```
     Ctrl<Key>O:no-op(RingBell) \n\
     <Key>Linefeed:no-op(RingBell)
```

In the supplied configuration files an accelarator line is provided with the following additional line.

```
     <Key>Return:doAction(moveto_action) \n\
```

1.7.10 Entry List Default Translations

```
Xlookup*listForm.translations: #override \
     <Btn1Down>, <Btn1Up>:Set() Notify() \n\
     <Btn2Down>, <Btn2Up>:Set() Notify()
```

In the supplied configuration files the user is allowed to read a listed entry by selecting with the first mouse button and browse below an entry by selecting with the second mouse button. This is performed with the following modification to the default translation.

```
Xlookup*listForm.translations: #override \
 <Btn1Down>, <Btn1Up>: Set() Notify() doAction(read_action) \n\
 <Btn2Down>, <Btn2Up>: Set() Notify() doAction(browse_action)
```

2. Miscellaneous Options

2.1 X Application Defaults File

A core application defaults file is read by Xlu at startup. It is possible to define an additional, or overriding, application defaults file from within *xlurc*, by using the *AppDefFile* option.

```
<AppDefaults> ::= ``AppDefFile'' ``:'' <filename>
<filename> ::= <string>
```

Xlu searches for the application defaults file in either the system wide configuration directory unless the filename is prepended with ‘‘/’’ (meaning an absolute file name), or ‘‘~’’ meaning a file in the users home directory.

2.2 Startup Windows

```
<StartupWindow> ::= ``Startup'' ``:'' <group_name_list>
<group_name_list> ::= <group_name>
                    | <group_name> ``,'' <group_name_list>
<group_name> ::= <label>
```

This option declares the window(s) to be displayed at start up time.

2.3 Bitmap Directory

```
<BitmapDirectory> ::= ``BitmapDirectory''
                      ``:'' <directory_name>
```

Set the directory in which bitmaps are to be found.

2.4 Help Directory

```
<HelpDirectory> ::= ``HelpDirectory''
                    ``:'' <directory_name>
```

Set the directory in which help text files are to be found.

1.7.6 Read Display

This primitive has a system action named *seealso* This allows DN type attributes in an entry to be passed to other windows for display or update.

The following configuration text shows how entry naming attribute can be passed to a window of type read.

```
ACTION : seealso_action : A.Activate
     (L.ReadDisplay, seeAlso, read, ``'', GlobalManaged)
```

Note that the second output window type name is NULL as it is not required.

1.7.7 Abort Directory Request

The *abort* operation causes any directory operations underway in the in the current window (the window from which the action has originated) to be aborted.

The syntax for this operation is:

```
<abort_action> ::= ``A.AbortRequest'' ``('' ``)''
```

1.7.8 Linking Actions to User events

Actions defined in the above manner can be linked to a user event occuring in any primitive. This linkage is made in a non-generic X application defaults file as an event translation resource (see Xt Intrinsics and Athena Widgets manuals for details). See the section Miscellaneous Options which describes how a local application defaults file can be named from within *xlurc*.

As an example, a define action called *quit* can be linked to a button primitive named *QuitButton* with the following line in an application defaults file:

```
Xlookup*QuitButton.translations: #override \
     <EnterWindow>:     set() \n\
     <LeaveWindow>:     unset() \n\
     <Btn1Down>, <Btn1Up>:     doAction(quit)
```

The defined action is invoked using the translation function *doAction* with the action name being its parameter. Notice that some primitives have default translations which must be added to your own local resources if any changes are made. In the above example the button is highlighted and reset on window enter and leave events respectively.

