The XSB System
Version 2.2
Volume 2: Libraries and Interfaces

April 4, 2000

Credits

Interfaces have become an increasingly important part of XSB. The interface from C to Prolog was implemented by David Warren as was the DLL interface; the interface from Prolog to C (foreign language interface) was developed by Jiyang Xu, Kostis Sagonas and Steve Dawson. The XSB-Java interface was written by Miguel Calejo as was the InterProlog user interface. The Oracle interface was written by Hassan Davulcu and Ernie Johnson. The documentation for XSB's POSIX regular expression and wildcard matching facilities was written by Michael Kifer. The interface to Perl pattern matching routines was written by Michael Kifer and Jin Yu. The ODBC interface was written by Lily Dong and Baoqiu Cui.

David Warren and Prasad Rao implemented the aggregate library. The SLX preprocessor was written by José Júlio Alferes and Luís Moniz Pereira. Unix-style scripting libraries were written by Terrance Swift, and the ordset library was written by Richard O'Keefe.

Contents

Chapter 1

Library Utilities

In this chapter we introduce some useful predicates that are supplied with the system. These predicates are available only when imported them from (or explicitly consult) the corresponding modules.

1.1 List Processing

The XSB library contains various list utilities, some of which are listed below. These predicates should be explicitly imported from the module specified after the skeletal specification of each predicate. There are a lot more useful list processing predicates in various modules of the XSB system, and the interested user can find them by looking at the sources.

`append(?List1, ?List2, ?List3)` basics

Succeeds if list `List3` is the concatenation of lists `List1` and `List2`.

`member(?Element, ?List)` basics

Checks whether `Element` unifies with any element of list `List`, succeeding more than once if there are multiple such elements.

`memberchk(?Element, ?List)` basics

Similar to `member`/2, except that `memberchk`/2 is deterministic, i.e. does not succeed more than once for any call.

`ith(?Index, ?List, ?Element)` basics

Succeeds if the `Index`th element of the list `List` unifies with `Element`. Fails if `Index` is not a positive integer or greater than the length of `List`. Either `Index` and `List`, or `List` and `Element`, should be instantiated (but not necessarily ground) at the time of the call.

`log_ith(?Index, ?Tree, ?Element)` basics

Succeeds if the `Index`th element of the Tree `Tree` unifies with `Element`. Fails if `Index` is not a positive integer or greater than the number of elements that can be in `Tree`. Either `Index` and `Tree`, or `Tree` and `Element`, should be instantiated (but not necessarily ground) at the time of the call. Tree is a list of full binary trees, the first being of depth 0, and each one being of depth one greater than its predecessor. So `log_ith/3` is very similar to `ith/3` except it uses a tree instead of a list to obtain log-time access to its elements.

`log_ith_bound(?Index, ?Tree, ?Element)` basics

is like `log_ith/3`, but only if the `Index`$^{\texttt{th}}$ element of `Tree` is nonvariable and equal to Element. This predicate can be used in both directions, and is most useful with Index unbound, since it will then bind `Index` and `Element` for each nonvariable element in `Tree` (in time proportional to $N * logN$, for N the number of nonvariable entries in `Tree`.)

`length(?List, ?Length)` basics

Succeeds if the length of the list `List` is `Length`. This predicate is deterministic if `List` is instantiated to a list of definite length, but is nondeterministic if `List` is a variable or has a variable tail. If `List` is uninstantiated, it is unified with a list of length `Length` that contains variables.

`same_length(?List1, ?List2)` basics

Succeeds if list `List1` and `List2` are both lists of the same number of elements. No relation between the types or values of their elements is implied. This predicate may be used to generate either list (containing variables as elements) given the other, or to generate two lists of the same length, in which case the arguments will be bound to lists of length $0, 1, 2, \ldots$.

`select(?Element, ?L1, ?L2)` basics

`List2` derives from `List1` by selecting (removing) an `Element` non-deterministically.

`reverse(+List, ?ReversedList)` basics

Succeeds if `ReversedList` is the reverse of list `List`. If `List` is not a proper list, `reverse/2` can succeed arbitrarily many times. It works only one way.

`perm(+List, ?Perm)` basics

Succeeds when `List` and `Perm` are permutations of each other. The main use of `perm/2` is to generate permutations of a given list. `List` must be a proper list. `Perm` may be partly instantiated.

`subseq(?Sequence, ?SubSequence, ?Complement)` basics

Succeeds when `SubSequence` and `Complement` are both subsequences of the list `Sequence` (the order of corresponding elements being preserved) and every element of `Sequence` which is not in `SubSequence` is in the `Complement` and vice versa. That is,

$$length(\texttt{Sequence}) = length(\texttt{SubSequence}) + length(\texttt{Complement})$$

for example, `subseq([1,2,3,4], [1,3], [2,4]).` The main use of `subseq/3` is to generate subsets and their complements together, but can also be used to interleave two lists in all possible ways.

`merge(+List1, +List2, ?List3)` listutil

Succeeds if `List3` is the list resulting from "merging" lists `List1` and `List2`, i.e. the elements of `List1` together with any element of `List2` not occurring in `List1`. If `List1` or `List2` contain duplicates, `List3` may also contain duplicates.

`absmerge(+List1, +List2, ?List3)` listutil

Predicate `absmerge/3` is similar to `merge/3`, except that it uses predicate `absmember/2` described below rather than `member/2`.

`absmember(+Element, +List)` listutil

Similar to `member/2`, except that it checks for identity (through the use of predicate `'=='/2`) rather than unifiability (through `'='/2`) of `Element` with elements of `List`.

`member2(?Element, ?List)` listutil

Checks whether `Element` unifies with any of the actual elements of `List`. The only difference between this predicate and predicate `member/2` is on lists having a variable tail, e.g. `[a, b, c | _ ]`: while `member/2` would insert `Element` at the end of such a list if it did not find it, Predicate `member2/2` only checks for membership but does not insert the `Element` into the list if it is not there.

`delete_ith(+Index, +List, ?Element, ?RestList)` listutil

Succeeds if the `Index`$^{\mathtt{th}}$ element of the list `List` unifies with `Element`, and `RestList` is `List` with `Element` removed. Fails if `Index` is not a positive integer or greater than the length of `List`.

`closetail(?List)` listutil

Predicate `closetail/1` closes the tail of an open-ended list. It succeeds only once.

1.2 Attributed Variables

Attributed variables are a special data type that associates variables with arbitrary attributes as well as supports extensible unification. Attributed variables have proven to be a flexible and powerful mechanism to extend a classic logic programming system with the ability of constraint solving. They have been implemented in SICStus [1] [6] and ECL^{i}PS^e [3].

Attributes of variables are compound terms whose arguments are the actual attribute values. They are defined with a declaration

:- **attribute** *AttributeSpec*, ..., *AttributeSpec*.

where each *Attributes* has the form *Functor/Arity*. Each file can have at most one such declaration.

Having declared some attribute names, these attributes can be added, updated and deleted from unbound variables using the following two predicates (`get_atts/2` and `put_atts/2`) defined in the module `atts`. For each declared attribute name, any variable can have at most one such attribute (initially it has none).

`get_atts(-Var, ?AccessSpec)` atts

Gets the attributes of `Var` according to `AccessSpec`. If `AccessSpec` is unbound, it will be bound to a list of all set attributes of `Var`. Non-variable terms in `Var` cause a type error. `AccessSpec` is either `+(Attribute)`, `-(Attribute)`, or a list of such (prefix `+` may be dropped for convenience). The prefixes in the `AccessSpec` have the following meaning:

[1] In XSB, we try to keep the implementation of attributed variables to be compatible with SICStus.

- `+(Attribute)`: The corresponding actual attribute must be present and is unified with `Attribute`.
- `-(Attribute)`: The corresponding actual attribute must be absent. The arguments of `Attribute` are ignored, only the name and arity are relevant.

`put_atts(-Var, +AccessSpec)` atts

Sets the attributes of `Var` according to `AccessSpec`. Non-variable terms in `Var` cause a type error. The effect of `put_atts/2` are undone on backtracking. The prefixes of `AccessSpec` have the following meaning:

- `+(Attribute)`: The corresponding actual attribute is set to `Attribute`. If the actual attribute was already present, it is simply replaced.
- `-(Attribute)`: The corresponding actual attribute is removed. If the actual attribute is already absent, nothing happens.

In a file that contains an attribute declaration, one has an opportunity to extend the default unification algorithm by defining the following predicate:

`verify_attributes(-Var, +Value)`

This predicate is called whenever an attributed variable `Var` (which has at least one attribute) is about to be bound to `Value` (a non-variable term or another attributed variable). When `Var` is to be bound to `Value`, a special interrupt called *attributed variable interrupt* is triggered, and then XSB's interrupt handler (written in Prolog) calls `verify_attributes/2`. If it fails, the unification is deemed to have failed. It may succeed non-deterministically, in which case the unification might backtrack to give another answer.

If `Value` is a non-variable term, `verify_attributes/2` usually inspects the attributes of `Var` and check whether they are compatible with `Value` and fail otherwise. If `Value` is another attributed variable, `verify_attributes/2` will typically merge the attributes of `Var` and `Value`, bind `Var` to `Value`, and then update their attributes. In either case, `verify_attributes/2` may determine the attributes of `Var` (or `Value`) by calling `get_atts/2`.

The predicate `verify_attributes/2` is also called *user-defined unification handler*. To help users define this handler, the following predicate is provided in module `machine`, which can be used to bind an attributed variable to an arbitrary term (might be another attributed variable) without triggering attributed variable interrupt and thus another level call of `verify_attributes/2`:

`attv_unify(-Var, +Value)` machine

This is an internal built-in predicate which is supposed to be used only in the definition of `verify_attributes/2`. It binds the attributed variable `Var` to `Value` without triggering attributed variable interrupt. `Value` is a non-variable term or another attributed variable.

Here, by giving the implementation of a simple finite domain constraint solver (see the file `fd.P` below), we show how these predicates for attributed variables can be used. In this example, only one attribute is declared: `dom/1`, and the value of this attribute is a list of terms.

```
%% File: fd.P
%%
%% A simple finite domain constrait solver implemented using attributed
%% variables.

:- import put_atts/2, get_atts/2 from atts.
:- import attv_unify/2 from machine.
:- import member/2 from basics.

:- attribute dom/1.

verify_attributes(Var, Value) :-
        get_atts(Var, dom(Da)),
        (var(Value)                           % Value is an attributed variable
         -> get_atts(Value, dom(Db)),         % has a domain
            intersection(Da, Db, [E|Es]),     % intersection not empty
            (Es = []                          % exactly one element
             -> attv_unify(Var, Value),       % bind Var to Value
                attv_unify(Var, E)            % bind Var (and Value) to E
             ;  attv_unify(Var, Value),       % bind Var to Value
                put_atts(Value, dom([E|Es])) % update Var's (and Value's)
                                              % attributes
            )
         ;  member(Value, Da),                % is Value a member of Da?
            attv_unify(Var, Value)            % bind Var to Value
        ).

intersection([], _, []).
intersection([H|T], L2, [H|L3]) :-
        member(H, L2), !,
        intersection(T, L2, L3).
intersection([_|T], L2, L3) :-
        intersection(T, L2, L3).

domain(X, Dom) :-
        var(Dom), !,
        get_atts(X, dom(Dom)).
domain(X, List) :-
        List = [El|Els],                      % at least one element
        (Els = []                             % exactly one element
         -> X = El                            % implied binding
         ;  put_atts(Fresh, dom(List)),       % create a new attributed variable
            X = Fresh                         % may call verify_attributes/2
        ).

show_domain(X) :-                             % print out the domain of X
        var(X),                               % X must be a variable
        get_atts(X, dom(D)),
        write('Domain of '), write(X),
        write(' is '), writeln(D).
```

The output of some example queries are listed below, from which we can see how attributed variables are unified using `verify_attributes/2`:

```
| ?- [fd].
[fd loaded]

yes
| ?- domain(X,[5,6,7,1]), domain(Y,[3,4,5,6]), domain(Z,[1,6,7,8]),
     show_domain(X), show_domain(Y), show_domain(Z).
Domain of _h474 is [5,6,7,1]
Domain of _h503 is [3,4,5,6]
Domain of _h532 is [1,6,7,8]

X = _h474
Y = _h503
Z = _h532

yes
| ?- domain(X,[5,6,7,1]), domain(Y,[3,4,5,6]), domain(Z,[1,6,7,8]),
     X = Y, show_domain(X), show_domain(Y), show_domain(Z).
Domain of _h640 is [5,6]
Domain of _h640 is [5,6]
Domain of _h569 is [1,6,7,8]

X = _h640
Y = _h640
Z = _h569

yes
| ?- domain(X,[5,6,7,1]), domain(Y,[3,4,5,6]), domain(Z,[1,6,7,8]),
     X = Y, Y = Z.

X = 6
Y = 6
Z = 6

yes
| ?-
```

1.3 Asserting Dynamic Code

The module `consult` in directory `lib` provides several handy library predicates that can assert the contents of a file into XSB's database. The use of these predicates may be necessary when the code needs to be `dynamic` (so that it is retractable), or when it contains atoms whose length is more than 255 that cannot be handled by the XSB compiler.

`load_dyn(+FileName)` consult

Asserts the contents of file `FileName` into the database. All existing clauses of the predicates in the file that already appear in the database, are retracted, unless there is a `multifile/1` declaration for them. Clauses in the file must be in a format that `read/1` will process. So, for example, operators are permitted. As usual, clauses of predicates are not retracted if they are compiled instead of dynamically asserted. All predicates are loaded into `usermod`. Module declarations such as `:- export` are ignored and a warning is issued.

Dynamically loaded files can be filtered through the XSB preprocessor. To do this, put the following in the source file:

```
:- compiler_options([xpp_on]).
```

Of course, the name `compiler_options` might seem like a misnomer here (since the file is not being compiled), but it is convenient to use the same directive both for compiling and loading, in case the same source file is used both ways.

`ensure_dyn_loaded(+FileName)` consult

Is similar to `load_dyn/1` except that it does nothing if the file has previously been loaded and the file has not been changed since. However the file will be reloaded if the index declaration of any predicate in that file has changed to require more indexing, or a larger hash table.

`load_dync(+FileName)` consult

Asserts the contents of file `FileName` into the database. All existing clauses of the predicates in the file that already appear in the database, are retracted unless there is a `multifile/1` directive for them. The terms in the file `FileName` must be in “canonical” format; that is, they must not use any operators (or list notation.) This is the format produced by the predicate `write_canonical/1`. (See `cvt_canonical/2` to convert a file from the usual `read/1` format to `read_canonical` format.) As usual, clauses of predicates are not retracted if they are compiled instead of dynamically asserted. All predicates are loaded into `usermod`. `:- export` declarations are ignored and a warning is issued.

Notice that this predicate can be used to load files of Datalog facts (since they will be in canonical format). This predicate is significantly faster than `load_dyn/1` and should be used when speed is important. A file that is to be dynamically loaded often but not often modified by hand should be loaded with this predicate. Use predicate `cvt_canonical/2` (see below) to convert a usual file to a format readable by this predicate.

As with `load_dyn/1`, the source file can be filtered through the C preprocessor. However, since all clauses in such a file must be in canonical form, the `compiler_options/1` directive should look as follows:

```
:-(compiler_options('.'(xpp_on,[]))).
```

`ensure_dync_loaded(+FileName)` consult

Is similar to `load_dync/1` except that it does nothing if the file has previously been loaded and the file has not been changed since. However the file will be reloaded if the index declaration of any predicate in that file has changed to require more indexing, or a larger hash table.

`cvt_canonical(+FileName1,+FileName2)` consult

Converts a file from standard term format to "canonical" format. The input file name is `FileName1`; the converted file is put in `FileName2`. This predicate can be used to convert a file in standard Prolog format to one loadable by `load_dync/1`.

1.4 Ground, Numbervars, Subsumption, Variant

`ground(+X)` basics

Succeeds if `X` is currently instantiated to a term that is completely bound (has no uninstantiated variables in it); otherwise it fails. Predicate `ground/1` has no associated error conditions.

`numbervars(+Term, +FirstN, ?LastN)` num_vars

This predicate provides a mechanism for grounding a (HiLog) term so that it may be analyzed. Each variable in the (HiLog) term `Term` is instantiated to a term of the form `'$VAR'(N)`, where `N` is an integer starting from `FirstN`. `FirstN` is used as the value of `N` for the first variable in `Term` (starting from the left). The second distinct variable in `Term` is given a value of `N` satisfying `"N is FirstN + 1"` and so on. The last variable in `Term` has the value `LastN-1`.

`numbervars(+Term)` num_vars

This predicate is defined as:

```
numbervars(Term, 0, _).
```

It is included solely for convenience.

`unnumbervars(+Term, +FirstN, ?Copy)` num_vars

This predicate is a partial inverse of predicate `numbervars/3`. It creates a copy of Term in which all subterms of the form `'$VAR'(<int>)` where `<int>` is not less than `FirstN` are uniformly replaced by variables. `'$VAR''` subterms with the same integer are replaced by the same variable. Also a version `unnumbervars/2` is provided which calls `unnumbervars/3` with the second parameter set to 0.

`subsumes(?Term1, +Term2)` subsumes

Term subsumption is a sort of one-way unification. Term `Term1` and `Term2` unify if they have a common instance, and unification in Prolog instantiates both terms to that (most general) common instance. `Term1` subsumes `Term2` if `Term2` is already an instance of `Term1`. For our purposes, `Term2` is an instance of `Term1` if there is a substitution that leaves `Term2` unchanged and makes `Term1` identical to `Term2`. Predicate `subsumes/2` does not work as described if `Term1` and `Term2` share common variables.

`subsumes_chk(+Term1, +Term2)` subsumes

The `subsumes_chk/2` predicate is true when `Term1` subsumes `Term2`; that is, when `Term2` is already an instance of `Term1`. This predicate simply checks for subsumption and does not bind any variables either in `Term1` or in `Term2`. `Term1` and `Term2` should not share any variables.

Examples:

```
| ?- subsumes_chk(a(X,f,Y,X),a(U,V,b,S)).

no
| ?- subsumes_chk(a(X,Y,X),a(b,b,b)).

X = _595884
Y = _595624
```

`variant(?Term1, ?Term2)` subsumes

This predicate is true when `Term1` and `Term2` are alphabetic variants. That is, you could imagine that `variant/2` as being defined like:

```
variant(Term1, Term2) :-
     subsumes_chk(Term1, Term2),
     subsumes_chk(Term2, Term1).
```

but the actual implementation of `variant/2` is considerably more efficient. However, in general, it does not work for terms that share variables; an assumption that holds for most (reasonable) uses of `variant/2`.

1.5 Lower-Level I/O

XSB has various low-level routines that support input and output, at both the term level and the character level. Unlike the standard Prolog stream I/O, the low-level routines use *XSB I/O ports* to refer to files. XSB I/O ports should not be confused with the file descriptors used by the OS Both are small integers, but they refer to different things. However, the OS file descriptors are objects returned by the C `open` function; XSB I/O ports indices into the internal XSB table of open files. The OS does not know about XSB I/O ports, while XSB (obviously) does know about the OS file descriptors. Typically XSB opens files for buffered I/O (whether using the stream I/O predicates or the predicates described here), so XSB I/O ports internally refer to `FILE` data structures (those returned by the C `fopen` function).

When it starts, XSB opens a number of standard I/O ports that it uses to print results, errors, debugging info, etc. The descriptors are described in the file `prolog_includes/standard.h`. This file provides the following symbolic definitions:

```
#define STDIN           0
#define STDOUT          1
#define STDERR          2
#define STDWARN         3    /* output stream for xsb warnings  */
#define STDMSG          4    /* output for regular xsb messages */
#define STDDBG          5    /* output for debugging info       */
#define STDFDBK         6    /* output for XSB feedback
                                (prompt/yes/no/Aborting/answers) */
```

```
#define AF_INET     0     /* XSB-side socket request for Internet domain */
#define AF_UNIX     1     /* XSB-side socket request for UNIX domain */
```

In addition, the file `emu/file_modes_xsb.h` provides the definitions for the file opening modes:

```
#define OREAD          0    /* open for read           */
#define OWRITE         1    /* open for write          */
#define OAPPEND        2    /* open for append         */
#define OSTRINGR       3    /* open string for reading */
#define OSTRINGW       4    /* open string for writing (not implemented) */
```

These definitions can be used in user programs, if the following is provided at the top of the source file:

```
compiler_options([xpp_on]).
#include "standard.h"
#include "file_modes_xsb.h"
```

(Note: the XSB preprocessor is not invoked on clauses typed into an interactive XSB session, so the above applies only to programs loaded from a file using `consult` and such.)

`current_input_port(-IOport)` curr_sym

See `current_output_port/1`.

`current_output_port(-IOport)` curr_sym

The above two predicates instantiate `IOport` to the XSB I/O port for the current user input and output (*i.e.*, the things that are manipulated through `see/seen` and `tell/told` predicates). Once the I/O port is obtained, it is possible to safely use the lower-level I/O predicates described below interchangeably with stream I/O.

`file_open(+FileName,+Mode,-IOport)` file_io

Opens a file with name `FileName` to be accessed in mode `Mode` and returns a file-descriptor in `IOport` that can be used to access the file. If `Mode` is atom "`r`", the the file is opened for reading; if it is "`w`", the file is opened for writing; if it is "`a`", the file is opened for appending. If `Mode` is "`sr`", then the string making up the atom `FileName` is treated as the contents of the file, and a descriptor is returned that allows "file" access to that string. This is how one can use XSB's term I/O routines to build terms from atoms. Mode "`sw`" is reserved for "open string for writing," but this has not been implemented as of yet.

The old-style mode specification, 0 (`OREAD`), 1 (`OWRITE`), 2 (`OAPPEND`), or 3 (`OSTRING`), is also supported.

`file_reopen(+FileName,+Mode,+IOport,-RetCode)` file_io

Takes an existing I/O port, closes it, then opens it and attaches it to a file. This can be used to redirect I/O from any of the standard streams to a file. For instance,

```
| ?- file_reopen('/dev/null', w, 3, Error).
```

redirects all warnings to the Unix black hole.

On success, `RetCode` is 0; on error, the return code is negative.

`tmpfile_open(-IOport)` file_io

Opens a temporary file with a unique filename. The file is deleted when it is closed or when the program terminates.

`file_clone(+SrcIOport,?DestIOport,-RetCode)` file_io

This is yet another way to redirect I/O. It is a prolog interface to the C `dup` and `dup2` system calls. If `DestIOport` is a variable, then this call creates a new XSB I/O port that is a clone of `SrcIOport`. This means that I/O sent to either descriptor goes to the same place. If `DestIOport` is not a variable, then it must be a number corresponding to a valid I/O port. In this case, XSB closes `DestIOport` and makes it into a clone on `SrcIOport`. For instance, suppose that 10 is a I/O port that is currently open for writing to file `foo.bar`. Then

```
| ?- file_clone(10,3,_).
```

causes all messages sent to XSB standard warnings port to go to file `foo.bar`. While this could be also done with `file_reopen`, there are things that only `file_clone` can do:

```
| ?- file_clone(1,10,_).
```

This means that I/O port 10 now becomes clone of standard output. So, all subsequent I/O will now go to standard output instead of `foo.bar`.

On success, `RetCode` is 0; on error, the return code is negative.

`file_close(+IOport)` file_io

Closes the file (or string) for descriptor `IOport`.

`fmt_read(+Fmt,-Term,-Ret)` file_io

`fmt_read(+IOport,+Fmt,-Term,-Ret)` file_io

These predicates provides a routine for reading data from the current input file (which must have been already opened by using `see/1`) according to a C format, as used in the C function `scanf`. To use it, it must be imported from the module `file_io`. `Fmt` must be a string of characters (enclosed in "") representing the format that will be passed to the C call to `scanf`. See the C documentation for `scanf` for the meaning of this string. The usual alphabetical C escape characters (*e.g.*, $\backslash n$) are recognized, but not the octal or the hexadecimal ones. Another difference with C is that, unlike most C compilers, XSB insists that a single `%` in the format string signifies format conversion specification. (Some C compilers might output `%` if it is not followed by a valid type conversion spec.) So, to output `%` you must type `%%`. Format can also be an atom enclosed in single quotes. However, in that case, escape sequences are not recognized and are printed as is.

`Term` is a term (*e.g.*, `args(X,Y,Z)`) whose arguments will be unified with the field values read in. (The functor symbol of `Term` is ignored.) Special syntactic sugar is provided for the case when the format string contains only one format specifier: If `Term` is a variable, `X`, then the predicate behaves as if `Term` were `arg(X)`.

