ALS Prolog Development Tools

Applied Logic Systems, Inc.
PO Box 400175
Cambridge, MA 02140 USA
Email: info@als.com
WWW: http://www.als.com

1 TTY Development Environment

The TTY Development interface for ALS Prolog is similar to the original DEC-10 system constructed in Edinburgh.

1.0.1 Starting up ALS Prolog

Starting up ALS Prolog varies from system to system. Under some systems such as ordinary Unix shells or DOS, one starts ALS Prolog by typing a shell command such as

```
C:> alspro
wizard% alspro
```

or

```
$ alspro
```

or

```
C:\> alspro
```

On others, such as the Macintosh, one clicks on an icon, which opens a windows.

The various versions usually show a startup banner such as the followingt:

```
ALS-Prolog Version 1.7
  Copyright (c) 1987-95 Applied Logic Systems
?-
```

1.0.2 Exiting Prolog

There are several ways to exit ALS Prolog. The normal way to exit is to submit the goal

```
?- halt.
```

from the Prolog shell or from a Prolog program. The second way to exit can only be accomplished from the top level of the Prolog shell. There, you can type a character (such as **Control-D** on Unix) or sequence. of characters (such as **Control-Z** followed by a return on DOS. or # at the beginning of a line on the Mac) which signifies closing the default input stream. See `halt/0` in the reference section. Finally, under the GUI or windowed interfaces, one can select an Exit menu button.

1.1 Asking Prolog to Do Something

The most common way of telling Prolog what you want it to do is to submit a ***goal***. If you are in the Prolog shell, and `?-` is the prompt, then anything you type is considered to be a goal. A goal must end with a period, ‘ . ’, followed by a white space character (carriage return, blank, etc.). This is called a ***full stop***. Goals must be correct Prolog terms. See Chapter 1 of the User Guide for a discussion concerning the construction of correct Prolog terms. The following is an example of a goal submitted from the ALS Prolog shell:

```
?- length([a,b,c],Answer).
Answer = 3
```

The goal issued was `length([a,b,c],Answer)`. The system responded by showing that the variable `Answer` was instantiated with the number `3`, and that the goal succeeded. The Prolog user had to press the `return` key after the

```
Answer = 3
```

was displayed, in order to get the `yes.` printed. Not all goals succeed, of course. For example the following goal fails:

```
?- length([a,b,c], 4).
no.
```

This goal fails because the list `[a,b,c]` is not four elements long.

After submitting a goal in the Prolog shell, if any variables have become instantiated, their values will be displayed to you as in the first example above with `length/2`. When this occurs, the shell waits for you to type either a ‘`;`’ followed by a `return`, or just a `return`. The two choices have the following effect:

- `;` — Forces the goal to fail, thus causing backtracking and retrying of the goal.
- `return` — Causes the goal to succeed.

If a recursive data structure is created, such as is done by the following goal

```
        ?- X = f(X).
```

the part of the Prolog shell which prints answers will go into a loop, and continue writing the data structure onto your screen until the structure gets too deep. When this happens, an ellipsis (`...`) is eventually displayed as shown below:

```
X = f(f(f(f(f(f(f(...)))))))
yes.
```

The actual depth of the structure shown by the answer printer is much deeper than is shown above.

Goals can also be submitted from within a file. There are two forms of submitting goals from files:

- Commands
- Queries

Commands are specified by the `:-` prefix, while queries are specified by the `?-` prefix. The only difference between the two is that queries write the message '`yes.`' to the screen if the goal succeeds, and '`no.`' if the goal fails, while commands do not write any result on the screen.

1.2 How to Load Prolog Programs

There are basically two ways of loading Prolog programs into the ALS Prolog system:

1. When you start `alspro` from the command line you can give a list of files for the Prolog system to load as programs.
2. If you want to load Prolog predicates from inside a program, or from the Prolog shell, you can use the consult/1 builtin in the following manner:

```
?- consult(File).
```

where `File` is *instantiated* to a Prolog program's file name. Alternatively, one can use

```
?- reconsult(File).
```

Finally, one can use the top-level list-as-reconsult construct:

```
?- [File1, File2,...].
```

For more information on how to use the consulting predicates, see Section 9.2.1 (*Consulting Program Files*) in the User Guide.

1.3 Stopping a Running Prolog Program

If you wish to interrupt a running ALS Prolog program, simply press the interrupt key (e.g., the Control-C key on Unix, the Control-Break key on DOS, the Apple-Period key combination on the Mac) for your system. You will be returned to the top level of the ALS Prolog shell.

1.4 How ALS Prolog Finds Prolog Files

When a request that a file be loaded is made (such as reconsult(myfile)), ALS Prolog looks for the file in the following manner:

1.4.1 Complex Pathnames

If the file is not a simple pathname, that is, any file with a ‘file-slash’ character (‘/ ’) in it (on Unix or DOS), or the ’file-color’ character (":") (on the Mac), the file will be loaded as specified. Some examples are:

```
?- consult('/usr/gorilla/banana.pro').
Consulting /usr/gorilla/banana.pro...
.../usr/gorilla/banana.pro consulted.
yes.

On the Mac:
?- consult('Usr:gorilla:banana.pro').
Consulting Usr:gorilla:banana.pro...
...Usr:gorilla:banana.pro consulted.
yes.
```

1.4.2 Simple Pathnames

If the file name is a simple pathname, then the file will be searched for in several directories, as follows:

1. The current directory is searched first;
2. Next, the directories listed on the ALSPATH environment variable (if it is defined) are searched in order of their left-to-right appearance;
3. Next, the directory named in the ALSDIR environment variable (if it is defined) is searched;
4. Finally, the directory in which the current image resides is searched.

If the specified file is located in any of the indicated directories, it will be loaded into ALS Prolog, and no further search is made for this file. (Thus, if multiple versions of a file exist in some of the indicated directories, only the first will be loaded.) The following example uses the Unix C shell to illustrate the use of the ALS Prolog pathlist. We assume for this example that ALS Prolog was installed in */usr/prolog*. Then the file *dc.pro* (which is one of ALS Prolog examples) will be contained in the directory */usr/prolog/alsdir/examples*. The following illustrates the use of ALSPATH:

```
wizard%setenv ALSPATH /usr/prolog/alsdir/examples:/
  usr/gorilla
wizard% alspro
ALS-Prolog Version 1.0
Copyright (c) 1987-90 Applied Logic Systems, Inc.

?- consult(dc).
Consulting dc...
.../usr/prolog/alsdir/examples/dc consulted.
yes.
```

The method used to implement the use of the ALSPATH pathlist actually provides greater flexibility than the foregoing discussion indicates. When ALS Prolog is initialized, it reads the Unix ALSPATH environment variable (if it is defined), together with the ALSDIR environment variable, and uses them to create a set of facts in the builtins module. These facts all have the following form:

```
searchdir(".../.../").
```

The expression within the quotes can be any meaningful Unix path describing a directory (hence the terminal ('/'). At startup time, the assertions correspond to the expressions found on the `ALSPATH` pathlist. Thus, the facts corresponding to the example above would be:

```
searchdir("/usr/prolog/alsdir/examples/").
searchdir("/usr/gorilla/").
searchdir("/usr/prolog/alsdir/").
searchdir("/usr/prolog/").
```

Note that the current directory (or '.' to represent it) does not appear among these facts; however, it is always automatically searched first. Additional entries can be made to this collection of facts by using `assert/1` (or `asserta/1` or `assertz/1`). For example, the goal

```
:-builtins:asserta(searchdir("chimpanze/")).
```

would cause the subdirectory *chimpanze* of the current directory to be searched immediately after the current directory and before any other directories. And

```
:-builtins:asserta(searchdir("../widget/")).
```

would cause the sibling subdirectory *widget* of the current directory to be searched immediately after the current directory and before any other directories. Assertions such as these can be placed in the ALS Prolog startup file (described below) to customize search paths for particular directries. See the User Guide for more information concerning loading files.

1.5 Controlling the Search Path

If you want to be able to consult some of your files that are not in your current directory, and you don't want to use absolute pathnames, you can put the directories where those files reside on a path searchlist called `ALSPATH`. In additon, you can add directories using the comannd-line switch -S at start-up time (see Section 1.7 (*ALS Prolog Command Line Options*)). The following is an example use of the `ALSPATH` variable on Unix or DOS:

```
setenv ALSPATH /usr/eddie/programs:/usr/sue/src/
  prolog
```

If you want to also automatically search the ALS Prolog directory, you could use the following:

```
   setenv ALSPATH /usr/eddie/programs:/usr/sue/src/
     prolog:\
               /usr/prolog/alsdir
```

The definition of the `ALSPATH` variable can also be placed in your *.cshrc* startup file. Combining the examples above, your *.cshrc* startup file might include the following lines:

```
   setenv ALSPATH /usr/eddie/programs:/usr/sue/src/
     prolog:\
               /usr/prolog/alsdir
```

On DOS, this would look like:

```
   set ALSPATH /usr/eddie/programs:/usr/sue/src/
     prolog:\
               /usr/prolog/alsdir
```

Note that even though it is running under DOS, ALS Prolog utilizes Unix-style directory separators.

On the Macintosh, environment variables do not exist. However, one can still utilize the effects of the ALSPATH variable. As noted in Section 1.4, ALS Prolog uses the value of the ALSPATH variable to create and assert facts for the predicate builtins:searchdir/1. The predicate which processes this is called `builtins:ss_init_searchdir/1.` In fact, the reading and processing of ALSPATH, when it exists, is done as follows (in *blt_shl.pro*):

```
   ss_init_searchdir
     :-
     getenv('ALSPATH',ALSPATH),
     ss_init_searchdir(ALSPATH).
```

What really happens in the code is that `ss_init_searchdir`/1 takes apart the value it has obtained for ALSPATH, and produces a list of atoms representing the individual directories in the path. It then calls a subsidiary predicate, `ss_init_searchdir0`/1 which recurses down this list, asserting the fact

```
   builtins:searchdir(SDir)
```

for each atom `SDir` on the list. So on the Mac, there are two approaches. On the one hand, one can directly make assertions on builtins:searchdir/1 as above to set up the search path. Or, one can directly call `ss_init_searchdir0`/1 with an appropriate argument. So one of the animals examples from the last section would work like this:

```
   :-builtins:ss_init_searchdir0(
     [’usr:prolog:alsdir:examples’,
       ‘usr:gorilla’]).
```

Such a call can be placed in the Prolog startup file or in one of your source files to occur automatically, as descirbed in the next section.

1.6 Using the Prolog Startup File

When ALS Prolog starts up, it looks first in the current directory and then in your home directory for a file named either *.alspro* (on Unix or the Mac) or *alspro.pro* (on DOS). After the Prolog builtins are loaded the *.alspro* (or *alspro.pro*) file is consulted if it exists. The purpose of the Prolog startup file is to allow you to automatically load various predicates and files which you routinely use, and to carry out possible customizations of your environment such as the modifications to the standard search path described in the previous section.

1.7 ALS Prolog Command Line Options

There are a number of options that can be included on the operating system shell command line when starting ALS Prolog. The following is a list of the options:

-g The option `-g` followed by an arbitrary Prolog goal, instructs ALS Prolog to run the goal when it starts up as if it was the first goal typed to the Prolog shell after the system is started. The goal might have to be quoted depending on the rules of the operating system shell you are running in, and if the goal contains any of your shell’s special characters. You do not have to put a *full stop* after a goal, and you can submit multiple goals, provided there is no white space anywhere in the given goals. When the submitted goal finishes running (with success or failure), control is passed to the normal Prolog shell unless the `-b` command line option has also been used, in which case control returns to the operating system shell.

-b The option `-b` prevents the normal the Prolog shell from running. This means control will return to the operating system shell when all command line processing is complete, including processing of source files and execution of `-g` goals.

-q The option `-q` causes all standard system loading messages to be suppressed, including the banner. One of the uses of `-q` is to permit you to use ALS Prolog as a Unix filter. Note that this does not turn off prompts issued by the Prolog shell.

-v The option `-v` turns on verbose mode. This causes all system loading messages, including some which are normally suppressed, to be printed.

-gic The option `-gic` ("generated in current") causes the **.obp* files which are generated for consulted files to be created in the current working directory of the running image when the files are consulted.

-gis The option `-gis` causes the **.obp* files which are created for consulted files to be created in the same directory as the source file from which they were generated.

-giac The option `-giac` causes the **.obp* files which are created for consulted files to be stored in a subdirectory of the current working directory of the running image when the files are consulted; the subdirectory corresponds to the architecture of the running ALS Prolog image. Thus, for example, **.obp* files generated by a Solaris2.4 image will be stored in a subdirectory named *solaris2.4*, etc.

-gias The option `-gias` causes the **.obp* files which are created for consulted files to be stored in a subdirectory of the directory containing the source **.pro* files when the files are consulted; the subdirectory corresponds to the architecture of the running ALS Prolog image. Thus, for example, **.obp* files generated by a Solaris2.4 image will be stored in a subdirectory named *solaris2.4*, etc.

-w The option `-w` causes calls to non-existent predicates to print an error message. This is useful for debugging, but can be annoying otherwise.

-p The option `-p` is used by ALS Prolog to distinguish between command line switches intended for the system and those switches intended for an application (whether invoked with the `-g` command line switch or from the Prolog shell). The `-p` divides the command line into two portions: *All* switches to the left of the `-p` are interpreted as being for the ALS Prolog system, while *all* switches to the right of the `-p` are interpreted as being intended for a Prolog application. To make the latter available to Prolog applications, when ALS Prolog is initialized, a list `SWITCHES` of atoms and UIAs representing the items to the right of the `-p` is created, and a fact command_line(SWITCHES`)` is asserted in module `builtins`. For example, the command line

```
alspro -g my_appl -b applfile -p -k fast -s
initstate foofile
```

would result in the following fact being asserted in module `builtins`:

```
command_line(['-k',fast,'-s',initstate,foofile]).
```

This assertion is always made, even when `-p` is not used, in which case the argument of `command_line/1` is the empty list. It is important to note that `command_line/1` is *not* exported from module `builtins`, so that accesses to it from other modules must be prefixed with '`builtins:`' as in

```
...,builtins:command_line(Cmds),...
```

-s This switch must be followed by a space and a path to a directory. The path is added to the `searchdir/1` sequence. Multiple occurrences of -s with a path may occur on the command line; the associated paths are processed and added to the searchdir/1 facts in order corresponding to their left-to-right occurrence on the command line. All paths occurring with -s on the command line are added to the searchdir/1 facts before any paths obtained from the ALSPATH environment variable.

-A, -a This switch must be followed by a space and a Prolog Goal (enclosed in single quotes if necessary to defeat the OS shell) and is used to force one or more assertions, as follows:

If `Goal` is of the form `M:(H1, ...Hk),` then each of `H1, ..., Hk` is asserted in module `M`. Thus,

```
     alspro -A ‘ice_cream:(jerry, ben)’
```

would cause the two facts facts `jerry` and `ben` to be asserted in module `ice_cream`..

If `Goal` is of the form `M:H`, then `H` is asserted in module `M`.

If `Goal` is of the form `(H1, ...Hk)`, then each of `H1, ..., Hk` is asserted in module user. Thus,

```
     alspro -A ‘(jerry, ben)’
```

would cause the two facts facts `jerry` and `ben` to be asserted in module `user.`

Otherwise, `Goal` is asserted in module user.

Note that occurrences of the `-A (or -a)` switch must occur to the left of any occurrence of the `-p` switch. (This switch was designed for use in makefiles.)

-heap

The option `-heap` followed immediately by space and a number *w* sets the size of the ALS Prolog *heap* to

w *1024,

where *w* is the number of K bytes to allocate. Heap overflow will cause exit to the operating system.

-stack

The option `-stack` followed immediately by a space and a number *w* sets the size of the ALS Prolog *stack* to

w *1024,

where *w* is the number of K bytes to allocate. Stack overflow will cause exit to the operating system.

These two options were formerly only controlled by the use of an environment variable, `ALS_OPTIONS`. Now, either or both the command-line and environment variable method can be used. Use of one of the command-line options overrides use of the corresponding option with the environment variable.

The `ALS_OPTIONS` environment variable is used as follows. If w1 and w2 are similar to the value w described above for -h and -s, then:

Under Bourne shell, Korn shell, and Bash:

```
ALS_OPTIONS=stack_size:w1,heap_size:w2
export ALS_OPTIONS
```

Under csh:

```
setenv ALS_OPTIONS stack_size:w1,heap_size:w2
```

Under MS Windows:

```
ALS_OPTIONS=stack_size:w1,heap_size:w2
```

[Under MS Windows 95, such a line is placed in the AUTOEXEC.BAT file. Under Windows NT, one uses the Environment section of the System Properties control panel.]

2 Using the Four-Port Debugger

The ALS Prolog Debugger allows you to debug a faulty program with the standard four-port model of Prolog execution. You can use the debugger to find where your program is faulty by looking at how procedures are being called, what values they are returning, and where they fail.

The debugger is written in Prolog, so it can be consulted like any other program. If a debugger command is issued and the debugger is currently not loaded, it will be automatically consulted. The debugger is contained in the file *debugger.pro*, which resides in the ALS directory *alsdir*. (On the Macintosh, the debugger is contained in the file *debugger* which resides in the folder *Interfaces*.)

On the Macintosh and the original (real mode) ALS Professional and Student Prologs for the PC, the debugger is an interpretive debugger; i.e., the debugger program implements a complete interpreter for Prolog (in Prolog) which decompiles and interprets the (compiled) code which has been loaded. On all other versions of ALS Prolog, the debugger is a much more sophisticated program (written in ALS Prolog) that utilizes the interrupt facilities of ALS Prolog to directly debug the compiled (native) code produced by the ALS Prolog compiler. It does this without requiring you to set any special flags during compilation (loading).

2.1 The Four-Port Model

The four-port model of Prolog execution provides a conceptual point of view for analyzing the flow of control during execution of a Prolog program. Think of each procedure in your Prolog program as having a box around it with four ports for getting in and out of the procedure.

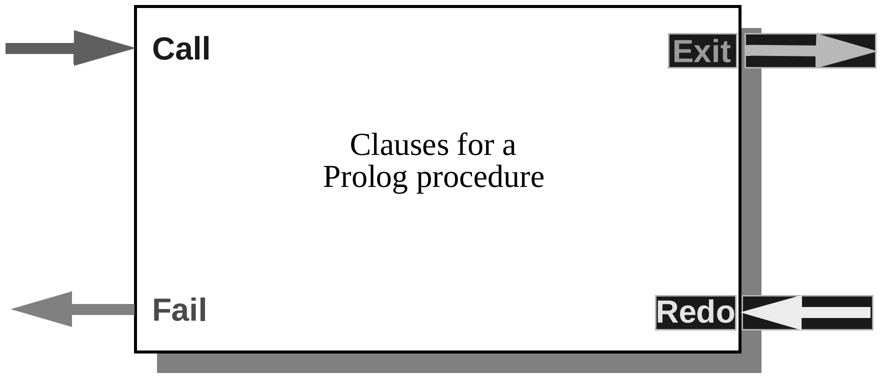


Figure 14. Generic Procedure Box.

The flow of program control can then be viewed as motion through one of the four ports. The ports are:

- `Call` is the entry point (in) to a procedure the first time it is called.
- `Exit` is the port (out) through which execution passes if the procedure succeeds.
- `Fail` is the port (out) through which execution passes if the procedure fails completely.
- `Redo (Retry)` is the port (in) through which execution passes when a later goal has failed and the procedure must try and find another solution, if possible.

Take, for example, the program

```
likes(john,Who) :-
  female(Who),
  likes(Who,wine).
likes(mary,wine).

female(susan).
female(mary).
```

Figure 15 shows model of the procedure `female/1`.

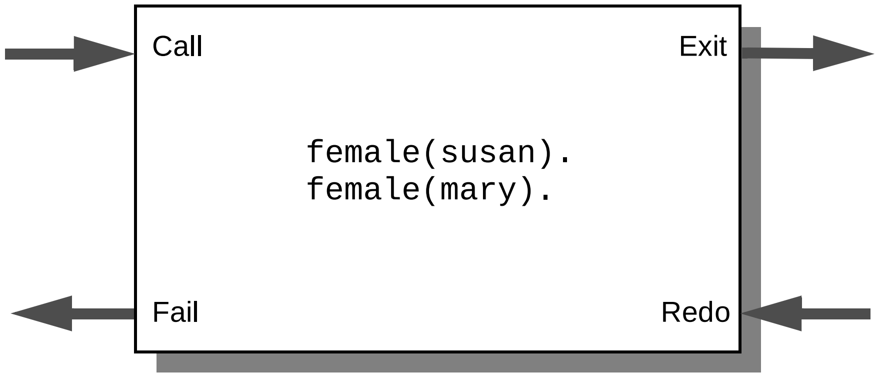


Figure 15. Procedure Box for `female/1.`

Suppose you make the query:

```
   ?- likes(john,Who).
```

The clause for `likes/2` is activated, and the debugger is ready to make the call to `female/1`. It enters the `female/1` procedure through the call port (see Figure 16 (a)) and picks up the first solution, binding `Who` to `susan`. Because the procedure succeeds, execution continues through the exit port of the procedure (as shown in Figure 16 (b)) and continues with the call `likes(susan,wine).`

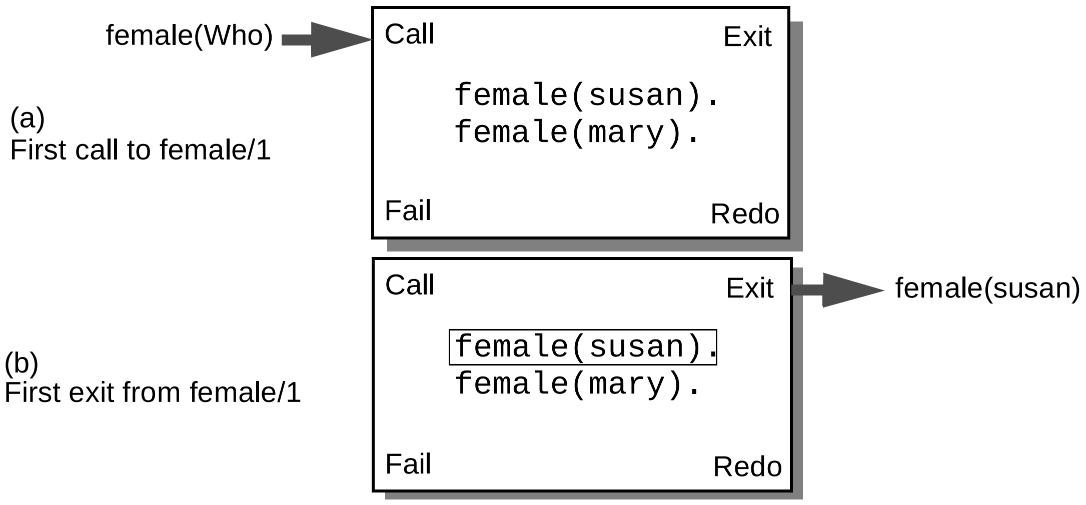


Figure 16. First call and exit tracing female/1.