These default translations are detailed in the following sections. The primitive name is defined in *xlurc* using the SETNAME option.

```
Form Dialog Default Translations
Xlookup*PrimitiveName*Text.translations: #override \
     <Key>Tab:     doChangeFocus() \n\
     Ctrl<Key>M:     no-op(RingBell) \n\
     Ctsrl<Key>O:     no-op(RingBell) \n\
     <Key>Linefeed:     no-op(RingBell) \n\
     <BtnDown>:     select-start() doSetFocus()
```

In the supplied configuration files an accelarator line is provided with the following additional line.

```
     <Key>Return:     doAction(lookup_action) \n\
```

1.7.9 UFN dialog Default Translations

```
Xlookup*movetoDialog.translations: #override \
     Ctrl<Key>M:no-op(RingBell) \n\
```

1.7.3 Activate Named Primitive

As well as having user actions associated with them, primitives can also have one or more system actions. For example the *moveto* primitive for entry naming has a system action named *moveto* which performs a directory match on the name currently entered into the dialog.

Full details of all system actions is given later. The syntax used to activate the system action of a primitive is:

```
<activate_action> ::= ``A.Activate''
     ``('' <activate_params> ``)''
<activate_params> ::= <primitive> ``,'' <sys_action_name>
     ``,'' <window_name> ``,'' <window_name>
     ``,'' <raise_style>
<sys_action_name> ::= <string>
<window_name> ::= <group_name>
```

The third and fourth items in the parameter list `<window_name>` refer to the window types that are to be raised and given output as a result of an activation. The window chosen for output depends on the action being performed, for example in the *moveto* action referred to above, the first named window type is used when a single match has been made, the second when two or more matches have been made. Where one or both output window types is not required the NULL string may be placed in the `<window_name>` parameters.

Note that an activation can only be made if the window from which the action originates contains the named primitive possesses the named system action.

All primitives that possess system actions can have user actions associated with them, e.g. the *moveto* primitive has a system action and its user defined user action is called when the RETURN key is pressed. A primitive's user action is not bound to its system action, so if the user action of a *moveto* is activated by pressing RETURN its system action isn't automatically called. This may seem a little strange, but is in fact necessary in order to ensure that the output is always directed to a defined window (using the activate operation) and to allow other operations to be performed prior to or after the system action.

The primitives which have system and user actions are:

1.7.4 Moveto

This has a system action named *moveto*. Here a UFN-like match is made against the string entered into the *moveto* dialogue box. The first output window type is used when a single match is made, the second when two or more matches are made.

The following example shows how the *moveto* operation can be used to send output to windows named read or list, depending on the number of matches made in the name match.

```
ACTION : moveto_action : A.Activate
     (L.MovetoEntry, moveto, read, list, GlobalManaged)
```

1.7.5 Lookup

The *lookup* primitive has a system action named *lookup*. An attribute based search is made on the basis of values entered into the *lookup* form. The output window parameters are consistent with the *moveto* action.

The following example shows how the *lookup* operation can be used to send output to windows named read or list, depending on the number of matches made in the search.

```
ACTION : lookup_action : A.Activate
     (L.Lookup, lookup, read, list, GlobalManaged)
```

1.7 Actions

Some primitive objects may have a user action associated with them. These actions can be linked to user events in an X resource file. In this way an action can be invoked by, say, a button click, or a keyboard press. The action associated with any primitive consists of a list of defined atomic operations, such as raising a window or performing a directory request. The general syntax for action specification is shown below. This is followed by definitions of each of the operations that can be used to make up an action.

```
<ActionDefinition> ::= ``Action'' ``:''
     ``('' <action_list> ``)''
<action_list> ::= ``('' <action_def> ``)''
     | ``('' <action> ``)'' <action_list>
<action_def> ::=  <quit_action>
     | <raise_action>
     | <close_action>
     | <keep_action>
     | <abort_action>
     | <activate_action>
```

1.7.1 Quit Application

Quit from the application. Syntax:

```
<quit_action> ::= ``A.QuitApp''
```

1.7.2 Raise, Close and Keep

These operations allow the user to manage the creation and update of application windows. The keep operation pins a window to the desktop and prevents its contents from being overwritten, i.e. to maintain some particular piece of information. Close pops down a window, whether it has been pinned with a keep or not. The raise operation causes a specified type of window to be created or raised to the top of the window stack, depending on one of four modes of operation:

- **Globally managed**
  A window is created if there are no other windows of the same type currently popped up and none of these have been pinned down, otherwise raise the first window of the same type found.