If the number of arguments exceeds the number of format specifiers, a warning is produced and the extra arguments remain uninstantiated. If the number of format specifiers exceeds the number of arguments, then the remainder of the format string (after the last matching specifier) is ignored.

Note that floats do not unify with anything. `Ret` must be a variable and it will be assigned a return value by the predicate: a negative integer if end-of-file is encountered; otherwise the number of fields read (as returned by `scanf`.)

`fmt_read` cannot read strings (that correspond to the `%s` format specifier) that are longer than 16K. Attempting to read longer strings will cause buffer overflow. It is therefore recommended that one should use size modifiers in format strings (*e.g.*, `%2000s`), if such long strings might occur in the input.

`fmt_write(+Fmt,+Term)` file_io

`fmt_write(+IOport,+Fmt,+Term)` file_io

This predicate provides a routine for writing data to the current output file (which must have been already opened by using `tell/1`) according to a C format, as used in the C function `printf`. To use it, it must be imported from the module `file_io`. `Fmt` must be a string of characters (enclosed in "") representing the format that will be passed to the C call to `scanf`. See the C documentation for `scanf` for the meaning of this string. The usual alphabetical C escape characters (*e.g.*, $\backslash n$) are recognized, but not the octal or the hexadecimal ones.

In addition to the usual C conversion specifiers, `%S` is also allowed. The corresponding argument can be any Prolog term. This provides an easy way to print the values of Prolog variables, etc.

Another difference with C is that, unlike most C compilers, XSB insists that a single `%` in the format string signifies format conversion specification. (Some C compilers might output `%` if it is not followed by a valid type conversion spec.) So, to output `%` you must type `%%`.

Format can also be an atom, but then escape sequences are not recognized.

`Term` is a term (*e.g.*, `args(X,Y,Z)`) whose arguments will be output. The functor symbol of `Term` is ignored.

Special syntactic sugar is provided for the following cases: If `Term` is a variable, `X`, then it is ignored and only the format string is printed. If `Term` is a string, integer or a float, then it is assumed that this is the only argument to be printed, *i.e.*, it is equivalent to specifying `arg(Term)`.

If the number of format specifiers is greater than the number of arguments to be printed, an error is issued. If the number of arguments is greater, then a warning is issued.

`fmt_write_string(-String,+Fmt,+Term)` file_io

This predicate works like the C function `sprintf`. It takes the format string and substitutes the values from the arguments of `Term` (*e.g.*, `args(X,Y,Z)`) for the formatting instructions %s, %d, etc. Additional syntactic sugar, as in `fmt_write`, is recognized. The result is available in `String`. `Fmt` is a string or an atom that represents the format, as in `fmt_write`.

If the number of format specifiers is greater than the number of arguments to be printed, an error is issued. If the number of arguments is greater, then a warning is issued.

fmt_write_string requires that the printed size of each argument (*e.g.*, X,Y,and Z above) must be less than 16K. Longer arguments are cut to that size, so some loss of information is possible. However, there is no limit on the total size of the output (apart from the maximum atom size imposed by XSB).

file_flush(+IOport, -Return) file_io

Any buffered data gets delivered. If the call is successful, Return is zero; otherwise EOF is returned.

file_seek(+IOport, +Offset, +Place, -Return) file_io

Sets the file position indicator for the next input or output operation. The position is Offset bytes from Place. The value of Place can be 0, 1, or 2, which correspond to the beginning of the file, the current position in the file, or the end of the file, respectively. If the call is successful, Return is set to zero.

file_pos(+IOport, -Position) file_io

Unifies Position with the position inside the file indicated by IOport.

file_truncate(+IOport, +Length, -Return) file_io

The regular file referenced by the I/O port IOport is chopped to have the size of Length bytes. Upon successful completion Return is set to zero.

file_write(+IOport,+Term) xsb_writ

Writes the term Term to the file (or string) with descriptor IOport.

file_read(+IOport,-Term) xsb_read

Reads a term from the file (or string) with descriptor IOport into Term. Note that the term must be terminated with a period (.) (whether it appears in a file or in a string.)

file_read(+IOport,-Term,-Vars) xsb_read

Reads a term from the file (or string) with descriptor IOport into Term, and returns in Vars an open-tailed list of pairs of names of variables and the variables themselves that appear in Term. For example, reading a term f(a,X,Y,X) would result in term being bound to f(a,_25,_26,_25) (for some internal variables) and Vars being bound to [vv('X',_25) vv('Y',_26) | _83]. Note that the pairing functor symbol is vv/2 and it must be imported from xsb_read along with this read predicate. Also note that Vars is not a proper list, but has a free variable instead of [] at its end.

file_read_canonical(+IOport,-Term,-Psc) machine

Reads a term that is in canonical format from the the I/O port indicated by IOport (as returned by file_open/3 or by stat_flag(10,IOport)), and returns it in Term. It also returns (in Psc) the psc address of the main functor symbol of the term, if it is the same as that of the previously read term, and the current term is a ground (non 0-ary) fact. (This is used for efficiency in the implementation of load_dync/1). Otherwise Psc is set to 0. To initialize its previous psc value to zero, this predicate can be called with IOport of -1000.

file_read_line(+IOport,-String,-IsFullLine) file_io

This is a low-level predicate that allows XSB to read input files efficiently, line by line. It returns the string read from IOport using the variable String. The output variable IsFullLine is 1, if the line read contains the newline character at the end. Otherwise, it is 0. This latter case arises in two situations: when the last line in the stream does not have a newline character or when the line is very long, longer than the buffer allocated for that purpose. (In such a case file_read_line/3 will read only as much as the buffer allows.)

This predicate fails on reaching the end of file.

Important: This predicate does not intern the string it reads from the input, so you cannot unify or compare the value read with anything. Moreover, if you use this predicate twice, then the second call replaces the value read by the first call. In other words, this predicate is very low-level and should be used with great care. One simple use of this facility is to copy one file into another. See `file_read_line_atom` and `file_read_line_list` for alternatives that are easier to use.

`file_read_line_atom(+IOport,-String,-IsFullLine)` file_io

This predicate is like `file_read_line`, but the line read from the input is interned. Therefore, `String` is instantiated to a normal atom. This predicate fails on reaching the end of file.

Important: At present, XSB does not have atom table garbage collector. Therefore, each line read from the file using this predicate gets stored in the atom table. Thus, large files can cause XSB to run out of memory. This problem will go away when atom table garbage collection is implemented.

`file_read_line_atom(-String,-IsFullLine)` file_io

Like `file_read_line_atom/3`, but `IOport` is not required. The file being read is the one previously opened with `see/1`.

`file_read_line_list(+IOport,-CharList,-IsFullLine)` file_io

This predicate is like `file_read_line_atom`, but the line read from the input is converted into a list of characters. This predicate is *much* more efficient than `fget_line/3` (see below), and is recommended when speed is important. This predicate fails on reaching the end of file.

`file_read_line_list(-String,-IsFullLine)` file_io

Like `file_read_line_list/3`, but `IOport` is not required. The file being read is the one previously opened with `see/1`.

`fget_line(+Str,-Inlist,-Next)` scrptutl

`fget_line/3` reads one line from the input stream `Str` and unifies `Inlist` to the list of ASCII integers representing the characters in the line, and `Next` to the line terminator, either a newline (`10`) or EOF (`-1`).

This predicate is obsolete and `file_read_line_list` should be used instead.

`file_write_line(+IOport, +String, +Offset)` file_io
Write `String` beginning with character `Offset` to the output file represented by the I/O port `IOport`. `String` can be an atom or a list of ASCII characters. This does *not* put the newline character at the end of the string (unless `String` already had this character). Note that escape sequences, like `\n`, are recognized if `String` is a character list, but are output as is if `String` is an atom.

`file_write_line(+String, +Offset)` file_io
Like `file_write_line/3`, but output goes to the currently open output stream.

`file_getbuf(+IOport, +BytesRequested, -String, -BytesRead)` file_io
Read `BytesRequested` bytes from file represented by I/O port `IOport` (which must already be open for reading) into variable `String`. This is analogous to `fread` in C. This predicate always succeeds. It does not distinguish between a file error and end of file. You can determine if either of these conditions has happened by verifying that `BytesRead` < `BytesRequested`.

Note: The string read is **not** interned. Please see explanations to `file_read_line` on this matter. Because of the difficulties in using this predicate, the predicates `file_getbuf_atom` and `file_getbuf_list` are often better alternatives.

`file_getbuf_atom(+IOport, +BytesRequested, -String, -BytesRead)` file_io
Like `file_getbuf`, but `String` is instantiated to an interned atom.

Note: because XSB does not have an atom table garbage collector yet, this predicate should not be used to read large files.

`file_getbuf_atom(+BytesRequested, -String, -BytesRead)` file_io
Like `file_getbuf_atom/4`, but reads from the currently open input stream (using `see/1`). This predicate always succeeds. It does not distinguish between a file error and end of file. You can determine if either of these conditions has happened by verifying that `BytesRead` < `BytesRequested`.

`file_getbuf_list(+IOport, +BytesRequested, -CharList, -BytesRead)` file_io
Like `file_getbuf_atom/4`, but `CharList` is instantiated to a list of characters that represent the string read from the input.

`file_getbuf_list(+BytesRequested, -String, -BytesRead)` file_io
Like `file_getbuf_list/3`, but reads from the currently open input stream (*i.e.*, with `see/1`).

`file_putbuf(+IOport, +BytesRequested, +String, +Offset, -BytesWritten)` file_io
Write `BytesRequested` bytes into file represented by I/O port `IOport` (which must already be open for writing) from variable `String` at position `Offset`. This is analogous to C `fwrite`. The value of `String` can be an atom or a list of ASCII characters.

`file_putbuf(+BytesRequested, +String, +Offset, -BytesWritten)` file_io
Like `file_putbuf/3`, but output goes to the currently open output stream.

1.6 String Manipulation

XSB has a number of powerful builtins that simplify the job of string manipulation. These builting are especially powerful when they are combined with pattern-matching facilities provided by the `regmatch` package described in Chapter 4.

`str_sub(+Sub, +Str, ?Pos)` string

Succeeds if `Sub` is a substring of `Str`. In that case, `Pos` unifies with the position where the match occurred.

`str_cat(+Str1, +Str2, ?Result)` string

Concatenates `Str1` with `Str2`. Unifies the result with `Result`.

In addition to this, the predicate `fmt_write_string/3` described in Section 1.5 can be used to concatenate strings and do much more. However, for simple string concatenation, `str_cat/3` is more efficient.

`str_length(+Str, ?Result)` string

Unifies the `Result` with the length of `Str`.

`substring(+String, +BeginOffset, +EndOffset, -Result)` string

`String` can be an atom or a list of characters, and the offsets must be integers. If `EndOffset` is negative, endof(`String`)+`EndOffset` is assumed. If `EndOffset` is an unbound variable, then end of string is assumed. If `BeginOffset` is less than 0, then 0 is assumed; if it is greater than the length of the string, then string end is assumed. If `EndOffset` is non-negative, but is less than `BeginOffset`, then empty string is returned.

The result returned in the fourth argument is a string, if `String` is an atom, or a list of characters, if so is `String`.

The `substring/4` predicate always succeeds (unless there is an error, such as wrong argument type).

Here are some examples:

```
| ?- substring('abcdefg', 3, 5, L).

L = de

| ?- substring("abcdefg", 4, -1, L).

L = [101,102]
```

(*i.e.*, L = ef represented using ASCII codes).

`string_substitute(+String, +SubstrList, +SubstitutionList, -OutStr)` string

`InputStr` can an atom or a list of characters. `SubstrList` must be a list of terms of the form `s(BegOffset, EndOffset)`, where the name of the functor is immaterial. The meaning of the offsets is the same as for `re_substring/4`. Each such term specifies a substring (between `BegOffset` and `EndOffset`; negative `EndOffset` stands for the end of string) to be replaced. `SubstitutionList` must be a list of atoms or character lists.

This predicate replaces the substrings specified in `SubstrList` with the corresponding strings from `SubstitutionList`. The result is returned in `OutStr`. `OutStr` is a list of characters, if so is `InputStr`; otherwise, it is an atom.

If `SubstitutionList` is shorter than `SubstrList` then the last string in `SubstitutionList` is used for substituting the extra substrings specified in `SubstitutionList`. As a special case, this makes it possible to replace all specified substrings with a single string.

As in the case of `re_substring/4`, if `OutStr` is an atom, it is not interned. The user should either intern this string or convert it into a list, as explained previously.

The `string_substitute/4` predicate always succeeds.

Here are some examples:

```
| ?- string_substitute('qaddf', [s(2,4)], ['123'] ,L).

L = qa123f

| ?- string_substitute('qaddf', [s(2,-1)], ['123'] ,L).

L = qa123

| ?- string_substitute("abcdefg", [s(4,-1)], ["123"],L).

L = [97,98,99,100,49,50,51]

| ?- string_substitute('1234567890123', [f(1,5),f(5,7),f(9,-2)], ["pppp", lll],X).

X = 1pppplll89lll

| ?- string_substitute('1234567890123', [f(1,5),f(6,7),f(9,-2)], ['---'],X).

X = 1---6---89---
```

1.7 Script Writing Utilities

Prolog, (or XSB) can be useful for writing scripts in a UNIX system. Prolog's simple syntax and declarative semantics make it especially suitable for scripts that involve text processing. Wherever noted, some of these functions are currently working under Unix only.

`date(?Date)` scrptutl

Unifies `Date` to the current date, returned as a Prolog term, suitable for term comparison. Currently this only works under Unix, is slow, and should be rewritten in C using `time()` and `localtime()`.
Example:

```
 > date
 Thu Feb 20 08:46:08 EST 1997
 > xsb -i
XSB Version 1.7
[sequential, single word, optimal mode]
| ?- [scrptutl].
[scrptutl loaded]

yes
| ?- date(D).
D = date(1997,1,20,8,47,41)

yes
```

`file_time(+FileName, -time(Time1,Time2))` file_io

Returns file's modification time. Because XSB steals 5 bits from each word, time must be returned as two words: Time1, representing the most significant digits, and Time2, representing the less significant digits.

`file_size(+FileName, -Size)` file_io

Returns file size.

`directory(+Path,?Directory)` directry

Unifies `Directory` with a list of files in the directory specified by path. Information about the files is similar to that obtained by `ls -l`, but transformed for ease of processing. This currently works for Unix only, is slow, and should be reimplemented in C using `opendir()` and `readdir()`.

`expand_filename(+FileName,-ExpandedName)` machine

Expands the file name passed as the first argument and binds the variable in the second argument to the expanded name. This includes expanding Unix tildas, prepending the current directory, etc. In addition, the expanded file name is "rectified" so that multiple repeated slashes are replaced with a single slash, the intervening "./" are removed, and "../" are applied so that the preceding item in the path name is deleted. For instance, if the current directory is `/home`, then `abc//cde/..///ff/./b` will be converted into `/home/abc/ff/b`.

Under NT and Windows, this predicates does rectification (using backslashes when appropriate), but it does not expand the tildas.

`tilde_expand_filename(+FileName,-ExpandedName)` machine

Like `expand_filename/2`, but only expands tildas and does rectification. This does not prepend the current working directory to relative file names.

`is_absolute_filename(+FileName)` machine

Succeeds, if file name is absolute; fails otherwise. This predicate works also under NT and Windows, *i.e.*, it recognizes drive letters, etc.

`parse_filename(+FileName,-Dir,-Base,-Extension)` machine

This predicate parses file names by separating the directory part, the base name part, and file extension. If file extension is found, it is removed from the base name. Also, directory names are rectified and if a directory name starts with a tilde (in Unix), then it is expanded. Directory names always end with a slash or a backslash, as appropriate for the OS at hand.

For instance, `~john///doe/dir1//../foo.bar` will be parsed into: `/home/john/doe/`, `foo`, and `bar` (where we assume that `/home/john` is what `~john` expands into).

`file_to_list(IOport, List)` scrptutl

s Read lines from an *open* I/O port. Return a list of terms, one per each line read. Each such term is a list of tokens on the corresponding line. Tokens are lists of characters separated by a space symbol (space, newline, return, tabs, formfeed). For instance, if `IOport` 10 is bound to a file

```
ads sdfdsfd ee
112 444
4555
```

then

```
| ?- file_to_list(10, L).
L = [[ads,sdfdsfd,ee],[112,444],[4555]]
yes
```

Note: `file_to_list/2` does not close the I/O port, so it is an application program responsibility.

1.8 Communication with Subprocesses

In the previous section, we have seen several predicates that allow XSB to create other processes. However, these predicates offer only a very limited way to communicate with these processes. The predicate `spawn_process/5` and friends come to the rescue. It allows to spawn any process (including multiple copies of XSB) and redirect its standard input and output to XSB I/O ports. XSB can then write to the process and read from it. The section of socket I/O describes yet another mode of interprocess communication.

In addition, the predicate `pipe_open/2` described in this section lets one create any number of pipes (that do not need to be connected to the standard I/O port) and talk to child processes through these pipes.

All predicates in this section, except `pipe_open/2` and `fd2ioport/2`, must be imported from module `shell`. The predicates `pipe_open/2` and `fd2ioport/2` must be imported from `file_io`.

`spawn_process(+CmdSpec,-StreamToProc,-StreamFromProc,-ProcStderrStream,-ProcId)`
Spawn a new process specified by `CmdSpec`. `CmdSpec` must be either a single atom or a *list* of atoms. If it is an atom, then it must represent a shell command. If it is a list, the first member of the list must be the name of the program to run and the other elements must be arguments to the program. Program name must be specified in such a way as to make sure the OS can find it using the contents of the environment variable `PATH`. Also note that pipes, I/O redirection and such are not allowed in command specification. That is, `CmdSpec` must represent a single command. (But read about process plumbing below and about the related predicate `shell/5`.)

The next three parameters of `spawn_process` are XSB I/O ports to the process (leading to the subprocess standard input), from the process (from its standard output), and a port capturing the subprocess standard error output. The last parameter is the system process id.

Here is a simple example of how it works.

```
| ?- import file_flush/2, file_read_line_atom/3 from file_io.
| ?- import file_nl/1 , file_write/2 from xsb_writ.

| ?- spawn_process([cat, '-'], To, From, Stderr, Pid),
     file_write(To,'Hello cat!'), file_nl(To), file_flush(To,_),
     file_read_line_atom(From,Y,_).

To = 3
From = 4
Stderr = 5
Pid = 14328
Y = Hello cat!

yes
```

Here we created a new process, which runs the "`cat`" program with argument "–". This forces `cat` to read from standard input and write to standard output. The next line writes a newline-terminated string to the XSB port `To` the, which is bound to the standard input of the `cat` process. The process then copies the input to the standard output. Since standard output of the process is redirected to the XSB port `From`, the last line in our program is able to read it and return in the variable `Y`.

Note that in the second line we used `file_flush/2`. Flushing the output is extremely important here, because XSB I/O ports are buffered. Thus, `cat` might not see its input until the buffer is filled up, so the above clause might hang. `file_flush/2` makes sure that the input is immediately available to the subprocess.

In addition to the above general schema, the user can tell spawn_process/5 to not open one of the communication ports or to use one of the existing communication ports. This is useful when you do not expect to write or read to/from the subprocess or when one process wants to write to another (see the process plumbing example below).

To tell that a certain port is not needed, it suffices to bind that port to an atom. For instance,

```
| ?- spawn_process([cat, '-'], To, none, none, _),
     file_nl(To), file_write(To,'Hello cat!'), file_nl(To), file_flush(To,_).


To = 3,
Hello cat!
```

reads from XSB and copies the result to standard output. Likewise,

```
| ?- spawn_process('cat test', none, From, none, _),
     file_read_line_atom(From, S, _).

From = 4
S = The first line of file `test'
```

In each case, only one of the ports is open. (Note that the shell command is specified as an atom rather than a list.) Finally, if both ports are suppressed, then spawn_process reduces to the usual shell/1 call (in fact, this is how shell/1 is implemented):

```
| ?- spawn_process([pwd], none, none).

/usr/local/foo/bar
```

On the other hand, if any one of the three port variables in spawn_process is bound to an already existing file port, then the subprocess will use that port (see the process plumbing example below).

One of the uses of XSB subprocesses is to create XSB servers that spawn subprocesses and control them. A spawned subprocess can be another XSB process. The following example shows one XSB process spawning another, sending it a goal to evaluate and obtaining the result:

```
| ?- spawn_process([xsb], To, From,Err,_),
     file_write(To,'assert(p(1)).'), file_nl(To), file_flush(To,_),
     file_write(To,'p(X), writeln(X).'), file_nl(To), file_flush(To,_),
     file_read_line_atom(From,XX,_).

XX = 1

yes
| ?-
```

Here the parent XSB process sends "`assert(p(1)).`" and then "`p(X), writeln(X).`" to the spawned XSB subprocess. The latter evaluates the goal and prints (via "`writeln(X)`") to its standard output. The main process reads it through the `From` port and binds the variable `XX` to that output.

Finally, we should note that the port variables in the `spawn_process` predicate can be used to do process plumbing, *i.e.*, redirect output of one subprocess into the input of another. Here is an example:

```
| ?- file_open(test,w,IOport),
     spawn_process([cat, 'data'], none, FromCat1, none, _),
     spawn_process([sort], FromCat1, IOport, none, _).
```

Here, we first open file `test`. Then `cat data` is spawned. This process has the input and standard error ports blocked (as indicated by the atom `none`), and its output goes into port FromCat1. Then we spawn another process, `sort`, which picks the output from the first process (since it uses the port `FromCat1` as its input) and sends its own output (the sorted version of `data`) to its output port `IOport`. However, `IOport` has already been open for output into the file `test`. Thus, the overall result of the above clause is tantamount to the following shell command:

```
cat data | sort > test
```

Important notes about spawned processes :

1. Asynchronous processes spawned by XSB do not disappear (at least on Unix) when they terminate, *unless* the XSB program executes a *wait* on them (see `process_control` below). Instead, such processes become defunct *zombies* (in Unix terminology); they do not do anything, but consume resources (such as file descriptors). So, when a subprocess is known to terminate, it must be waited on.

2. The XSB parent process must know how to terminate the asynchronous subprocesses it spawns. The drastic way is to kill it (see `process_control` below). Sometimes a subprocess might terminate by itself (*e.g.*, having finished reading a file). In other cases, the parent and the child programs must agree on a protocol by which the parent can tell the child to exit. The programs in the XSB subdirectory `examples/subprocess` illustrate this idea. If the child subprocess is another XSB process, then it can be terminated by sending the atom `end_of_file` or `halt` to the standard input of the child. (For this to work, the child XSB must waiting at the prompt).

3. It is very important to not forget to close the I/O ports that the parent uses to communicate with the child. These are the ports that are provided in arguments 2,3,4 of `spawn_process`. The reason is that the child might terminate, but these ports will remain open, since they belong to the parent process. As a result, the parent will own defunct I/O ports and might eventually run out of file descriptors.

`process_status(+Pid,-Status)`

This predicate always succeeds. Given a process id, it binds the second argument (which must be an unbound variable) to one of the following atoms: `running`, `stopped`, `exited` (normally), `aborted` (abnormally), `invalid`, and `unknown`. The `invalid` status is given to processes that never existed or that are not children of the parent XSB process. The `unknown` status is assigned when none of the other statuses can be assigned.

Note: process status (other than `running`) is system dependent. Windows does not seem to support `stopped` and `aborted`. Also, processes killed using the `process_control` predicate (described next) are often marked as `invalid` rather than `exited`, because Windows seems to lose all information about such processes.

`process_control(+Pid,+Operation)`

Perform a process control `operation` on the process with the given `Pid`. Currently, the only supported operations are `kill` and `wait` (both must be atoms). The former causes the process to exit unconditionally, and the latter waits for process completion.

This predicate succeeds, if the operation was performed successfully. Otherwise, it fails. The `wait` operation fails if the process specified in `Pid` does not exist or is not a child of the parent XSB process.

The `kill` operation might fail, if the process to be killed does not exist or if the parent XSB process does not have the permission to terminate that process. Unix and Windows have different ideas as to what these permissions are. See *kill(2)* for Unix and *TerminateProcess* for Windows.

Note: under Windows, the programmer's manual warns of dire consequences if one kills a process that has DLLs attached to it.

`get_process_table(-ProcessList)`

Binds `ProcessList` to the list of terms, each describing one of the active XSB subprocesses (created via `spawn_process/5`). Each term has the form:

```
process(Pid,ToStream,FromStream,StderrStream,CommandLine).
```

The first argument in the term is the process id of the corresponding process, the next three arguments describe the three standard ports of the process, and the lat is an atom that shows the command line used to invoke the process. This predicate always succeeds.