The call `likes(susan,wine)` fails, because neither clause for `likes/2` matches with `likes(susan,wine)` in the first argument. Because of this failure, another solution to `female/1` is sought. The program re-enters the `female/1` procedure, this time through the redo port (Figure 17 (c)).

Entering a procedure through a redo port means that a solution to the procedure was not accepted in later parts of the program and another solution for the call is required. In the example here, after re-entering the procedure through the redo port, execution will first unbind `Who`, and then bind `Who` to `mary`, leaving the procedure by passing through the exit port, as shown in Figure 17 (d).

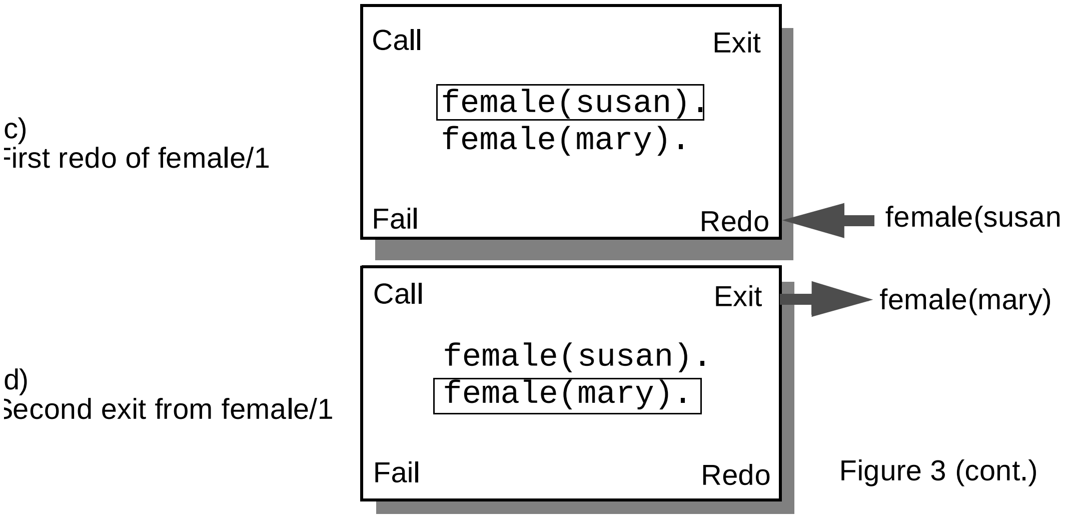


Figure 17. Redo and exit tracing female/1.

The call `likes(mary,wine)` succeeds, so the original goal succeeds:

```
?- likes(john,Who).
Who = mary
```

If you want Prolog to look for another answer, press ‘`;`’ (semi-colon) followed by `Return`. This will cause failure to occur, forcing the search for another solution. In this example, execution re-enters the procedure box for `female/1` through the redo port (See Figure Figure 18 (e)). However, there are no more solutions for `female/1`, so the procedure must fail. Execution then leaves the `female/1` box through the fail port, as shown in Figure 18 (f) . The failure of `female/1` causes the second clause of `likes/2` to be tried. Because the goal does not match the second clause of `likes/2`, the original goal, `likes(john,Who)`, now fails.

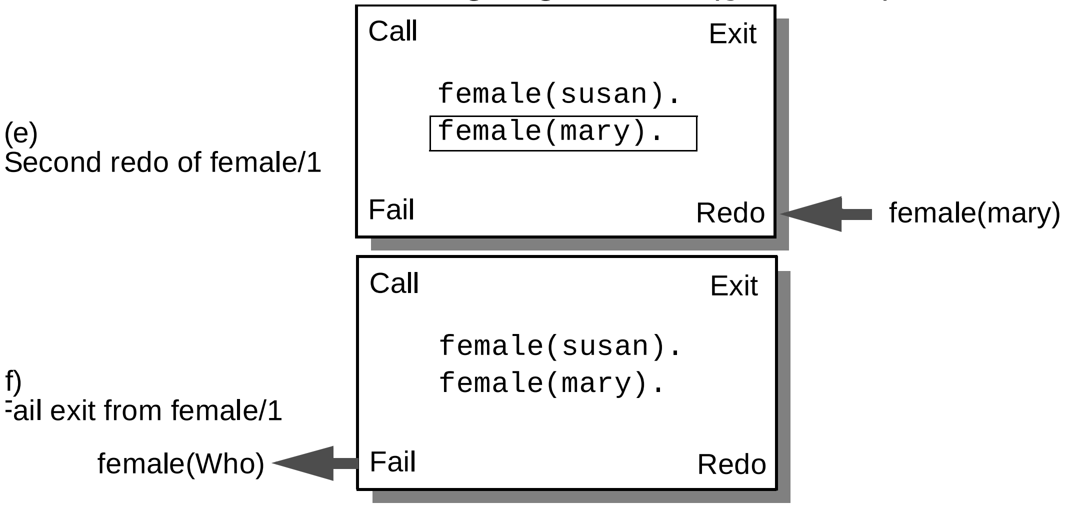


Figure 18. Redo and fail tracing female/1.

2.2 Creeping Along With the Debugger

The debugger helps you to find errors in your program by letting you look at the control flow of the program as well as showing you the variable bindings as the program goes through each of the ports. You can control how much information is printed by the command you give to the debugger when it stops at a port.

As an example, trace the execution of the goal `likes(john,Who)`. You can do this by using the debugger builtin trace/1:

```
   ?- trace likes(john,Who).
```

The debugger will then answer with

```
   (1) 1 call: likes(john,_93) ?
```

This means that the debugger is waiting at the entrance to the call port of the procedure `likes/2`. The current goal is printed for the call, as well as all the arguments to the call. The program text calls the `likes/2` procedure with the arguments `john` and `Who`. The debugger is only able to print out the internal names for variables, rather than the names found in the source code for the procedures in the program. In this case, the variable `Who` appears as `_93`. The number `93` is not important, since it is merely a place holder. The actual number assigned will vary depending on the prior state of the Prolog system.

The integer in parentheses is the number of the procedure box the debugger is currently examining. The next number (following the number in parentheses) is the level of the call. This number tells you the depth of the computation. In this example, the original goal `likes(john,Who)` runs at level 1, and all the subgoals in the clause run at level 2. If the call to female had any subgoals, those subgoals would take place at level 3.

You have a choice of how the debugger is going to continue examining the program. When you want to move slowly through the program, looking at everything there is to see, you use the *creep* command. To creep, you should respond with '`c`' followed by `return` at the `?` prompt of the debugger. `Return` alone will also make the debugger creep. However, for the sake of readability, the '`c`' will appear explicitly in all the examples.

```
   (1) 1 call: likes(john,_93) ? c
   (2) 2 call: female(_93) ?
```

The debugger is now at the call port for the procedure `female/1`. You can see by the last line that the program is calling `female/1` with one unbound variable and that this call is taking place at level 2 of the computation. To continue the computation, you can creep ahead:

```
   (2) 2 call: female(_93) ? c
   (2) 2 exit: female(susan) ?
```

After entering `female/1`, the program picks up the first solution and binds the variable `_93` to `susan`. After this, the program leaves the exit port for `female/1` and returns to the interior of the first clause for `likes/2`.

```
   (2) 2 exit: female(susan) ? c
   (3) 2 call: likes(susan,wine) ?
```

The debugger now enters `likes/2` through the call port. The number in parentheses has changed from `2` to `3` to show that this is a new procedure call. However, the call depth is still `2`.

```
(3) 2 call: likes(susan,wine) ? c
(3) 2 fail: likes(susan,wine) ?
```

Because there are no clauses in `likes/2` that match `likes(susan,wine)` the call to `likes/2` fails, and the debugger exits `likes/2` through the fail port.

```
(3) 2 fail: likes(susan,wine) ? c
(2) 2 redo: female(_93) ?
```

The debugger now re-enters `female/1` through the redo port, because the call to `likes/2` has failed, causing the search for another solution to `female/1`. Note that the number in parentheses becomes `2` again because procedure box `2` (for `female/1`) is being re-entered.

```
(2) 2 redo: female(_93) ? c
(2) 2 exit: female(mary) ?
```

Another solution to `female/1` is found, so the debugger exits through the exit port, and then enters `likes/2`. Since the call to `likes/2` is a new call, the number in parentheses becomes `4`.

```
(2) 2 exit: female(mary) ? c
(4) 2 call: likes(mary,wine) ? c
(4) 2 exit: likes(mary,wine) ? c
(1) 1 exit: likes(john,mary) ? c
Who = mary
```

Creeping the rest of the way through the program, you end up with a solution to the query:

```
?- likes(john,Who).
Who = mary ;
```

If you ask to see another solution to the query (by typing a semicolon), the debugger will pick up where it left off:

```
(4) 2 fail: likes(mary,wine) ?
```

The call `likes(mary,wine)` fails, and the computation creeps along.

```
(4) 2 redo: likes(mary,wine) ? c
(4) 2 fail: likes(mary,wine) ?
```

The debugger re-enters `female/1` through the redo port, but because no more solutions can be found, it leaves through the fail port. The rest of the trace looks like this:

```
(4) 2 fail: likes(mary,wine) ? c
(1) 1 redo: likes(john,_93) ? c
(1) 1 fail: likes(john,_93) ? c
no.
```

2.3 Additional Debugger Commands

If your program is large and/or complicated, looking at every port call in the program's execution is often tiresome and unnecessary. Once a procedure is debugged, it is no longer necessary to trace its execution in detail.

However, other procedures may still contain errors. In this case, you want to examine the execution of the questionable procedures without looking at the execution of the correct portions of the program. This can be done by limiting the amount of information printed by the debugger..

2.3.1 Skipping Portions of Code

The first method of limiting information is the *skip* command. This causes the debugger to run the current goal, while suppressing the port information for all subgoals in the interior of the call. However, if the traced goal fails because of the failure of a goal submitted after the traced goal, the debugger will print out all subgoals of the traced goal at their fail port. The following example demonstrates this behavior:

```
?- trace likes(john,Who).
(1) 1 call: likes(john,_93) ? s
(1) 1 exit: likes(john,mary) ? c
Who = mary;
(4) 2 fail: likes(mary,wine) ? c
(2) 2 fail: female(_93) ?
(1) 1 redo: likes(john,_93) ?
```

In this trace, the debugger skips the execution `likes(john,Who)`, and doesn't stop at any of the ports inside the call to `likes/2`. However, when the call fails because of the **;\hveleven** return command in the Prolog shell's answer showing mode, the fail ports from the interior of that call are printed.

2.3.2 Ignoring Even More Execution

The *big skip* command is similar to the skip command above, except that the internal failure ports are not printed when a call fails. You tell the debugger to do a big skip by typing **S** (uppercase 'S') followed by \ReturnKey . A big skip is also *much* more efficient that a normal skip.

```
   ?- trace likes(john,Who).
   (1) 1 call: likes(john,_93) ? S
   (1) 1 exit: likes(john,mary) ? c
   Who = mary ;
   (1) 1 fail: likes(john,_93) ?
```

In this case, only the original call to `likes/2` prints out a failure report. All the other failures are suppressed by the big skip in call number 1.

2.3.3 Getting Back Ignored Execution

Sometimes you might skip over a call only to find that the call failed for some unknown reason. In other cases, the variable instantiations produced by a call are not what you expected. The *retry* command gives you another chance to trace the same call again, presumably in more detail. In the following example, the debugger is waiting at the exit port of call number 1. You can see exactly why `likes(john,mary)`succeeded by retrying the call and creeping through its execution:

```
   (1) 1 exit: likes(john,mary) ? r
   (1) 1 call: likes(john,_93) ? c
   (2) 2 call: female(_93) ? c
   (2) 2 exit: female(susan) ?
```

After a retry command, the state of the program is reset to the way it was just before the retried goal ran, except for side effects. Side effects not undone by a retry include modifications to the database via `assert/1` or `retract/1`.

2.4 Changing the Leashing

After watching all of the ports during a trace, you might find that you want don't want to stop at every port that passes by. For example, it might not be useful to see the fail and exit ports in the execution of your program. By changing the *leashing* of the debugger via leash/1, you can control which ports are actually printed. The following goal disables the fail and exit ports, and enables the call and redo ports.

```
?- leash([call,redo]).
```

If you only want to set leashing on one port, you can omit the square brackets, as in

```
leash(call).
```

`leash(all)` enables the printing of every port.

2.5 Spying on Code

The spy/1 builtin allows you to set *spy points* in your program. A spy point will interrupt the normal execution of your program and begin tracing at every call to a particular procedure. This is useful when most of your program is working correctly, but a few isolated procedures still need attention. The following example uses a program that takes derivatives of a function and simplifies the result:

```
diff :-
  write('type in: Var,Fn'), nl,
  read((X,F)),
  diff(X,F,Answer),
  write(Answer), nl,
  simplify(Answer,Simple),
  write(Simple), nl.

simplify(A+B,Sum) :-
  number(A),
  number(B), !,
  Sum is A+B.
simplify(Exp,Exp).
```

Suppose that you are confident that `diff/3` itself works correctly, but suspect that there are bugs inside `simplify/2`. Rather than tracing the entire program, you can place a spy point on `simplify/2`:

```
?- spy simplify/2.
```

When your program attempts to call `simplify/2`, the debugger will take control and let you begin tracing.

```
?- diff.
type in Var,Fn
x,x+x.
1+1
(1) 1 call: simplify(1+1,_255) ? c
(2) 2 call: integer(1) ? c
(2) 2 exit: integer(1) ?
```

The debugger only stops at the ports for `simplify/2` and its subgoals. When `simplify/2` succeeds or fails, normal program execution resumes until the next spy point. In this case, there are no more spy points, so the program simply prints out its answer.

```
(1) 1 exit: simplify(1+1,2) ? c
2
```

If for any reason `simplify/2` tries to find another solution, the debugger will wake up again at the redo port and allow you to continue tracing.

2.6 Leaping Ahead

The debugger also lets you to *leap* from the trace of a call to the next spy point. Leaping suppresses the normal action of the debugger, allowing the program to continue uninhibited until it runs into the next spy point.

In the following example, spy points are placed on `diff/0` and `simplify/2`. As soon as the program starts, the spy point at `diff/0` activates the debugger. Then a leap command suppresses the debugger until the call to `simplify/2`, where the debugger picks up again:

```
?- spy simplify/2, spy diff/0.
yes.
```

```
?- diff.
(1) 1 call: diff ? l
type in: Var,Fn
x,x+x.
(7) 2 call: simplify(1+1,_255) ? s
(7) 2 exit: simplify(1+1,2) ? c
(8) 11 call: write(2) ?
```

2.7 Turning Off Spy Points

The nospy/0 builtin removes all spy points from your program, while `nospy/1` removes a specific spy point:

```
?- nospy a/1.
Spy point removed for user:a/1.
yes.
```

2.8 Getting Help

Typing ‘h’ at the `?` prompt of the debugger gives a summary of the debugger commands.

2.9 Exiting the Debugger

There are two ways to leave the debugger. The *abort* command (`a`) abandons the current computation, and returns control to the Prolog shell. The *exit* command (`e`) leaves the debugger, causing ALS Prolog to exit.

3 ALS Integrated Development Environment

The ALS Integrated Development Environment (IDE) provides a GUI-based developer-friendly setting for developing ALS Prolog programs. Start the ALS IDE either by clicking on its icon (Windows or Macintosh, or CDE versions of Unix), or

by issuing

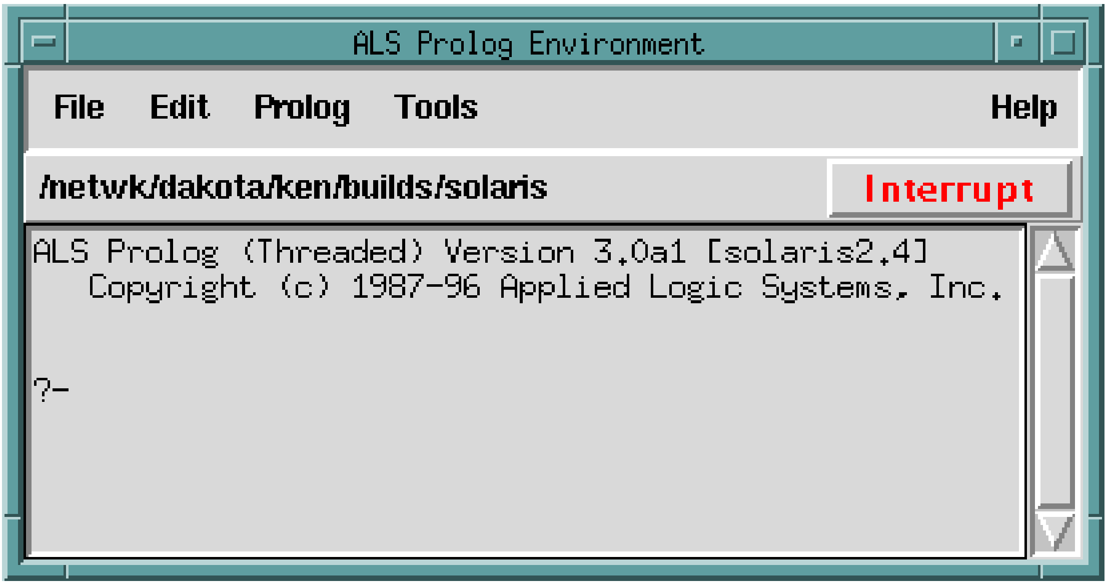
```
alsdev
```

in an appropriate command window. The IDE displays an initial spash screen

while it loads, and then replaces the spash screen with the main listener window.

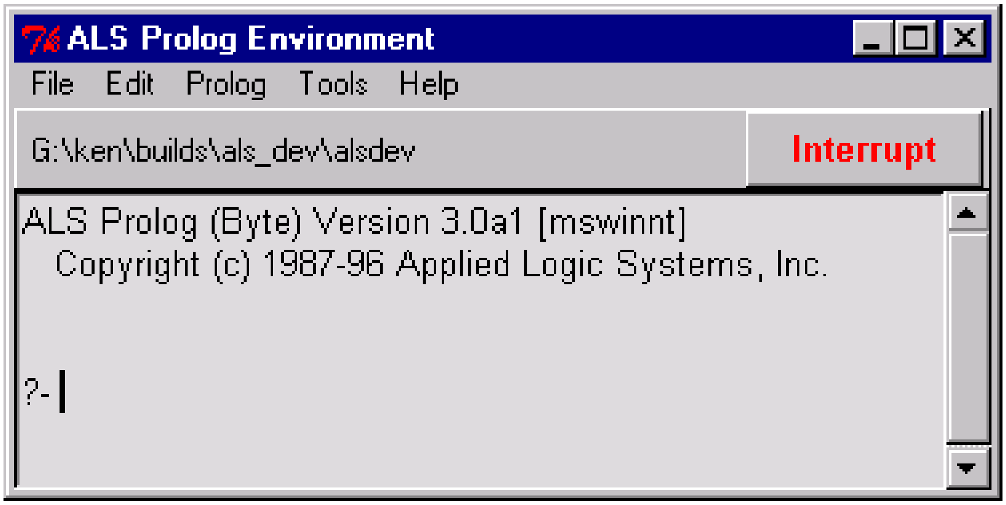
3.1 Main Environment Window

The details of the appearance of the ALS IDE windows will vary across the platforms. Here is what reduced-size versions of the main window look like on Unix::



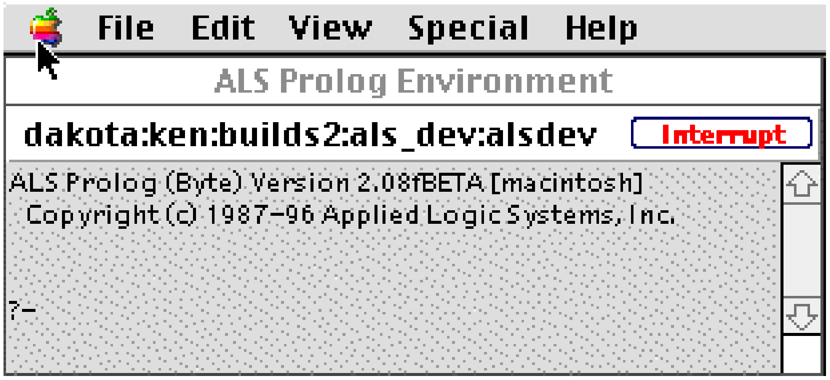
```
ALS Prolog (Threaded) Version 3.0a1 [solaris2.4]
   Copyright (c) 1987-96 Applied Logic Systems, Inc.


?-
```

and on Windows:



```
ALS Prolog (Byte) Version 3.0a1 [mswinnt]
   Copyright (c) 1987-96 Applied Logic Systems, Inc.


?-
```

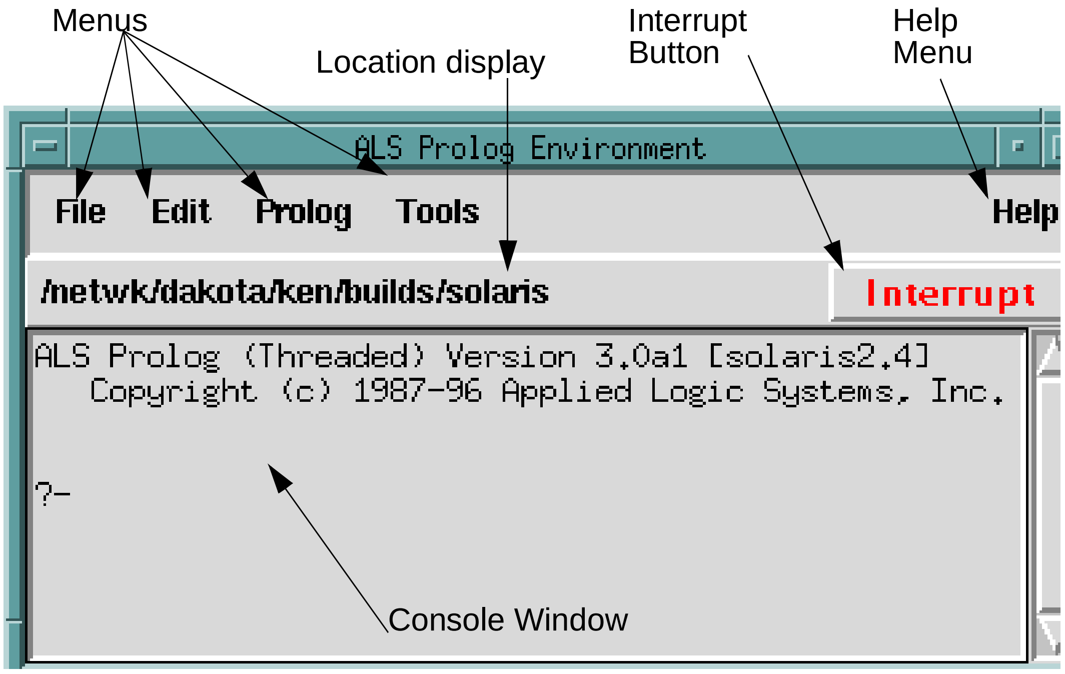
and on Macintosh:



```
ALS Prolog (Byte) Version 2.08fBETA [macintosh]
 Copyright (c) 1987-96 Applied Logic Systems, Inc.

?-
```

Throughout the rest of this section and others dealing with aspects of the ALS IDE, we will not attempt to show all three versions of each and every window or display. Instead, we will typically show just one, since they are all quite similar.

