Manual for PROLOGIO: a Prolog-compatible I/O and RPC library for C++

Julian Smart
Decision Support Group
Artificial Intelligence Applications Institute
80 South Bridge
University of Edinburgh
EH1 1HN

April 1995

Contents

Summary

PROLOGIO is a C++ library for performing two main functions:

1. reading and writing a subset of Prolog syntax
2. implementing a remote procedure call facility (RPC) for client-server communication

This document describes the PROLOGIO API (Application Programming Interface). It can be used to develop programs with readable, robust and Prolog-compatible data files, as well as making RPC communication simple to program on UNIX and PCs. The library requires the wxWindows library for XView and Windows 3.

The Prolog I/O and RPC facilities are combined in the one package since they both relate to reading and writing complex structures to and from ASCII strings or files, and use the same YACC and LEX-based parser to do so.

1. Introduction

During the development of the tool HARDY within the AIAI, a need arose for a data file format for C++ that was easy for both humans and programs to read, was robust in the face of fast-moving software development, and that provided some compatibility with AI languages such as Prolog and LISP.

The result was the PROLOGIO library, which is able to read and write a Prolog-like attribute-value syntax, and is additionally capable of writing LISP syntax for no extra programming effort. The advantages of such a library are as follows:

1. The data files are readable by humans
2. I/O routines are easier to write and debug compared with using binary files
3. The files are robust: unrecognised data will just be ignored by the application
4. Inbuilt hashing gives a random access capability, useful for when linking up C++ objects as data is read in
5. Prolog and LISP programs can load the files using a single command

The library was extended to use the ability to read and write Prolog-like structures for remote procedure call (RPC) communication. The next two sections outline the two main ways the library can be used.

1.1. PROLOGIO for data file manipulation

The fact that the output is in Prolog syntax may be irrelevant for most programmers, who just need a reasonable I/O facility. Typical output looks like this:

```
diagram_definition(type = "Spirit Belief Network").

node_definition(type = "Model",
  image_type = "Diamond",
  attribute_for_label = "name",
  attribute_for_status_line = "label",
  colour = "CYAN",
  default_width = 120,
  default_height = 80,
  text_size = 10,
  can_resize = 1,
  has_hypertext_item = 1,
  attributes = ["name", "combining_function", "level_of_belief"]).

arc_definition(type = "Potentially Confirming",
  image_type = "Spline",
  arrow_type = "End",
  line_style = "Solid",
  width = 1,
  segmentable = 0,
  attribute_for_label = "label",
  attribute_for_status_line = "label",
  colour = "BLACK",
  text_size = 10,
  has_hypertext_item = 1,
  can_connect_to = ["Evidence", "Cluster", "Model", "Evidence",
"Evidence", "Cluster"],
  can_connect_from = ["Data", "Evidence", "Cluster", "Evidence",
"Data", "Cluster"]).
```

This is substantially easier to read and debug than a series of numbers and strings.

Note the object-oriented style: a file comprises a series of *clauses*. Each clause is an object with a *functor* or object name, followed by a list of attribute-value pairs enclosed in parentheses, and finished with a full stop. Each attribute value may be a string, a word (no quotes), an integer, a real number, or a list with potentially recursive elements.

The way that the facility is used by an application to read in a file is as follows:

1. The application creates a PrologDatabase instance.
2. The application tells the database to read in the entire file.
3. The application searches the database for objects it requires, decomposing the objects using the PROLOGIO API. The database may be hashed, allowing rapid linking-up of application data.
4. The application deletes or clears the PrologDatabase.

Writing a file is just as easy:

1. The application creates a PrologDatabase instance.
2. The application adds objects to the database using the API.
3. The application tells the database to write out the entire database, in Prolog, LISP or CLIPS notation.
4. The application deletes or clears the PrologDatabase.

To use the library, include "wx.h", "read.h" and link with libproio.a (UNIX) or prologio.lib (PC) and the wxWindows library.

