Quick Start

LogiQL is a powerful language containing many features to be described in the upcoming chapters. The text includes numerous embedded examples and exercises that you are encouraged to try out for yourself. There are several ways to actually run your programs including a Web-based interpreter, a sophisticated middleware interface, and the one we choose to use in this book, a command-line interface, called `lb`.

The `lb` command and its subcommands are described in Appendix A, but in this section, we will take you through a short example so you can quickly get started using LogiQL. The example covers how you use `lb` and LogiQL to declare predicates and constraints, define rules, enter data, and query the results. In particular, it computes for each member of the British monarchy those other monarchs who were his or her ancestors.

The first thing you need to do for any application is to create a workspace that will contain your data and program. The `lb create` command can do this for you as follows:

```
lb create royalty
```

Here we give the name to our workspace. The system should respond with the message `created workspace 'royalty'` indicating that the workspace has been successfully created.

As with most programming languages, in LogiQL you need to declare the properties of the data with which you will be working. Our example concerns the British monarchy, so it is natural to declare a data type to represent them. A declaration to do this looks like the following:

```
Monarch(m), hasMonarchName(m:s) -> string(s).
```

The declaration says that a new predicate called `Monarch` exists and that `Monarch` entities will be referred to using their names via the

hasMonarchName predicate, where the monarch name itself is represented internally with a LogiQL string value.

To install this declaration into the royalty workspace, you can use the addblock subcommand of the lb command as follows:

```
lb addblock royalty 'Monarch(m), hasMonarchName(m:s)
   -> string(s).'
```

The command indicates that a new block of code is being added to the royalty workspace. Note the use of apostrophes so that the command shell you are using will ignore any character that has special meaning to it. Then the system responds with the message added block 'block_1Z1I6CCX' where block_1Z1I6CCX is the internal name that the system has given to your program segment.

The data with which your program is concerned has to do with parenthood facts. So you can now declare a predicate to hold this information:

```
lb addblock royalty 'parentOf[m1] = m2 -> Monarch(m1),
  Monarch(m2).'
```

That is, parentOf is a predicate that when given a Monarch argument (m1) responds with another Monarch (m2) that is m1's parent.

In a similar manner, you can add constraints preventing your data from containing some mistakes, such as someone being their own parent:

```
lb addblock royalty '!parentOf[m1] = m1.'
```

Now that you have declared your predicates and a constraint, you can actually enter the data into the workspace. This time you will use the exec subcommand of lb as follows:

```
lb exec royalty '
  +parentOf["George II"]     = "George I".
  +parentOf["George IV"]     = "George III".
  +parentOf["William IV"]    = "George III".
  +parentOf["Edward VII"]    = "Victoria".
  +parentOf["George V"]      = "Edward VII".
  +parentOf["Edward VIII"]   = "George V".
  +parentOf["George VI"]     = "George V".
  +parentOf["Elizabeth II"]  = "George VI".
'
```

In the case of the `exec` subcommand, no feedback is given to you for successful completion. However, if you wish to see the contents of the `parentOf` predicate in the workspace, you can use the `print` subcommand, as follows: `lb print royalty parentOf`.

Your results should look like the following, where the numbers within square brackets are internal identifiers for each of the `Monarch` entities in the workspace:

```
[10000000001] "George II"   [10000000006] "George I"
[10000000002] "George V"    [10000000005] "Edward VII"
[10000000003] "George IV"   [10000000000] "George III"
[10000000004] "Edward VIII" [10000000002] "George V"
[10000000005] "Edward VII"  [10000000012] "Victoria"
[10000000007] "Elizabeth II"[10000000013] "George VI"
[10000000013] "George VI"   [10000000002] "George V"
[10000000015] "William IV"  [10000000000] "George III"
```

You are now ready to write a rule to compute ancestors. An ancestor is either a parent or the ancestor of a parent. This relationship can be expressed in LogiQL with the following code:

```
lb addblock royalty '
  ancestorOf(m1, m2) -> Monarch(m1), Monarch(m2).
  ancestorOf(m1, m2) <- parentOf(m1, m2).
  ancestorOf(m1, m2) <- parentOf(m1, m3),
      ancestorOf(m3, m2).
'
```

To see the results, you can use the `lb query` subcommand. A LogiQL query looks much like a rule, except that the name of the predicate being defined begins with an underscore, such as in the following. (Note, we have also used `lb echo` to dress up the output a bit.)

```
lb echo "Ancestors of British monarchs who were also
monarchs"
lb query royalty '
   _(s1, s2) <-
       ancestorOf(m1, m2),
       hasMonarchName(m1, s1),
       hasMonarchName(m2:s2).
'
```

Here are the results you should expect to see:

```
"Ancestors of British monarchs who were also monarchs"
/– – – – – – – – - _– – – – – – – – -\
"Edward VII"    "Victoria"
"Edward VIII"   "Edward VII"
"Edward VIII"   "George V"
"Edward VIII"   "Victoria"
"Elizabeth II"  "Edward VII"
"Elizabeth II"  "George V"
"Elizabeth II"  "George VI"
"Elizabeth II"  "Victoria"
"George II"     "George I"
"George IV"     "George III"
"George V"      "Edward VII"
"George V"      "Victoria"
"George VI"     "Edward VII"
"George VI"     "George V"
"George VI"     "Victoria"
"William IV"    "George III"
\– – – – – – – – - _– – – – – – – – -/
```

Finally, to clean up when you are done, you can delete the workspace using `lb delete royalty`. You should then expect to see the following message: `deleted workspace 'royalty'`.

You have now completed a very quick run through the basic process of entering and executing a LogiQL program. The details of the language constructs and the situations in which you should use them are described in detail in the chapters to follow. Enjoy!