.)

The instance will **not be created** until after the `launch` has been successfully completed, which is why there will not be a student called 'exit'. So, '*when S is a new student*' really means '*when S proposes to become a new student*'.

Note that you cannot use this when saving and re-using a runtime modification, i.e., using the **action** `start`. This is because when the information is reloaded, the instances are created again and this will fire the `launch` which will ask the nationality again. This is *not* the desired effect!

8.2 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 the checks succeed. If any 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 outline structure 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) ] .
```

If you have not yet got a slot for credits, alter your code to add one. Retrieve the `change_status` routine from your first file and add a `constraint` which only allows you to choose a status compatible with the number of credits.

Here's a hint to get you going:

```
when the status etc
check that credits < some_number and
check that credits > some_other_number
otherwise.error....
```

Try this with just one to make sure it works.

See whether you can place multiple constraints on one slot.

Would it be better not to ask for the status but to put this in automatically, based on the credits? If so then it could probably be done with a demon - see next section .

Note: With constraints and demons, you must be very careful not to introduce loops. One 'popular' way of doing this is to have a constraint or demon which itself updates a slot-value which in turn invokes the same procedure to be invoked. This will result in the dreaded 'Error 2, local stack full' message. You can best investigate this by including write statements at the appropriate points within your procedures.

Note: The order within your code of constraints and demons relative to frames and instances definitions is significant. Compiling a new frame or instance will run any existing constraint or demon in memory, even ones left over from a previous compilation of the file. If in doubt, initialise the workspace before you recompile. Otherwise you can experience the above problem at compile time as well as run-time.

8.3 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 outline structure for a demon is:

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

Adjust your program not to ask for the status but to accept a change in the number of credits. Use a demon to update the status slot and echo a confirmation to the screen.

Again, check this with just one, e.g., if credits > 30 then *action_name*. If we can't use multiple demons then we could do the same thing using a ruleset with separate rules for each status. Then the action in the demon could 'invoke' the ruleset.

Another possibility is to use four relations :

```
relation status(Credits, freshman)
  if Credits >= 0 and Credits < 30.
```

```

relation status(Credits, sophomore)
  if Credits >= 30 and Credits < 60.

relation status(Credits, junior)
  if Credits >= 60 and Credits < 90.

relation status(Credits, senior)
  if Credits >=90.

```

8.4 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 outline structure of a `watchdog` is as follows:

```

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

```

This is a ‘natural’ for password protection.

```

when the status of student is requested then do password....

```

assuming there is some `action` called `password`. In a real program you might encrypt `password` so that it is not in the source code or can’t be changed easily or depends on who the user is.

Learn the keywords: launch, constraint, demon, watchdog

9 Directives and conditions

Now, let's look at **do** and **check that**.

9.1 Directives

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

9.1.1 Assignments

The changing of global variables, frames and instances is known as *assignment*. There are two kinds of assignment, direct assignments and assignments which happen as the result of creation of a new frame or instance.

9.1.1.1 Direct Assignments

A *direct assignment* consists of a variant on the left hand side and an expression on the right hand side of an assignment operator. 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 { 'Gone With the Wind', 'Star Wars' }
```

9.1.1.2 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

```
'Tiddles' is another cat whose owner is alexander
plant33 is a new plant whose size is medium
```

9.1.2 Database Maintenance

The addition and removal of facts and exceptions is classed as *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 following 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
```

9.2 Conditions

A *condition* 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.

9.2.1 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
```

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 70

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

9.2.2 Direct Comparison

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

```
greater than [or equal to]
[at or] above
less than [or equal to]
[at or] below
```

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
```

9.2.3 Relative Comparison

The *relative comparison* of two terms is determined by their relative positions within a group. 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
```

9.2.4 Set Membership

To test set membership, use the KSL keywords

```
include(s)
included in
do(es) 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
```

Flex uses a technique known as deferencing to expand names into actual arguments. Because groups are not always dereferenced, you need to be careful. For example, this works correctly:

```
group colour
    white, black, green .
```

```
action test ;
  do check that X is some instance of colour
  and write( X ) .
```

whereas the following code, which should work, just fails!

```
action test ;
  do check that X is included in colour
  and write( X ) .
```

9.2.5 Procedure Calls

A condition can be a direct call to some procedure, either a **relation** or **action**, or a Prolog predicate (either built-in or user-defined).

9.3 Conjunctions and Disjunctions

Conditions may be logically combined using **and** and **or**

Examples

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

9.4 Context Switching

If you wish to use a condition where a directive is expected, then the context can be *switched* by inserting the word(s) **check [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 .
```

10 Miscellaneous

10.1 Functions

You can define a function in Flex using any of the following outline structures:

```
function function_name =  
    expression .  
  
function function_name = Variable  
    where condition(s) .  
  
function function_name = Variable  
    if condition(s)  
    then expression (s)  
    else expression (s) .
```

For instance, a mathematical function could be:

```
function taxed( Amount ) = T  
    where T is Amount * 1.175 .  
  
question spending  
    Please enter how much you plan to spend ;  
    input integer .  
  
action spend ;  
    do ask spending  
    and check that X = spending  
    and write( taxed(X) )  
    and nl .
```

In fact, you should not need to introduce the logical variable, try:

```
action spend ;  
    do ask spending  
    and write( taxed(spending) )  
    and nl .
```

If you find you cannot use an arithmetic operator within your function, then just declare it using a Prolog predicate.