- **Locally managed**
  Similar to Globally managed except that the rule is only applied to the children of a particular window.

- **New always**
  Always create a new window.

- **Current**
  Raise the current window, i.e. the window from which this action originated.

- **Single**
  Only ever create one window of the given type. A newly raised or created window inherits the window attributes (current entry, etc.) of its parent.

Syntax:

```
<raise_action> ::= ``A.RaiseObject''
     ``('' <object_name> ``,'' <raise_style> ``)''
<raise_style> ::= ``GlobalManaged'' | ``LocalManaged''
     | ``New'' | ``Current'' | ``Single''
<close_action> ::= ``A.CloseObject'' ``('' ``)''
<keep_action> ::= ``A.KeepObject'' ``('' ``)''
```

- **Help text**
  The help text attribute at present only applies to the help display primitive, though the help system may be improved in future release to allow specific help text to be applied to any primitive. Each help text is stored in a separate file. The attribute is set to the name of a file stored in the help directory (see Miscellaneous Options section).

- **Name**
  The name of the X-toolkit widget associated with a primitive. This allows labels and other Intrinsics based resources to be set using an X application defaults file.

The syntax for primitive definition is:

```
<PrimitiveDefinition> ::= <primitive> <prim_attr_list>
<primitive> ::=   ``L.StatusBar''
     | ``L.Button''
     | ``L.MovetoEntry''
     | ``L.Lookup''
     | ``L.TitleBar''
     | ``L.ReadDisplay''
     | ``L.ModifyEntry''
     | ``L.ErrorView''
     | ``L.HelpWindow''
<prim_attr_list> ::=     ``('' <attr_def_list> ``)''
<attr_def_list> ::=      <attr_def> | <attr_def> <attr_def_list>
<attr_def> ::=     ``('' <attr> ``)''
<attr> ::=``BITMAP'' <filename>
          | ``HELPFILE'' <filename>
          | ``HEIGHT'' <number>
          | ``WIDTH'' <number>
          | ``EXPAND'' <boolean>
          | ``INSENSITIVE'' <state_set>
          | ``SETNAME'' <label>
<filename> ::= <string>
<state_set> ::= ``('' <state_list> ``)''
<state_list> ::= <state> | <state> ``,'' <state_list>
<state> ::= ``DSwait''
     | ``NOentry''
     | ``NOlist''
     | ``NOerror''
```

The insensitivity states are:

- **DSwait**
  Insensitive whilst waiting for the results of a directory request.

- **NOentry**
  The parent window doesn't have an associated directory name attribute.

- **NOlist**
  The parent window doesn't have an associated list of directory names.

- **NOerror**
  The parent window doesn't have an associated list of directory errors.

```
Layout : ExampleWindow : VERTICAL
{
     button_group
     L.ReadDisplay
     L.StatusBar
}
Layout : button_group : HORIZONTAL
{
     button_1
     button_2
     button_3
}
```

1.6.2 Primitive Objects

The list of user interface primitives is:

- **Button**
  A button.
- **Status bar**
  A text object displaying status and error messages.
- **Entry display**
  A window object used to display directory entries.
- **Lookup**
  A form dialog for attribute based directory searches.
- **Moveto**
  A dialog for UFN based directory searches.
- **Entry list**
  A list box used to display directory entries.
- **Help**
  A viewport used to display help text and/or graphics.
- **Modify form**
  A form dialog for entry addition, modification, renaming and deleting.

Each primitive has a number of settable attributes. These are:

- **Width and Height**
  The dimensions of the object.
- **Expand**
  A boolean defining how the object behaves when a window is resized (i.e. attempt to grow or not).
- **Insensitivity**
  The states in which a primitive becomes insensitive to keyboard and mouse events. The allowed states are: waiting for result of a directory request, no current entry, no current list of entries or no current list of errors.
- **Actions**
  Some primitives can have user actions associated with them. These actions are invoked by linking them to user events. For example, an action to read and display entries may be attached to a mouse button event in a window containing a list of entries. Actions are described in detail in the next section.