`shell(+CmdSpec,-StreamToProc, -StreamFromProc, -ProcStderr, -ErrorCode)`

The arguments of this predicate are similar to those of `spawn_process`, except for the following: (1) The first argument is an atom or a list of atoms, like in `spawn_process`. However, if it is a list of atoms, then the resulting shell command is obtained by string concatenation. This is different from `spawn_process` where each member of the list must represent an argument to the program being invoked (and which must be the first member of that list). (2) The last argument is the error code returned by the shell command and not a process id. The code -1 and 127 mean that the shell command failed.

The `shell/5` predicate is similar to `spawn_process` in that it spawns another process and can capture that process' input and output ports.

The important difference, however, is that XSB will ways until the process spawned by `shell/5` terminates. In contrast, the process spawned by `spawn_process` will run concurrently with XSB. In this latter case, XSB must explicitly synchronize with the spawned subprocess using the predicate `process_control/2` (using the `wait` operation), as described earlier.

The fact that XSB must wait until `shell/5` finishes has a very important implication: the amount of data the can be sent to and from the shell command is limited (1K is probably safe). This is because the shell command communicates with XSB via pipes, which have limited capacity. So, if the pipe is filled, XSB will hang waiting for `shell/5` to finish and `shel/5` will wait for XSB to consume data from the pipe. Thus, use `spawn_process/5` for any kind of significant data exchange between external processes and XSB.

Another difference between these two forms of spawning subprocesses is that `CmdSpec` in `shell/5` can represent *any* shell statement, including those that have pipes and I/O redirection. In contrast, `spawn_process` only allows command of the form "program args". For instance,

```
| ?- file_open(test,w,IOport),
     shell('cat | sort > data', IOport, none, none, ErrCode)
```

As seen from this example, the same rules for blocking I/O streams apply to `shell/5`. Finally, we should note that the already familiar standard predicates `shell/1` and `shell/2` are implemented using shell/5.

Notes:

1. With `shell/5`, you do not have to worry about terminating child processes: XSB waits until the child exits automatically. However, since communication pipes have limited capacity, this method can be used only for exchanging small amounts of information between parent and child.
2. The earlier remark about the need to close I/O streams to the child *does* apply.

`pipe_open(-ReadPipe, -WritePipe)`

Open a new pipe and return the read end and the write end of that pipe. If the operation fails, both `ReadPipe` and `WritePipe` are bound to negative numbers.

The pipes returned by the `pipe_open/2` predicate are small integers that represent file descriptors used by the underlying OS. They are **not XSB I/O ports**, and they cannot be used for I/O directly. To use them, one must call the `fd2ioport/2` predicate, as described next.[2]

`fd2ioport(+Pipe, -IOport)`

Take a pipe and convert it to an XSB I/O port that can be used for I/O. This predicate is

[2] XSB does not convert pipes into I/O ports automatically. Because of the way XSB I/O ports are represented, they are not inherited by the child process and they do not make sense to the child process (especially if the child is not another xsb). Therefore, we must pass the child processes an OS file descriptor instead. The child then converts these descriptor into XSB I/O ports.

needed because pipes must be associated with XSB I/O ports before any I/O can be on them by an XSB program.

The best way to illustrate how one can create a new pipe to a child (even if the child has been created earlier) is to show an example. Consider two programs, `parent.P` and `child.P`. The parent copy of XSB consults `parent.P`, which does the following: First, it creates a pipe and spawns a copy of XSB. Then it tells the child copy of XSB to assert the fact `pipe(RP)`, where `RP` is a number representing the read part of the pipe. Next, the parent XSB tells the child XSB to consult the program `child.P`. Finally, it sends the message `Hello!`.

The `child.P` program gets the pipe from predicate `pipe/1` (note that the parent tells the child XSB to first assert `pipe(RP)` and only then to consult the `child.P` file). After that, the child reads a message from the pipe and prints it to its standard output. Both programs are shown below:

```
%% parent.p
:- import pipe_open/2, fd2ioport/2, fmt_write/3, file_flush/2 from file_io.
%% Create the pipe and pass it to the child process
?- pipe_open(RP,WP),
   %% WF is now the XSB I/O port bound to the write part of the pipe
   fd2ioport(WP,WF),
   %% We aren't going to read from child, so let's close the pipe coming
   %% from it -- we don't want to run out of file descriptors!!!
   fd2ioport(RP,RF), file_close(RF),
   %% ProcInput becomes the XSB stream leading directly to the child's stdin
   spawn_process(xsb, ProcInput, block, block, Process),
   %% Tell the child where the reading part of the pipe is
   fmt_write(ProcInput, "assert(pipe(%d)).\n", arg(RP)),
   fmt_write(ProcInput, "[child].\n", _),
   file_flush(ProcInput, _),
   %% Pass a message through the pipe
   fmt_write(WF, "Hello!\n", _),
   file_flush(WF, _),
   fmt_write(ProcInput, "end_of_file.\n",_), % send end_of_file atom to child
   file_flush(ProcInput, _),
   %% wait for child (so as to not leave zombies around;
   %% zombies quit when the parent finishes, but they consume resources)
   process_control(Process, wait),
   %% Close the ports used to commuicate with the process
   %% Otherwise, the parent might run out of file descriptors
   %% (if many processes were spawned)
   file_close(ProcInput), file_close(WF).


%% child.P
:- import fd2ioport/2 from file_io.
```

```
:- import file_read_line_atom/3 from file_io.
:- dynamic pipe/1.
?- pipe(P), fd2ioport(P,F),
   %% Acknowledge receipt of the pipe
   fmt_write("\nPipe %d received\n", arg(P)),
   %% Get a message from the parent and print it to stdout
   file_read_line_atom(F, Line,_), write('Message was: '), writeln(Line).
```

This produces the following output:

```
| ?- [parent].                    <- parent XSB consults parent.P
[parent loaded]
yes
| ?- [xsb_configuration loaded]   <- parent.P spawns a child copy of XSB
[sysinitrc loaded]                   Here we see the startup messages of
[packaging loaded]                   the child copy
XSB Version 2.0 (Gouden Carolus) of June 27, 1999
[i686-pc-linux-gnu; mode: optimal; engine: slg-wam; scheduling: batched]
| ?-
yes
| ?- [Compiling ./child]          <- The child copy of received the pipe from
[child compiled, cpu time used: 0.1300 seconds]     the parent and then the
[child loaded]                                      request to consult child.P
Pipe 15 received                  <- child.P acknowledges receipt of the pipe
Message was: Hello!               <- child.P gets the message and prints it
yes
```

Observe that the parent process is very careful about making sure that the child terminates and also about closing the I/O ports after they are no longer needed.

Finally, we should note that this mechanism can be used to communicate through pipes with non-XSB processes as well. Indeed, an XSB process can create a pipe using `pipe_open` (*before* spawning a child process), pass one end of the pipe to a child process (which can be a C program), and use `fd2ioport` to convert the other end of the pipe to an XSB file. The C program, of course, does not need `fd2ioport`, since it can use the pipe file handle directly. Likewise, a C program can spawn off an XSB process and pass it one end of a pipe. The XSB child-process can then convert this pipe fd to a file using `fd2ioport` and then talk to the paren C program.

`sys_exit(-ExitCode)`

This predicate causes XSB subprocess to exit unconditionally with the exit code `ExitCode`. Normally `0` is considered to indicate normal termination, while other exit codes are used to report various degrees of abnormality.

1.9 Socket I/O

The XSB socket library defines a number of predicates for communication over BSD-style sockets. Most are modeled after and are interfaces to the socket functions with the same name. For detailed information on sockets, the reader is referred to the Unix man pages (another good source is *Unix Network Programming*, by W. Richard Stevens). Several examples of the use of the XSB sockets interface can be found in the `XSB/examples/` directory in the XSB distribution.

XSB supports two modes of communication via sockets: *stream-oriented* and *message-oriented.* In turn, stream-oriented communication can be *buffered* or *character-at-a-time.*

To use *buffered* stream-oriented communication, system socket handles must be converted to XSB I/O ports using `fd2ioport/2` and then the regular low-level file I/O primitives (described in Section 1.5) are used. In stream-oriented communication, messages have no boundaries, and communication appears to the processes as reading and writing to a file. At present, buffered stream-oriented communication works under Unix only.

Character-at-a-time stream communication is accomplished using the primitives `socket_put/3` and `socket_get0/3`. These correspond to the usual Prolog `put/1` and `get0/1` I/O primitives.

In message-oriented communication, processes exchange messages that have well-defined boundaries. The communicating processes use `socket_send/3` and `socket_recv/3` to talk to each other. XSB messages are represented as strings where the first four bytes (`sizeof(int)`) is an integer (represented in the binary network format — see the functions `htonl` and `ntohl` in socket documentation) and the rest is the body of the message. The integer in the header represents the length of the message body.

We now describe the XSB socket interface. All predicates below must be imported from the module `socket`. Note that almost all predicates have the last argument that unifies with the error code returned from the corresponding socket operation. This argument is explained separately.

General socket calls. These are used to open/close sockets, to establish connections, and set special socket options.

`socket(-Sockfd, ?ErrorCode)`
: A socket `Sockfd` in the AF_INET domain is created. (The AF_UNIX domain is not yet implemented). `Sockfd` is bound to a small integer, called socket descriptor or socket handle.

`socket_set_option(+Sockfd,+OptionName,+Value)`
: Set socket option. At present, only the `linger` option is supported. "Lingering" is a situation when a socket continues to live after it was shut down by the owner. This is used in order to let the client program that uses the socket to finish reading or writing from/to the socket. `Value` represents the number of seconds to linger. The value -1 means do not linger at all.

`socket_close(+Sockfd, ?ErrorCode)`
: `Sockfd` is closed. Sockets used in `socket_connect/2` should not be closed by `socket_close/1` as they will be closed when the corresponding stream is closed.

`socket_bind(+Sockfd,+Port, ?ErrorCode)`
The socket `Sockfd` is bound to the specified local port number.

`socket_connect(+Sockfd,+Port,+Hostname,?ErrorCode)`
The socket Sockfd is connected to the address (`Hostname` and `Port`).

`socket_listen(+Socket, +Length, ?ErrorCode)`
The socket `Sockfd` is defined to have a maximum backlog queue of `Length` pending connections.

`socket_accept(+Sockfd,-SockOut, ?ErrorCode)`
Block the caller until a connection attempt arrives. If the incoming queue is not empty, the first connection request is accepted, the call succeeds and returns a new socket, `SockOut`, which can be used for this new connection.

Buffered, message-based communication. These calls are similar to the `recv` and `send` calls in C, except that XSB wraps a higher-level message protocol around these low-level functions. More precisely, `socket_send/3` prepends a 4-byte field to each message, which indicates the length of the message body. When `socket_recv/3` reads a message, it first reads the 4-byte field to determine the length of the message and then reads the remainder of the message.

All this is transparent to the XSB user, but you should know these details if you want to use these details to communicate with external processes written in C and such. All this means that these external programs must implement the same protocol. The subtle point here is that different machines represent integers differently, so an integer must first be converted into the machine-independent network format using the functions `htonl` and `ntohl` provided by the socket library. For instance, to send a message to XSB, one must do something like this:

```
char *message, *msg_body;
unsigned int msg_body_len, network_encoded_len;

  msg_body_len = strlen(msg_body);
  network_encoded_len = (unsigned int) htonl((unsigned long int) msg_body_len);
  memcpy((void *) message, (void *) &network_encoded_len, 4);
  strcpy(message+4, msg_body);
```

To read a message sent by XSB, one can do as follows:

```
int actual_len;
char lenbuf[4], msg_buff;
unsigned int msglen, net_encoded_len;

  actual_len = (long)recvfrom(sock_handle, lenbuf, 4, 0, NULL, 0);
  memcpy((void *) &net_encoded_len, (void *) lenbuf, 4);
  msglen = ntohl(net_encoded_len);
```

```
msg_buff = calloc(msglen+1, sizeof(char)); // check if this suceeded!!!
recvfrom(sock_handle, msg_buff, msglen, 0, NULL, 0);
```

If making the external processes follow the XSB protocol is not practical (because you did not write these programs), then you should use the character-at-a-time interface or, better, the buffered stream-based interface both of which are described in this section. At present, however, the buffered stream-based interface does not work on Windows.

`socket_recv(+Sockfd,-Message, ?ErrorCode)`
: Receives a message from the connection identified by the socket descriptor `Sockfd`. Binds `Message` to the message. `socket_recv/3` provides a message-oriented interface. It understands message boundaries set by `socket_send/3`.

`socket_send(+Sockfd,+Message, ?ErrorCode)`
: Takes a message (which must be an atom) and sends it through the connection specified by `Sockfd`. `socket_send/3` provides message-oriented communication. It prepends a 4-byte header to the message, which tells `socket_recv/3` the length of the message body.

Stream-oriented, character-at-a-time interface. Internally, this interface uses the same `sendto` and `recvfrom` socket calls, but they are executed for each character separately. This interface is appropriate when the message format is not known or when message boundaries are determined using special delimiters.

`socket_get0/3` creates the end-of-file condition when it receives the end-of-file character `CH_EOF_P` (a.k.a. 255) defined in `char_defs.h` (which must be included in the XSB program). C programs that need to send an end-of-file character should send `(char)-1`.

`socket_get0(+Sockfd, -Char, ?ErrorCode)`
: The equivalent of `get0` for sockets.

`socket_put(+Sockfd, +Char, ?ErrorCode)`
: Similar to put/1, but works on sockets.

Socket-probing. These calls need more documentation.

`socket_select(+SymConName,+Timeout,-ReadSockL,-WriteSockL,-ErrSockL,?ErrorCode)`
: `SymConName` must be an atom that denotes an existing connection, which must be previously created with `socket_set_select/4` (described below). `ReadSockL`, `WriteSockL`, `ErrSockL` are lists of socket handles (as returned by `socket/2`) that specify the available sockets that are available for reading, writing, or on which exception conditions occurred.

`socket_set_select(+SymConName,+ReadSockFdLst,+WriteSockFdLst,+ErrorSockFdLst)`
: Creates a connection under the symbolic name `SymConName` (an atom) for subsequent use by `socket_select/6`. `ReadSockFdLst`, `WriteSockFdLst`, and `ErrorSockFdLst` are lists of sockets for which `socket_select/6` is to be monitoring read, write, or exception conditions.

`socket_select_destroy(+SymConName)`
Destroys the specified connection.

Error codes. The error code argument unifies with the error code returned by the corresponding socket commands. The error code -2 signifies *timeout* for timeout-enabled primitives (see below). The error code of zero signifies normal termination. Positive error codes denote specific failures, as defined in BSD sockets. When such a failure occurs, an error message is printed, but the predicate succeeds anyway. The specific error codes are part of the socket documentation. Unfortunately, the symbolic names and error numbers of these failures are different between Unix compilers and Visual C++. Thus, there is no portable, reliable way to refer to these error codes. The only reliably portable error codes that can be used in XSB programs defined through these symbolic constants:

```
#include "socket_defs_xsb.h"

#define SOCK_OK       0      /* indicates sucessful return from socket      */
#define SOCK_EOF     -1      /* end of file in socket_recv, socket_get0     */

#include "timer_defs_xsb.h"

#define TIMEOUT_ERR -2                  /* Timeout error code */
```

Timeouts. XSB socket interface allows the programer to specify timeouts for certain operations. If the operations does not finish within the specified period of time, the operation is aborted and the corresponding predicate succeeds with the `TIMEOUT_ERR` error code. The following primitives are timeout-enabled: `socket_connect/4`, `socket_recv/3`, `socket_send/3`, `socket_get0/3`, and `socket_put/3`. To set a timeout value for any of the above primitives, the user should execute `set_timer/1` right before the subgoal to be timed.

The most common use of timeouts is to either abort or retry the operation that timed out. For the latter, XSB provides the `sleep/1` primitive, which allows the program to wait for a few seconds before retrying.

The `set_timer/1` and `sleep/1` primitives are described below. They are standard predicates and do not need to be explicitly imported.

`set_timer(+Seconds)`
Set timeout value. If a timer-enabled goal executes after this value is set, the clock begins ticking. If the goal does not finish in time, it succeeds with the error code set to `TIMEOUT_ERR`. The timer is turned off after the goal executes (whether timed out or not and whether it succeeds or fails). This goal always succeeds.

Note that if the timer is not set, the timer-enabled goals execute "normally," without timeouts. In particular, they might block (say, on `socket_recv`, if data is not available).

`sleep(+Seconds)`
Put XSB to sleep for the specified number of seconds. Execution resumes after the `Seconds` number of seconds. This goal always succeeds.

Here is an example of the use of the timer:

```
:- compiler_options([xpp_on]).
#include "timer_defs_xsb.h"

?- set_timer(3),  % wait for 3 secs
   socket_recv(Sockfd, Msg, ErrorCode),
   (ErrorCode == TIMEOUT_ERR
   -> writeln('Socket read timed out, retrying'),
      try_again(Sockfd)
   ;  write('Data received: '), writeln(Msg)
   ).
```

Apart from the above timer-enabled primitives, a timeout value can be given to `socket_select/6` directly, as an argument.

Buffered, stream-oriented communication. In Unix, socket descriptors can be "promoted" to file streams and the regular read/write commands can be used with such streams. In XSB, such promotion can be done using the following predicate:

`fd2ioport(+Pipe, -IOport)` shell

Take a socket descriptor and convert it to an XSB I/O port that can be used for regular file I/O.

Once `IOport` is obtained, all I/O primitives described in Section 1.5 can be used. This is, perhaps, the easiest and the most convenient way to use sockets in XSB. (This feature has not been implemented for NT yet.)

Here is an example of the use of this feature:

```
:- compiler_options([xpp_on]).
#include "socket_defs_xsb.h"

?- (socket(Sockfd, SOCK_OK)
   ->   socket_connect(Sockfd1, 6020, localhost, Ecode),
        (Ecode == SOCK_OK
        -> fd2ioport(Sockfd, SockIOport),
           file_write(SockIOport, 'Hello Server!')
        ;  writeln('Can''t connect to server')
        ),
    ;   writeln('Can''t open socket'), fail
    ).
```

1.10 Arrays

The module `array1` in directory `lib` provides a very simple backtrackable array implementation. The predicates through which the array objects are manipulated are:

`array_new(-Array, +Size)` array1

Creates a one dimensional empty array of size `Size`. All the elements of this array are variables.

`array_elt(+Array, +Index, ?Element)` array1

True iff `Element` is the `Index`-th element of array `Array`.

`array_update(+Array, +Index, +Elem, -NewArray)` array1

Updates the array `Array` such that the `Index`-th element of the new array is `Elem` and returns the new array in `NewArray`. The implementation is quite efficient in that it avoids the copying of the entire array.

A small example that shows the use of these predicates is the following:

```
| ?- import [array_new/2, array_elt/3, array_update/4] from array1.

yes
| ?- array_new(A, 4), array_update(A,1,1,B), array_update(B,2,2,C),
     ( array_update(C,3,3,D), array_elt(D,3,E)
     ; array_update(C,3,6,D), array_elt(D,3,E)
     ; array_update(C,3,7,D), array_elt(D,3,E)
     ).

A = array(1,2,3,_874600)
B = array(1,2,3,_874600)
C = array(1,2,3,_874600)
D = array(1,2,3,_874600)
E = 3;

A = array(1,2,6,_874600)
B = array(1,2,6,_874600)
C = array(1,2,6,_874600)
D = array(1,2,6,_874600)
E = 6;

A = array(1,2,7,_874600)
B = array(1,2,7,_874600)
C = array(1,2,7,_874600)
D = array(1,2,7,_874600)
E = 7;

no
```

1.11 Asserts/Retracts using Tries

In Version 2.2, trie asserted code has been merged with standard asserted code. If the user wishes to use tries for dynamic code, the recommended programming practice is as outlined in the section *Modification of the Database* in Volume 1. For compatibility with previous versions, the predicates `trie_assert/1`, `trie_retract/1`, `trie_retractall/1`, `trie_retract_nr/1`, `abolish_trie_asserted/1` and `trie_dynamic/1` can be imported from the module `tables`. However, if the current index specification of these predicates is `trie` (again, see the section *Modification of the Database* in Volume 1, the predicates are defined as `assert/1`, `retract/1`, `retractall/1`, `retract_nr/1`, `abolish/1` and `dynamic/1` respectively. If the index specification is other than `tries`, the predicate will issue a warning message and have no effect on the database.

1.12 Extended Logic Programs

As explained in the section *Using Tabling in XSB*, XSB can compute normal logic programs according to the well-founded semantics. In fact, XSB can also compute *Extended Logic Programs*, which contain an operator for explicit negation (written using the symbol `-` in addition to the negation-by-failure of the well-founded semantics (`\+` or `not`). Extended logic programs can be extremely useful when reasoning about actions, for model-based diagnosis, and for many other uses [2]. The library, slx provides a means to compile programs so that they can be executed by XSB according to the *well-founded semantics with explicit negation* [1]. Briefly, WFSX is an extension of the well-founded semantics to include explicit negation and which is based on the *coherence principle* in which an atom is taken to be default false if it is proven to be explicitly false, intuitively:

$$-p \Rightarrow not\ p.$$

This section is not intended to be a primer on extended logic programming or on WFSX semantics, but we do provide a few sample programs to indicate the action of WFSX. Consider the program

```
s:- not t.

t:- r.
t.

r:- not r.
```

If the clause `-t` were not present, the atoms `r, t, s` would all be undefined in WFSX just as they would be in the well-founded semantics. However, when the clause `t` is included, `t` becomes true in the well-founded model, while `s` becomes false. Next, consider the program

```
s:- not t.

t:- r.
-t.

r:- not r.
```

In this program, the explicitly false truth value for `t` obtained by the rule `-t` overrides the undefined truth value for `t` obtained by the rule `t:- r`. The WFSX model for this program will assign the truth value of `t` as false, and that of `s` as true. If the above program were contained in the file `test.P`, an XSB session using `test.P` might look like the following:

```
> xsb

| ?- [slx].
[slx loaded]

yes
| ?- slx_compile('test.P').
[Compiling ./tmptest]
[tmptest compiled, cpu time used: 0.1280 seconds]
[tmptest loaded]

| ?- s.

yes
| ?- t.

no
| ?- naf t.

yes
| ?- r.

no
| ?- naf r.

no
| ?- und r.

yes
```

In the above program, the query `?- t.` did not succeed, because `t` is false in WFSX: accordingly the query `naf t` did succeed, because it is true that t is false via negation-as-failure, in addition to `t` being false via explicit negation. Note that after being processed by the SLX preprocessor, `r` is undefined but does not succeed, although `und r` will succeed.

We note in passing that programs under WFSX can be paraconsistent. For instance in the program.

```
p:- q.

q:- not q.
-q.
```

both `p` and `q` will be true *and* false in the WFSX model. Accordingly, under SLX preprocessing, both `p` and `naf p` will succeed.

`slx_compile(+File)` slx

Preprocesses and loads the extended logic program named `File`. Default negation in `File` must be represented using the operator `not` rather than using `tnot` or `\+`. If `L` is an objective literal (e.g. of the form A or $-A$ where A is an atom), a query `?- L` will succeed if `L` is true in the WFSX model, `naf L` will succeed if `L` is false in the WFSX model, and `und L` will succeed if `L` is undefined in the WFSX model.

1.13 Generalized Annotated Programs

Generalized Annotated Programs (GAPs) [5] offer a powerful computational framework for handling paraconsistentcy and quantitative information within logic programs. The tabling of XSB is well-suited to implementing GAPs, and the gap library provides a meta-interpreter that has proven robust and efficient enough for a commercial application in data mining. The current meta-interpreter is limited to range-restricted programs.

A description of GAPs along with full documentation for this meta-interpreter is provided in [7] (currently also available at `http://www.cs.sunysb.edu/ tswift`). Currently, the interface to the GAP library is through the following call.

`meta(?Annotated_atom)` gap

If `Annotated_atom` is of the form `Atom:[Lattice_type,Annotation]` the meta-interpreter computes bindings for `Atom` and `Annotation` by evaluating the program according to the definitions provided for `Lattice_type`.

Chapter 2

Foreign Language Interface

When XSB is used to build real-world systems, a foreign-language interface may be necessary to:

- combine XSB with existing programs and libraries, thereby forming composite systems;
- interface XSB with the operating system, graphical user interfaces or other system level programs;
- speed up certain critical operations.