1.2. PROLOGIO for RPC

An RPC package is required because one application cannot simply call into another's program area. The arguments and return value(s) need to be wrapped up at one end and parsed at the other. This library provides a convenient way of doing so, allowing a range of types of data to be passed between applications, namely integers, reals, strings, words (unquoted strings), and lists of these types (which can include further lists). Therefore two main capabilities are required: a way of encoding information in a string, and a way of communicating this string between applications. The basic PROLOGIO functions described above are used for the first, and the wxWindows implementation of DDE is used for the second.

The RPC facility is used in much the same way as the DDE facility in wxWindows, only the base classes to use are rpcServer, rpcClient and rpcConnection. From these derive your own client, server and connection classes, overriding **OnAcceptConnection** (server) and **OnMakeConnection**(client) as usual. You now have a choice of how the server connection responds to calls: either you can install your own **OnCall** handler which accepts and returns

Prolog structures, or you may install an **rpcCallTable** so calls get automatically routed to appropriate C++ functions. If no call table has been installed, the library will call**OnCall** instead.

Note that the topic name when using the RPC package must always be "RPC".

RPC is implemented as follows. When the client calls **rpcConnection::Call** with a **PrologExpr** structure as argument, this structure is written to a string and **Execute** is called. At the server end, the data is parsed back into a **PrologExpr** structure, and either **OnCall** is called, or, if there is a call table, a functor (procedure name) matching the supplied name is searched for. If one is found, the argument types are checked. If they are incorrect, an error structure is sent back to the client. If they are correct, the appropriate user-supplied C++ function implementing this command is called, which returns another **PrologExpr**. This structure is written to a string, and when the client side **Requests** the result, is returned. The client side parses the result and returns a **PrologExpr** as the result of the original **Call**.

To the client application, the **Call** is like a normal procedure call except the name and arguments need to be wrapped up in a **PrologExpr** structure, and the result is also a **PrologExpr**. Similarly, the server application implements a series of functions, one per command, receiving and returning **PrologExpr** structures.

To use the library, include "wx.h", "read.h", "prorpc.h" and link with libproio.a (UNIX) or prologio.lib (PC) and the wxWindows library.

1.3. Availability and location of PROLOGIO

PROLOGIO is distributed as part of the wxWindows GUI library. Anyone interested in using the library can contact Julian Smart, email J.Smart@ed.ac.uk, tel. 031-650-2746.

2. PROLOGIO compilation

To make use of PROLOGIO, you currently need the following:

1. GNU C++ version 2.1 or later, or Microsoft C/C++ version 7 or Visual C++
2. The PROLOGIO library (libproio.a or prologio.lib) and header files
3. The wxWindows library (available from Julian Smart at AIAI)

For UNIX compilation, ensure that YACC and LEX or FLEX are on your system. Check that the makefile uses the correct programs: a common error is to compile y_tab.c with a C++ compiler. Edit the CCLEX variable in make.env to specify a C compiler. Also, do not attempt to compile lex_yy.c since it is included by y_tab.c.

For DOS compilation, the simplest thing is to copy dosyacc.c to y_tab.c, and doslex.c to lex_yy.c. It is y_tab.c that must be compiled (lex_yy.c is included by y_tab.c) so if adding source files to a project file, ONLY add y_tab.c plus the .cc files. If you wish to alter the parser, you will need YACC and FLEX on DOS.

The DOS tools are available at the AIAI ftp site, in the tools directory. Note that for FLEX installation, you need to copy flex.skl into the directory c:/lib.

If you are using Borland C++ and wish to regenerate lex_yy.c and y_tab.c you need to generate lex_yy.c with FLEX and then comment out the 'malloc' and 'free' prototypes in lex_yy.c. It will compile with lots of warnings. If you get an undefined _PROIO_YYWRAP symbol when you link, you need to remove USE_DEFINE from the makefile and recompile. This is because the parser.y file has a choice of defining this symbol as a function or as a define, depending on what the version of FLEX expects. See the bottom of parser.y, and if necessary edit it to make it compile in the opposite way to the current compilation.