```
function ip( A ) = B  
    where T is my_ip(A,B) .
```

% Prolog code follows:

```
my_ip( A, B ) :-  
    B is ip(A).
```

Note: to disambiguate between atoms and functions, you should include an argument in your function definition (even if you don't use it).

```
function twelve( _ ) = 12 .
```

10.2 Importing Records

You can import data by writing some code to connect to a data source, read the records one at a time, and create a new instance for each record.

A sensible mapping would be:

```
TABLE NAME --> frame name
COLUMN NAME --> attribute name
```

So for the table

```
employee( first_name TEXT, last_name TEXT, age INTEGER )
```

we could have:

```
frame employee ;
  default first_name is '' and
  default last_name is '' and
  default age is 0 .
```

Now, we can create a new frame instance for say, Fred, by:

```
X is a new_instance of employee
  whose first_name is 'Fred' and
  whose last_name is 'Smith' and
  whose age is 42 .
```

By downloading the database, we can create a new instance for each row in the table. There is no indexing for frame instances. In general, instances are NOT referenced by their instance identifier but by their attribute values. That is, do not refer to instance 123 of the employee frame, but refer to:

```
X is some instance of employee whose first_name is 'Fred'
```

11 Traversing the Frame Heirarchy

11.1 Inherited Values

The following code will allow us to visit all frames and instances level by level looking for the value of a given attribute.

```
action find_att( Frame, Attribute );
  for every X is a type of Frame
    do if X is an instance of Frame
      then write('instance ') and display_att( Frame, Attribute )
      else write('frame   ') and display_att( Frame, Attribute )
        and find_att( X, Attribute)
      end if
    end for .
```

```
action display_att( Frame, Attribute );
  do fwrite(A, 20. 0, Frame) and tab(15)
  and write( Frame`s Attribute ) and nl .
```

11.2 Identification algorithm

The following Prolog program can be used to find examples of instances with particular attributes, for example all students living at Texas; it also finds examples of instances without certain attributes, for example all students not living at Kensington.

‘Class’ represents the frame or the instances.

‘Positive’ is a list of the positive attributes.

‘Negative’ is a list of the negative attributes.

Type the following code into a Prolog window, one called `utilities` perhaps and add it to every Project that needs it..

```
identify( Class, Positive, Negative ):-
( isa_frame(Class, _); isa_instance(Class, _) ),
forall( member(Attribute-Value,Positive),
        lookup(Attribute, Class, Value) ),
\+ (member(Attribute-Value, Negative),
    lookup(Attribute, Class, Value)).
```

To use it, you supply the values you want. If you don't want to exclude attributes, set Negative to the empty list [] in your call. You can then enter the following into the console window (or include it within an action):

```
?- identify(Name, [status-freshman], []).
```

You should get (something like) as the output:

```
: Name = anton
: Name = margaret
?-
```

12 Troubleshooting

Let's look at some common run-time error messages and unexpected behaviour which can easily be encountered when using Flex. Given the previous code for students and Maria consider the following actions:

```
action temp1;  
  do write( maria`s nationality ).
```

This is correct and we get the answer we expect:

```
spanish
```

```
action temp2;  
  do write( maria`s Nationality ).
```

The attribute is a variable, so we get an error message:

```
Unbound attribute given for frame ... maria
```

```
action temp3;  
  do write( Maria`s nationality ).
```

The frame/instance is a variable, so we get an error message:

```
Unbound frame given for attribute ... nationality
```

```
action temp4;  
  do write( maria`s nnationality ).
```

The attribute nnationality does not exist, so we get an unexpected answer:

```
nnationality@maria.
```

```
action temp5;  
  do write( mmaria`s nationality ).
```

The frame/instance mmaria does not exist, so we get an unexpected answer:

```
nationality@mmaria.
```

```
action temp6a;
do maria`s age becomes 3 + 1
and write(maria`s age).
```

This is correct, Flex can add together two numbers and we get the answer we expect:

4

```
action temp6b;
do maria`s age becomes Y + 1
and write(maria`s age).
```

Because Y is a variable, Flex cannot compute a value and we get an unexpected answer, which is an expression containing a system generated variable:

`_1074+1.`

```
action temp6c;
do maria`s age becomes 3 / 0
and write(maria`s age).
```

Division by zero causes an arithmetic error in the underlying arithmetic expression handler, and we get an error message something like

Error(53) Arithmetic error Call: `_267` is 3/0.