XSB has both the high-level and the low-level interfaces to C. The low-level interface is much more flexible, but it requires greater attention to details of how the data is passed between XSB and C. To connect XSB to a C program using the high-level interface requires very little work, but the program must be used "as is" and it must take the input and produce the output supported by this high-level interface. We first describe the low-level interface.

2.1 Compiler Directives for Foreign C Modules

Foreign predicates must always appear in modules, which can contain only foreign predicates. The main difference between a normal module and a foreign module is the very natural one: the source file of the module implementation, which is in C, must appear in a `*.c` file rather than a `*.P` file. This `*.c` file cannot contain a `main()` function. Furthermore, a `*.P` file with the same name *must not* be present or else the `*.c` file is ignored and the module is compiled as a regular Prolog module. The interface part of a foreign module, which has the same syntax as that of a normal module, is written in Prolog and hence must appear in a `*.H` file. This `*.H` file contains `export` declarations for each and every one of the foreign predicates that are to be used by other modules. Here is an example of a `.H` file for a foreign module:

```
:- export minus_one/2, my_sqrt/2, change_char/4.

:- ldoption('-lm').     % link together with the math library
```

Directives such as `index`, `hilog`, `table`, `auto_table` or even `import` make no sense in the case of a foreign module and thus are ignored by the compiler. However, another directive, namely `ldoption`, is recognized in a foreign module and is used to instruct the dynamic loading and linking of the module. The syntax of the `ldoption` directive is simply:

```
:- ldoption(Option).
```

where `Option` should either be an atom or a list of atoms. Multiple `ldoption` directives may appear in the same `.H` file of a foreign module.

The foreign language interface of XSB uses the Unix command `ld` that combines object programs to create an executable file or another object program suitable for further `ld` processing. Version 2.2 of XSB assumes that the `ld` command resides in the file `/usr/bin/ld`.

C functions that implement foreign predicates must return values of type `int`. If a non-zero is returned, the foreign predicate succeeds; a zero return value means failure.

A well-designed foreign predicate must check that its arguments are of the correct types and modes. However, such checks can also be done using Prolog-side wrappers that invoke a foreign predicate.

At the C level, the procedure that implements the foreign predicate must have the same name as the predicate (that is declared in the `*.H` file), and it must be *parameterless*. The Prolog level arguments are converted to C data structures through several predefined functions rather than through direct parameter passing.

In the current implementation, the Prolog procedures that are attached to foreign predicates are deterministic, in the sense that they succeed at most once for a given call and are not re-entered on backtracking. Note that this requirement imposes no serious limitation, since it is always possible to divide a foreign predicate into the part to be done on the first call and the part to be redone on backtracking. Backtracking can then take place at the Prolog level where it is more naturally expressed.

A foreign module can be `compile`d or `consult`ed just like a normal Prolog module. Currently, predicates `consult/[1,2]` recompile both the `*.c` and the `*.H` files of a foreign module when at least one of them has been changed from the time the corresponding object files have been created (see the section *Compiling and Consulting* in Volume 1. The C compiler used to compile the `*.c` files can be set as a compilation option or defaults to that used for the configuration of XSB (refer to the section *Getting Started with XSB* in Volume 1. Moreover, the user can control the compiler options that can be passed to the C compiler. To give an example, the following command will compile file `file.c` using the Gnu C Compiler with optimization and by including `/usr/local/X11/R6/include` to the directories that will be searched for header files.

```
:- consult(file, [cc(gcc), cc_opts('-O2 -I/usr/local/X11/R6/include')]).
```

If no C compiler options are specified, the compilation of the C-file defaults to *CC* `-c file.c` where *CC* is the name of the C compiler used to install XSB. Any Prolog compiler options are ignored when compiling a foreign module.

2.2 Foreign Modules That Link Dynamically with Other Libraries

Sometimes a foreign module might have to link dynamically with other (non-XSB) libraries. Typically, this happens when the foreign module implements an interface to a large external library of utilities. One example of this is the package `libwww` in the XSB distribution, which provides a high-level interface to the W3C's Libwww library for accessing the Web. The library is compiled into a set of shared objects and the `libwww` module has to link with them as well as with XSB.

The problem here is that the loader must know at run time where to look for the shared objects to link with. On Unix systems, this is specified using the environment variable `LD_LIBRARY_PATH`; on Windows, the variable name is `LIBPATH`. For instance, under Bourne shell or its derivatives, the following will do:

```
LD_LIBRARY_PATH=dir1:dir2:dir3
export LD_LIBRARY_PATH
```

One problem with this approach is that this variable must be set before starting XSB. The other problem is that such a global setting might interact with other foreign modules.

To alleviate the problem, XSB dynamically sets `LD_LIBRARY_PATH` (`LIBPATH` on Windows) before loading foreign modules by adding the directories specified in the `-L` option in `ldoption`. Unfortunately, this works on some systems (Linux), but not on others (Solaris). One route around this difficulty is to build a runtime library search path directly into the object code of the foreign module. This can be specified using a loader flag in `ldoption`. The problem here is that different systems use a different flag! To circumvent this, XSB provides a predicate that tries to guess the right flag for your system:

```
runtime_loader_flag(+Hint,-Flag)
```

Currently it knows about a handful of the most popular systems, but this will be expanded. The argument `Hint` is not currently used. It might be used in the future to provide `runtime_loader_flag` with additional information that can improve the accuracy of finding the right runtime flags for various systems.

The above predicate can be used as follows:

```
...,
runtime_loader_flag(_,Flag),
fmt_write_string(LDoptions, '%sdir1:dir2:dir2 %s', args(Flag,OldLDoption)),
fmt_write(File, ':- ldoption(%s).', LDoptions),
file_nl(File).
```

2.3 Passing Data between XSB and C

The XSB foreign language interface can be split in two parts. The *basic* interface supports the exchange of Prolog's atomic data types (atoms, integers, and floating-point numbers). The *advanced* interface allows passing lists and terms between XSB and C.

2.3.1 Exchanging Basic Data Types

The basic interface assumes that correct modes (*i.e.*, input or output) and types are being passed between C and the Prolog level. So, output unification should be explicitly performed in the Prolog level. The function prototypes should be declared before the corresponding functions are used. This is done by including the `"cinterf.h"` header file. Under Unix, the XSB foreign C interface automatically finds this file in the `XSB/emu` directory. Under Windows, the user must compile and create the DLL out of the C file manually, so the compiler option '`/I...\XSB\emu`' is necessary.

The following C functions are used to convert basic Prolog and C data types to each other.

`int ptoc_int(int N)`
: Argument `N` is assumed to hold a Prolog integer, and this function returns its integer value in C format.

`float ptoc_float(int N)`
: Argument `N` is assumed to hold a Prolog floating point number, and this function returns its floating point value in C format. (Precision is less than single word floating point).

`char *ptoc_string(int N)`
: Argument `N` is assumed to hold a Prolog atom, and this function returns the C string (of type `char *`) that corresponds to this Prolog atom.

`void ctop_int(int N, int V)`
: Argument `N` is assumed to hold a Prolog free variable, and this function binds that variable to an integer of value `V`.

`void ctop_float(int N, float V)`
: Argument `N` is assumed to hold a Prolog free variable, and this function binds that variable to a floating point number of value `V`.

`void ctop_string(int N, char * V)`
: Argument `N` is assumed to hold a Prolog free variable, and this function binds that variable to a Prolog atom of value `V`. In C, `V` is of type `char *`.

  Note that the atom of value `V` is not interned, i.e. it is not inserted into the Prolog atom table. For that reason, the `string_find(char *V, int Insert)` function should be used. Function `string_find()` searches the symbol table for the symbol, and if the symbol does not appear there and the value of `Insert` is non-zero, it inserts it. Thus, the most common use of this function is as follows:

```
ctop_string(N, string_find(V, 1))
```

  Refer to the example `simple_foreign` in the `examples` directory to see a use of this function.

Examples of Using the Basic C interface

We end by a very simple example of using the foreign language interface of XSB. The programs above and below are programs `simple_foreign.{H,c}` in the `examples` directory.

```
#include <math.h>
#include <stdio.h>
#include <string.h>
#include <alloca.h>

/*----- Make sure your C compiler finds the following header file.    -----
  ----- The best way to do this is to include the directory XSB/emu   -----
  ----- on compiler's command line with the -I (/I in Windows) option -----*/

#include "cinterf.h"

/*----------------------------------------------------------------------*/

int minus_one(void)
{
   int  i = ptoc_int(1);

   ctop_int(2, i-1);
   return TRUE;
}

/*----------------------------------------------------------------------*/

int my_sqrt(void)
{
   int i = ptoc_int(1);

   ctop_float(2, (float) pow((double)i, 0.5));
   return TRUE;
}

/*----------------------------------------------------------------------*/

int change_char(void)
{
   char *str_in;
   int  pos;
   int c;
   char *str_out;

   str_in = (char *) ptoc_string(1);
   str_out = (char *) alloca(strlen(str_in)+1);
   strcpy(str_out, str_in);
   pos = ptoc_int(2);
```

```
   c = ptoc_int(3);
   if (c < 0 || c > 255) /* not a character */
     return FALSE; /* this predicate will fail on the Prolog side */

   str_out[pos-1] = c;

   /* Now that we have constructed a new symbol, we must ensure that it
      appears in the symbol table.  This can be done using function
      string_find() that searches the symbol table for the symbol, and
      if the symbol does not appear there, it inserts it.  If we are
      sure that the symbol already appeared in the symbol table there
      is no need to use string_find().
    */

   ctop_string(4, (char *) string_find(str_out,1));  /* 1 = INSERT */
   return TRUE;
}

/*----------------------------------------------------------------------*/
```

Here is a sample session illustrating the use of these files.

```
XSB Version 2.0 (Gouden Carolus) of June 26, 1999
[i686-pc-linux-gnu; mode: optimal; engine: slg-wam; scheduling: batched]
| ?- [simple_foreign].
[Compiling C file ./simple_foreign.c using gcc]
[Compiling Foreign Module ./simple_foreign]
[simple_foreign compiled, cpu time used: 0.0099993 seconds]
[simple_foreign loaded]

yes
| ?- change_char('Kostis', 2, 119, TempStr), % 119 is w
     change_char(TempStr, 5, 104, GrkName).  % 104 is h

TempStr = Kwstis
GrkName = Kwsths;

no
| ?- minus_one(43, X).

X = 42;

no
| ?- minus_one(43, 42).                   % No output unification is allowed
Wrong arg in ctop_int 2a2 (Reg = 2)

yes
| ?- my_sqrt(4,X).

X = 2
```

```
yes
| ?- my_sqrt(23,X).

X = 4.7958;

no
```

There are additional sample programs in the `examples` directory that exhibit most of the features of the foreign language interface.

2.3.2 Exchanging Complex Data Types

The advanced XSB/Prolog interface uses only one data type: `prolog_term`. A Prolog term (as the name suggests) can be bound to any XSB term. On the C side, the type of the term can be checked and then processed accordingly. For instance, if the term turns out to be a structure, then it can be decomposed and the functor can be extracted along with the arguments. If the term happens to be a list, then it can be processed in a loop and each list member can be further decomposed into its atomic components. The advanced interface also provides functions to check the types of these atomic components and for converting them into C types.

As with the basic C interface, the file `emu/cinterf.h` must be included in the C program in order to make the prototypes of the relevant functions known to the C compiler.

The first set of functions is typically used to check the type of Prolog terms passed into the C program.

`bool is_var((prolog_term) T)`
: `is_var(T)` returns TRUE if `T` represents an XSB variable, and FALSE otherwise.

`bool is_int((prolog_term) T)`
: `is_int(T)` returns TRUE if `T` represents an XSB integer value, and FALSE otherwise.

`bool is_float((prolog_term) T)`
: `is_float(T)` returns TRUE if `T` represents an XSB float value, and FALSE otherwise.

`bool is_string((prolog_term) T)`
: `is_string(T)` returns TRUE if `T` represents an XSB atom value, and FALSE otherwise.

`bool is_functor((prolog_term) T)`
: `is_functor(T)` returns TRUE if `T` represents an XSB structure value (not a list), and FALSE otherwise.

`bool is_list((prolog_term) T)`
: `is_list(T)` returns TRUE if `T` represents an XSB list value (not nil), and FALSE otherwise.

`bool is_nil((prolog_term) T)`
: `is_nil(T)` returns TRUE if `T` represents an XSB `[]` (nil) value, and FALSE otherwise.

After checking the types of the arguments passed in from the Prolog side, the next task usually is to convert Prolog data into the types understood by C. This is done with the following functions. The first three convert between the basic types. The last two extract the functor name and the arity. Extraction of the components of a list and the arguments of a structured term is explained later.

`int p2c_int((prolog_term) V)`
: The prolog_term argument must represent an integer, and `p2c_int` returns the value of that integer.

`double p2c_float((prolog_term) V)`
: The prolog_term argument must represent a floating point number, and `p2c_float` returns the value of that floating point number.

`char *p2c_string((prolog_term) V)`
: The prolog_term argument must represent an atom, and `p2c_string` returns the name of that atom as a string. The pointer returned points to the actual atom name in XSB's space, and thus it must NOT be modified by the calling program.

`char *p2c_functor((prolog_term) V)`
: The prolog_term argument must represent a structured term (not a list). `p2c_functor` returns the name of the main functor symbol of that term as a string. The pointer returned points to the actual functor name in XSB's space, and thus it must NOT be modified by the calling program.

`int p2c_arity((prolog_term) V)`
: The prolog_term argument must represent a structured term (not a list). `p2c_arity` returns the arity of the main functor symbol of that term as an integer.

The next batch of functions support conversion of data in the opposite direction: from basic C types to the type `prolog_term`. These `c2p_*` functions all return a boolean value TRUE if successful and FALSE if unsuccessful. The XSB term argument must always contain an XSB variable, which will be bound to the indicated value as a side effect of the function call.

`bool c2p_int((int) N, (prolog_term) V)`
: `c2p_int` binds the prolog_term V (which must be a variable) to the integer value N.

`bool c2p_float((double) F, (prolog_term) V)`
: `c2p_float` binds the prolog_term V (which must be a variable) to the (double) float value F.

`bool c2p_string((char *) S, (prolog_term) V)`
: `c2p_string` binds the prolog_term V (which must be a variable) to the atom whose name is the value of S, which must be of type char *.

The following functions create Prolog data structures within a C program. This is usually done in order to pass these structures back to the Prolog side.

`bool c2p_functor((char *) S, (int) N, (prolog_term) V)`
: `c2p_functor` binds the prolog_term V (which must be a variable) to an open term whose main functor symbol is given by S (of type char *) and whose arity is N. An open term is one with all arguments as new distinct variables.

`bool c2p_list((prolog_term) V)`
: `c2p_list` binds the prolog_term V (which must be a variable) to an open list term, i.e., a list term with both car and cdr as new distinct variables. Note: to create an empty list use the function `c2p_nil` described below.

`bool c2p_nil((prolog_term) V)`
: `c2p_nil` binds the prolog_term V (which must be a variable) to the atom `[]` (nil).

`prolog_term p2p_new()`
: Create a new Prolog variable. This is sometimes needed when you want to create a Prolog term on the C side and pass it to the Prolog side.

To use the above functions, one must be able to get access to the components of the structured Prolog terms. This is done with the help of the following functions:

`prolog_term p2p_arg((prolog_term) T, (int) A)`
: Argument T must be a prolog_term that is a structured term (but not a list). A is a positive integer (no larger than the arity of the term) that specifies an argument position of the term `T`. `p2p_arg` returns the A^{th} subfield of the term `T`.

`prolog_term p2p_car((prolog_term) T)`
: Argument T must be a prolog_term that is a list (not nil). `p2p_car` returns the car (i.e., head of the list) of the term T.

`prolog_term p2p_cdr((prolog_term) T)`
: Argument T must be a prolog_term that is a list (not nil). `p2p_car` returns the cdr (i.e., tail of the list) of the term T.

It is very important to realize that these functions return the actual Prolog term that is, say, the head of a list or the actual argument of a structured term. Thus, assigning a value to such a prolog term also modifies the head of the corresponding list or the relevant argument of the structured term. It is precisely this feature that allows passing structured terms and lists from the C side to the Prolog side. For instance,

```
prolog_term plist,        /* a Prolog list           */
            structure;    /* something like f(a,b,c) */
prolog_term tail, arg;
..........
tail = p2p_cdr(plist);         /* get the list tail  */
arg  = p2p_arg(structure, 2);  /* get the second arg */
```

```
/* Assume that the list tail was supposed to be a prolog variable */
if (is_var(tail))
   c2p_nil(tail);  /* terminate the list */
else {
   fprintf(stderr, "Something wrong with the list tail!");
   exit(1);
}
/* Assume that the argument was supposed to be a prolog variable */
c2p_string("abcdef", arg);
```

In the above program fragment, we assume that both the tail of the list and the second argument of the term were supposed to be bound to Prolog variables. In case of the tail, we check if this is, indeed, the case. In case of the argument, no checks are done; XSB will issue an error (which might be hard to track down) if the second argument is not currently bound to a variable.

The last batch of functions is useful for passing data in and out of the Prolog side of XSB. The first function is the only way to get a `prolog_term` out of the Prolog side; the second function is sometimes needed in order to pass complex structures from C into Prolog.

`prolog_term reg_term((int) R)`
: Argument R is an argument number of the Prolog predicate implemented by this C function (range 1 to 255). The function `reg_term` returns the prolog_term in that predicate argument.

`bool p2p_unify(prolog_term T1, prolog_term T2)`
: Unify the two Prolog terms. This is useful when an argument of the Prolog predicate (implemented in C) is a structured term or a list, which acts both as input and output parameter.

For instance, consider the Prolog call `test(X, f(Z))`, which is implemented by a C function with the following fragment:

```
prolog_term newterm, newvar, z_var, arg2;
.....
/* process argument 1 */
c2p_functor("func",1,reg_term(1));
c2p_string("str",p2p_arg(reg_term(1),1));
/* process argument 2 */
arg2 = reg_term(2);
z_var = p2p_arg(arg2, 1);  /* get the var Z */
/* bind newterm to abc(V), where V is a new var */
c2p_functor("abc", 1, newterm);
newvar = p2p_arg(newterm, 1);
newvar = p2p_new();
....
/* return TRUE (success), if unify; FALSE (failure) otherwise */
return p2p_unify(z_var, newterm);
```

On exit, the variable X will be bound to the term `func(str)`. Processing argument 2 is more interesting. Here, argument 2 is used both for input and output. If `test` is called as above, then on exit Z will be bound to `abc(_h123)`, where `_h123` is some new Prolog variable. But if the call is `test(X,f(1))` or `test(X,f(Z,V))` then this call will *fail* (fail as in Prolog, *i.e.*, it is not an error), because the term passed back, `abc(_h123)`, does not unify with `f(1)` or `f(Z,V)`. This effect is achieved by the use of `p2p_unify` above.

We conclude with two real examples of functions that pass complex data in and out of the Prolog side of XSB. These functions are part of the Posix regular expression matching package of XSB. The first function uses argument 2 to accept a list of complex prolog terms from the Prolog side and does the processing on the C side. The second function does the opposite: it constructs a list of complex Prolog terms on the C side and passes it over to the Prolog side in argument 5.

```
/* XSB string substitution entry point: replace substrings specified in Arg2
   with strings in Arg3.
   In:
       Arg1: string
       Arg2: substring specification, a list [s(B1,E1),s(B2,E2),...]
       Arg3: list of replacement string
   Out:
       Arg4: new (output) string
   Always succeeds, unless error.
*/
int do_regsubstitute__(void)
{
  /* Prolog args are first assigned to these, so we could examine the types
     of these objects to determine if we got strings or atoms. */
  prolog_term input_term, output_term;
  prolog_term subst_reg_term, subst_spec_list_term, subst_spec_list_term1;
  prolog_term subst_str_term=(prolog_term)0,
    subst_str_list_term, subst_str_list_term1;
  char *input_string=NULL;    /* string where matches are to be found */
  char *subst_string=NULL;
  prolog_term beg_term, end_term;
  int beg_offset=0, end_offset=0, input_len;
  int last_pos = 0; /* last scanned pos in input string */
  /* the output buffer is made large enough to include the input string and the
     substitution string. */
  char subst_buf[MAXBUFSIZE];
  char *output_ptr;
  int conversion_required=FALSE; /* from C string to Prolog char list */

  input_term = reg_term(1);  /* Arg1: string to find matches in */
  if (is_string(input_term)) /* check it */
    input_string = string_val(input_term);
  else if (is_list(input_term)) {
```

```
    input_string =
      p_charlist_to_c_string(input_term, input_buffer, sizeof(input_buffer),
                             "RE_SUBSTITUTE", "input string");
    conversion_required = TRUE;
  } else
    xsb_abort("RE_SUBSTITUTE: Arg 1 (the input string) must be an atom or a character list");

  input_len = strlen(input_string);

  /* arg 2: substring specification */
  subst_spec_list_term = reg_term(2);
  if (!is_list(subst_spec_list_term) && !is_nil(subst_spec_list_term))
    xsb_abort("RE_SUBSTITUTE: Arg 2 must be a list [s(B1,E1),s(B2,E2),...]");

  /* handle substitution string */
  subst_str_list_term = reg_term(3);
  if (! is_list(subst_str_list_term))
    xsb_abort("RE_SUBSTITUTE: Arg 3 must be a list of strings");

  output_term = reg_term(4);
  if (! is_var(output_term))
    xsb_abort("RE_SUBSTITUTE: Arg 4 (the output) must be an unbound variable");

  subst_spec_list_term1 = subst_spec_list_term;
  subst_str_list_term1 = subst_str_list_term;

  if (is_nil(subst_spec_list_term1)) {
    strncpy(output_buffer, input_string, sizeof(output_buffer));
    goto EXIT;
  }
  if (is_nil(subst_str_list_term1))
    xsb_abort("RE_SUBSTITUTE: Arg 3 must not be an empty list");

  /* initialize output buf */
  output_ptr = output_buffer;

  do {
    subst_reg_term = p2p_car(subst_spec_list_term1);
    subst_spec_list_term1 = p2p_cdr(subst_spec_list_term1);

    if (!is_nil(subst_str_list_term1)) {
      subst_str_term = p2p_car(subst_str_list_term1);
      subst_str_list_term1 = p2p_cdr(subst_str_list_term1);

      if (is_string(subst_str_term)) {
```

```
        subst_string = string_val(subst_str_term);
      } else if (is_list(subst_str_term)) {
        subst_string =
          p_charlist_to_c_string(subst_str_term, subst_buf, sizeof(subst_buf),
                                 "RE_SUBSTITUTE", "substitution string");
      } else
        xsb_abort("RE_SUBSTITUTE: Arg 3 must be a list of strings");
    }

    beg_term = p2p_arg(subst_reg_term,1);
    end_term = p2p_arg(subst_reg_term,2);

    if (!is_int(beg_term) || !is_int(end_term))
      xsb_abort("RE_SUBSTITUTE: Non-integer in Arg 2");
    else{
      beg_offset = int_val(beg_term);
      end_offset = int_val(end_term);
    }
    /* -1 means end of string */
    if (end_offset < 0)
      end_offset = input_len;
    if ((end_offset < beg_offset) || (beg_offset < last_pos))
      xsb_abort("RE_SUBSTITUTE: Substitution regions in Arg 2 not sorted");

    /* do the actual replacement */
    strncpy(output_ptr, input_string + last_pos, beg_offset - last_pos);
    output_ptr = output_ptr + beg_offset - last_pos;
    if (sizeof(output_buffer)
        > (output_ptr - output_buffer + strlen(subst_string)))
      strcpy(output_ptr, subst_string);
    else
      xsb_abort("RE_SUBSTITUTE: Substitution result size %d > maximum %d",
                beg_offset + strlen(subst_string),
                sizeof(output_buffer));

    last_pos = end_offset;
    output_ptr = output_ptr + strlen(subst_string);

  } while (!is_nil(subst_spec_list_term1));

  if (sizeof(output_buffer) > (output_ptr-output_buffer+input_len-end_offset))
    strcat(output_ptr, input_string+end_offset);

 EXIT:
  /* get result out */
```

```
  if (conversion_required)
    c_string_to_p_charlist(output_buffer,output_term,"RE_SUBSTITUTE","Arg 4");
  else
    /* DO NOT intern. When atom table garbage collection is in place, then
       replace the instruction with this:
                 c2p_string(output_buffer, output_term);
       The reason for not interning is that in Web page
       manipulation it is often necessary to process the same string many
       times. This can cause atom table overflow. Not interning allows us to
       circumvent the problem.  */
    ctop_string(4, output_buffer);

  return(TRUE);
}


/* XSB regular expression matcher entry point
   In:
       Arg1: regexp
       Arg2: string
       Arg3: offset
       Arg4: ignorecase
   Out:
       Arg5: list of the form [match(bo0,eo0), match(bo1,eo1),...]
             where bo*,eo* specify the beginning and ending offsets of the
             matched substrings.
             All matched substrings are returned. Parenthesized expressions are
             ignored.
*/
int do_bulkmatch__(void)
{
  prolog_term listHead, listTail;
  /* Prolog args are first assigned to these, so we could examine the types
     of these objects to determine if we got strings or atoms. */
  prolog_term regexp_term, input_term, offset_term;
  prolog_term output_term = p2p_new();
  char *regexp_ptr=NULL;      /* regular expression ptr               */
  char *input_string=NULL;    /* string where matches are to be found */
  int ignorecase=FALSE;
  int return_code, paren_number, offset;
  regmatch_t *match_array;
  int last_pos=0, input_len;
  char regexp_buffer[MAXBUFSIZE];

  if (first_call)
```

```
    initialize_regexp_tbl();

  regexp_term = reg_term(1);  /* Arg1: regexp */
  if (is_string(regexp_term)) /* check it */
    regexp_ptr = string_val(regexp_term);
  else if (is_list(regexp_term))
    regexp_ptr =
      p_charlist_to_c_string(regexp_term, regexp_buffer, sizeof(regexp_buffer),
                             "RE_MATCH", "regular expression");
  else
    xsb_abort("RE_MATCH: Arg 1 (the regular expression) must be an atom or a character list");

  input_term = reg_term(2);  /* Arg2: string to find matches in */
  if (is_string(input_term)) /* check it */
    input_string = string_val(input_term);
  else if (is_list(input_term)) {
    input_string =
      p_charlist_to_c_string(input_term, input_buffer, sizeof(input_buffer),
                             "RE_MATCH", "input string");
  } else
    xsb_abort("RE_MATCH: Arg 2 (the input string) must be an atom or a character list");

  input_len = strlen(input_string);

  offset_term = reg_term(3); /* arg3: offset within the string */
  if (! is_int(offset_term))
    xsb_abort("RE_MATCH: Arg 3 (the offset) must be an integer");
  offset = int_val(offset_term);
  if (offset < 0 || offset > input_len)
    xsb_abort("RE_MATCH: Arg 3 (=%d) must be between 0 and %d", input_len);

  /* If arg 4 is bound to anything, then consider this as ignore case flag */
  if (! is_var(reg_term(4)))
    ignorecase = TRUE;

  last_pos = offset;
  /* returned result */
  listTail = output_term;
  while (last_pos < input_len) {
    c2p_list(listTail); /* make it into a list */
    listHead = p2p_car(listTail); /* get head of the list */

    return_code = xsb_re_match(regexp_ptr, input_string+last_pos, ignorecase,
                               &match_array, &paren_number);
    /* exit on no match */
```

```
    if (! return_code) break;

    /* bind i-th match to listHead as match(beg,end) */
    c2p_functor("match", 2, listHead);
    c2p_int(match_array[0].rm_so+last_pos, p2p_arg(listHead,1));
    c2p_int(match_array[0].rm_eo+last_pos, p2p_arg(listHead,2));

    listTail = p2p_cdr(listTail);
    last_pos = match_array[0].rm_eo+last_pos;
  }
  c2p_nil(listTail); /* bind tail to nil */
  return p2p_unify(output_term, reg_term(5));
}
```