To test out PROLOGIO, compile the test program (test.exe), find badcase.txt in the docs directory, and try loading it into the test program. Then save it to another file. If the second is identical to the first, PROLOGIO is in a working state.

3. Bugs and future developments

3.1. Bugs

These are the known bugs:

1. Functors are permissable only in the main clause (object). Therefore nesting of structures must be done using lists, not predicates as in Prolog.

2. There is a limit to the size of strings read in (about 5000 bytes).

3.2. Future developments

None envisaged as yet.

4. Tutorial

This chapter is a brief introduction to using the PROLOGIO package.

First, some terminology. A *Prolog database* is a list of *clauses*, each of which represents an object or record which needs to be saved to a file. A clause has a *functor* (name), and a list of attributes, each of which has a value. Attributes may take the following types of value: string, word, integer, floating point number, and list. A list can itself contain any type, allowing for nested data structures.

Consider the following code.

```
PrologDatabase db;

PrologExpr *my_clause = new PrologExpr("object");
my_clause->AddAttributeValue("id", (long)1);
my_clause->AddAttributeValueString("name", "Julian Smart");
db.Append(my_clause);

ofstream file("my_file");
db.WriteProlog(file);
```

This creates a database, constructs a clause, adds it to the database, and writes the whole database to a file. The file it produces looks like this:

```
object(id = 1,
  name = "Julian Smart").
```

To read the database back in, the following will work:

```
PrologDatabase db;
db.ReadProlog("my_file");

db.BeginFind();

PrologExpr *my_clause = db.FindClauseByFunctor("object");
int id = 0;
char *name = copystring("None found");

my_clause->AssignAttributeValue("id", &id);
my_clause->AssignAttributeValue("name", &name);

cout << "Id is " << id << ", name is " << name << "\n";
```

Note the setting of defaults before attempting to retrieve attribute values, since they may not be found.

5. Class reference

These are the main PROLOGIO classes.

5.1. ClipsTemplate: wxObject

A **ClipsTemplate** object represents a simplified representation of a CLIPS deftemplate. ClipsTemplates can be added to a list stored with the **PrologDatabase** to be written out along with the deffacts contained in the database. They may optionally be used to filter out unwanted slots when the facts are being written out.

ClipsTemplate::ClipsTemplate

void ClipsTemplate(**char ***_name_)

Construct a new template.

ClipsTemplate::AddSlot

void AddSlot(**char ***_slot_name_, **char ***_default = NULL_, **Bool** _multi=FALSE_)

Add a slot to a CLIPS template definition, with optional default and multislot flag (currently only one multislot allowed per template).

ClipsTemplate::SlotExists

ClipsTemplateSlot * SlotExists(**char ***_slot_name_)

Returns a template slot if it exists.

ClipsTemplate::Write

void Write(**ostream&** _stream_)

Write the template to the given stream.

5.2. PrologDatabase: wxList

The **PrologDatabase** class represents a database, or list, of Prolog-like expressions. Instances of this class are used for reading, writing and creating data files.

PrologDatabase::PrologDatabase

void PrologDatabase(**proioErrorHandler** _handler = 0_)

Construct a new, unhashed database, with an optional error handler. The error handler must be a function returning a Bool and taking an integer and a string argument. When an error occurs when reading or writing a database, this function is called. The error is given as the first argument (currently one of PROIO_ERROR_GENERAL, PROIO_ERROR_SYNTAX) and an error message is given as the second argument. If FALSE is returned by the error handler, processing of the PROLOGIO operation stops.

Another way of handling errors is simply to call *GetErrorCount* (page 9) after the operation, to check whether errors have occurred, instead of installing an error handler. If the error count is more than zero, *WriteProlog* (page 11) and *ReadProlog* (page 10) will return FALSE to the application.

For example:

```
Bool myErrorHandler(int err, chat *msg)
{
  if (err == PROIO_ERROR_SYNTAX)
  {
    wxMessageBox(msg, "Syntax error");
  }
  return FALSE;
}

PrologDatabase database(myErrorHandler);
```

void PrologDatabase(**PrologType** *type*, **char ****attribute*,
int *size = 500*, **proioErrorHandler** *handler = 0*)

Construct a new database hashed on a combination of the clause functor and a named attribute (often an integer identification).