The parts of the main window are:



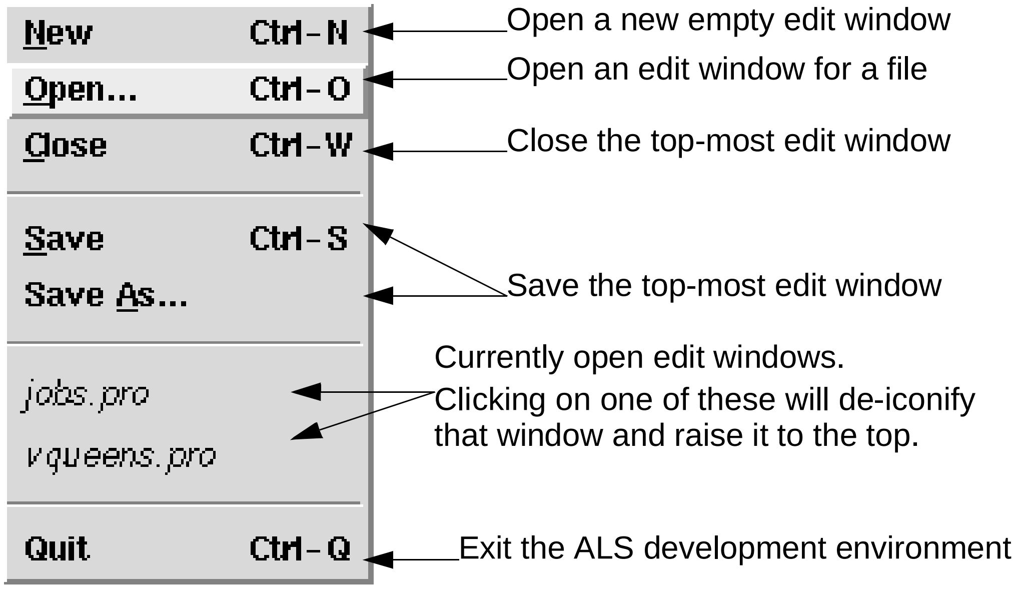
```
ALS Prolog (Threaded) Version 3.0a1 [solaris2.4]
   Copyright (c) 1987-96 Applied Logic Systems, Inc.

?-
```

The Menus will be described below. The Location Display shows the current directory or folder. The Interrupt Button is used to interrupt prolog computations, while the Help Menu will in the future provide access to the help system (which can also be run separately). The Console Window is used to submit goals to the system and to view results.

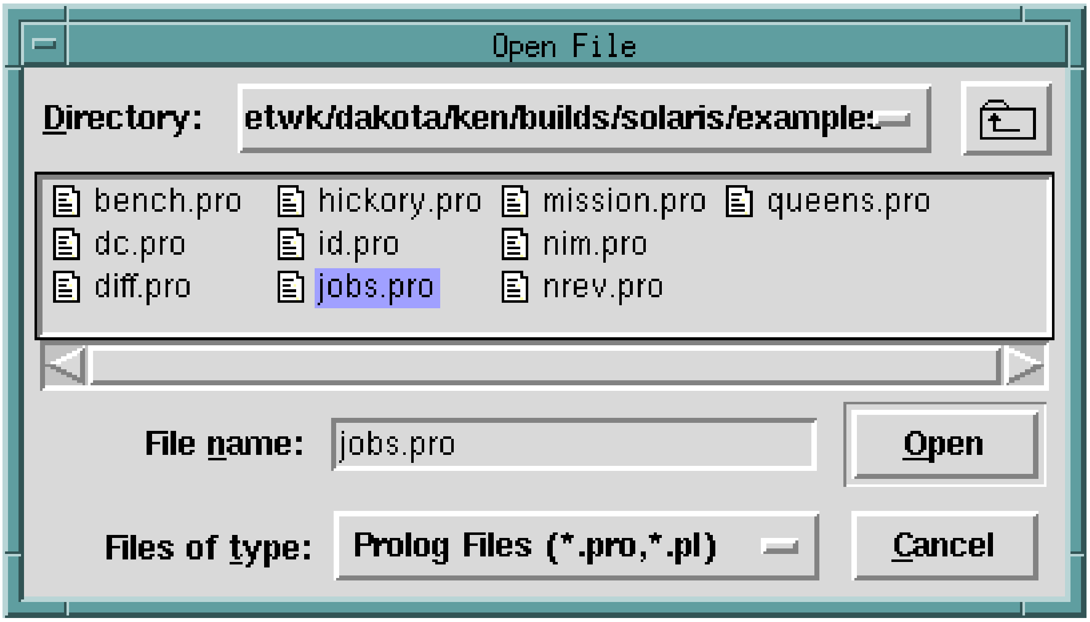
3.2 Menus

The ALS IDE is undergoing steady development. Some of the planned features are not yet implemented, and so menu items corresponding to them will show as grayed-out. The options indicated by the accellerator keys on the menus apply when the focus (insertion cursor) is located over the main listener window, over the debugger window, or over any editor window.

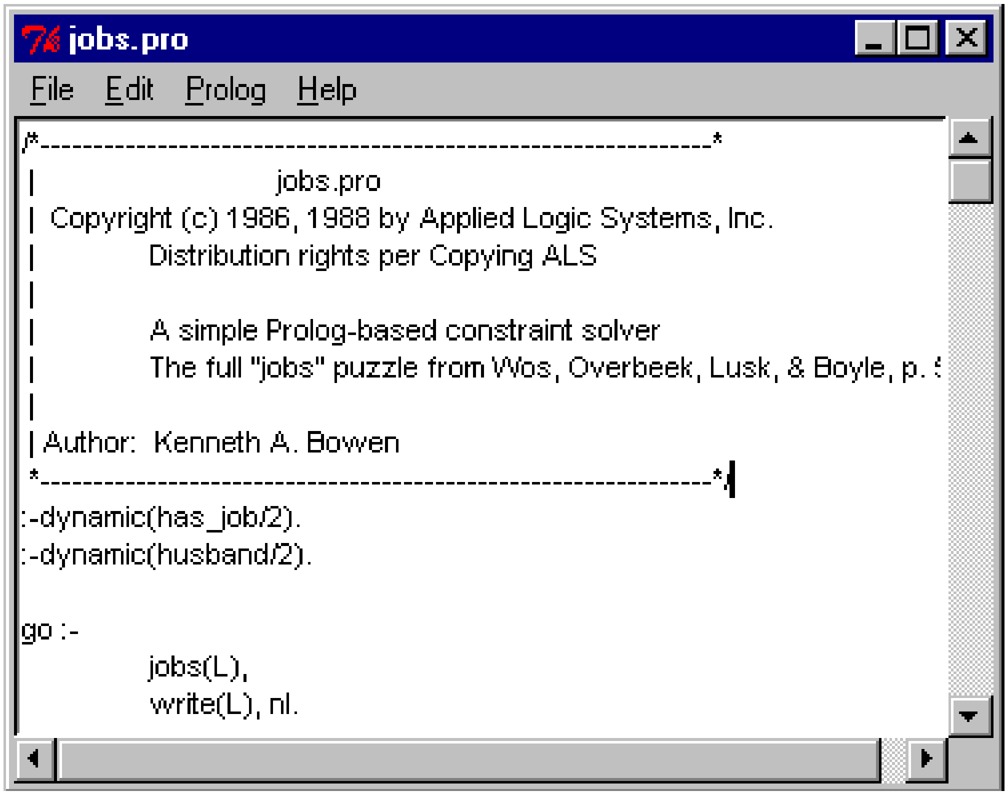
3.2.1 File Menu


New Ctrl-N
Open a new empty edit window
Open... Ctrl-O
Open an edit window for a file
Close Ctrl-W
Close the top-most edit window
Save Ctrl-S
Save As...
Save the top-most edit window
jobs.pro
vqueens.pro
Currently open edit windows.
Clicking on one of these will de-iconify that window and raise it to the top.
Quit Ctrl-Q
Exit the ALS development environment

New, Open, Close, Save, and SaveAs apply to editor windows, and have their usual meanings. New opens a fresh editor window with no content, while Open allows one to select an existing file for editing. The open file dialog looks like this : :



Selecting a file to open produces a window looking like the following:

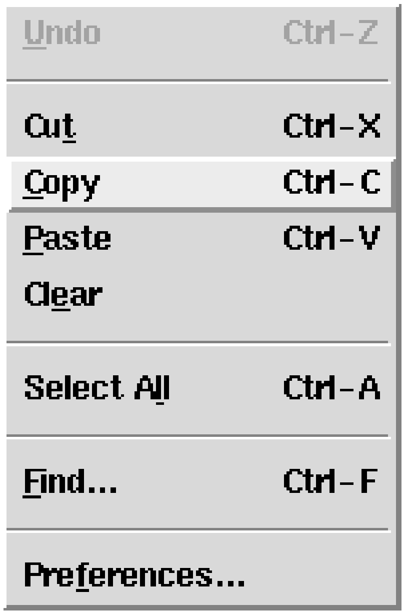


```
/*-------------------------------------------------------------*
 |                           jobs.pro
 | Copyright (c) 1986, 1988 by Applied Logic Systems, Inc.
 |            Distribution rights per Copying ALS
 |
 |            A simple Prolog-based constraint solver
 |            The full "jobs" puzzle from Wos, Overbeek, Lusk, & Boyle, p. 5
 |
 | Author:  Kenneth A. Bowen
 *-------------------------------------------------------------*/
:-dynamic(has_job/2).
:-dynamic(husband/2).


go :-
            jobs(L),
            write(L), nl.
```

The Close, Save and SaveAs entries from the file menu apply to the edit window having the current focus.

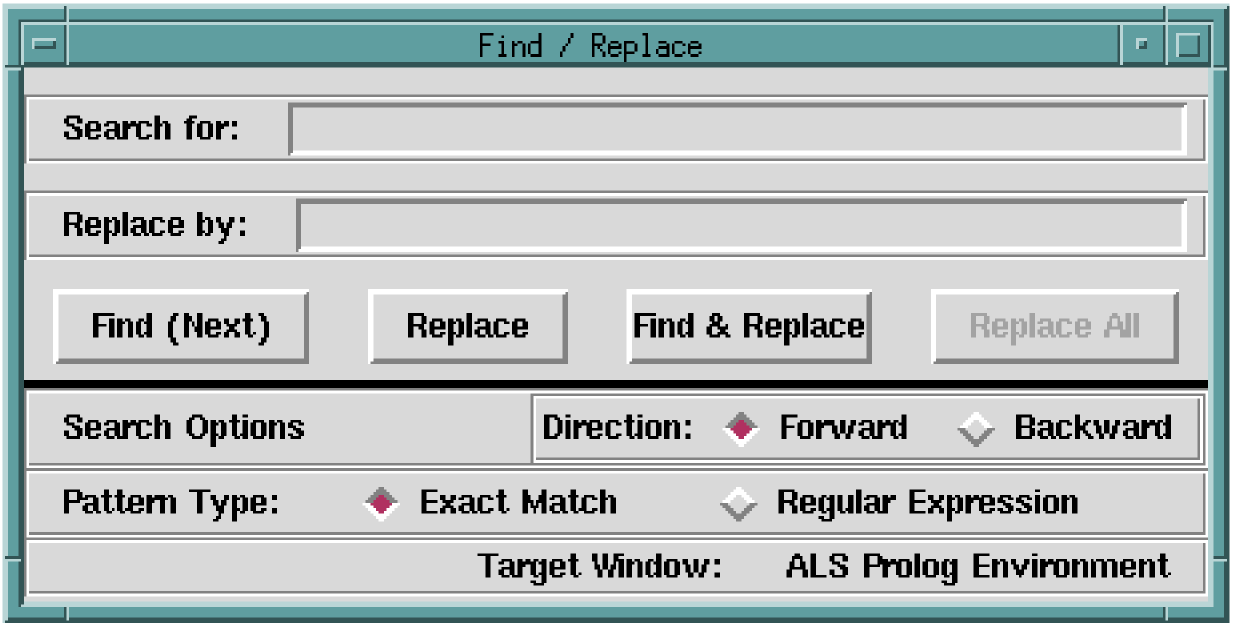
Quit allows you to exit from the ALS Prolog IDE.

3.2.2 Edit Menu



Cut, Copy, Paste, and Clear apply as usual to the current selection in an editor window. Copy also applies in the main llistener window. Paste in the main window always pastes into the last line of the window. Select All only applies to editor windows.

The Find button brings up the following dialog:



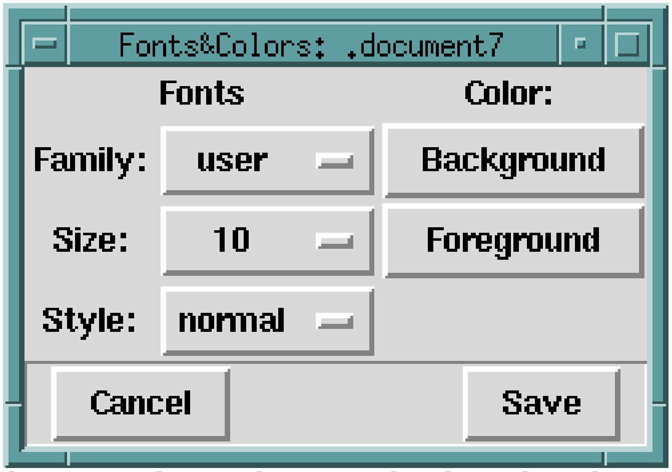
Direction: [✓ Forward] Backward

Pattern Type: [✓ Exact Match] Regular Expression

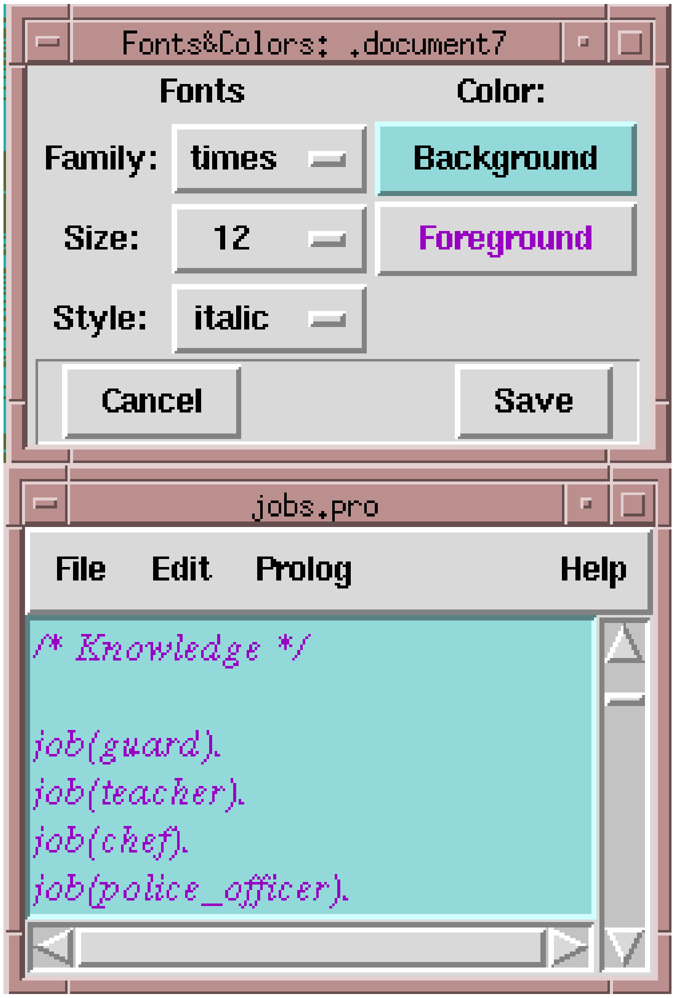
Target Window: ALS Prolog Environment

This dialog allows one to search in whatever window is top-most, and to carry out replacements in editor windows. If the text which has been typed into the Search for: box is located, the window containing it is adjusted so that the sought-for text is approximately centered vertically, and the text is highlighted.

The Preferences choice produces the following popup window: Selections made



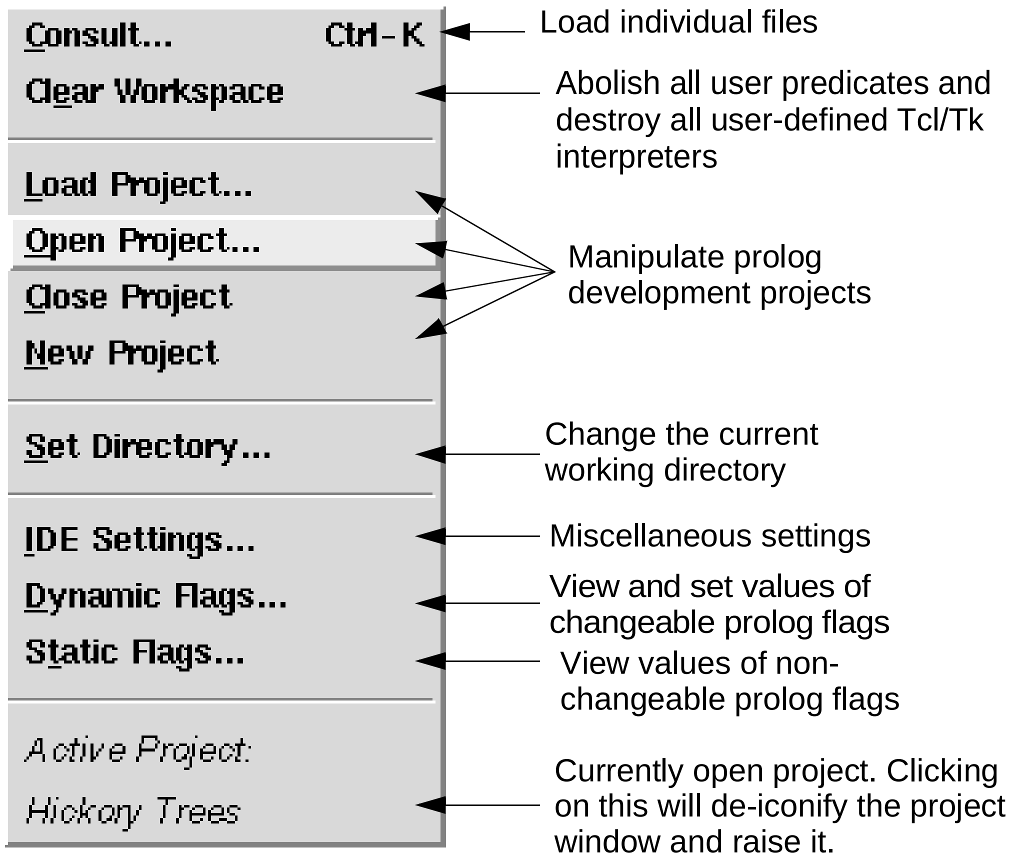
using any of the buttons on this window immediately apply to the window (listener, debugger, or editor window) from which the Preferences button was pressed. For example:



```
/* Knowledge */

job(guard).
job(teacher).
job(chef).
job(police_officer).
```

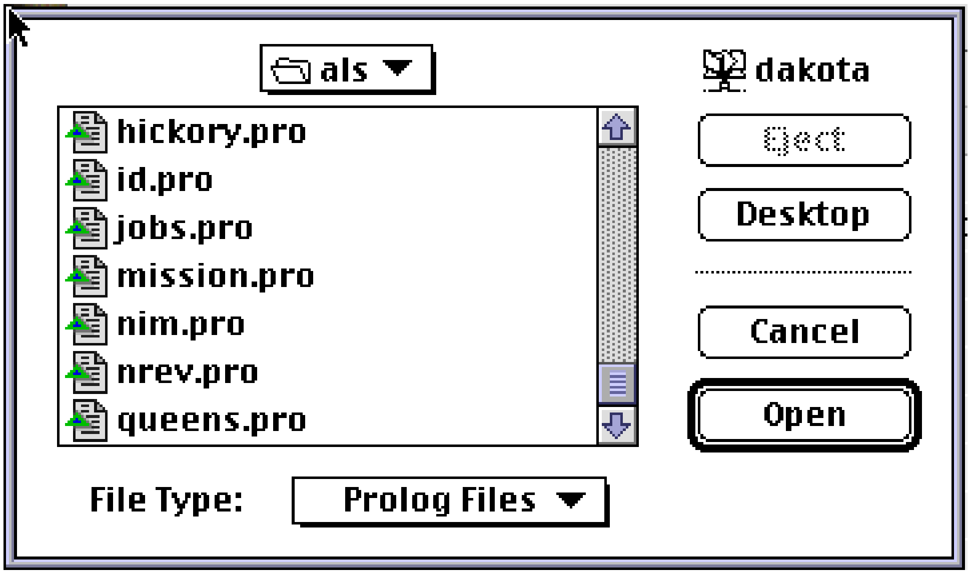
Selecting Cancel simply removes the Fonts & Colors window without undoing any changes. Selecting Save records the selected preferences in the inititialization file (*alsdev.ini*) which is read at start-up time, and also records the selections globally for the current session. Although no existing editor windows are changed, all new editor windows created will use the newly recorded preferences. Preferences for the main listener window and the debugger window are saved separately from the editor window preferences.