The description label is included for future support, and must always be specified, though only an empty string makes sense at this stage.

The syntax of an entry class definition is:

```
<EntryClassDefinition> ::=
          ``ENTRYclass'' ``:''
          <label> ``:''
          <root_object_class> ``:''
          <rdn_type> ``:''
          <objectclass_list> ``:''
          <group_list>
<label> ::= <string>
<root_object_class> ::= <objectclass>
<rdn_type> ::= <attribute>
<objectclass_list> ::= <objectclass>
         | <objectclass> ‘‘,’’ <objectclass_list>
<group_list> ::= <string_list>
```

The default entry classes are: Person, Job, Department, Organization and Room.

1.6 User Interface

The front end to Xlu is highly configurable, enabling interfaces to be constructed that are adapted to particular tastes or a particular task.

The following sections describe the approach taken towards front end tailoring and detail the syntax used in the tailor file.

1.6.1 Windows

Application windows have a number of pieces of associated data: a entry name, a list of entry names and a list of errors. These are referred to as the attributes of a window. They are not fixed and will change as the results of searches and other directory requests come in. In some circumstances they can be passed between windows, e.g. when a search initiated in one window results in single match, the resultant directory name might be passed to another window in order to be read and then displayed.

Windows are formed from groups of primitive objects. Primitives are the building blocks of the interface. An example of a primitive is a button object (the full list is described in later sections). A group can consist of a single primitive or itself contain sub-groups. Windows are thus constructed from nested groups of primitive objects. Further, the layout of a window can be specified by flagging groups as having vertical or horizontal orientation.

The syntax used to define windows is:

```
<WindowDefinition> ::= <group_definition>
<group_definition> ::= ``Layout'' ``:'' <label>
     ``:'' <orientation> ``:''
     ``{'' <group_component_list> ``}''
<group_component_list> ::= <group_component>
     | <group_component> <group_component_list>
<group_component> ::= <PrimitiveDefinition>
     | <group_definition>
<orientation> ::= ``VERTICAL'' | ``HORIZONTAL''
```

The following examples show how this might be used to define a window containing a top row of three horizontal buttons and a read window and status bar arranged vertically below them. Note that the syntax for primitive definition is given in the next section.

```
     Organization:
     # first search under ``c=GB''
     # (has objectclass=country),
     ``c=gb'' 2.5.6.2,
     # if that fails try ``c=US''
     # (has objectclass=country).
     ``c=us'' 2.5.6.2
SEARCHrule :
     # When searching for a person
     Person :
     # and only a Department value
     # has been supplied
     Department :
     # search under ``Brunel University''
     # (has objectclass=organization)
     ``c=GB@o=Brunel University'' 2.5.6.4
```

Note that entries can be specified by a reference to the entry database rather than with a full DN.

1.5 Entry Update and Addition

The behaviour and presentation of the entry update form depends on the class of the entry being modified. In the context of entry update, etc., the class of an entry is specified by the following pieces of information:

- **Root object class**
  The root object class is used to identify the class of an entry, this is usually a straightforward choice, e.g. the root object class of a person's entry is person.

- **Label**
  A reference label.

- **RDN type**
  The attribute value to be used for the RDN of the entry when renaming or adding a new entry. Multi valued RDNs are not yet supported.

- **Object class set**
  The object class set of the entry (not including the root object class). This defines the attributes that can be present in the entry on an optional or mandatory basis.

- **Attribute groups**
  An entry can have a set of user accessible attributes. This set of attributes is divided into a set of logical attribute groups, e.g. a Personal attribute group may contain commonName, surname, telephoneNumber, etc. Attribute groups are accessed individually. Entry information is thus presented in a piecemeal and structured fashion.

Firstly an attribute group is defined as follows:

```
<AttributeGroupDefinition> ::= ``ATTRgroup'' ``:''
     <label> ``:''
     <description> ``:''
     <attribute_list>
<label> ::= <string>
<description> ::= <label>
<attribute_list> ::=   <attribute>
                     | <attribute> ``,'' <attribute_list>
```

Note that the objectclass of an entry given in the default path is required by the search algorithm.