See above for an explanation of the error handler.

PrologDatabase::~PrologDatabase

void ~PrologDatabase(**void**)

Delete the database and contents.

PrologDatabase::AddTemplate

void AddTemplate(**ClipsTemplate ****temp*)

Add a CLIPS template definition to the template list associated with the database.

PrologDatabase::Append

void Append(**PrologExpr ****clause*)

Append a clause to the end of the database. If the database is hashing, the functor and a user-specified attribute will be hashed upon, giving the option of random access in addition to linear traversal of the database.

PrologDatabase::BeginFind

void BeginFind(**void**)

Reset the current position to the start of the database. Subsequent *FindClause* calls will move the pointer.

PrologDatabase::ClearDatabase

void ClearDatabase(**void**)

Clears the contents of the database.

PrologDatabase::FindClause

Various ways of retrieving clauses from the database. A return value of NULL indicates no (more) clauses matching the given criteria. Calling the functions repeatedly retrieves more matching clauses, if any.

PrologExpr * FindClause(**long** *id*)

Find a clause based on the special "id" attribute.

PrologExpr * FindClause(**char ***attribute*, **char ***value*)

Find a clause which has the given attribute set to the given string or word value.

PrologExpr * FindClause(**char ***attribute*, **long** *value*)

Find a clause which has the given attribute set to the given integer value.

PrologExpr * FindClause(**char ***attribute*, **float** *value*)

Find a clause which has the given attribute set to the given floating point value.

PrologDatabase::FindClauseByFunctor

PrologExpr * FindClauseByFunctor(**char *** *functor*)

Find the next clause with the specified functor.

PrologDatabase::FindTemplate

ClipsTemplate * FindTemplate(**char ***template_name*)

Finds a named CLIPS template definition associated with the database.

PrologDatabase::GetErrorCount

int GetErrorCount(**void**)

Returns the number of errors encountered during the last read or write operation.

PrologDatabase::HashFind

PrologExpr * HashFind(**char *** *functor*, **long** *value*)

Finds the clause with the given functor and with the attribute specified in the database constructor having the given integer value.

For example,

```
// Hash on a combination of functor and integer "id" attribute when
reading in
PrologDatabase db(PrologInteger, "id");

// Read it in
db.ReadProlog("data");

// Retrieve a clause with specified functor and id
PrologExpr *clause = db.HashFind("node", 24);
```

This would retrieve a clause which is written: `node(id = 24, ..., )`.

PrologExpr * HashFind(**char *** *functor*, **char ****value*)

Finds the clause with the given functor and with the attribute specified in the database constructor having the given string value.

PrologDatabase::ReadProlog

Bool ReadProlog(**char *** *filename*)

Reads in the given file, returning TRUE if successful.

PrologDatabase::ReadPrologFromString

Bool ReadPrologFromString(**char *** *buffer*)

Reads a Prolog database from the given string buffer, returning TRUE if successful.

PrologDatabase::WriteClips

void WriteClips(**ostream&** *stream*)

Writes the database as a CLIPS deftemplate and deffacts file.

PrologDatabase::WriteClipsFiltering

void WriteClipsFiltering(**ostream&** *stream*)

Writes the database as a CLIPS deftemplate and deffacts file, filtering slots in the facts according to the template definitions (e.g. to reduce the size of the output file by not including irrelevant slots).

PrologDatabase::WriteLisp

void WriteLisp(**ostream&** *stream*)

Writes the database as a LISP-format file.

PrologDatabase::WriteProlog

void WriteProlog(**ostream&** *stream*)

Writes the database as a Prolog-format file.

5.3. PrologExpr

The **PrologExpr** class is the building brick of Prolog expressions. It can represent both a clause, and any subexpression (long integer, float, string, word, or list).