2.4 High Level Foreign Predicate Interface

The high-level foreign predicate interface was designed to release the programmer from the burden of having to write low-level code to transfer data from XSB to C and vice-versa. Instead, all the user needs to do is to describe each C function and its corresponding Prolog predicates in the .H files. The interface then automatically generates the *wrappers* that translate Prolog terms and structures to proper C types, and vice-versa. The *wrappers* are then automatically used when the foreign predicates are compiled.[1]

2.4.1 Declaration of high level foreign predicates

The basic format of a foreign predicate declaration is:

```
:- foreign_pred predname([+-]parg1, [+-]parg2,...)
        from funcname(carg1:type1, carg2:type2, ...):functype.
```

where:

`predname`
: is the name of the foreign predicate. This is the name of the Prolog predicate that will be created.

`parg1, parg2, ...`
: are the predicate arguments. Each argument is preceded by either '+' or '-', indicating its mode as input or output respectively. The names of the arguments must be the same as those used in the declaration of the corresponding C function. If a C argument is used both for input and output, then the corresponding Prolog argument can appear twice: once with "+" and once with "-". Also, a special argument `retval` is used to denote the argument that corresponds to the return value of the C function; it must always have the mode '-'.

[1] Please see the special instructions for Windows.

`funcname`
: is the name of the C function being *wrapped.* This is the C function given by the user, which will be exported as a Prolog foreign predicate.

`carg1, carg2, ...`
: is the list of arguments of the C function. The names used for the arguments must match the names used in the Prolog declaration.

`type1, type2, ...`
: are the types associated to the arguments of the C function. This is not the set of C types, but rather a set of descriptive types, as defined in Table 2.4.1.

`functype`
: is the return type of the C function.

Table 2.4.1 provides the correspondence between the types allowed on the C side of a foreign module declaration and the types allowed on the Prolog side of the declaration.

In all modes and types, checks are performed to ensure the types of the arguments. Also, all arguments of type '-' are checked to be free variables at call time.

2.4.2 Compiling the foreign module on Windows

Due to the complexity of creating makefiles for the different compilers under Windows, XSB doesn't attempt to compile and build DLL's for the Windows foreign modules.

Instead, the user has to create the DLL herself. The process is, roughly, as follows: first, compile the module from within XSBṪhis will create the XSB-specific object file, and the *wrappers.* The *wrappers* are created in a file named `xsb_wrap_`*modulename*`.c`.

Then, the user has to create a project, using the compiler of choice, for a dynamically-linked library. In this project, the user must include the source code of the module along with the *wrapper* created by XSB. In addition, this DLL should be linked against the library

```
XSB\config\x86-pc-windows\bin\xsb.lib
```

which is distributed with XSB.

Descriptive Type	Mode Usage	Associated C Type	Comments
int	+	int	integer numbers
float	+	double	floating point numbers
atom	+	unsigned long	atom represented as an unsigned long
chars	+	char *	the textual representation of an atom is passed to C as a string
chars(*size*)	+	char *	the textual representation of an atom is passed to C as a string in a buffer of size *size*
string	+	char *	a prolog list of characters is passed to C as a string
string(*size*)	+	char *	a prolog list of characters is passed to C as a string
term	+	prolog_term	the unique representation of a term
intptr	+	int *	the location of a given integer
floatptr	+	double *	the location of a given floating point number
atomptr	+	unsigned long *	the location of the unique representation of a given atom
charsptr	+	char **	the location of the textual representation of an atom
stringptr	+	char **	the location of the textual representation of a list of characters
termptr	+	prolog_term *	the location of the unique representation of a term
intptr	-	int *	the integer value returned is passed to Prolog
floatptr	-	double *	the floating point number is passed back to Prolog
charsptr	-	char **	the string returned is passed to Prolog as an atom
stringptr	-	char **	the string returned is passed back as a list of characters
atomptr	-	unsigned long *	the number returned is passed back to Prolog as the unique representation of an atom
termptr	-	prolog_term *	the number returned is passed to Prolog as the unique representation of a term
chars(*size*)	+-	char *	the atom is copied from Prolog to a buffer, passed to C and converted back to Prolog afterwards
string(*size*)	+-	char *	the list of characters is copied from Prolog to a buffer, passed to C and back to Prolog afterwards
intptr	+-	int *	an integer is passed from Prolog to C and from C back to Prolog
floatptr	+-	double *	a float number is passed from Prolog to C, and back to Prolog
atomptr	+-	unsigned long *	the unique representation of an atom is passed to C, and back to Prolog
charsptr	+-	char **	the atom is passed to C as a string, and a string is passed to Prolog as an atom
stringptr	+-	char **	the list of characters is passed to C, and a string passed to Prolog as a list of characters
termptr	+-	prolog_term *	the unique representation of a term is passed to C, and back to Prolog

Table 2.1: Allowed combinations of types and modes, and their meanings

Chapter 3

Calling XSB from C

There are many situations in which it may be desirable to use XSB as a rule-processing subcomponent of a larger system, which is written in another language. To do this, one wants to be able to *call* XSB from the host language, often C, providing queries for XSB to evaluate and retrieving back the answers. An interface for calling XSB from C (or other language) is provided for this purpose and is described in this chapter. Simple examples of the use of this interface are given in the `XSB/examples/c_calling_XSB` subdirectory, in files `cmain.c`, `cmain2.c`, `ctest.P`, and `Makefile`.

3.1 C Functions for Calling XSB

XSB provides several C functions (declared in `XSB/emu/cinterf.h` and defined in `XSB/emu/cinterf.c`), which can be called from C to interact with XSB as a subroutine. These functions allow a C program to initialize XSB (most easily with a call to `xsb_init_string(options)`) and then to interact with XSB to have it execute *commands* or *queries*. A command is a deterministic query which simply succeeds or fails (without returning any interesting data value.) A non-deterministic query can be evaluated so that its answers are retrieved one at a time, as they are produced. There are several levels of interface provided. The highest level interface uses the XSB-specific C-type definition for variable-length strings (Section 3.2), to communicate queries to XSB and to get answers back. The `xsb_command_string(cmd)` function allows you to pass a command as a (period-terminated) string to XSB. The `xsb_query_string_string(query,buff,sep)` function allows you to pass a query to XSB as a string, and have its (first) answer returned as a string. Subsequent answers can be calculated and retrieved using `xsb_next_string(buff,sep)`.

The second level provides routines that return answers with an interface that does not require variable-length strings. The routines at this level are:

- `xsb_query_string_string_b(query,buff,bufflen,anslen,sep)`,
- `xsb_next_string_b(buff,bufflen,anslen,sep)`, and
- `xsb_get_last_answer(buff,bufflen,anslen)`.

They are normally intended to be used with the initialization routines above.

There are lower-level interfaces that allow you to manipulate directly the XSB data structures (both to construct queries and to retrieve answers) and thus avoid the overhead of converting to and from strings. See the detailed descriptions of the routines below to see how to use the lower level interface.

Currently, only one query can be active at a time. I.e., one must completely finish processing one query (either by retrieving all the answers for it, or by issuing a call to `xsb_close_query()`, before trying to evaluate another. The routines to perform all these functions are described below:

`int xsb_init(int argc, char *argv[])`

This is a C-callable function that initializes XSB. It must be called before any other calls can be made. `argc` is the count of the number of arguments in the `argv` vector. The `argv` vector is exactly as would be passed from the command line to XSB. It must contain at least the following two things:

- `argv`[0] must be an absolute or relative path name of the XSB installation directory (*i.e.*, `$XSB_DIR`. Here is an example, which assumes that we invoke the C program from the XSB installation directory.

```
int main(int argc, char *argv[])
{
  int myargc = 2;
  char *myargv[2];

  /* XSB_init relies on the calling program to pass the addr of the XSB
     installation directory. From here, it will find all the libraries */
  myargv[0] = ".";
  myargv[1] = "-n";

  /* Initialize xsb */
  xsb_init(myargc,myargv);
```

- `argv`[1] must be the `-n` flag. This flag tells XSB not to start the read-eval-print top loop, but to act as a subroutine to a calling C routine.

Other flags are optional, but can be used to modify sizes of the various spaces used in XSB. `xsb_init` returns 0 if initialization is completed, and 1 if an error is encountered.

`int xsb_init_string(char *options)`

This is a variant of `xsb_init` which takes the command line as a string argument (rather than as a argc/argv pair.) For example, a call could be

```
xsb_init_string(". -n");
```

Note that just as with `xsb_init`, you must pass the path name of the XSB installation directory. In the above, we pass ".", assuming that we are invoking the C program from the

XSB installation directory. The parameters following the file name are just as those that could appear on a command line. The function of this subroutine is exactly the same as `xsb_init`, and its return codes are the same.

`int xsb_command()`
This function passes a command to XSB. No query can be active when this command is called. Before calling `xsb_command`, the calling program must construct the XSB term representing the command in register 1 in XSB's space. This can be done by using the `c2p_*` (and `p2p_*`) routines, which are described in Section 2.3.2 below. Register 2 may also be set before the call to `xsb_query` (using `xsb_make_vars(int)` and `xsb_set_var_*()`) in which case any variables set to values in the `ret/n` term will be so bound in the call to the command goal. `xsb_command` invokes the command represented in register 1 and returns 0 if the command succeeds and 1 if it fails. In either case it resets register 1 back to a free variable. If there is an error, it returns 2.

`int xsb_command_string(char *cmd)`
This function passes a command to XSB. The command is a string consisting of a term that can be read by the XSB reader. The string must be terminated by a period (.). Any previous query must have already been closed. In all other respects, `xsb_command_string` is similar to `xsb_command`.

`int xsb_query()`
This function passes a query to XSB. Any previous query must have already been closed. A query is expected to return possibly multiple data answers. The first is found and made available to the caller as a result of this call. To get subsequent answers, `xsb_next` must be called. Before calling `xsb_query` the caller must construct the term representing the query in XSB's register 1 (using routines described in Section 2.3.2 below.) If the query has no answers (i.e., just fails), register 1 is set back to a free variable and `xsb_query` returns 1. If the query has at least one answer, the variables in the query term in register 1 are bound to those answers and `xsb_query` returns 0. In addition, register 2 is bound to a term whose main functor symbol is `ret/n`, where n is the number of variables in the query. The main subfields of this term are set to the variable values for the first answer. (These fields can be accessed by the functions `p2c_*`, or the functions `xsb_var_*`, described in Section 2.3.2 below.) Thus there are two places the answers are returned. Register 2 is used to make it easier to access them. To get subsequent answers, `xsb_next` must be called. Register 2 may also be set before the call to `xsb_query` (using `xsb_make_vars(int)` and `xsb_set_var_*()`) in which case any variables set to values in the `ret/n` term will be so bound in the call to the goal.

`int xsb_query_string(char *query)`
This function passes a query to XSB. The query is a string consisting of a term that can be read by the XSB reader. The string must be terminated with a period (.). Any previous query must have already been closed. In all other respects, xsb_query_string is similar to xsb_query, except the only way to retrieve answers is through Register 2. The ability to create the return structure and bind variables in it is particularly useful in this function.

`int xsb_query_string_string(char *query, VarString *buff, char *sep)`
This function is a variant of `xsb_query_string` that returns its answer (if there is one) as a

string. An example call is:

```
rc = xsb_query_string_string("append(X,Y,[a,b,c]).",buff,";");
```

The first argument is the period-terminated query string. The second argument is a variable string buffer in which the subroutine returns the answer (if any.) The variable string data type `VarString` is explained in Section 3.2. (Use the following function if you cannot declare a parameter of this type in your programming language.) The last argument is a string provided by the caller, which is used to separate fields in the returned answer. For the example query, buff would be set to the string:

```
[];[a,b,c]
```

which is the first answer to the append query. There are two fields of this answer, corresponding to the two variables in the query, `X` and `Y`. The bindings of those variables make up the answer and the individual fields are separated by the `sep` string, here the semicolon (`;`). Its returns are just as for `xsb_query_string`. In the answer string, XSB atoms are printed in their in their standard print form (without quotes). Complex terms are printed in a canonical form, with atoms quoted if necessary, and lists produced in the normal list notation.

`int xsb_query_string_string_b(char *query, char *buff, int bufflen, int *anslen, char *sep)`

This function provides a lower-level interface to `xsb_query_string_string` (not using the `VarString` type), which makes it easier for non-C callers (such as Visual Basic or Delphi) to access XSB functionality. The first and last arguments are the same as in `xsb_query_string_string`. The `buff`, `bufflen`, and `anslen` parameters are used to pass the answer (if any) back to the caller. `buff` is a buffer provided by the caller in which the answer is returned. `bufflen` is the length of the buffer (`buff`) and is provided by the caller. `anslen` is returned by this routine and is the length of the computed answer. If that length is less than `bufflen`, then the answer is put in `buff` (and null-terminated). If the answer is longer than will fit in the buffer (with the null terminator), then the answer is not copied to the buffer and 3 is returned. In this case the caller can retrieve the answer by providing a bigger buffer (of size greater than the returned `anslen`) in a call to `xsb_get_last_answer_string`.

`int xsb_get_last_answer_string(char *buff, int bufflen, int *anslen)`

This function is used only when a call to `xsb_query_string_string_b` or to `xsb_next_string_b` returns a 3, indicating that the buffer provided was not big enough to contain the computed answer. In that case the user may allocate a larger buffer and then call this routine to retrieve the answer (that had been saved.) Only one answer is saved, so this routine must called immediately after the failing call in order to get the right answer. The parameters are the same as the 2nd through 4th parameters of `xsb_query_string_string_b`.

`int xsb_next()`

This routine is called after `xsb_query` (which must have returned 0) to retrieve more answers. It rebinds the query variables in the term in register 1 and rebinds the argument fields of the `ret/n` answer term in register 2 to reflect the next answer to the query. It returns 0 if an answer is found, and returns 1 if there are no more answers and no answer is returned. On a return of 1, the query has been closed. After a query is closed, another `xsb_command` or `xsb_query` invocation can be made.

`int xsb_next_string(VarString *buff,char *sep)`

This routine is a variant of `xsb_next` that returns its answer (if there is one) as a string. Its treatment of answers is just as `xsb_query_string_string`. For example after the example call

```
rc = xsb_query_string_string("append(X,Y,[a,b,c]).",buff,";");
```

which returns with buff set to

```
[];[a,b,c]
```

Then a call:

```
rc = xsb_next_string(buff,";");
```

returns with buff set to

```
[a];[b,c]
```

the second answer to the indicated query. `xsb_next_string` returns codes just as `xsb_next`.

`int xsb_next_string_b(char *buff, int bufflen, int *anslen, char *sep)`

This routine is a variant of `xsb_next_string` that does not use the `VarString` type. Its parameters are the same as the 2nd through 5th parameters of `xsb_query_string_string_b`. The next answer to the current query is returned in `buff`, if there is enough space. If the buffer would overflow, this routine returns 3, and the answer can be retrieved by providing a larger buffer in a call to `xsb_get_last_answer_string_b`. In any case, the length of the answer is returned in `anslen`.

`int xsb_close_query()`

This routine closes a query, before all its answers have been retrieved. Since XSB is (usually) a tuple-at-a-time system, answers that are not retrieved are not computed. It is an error to call `xsb_query` again without first either retrieving all the answers to the previous query or calling `xsb_close_query` to close it.

`int xsb_close()`

This routine closes the entire connection to XSB. After this, no more calls can be made (including calls to `xsb_init`.)

3.2 The Variable-length String Data Type

XSB uses variable-length strings to communicate with certain C subroutines when the size of the output that needs to be passed from the Prolog side to the C side is not known. Variable-length strings adjust themselves depending on the size of the data they must hold and are ideal for this situation. For instance, as we have seem the two subroutines `xsb_query_string_string(query,buff,sep)` and `xsb_next_string(buff,sep)` use the variable string data type, `VarString`, for their second argument. To use this data type, make sure that

```
#include "cinterf.h"
```

appears at the top of the program file. Variables of the `VarString` type are declared using a macro that must appear in the declaration section of the program:

```
vstrDEFINE(buf);
```

There is one important consideration concerning VarString with the *automatic* storage class: they must be *destroyed* on exit (see `vstrDESTROY`, below) from the procedure that defines them, or else there will be a memory leak. It is not necessary to destroy static `VarString`'s.

The public attributes of the type are `int length` and `char *string`. Thus, `buf.string` represents the actual contents of the buffer and `buf.length` is the length of that data. Although the length and the contents of a `VarString` string is readily accessible, the user **must not** modify these items directly. Instead, he should use the macros provided for that purpose:

- `vstrSET(VarString *vstr, char *str)`: Assign the value of the regular null-terminated C string to the `VarString` `vstr`. The size of `vstr` is adjusted automatically.
- `vstrSETV(VarString *vstr1, VarString *vstr2)`: Like `vstrSET`, but the second argument is a variable-length string, not a regular C string.
- `vstrAPPEND(VarString *vstr, char *str)`: Append the null-terminated string `str` to the `VarString` `vstr`. The size of `vstr` is adjusted.
- `vstrPREPEND(VarString *vstr, char *str)`: Like `vstrAPPEND`, except that `str` is prepended.
- `vstrAPPENDV(VarString *vstr1, VarString *vstr2)`: Like `vstrAPPEND`, except that the second string is also a `VarString`.
- `vstrPREPENDV(VarString *vstr1, VarString *vstr2)`: Like `vstrAPPENDV`, except that the second string is prepended.
- `vstrCOMPARE(VarString *vstr1, VarString *vstr2)`: Compares two `VarString`. If the first one is lexicographically larger, then the result is positive; if the first string is smaller, than the result is negative; if the two strings have the same content (*i.e.*, `vstr1->string` equals `vstr2->string` then the result is zero.
- `vstrSTRCMP(VarString *vstr, char *str)`: Like `vstrCOMPARE` but the second argument is a regular, null-terminated string.

- `vstrAPPENDBLK(VarString *vstr, char *blk, int size)`: This is like `vstrAPPEND`, but the second argument is not assumed to be null-terminated. Instead, `size` characters pointed to by `blk` are appended to `vstr`. The size of `vstr` is adjusted, but the content is *not* null terminated.
- `vstrPREPENDBLK(VarString *vstr, char *blk, int size)`: Like `vstrPREPEND`, but `blk` is not assumed to point to a null-terminated string. Instead, `size` characters from the region pointed to by `blk` are prepended to `vstr`.
- `vstrNULL_TERMINATE(VarString *vstr)`: Null-terminates the `VarString` string `vstr`. This is used in conjunction with `vstrAPPENDBLK`, because the latter does not null-terminate variable-length strings.
- `vstrENSURE_SIZE(VarString *vstr, int minsize)`: Ensure that the string has room for at least `minsize` bytes. This is a low-level routine, which is used to interface to procedures that do not use `VarString` internally. If the string is larger than `minsize`, the size might actually shrink to the nearest increment that is larger `minsize`.
- `vstrSHRINK(VarString *vstr, int increment)`: Shrink the size of `vstr` to the minimum necessary to hold the data. `increment` becomes the new increment by which `vstr` is adjusted. Since `VarString` is automatically shrunk by `vstrSET`, it is rarely necessary to shrink a `VarString` explicitly. However, one might want to change the adustment increment using this macro (the default increment is 128).
- `vstrDESTROY(VarString *vstr)`: Destroys a `VarString`. Explicit destruction is necessary for `VarString`'s with the automatic storage class. Otherwise, memory leak is possible.

3.3 Passing Data into an XSB Module

The previous chapter described the low-level XSB/C interface that supports passing the data of arbitrary complexity between XSB and C. However, in cases when data needs to be passed into an executable XSB module by the main C program, the following higher-level interface should suffice. (This interface is actually implemented using macros that call the lower level functions.) These routines can be used to construct commands and queries into XSB's register 1, which is necessary before calling `xsb_query()` or `xsb_command()`.

`void xsb_make_vars((int) N)`

`xsb_make_vars` creates a return structure of arity `N` in Register 2. So this routine may called before calling any of `xsb_query`, `xsb_query_string`, `xsb_command`, or `xsb_command_string` if parameters are to be set to be sent to the goal. It must be called before calling one of the `xsb_set_var_*` routines can be called. `N` must be the number of variables in the query that is to be evaluated.

`void xsb_set_var_int((int) Val, (int) N)`

`set_and_int` sets the N^{th} field in the return structure to the integer value `Val`. It is used to set the value of the N^{th} variable in a query before calling `xsb_query` or `xsb_query_string`. When called in XSB, the query will have the N^{th} variable set to this value.

`void xsb_set_var_string((char *) Val, (int) N)`
: `set_and_string` sets the $\mathtt{N}^{th}$ field in the return structure to the atom with name `Val`. It is used to set the value of the $\mathtt{N}^{th}$ variable in a query before calling `xsb_query` or `xsb_query_string`. When called in XSB, the query will have the $\mathtt{N}^{th}$ variable set to this value.