The syntax for default paths is:

```
<PathSpecification> ::= ``SEARCHrule'' ``:''
                        <target_class_list> ``:''
                        <context_class_list> ``:''
                        <path_list>
<target_class_list> ::= <label_list>
<search_class_list> ::= <label_list>
<label_list> ::= <string_list>
<path_list> ::= <dn> <objectclass>
              | <dn> <objectclass> ``,'' <path_list>
```

1.4.4 Examples

This is all quite complex. The example below should help things to become clearer. First some search classes are defined which tell Xlu which classes of entry will be referenced, and how they may be searched for:

```
searchClass :
     Person :   # Class label
     2.5.6.6 :  # objectClass of class is person
     2.5.4.3 :  # Search on cn attribute
     2.5.6.4,   # Search for a person under orgs
     2.5.6.5    # and orgunits
searchClass :
     Organization :
     2.5.6.4 :      # objectClass of class is organization
     2.5.4.10 :     # Search on orgName attribute
     2.5.6.2        # Parent can only be country
searchClass :
     Department :
     2.5.6.5 :    # objectClass of class is orgUnit
     2.5.4.11 :   # Search on ouName attribute
     2.5.6.4      # Parent cam only be organization
searchClass :
     Country :
     2.5.6.2 :        # objectClass is country
     0.9.2342.19200300.99.1.8 : # Search on friendly country
     ``''                       # Parent can only be root
```

Now define a target class corresponding to the search class ‘‘person’’ above:

```
TARGETclass :
     Person :  # Person has these
     Country, Organization, Department # context classes...
```

The default search path for target class ‘‘Person’’ can now be defined.

```
SEARCHrule :
     # When searching for a person
     Person :
     # and an Organization value
     # has been supplied
```

search class comprises:

- **Object class**
  Defines the type of entry, e.g. person.

- **Search types**
  A set of attributes that can be used to search for an entry, e.g. commonName can be used to search for a person.

- **Parent types**
  A set of object classes that defines the types of entry that may be parents of entries of this class. Note that ancestors more than one level above can be included if that ancestor is likely to be at the second (or lower) level of the DIT, i.e. at a level where subtree searching becomes reasonable.

- **Label**
  Used to key a search class in the configuration file.

The syntax for search classes is;

```
<SearchClassDefinition> ::= ``SEARCHclass'' ``:''
                            <label> ``:''
                            <objectclass> ``:''
                            <search_types> ``:''
                            <parent_types>
<label> ::= <string>
<search_types> ::= <attribute_list>
<parent_types> ::= <objectclass_list>
```

1.4.2 Target class

A class of entry that can be searched for by the user. A target class must also have an associated search class definition. Target classes are defined by:

- **Target name**
  A reference label. This must tally with the corresponding search class label.

- **Target class**
  The search class of the target entry class.

- **Context classes**
  A list of search classes that may be used in order to locate the target. For example when searching for a target of class Person, the context classes might be Country, Locality, Organization, Department.

The syntax for target classes is:

```
<TargetClassDefinition> ::= ``TARGETclass'' ``:''
                            <label> ``:''
                            <context_class_list>
<label> ::= <string>
<context_class_list> ::= <label_list>
```

1.4.3 Default path

The choice of default search path for form searches depends on the target class and the values supplied within the set of context classes. This means that a path is used when a corresponding set of context classes has been given values, i.e. when searching for a target of class Person then a different set of defaults might be required when no context values have been supplied than when values for Department and Organization have been supplied.

1.1 Entry and Attribute Display

The following options control the attribute type which are displayed on screen and the names used to label them.