PrologExpr::PrologExpr

void PrologExpr(**char ****functor*)

Construct a new clause with this form, supplying the functor name.

void PrologExpr(**PrologType** *type*, **char ****word_or_string*,
Bool *allocate = TRUE*)

Construct a new empty list, word (will be output with no quotes) or a string, depending on the value of *type* (**PrologList**, **PrologWord**, **PrologString**).

If a word or string, the **allocate** parameter determines whether a new copy of the value is allocated, the default being TRUE.

void PrologExpr(long *the_int*)

Construct an integer expression.

void PrologExpr(**float** *the_float*)

Construct a floating point expression.

void PrologExpr(**wxList ****the_list*)

Construct a list expression. The list's nodes' data should themselves be **PrologExpr**s.

The current version of this library no longer uses the **wxList**internally, so this constructor turns the list into its internal format (assuming a non-nested list) and then deletes the supplied list.

PrologExpr::~PrologExpr

void ~PrologExpr(**void**)

Destructor.

PrologExpr::AddAttributeValue

Use these on clauses ONLY. Note that the functions for adding strings and words must be differentiated by function name which is why they are missing from this group (see *AddAttributeValueString* and*AddAttributeValueWord*).

void AddAttributeValue(**char ****attribute*, **float** *value*)

Adds an attribute and floating point value pair to the clause.

void AddAttributeValue(**char ****attribute*, **long** *value*)

Adds an attribute and long integer value pair to the clause.

void AddAttributeValue(**char ****attribute*, **wxList ****value*)

Adds an attribute and list value pair to the clause, converting the list into internal form and then deleting **value**. Note that the list should not contain nested lists (except if in internal **PrologExpr** form.)

void AddAttributeValue(**char ****attribute*, **PrologExpr *****value*)

Adds an attribute and PrologExpr value pair to the clause. Do not delete *value* once this function has been called.

PrologExpr::AddAttributeValueString

void AddAttributeValueString(**char ****attribute*, **char ****value*)

Adds an attribute and string value pair to the clause.

PrologExpr::AddAttributeValueStringList

void AddAttributeValueStringList(**char ****attribute*, **wxList ****value*)

Adds an attribute and string list value pair to the clause.

Note that the list passed to this function is a list of strings, NOT a list of **PrologExpr**s; it gets turned into a list of **PrologExpr**s automatically. This is a convenience function, since lists of strings are often manipulated in C++.

PrologExpr::AddAttributeValueWord

void AddAttributeValueWord(**char ****attribute*, **char ****value*)

Adds an attribute and word value pair to the clause.

PrologExpr::Append

void Append(**PrologExpr ***_value_)

Append the **value** to the end of the list. The **PrologExpr** must be a list.

PrologExpr::Arg

PrologExpr * Arg(**PrologType** *typ*, **int** *n*)

Get nth arg of the given clause (starting from 1). NULL is returned if the expression is not a clause, or *n* is invalid, or the given type does not match the actual type. See also *Nth* (page 15).

PrologExpr::Insert

void Insert(**PrologExpr ***_value_)

Insert the **value** at the start of the list. The **PrologExpr** must be a list.

PrologExpr::AssignAttributeValue

These functions are the easiest way to retrieve attribute values, by passing a pointer to variable. If the attribute is present, the variable will be filled with the appropriate value. If not, the existing value is left alone. This style of retrieving attributes makes it easy to set variables to default values before calling these functions; no code is necessary to check whether the attribute is present or not.

void AssignAttributeValue(**char ***_attribute_, **char ****_value_)

Retrieve a string (or word) value.

void AssignAttributeValue(**char ***_attribute_, **float ***_value_)

Retrieve a floating point value.

void AssignAttributeValue(**char ***_attribute_, **int ***_value_)

Retrieve an integer value.

void AssignAttributeValue(**char ***_attribute_, **long ***_value_)

Retrieve a long integer value.

void AssignAttributeValue(**char ***_attribute_, **PrologExpr ****_value_)

Retrieve a PrologExpr pointer.