`void xsb_set_var_float((float) Val, (int) N)`
: `set_and_float` sets the $\mathtt{N}^{th}$ field in the return structure to the floating point number with value `Val`. It is used to set the value of the $\mathtt{N}^{th}$ variable in a query before calling `xsb_query` or `xsb_query_string`. When called in XSB, the query will have the $\mathtt{N}^{th}$ variable set to this value.

`void xsb_var_int((int) N)`
: `xsb_var_int` is called after `xsb_query` or `xsb_query_string` returns an answer. It returns the value of the $\mathtt{N}^{th}$ variable in the query as set in the returned answer. This variable must have an integer value.

`void xsb_var_string((int) N)`
: `xsb_var_string` is called after `xsb_query` or `xsb_query_string` returns an answer. It returns the value of the $\mathtt{N}^{th}$ variable in the query as set in the returned answer. This variable must have an atom value.

`void xsb_var_float((int) N)`
: `xsb_var_float` is called after `xsb_query` or `xsb_query_string` returns an answer. It returns the value of the $\mathtt{N}^{th}$ variable in the query as set in the returned answer. This variable must have an floating point value.

3.4 Creating an XSB Module that Can be Called from C

To create an executable that includes calls to the above C functions, these routines, and the XSB routines that they call, must be included in the link (`ld`) step.

Unix instructions: You must link your C program, which should include the main procedure, with the XSB object file located in

```
$XSB_DIR/config/<your-system-architecture>/saved.o/xsb.o
```

Your program should include the file `cinterf.h` located in the `XSB/emu` subdirectory, which defines the routines described earlier, which you will need to use in order to talk to XSB. It is therefore recommended to compile your program with the option `-I$XSB_DIR/XSB/emu`.

The file `$XSB_DIR/config/your-system-architecture/modMakefile` is a makefile you can use to build your programs and link them with XSB. It is generated automatically and contains all the right settings for your architecture, but you will have to fill in the name of your program, etc.

It is also possible to compile and link your program with XSB using XSB itself as follows:

```
:- xsb_configuration(compiler_flags,CFLAGS),
       xsb_configuration(loader_flags,LDFLAGS),
       xsb_configuration(config_dir,CONFDIR),
       xsb_configuration(emudir,EMUDIR),
       xsb_configuration(compiler,Compiler),
       str_cat(CONFDIR, '/saved.o/', ObjDir),
       write('Compiling myprog.c ... '),
       shell([Compiler, ' -I', EMUDIR, ' -c ', CFLAGS, ' myprog.c ']),
       shell([Compiler, ' -o ', './myprog ',
              ObjDir, 'xsb.o ', ' myprog.o ', LDFLAGS]),
       writeln(done).
```

This works for every architecture and is often more convenient than using the make files.

There are simple examples of C programs calling XSB in the `$XSB_DIR/examples/c_calling_XSB` directory, in files `cmain.c`, `ctest.P`, `cmain2.c`.

Windows instructions: To call XSB from C, you must build it as a DLL, which is done as follows:

```
cd $XSB_DIR\XSB\build
makexsb_wind DLL="yes"
```

The DLL, which you can call dynamically from your program is then found in

`$XSB_DIR\config\x86-pc-windows\bin\xsb.dll`

Since your program must include the file `cinterf.h`, it is recommended to compile it with the option `/I$XSB_DIR\XSB\emu`.

Chapter 4

XSB's POSIX Regular Expression and Wildcard Matching Packages

By Michael Kifer

XSB has an efficient interface to POSIX pattern regular expression and wildcard matching functions. To take advantage of these features, you must build XSB using a C compiler that supports POSIX 1.0 (for regular expression matching) and the forthcoming POSIX 2.0 (for wildcard matching). The recent versions of GCC and SunPro compiler will do, as probably will many other compilers. This also works under Windows, provided you install Cygnus' CygWin and use GCC to compile.

4.1 Regular Expression Matching and Substitution

The following discussion assumes that you are familiar with the syntax of regular expressions and have a reasonably good idea about their capabilities. One easily accessible description of POSIX regular expressions is found in the on-line Emacs manual.

The regular expression matching functionality is provided by the package called `Regmatch`. To use it interactively, type:

```
:- [regmatch].
```

If you are planning to use pattern matching from within an XSB program, then you need to include the following directive:

```
:- import re_match/5, re_bulkmatch/5,
          re_substitute/4, re_substring/4,
          re_charlist_to_string/2
   from regmatch.
```

Matching. The predicates `re_match/5` and `re_bulkmatch/5` perform regular expression matching. The predicate `re_substitute/4` replaces substrings in a list with strings from another list and returns the resulting new string.

The `re_match/5` predicate has the following calling sequence:

```
re_match(+Regexp, +InputStr, +Offset, ?IgnoreCase, -MatchList)
```

`Regexp` is a regular expression, *e.g.*, "`abc([^;,]*); (dd|ee)*;`". It can be a Prolog atom or string (*i.e.*, a list of characters). The above expression matches any substring that has "abc" followed by a sequence of characters none of which is a ";" or a ",", followed by a "; ", followed by a sequence that consists of zero or more of "dd" or "ee" segments, followed by a ";". An example of a string where such a match can be found is "`123abc&*^; ddeedd;poi`".

`InputStr` is the string to be matched against. It can be a Prolog atom or a string (list of characters). `Offset` is an integer offset into the string. The matching process starts at this offset. `IgnoreCase` indicates whether the case of the letters is to be ignored. If this argument is an uninstantiated variable, then the case is *not* ignored. If this argument is bound to a non-variable, then the case *is* ignored.

The last argument, `MatchList`, is used to return the results. It must unify with a list of the form:

```
[match(beg_off0,end_off0), match(beg_off1,end_off1), ...]
```

The term `match(beg_off0,end_off0)` represents the substring that matches the *entire* regular expression, and the terms `match(beg_off1,end_off1)`, ..., represent the matches corresponding to the *parenthesized subexpressions* of the regular expression. The terms `beg_off` and `end_off` above are integers that specify beginning and ending offsets of the various matches. Thus, `beg_off0` is the offset into `InputStr` that points to the start of the maximal substring that matches the entire regular expression; `end_off0` points to the end of such a substring. In our case, the maximal matching substring is "`abc&*^; ddeedd;`" and the first term in the list returned by

```
| ?- re_match('abc([^;,]*); (dd|ee)*;', '123abc&*^; ddeedd;poi', 0, _,L).
```

is `match(3,18)`.

The most powerful feature of POSIX pattern matching is the ability to remember and return substrings matched by parenthesized subexpressions. When the above predicate succeeds, the terms 2,3, etc., in the above list represent the offsets for the matches corresponding to the parenthesized expressions 1,2,etc. For instance, our earlier regular expression "`abc([^;,]*); (dd|ee)*;`" has two parenthetical subexpressions, which match "`&*^`" and "`dd`, respectively. So, the complete output from the above call is:

```
L = [match(3,18),match(6,9),match(15,17)]
```

The maximal number of parenthetical expressions supported by the Regmatch package is 30. Partial matches to parenthetical expressions 31 and over are discarded.

The match-terms corresponding to parenthetical expressions can sometimes report "*no-use.*" This is possible when the regular expression specifies that zero or more occurrences of the parenthesized subexpression must be matched, and the match was made using zero subexpressions. In this case, the corresponding match term is `match(-1,-1)`. For instance,

```
| ?- re_match('ab(de)*', 'abcd',0,_,L).
L = [match(0,2),match(-1,-1)]
yes
```

Here the match that was found is the substring "ab" and the parenthesized subexpression "de" was not used. This fact is reported using the special match term `match(-1,-1)`.

Here is one more example of the power of POSIX regular expression matching:

```
| ?- re_match("a(b*|e*)cd\\1",'abbbcdbbbbbo', 0, _, M).
```

Here the result is:

```
M = [match(0,9),match(1,4)]
```

The interesting features here are the positional parameter \\1 and the alternating parenthetical expression `a(b*|e*)`. The alternating parenthetical expression here can match any sequence of b's *or* any sequence of e's. Note that if the string to be matched is not known when we write the program, we will not know a priori which sequence will be matched: a sequence of b's or a sequence of e's. Moreover, we do not even know the length of that sequence.

Now, suppose, we want to make sure that the matching substrings look like this:

```
abbbcdbbb
aeeeecdeeee
abbbbbbcdbbbbbb
```

How can we make sure that the suffix that follows "cd" is exactly the same string that is stuck between "a" and "cd"? This is what \\1 precisely does: it represents the substring matched by the first parenthetical expression. Similarly, you can use \\2, etc., if the regular expression contains more than one parenthetical expression.

The following example illustrates the use of the offset argument:

```
| ?- re_match("a(b*|e*)cd\\1",'abbbcdbbbbboabbbcdbbbbbo',2,_,M).

M = [match(12,21),match(13,16)]
```

Here, the string to be matched is double the string from the previous example. However, because we said that matching should start at offset 2, the first half of the string is not matched.

The `re_match/5` predicate fails if `Regexp` does not match `InputStr` or if the term specified in `MatchList` does not unify with the result produced by the match. Otherwise, it succeeds.

We should also note that parenthetical expressions can be represented using the `\(...\)` notation. What if you want to match a "(" then? You must escape it with a "\\" then:

```
| ?- re_match("a(b*)cd\\(",'abbbcd(bbo', 0, _, M).

M = [match(0,7),match(1,4)]
```

Now, what about matching the backslash itself? Try harder: you need four backslashes:

```
| ?- re_match("a(b*)cd\\\\",'abbbcd\bbo', 0, _, M).

M = [match(0,7),match(1,4)]
```

The predicate `re_bulkmatch/5` has the same calling sequence as `re_match/5`, and the meaning of the arguments is the same, except the last (output) argument. The difference is that `re_bulkmatch/5` ignores parenthesized subexpressions in the regular expression and instead of returning the matches corresponding to these parenthesized subexpressions it returns the list of all matches for the top-level regular expression. For instance,

```
| ?- re_bulkmatch('[^a-zA-Z0-9]+', '123&*-456 )7890% 123', 0, 1, X).

X = [match(3,6),match(9,11),match(15,17)]
```

Extracting the matches. The predicate `re_match/5` provides us with the offsets. How can we actually get the matched substrings? This is done with the help of the predicate `re_substring/4`:

```
re_substring(+String, +BeginOffset, +EndOffset, -Result).
```

This predicate works exactly like `substring/4` described in Section 1.6, except that the resulting substring is not interned (if it is an atom). All you can do with this string is to immediately convert it into a list (using `atom_codes/2`) or into a true atom (using `intern_string/2`, which must be imported from module machine).

The reason for these complications is to allow the user to control the size of the atom table. At present, XSB does not have atom table garbage collection, so heavy use of string manipulation functions can result in atom table overflow. This danger is particularly severe when XSB is used for processing HTML pages. This predicate will become an alias to `substring/4` when atom garbage collection will be added to XSB.

On the other hand, converting strings into lists (without interning them first) is safe, because lists are garbage-collected in XSB Version 2.0.

Here is a complete example that shows matching followed by a subsequent extraction of the matches:

```
| ?- import intern_string/2 from machine.
```

```
| ?- Str = 'abbbcd\bbo',
      re_match("a(b*)cd\\\\",Str,0,_,[match(X,Y), match(V,W)|L]),
      re_substring(Str,X,Y,UninternedMatch),
      intern_string(UninternedMatch,Match),
      re_substring(Str,V,W,UninternedParen1),
      atom_codes(UninternedParen1,Paren1).

Str = abbbcd\bbo
X = 0
Y = 7
V = 1
W = 4
L = []
UninternedMatch = abbbcd\
Match = abbbcd\
UninternedParen1 = bbb
Paren1 = [98,98,98]
```

Note that the strings `UninternedMatch` and `UninternedParen1` cannot be used by themselves. In the first case, we converted the string into a Prolog atom and in the second case into a string. The resulting objects (`Match` and `Paren1`) can be used in further computations.

Observe that XSB reports that `UninternedMatch` and `UninternedParen1` are both equal the string "bbb", while `Match` — the atom obtained from `UninternedMatch` — is different. This is because `UninternedMatch` and `UninternedParen1` are uninterned and both occupy the same physical space. Thus, the second call to `re_substring/4` overrides the value stored in this location by the first call.

Substitution. The predicate `re_substitute/4` has the following invocation:

```
re_substitute(+InputStr, +SubstrList, +SubstitutionList, -OutStr)
```

This predicate works exactly like `string_substitute/4` described in Section 1.6, except that the result of the substitution is not interned (for the same reason as in `re_substring/4`. This predicate will become an alias to `string_substitute/4` when atom garbage collection will be added to XSB.

```
| ?- re_bulkmatch('[^a-zA-Z0-9]+', '123&*-456 )7890| 123', 0, _, X),
     re_substitute('123&*-456 )7890| 123', X, ['+++'], Y).

X = [match(3,6),match(9,11),match(15,17)]
Y = 123+++456+++7890+++123
```

Efficiency considerations.

- Try not to work with too many regular expressions at once. Before a regular expression can be used, it must be compiled (which `re_match/5` does automatically). `re_match/5` maintains a cache of compiled regular expressions, so they do not need to be compiled each time they are used. However, if more than 8–10 expressions are used simultaneously, repeated recompilation might result.
- When a list of characters is passed to any one of the above predicates, it is converted into a C string. This can be expensive, if done too often for the same string.

  One way to circumvent the problem is to use `atom_codes/2` to first convert the list into an atom and then use that atom repeatedly in the match operations. One problem here might be the aforementioned overflow of the atom table. So, if this is a concern, the following predicate (which always succeeds) can help:

  ```
  re_charlist_to_string(+ListOfCharacters, -String)
  ```

  This predicate converts lists of characters into uninterned strings, which can be used without the fear of atom table overflow:

  ```
  | ?- re_charlist_to_string("abcdefg",L).

  L = abcdefg
  ```

  The resulting string can be passed to `re_match/5`, `re_substitute/5`, and `re_substring/4` for further processing.

  Note, however: you cannot call `re_charlist_to_string` before you finished working with the string generated by the previous call: all calls to this function use the same static buffer to hold the output string, so each subsequent call to `re_charlist_to_string` will override the previously generated strings.

4.2 Wildcard Matching and Globing

These interfaces are implemented using the `Wildmatch` package of XSB. This package provides the following functionality:

1. Telling whether a wildcard, like the ones used in Unix shells, match against a given string. Wildcards supported are of the kind available in tcsh or bash. Alternating characters (*e.g.*, "`[abc]`" or "`[^abc]`") are supported.
2. Finding the list of all file names in a given directory that match a given wildcard. This facility generalizes `directory/2` (in module `directory`), and it is much more efficient.
3. String conversion to lower and upper case.

To use this package, you need to type:

```
| ?- [wildmatch].
```

If you are planning to use it in an XSB program, you need this directive:

```
:- import glob_directory/4, wildmatch/3, convert_string/3 from wildmatch.
```

The calling sequence for `glob_directory/4` is:

```
glob_directory(+Wildcard, +Directory, ?MarkDirs, -FileList)
```

The parameter `Wildcard` can be either a Prolog atom or a Prolog string. `Directory` is also an atom or a string; it specifies the directory to be globbed. `MarkDirs` indicates whether directory names should be decorated with a trailing slash: if `MarkDirs` is bound, then directories will be so decorated. If MarkDirs is an unbound variable, then trailing slashes will not be added.

`FileList` gets the list of files in `Directory` that match `Wildcard`. If `Directory` is bound to an atom, then `FileList` gets bound to a list of atoms; if `Directory` is a Prolog string, then `FileList` will be bound to a list of strings as well.

This predicate succeeds is at least one match is found. If no matches are found or if `Directory` does not exist or cannot be read, then the predicate fails.

The calling sequence for `wildmatch/3` is as follows:

```
wildmatch(+Wildcard, +String, ?IgnoreCase)
```

`Wildcard` is the same as before. `String` represents the string to be matched against `Wildcard`. Like `Wildcard`, `String` can be an atom or a string. `IgnoreCase` indicates whether case of letters should be ignored during matching. Namely, if this argument is bound to a non-variable, then the case of letters is ignored. Otherwise, if `IgnoreCase` is a variable, then the case of letters is preserved.

This predicate succeeds when `Wildcard` matches `String` and fails otherwise.

The calling sequence for `convert_string/3` is as follows:

```
convert_string(+InputString, +OutputString, +ConversionFlag)
```

The input string must be an atom or a character list. The output string must be unbound. Its type will "atom" if so was the input and it will be a character list if so was the input string. The conversion flag must be the atom `tolower` or `toupper`.

This predicate always succeeds, unless there was an error, such as wrong type argument passed as a parameter.

Chapter 5

Using Perl as a Pattern Matching and String Substitution Server

By Michael Kifer and Jin Yu

XSB has an efficient interface to the Perl interpreter, which allows XSB programs to use powerful Perl pattern matching capabilities. this interface is provided by the `Perlmatch` package. You need Perl 5.004 or later to be able to take advantage of this service.

This package is mostly superseded by the the more efficient POSIX `Regmatch` package described in the previous section. However, Perl regular expressions provide certain features not available in the `Regmatch` package, such as the ability to perform global replacements of matched susbstrings. Also, the `Perlmatch` package has a different programming interface, which is modeled after the interface provided by Perl itself. So, if you are a big fan of Perl, this package is for you.

The following discussion assumes that you are familiar with the syntax of Perl regular expressions and have a reasonably good idea about the capabilities of Perl matching and string substitution functions.

In the interactive mode, you must first load the `Perlmatch` package:

```
:- [perlmatch].
```

In a program, you must import the package predicates:

```
:- import bulk_match/3, get_match_result/2, try_match/2,
  next_match/0, perl_substitute/3, load_perl/0, unload_perl/0
   from perlmatch.
```

5.1 Iterative Pattern Matching

There are two ways to do matching. One is to first do the matching operation and then count the beans. To find the first match, do:

```
:- try_match( +String, +Pattern ).
```

Both arguments must be of the XSB string data types. If there is a match in the string, the submatches $1, $2, etc., the prematch substring $‘ (*i.e.*, the part before the match), the postmatch substring $’ (*i.e.*, the part after the match), the whole matched substring $&, and the last parentheses match substring $+ will be stored into a global data structure, and the predicate try_match(string, pattern) succeeds. If no match is found, this predicate fails.

The ability to return parts of the match using the Perl variables $+, $1, $2, etc., is an extremely powerful feature of Perl. As we said, a familiarity with Perl matching is assumed, but we'll give an example to stimulate the interest in learning these capabilities of Perl. For instance, `m/(\d+)\.?(\d*)`(+)/— matches a valid number. Moreover, if the number had the form ‘xx.yy’, then the Perl variable $1 will hold ‘xx’ and $1 will hold ‘yy’. If the number was of the form ‘.zz’, then $1 and $2 will be empty, and $3 will hold ‘zz’.

XSB-Perl interface provides access to all these special variables using the predicate `get_match_result()`. The input variables string and pattern are of XSB string data types. For example:

For instance,

```
:- try_match('First 11.22 ; next:  3.4', 'm/(+)?(*)/g').
yes.
```

finds the character which precedes by ’s’ in the string ’this is a test’. The first match is ‘is’.

Now, we can use `get_match_result()` to find the submatches. The first argument is a tag, which can be 1 through 20, denoting the Perl variables $1 – $20. In addition, the entire match can be found using the tag `match`, the part before that is hanging off the tag `prematch` and the part of the string to the right of the match is fetched with the tag `postmatch`. For instance, in the above, we shall have:

This function is used to fetch the pattern match results $1, $2, etc., $‘, $’, $& and $+, as follows:

```
:- get_match_result(1,X).
X=11
yes
:- get_match_result(2,X).
X=22
yes
:- get_match_result(3,X).
no
:- get_match_result(4,X).
no
:- get_match_result(prematch,X)
X=First  (including 1 trailing space)
yes
:- get_match_result(postmatch,X)
X= ; next 3.4
```

```
yes
:- get_match_result(match,X)
11.22
yes
```

As you noticed, if a tag fetches nothing (like in the case of Perl variables $3, $4, etc.), then the predicate fails.

The above is not the only possible match, of course. If we want more, we can call:

```
:- next_match.
```

This will match the second number in the string. Correspondingly, we shall have:

```
:- get_match_result(1,X).
X=3
yes
:- get_match_result(2,X).
X=4
yes
:- get_match_result(3,X).
no
:- get_match_result(4,X).
no
:- get_match_result(prematch,X)
X=First 11.22 ; next
yes
:- get_match_result(postmatch,X)
no
:- get_match_result(match,X)
3.4
yes
```

The next call to `next_match` would fail, because there are no more matches in the given string. Note that `next_match` and `get_match_result` do not take a string and a pattern as argument—they work off of the previous `try_match`. If you want to change the string or the pattern, call `try_match` again (with different parameters).

Note: To be able to iterate using `next_match`, the perl pattern must be *global i.e.*, it must have the modifier 'g' at the end (*e.g.*, `m/a.b*/g`). Otherwise, `next_match` simply fails and only one (first) match will be returned.

5.2 Bulk Matching

XSB-perl interface also supports *bulk* matching through the predicate `bulk_match/3`. Here, you get all the substrings that match the patters at once, in a list, and then you can process the matches as you wish. However, this does not give you full access to submatches. More precisely, if you use parenthesized expressions, then you get the list of non-null values in the variables $1, $2, etc. If you do not use parenthesized regular expressions, you get the result of the full match. For instance,

```
:- bulk_match('First 11.22 ; next: 3.4', 'm/(\d+)\.?(\d*)/g', Lst).
Lst=[11,22,3,4]
yes
:- bulk_match('First 11.22 ; next: 3.4', 'm/\d+\.?\d*/g', Lst).
Lst=[11.22,3.4]
yes
```

`bulk_match/3` never fails. If there is no match, it returns an empty list.

Please note that you must specify 'g' at the end of the pattern in order to get something useful. This ia Perl thing! If you do not, instead of returning a list of matches, Perl will think that you just want to test if there is a match or not, and it will return `[1]` or `[]`, depending on the outcome.

5.3 String Substitution

The last feature of the XSB-Perl interface is string substitution based on pattern matching. This is achieved through the predicate `string_substitute/3`:

```
:- perl_substitute(+String, +PerlSubstitutionExpr, -ResultString).
```

We assume you are familiar with the syntax of Perl substitution expressions. Here we just give an example of what kind of things are possible:

```
:- perl_substitute('this is fun', 's/(this) (is)(.*)/\2 \1\3?/', Str).
Str=is this fun?
```

5.4 Unloading Perl

Playing with Perl is nice, but this also means that both XSB and the Perl interpreter are loaded in the main memory. If you do not need Perl for some time and memory is at premium, you can unload the Perl interpreter:

```
:- unload_perl.
```

This predicate always succeeds. If you need Perl matching features later, you can always come back to it: it is loaded automatically each time you use a pattern matching or a string substitution predicate.

Chapter 6

Libwww: The XSB Internet Access Package

By Michael Kifer

6.1 Features and Configuration

This package was inspired by the PiLLoW project. The XSB Libwww package offers much better performance and a superset of the PiLLoW functionality as related to the HTTP protocol, but this package does not implement the part of PiLLoW that deals with construction of Web pages.

The XSB Libwww is implemented in C and relies on the basic HTTP functions provided by the Libwww library developed by the WWW Consortium (`http://www.w3c.org/Library`). Therefore, this library must be installed in order for the XSB Libwww package to work. In addition, XSB must be configured to work with the Libwww library as follows:

```
configure --with-libwww=directory-where-Libwww-is-installed
```

One of the most important aspects of the Libwww package is that it allows XSB to dispatch multiple HTTP requests, which interleave their Web access phases. This can be a significant performance boost. Furthermore, the HTNL and the XML parsers begin their work as the fragments of pages arrive, so by the time the page is fully accepted, it is also parsed. Here is a list of features provided by the XSB Libwww package:

- HTML-4 parser.
- XML parser (non-validating).
- Page fetching (without parsing).
- Form handling.

- HTTP header information.
- Multiple, interleaved HTTP requests.
- Basic and digest authentication.
- Redirection and proxies.