3.2.3 Prolog Menu


Consult... Ctrl-K
Clear Workspace
Load Project...
Open Project...
Close Project
New Project
Set Directory...
IDE Settings...
Dynamic Flags...
Static Flags...
Active Project:
Hickory Trees
Load individual files
Abolish all user predicates and destroy all user-defined Tcl/Tk interpreters
Manipulate prolog development projects
Change the current working directory
Miscellaneous settings
View and set values of changeable prolog flags
View values of non-changeable prolog flags
Currently open project. Clicking on this will de-iconify the project window and raise it.

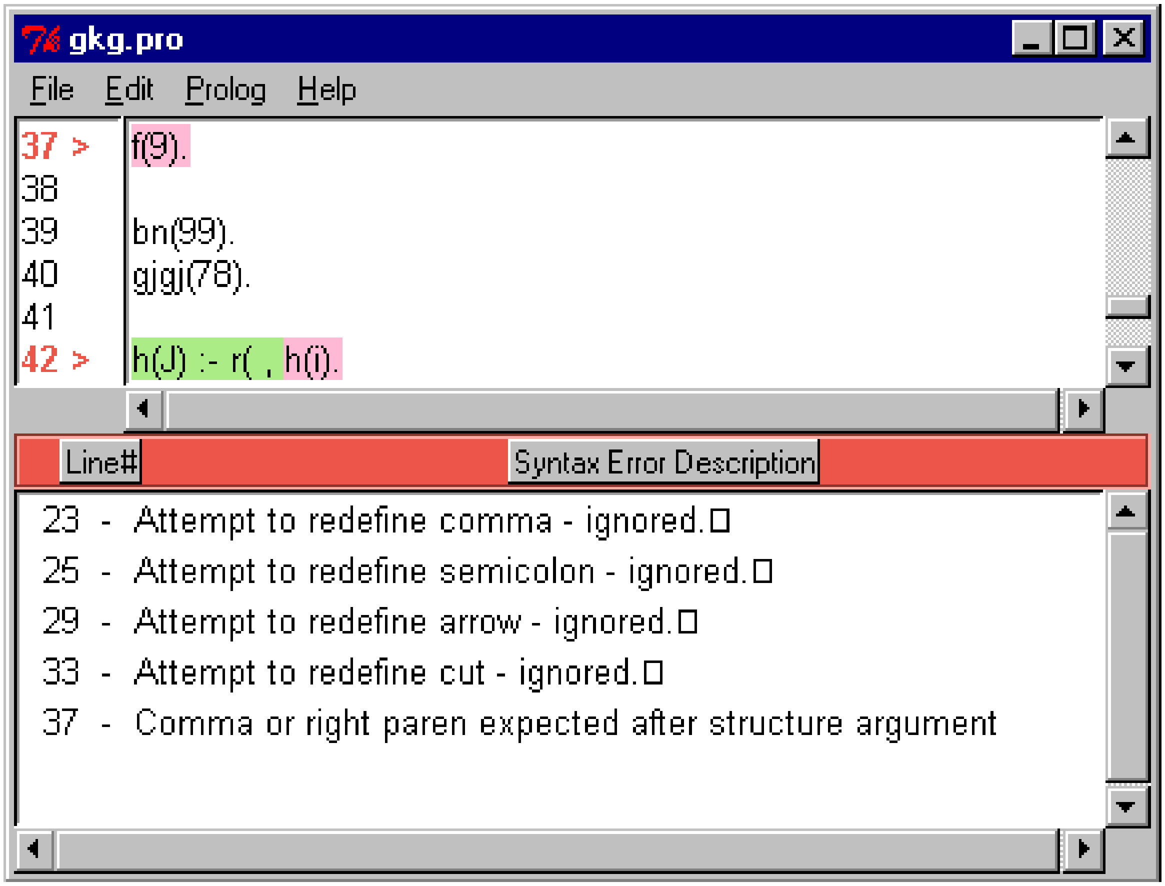
Choosing Consult produces two different behaviors, depending on whether the Prolog menu was pulled down from the main listener or debugger windows, or from an editor window. Using the accelerator key sequence (*Ctrl-K* on Unix and Windows, and *<AppleKey>K* on Macintosh) produces equivalent behaviors in the different settings. If Consult is chosen from the main listener or debugger windows, a file selection dialog appears, and the selected file is (re)consulted into the current

Prolog database:



In contrast, if Consult is selected from an edit window (or the *Ctrl/<apple>-K* accellerator key is hit over that window), the file associated with that window is (re)consulted into the Prolog database; if unsaved changes have been made in the editor window, the file/window is first Saved before consulting.

If a file containing syntax errors is consulted, these errors are collected, an editor window into the file is opened, and the errors are displayed, as indicated below:



```
37 >  f(9).
38
39     bn(99).
40     gjgj(78).
41
42 >  h(J) :- r( , h(i).
```

```
23  -  Attempt to redefine comma - ignored.
25  -  Attempt to redefine semicolon - ignored.
29  -  Attempt to redefine arrow - ignored.
33  -  Attempt to redefine cut - ignored.
37  -  Comma or right paren expected after structure argument
```

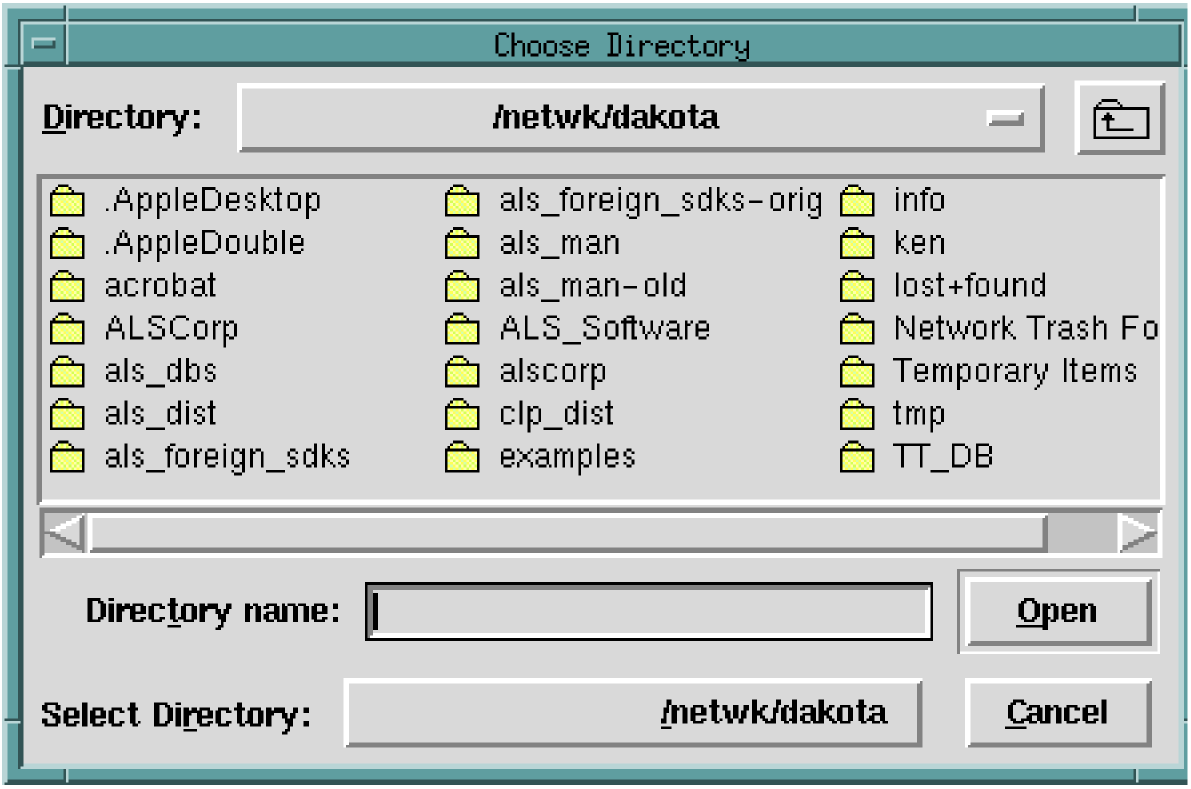
Line numbers are added on the left, and the lines on which errors occur are marked in red. Lines in which an error occurs at a specified point are marked in a combination of red and green, with the change in color indicating the point at which the error occurs. Erroneous lines without such a specific error point, such as attempts to redefine comma, are marked all in red. A scrollable pane is opened below the edit window, and all the errors that occurred are listed in that pane. Double clicking on one the listed errors in the lower pane will cause the upper window to scroll until the corresponding error line appears in the upper pane. The upper pane is an ordinary error window, and the errors can be corrected and the file saved. Choosing

Prolog> Consult, or typing ^K will cause the file to be reconsulted, and the error panes to be closed.

Clear Workspace causes all procedures which have been consulted to be abolished, , including clauses which have been dynamically asserted. In addition, future releases, all user-defined Tc/Tkl interpreters will also be destroyed.

The four Project-related buttons as well as the bottom entry on this menu are described in the next section.

Set Directory ... allows you to change the current working directory. Here is the Unix version of this dialog::



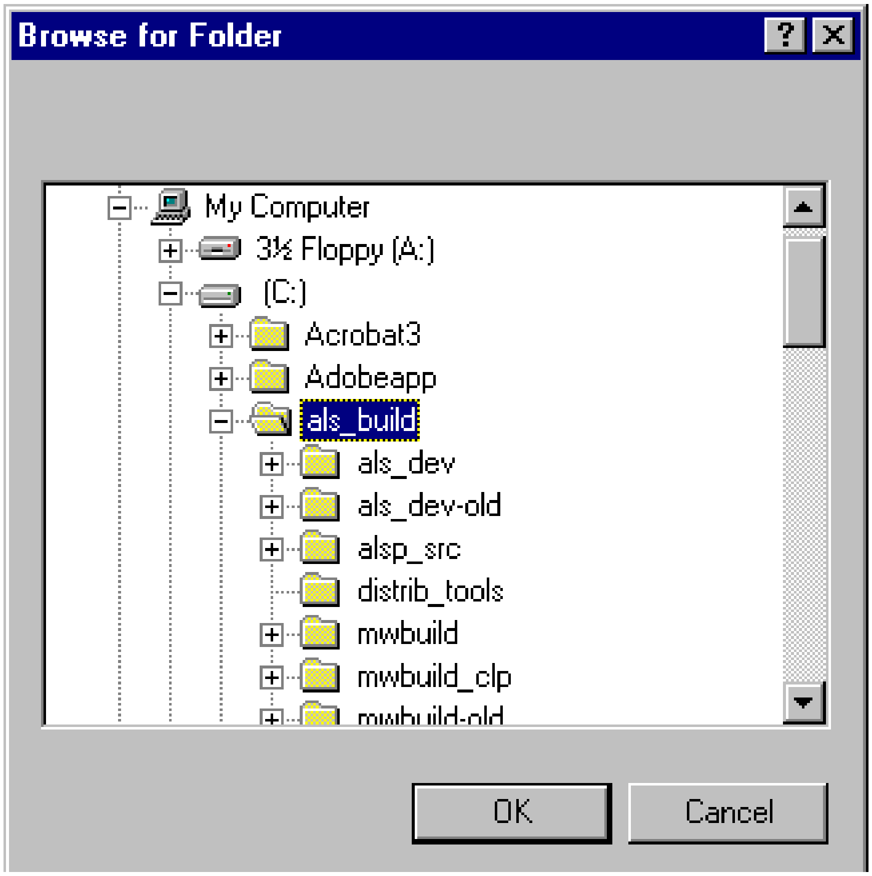
Directory: /netwk/dakota

```
.AppleDesktop       als_foreign_sdks-orig  info
.AppleDouble        als_man                ken
acrobat             als_man-old            lost+found
ALSCorp             ALS_Software           Network Trash Fo
als_dbs             alscorp                Temporary Items
als_dist            clp_dist               tmp
als_foreign_sdks    examples               TT_DB
```

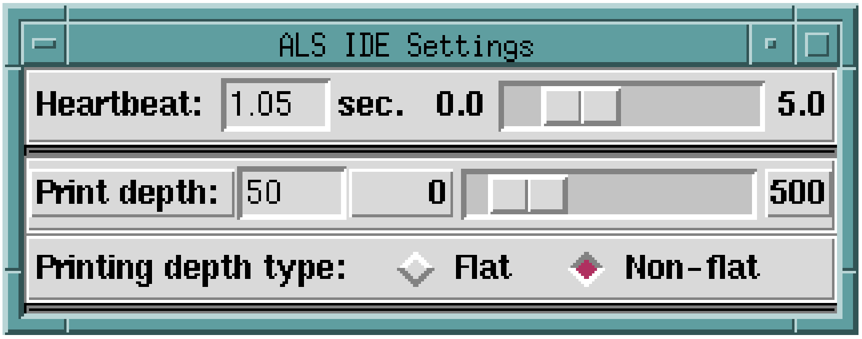
Directory name:

Select Directory: /netwk/dakota

And here is the Windows version:



Selecting IDE Settings raises the following dialog:



Heartbeat: 1.05 sec. 0.0 5.0

Print depth: 50 0 500

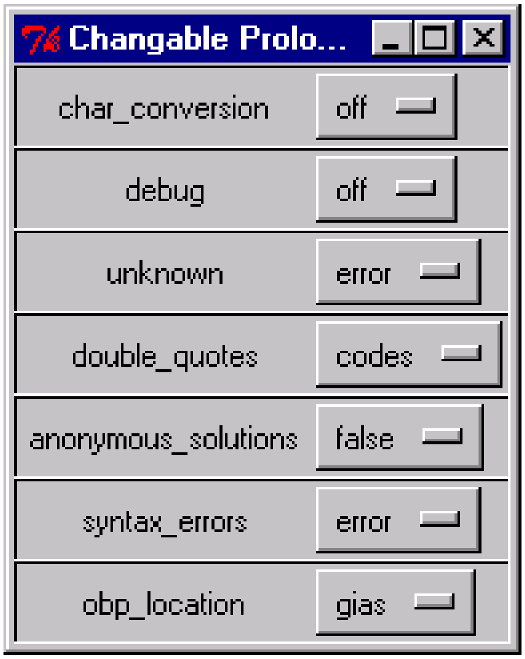
Printing depth type: Flat [✓ Non-flat]

The Heartbeat is the time interval between moments when a Prolog program temporarily yields control to the Tcl/Tk interface to allow for processing of GUI events, including clicks on the Interrupt button.

The Print depth setting contols how deep printing of nested terms will proceed; when the depth limit is reached, some representation (normally ‘*’ or ‘...’) is printed instead of continuing with the nested term.

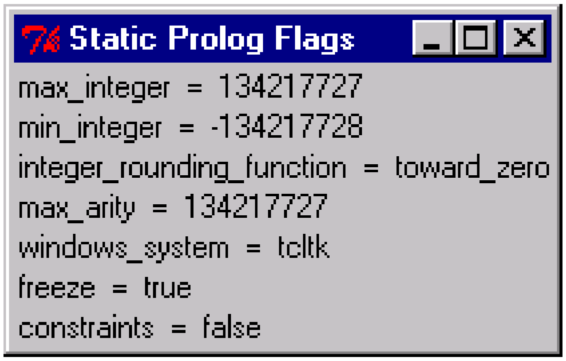
The Printing depth type setting determines whether traversing a list or the top-level arguments of a term increases the print depth counter. The Flat setting indicates that the counter will not increase as one traverses a list or the top level of a term, while Non-flat specifies that the counter will increase.

Selecting Dynamic Flags produces a popup window which displays the current values of all of the changable Prolog flags in the system, and allows one to reset any of those values:



char_conversion	off
debug	off
unknown	error
double_quotes	codes
anonymous_solutions	false
syntax_errors	error
obp_location	gias

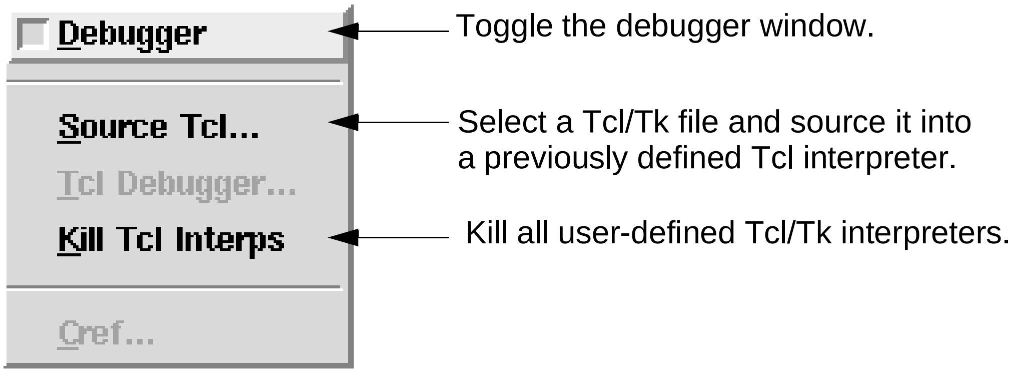
Selecting Static Flags producs a popup window displaying the values of all of the unchangable Prolog flags for the system:


Static Prolog Flags

```
max_integer = 134217727
min_integer = -134217728
integer_rounding_function = toward_zero
max_arity = 134217727
windows_system = tcltk
freeze = true
constraints = false
```

3.2.4 Tools Menu

This menu is expected to grow with new entries in future releases. Currently:


Debugger
Source Tcl...
Tcl Debugger...
Kill Tcl Interps
Cref...
Toggle the debugger window.
Select a Tcl/Tk file and source it into a previously defined Tcl interpreter.
Kill all user-defined Tcl/Tk interpreters.

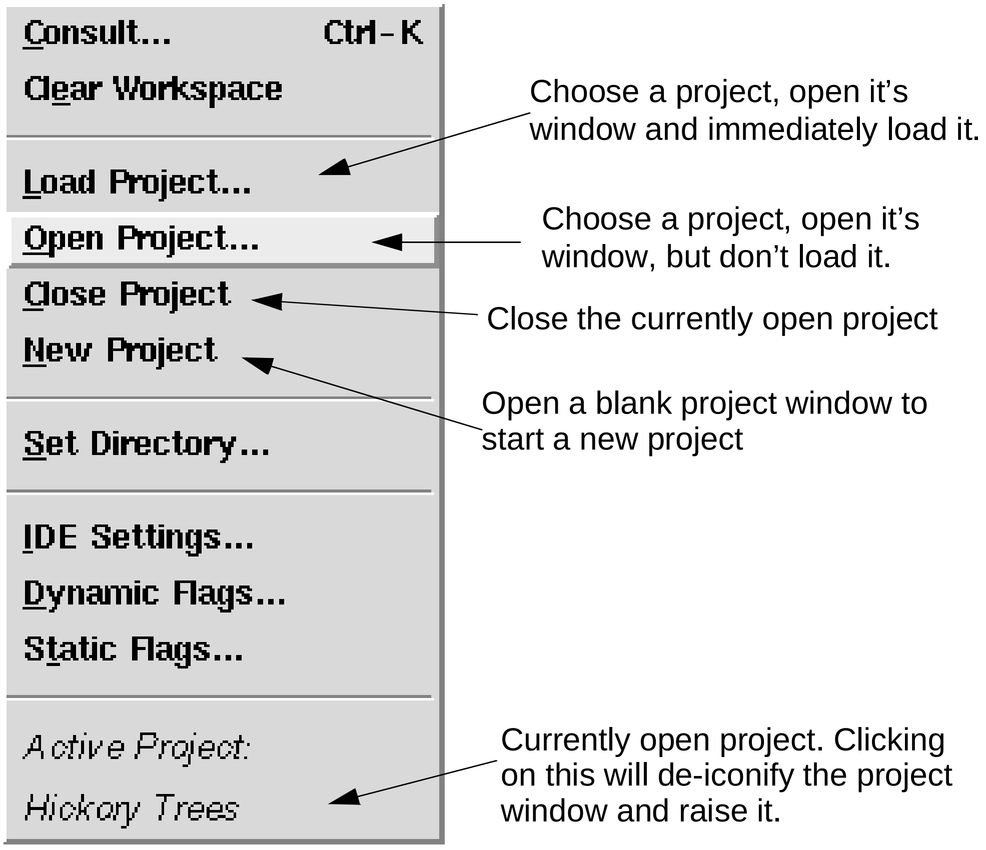
Selecting Debugger provides access to the GUI Debugger; this will be described in detail in Chapter 5 (*Using the ALS IDE Debugger*) .