1.1.1 Attribute Naming

Attribute types can be given user friendly names, e.g. ‘‘Name’’ instead of ‘‘commonName’’, ‘‘Department’’ instead of ‘‘organizationalUnit’’.

```
<AttributeName> ::= ``ATTRname'' ``:'' <attr_name_list>
<attr_name_list> ::=   <attr_name> ``,'' <attr_name_list>
                     | <attr_name>
<attr_name> ::= <attribute> <friendly_name>
<friendly_name> ::= <string>
```

1.2 Entry Database

Search paths are specified by reference to a database of entries. Each entry in the database is given a unique label, which is used to key that entry (for configuration purposes only).

```
<EntryDBSpec> ::= ``ENTRYdb'' ``:'' <key> ``:'' <dn>
<key> ::= <string>
```

References to the entry database must be made after the database has been defined.

1.3 User Friendly Naming

A local context for UFN type searches is specified by the *UFNpath* option.

```
<UfnSearchPath> ::= ``UFNpath'' ``:'' <range>
                    ``:'' <dn_list>

<dn_list> ::= <string_list>
```

The *UFNpath* option provides a local context for UFN type searches. Here a set of default base objects is used when a name supplied by the user falls into the appropriate component number range. For example the following path would be used if the two component name *jo random, comp sci* were supplied.

```
UFNpath : 2 : ``c=gb@o=brunel university'', ``c=gb'', ```
```

In this case a match would be attempted by searching under ‘‘o=brunel university’’ first, if this failed a match would be attempted under ‘‘c=gb’’, then finally under the root of the DIT (here represented by an empty string).

Note that references to entries in the entry database can be used instead of full DNs.

1.4 Form Based Search

Configuration of the form based search is fairly complex, though in general only the defaults (as defined in entry database) will require editing.

The configuration syntax has been designed to permit a high degree of flexibility in order to allow searches for various classes of entry and various shapes of DIT.

Form based search is made on the basis of three pieces of information: search class, target and default search path. These are described in the ensuing sections.

1.4.1 Search class

A class of entry and information about that class. This information is used to establish the relationship of a particular type of entry in the DIT, and how it may be searched for. The information defined in a

Configuring Xlookup

Xlookup, or Xlu for short, is an X-windows interface to the X.500 directory. This chapter describes the tailoring options available and how to make use of them.

The main configuration areas are:

- **Local context for search algorithms**
  The local context defines the behaviour of searches made by the user. It targets searches toward parts of the DIT that are most likely to contain required information.

- **Entry presentation**
  The attributes displayed and the names given to particular attribute types.

- **Entry update form**
  The update form for entry addition and modification can be tailored to suit the particular class of entry being updated. For example, complex or otherwise undesirable attributes can be hidden from view, defaults can be given to attribute value, entries can be divided into a logical set of attribute groups.

- **User interface composition and action**
  The content, functional and visual, of the user interface is tailorable. Windows are built from a number of defined building blocks, e.g. a button, a directory entry display, or a message box. Interactions between windows can be defined, e.g. the results from a search made in one window can be ‘‘sent’’ for display to a new or already existing window.

In general though, little editing of the default tailor files is necessary.

1. Running Xlookup

During initialization the system *dsaptailor* file and the users *quipurc* are read as well as an Xlookup specific tailor file, *xlurc*. Read the Quipu manual for details of the *dsaptailor* and *quipurc* files.

Xlookup is invoked with the command *xlu*. The following command line options are accepted.

-t Alternate dsaptailor.

-T Alternate oidtable.

-u Distinguished name of user to bind as.

-p Password.

-l Alternate *xlurc*.

-n Don’t read *.quipurc*.

The following sections describe the various configuration options available in *xlurc*.

Note that in these sections, attributes and object classes `<attribute>` and `<objectclass>` in the given grammars) can be specified by a numeric OID or by the labels applied to them in the OID table. Numeric OIDs are used in examples as these are invariant across OID tables.

String literals can contain any ASCII character if double quoted. Any string that contains spaces, e.g. a distinguished name, must also be double quoted. Numeric values are read as denary and the boolean type is represented by the strings ‘‘TRUE’’ and ‘‘FALSE’’.

String lists are defined as a sequence of comma separated string tokens. Lists of this type are used to define DN and OID (attribute types or object classes) lists.

All keywords and names are checked case-insensitively.