6.2 Accessing Internet with Libwww

To start using the package, you must load it first:

```
:- [libwww].
```

The general form of a Web call is as follows:

```
:- libwww_request([request1, request2, ..., request_n]).
```

Each request has the following syntax:

```
request_type(+URL, +RequestParams, -ResponseParams, -Result, -Status)
```

The request type functor must be either `htmlparse`, `xmlparse`, `fetch`, or `header`. The first two are requests to parse HTML/XML pages, respectively. `Fetch` is a request to bring in a page without parsing, nd `header` is a request to retrieve only the header information (which is returned in the `ResponseParams` argument—see below). The URL must be an atom or a string (list of characters).[1] Request parameters must be either a variable (in which case the request is considered to not have special parameters) or a list. The following terms are allowed in that list:

- `timeout(+Secs)` – request timeout. If it is not specified, a default value (5 seconds) is used. Only the first request in a list should have the timeout value set. Timeouts that appear in subsequent requests are ignored.
- `authentication([c(+Realm,+Username,+Pasword),...])` – If the site requires authentication, you should specify it in a list as an argument to the `authentication/1` functor. `Realm` is a string that the servers return to let applications know which username/password pair to send (in case the application works with several pages that require different authentication). They are used as page identifiers for the authentication purposes. If `Realm` is an atom (*e.g.*, `authentication('FooSite', boo, moo)`), then when a Web server requests authentication for the `FooSite` realm, the Libwww package will send the foo/moo user-password pair.

  If `Realm` is a variable, then it is considered to match every realm. The Libwww package searches for matching authentication triples in the order they appear in the authentication list. Thus, the triple where `Realm` is a variable should appear last.

[1] The string feature will be deprecated when XSB will have working atom garbage collection. When URL is a list of characters, then Result is also a list of characters, which eases the burden on the atom table and allows XSB to work longer before memory is exhausted.

- `formdata([attval_pair1, attval_pair2, ...])` — list of attribute/value pairs to fill out a form (in case `URL` is a CGI script). Each attribute/value pair must be an atom of the form `attr=val`.
- `selection(Taglist1,Taglist2,Taglist3)` — if the request is `htmlparse` or `xmlparse`, then this term provides control over which tags to parse. `Taglist1` is a list of tags that specifies inside which tags to parse. For instance, if it is `[ul,form]` than parsing will be done only inside these elements. Other elements will be ignored. `Taglist2` tells the system to stop parsing inside the corresponding elements. For instance, `[table]` means that parsing should be done only inside `ul` and `form` elements. However, if we hit a table during parsing, then parsing should stop unless we hit `ul` or `form` inside the `table` element. This switching of parsing on and off can continue to arbitrary depth. `Taglist3` is a list of tags that are to be ignored completely. That is, the parsing process will simply strip these tags (but not the text inside them). For instance, if `Taglist3` is `[p,i]` and the page contains "<p>foo <i>moo</i>" then parsing will be done as if the page contained just "`foo moo`".

The `ResponseParams` argument is a list of terms returned by the `libwww_request` call. It contains two kinds of information: header information and sub-request information. The header information consists of terms like: `header('Content-Type', 'text/html')`, `header('Server', 'Netscape-Enterprise/3.6 SP2')`, etc., as defined by the HTTP protocol (`header/2` is a functor and its arguments are atoms). The sub-request information consists of terms of the form: `subrequest('http://www.foo.org/test/file.html',-401)`. It indicates that during processing of the current request, it was necessary to access another page, `http://www.foo.org/test/file.html`, but the server responded with the error code -401 (authentication error). Such sub-requests might be spawned during XML parsing.

The `Result` of a `libwww_request` call depends on the request type. In case of `fetch` it is an atom or a list of characters (depending on whether `URL` was specified as an atom or a list of characters), or it might be an unbound variable in case of an error. For `header` requests, `Result` is always an unbound variable.

For `htmlparse` and `xmlparse`, Result is a variable in case of an error and a complex term otherwise. In the latter case, it is a list of the form `[elt1,...,elt_n]`, where each `elt_i` is of the form:

```
elt(tag, [attval(attrname,value),...], [elt1',...,elt'_m])
```

The second argument here represents the list of attribute-value pairs. In HTML, some attributes, like `checked`, can be binary, in which case the corresponding value will be unbound. The third argument represents HTML or XML elements that are within the scope of `tag`. These elements have the same syntax as the parent element: `elt(tag',attrs,sub-elements)`. If a tag has no attributes or if it does not have sub-elements, the corresponding lists will be empty. One special tag, `pcdata`, is introduced to represent pieces of text that appear in the document. This tag is our own creation—neither HTML nor XML use tags to represent text. One important difference between `pcdata` and other tags is that the third argument in `elt(pcdata,...,...)` is an atom or a list of characters, not a list (unlike other tags). If URL was specified as an atom, then the third argument of the pcdata-element is an atom as well. If URL is a character list, then so is the corresponding argument in the pcdata-element.

Finally, Status is bound to an integer that represents the return code from the HTTP request. A complete list of return codes is given in XSB/prolog_includes/http_errors.h. If you need to refer to error codes in your Prolog application, it is advisable to use symbolic notation. To make this happen, put the following lines at the top of your program:

```
:- compiler_options([xpp_on]).
#include "http_errors.h"
```

The Libwww package also includes a predicate that is convenient for providing English language explanations to the errors:

```
:- import http_liberr/3 from usermod.
```

The first argument of this predicate is the error code, the second is an explanation in English, and the last is the class of the error (*e.g.*, internal, server error, client error, etc.). For full details see XSB/packages/libwww/http_liberr.P. Note that the code for a successful call is HT_LOADED (=200), not zero or one!

6.3 Example

Here is a complete example:

```
| ?- libwww_request([xmlparse('http://public.org/test/simple1.xml',[timeout(4)],
                                P,Y,Z)]),
      http_liberr(Z,Explanation,Class).

P = [header(Content-Type,text/html),
     subrequest('http://public.org/secret/001.ent',-401)]
Y = [elt(doc,[],[elt(pcdata,[],' '),
                 elt(foo,[attval(att1,123),attval(att2,ppp)],
                         [elt(pcdata,[],'Test1 '),elt(pcdata,[],' Test2 '),
                          elt(pcdata,[],adsdd),elt(pcdata,[], )]),
                 elt(pcdata,[],  ),
                 elt(a,[],[elt(pcdata,[], aaaaaaaaaa),
                 elt(pcdata,[],' '),
                 elt(b,[attval(att,1)],[]),
                 elt(pcdata,[],' '),
                 elt(c,[attval(att,2)],[]),
                 elt(pcdata,[],' '),
                 elt(d,[],[elt(pcdata,[],ddddddd),
                 elt(f,[],[elt(pcdata,[],kkkkkkk)]),
```

```
                     elt(pcdata,[],abc)]),elt(pcdata,[],'  ')])])]
Z = 200
Explanation = 'OK'
Class = 'success'
```

The above is a successful (because of the return code 200) request to parse an XML page. This page apparently had a reference to an external entity that was located in a protected domain. Since we did not supply authentication information, the call returned authentication failure for that subrequest (as indicated by the term `subrequest('http://public.org/secret/001.ent',-401)` in the fourth argument).

Chapter 7

XSB - Oracle Interface

By Hassan Davulcu and Ernie Johnson

7.1 Introduction

The XSB - Oracle interface provides the programmer with two levels of interaction. The first, *relation level interface*, offers a tuple-at-a-time retrieval of information from the Oracle tables. The second, *view level interface*, can translate an entire Prolog clause into a single SQL query to the Oracle, including joins and aggregate operations.

This interface allows Oracle tables to be accessed from XSB's environment as though they existed as facts. All database accesses are done on the fly allowing XSB to sit alongside other concurrent tasks.

Our interface gives an Oracle programmer all the features of Prolog as a query language including intensional database specification, recursion, the ability to deal with incomplete knowledge, inference control through the *cut* operation, and the representation of negative knowledge through negation.

7.1.1 Interface features

- Concurrent access for multiple XSB systems to Oracle 7.1.3 running under Solaris
- Full data access and cursor transparency including support for
  - Full data recursion through XSB's tabling mechanism
  - Runtime type checking
  - Automatic handling of NULL values for insertion, deletion and querying
  - Partial recovery for cursor losses due to cuts
- Full access to Oracle's SQLplus including

  - Transaction support
  - Cursor reuse for cached SQL statements with bind variables (by avoiding re-parsing and re-declaring).
  - Caching compiler generated SQL statements with bind variables and efficient cursor management for cached statements

- A powerful Prolog / SQL compiler based on [4].
- Full source code availability for ports to other versions of Oracle or other platforms
- Independence from database schema by employing *relation level*
- Performance as SQL by employing *view level*
- No mode specification is required for optimized view compilation

7.2 Installation:

The instructions below assume that Oracle is currently installed.

Unix instructions:

1. Set `LDFLAGS` to indicate the Oracle libraries needed to build the system. For instance:

   `LDFLAGS=-lclntsh -lcommon -lcore4 -lnlsrtl3`

   or

   `setenv LDFLAGS "-lclntsh -lcommon -lcore4 -lnlsrtl3"`

   depending on the shell that you are using. Note that libraries might be different depending on the version of Oracle or the OS in use. Also, the order of these libraries in the list is usually important.

2. When running `configure` during XSB installation, add these options: `--with-oracle` and `--site-static-libraries=ORACLE_LIB_PATH`, where `ORACLE_LIB_PATH` is the directory that has the Oracle client libraries.

   Sometimes, building Oracle requires that you use a C compiler other than the default one. Use `--with-cc=your-compiler` to tell configure which compiler to use.

3. run `configure` with appropriate options specified on command line.

4. when done, the `configure` script will tell you whether you should build XSB with just `makexsb` or pass an additional option.

5. When `makexsb` is done, it will tell use if you need to run XSB using the usual `.../bin/xsb` script or, maybe, something like `.../bin/xsb-ora`.

6. after starting XSB, load `ora_call.P` by `[ora_call].`

Windows instructions: To build XSB with Oracle support, type the following in the `emu` directory:

```
NMAKE /f "MS_VC_Mfile.mak" CFG="release" ORACLE="yes" SITE_LIBS="libraries"
```

The `SITE_LIBS` parameter should include the list of necessary Oracle support libraries (per Oracle instructions). When the compiler is done, the XSB executable is found in its usual place:

```
$XSB_DIR\config\x86-pc-windows\bin\xsb.exe
```

7.3 Using the interface:

7.3.1 Connecting to and disconnecting from Oracle:

Assuming the Oracle server is running, you have an account, and that the environment variables `ORACLE_SID, ORACLE_HOME` are set, you can login to Oracle by invoking `db_open/1` as:

```
| ?- db_open(oracle(name, passwd)).
```

If the login is successful, there will be a response of `yes`.

To reach a remote server you can use:

```
| ?- db_open(oracle('name@dblink', passwd)).
```

where `dblink` is the protocol, machine and server instance name. For example, `SCOTT@T:compserv1gw:INST` tells the runtime system we want to contact an oracle server instance whose `ORACLE_SID` is `INST` on the host `compserv1gw` using the TCP/IP protocol.

To disconnect from the current session use:

```
| ?- db_close.
```

7.3.2 Accessing an Oracle table: (relation level interface)

Assuming you have access permission for the table you wish to import, you can use `db_import/2` as:

```
| ?- db_import('TABLENAME'('FIELD1', 'FIELD2', .., 'FIELDn'), 'Pname').
```

where `'TABLENAME'` is the name of the table you wish to access and `'Pname'` is the name of the predicate you wish to use to access the table from XSB. `'FIELD1'` through `'FIELDn'` are the exact attribute names as defined in the database catalog. The chosen attributes define the view and the order of arguments for the database predicate `'Pname'`. For example, to create a link to the `DEPT` table through the `'dept'` predicate:

```
| ?- db_import('DEPT'('DEPTNO','DNAME','LOC'),dept).

yes
| ?- dept(Deptno, Dname, Loc).

Deptno = 10
Dname = ACCOUNTING
Loc = NEW YORK
```

Backtracking can then be used to retrieve the next row of the table `DEPT`.

Records with particular field values may be selected in the same way as in Prolog. (In particular, no mode specification for database predicates is required). For example:

```
| ?- dept(A, 'ACCOUNTING', C).
```

generates the query:

```
SELECT DEPTNO, LOC
FROM DEPT rel1
WHERE rel1.DNAME = :BIND1;
```

and

```
| ?- dept('NULL'(_), 'ACCOUNTING', C).
```

generates: (See section 7.3.7)

```
SELECT NULL , rel1.DNAME , rel1.LOC
FROM DEPT rel1
WHERE rel1.DEPTNO IS NULL AND rel1.DNAME = :BIND1;
```

During the execution of this query the `:BIND1` variable will be bound to `'ACCOUNTING'`.
If a field includes a quote (′) then this should be represented by using two quotes.

Note that the relation level interface can be used to define and access simple project views of single tables. For example:

```
| ?- db_import('DEPT'('DEPTNO','DNAME'),deptview).
```

defines `deptview/2`.

The predicate `db_import/2` (and other Oracle interface predicates) automatically asserts data dictionary information. You can use the Prolog predicate `listing/2` to see the asserted data dictionary information at any time.

Note: as a courtesy to Quintus Prolog users we have provided compatibility support for some PRODBI predicates which access tables at a relational level.

```
i) | ?- db_attach(Pname, table(Tablename)).
```

eg. execute

```
| ?- db_attach(dept, table('DEPT')).
```

then execute

```
| ?- dept(Depno, Dname, Loc).
```

to retrieve the rows.

```
ii) | ?- db_record('DEPT', R).

    R = [20,RESEARCH,DALLAS];

    R = ...
```

You can use `db_record/2` to treat the whole database row as a single list structure.

7.3.3 The view level interface:

The view level interface can be used for the definition of rules whose bodies includes only imported database predicates (by using the relation level interface) described above and aggregate predicates (defined below). In this case, the rule is translated into a complex database query, which is then executed taking advantage of the query processing ability of the database system.

One can use the view level interface through the predicate `db_query/2`:

```
| ?- db_query('Rulename'(Arg1, ...  , Argn), DatabaseGoal).
```

All arguments are standard Prolog terms. `Arg_1` through `Arg_n` defines the attributes to be retrieved from the database, while `DatabaseGoal` defines the selection restrictions and join conditions.

The compiler is a simple extension of [4] which generates SQL queries with bind variables and handles NULL values as described below (see section 7.3.7). It allows negation, the expression of arithmetic functions, and higher-order constructs such as grouping, sorting, and aggregate functions.

Database goals are translated according to the following rules from [4]:

- Disjunctive goals translate to distinct SQL queries connected through the UNION operator.
- Goal conjunctions translate to joins.

- Negated goals translate to negated EXISTS subqueries.
- Variables with single occurrences in the body are not translated.
- Free variables translate to grouping attributes.
- Shared variables in goals translate to equi-join conditions.
- Constants translate to equality comparisons of an attribute and the constant value.
- Nulls are translated to `IS NULL` conditions.

For more examples and implementation details see the demo in `$XSB_DIR/examples/xsb_ora_demo.P`, and [4].

In the following, we show the definition of a simple join view between the two database predicates *emp* and *dept*.

Assuming the declarations:

```
| ?- db_import('EMP'('ENAME','JOB','SAL','COMM','DEPTNO'),emp).

| ?- db_import('DEPT'('DEPTNO','DNAME','LOC'),dept).

use:

| ?- db_query(rule1(Ename,Dept,Loc),
          (emp(Ename,_,_,_,Dept),dept(Dept,Dname,Loc))).
yes

| ?- rule1(Ename,Dept,Loc).
```

generates the SQL statement:

```
SELECT rel1.ENAME , rel1.DEPTNO , rel2.LOC
FROM emp rel1 , DEPT rel2
WHERE rel2.DEPTNO = rel1.DEPTNO;

Ename = CLARK
Dept = 10
Loc = NEW YORK
```

Backtracking can then be used to retrieve the next row of the view.

```
| ?- rule1('CLARK',Dept,'NULL'(_)).
```

generates the SQL statement:

```
SELECT rel1.ENAME , rel1.DEPTNO , NULL
FROM emp rel1 , DEPT rel2
WHERE rel1.ENAME = :BIND1 AND rel2.DEPTNO = rel1.DEPTNO AND rel2.LOC IS NULL;
```

The view interface also supports aggregate functions predicates sum, avg, count, min and max. For example

```
| ?- db_query(a(X),(X is avg(Sal,A1 ^ A2 ^ A4 ^ A5 ^ emp(A1,A2,Sal,A4,A5)))).


yes.
| ?- a(X).

generates the query :

SELECT AVG(rel1.SAL)
FROM emp rel1;

X = 2023.2

yes
```

A more complicated example:

```
| ?- db_query(harder(A,B,D,E,S),
                           (emp(A,B,S,E,D),
                            not dept(D,P,C),
                            not (A = 'CAROL'),
                            S > avg(Sal,A1 ^ A2 ^ A4 ^ A5 ^ A6 ^ A7 ^(
                                    emp(A1,A2,Sal,A4,A5),
                                    dept(A5,A7,A6),
                                    not (A1 = A2))))).


| ?- harder(A,B,D,E,S).
```

generates the SQL query:

```
SELECT rel1.ENAME , rel1.JOB , rel1.DEPTNO , rel1.COMM , rel1.SAL
```

```
FROM emp rel1
WHERE NOT EXISTS
       (SELECT *
        FROM DEPT rel2
        WHERE rel2.DEPTNO = rel1.DEPTNO)
   AND rel1.ENAME <> 'CAROL'
   AND rel1.SAL >
(SELECT AVG(rel3.SAL)
         FROM emp rel3 , DEPT rel4
 WHERE rel4.DEPTNO = rel3.DEPTNO
            AND rel3.ENAME <> rel3.JOB);


A = SCOTT
B = ANALYST
D = 50
E = NULL(null1)
S = 2300
```

All database rules defined by db_query can be queried with any mode: For example:

```
| ?- harder(A,'ANALYST',D,'NULL'(_),S).
```

generates the query:

```
SELECT rel1.ENAME , rel1.JOB , rel1.DEPTNO , NULL , rel1.SAL
FROM emp rel1
WHERE rel1.JOB = :BIND1 AND rel1.COMM IS NULL AND NOT EXISTS
(SELECT *
FROM DEPT rel2
WHERE rel2.DEPTNO = rel1.DEPTNO
) AND rel1.ENAME <> 'CAROL' AND rel1.SAL >
(SELECT AVG(rel3.SAL)
FROM emp rel3 , DEPT rel4
WHERE rel4.DEPTNO = rel3.DEPTNO AND rel3.ENAME <> rel3.JOB
);


A = SCOTT
D = 50
S = 2300;

no
```

Notice that at each call to a database relation or rule, the communication takes place through bind variables. The corresponding restrictive SQL query is generated, and if this is the first call with that adornment, it is cached. A second call with same adornment would try to use the same database cursor if still available, without parsing the respective SQL statement. Otherwise, it would find an unused cursor and retrieve the results. In this way efficient access methods for relations and database rules can be maintained throughout the session.

7.3.4 Connecting to an SQL query

It is also possible to connect to any SQL query using the `db_sql_select/2` predicate which takes an SQL string as its input and returns a list of field values. For example:

```
| ?- db_sql_select('SELECT * FROM EMP',L).

L = [7369,SMITH,CLERK,7902,17-DEC-80,800,NULL,20];

L = etc ...
```

And you can use db_sql/1 for any other non-query SQL statement request. For example:

```
| ?- db_sql('create table test ( test1 number, test2 date)').

yes
```

7.3.5 Insertions and deletions of rows

Inserts are communicated to the database *array at a time*. To flush the buffered inserts one has to invoke `flush/0` at the end of his inserts.

For setting the size of the *input array* See section 7.3.6.

Assuming you have imported the related base table using `db_import/2`, you can insert to that table by using `db_insert/2` predicate. The first argument is the declared database predicate for insertions and the second argument is the imported database relation. The second argument can be declared with with some of its arguments bound to constants. For example assuming `empall` is imported through `db_import`:

```
|?- db_import('EMP'('EMPNO','ENAME','JOB','MGR','HIREDATE','SAL','COMM',
'DEPTNO'), empall).
yes
| ?- db_insert(emp_ins(A1,A2,A3,A4,A5,A6,A7),(empall(A1,A2,A3,A4,A5,A6,A7,10))).
```

```
yes
| ?- emp_ins(9001,'NULL'(35),'qqq',9999,'14-DEC-88',8888,'NULL'(_)).

yes
```

Inserts the row: 9001,NULL,'qqq',9999,'14-DEC-88',8888,NULL,10 Note that any call to `emp_ins/7` should have all its arguments bound.

See section 7.3.7 for information about NULL values.

Deletion of rows from database tables is supported by the `db_delete/2` predicate. The first argument is the declared delete predicate and the second argument is the imported database relation with the condition for requested deletes, if any. The condition is limited to simple comparisons. For example assuming `dept/3` is imported as above:

```
| ?- db_delete(dept_del(A), (dept(A,'ACCOUNTING',B), A > 10)).

yes
```

After this declaration you can use:

```
| ?- dept_del(10).
```

to generate the SQL statement:

```
DELETE DEPT rel1
WHERE rel1.DEPTNO = :BIND1
      AND rel1.DNAME = 'ACCOUNTING'
      AND rel1.DEPTNO > 10;
```

Note that you have to commit your inserts or deletes to tables to make them permanent. (See section 7.3.11).

7.3.6 Input and Output arrays

To enable efficient *array at a time* communication between the XSB client and the database server we employ *input* and *output* buffer areas.

The *input* buffer size specifies the size of the array size to be used during *insertions*. The *output* buffer size specifies the size of the array size to be used during *queries*. The default sizes of these arrays are set to 200. The sizes of these arrays can be queried by `stat_flag/2` and they can be modified by `stat_set_flag/2`. The flag number assigned for input array length is 58 and the flag number assigned for output array length is 60.

7.3.7 Handling NULL values

The interface treats NULL's by introducing a single valued function `'NULL'/1` whose single value is a unique (Skolem) constant. For example a NULL value may be represented by

```
'NULL'(null123245)
```

Under this representation, two distinct NULL values will not unify. On the other hand, the search condition `IS NULL Field` can be represented in XSB as `Field = 'NULL'(_)`

Using this representation of NULL's the following protocol for queries and updates is established.

Queries:

```
| ?- dept('NULL'(_),_,_).
```

Generates the query:

```
SELECT NULL , rel1.DNAME , rel1.LOC
  FROM DEPT rel1
  WHERE rel1.DEPTNO IS NULL;
```

Hence, `'NULL'(_)` can be used to retrieve rows with NULL values at any field.

`'NULL'/1` fails the predicate whenever it is used with a bound argument.

```
| ?- dept('NULL'(null2745),_,_).  → fails always.
```

Query Results:

When returning NULL's as field values, the interface returns `NULL/1` function with a unique integer argument serving as a skolem constant.

Notice that the above guarantees the expected semantics for the join statements. In the following example, even if `Deptno` is NULL for some rows in `emp` or `dept` tables, the query still evaluates the join successfully.

```
| ?- emp(Ename,_,_,_,Deptno),dept(Deptno,Dname,Loc)..
```

Inserts:

To insert rows with NULL values you can use `Field = 'NULL'(_)` or `Field = 'NULL'(null2346)`. For example:

```
| ?- emp_ins('NULL'(_), ...).  → inserts a NULL value for ENAME

| ?- emp_ins('NULL'('bound'), ...)  → inserts a NULL value for ENAME.
```

Deletes:

To delete rows with NULL values at any particular `FIELD` use `Field = 'NULL'(_)`, `'NULL'/1` with a free argument. When `'NULL'/1` 's argument is bound it fails the delete predicate always. For example:

```
| ?- emp_del('NULL'(_), ..).  → adds ENAME IS NULL to the generated SQL statement

| ?- emp_del('NULL'('bound'), ...).  → fails always
```

The reason for the above protocol is to preserve the semantics of deletes, when some free arguments of a delete predicate get bound by some preceding predicates. For example in the following clause, the semantics is preserved even if the `Deptno` field is NULL for some rows.

```
| ?- emp(_,_,_,_,Deptno), dept_del(Deptno).
```

7.3.8 Data dictionary

The following utility predicates access the data dictionary. Users of Quintus Prolog may note that these predicates are all PRODBI compatible. The following predicates print out the indicated information:

db_show_schema(accessible) Shows all accessible table names for the user. This list can be long!

db_show_schema(user) Shows just those tables that belongs to you.

db_show_schema(tuples('TABLE')) Shows the contents of the base table named `'TABLE'`.

db_show_schema(arity('TABLE')) The number of fields in the table `'TABLE'`.

db_show_schema(columns('TABLE')) The field names of a table.

For retrieving above information use:

- db_get_schema(accessible,List)
- db_get_schema(user,List)
- db_get_schema(tuples('TABLE'),List)
- db_get_schema(arity('TABLE'),List)
- db_get_schema(columns('TABLE'),List)

The results of above are returned in List as a list.