Source Tcl allows one to “source” a Tcl/Tk file into a user/program-defined Tcl interpreter; the dialog prompts you for the name of the interpreter.

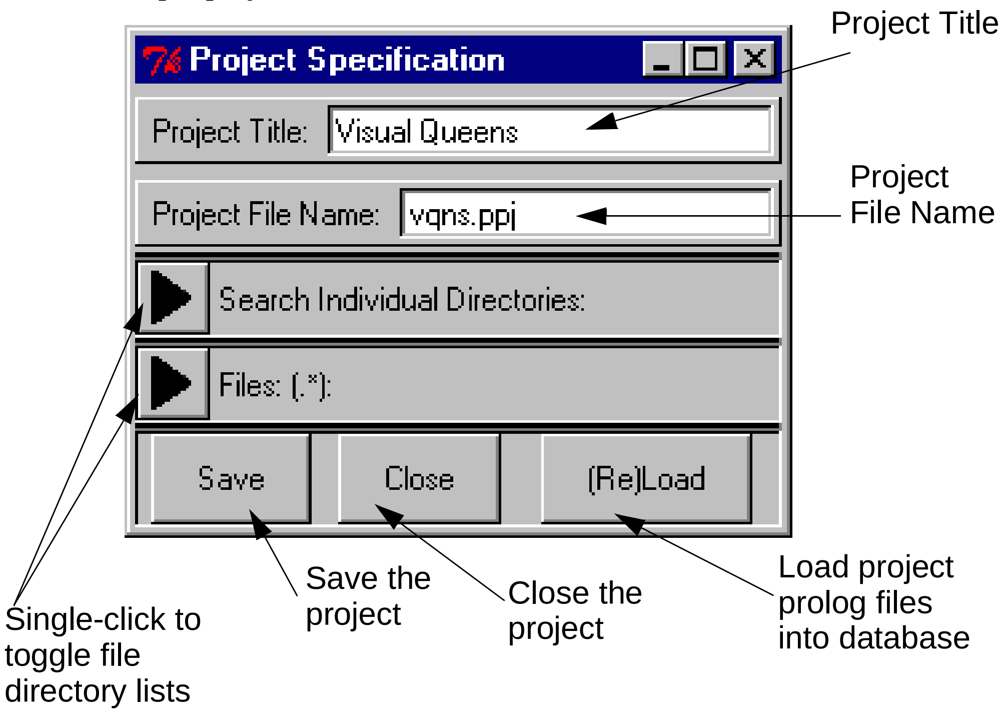
Kill Tcl Interps destroys all user-defined Tcl/Tk interpreters.

4 Prolog Development Projects

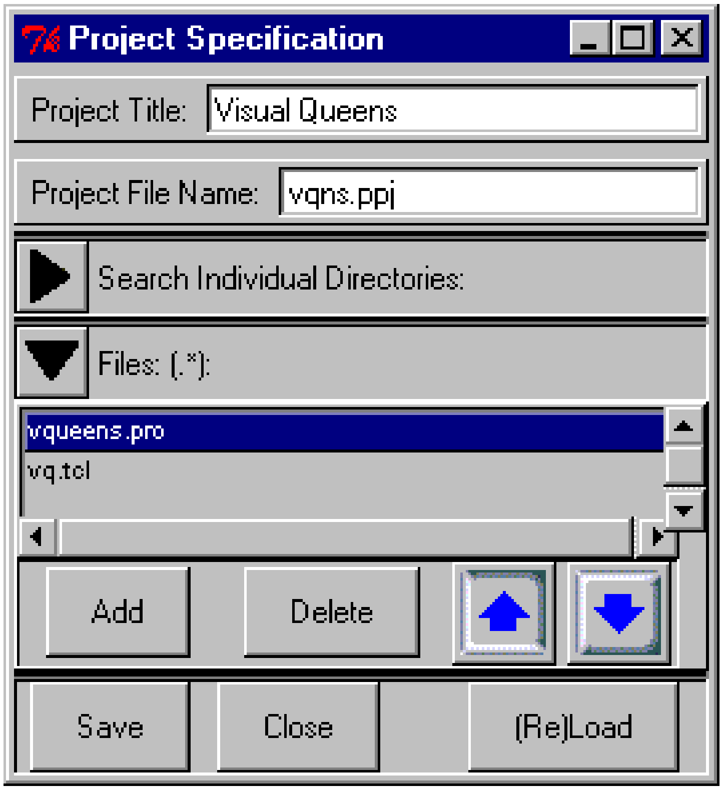
The ALS Prolog Development Environment (alsdev) supports a simple concept of development project which nonetheless, is quite useful. At it's most elementary, a project is just a named collection of files. Projects are accessed through the Prolog menu introduced in the preceeding section: Once again:


Consult... Ctrl-K
Clear Workspace
Load Project...
Open Project...
Close Project
New Project
Set Directory...
IDE Settings...
Dynamic Flags...
Static Flags...
Active Project:
Hickory Trees
Choose a project, open it's window and immediately load it.
Choose a project, open it's window, but don't load it.
Close the currently open project
Open a blank project window to start a new project
Currently open project. Clicking on this will de-iconify the project window and raise it.

Here is a sample project window:


Project Specification
Project Title: Visual Queens
Project File Name: vqns.ppj
Search Individual Directories:
Files: (.*):
Save
Close
(Re)Load
Project Title
Project File Name
Single-click to toggle file directory lists
Save the project
Close the project
Load project prolog files into database

Clicking on the triangular arrow button to the left of ‘Files:’ causes the list of files incorporated in the project to open, as shown below:

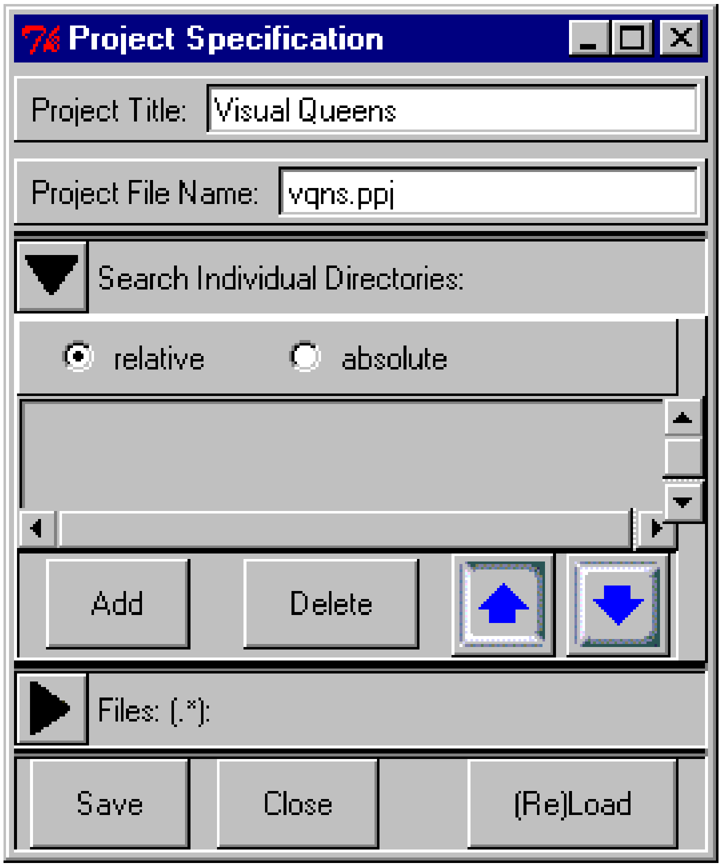


Files in the list can be deleted by selecting them, and the clicking the Delete button. New files may be added by clicking the Add button. This will raise a file selection dialog which allows you to select one or several files at the same time, from either the current directory, or other directories.

When the project is loaded, by clicking the (Re)Load button, the prolog files in the list (those with *.pro or *.pl extensions) are loaded in the order they appear. One can change the order of the files by single-clicking one to select it, and the using the blue up/down arrows to move it in the list.

Finally, double-clicking a file in the list will cause an editor window to be opened for that file.

Complex projects may incorporate files from several different directories. These can be specified using the Search Directories list, which is opened by toggling the other arrow button on the left:



Project Title: Visual Queens

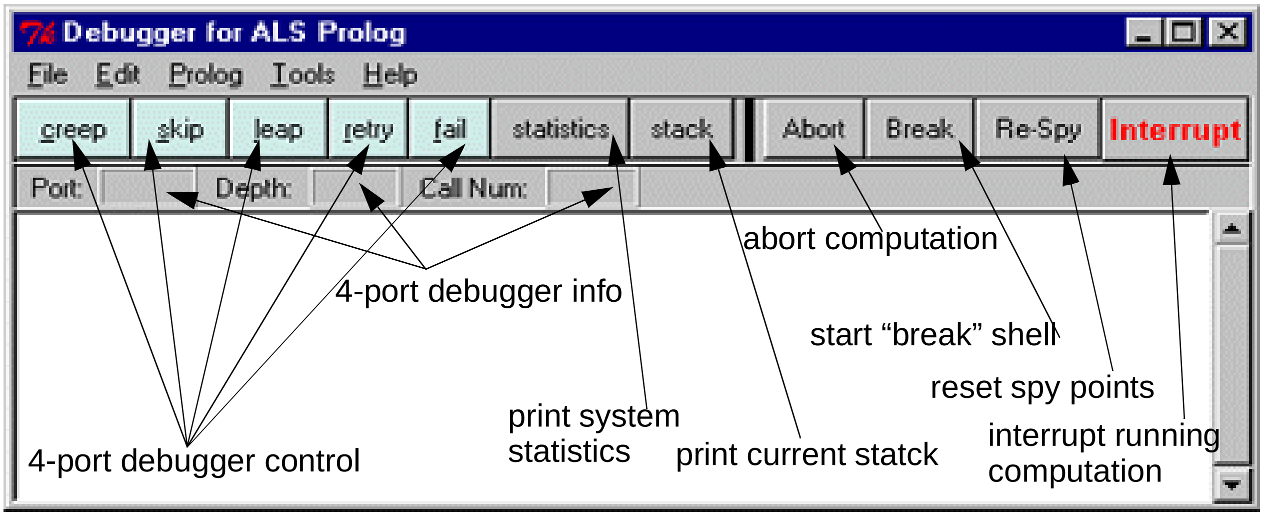
Project File Name: vqns.ppj

[✓ relative] absolute

As with files, one can single-click an entry in this list to select it, and remove it with the Delete button, or change its position using the blue up/down arrows. New directories are added by clicking Add, which raises a directory selection dialog. If the ‘relative’ radio button is checked, the new added path will be represented relative to the current working directory if this is possible. Otherwise, the path is represented in absolute form.

5 Using the ALS IDE Debugger

Selecting the Debugger entry from the Tools menu causes the primary debugger window to appear::



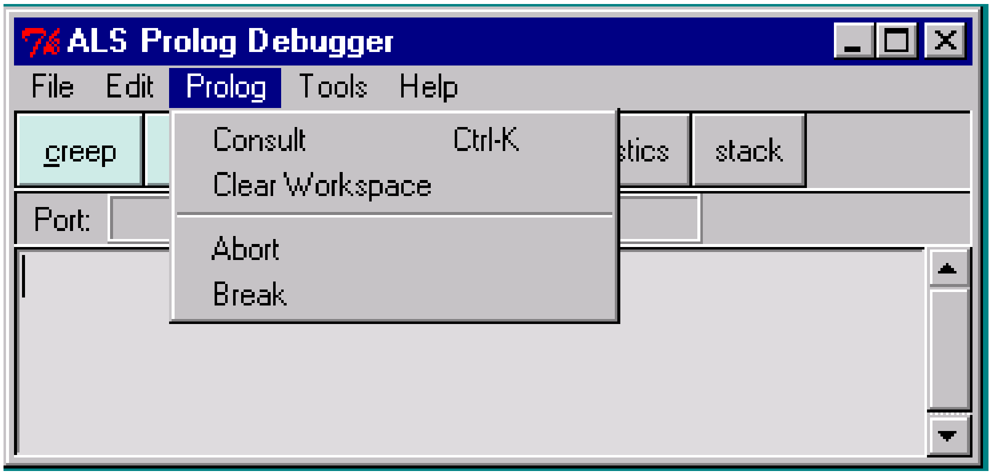
The debugger combines a traditional prolog four-port debugger (as described in Chapter 2 (*Using the Four-Port Debugger*)) with a source-code trace debugger. The details of the debugger action and the source trace will be described below. First we will examine the menus and buttons on the debugger window.

5.1 Debugger Window Menus.

The first two debugger menus, File and Edit, provide the same facilities as discussed for all other windows in Chapter 3 (*ALS Integrated Development Environment*) .

5.1.1 Prolog Menu.

The top two items of the Prolog menu are also the same as earlier:



However, the lower portion has been replaced with two debugger-related items:

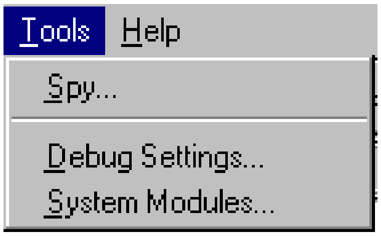
Abort -- choosing this causes the computation currently being traced to be aborted (effectively, abort/0 is invoked).

Break -- choosing this causes a break shell to be started without disturbing the current computation being traced.

If no computation is being traced, the Abort and Break choices have no effect.

5.1.2 Tools Menu

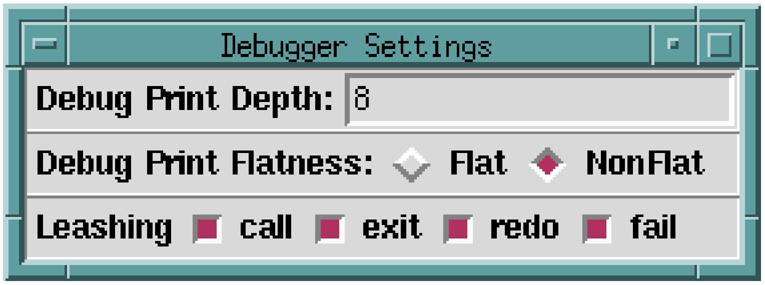
The Tools Menu



is completely specific to the debugger.

Choosing Spy allows one to selectively set and remove spy points. This will be discussed in detail below.

Choosing Debug Settings raise the following popup window:

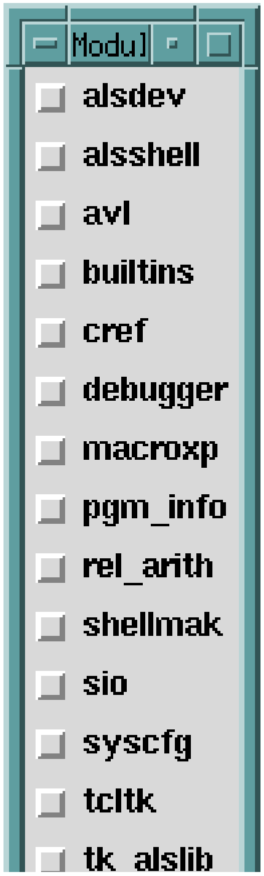


Debugger Settings

Debug Print Depth: 8

Debug Print Flatness: Flat [✓ NonFlat]

Leashing [✓ call] [✓ exit] [✓ redo] [✓ fail]



Modul

alsdev
alsshell
avl
builtins
cref
debugger
macroxp
pgm_info
rel_arith
shellmak
sio
syscfg
tcltk
tk_alslib

These settings control which debugger ports are shown, and also control the appearance of the lines printed for the ports which are show.

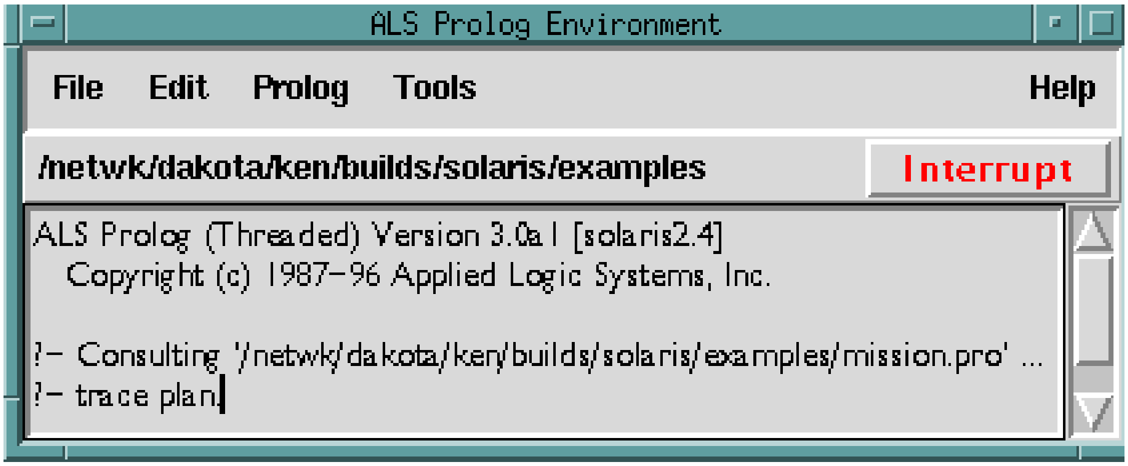
Choosing System Modules contols whether or not tracing will go inside predicates defined in various system modules. The dialog:appears to the left. Toggling one of these buttons allows a trace to show the interior of predicates defined in that module.

5.2Tracing with the IDE Debugger.

Suppose we have raised the debugger, consulted the *mission.pro* example from the supplied example programs. Then begin tracing by typing

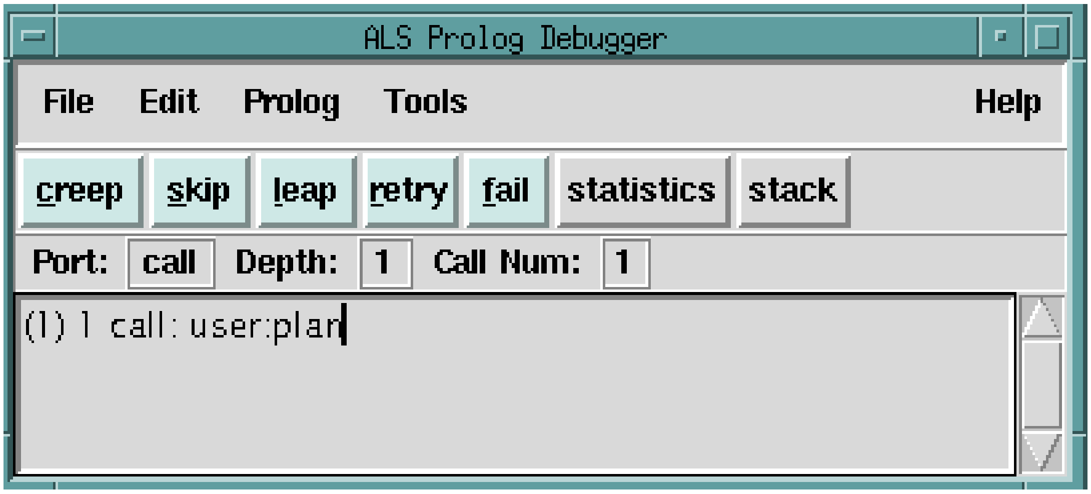
```
trace plan.
```

in the main listener window, as show below: :

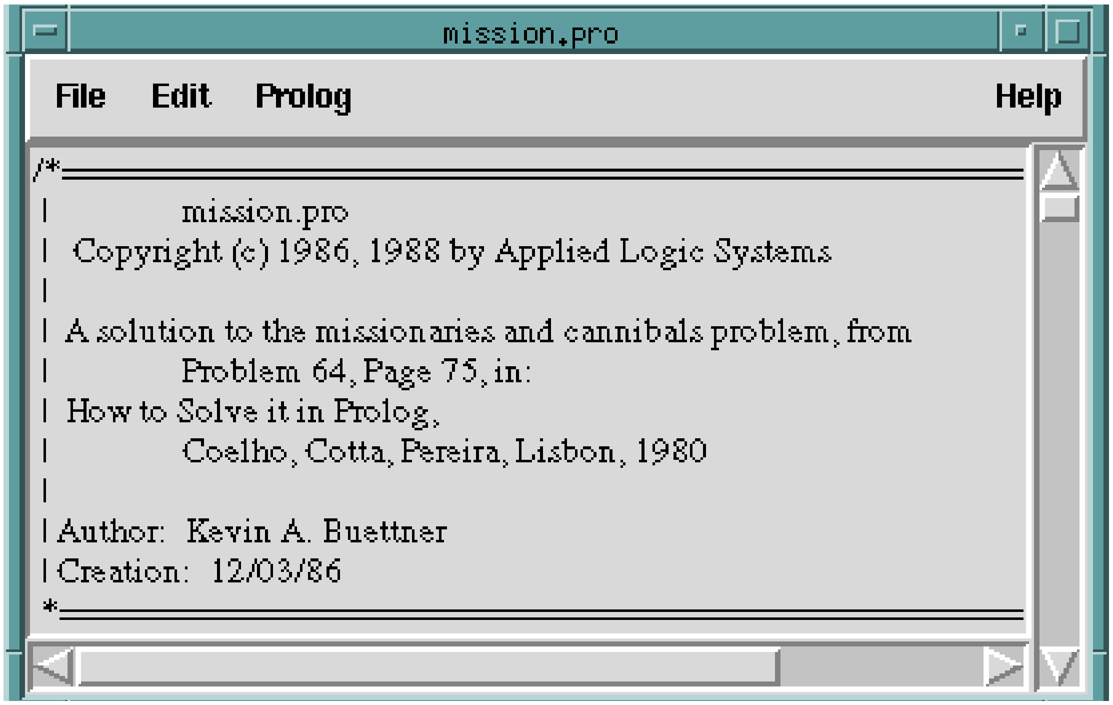


```
ALS Prolog (Threaded) Version 3.0a1 [solaris2.4]
   Copyright (c) 1987-96 Applied Logic Systems, Inc.

?- Consulting '/netwk/dakota/ken/builds/solaris/examples/mission.pro' ...
?- trace plan.
```