PrologExpr::AssignAttributeValueStringList

void AssignAttributeValueStringList(**char** ***attribute*, **wxList** ***value*)

Use this on clauses ONLY. See above for comments on this style of attribute value retrieval. This function expects to receive a pointer to a new list (created by the calling application); it will append strings to the list if the attribute is present in the clause.

PrologExpr::AttributeValue

PrologExpr * AttributeValue(**char** ***word*)

Use this on clauses ONLY. Searches the clause for an attribute matching *word*, and returns the value associated with it.

PrologExpr::Copy

PrologExpr * Copy(**void**)

Recursively copies the expression, allocating new storage space.

PrologExpr::DeleteAttributeValue

void DeleteAttributeValue(**char** ***attribute*)

Use this on clauses ONLY. Deletes the attribute and its value (if any) from the clause.

PrologExpr::Functor

char * Functor(**void**)

Use this on clauses ONLY. Returns the clause's functor (object name).

PrologExpr::GetClientData

wxObject * GetClientData(**void**)

Retrieve arbitrary data stored with this clause. This can be useful when reading in data for storing a pointer to the C++ object, so when another clause makes a reference to this clause, its C++ object can be retrieved. See *SetClientData*.

PrologExpr::GetFirst

PrologExpr * GetFirst(**void**)

If this is a list expression (or clause), gets the first element in the list.

See also *GetLast* (page 15), *GetNext* (page 15), *Nth* (page 15).

PrologExpr::GetLast

PrologExpr * GetLast(**void**)

If this is a list expression (or clause), gets the last element in the list.

See also *GetFirst* (page 14), *GetNext* (page 15), *Nth* (page 15).

PrologExpr::GetNext

PrologExpr * GetNext(**void**)

If this is a node in a list (any PrologExpr may be a node in a list), gets the next element in the list.

See also *GetFirst* (page 14), *GetLast* (page 15), *Nth* (page 15).

PrologExpr::IntegerValue

long IntegerValue(**void**)

Returns the integer value of the expression.

PrologExpr::Nth

PrologExpr * Nth(**int** *n*)

Get nth arg of the given list expression (starting from 0). NULL is returned if the expression is not a list expression, or *n* is invalid. See also *Arg* (page 13).

Normally, you would use attribute-value pairs to add and retrieve data from objects (clauses) in a data file. However, if the data gets complex, you may need to store attribute values as lists, and pick them apart yourself.

Here is an example of using lists.

```
int regionNo = 1;
char regionNameBuf[20];

PrologExpr *regionExpr = NULL;
sprintf(regionNameBuf, "region%d", regionNo);

// Keep getting attribute values until no more regions.
while (regionExpr = clause->AttributeValue(regionNameBuf))
{
  /*
   * Get the region information
   *
   */

  char *regionName = NULL;
```

```
    char *formatString = NULL;
    Bool formatMode = FORMAT_NONE;
    int fontSize = 10;
    int fontFamily = wxSWISS;
    int fontStyle = wxNORMAL;
    int fontWeight = wxNORMAL;
    char *textColour = NULL;

    if (regionExpr->Type() == PrologList)
    {
      PrologExpr *nameExpr = regionExpr->Nth(0);
      PrologExpr *stringExpr = regionExpr->Nth(1);
      PrologExpr *formatExpr = regionExpr->Nth(2);
      PrologExpr *sizeExpr = regionExpr->Nth(3);
      PrologExpr *familyExpr = regionExpr->Nth(4);
      PrologExpr *styleExpr = regionExpr->Nth(5);
      PrologExpr *weightExpr = regionExpr->Nth(6);
      PrologExpr *colourExpr = regionExpr->Nth(7);

      regionName = copystring(nameExpr->StringValue());
      formatString = copystring(stringExpr->StringValue());
      formatMode = (int)formatExpr->IntegerValue();
      fontSize = (int)sizeExpr->IntegerValue();
      fontFamily = (int)familyExpr->IntegerValue();
      fontStyle = (int)styleExpr->IntegerValue();
      fontWeight = (int)weightExpr->IntegerValue();
      textColour = copystring(colourExpr->StringValue());
    }
    wxFont *font = wxTheFontList->FindOrCreateFont(fontSize,
fontFamily, fontStyle, fontWeight);
    ObjectRegion *region = new ObjectRegion;
    region->regionName = regionName;
    region->formatString = formatString;
    region->formatMode = formatMode;
    region->regionFont = font;
    region->textColour = textColour;

    regions.Append(region);

    regionNo ++;
    sprintf(regionNameBuf, "region%d", regionNo);
  }
```

PrologExpr::RealValue

float RealValue(**void**)

Returns the floating point value of the expression.