```
action temp6c;
do maria`s age becomes 3 ^ 99 ^ 99 ^ 99
and write(maria`s age).
```

Arithmetic overflow causes an arithmetic error in the underlying arithmetic expression handler, and we get an error message something like

Error(53) Arithmetic error Call: `_291` is 1.842542471780331458e4676^99

```
action temp6d;
do maria`s Age becomes 3 + 1
and write(maria`s age).
```

The attribute is a variable, so we get an error message:

Unbound attribute given for frame ... maria

```
action temp6e;
do Age becomes 3 + 1
and write(maria`s age).
```

Age is a logical variable so we get, so we get an error message

Cannot assign a value to a logical variable ... Age.

13 Useful Prolog routines

Let's look at some useful Prolog routines:

13.1 Listing code

```
?- listing .
```

This displays the current internal state of the Prolog system. We can also list specific things. For example, if you load and compile `Robbie.ksl`, you can do:

```
?- listing( question ) .
```

and get something like:

```
question(shopping, ['What', is, on, your, shopping, list, today, ?],
multiple(goods), text(['I', need, to, check, your, shopping, and,
then, pack, it, into, bags])).
```

```
question(drink, ['You', must, select, a, drink, !], single(drink),
text(['There', are, nibbles, on, your, shopping, list])).
```

and:

```
?- listing( launch ) .
```

should display something like:

```
% launch/5
```

```
launch(new_carrier, A, carrier, true, (prove(write('Need a new
carrier : ')), prove(write(A)), prove(nl))).
```

To list a relation or action, we need to use the real name of the routine.

```
?- listing( choose_bag ) .
```

should display something like:

```
% choose_bag/2
```

```
choose_bag(A, B) :-
    equality(A, (C:some_instance(carrier, C))),
    prove(length(contents@A, D)),
    ( equality(size@B, large)
      -> comparison(<, D, 1)
      ; comparison(<, D, 3) ),
    !.
choose_bag(A, _) :-
    prove(gensym(bag_number_, A)),
    new_instance(A, carrier, []).
```

13.2 Displaying values

We've already seen how `write/1` can be used to display items to the current output channel (normally the Console window). This routine removes any quotation marks; sometimes, we do want to see them:

```
?- write( '1' ) .
1
```

```
?- writeq( '1' ) .  
  '1'
```

```
?- write( 'A' ).  
A
```

```
?- writeq( 'A' ).  
'A'
```

We can output the values of flex items too using a combination of `write/1` and `prove/1`. Given a question named, `drink`, we get:

```
?- write( drink ) .  
  drink
```

```
?- prove( write ( drink ) ) .  
  beer
```

Notice, that if the question does not currently have a value, i.e. it has not yet been asked, then the Flex system will automatically ask the question.

It is often useful to display the current value for a Frame-Slot combination. Here's an action to do just that:

```
action writer( Frame, Slot );  
  do write( Slot of Frame ) and nl .
```

Which generates:

```
?- listing( writer ).
```

```
% writer/2  
writer(A, B) :-  
  prove(write(B@A)),  
  prove(nl).
```

Now we can run:

```
?- writer( beer, condition ).  
liquid
```

Notice if we try displaying the value of a non-existent slot, we get:

```
?- writer( bbeer, condition ).  
condition@bbeer
```

Sometimes, we may wish to truncate the display of floating-point numbers. We can use `fread/4`:

```
?- X is 10/3.  
X = 3.3333333333333333
```

```
?- X is 10/3, fwrite( f, 10, 3, X).  
  3.333
```

```
?- X is 10/3, fwrite( f, 10, 5, X).  
  3.33333
```

```
:- X is 10/3, fwrite( f, 12, 5, X).  
  3.33333
```

13.3 Membership

Sets in Flex, denoted by {}, map on to Prolog lists, denoted by square brackets. Prolog is very strong in the area of list processing, and Flex inherits this.

We can check that a list contains an item using `member/2`.

```
?- member( X, [alp, bet, gam ] ).  
No.1 : X = alp  
No.2 : X = bet  
No.3 : X = gam
```

We can find the position of an item using `member/3`.

```
?- member( X, [alp, bet, gam ], P ).  
No.1 : X = alp, P = 1  
No.2 : X = bet, P = 2  
No.3 : X = gam, P = 3
```

We can delete items using `remove/3`.

```
?- remove( X, [alp, bet, gam ], P ).  
No.1 : X = alp, P = [bet, gam]  
No.2 : X = bet, P = [alp, gam]  
No.3 : X = gam, P = [alp, bet]
```

We can add items using `append/3`.

```
?- append( [delta], [alp, bet, gam ], P ).  
P = [delta, alp, bet, gam]
```

We can find the number of items using `length/2`.

```
?- length( [delta, alp, bet, gam], P ).  
P = 4
```

13.4 Misc

We can switch on the Prolog debugger using:

```
?- trace.
```

and turn it off using:

```
?- notrace.
```

We can re-direct output to a file using:

```
?- tell( 'myresults.txt' ).
```

and reset the current output stream using:

```
?- told.
```

Most output is buffered. If we want to see immediately any output, we use:

```
?- ttyflush.
```