The debugger will immediately open an editor window containing the *mission.pro* file, while at the same time starting the four-port trace in the debugger window:

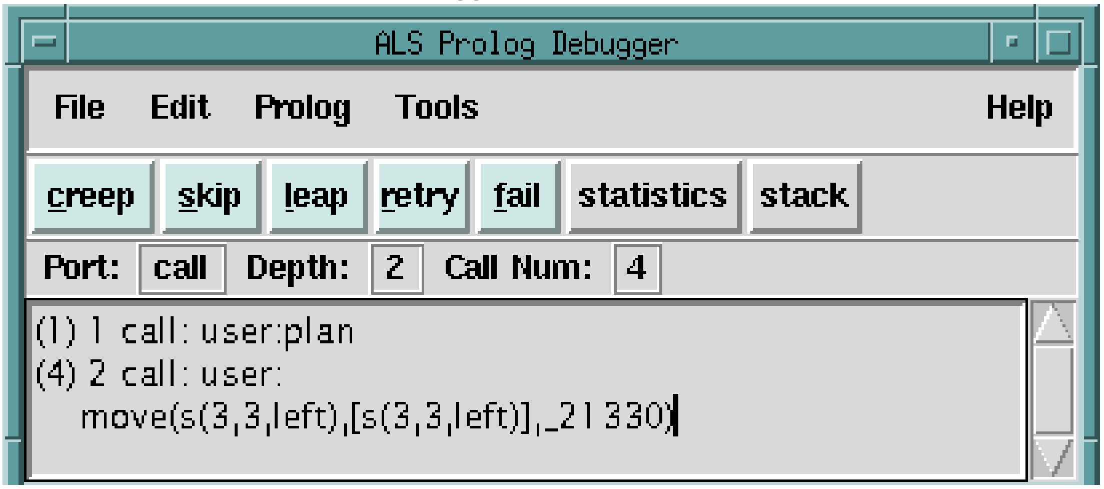


```
(1) 1 call: user:plan
```



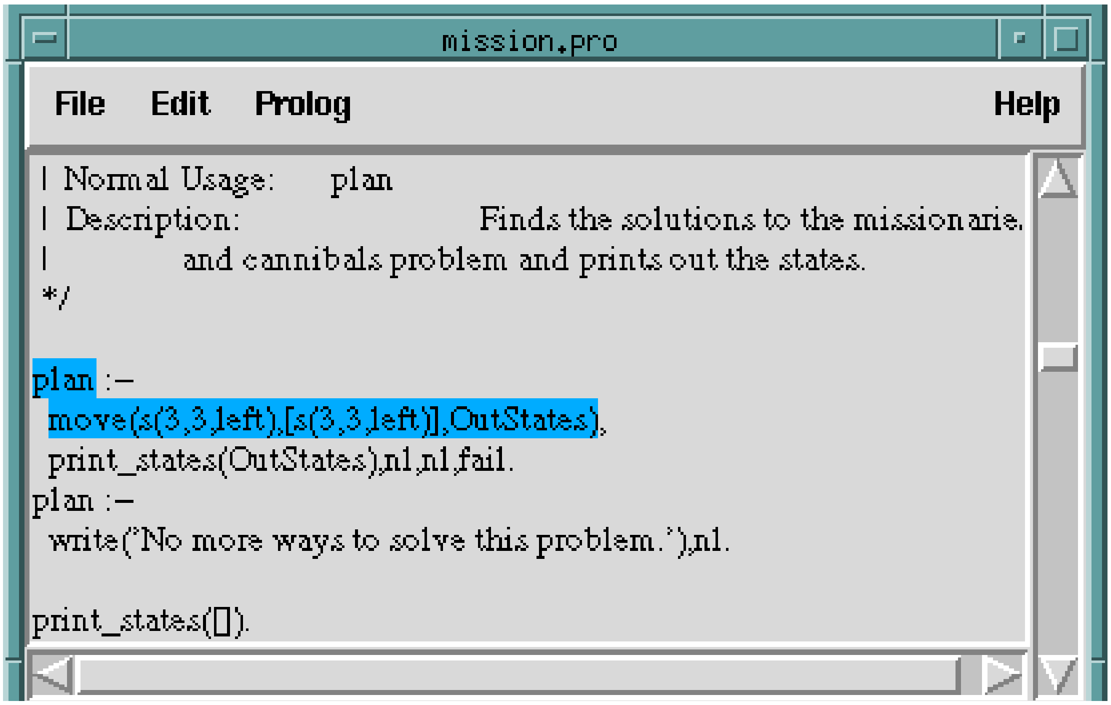
```
/*=====================================================================
|            mission.pro
|  Copyright (c) 1986, 1988 by Applied Logic Systems
|
| A solution to the missionaries and cannibals problem, from
|            Problem 64, Page 75, in:
| How to Solve it in Prolog,
|            Coelho, Cotta, Pereira, Lisbon, 1980
|
| Author:  Kevin A. Buettner
| Creation:  12/03/86
*=====================================================================
```

Click once on creep on the debugger window, and we see:



Port: call   Depth: 2   Call Num: 4

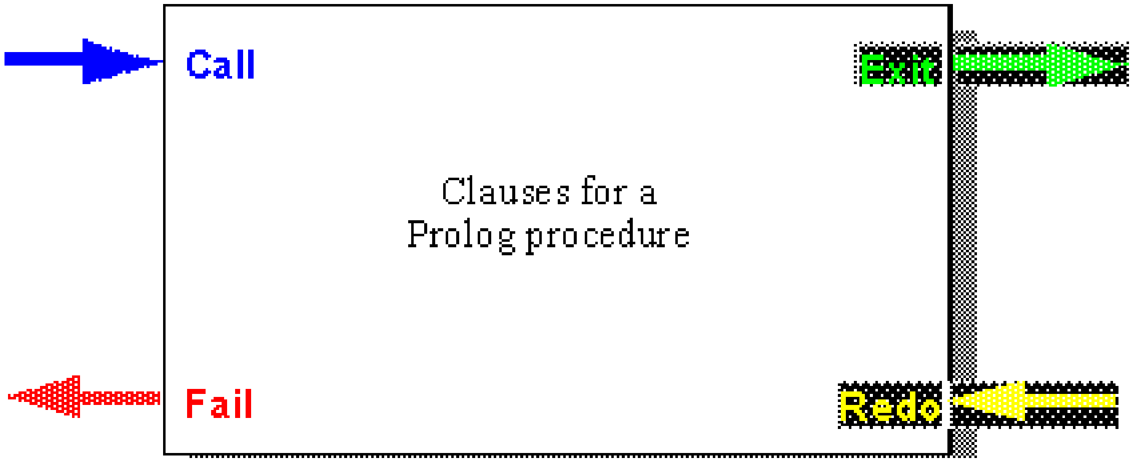
```
(1) 1 call: user:plan
(4) 2 call: user:
   move(s(3,3,left),[s(3,3,left)],_21330)
```



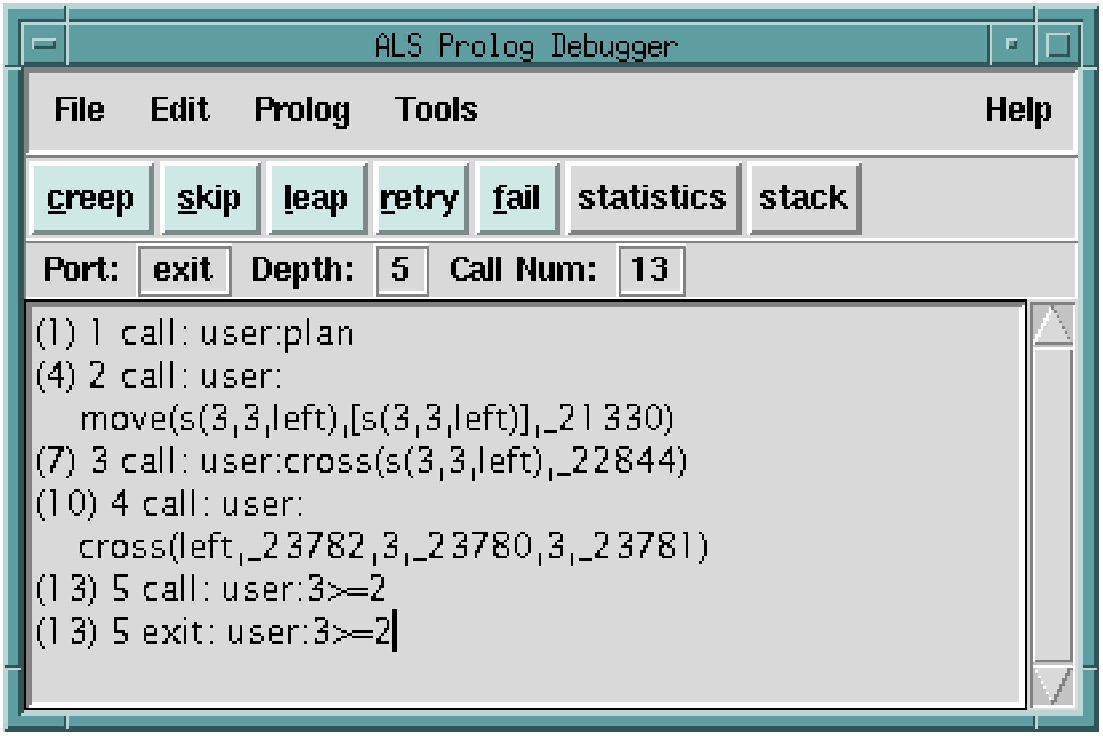
```
| Normal Usage:     plan
| Description:                    Finds the solutions to the missionarie
|          and cannibals problem and prints out the states.
*/

plan :-
 move(s(3,3,left),[s(3,3,left)],OutStates),
 print_states(OutStates),nl,nl,fail.
plan :-
 write('No more ways to solve this problem.'),nl.

print_states([]).
```

The call `move(...)` in the source code window which corresponds to the current call in the 4-port debugger window is highlighted in blue. In addition, the head of the clause in which the current call occurs is also highlighted in blue. Note that the window also automatically scrolled so that this code is now visible.

The coloring used in the source code window corresponds to the port colors shown in the four-port model diagram Figure 14 (*Generic Procedure Box.*), repeated here:

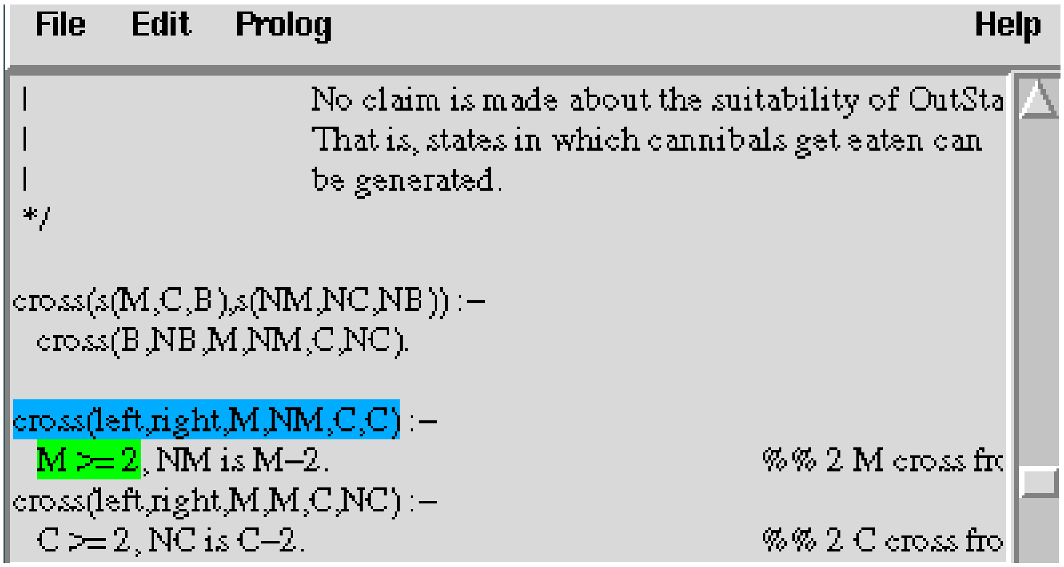


Click on creep four more times, and the situation is now:



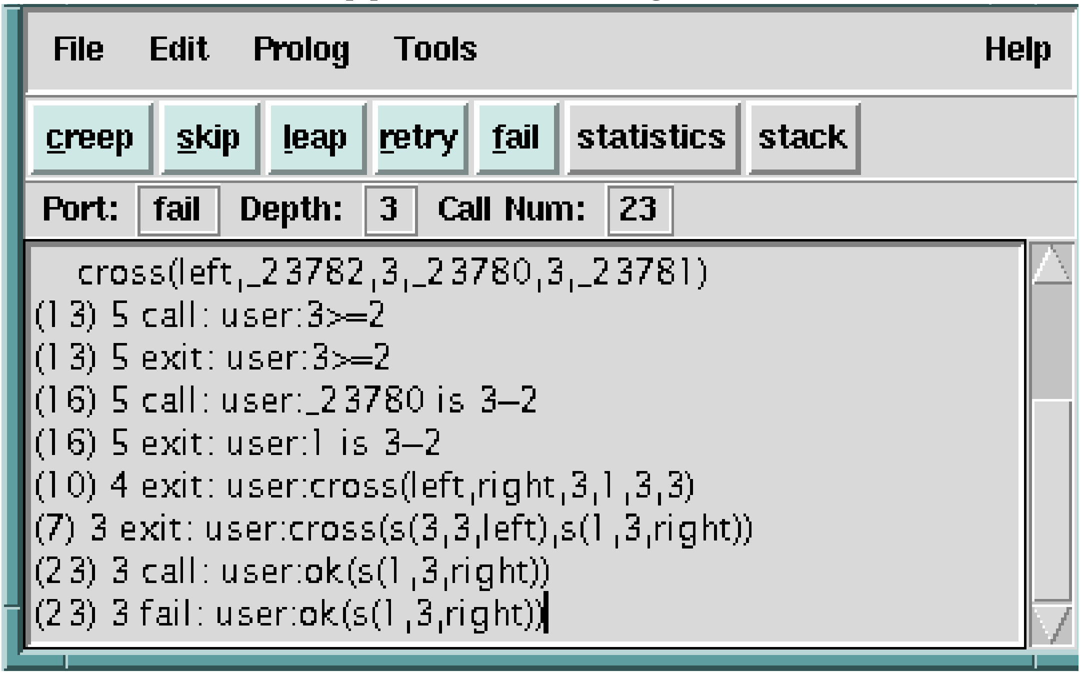
Port: exit Depth: 5 Call Num: 13

```
(1) 1 call: user:plan
(4) 2 call: user:
    move(s(3,3,left),[s(3,3,left)],_21330)
(7) 3 call: user:cross(s(3,3,left),_22844)
(10) 4 call: user:
    cross(left,_23782,3,_23780,3,_23781)
(13) 5 call: user:3>=2
(13) 5 exit: user:3>=2
```



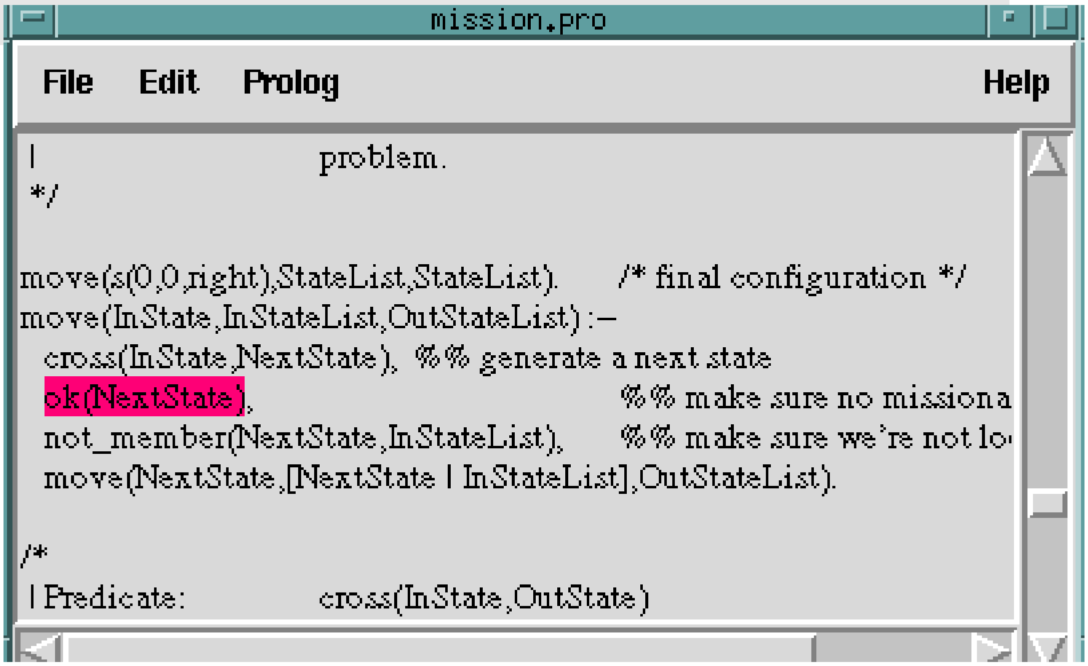
```
|                    No claim is made about the suitability of OutSta
|                    That is, states in which cannibals get eaten can
|                    be generated.
*/

cross(s(M,C,B),s(NM,NC,NB)) :-
  cross(B,NB,M,NM,C,NC).

cross(left,right,M,NM,C,C) :-
  M >= 2, NM is M-2.                           %% 2 M cross fr
cross(left,right,M,M,C,NC) :-
  C >= 2, NC is C-2.                           %% 2 C cross fro
```

Seven more clicks on creep produces the following:



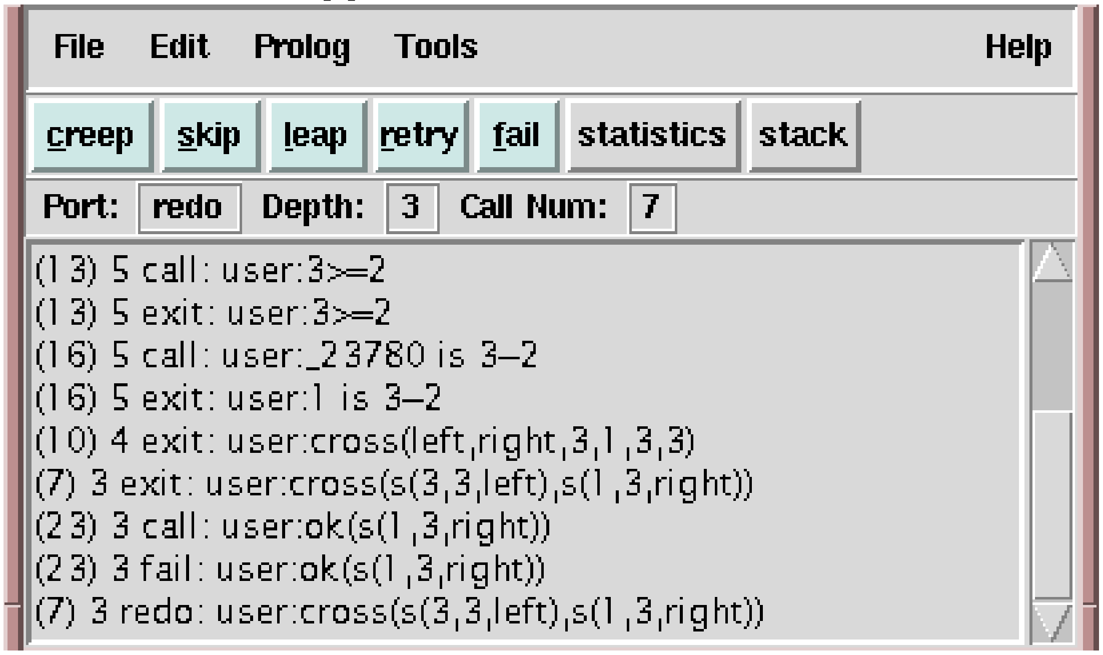
Port: fail Depth: 3 Call Num: 23

```
   cross(left,_23782,3,_23780,3,_23781)
(13) 5 call: user:3>=2
(13) 5 exit: user:3>=2
(16) 5 call: user:_23780 is 3-2
(16) 5 exit: user:1 is 3-2
(10) 4 exit: user:cross(left,right,3,1,3,3)
(7) 3 exit: user:cross(s(3,3,left),s(1,3,right))
(23) 3 call: user:ok(s(1,3,right))
(23) 3 fail: user:ok(s(1,3,right))
```