PrologExpr::SetClientData

void SetClientData(**wxObject ****data*)

Associate arbitrary data with this clause. This can be useful when reading in data for storing a pointer to the C++ object, so when another clause makes a reference to this clause, its C++ object can be retrieved. See *GetClientData*.

PrologExpr::StringValue

char * StringValue(**void**)

Returns the string value of the expression.

PrologExpr::Type

PrologType Type(**void**)

Returns the type of the expression. **PrologType** is defined as follows:

```
typedef enum {
    PrologNull,
    PrologInteger,
    PrologReal,
    PrologWord,
    PrologString,
    PrologList
} PrologType;
```

PrologExpr::WordValue

char * WordValue(**void**)

Returns the word value of the expression.

PrologExpr::WriteClipsClause

void WriteClipsClause(**ostream&** *stream*, **Bool** *filtering=FALSE*, **PrologDatabase** ***database = NULL*)

Writes the clause to the given stream in CLIPS 'deffacts' format. If **filtering** is TRUE, the deftemplates associated with **database** are used to filter out slots which exist in the CLIPS facts but not in the deftemplates (see **PrologDatabase::AddTemplate**).

PrologExpr::WriteLispExpr

void WriteLispExpr(**ostream&** *stream*)

Writes the expression or clause to the given stream in LISP format. Not normally needed, since the whole **PrologDatabase** will usually be written at once. Lists are enclosed in parentheses will no commas.

PrologExpr::WritePrologClause

void WritePrologClause(**ostream&** *stream*)

Writes the clause to the given stream in Prolog format. Not normally needed, since the whole **PrologDatabase** will usually be written at once. The format is: functor, open parenthesis, list of comma-separated expressions, close parenthesis, full stop.

PrologExpr::WritePrologExpr

void WritePrologExpr(**ostream&** *stream*)

Writes the expression (not clause) to the given stream in Prolog format. Not normally needed, since the whole **PrologDatabase** will usually be written at once. Lists are written in square bracketed, comma-delimited format.

5.4. Functions and Macros

Below are miscellaneous functions and macros associated with PrologExpr objects.

Functions

PrologExpr * wxMakeCall(**char** **functor*, **...**)

Make a PrologExpr clause from a functor and a list of PrologExpr objects *terminated with a zero (or NULL)*. Since this is normally used for making a procedure-call expression, the arguments will not be attribute-value pairs but straightforward data types.

char * wxCheckClauseTypes(**PrologExpr ****expr*,**wxList ****type_list*)

Compares the types of the arguments of *expr* (assumed to be a procedure call expression) against a list of types. Returns NULL if no error, or an error string if there is a type error.

char * wxCheckTypes(**PrologExpr ****expr*, **...**)

Compares the types of the arguments of *expr* (assumed to be a procedure call expression) against the remaining arguments, terminated with a NULL. Returns NULL if no error, or an error string if there is a type error.

```
char *s = wxCheckTypes(expr, PrologInteger, PrologReal, 0);
```

Bool wxIsFunctor(**PrologExpr ****expr*, **char ****functor*)

Checks that the functor of *expr* is *functor*.

Macros

The following macros have been defined to make constructing PrologExpr objects slightly easier:

```
#define wxMakeInteger(x) (new PrologExpr((long)x))
#define wxMakeReal(x)    (new PrologExpr((float)x))
#define wxMakeString(x)  (new PrologExpr(PrologString, x))
#define wxMakeWord(x)    (new PrologExpr(PrologWord, x))
#define wxMake(x)        (new PrologExpr(x))
```

Index