7.3.9 Other database operations:

db_create_table('TABLE_NAME','FIELDS') `FIELDS` is the field specification as in SQL.

```
eg. db_create_table('DEPT', 'DEPTNO NUMBER(2),
                             DNAME VARCHAR2(14),
                             LOC VARCHAR2(13)').
```

db_create_index('TABLE_NAME','INDEX_NAME', index(_,Fields)) `Fields` is the list of columns for which an index is requested. For example:

```
db_create_index('EMP', 'EMP_KEY', index(_,'DEPTNO, EMPNO')).
```

db_delete_table('TABLE_NAME') To delete a table named `'TABLE_NAME'`

db_delete_view('VIEW_NAME') To delete a view named `'VIEW_NAME'`

db_delete_index('INDEX_NAME') To delete an index named `'INDEX_NAME'`

These following predicates are the supported PRODBI syntax for deleting and inserting rows:

db_add_record('DEPT',[30,'SALES','CHICAGO']) arguments are a list composed of field values and the table name to insert the row.

delete_record('DEPT', [40,_,_]) to delete rows from `'DEPT'` matching the list of values mentioned in second argument.

For other SQL statements use `db_sql/1` with the SQL statement as the first argument. For example:

```
db_sql('grant connect to fred identified by bloggs')).
```

7.3.10 Interface Flags:

If you wish to see the SQL query generated by the interface use the predicate `db_flag/3`. The first parameter indicates the function you wish to change. The second argument is the old value, and the third argument specifies the new value. For example:

```
| ?- db_flag(show_query, Old, on).

     Old = off
```

SQL statements will now be displayed for all your queries (the default). To turn it off use `db_flag(show_query,on, off)`.

To enable you to control the error behavior of either the interface or Oracle database use `db_flag/3` with fail_on_error as first argument. For example:

`| ?- db_flag(fail_on_error, on, off)` Gives all the error control to you, (default), hence all requests to Oracle returns true. You have to check each action of yours and take the responsibility for your actions. (See 7.3.12)

`| ?- db_flag(fail_on_error, off, on)` Interface fails whenever something goes wrong.

7.3.11 Transaction management

Normally any changes to the database will not be committed until the user disconnects from the database. In order to provide the user with some control over this process, db_transaction/1 is provided.

db_transaction(commit) Commits all transactions up to this point.

db_transaction(rollback) Rolls back all transactions since the last commit.

Other services provided by Oracle such that `SET TRANSACTION` can be effected by using `db_sql/1`.

Note that depending on Oracle's MODE of operation some or all data manipulation statements may execute a commit statement implicitly.

7.3.12 SQLCA interface

You can use `db_SQLCA/2` predicate to access the `SQLCA` for error reporting or other services.

db_SQLCA(Comm, Res) Where Comm is any one of the below and Res is the result from Oracle.

- SQLCODE: The most recent error code
- SQLERRML: Length of the most recent error msg
- SQLERRMC: The error msg

```
eg. | ?- db_SQLCA('SQLERRD'(2), Rows).
```

returns in Rows number of rows processed by the most recent statement.

For SQLCAID, SQLCABC,SQLERRP, 'SQLERRD'(0) to 'SQLERRD'(5), 'SQLWARN'(0), to 'SQLWARN'(5),'SQLEXT' see `ORACLE's C PRECOMPILER` user's manual.

7.3.13 Datalog

You can write recursive Datalog queries with exactly the same semantics as in XSB using imported database predicates or database rules. For example assuming `db_parent/2` is an imported database predicate, the following recursive query computes its transitive closure.

```
:- table(ancestor/2).
ancestor(X,Y) :- db_parent(X,Y).
ancestor(X,Z) :- ancestor(X,Y), db_parent(Y,Z).
```

7.3.14 Guidelines for application developers

1. Try to group your database predicates and use the view level interface to generate efficient SQL queries.
2. Avoid cuts over cursors since they leave cursors open and can cause a leak of cursors.
3. Whenever you send a query get all the results sent by the Oracle by backtracking to avoid cursor leaks. This interface automatically closes a cursor only after you retrieve the last row from the active set.
4. Try to use tabled database predicates for cashing database tables.

7.4 Demo

A file demonstrating most of the examples introduced here is included with this installation in the `examples` directory. Load the package and call the goal go/2 to start the demo which is self documenting. Do not forget to load `ora_call.P` first.

```
| ?- [ora_call].
| ?- [ora_demo].
[ora_demo loaded]

yes
| ?- go(user, passwd).
```

where user is your account name, and passwd is your passwd.

7.5 Limitations

The default limit on open cursors per session in most Oracle installations is 50, which is also the default limit for our interface. There is also a limit imposed by the XSB interface of 100 cursors, which can be changed upon request [1]. If your Oracle installation allows more than 50 cursors but less then 100 then change the line

```
#define MAX_OPEN_CURSORS 20
```

in XSB_DIR/emu/orastuff.pc to your new value, and uncomment enough many cases to match the above number of cursors plus one, in the switch statements. Currently this number is 21. Then re-build the system. In XSB_DIR/emu/orastuff.pc we provide code for up to 100 cursors. The last 80 of these cursors are currently commented out.

7.6 Error msgs

ERR - DB: Connection failed For some reason you can not connect to Oracle.

- Diagnosis: Try to see if you can run sqlplus. If not ask your Oracle admin about it.

ERR - DB: Parse error The SQL statement generated by the Interface or the first argument to `db_sql/1` or `db_sql_select/2` can not be parsed by the Oracle. The character number is the location of the error.

- Diagnosis: Check your SQL statement. If our interface generated the erroneous statement please contact us at `xsb-contact@cs.sunysb.edu`.

ERR - DB: No more cursors left Interface run out of non-active cursors either because of a leak (See 7.3.14) or you have more then MAX_OPEN_CURSORS concurrently open searches.

- Diagnosis: System fails always with this error. db_transaction(rollback) or db_transaction(commit) should resolve this by freeing all cursors. Please contact us for more help since this error is fatal for your application.

ERR - DB: FETCH failed Normally you should never get this error if the interface running properly.

- Diagnosis: Please contact us at xsb-contact@cs.sunysb.edu

7.7 Future work

We plan to write a precompiler to detect base conjunctions (a sequence of database predicates and arithmetic comparison predicates) to build larger more restrictive base conjuncts by classical methods of rule composition, predicate exchange etc. and then employ the view level interface to generate more efficient queries and programs. Also we want to explore the use of tabling for caching of data and queries for optimization.

[1] e-mail xsb-contact@cs.sunysb.edu

Chapter 8

XSB-ODBC Interface

By Baoqiu Cui and Lily Dong

8.1 Introduction

The XSB-ODBC interface is the PC platform counterpart to XSB-Oracle interface on UNIX systems. It allows XSB users to access data in any ODBC compliant database management system (DBMS). Using this uniform interface, information in different DBMS's can be accessed as though it existed as Prolog facts. Similar to its counterpart on UNIX platforms, XSB-ODBC interface provides users with two levels of interaction: a *relation level* and a *view level*. The former offers a tuple-at-a-time retrieval of information from ODBC data sources while the latter can translate an entire Prolog clause into a single SQL query, including joins and aggregate operations, which gives XSB users all the features of Prolog as a query language such as intentional database specification, recursion etc. A listing of the features that XSB-ODBC interface provides is as follows:

- Concurrent access from multiple XSB processes to a single DBMS
- Full data access and cursor transparency including support for
  - Full data recursion through XSB's tabling mechanism
  - Runtime type checking
  - Automatic handling of NULL values for insertion, deletion and querying
  - Partial recovery for cursor losses due to cuts
- Full access to data source including
  - Transaction support
  - Cursor reuse for cached SQL statements with bind variables (by avoiding re-parsing and re-declaring).
  - Caching compiler generated SQL statements with bind variables and efficient cursor management for cached statements

- A powerful Prolog / SQL compiler based on [4].
- Full source code availability
- Independence from database schema by employing a *relation level*
- Performance as SQL by employing a *view level*
- No mode specification is required for optimized view compilation

We use the `Hospital` database as our example to illustrate the usage of XSB-ODBC interface in this manual. We assume the basic knowledge of Microsoft ODBC interface and its ODBC administrator throughout the text. Please refer to "Inside Windows^{TM} 95" for information on this topic.

8.2 Using the Interface

The XSB-ODBC module has to be loaded before the interface can be used. To load it, type in "`[odbc_call].`" at the XSB prompt.

8.2.1 Connecting to and Disconnecting from Data Sources

Assuming that the data source to be connected to is available, i.e. it has an entry in `ODBC.INI` file which can be checked by running Microsoft ODBC Administrator, it can be connected to in the following way:

```
| ?- odbc_open(data_source_name, username, passwd).
```

If the connection is successful, the system will give a positive response of `yes`. This step is necessary before anything can be done with the data sources since it gives XSB the opportunity to initialize system resources for the session.

To close the current session use:

```
| ?- odbc_close.
```

and XSB will give all the resources it allocated for this session back to the system.

8.2.2 Accessing Tables in Data Sources

There are several ways that can be used to extract information from or modify a table in a data source. Users can access a table using the relation level interface or view level interface which XSB provides or they can have their SQL statements executed directly without having XSB process it. In general, uses are required to firstly use `odbc_import/2` to give XSB the information about columns in the table of interest, expect for the cases such as direct execution of SQL statements and data dictionary operations, etc. and XSB will automatically assert data dictionary information(some other ODBC interface predicates can cause XSB to do this too). The Prolog predicate `listing/2` can be used (if it's available) to see the asserted data dictionary information at any time.

The syntax of `odbc_import/2` is as follows:

```
| ?- odbc_import('TableName'('FIELD1', 'FIELD2', ..., 'FIELDn'),
                              'PredicateName').
```

where `'TableName'` is the name of the table that is desired for accessing and `'PredicateName'` is the name of the predicate for future table operations from XSB. `'FIELD1'`, `'FIELD2'`, ... , `'FIELDn'` are the exact attribute names(case sensitive) as defined in the table schema. The chosen columns define the view and the order of arguments for the database predicate `'PredicateName'`.

For example, to create a link to the `Test` table through the `'test'` predicate:

```
| ?- odbc_import('Test'('TId','TName','Length','Price'),test).

yes
```

8.2.3 Using the Relation Level Interface

Once the links between tables and predicates have been successfully established, information can then be extracted from these tables using the corresponding predicates. Continuing from the above example, now rows from the table `Test` can be obtained:

```
| ?- test(TId, TName, L, P).

TId = t001
TName = X-Ray
L = 5
P = 100
```

Backtracking can then be used to retrieve the next row of the table `Test`.

Records with particular field values may be selected in the same way as in Prolog; no mode specification for database predicates is required. For example:

```
| ?- test(TId, 'X-Ray', L, P).
```

generates the query:

```
SELECT rel1.TId, rel1.TName, rel1.Length, rel1.Price
```

```
FROM Test rel1
WHERE rel1.TName = ?
;
```

and

```
| ?- test('NULL'(_), 'X-Ray', L, P).
```

generates: (See Section 7.3.7)

```
SELECT NULL , rel1.TName, rel1.Length, rel1.Price
FROM Test rel1
WHERE rel1.TId IS NULL AND rel1.TName = ?
;
```

During the execution of this query the bind variable `?` will be bound to `'X-Ray'`.

Note that if a field includes a quote (′) then this should be represented by using two quotes.

Also as a courtesy to Quintus Prolog users we have provided compatibility support for some PRODBI predicates which access tables at a relational level.

```
i) | ?- odbc_attach(PredicateName, table(TableName)).
```

eg. invoke

```
| ?- odbc_attach(test2, table('Test')).
```

and then execute

```
| ?- test2(TId, TName, L, P).
```

to retrieve the rows.

```
ii) | ?- odbc_record('Test', R).

    R = [t001, X-Ray, 5, 100];

    R = ...
```

You can use `odbc_record/2` to treat the whole database row as a single list structure.

8.2.4 The View Level Interface

The view level interface can be used for the definition of rules whose bodies includes only imported database predicates (by using the relation level interface) described above and aggregate predicates (defined below). When they are invoked, rules are translated into complex database queries, which are then executed taking advantage of the query processing ability of the DBMS's.

One can use the view level interface through the predicate `odbc_query/2`:

```
| ?- odbc_query('RuleName'(ARG1, ..., ARGn), DatabaseGoal).
```

All arguments are standard Prolog terms. `ARG1`, `ARG2`, ..., `ARGn` defines the attributes to be retrieved from the database, while `DatabaseGoal` defines the selection restrictions and join conditions.

The compiler is a simple extension of [4] which generates SQL queries with bind variables and handles NULL values as described in Section 7.3.7. It allows negation, the expression of arithmetic functions, and higher-order constructs such as grouping, sorting, and aggregate functions.

Database goals are translated according to the following rules from [4]:

- Disjunctive goals translate to distinct SQL queries connected through the UNION operator.
- Goal conjunctions translate to joins.
- Negated goals translate to negated EXISTS subqueries.
- Variables with single occurrences in the body are not translated.
- Free variables translate to grouping attributes.
- Shared variables in goals translate to equi-join conditions.
- Constants translate to equality comparisons of an attribute and the constant value.
- Nulls are translated to `IS NULL` conditions.

For more examples and implementation details see [4].

In the following, we show the definition of a simple join view between the two database predicates *Room* and *Floor*.

Assuming the declarations:

```
| ?- odbc_import('Room'('RoomNo','CostPerDay','Capacity','FId'),room).

| ?- odbc_import('Floor'('FId','','FName'),floor).
```

use

```
| ?- odbc_query(rule1(RoomNo,FName),
                  (room(RoomNo,_,_,FId),floor(FId,_,FName))).
yes

| ?- rule1(RoomNo,FloorName).
```

Prolog/SQL compiler generates the SQL statement:

```
SELECT rel1.RoomNo , rel2.FName FROM Room rel1 , Floor rel2
WHERE rel2.FId = rel1.FId;

RoomNo = 101
FloorName = First Floor
```

Backtracking can then be used to retrieve the next row of the view.

```
| ?- rule1('101','NULL'(_)).
```

generates the SQL statement:

```
SELECT rel1.RoomNo, NULL
FROM Room rel1 , Floor rel2
WHERE rel1.RoomId = ? AND rel2.FId = rel1.FId AND rel2.FName IS NULL;
```

The view interface also supports aggregate functions predicates such as sum, avg, count, min and max. For example

```
| ?- odbc_import('Doctor'('DId', 'FId', 'DName','PhoneNo','ChargePerMin'),doctor).

yes
| ?- odbc_query(avgchargepermin(X),
                (X is avg(ChargePerMin, A1 ^ A2 ^ A3 ^ A4 ^
                          doctor(A1,A2, A3,A4,ChargePerMin)))).

yes
| ?- avgchargepermin(X).

SELECT AVG(rel1.ChargePerMin)
FROM doctor rel1;

X = 1.64
```

```
yes
```

A more complicated example:

```
| ?- odbc_query(nonsense(A,B,C,D,E),
                (doctor(A, B, C, D, E),
                 not floor('First Floor', B),
                 not (A = 'd001'),
                 E > avg(ChargePerMin, A1 ^ A2 ^ A3 ^ A4 ^
                         (doctor(A1, A2, A3, A4, ChargePerMin))))).

| ?- nonsense(A,'4',C,D,E).

SELECT rel1.DId , rel1.FId , rel1.DName , rel1.PhoneNo , rel1.ChargePerMin
FROM doctor rel1
WHERE rel1.FId = ? AND NOT EXISTS
(SELECT *
FROM Floor rel2
WHERE rel2.FName = 'First Floor' and rel2.FId = rel1.FId
) AND rel1.Did <> 'd001' AND rel1.ChargePerMin >
(SELECT AVG(rel3.ChargePerMin)
FROM Doctor rel3
);


A = d004
C = Tom Wilson
D = 516-252-100
E = 2.5
```

All database rules defined by odbc_query can be queried with any mode.

Note that at each call to a database relation or rule, the communication takes place through bind variables. The corresponding restrictive SQL query is generated, and if this is the first call with that adornment, it is cached. A second call with same adornment would try to use the same database cursor if still available, without parsing the respective SQL statement. Otherwise, it would find an unused cursor and retrieve the results. In this way efficient access methods for relations and database rules can be maintained throughout the session.

Also the relation level interface can be used to define and access simple project views of single tables. For example:

```
| ?- odbc_import('Room'('RoomNo','Capacity'),roomview).
```

defines `roomview/2`.

8.2.5 Insertions and Deletions of Rows

Insertion and deletion operations can also be performed on an imported table. The two predicates to accomplish these operations are `odbc_insert/2` and `odbc_delete/2`. The syntax of `odbc_insert/2` is as follows: the first argument is the declared database predicate for insertions and the second argument is some imported data source relation. The second argument can be declared with some of its arguments bound to constants. For example after `Room` is imported through `odbc_import`:

```
|?- odbc_import('Room'('RoomNo','CostPerDay','Capacity','FId'), room).
yes
```

Now we can do

```
| ?- odbc_insert(room_ins(A1,A2,A3),(room(A1,A2,A3,'3'))).

yes
| ?- room_ins('306','NULL'(_),2).

yes
```

This will insert the row: ('306',NULL, 2,'3') into the table `Room`. Note that any call to `room_ins/7` should have all its arguments bound.

See Section 7.3.7) for information about NULL value handling.

The first argument of `odbc_delete/2` predicate is the declared delete predicate and the second argument is the imported data source relation with the condition for requested deletes, if any. The condition is limited to simple comparisons. For example assuming `Room/3` has been imported as above:

```
| ?- odbc_delete(room_del(A), (room('306',A,B,C), A > 2)).

yes
```

After this declaration you can use:

```
| ?- room_del(3).
```

to generate the SQL statement:

```
DELETE From Room rel1
WHERE rel1.RoomNo = '306' AND rel1.CostPerDay = ? AND ? > 2
;
```

Note that you have to commit your inserts or deletes to tables to make them permanent. (See section 8.2.9).

8.2.6 Direct Execution of SQL statements

It is also possible to execute SQL statements directly. `odbc_sql_select/2` and `odbc_sql/1` predicates provide this feature. The former takes an SQL query string as its input and returns a list of field values. For example:

```
| ?- odbc_sql_select('SELECT * FROM Test', R).

R = [t001, X-Ray, 5, 100];

...
```

The latter, `odbc_sql/1`, can be used for any other non-query SQL statement request:

```
| ?- odbc_sql('Create Table MyTable( Column1 DataType1, Column2 DataType2)').

yes
```

8.2.7 Access to Data Dictionaries

The following utility predicates provide users the tools to access data dictionaries. Users of Quintus Prolog may note that these predicates are all PRODBI compatible. A brief description of these predicates is as follows:

odbc_show_schema(accessible) Shows all accessible table names for the user. This list can be long!

odbc_show_schema(user) Shows just those tables that belongs to user.

odbc_show_schema(tuples('Table')) Shows the contents of the base table named `'Table'`.

odbc_show_schema(arity('Table')) The number of fields in the table `'Table'`.

odbc_show_schema(columns('Table')) The field names of a table.

For retrieving above information use:

- odbc_get_schema(accessible,List)
- odbc_get_schema(user,List)

- odbc_get_schema(arity('Table'),List)
- odbc_get_schema(columns('Table'),List)

The results of above are returned in List as a list.

8.2.8 Other Database Operations

odbc_create_table('TableName','FIELDs') `FIELDS` is the field specification as in SQL.

```
eg. odbc_create_table('MyTable', 'Col1 NUMBER,
                                  Col2 TEXT(50),
                                  Col3 TEXT(13)').
```

odbc_create_index('TableName','IndexName', index(_,Fields)) `Fields` is the list of columns for which an index is requested. For example:

```
odbc_create_index('Doctor', 'DocKey', index(_,'DId')).
```

odbc_delete_table('TableName') To delete a table named `'TableName'`

odbc_delete_view('ViewName') To delete a view named `'ViewName'`

odbc_delete_index('IndexName') To delete an index named `'IndexName'`

These following predicates are the supported PRODBI syntax for deleting and inserting rows:

odbc_add_record('Floor',['Seventh Floor','7']) arguments are a list composed of field values and the table name to insert the row.

odbc_delete_record('Floor', ['Seventh Floor',_]) to delete rows from `'Floor'` matching the list of values mentioned in second argument.

For other SQL statements use `odbc_sql/1` with the SQL statement as the first argument. For example:

```
odbc_sql('grant connect to fred identified by bloggs')).
```

8.2.9 Transaction Management

Depending on how the transaction options are set in ODBC.INI for data sources, any changes to the data source tables may not be committed(changes become permanent) until the user expicitly issues a commit statement. The predicate `odbc_transaction/1` is provided in this sense.

odbc_transaction(commit) Commits all transactions up to this point.

odbc_transaction(rollback) Rolls back all transactions(discard the changes made) since the last commit.

8.2.10 Handling NULL Values

Null value is handled in the same way as that of XSB Oracle interface. Please refer to Section 7.3.7) for details.

8.2.11 Interface Flags

Users are given the option to monitor the SQL queries generated by the interface and their execution status by using the predicate `db_flag/3`. The first parameter indicates the function to be changed. The second argument is the old value, and the third argument specifies the new value. For example:

```
| ?- odbc_flag(show_query, Old, on).

     Old = off
```

SQL statements will now be displayed for all SQL queries (the default). To turn it off use `odbc_flag(show_query,on, off)`. The default value of `show_query` is on.

To control the error behavior of either the interface or data sources use `odbc_flag/3` with fail_on_error as first argument. For example:

`| ?- odbc_flag(fail_on_error, on, off)` Gives all the error control to users, hence all requests to data sources return true. It's users' responsibility to check each of their actions and do error handling.

`| ?- odbc_flag(fail_on_error, off, on)` Interface fails whenever error occurs.

The default value of fail_on_error is on.

8.2.12 Datalog

Users can write recursive Datalog queries with exactly the same semantics as in XSB using imported database predicates or database rules. For example assuming `odbc_parent/2` is an imported database predicate, the following recursive query computes its transitive closure.

```
:- table(ancestor/2).
ancestor(X,Y) :- odbc_parent(X,Y).
ancestor(X,Z) :- ancestor(X,Y), odbc_parent(Y,Z).
```

8.3 Limitation and Guidelines for Application Developers

Since XSB-ODBC interface is a simulation of XSB-ORACLE interface on UNIX platform, it inherits all limitations of the XSB-ORACLE interface, i.e. limited number of usable cursors, cursor leaking when using cuts and etc. Hence the guidelines for XSB-ORACLE interface application developers are also for XSB-ODBC interface application developers. Please refer to Sections 7.3 and 7.5 for datails.

8.4 Error messages

ERR - DB: Connection failed For some reason the attempt to connect to data source failed.

- Diagnosis: Try to see if the data source has been registered with Microsoft ODBC Administrator, the username and password are correct and MAXCURSORNUM is not set to a very large number.

ERR - DB: Parse error The SQL statement generated by the Interface or the first argument to `odbc_sql/1` or `odbc_sql_select/2` can not be parsed by the data source driver.

- Diagnosis: Check the SQL statement. If our interface generated the erroneous statement please contact us at `xsb-contact@cs.sunysb.edu`.

ERR - DB: No more cursors left Interface run out of non-active cursors either because of a leak (See Section 7.3) or no more free cursors left.

- Diagnosis: System fails always with this error. odbc_transaction(rollback) or odbc_transaction(commit) should resolve this by freeing all cursors.

ERR - DB: FETCH failed Normally this error should not occur if the interface running properly.

- Diagnosis: Please contact us at xsb-contact@cs.sunysb.edu

Bibliography

[1] J. Alferes, C. Damasio, and L. Pereira. A logic programming system for non-monotonic reasoning. *Journal of Automated Reasoning*, 1995.

[2] J. Alferes and L. M. Pereira. *Reasoning with Logic Programming*, volume 1111. Springer-Verlag LNAI, 1996.

[3] P. Brisset, et al. *ECL^iPS^e 4.0 User Manual.* IC-Parc at Imperial College, London, July 1998.

[4] Christoph Draxler. Prolog to SQL compiler, Version 1.0. Technical report, CIS Centre for Information and Speech Processing Ludwig-Maximilians-University, Munich, 1992.

[5] M. Kifer and V.S. Subrahmanian. Theory of generalized annotated logic programming and its applications. *J. Logic Programming*, 12(4):335–368, 1992.

[6] The Intelligent Systems Laboratory. *SICStus Prolog User's Manual Version 3.7.1.* Swedish Institute of Computer Science, October 1998.

[7] T. Swift. Tabling for non-monotonic programming. *Annals of Mathematics and Artificial Intelligence*, 1999. To Appear. Available at http://cs.sunysb.edu/ tswift.