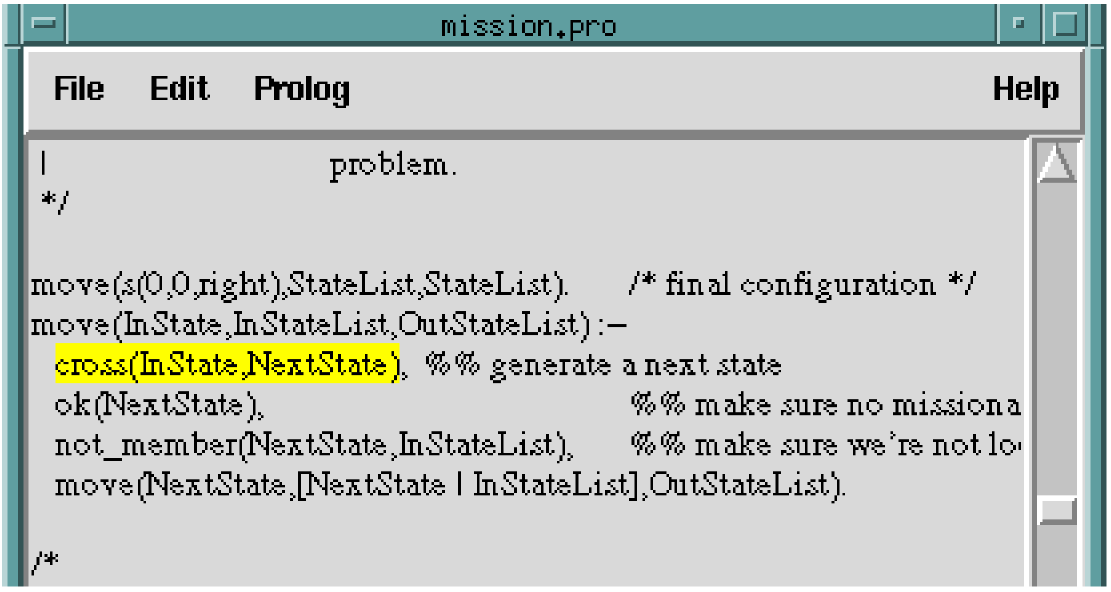


```
|                          problem.
*/

move(s(0,0,right),StateList,StateList).     /* final configuration */
move(InState,InStateList,OutStateList) :-
  cross(InState,NextState),  %% generate a next state
  ok(NextState),                            %% make sure no missiona
  not_member(NextState,InStateList),        %% make sure we're not lo
  move(NextState,[NextState | InStateList],OutStateList).

/*
| Predicate:              cross(InState,OutState)
```

One more click on creep produces:


File Edit Prolog Tools Help
creep skip leap retry fail statistics stack
Port: redo Depth: 3 Call Num: 7

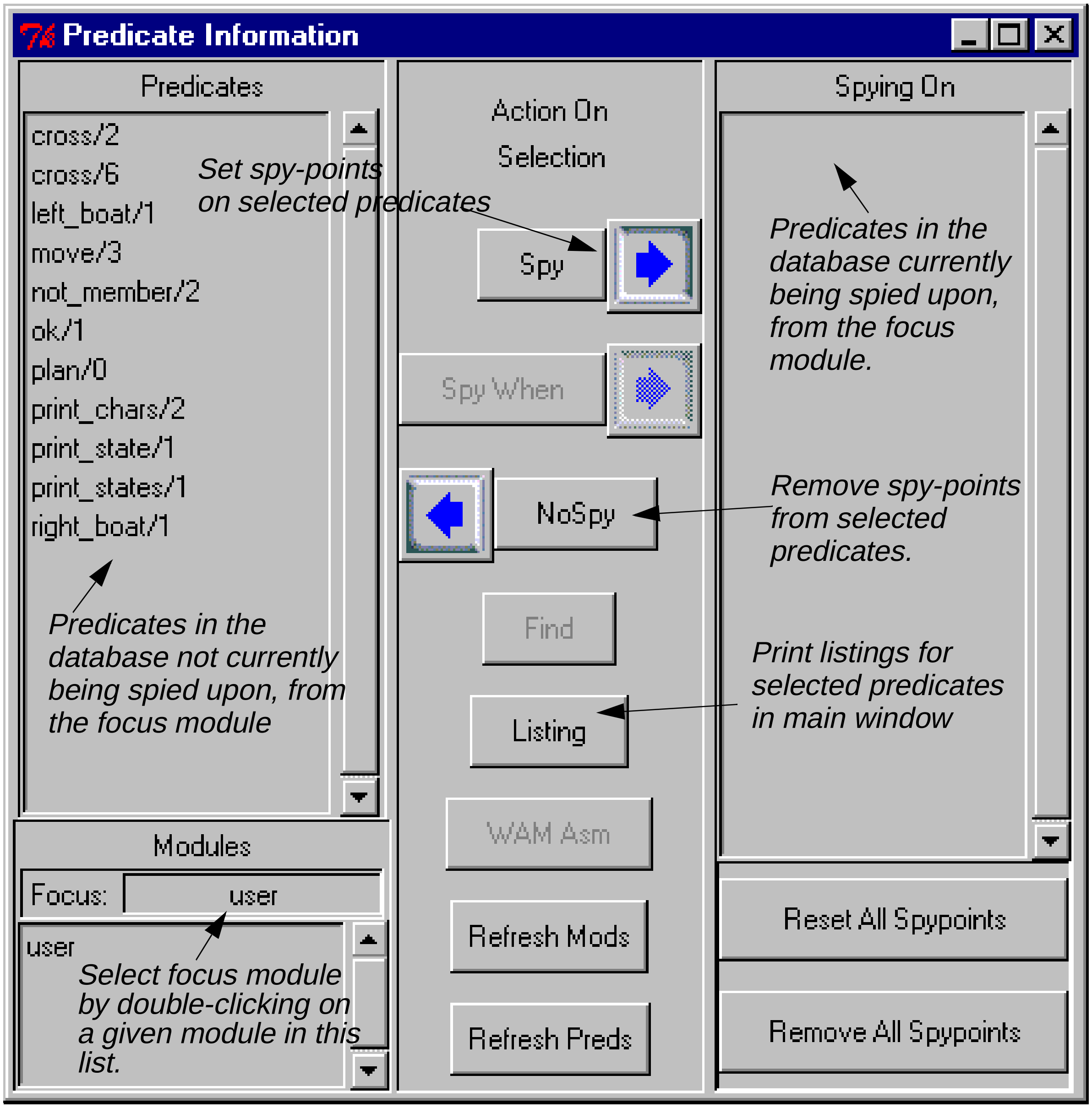
```
(13) 5 call: user:3>=2
(13) 5 exit: user:3>=2
(16) 5 call: user:_23780 is 3-2
(16) 5 exit: user:1 is 3-2
(10) 4 exit: user:cross(left,right,3,1,3,3)
(7) 3 exit: user:cross(s(3,3,left),s(1,3,right))
(23) 3 call: user:ok(s(1,3,right))
(23) 3 fail: user:ok(s(1,3,right))
(7) 3 redo: user:cross(s(3,3,left),s(1,3,right))
```



```
|                          problem.
 */

move(s(0,0,right),StateList,StateList).      /* final configuration */
move(InState,InStateList,OutStateList) :-
  cross(InState,NextState),  %% generate a next state
  ok(NextState),                            %% make sure no missiona
  not_member(NextState,InStateList),        %% make sure we're not lo
  move(NextState,[NextState | InStateList],OutStateList).

/*
```

5.3 Spying with the ALS IDE

Choosing Spy... from the Debugger Tools menu raises the following dialog:


Predicate Information
Predicates
cross/2
cross/6
left_boat/1
move/3
not_member/2
ok/1
plan/0
print_chars/2
print_state/1
print_states/1
right_boat/1
Set spy-points on selected predicates
Predicates in the database not currently being spied upon, from the focus module
Modules
Focus:
user
user
Select focus module by double-clicking on a given module in this list.
Action On Selection
Spy
Spy When
NoSpy
Find
Listing
WAM Asm
Refresh Mods
Refresh Preds
Spying On
Predicates in the database currently being spied upon, from the focus module.
Remove spy-points from selected predicates.
Print listings for selected predicates in main window
Reset All Spypoints
Remove All Spypoints

All modules currently known are shown in the listbox in the lower left corner. One selects a focus module by double-clicking on one of the elements of this list. All of the predicates defined in the focus module are shown initially in the left-hand large listbox headed Predicates. Occasionally, when files have been reconsulted, one must use the Refresh Mods and Refresh Preds buttons to update these lists.

Selecting one or more predicates in the Predicates listbox, and then clicking on the Spy (or the right-pointing blue arrow) button will cause spy points to be placed on each of the selected predicates. Each of the selected predicates will also be moved from the left listbox to the right listbox labelled Spying On. Double-clicking on a single predicate from the left column will also cause it to be spied upon and moved to the right column. Similarly, spy points can be removed by selecting one or more predicates from the right column and clicking on the No Spy (or the left-pointing blue arrow) button. Double-clicking a single predicate in the right column also removes its spy-point.

If one or more predicates have been selected, clicking on Listing will cause their definitions to be printed in the main listener window.

Various actions, such as consulting, can cause spy-points to be disabled. The Reset All Spypoints causes all current spy-points to be re-activated. The Remove All Spypoints button causes all current spypoints to be removed.

6 Using the GUI Library

The ALS library includes a growing collection of routines designed to make it easy to utilize various GUI constructs easly from ALS Prolog.

6.1 Initializing the GUI library.

In order to make use of these routines, one must first initialize the library. This is accomplished with the predicate `init_tk_alslib/0.` This initializes a Tcl/Tk interpreter named `tcli` (see the next Chapter), and sources (loads) the associated Tcl/Tk code into that interpreter. This call is really defined as

```
init_tk_alslib :- init_tk_alslib(tcli, _).
```

If `Interp` is an atom intended to name a Tcl/Tk interpreter, then

```
init_tk_alslib(Interp, Path)
```

creates a Tcl/Tk interpreter named `Interp`, locates the adjunct TclTk code, returns the path to the directory containing that code in `Path`, and sources that code into `Interp`.

All of the calls in the library are organized in a similar style: there is a default version which references the default interpreter `tcli`, and there is a general version allowing one to use the same functionality with any other interpreter.

6.2 Dialogs.

6.2.1 Information dialogs.

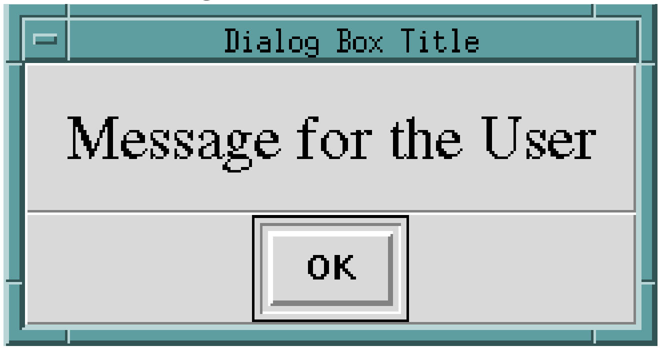
```
 info_dialog(Msg) :-
    info_dialog(Msg, ‘Info’).
 info_dialog(Msg, Title) :-
    info_dialog(tcli, Msg, Title).
```

`info_dialog(Interp, Msg, Title)`

The call

```
?-info_dialog(‘Message for the User’,
            ‘Dialog Box Title’).
```

produces the information dialog:



Message for the User

6.2.2 Yes-no dialogs.

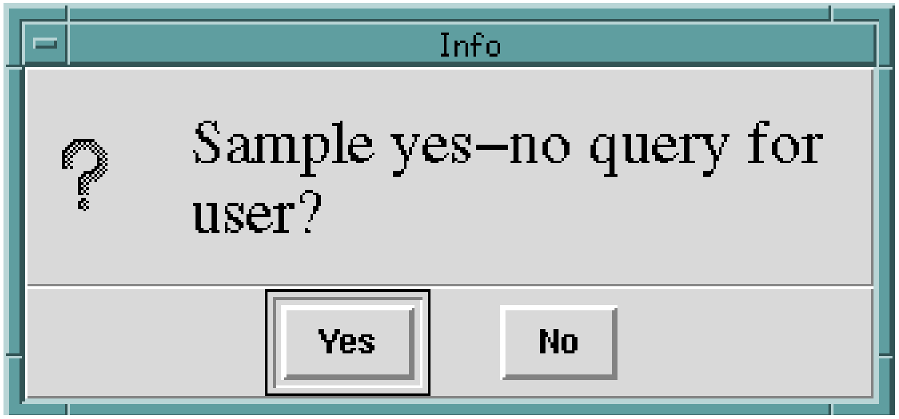
```
yes_no_dialog(Msg, Answer)
    :-
    yes_no_dialog(Msg, ‘Info’, Answer).
yes_no_dialog(Msg, Title, Answer)
    :-
    yes_no_dialog(tcli, Msg, Title, Answer).
yes_no_dialog(Interp, Msg, Title, Answer)
    :-
    yes_no_dialog(Interp, Msg, Title, ‘Yes’, ‘No’, Answer).
```

`yes_no_dialog(Interp, Msg, Title, YesLabel, NoLabel, Answer)`

The call

```
?- yes_no_dialog(‘Sample yes-no query for user?’,
                 Answer).
```

produces the following popup dialog:



Sample yes–no query for user?

If the user clicks “Yes”, the result is

```
   Answer = Yes
```

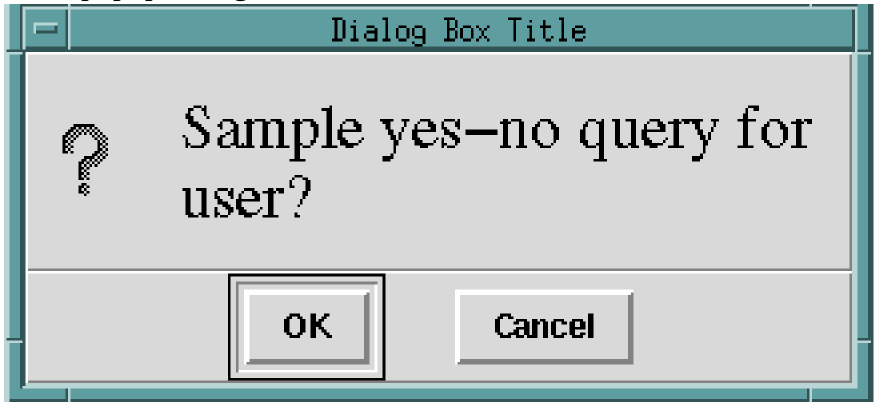
while clicking “No” yields

```
   Answer = No.
```

The call

```
   yes_no_dialog(tcli,
                   ‘Sample yes-no query for user?’,
                   ‘Dialog Box Title’,
                   ‘OK’, ‘Cancel’, Answer).
```

produces the popup dialog



Sample yes–no query for user?

Clicking “OK” yields

```
   Answer = OK
```

while clicking “Cancel” yeilds

```
   Answer = Cancel.
```

6.3 Choices from lists.

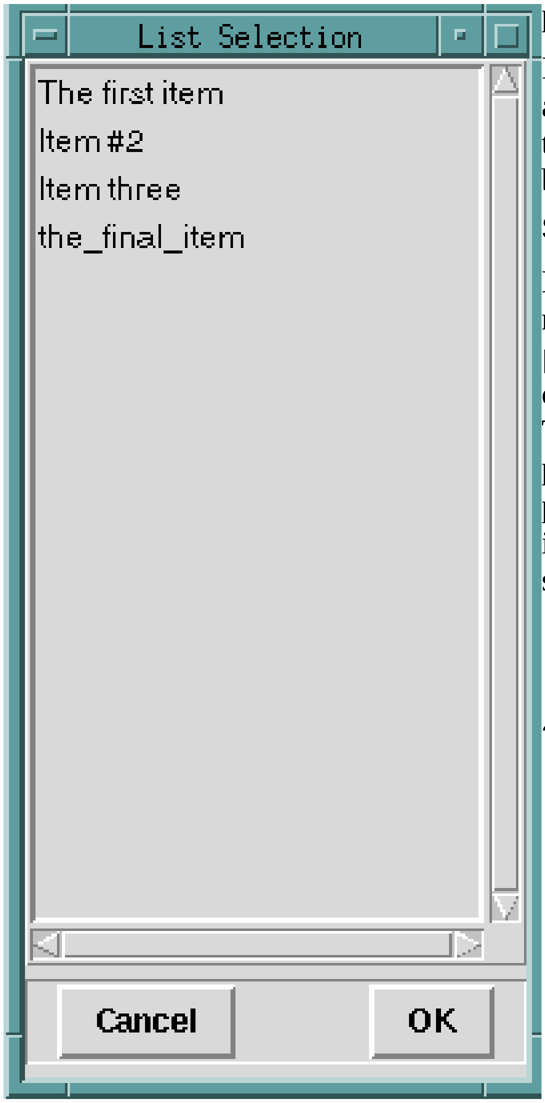
popup_select_items(SourceList, ChoiceList)

popup_select_items(SourceList, Options, ChoiceList)

popup_select_items(Interp, SourceList, Options, ChoiceList)

The call

```
   ?- popup_select_items(
          [‘The first item’,’Item #2’,
           ‘Item three’, the_final_item],
        Selection).
```



The first item
Item #2
Item three
the_final_item

produces the popup shown at the left.:

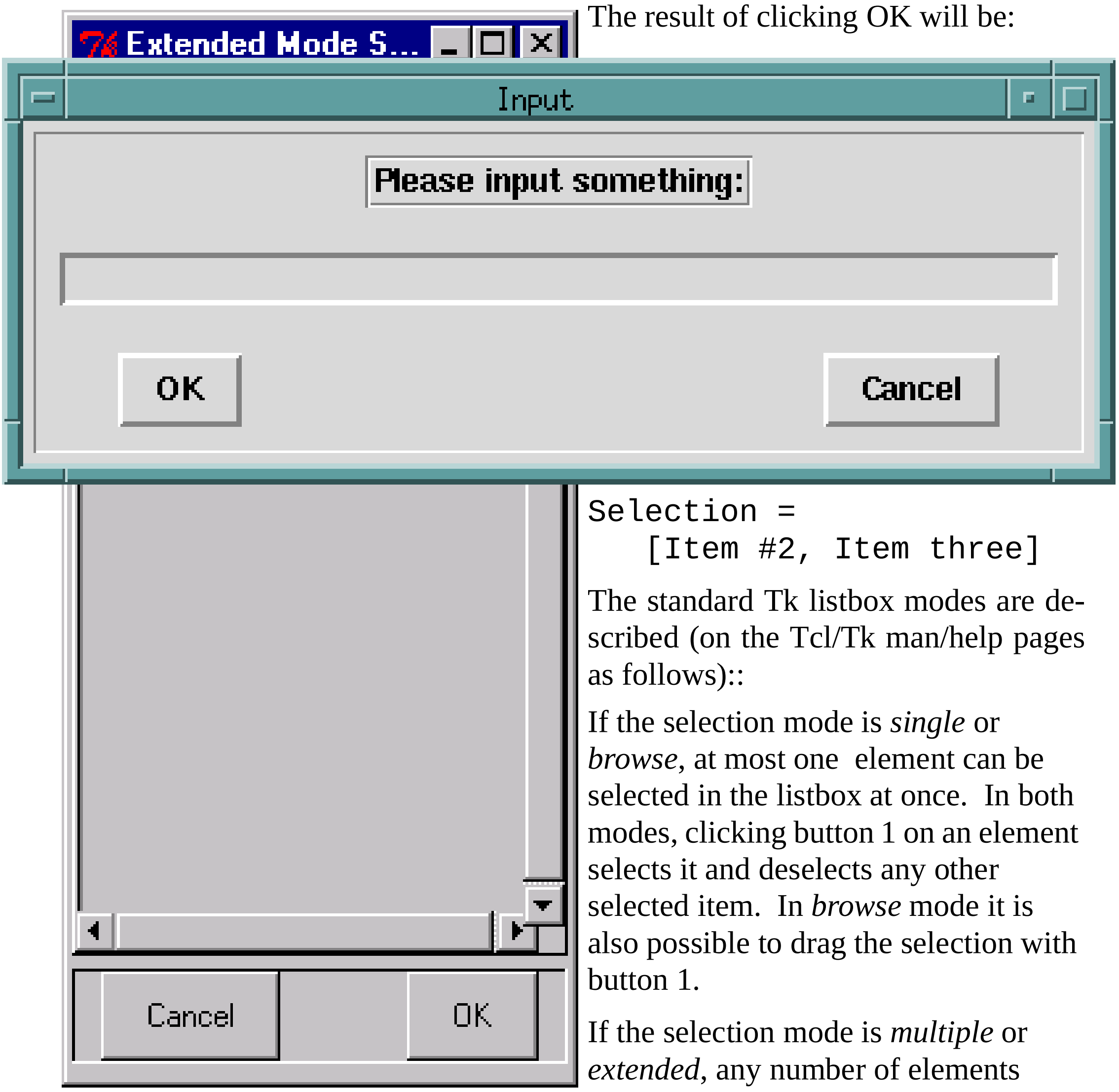
In this case, the user is allowed to select a single item; if the user selected "Item three" and clicked OK, the result would be:

```
Selection = [Item three].
```

Even though in this case the user was restricted to selection of one item, the `popup_select_items/_` predicate returns a list of the selected items. The `Options` argument for popup_select_items/[3,4] allows the programmer to place the popup list box in any of the other standard Tk listbox selection modes. For example, the call

```
?- popup_select_items(
  [‘The first item’,
   ’Item #2’,
   ‘Item three’,
   the_final_item],
  [mode=extended,
   title=
’Extended Mode Selection’
  ],
   Selection).
```

will popup a list box whose appearance is identical (apart from the different title requested) to the previous listbox. However, it will permit selection of ranges of elements, as seen below:



The result of clicking OK will be:

```
Selection =
   [Item #2, Item three]
```

The standard Tk listbox modes are described (on the Tcl/Tk man/help pages as follows)::

If the selection mode is *single* or *browse*, at most one element can be selected in the listbox at once. In both modes, clicking button 1 on an element selects it and deselects any other selected item. In *browse* mode it is also possible to drag the selection with button 1.

If the selection mode is *multiple* or *extended*, any number of elements may be selected at once, including discontiguous ranges. In *multiple* mode, clicking button 1 on an element toggles its selection state without affecting any other elements. In *extended* mode, pressing button 1 on an element selects it, deselects everything else, and sets the anchor to the element under the mouse; dragging the mouse with button

1 down extends the selection to include all the ele ments between the anchor and the element under the mouse, inclusive.

6.4 Inputting atoms (answering questions).

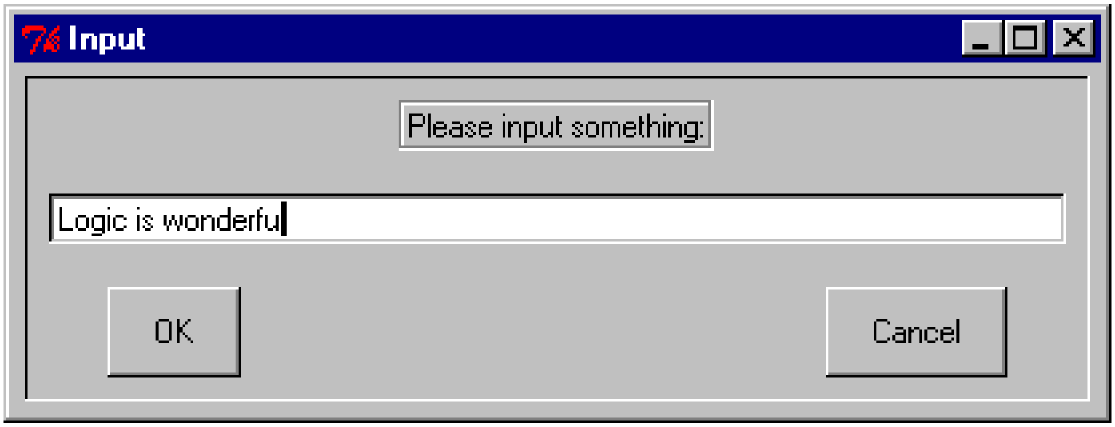
```
atomic_input_dialog(Msg, Atom)
    :-
    atomic_input_dialog(Msg, ‘Input’, Atom).
atomic_input_dialog(Msg, Title, Atom)
    :-
    atomic_input_dialog(tcli, Msg, Title, Atom).
```

`atomic_input_dialog(Interp, Msg, Title, Atom)`

This is a useful method of obtaining input from users. For example, the call

```
?- atomic_input_dialog(‘Please input something:’,
  Atom).
```

will popup the following window:



If the user types

Logic is wonderful

then the result would be

```
Atom = Logic is wonderful
```

6.5 File selection dialogs

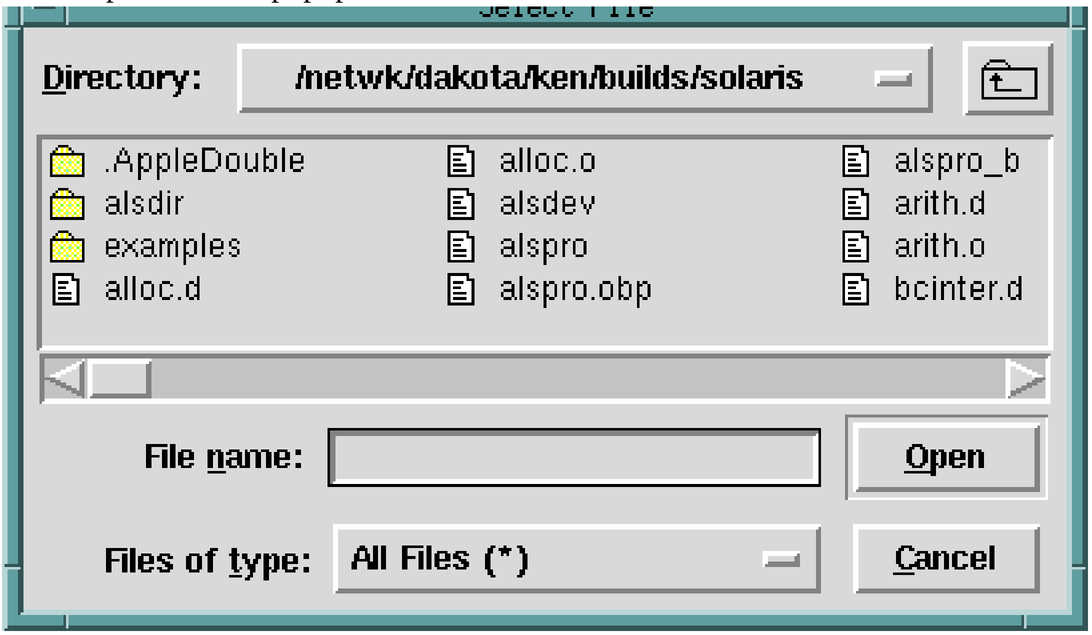
```
file_select_dialog(FileName)
    :-
    file_select_dialog(tcli, [title='Select File'],
  FileName).
file_select_dialog(Options, FileName)
    :-
    file_select_dialog(tcli, Options, FileName).
```

`file_select_dialog(Interp, Options, FileName)`

The call

```
?- file_select_dialog(File).
```

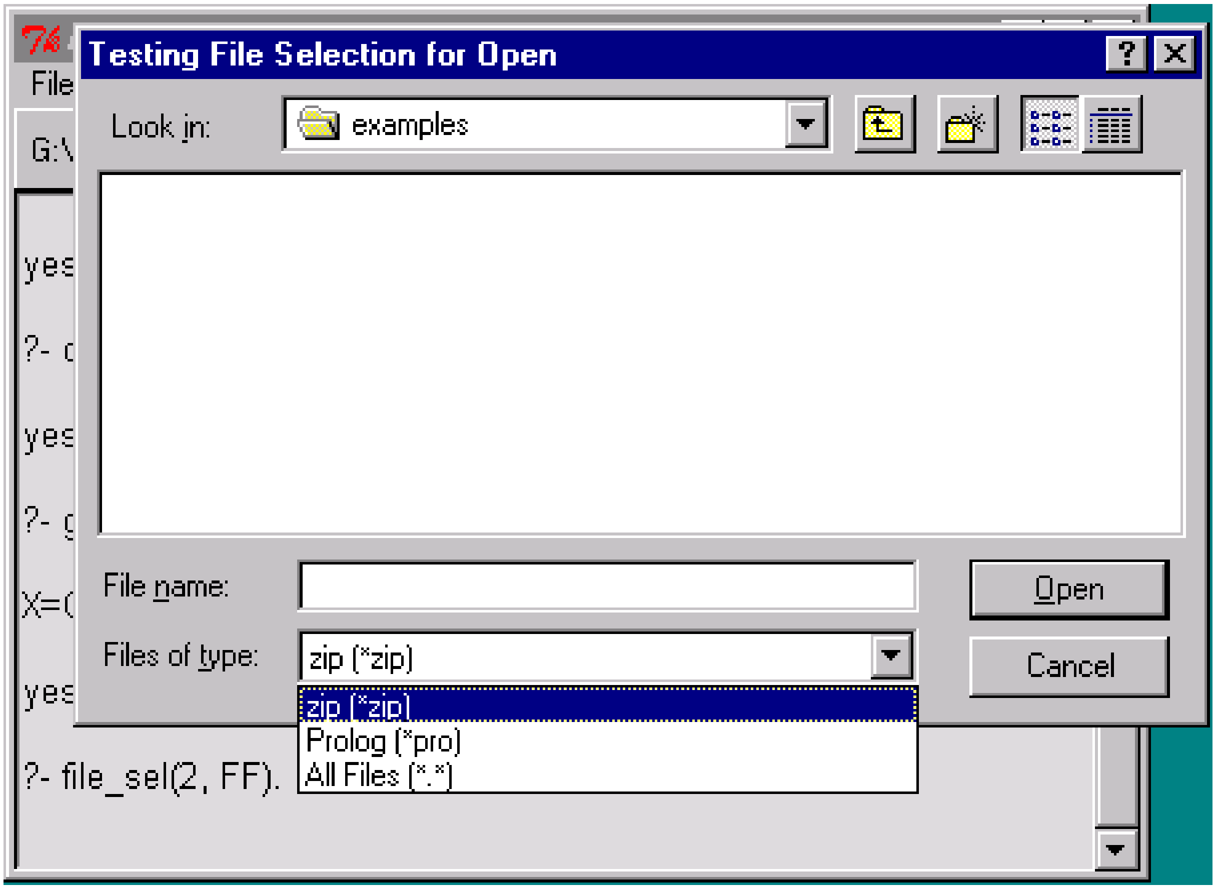
would produce this popup:



The call

```
?- file_select_dialog(
        [title='Testing File Selection for Open',
         filetypes= [[zip,[zip]],
                     ['Prolog',['pro']],
                     ['All Files',['*']] ]
         ],   File).
```

would produce



6.6 Displaying Images.

These routines provide simple access from ALS Prolog to the image routines of Tk. They will be extended. The current versions support gif images, but the routines can be extended to any of the types Tk supports. To display images, one must specify a path to the image file, and must first produce an internal Tk form of the image. This is done with:

```
    create_image(ImagePath, ImageName)
```

```
        :-
        create_image(tcli, ImagePath, ImageName).
```
`create_image(Interp, ImagePath, ImageName)`

Assume that

pow_wow_dance.gif

is a file in the current directory. Then the call

?- create_image(‘pow_wow_dance.gif’, pow_wow).

will create the internal form of this image and associate the name pow_wow with it. Display of images which have been created is accomplished with:

```
display_image(ImageName)
        :-
        display_image(tcli, ImageName, []).
display_image(Interp, ImageName, Options)
```

Thus, the call

?-display_image(pow_wow).

produces

6.7 Adding to the ALS IDE main menubar.

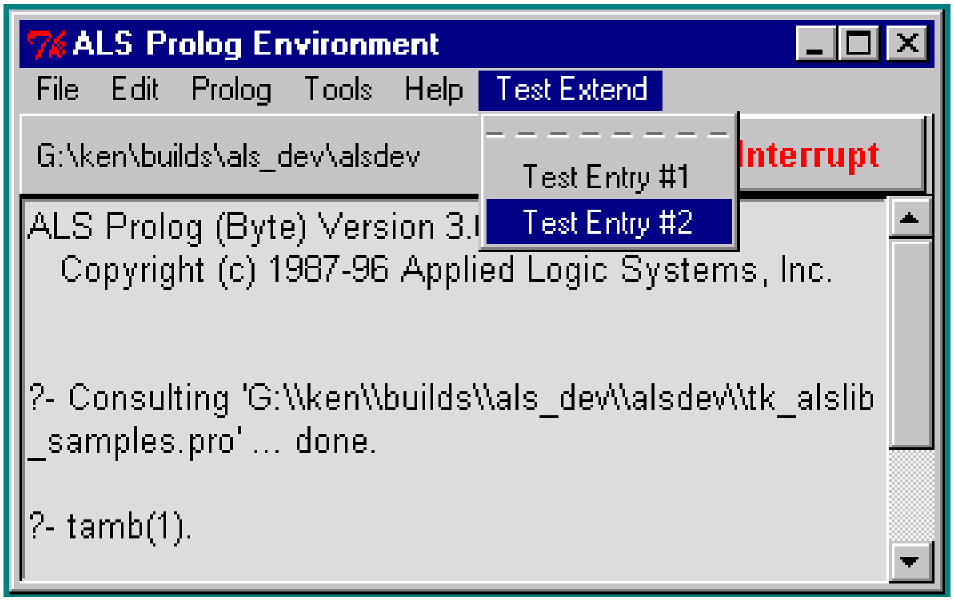
Simple additions to the main menubar are often useful. The general call to accomplish this is:

```
    extend_main_menubar(Label, MenuEntriesList)
```

`Label` should be the label which will appear on the menu bar, and `MenuEntriesList` is a Prolog list describing the menu entries. The simplest list consists only of labels which will occur on the pull-down menu. Thus, after executing the call

```
    ?- extend_main_menubar('Test Extend',
                [‘Test Entry #1’, ‘Test Entry #2’]).
```

the main listener window would look like this when clicking on the newly added menubar entry:



```
G:\ken\builds\als_dev\alsdev
ALS Prolog (Byte) Version 3.I
  Copyright (c) 1987-96 Applied Logic Systems, Inc.


?- Consulting 'G:\\ken\\builds\\als_dev\\alsdev\\tk_alslib
_samples.pro' ... done.

?- tamb(1).
```

This is somewhat uninteresting, however, since these menu items won’t do anything. To add simple behaviors to the menu entry, one uses expressions of the form

Label + Behnavior

where Label is the menu label which will appear, and Behavior is either a ground Prolog term, or is a prolog term of the form

tcl(Expr)

where Expr is a quoted atom describing a Tcl/Tk function call. Thus, if we replace the call considered above by the following,

```
?- extend_main_menubar(‘Test Extend’,
            [‘Test Entry #1’ + tcl(‘bell’),
             ‘Test Entry #2’ + test_write
            ]),
```

where

```
test_write
    :-
    printf(user_output, ‘This is a test ...\n’, []),
    flush_input(user_input).
```

then the appearance of the main menu and the new pulldown will be the same, but chooseing Test Entry #1 will cause the bell to ring, and choosing Test Entry #2 will cause

```
This is a test ...
```

to be written on the listener console.

To summarize thus far, the main menu bar can be extended by calling:

```
extend_main_menubar(Label, MenuEntriesList)
```

where:

- `Label` is an atom (suitable for extending a Tk window path);
- `MenuEntriesList` is a list of menu entry descriptors.

And a *menu entry descriptor* is either an `Atom` alone, or an expression of the form

```
Atom + Expr
```

where `Atom` is a prolog atom which will serve as the new menu entry, and `Expr` is a *menu entry action expression,* which can be one of the following:

- `tcl(TclExpr)`
- `cascade(SubLabel, SubList)`
- `PrologCall`

Here, `TclExpr` can be any Tcl/Tk expression for evaulation, and `PrologCall` is any ground Prolog goal. The new entry

```
cascade(SubLabel, SubList)
```

allows one to create menu entries which are themselves cascades. In this case, `SubLabel` must be an atom which will serve as the entry's label, and `SubList` is (recursively) a list of menu entry descriptors.

Here are several useful predicates for working with menus and menu entries:

```
menu_entries_list(MenuPath, EntriesList)
  :-
  menu_entries_list(tcli, MenuPath, EntriesList).
```
`menu_entries_list(Interp, MenuPath, EntriesList)`

If `MenuPath` is a Tk path to a menu (top level or subsidiary), then `EntriesList` will be the list of labels for the entries on that menu, in order. For example,

```
?- menu_entries_list(shl_tcli,
            '.topals.mmenb', EntriesList).
EntriesList = [File, Edit, Prolog, Tools, Help].
```

When one indexes menu entries, the indicies are integers beginning at 0.

```
path_to_main_menu_entry(Index, SubMenuPath)
    :-
    path_to_menu_entry(shl_tcli, '.topals.mmenb',
                                Index, SubMenuPath).

path_to_menu_entry(MenuPath, Index, SubMenuPath)
    :-
```

```
        path_to_menu_entry(tcli, MenuPath, Index, SubMenuPath).
```

`path_to_menu_entry(Interp, MenuPath, Index, SubMenuPath)`

If `MenuPath` is a Tk path to a menu (top level or subsidiary), and if `Index` is an integer >= 0, and if the `Index'th` entry of `MenuPath` is a cascade, so that it has an associated menu, then `SubMenuPath` is a path to that associated menu. Thus,

```
?- path_to_main_menu_entry(4, SubMenuPath).
SubMenuPath = .topals.mmenb.help
```

Finally, one can add new entries at the ends (bottoms) of existing menu cascades, as follows:

```
add_to_main_menu_entry(Index, Entry)
    :-
    path_to_main_menu_entry(Index, MenuPath),
    extend_cascade(Entry, MenuPath, shl_tcli).
```

`extend_cascade(Entry, MenuPath, Interp)`

For example,

```
?- add_to_main_menu_entry(3,,
                 'My Entry' + test_write).
```

will add an entry at the end of the Tools cascade. The predicate

```
extend_cascade(Entry, MenuPath, Interp)
```

accomplishes adding the `Entry` to the end of menu `MenuPath` under interpreter `Interp`.

7 ALS Prolog - TCL/TK Interface

7.1 Introduction and Overview

The interface between ALS Prolog and Tcl/Tk allows prolog programs to create, manipulate and destroy Tcl/Tk interpreters, to submit Tcl/Tk expressions for evaulation in those interpreters, and to allow expressions being evaluated to make calls back into Prolog. Computed data can be passed in both directions:

- A Tcl/Tk function called from Prolog can return a value to Prolog;
- A Prolog goal called from Tcl/Tk can bind Tcl variables to computed values.

The conversions between the datatypes of the two languages are described in the next section.

7.2 Prolog to Tcl Type Conversion

7.3 Prolog to Tcl Interface Predicates

7.3.1 tcl_new(?Interpreter)
tk_new(?Interpreter)

`tcl_new/1` creates a new Tcl interpreter. If the `Interpreter` argument is an uninstantiated (Prolog) variable, then a unique name is generated for the interpreter. If `Interpreter` is a atom, the new Tcl interpreter is given that name.

`tk_new/1` functions in the same manner as `tcl_new/1`, except that the newly-created Tcl interpreter is initialized with theTk package.

Examples

tcl_new(i). Succeeds, creating a Tcl interpreter named i.

tcl_new(X). Succeeds, unifying X with the atom ‘interp1’.

Errors

- Interpreter is not an atom or variable.

  type_error(atom_or_variable).
- The atom Interpreter has already been use as the name of a Tcl interpreter.

  permission_error(create,tcl_interpreter,Interpreter)
- Tcl is unable to create the interpreter.

  tcl_error(message)

7.3.2 tcl_call(+Interpreter, +Script, ?Result)
tcl_eval(+Interpreter, +ArgList, ?Result)

Tcl_call and Tcl_eval both execute a script using the Tcl interpreter and returns the Tcl result in Result. Tcl_call passes the Script argument as a single argument toTcl's eval command. Tcl_eval passes the elements of ArgList as arguments to the Tcl's eval command, which concatenates the arguments before evalating them.

Tcl_call's Script can take the following form:

- List - The list is converted to a Tcl list and evaluated by the Tcl interpreter. The list may contain, atoms, numbers and lists.
- Atom - The atom is converted to a string and evaluated by the Tcl interpreter.

Tcl_eval's ArgList may contain atoms, numbers or lists.

Examples

tcl_call(i, [puts, abc], R). Prints 'abc' to standard output, and bind R to ''.

tcl_call(i, [set, x, 3.4], R). Sets the Tcl variable x to 3.4 and binds R to 3.4.

tcl_call(i, 'set x', R). Binds R to 3.4.

tcl_eval(i, ['if [file exists ', Name, '] puts file-found'], R).

Errors

- Interpreter is not an atom.
- Script is not an atom or list.
- Script generates a Tcl error.

  tcl_error(message)

7.3.3 tcl_coerce_number(+Interpreter, +Object, ?Number) tcl_coerce_atom(+Interpreter, +Object, ?Atom) tcl_coerce_list(+interpreter, +Object, ?List)

These three predicates convert the object Object to a specific Prolog type using the Tcl interpreter Interpreter. Object can be an number, atom or list. If the object is already the correct type, then it is simple bound to the output argument. If the object cannot be converted, an error is generated.

Examples

tcl_coerce_number(i, ‘ 1.3’, N) Succeeds, binding N to the float 1.3

tcl_coerce_number(i, 1.3, N) Succeeds, binding N to the float 1.3

tcl_coerce_number(i, ‘abc’, N) Generates an error.

tcl_coerce_atom(i, [a, b, c], A) Succeeds, binding A to ‘a b c’

tcl_coerce_atom(i, 1.4, A) Succeeds, binding A to ‘1.4’

tcl_coerce_list(i, ‘a b c’, L) Succeeds, binding L to [a, b, c]

tcl_coerce_list(i, 1.4, L) Succeeds, binding L to [1.4]

tcl_coerce_list(i, ‘’, L) Succeeds, binding L to []

Errors

- Interpreter is not an atom.
- Object is not a number, atom or list.
- Object cannot be converted to the type.

tcl_error(message)

7.3.4 tcl_delete(+Interpreter) tcl_delete_all

Tcl_delete deletes the interpreter name Interpreter.

Tcl_delete_all deletes all Tcl interpreters created by tcl_new/1.

7.4 Tcl Prolog Interface

7.4.1 prolog - call a prolog term

Synopsis

prolog *option ?arg arg... ?*

Description

The prolog command provides methods for executing a prolog query in ALS Prolog. Option indicates how the query is expressed. the valid options are:

prolog call *module predicate ?-type arg ...?*

Directly calls a predicate in a module with type-converted arguments. The command returns 1 if the query succeeds, or 0 if it fails. The arguments can take the following forms:

-number arg

Passes arg as an integer or floating point number.

-atom arg

Passes arg as an atom.

-list arg

Passes arg as a list.

-var varName

Passes an unbound Prolog varaible. When the Prolog variable is bound, the Tcl variable with the name varName is set to the binding.

prolog read_call *termString ?varName ...?*

The string termString is first read as a prolog term and then called. The command returns 1 if the query succeeds, or 0 if it fails. The optional variables named by the varName arguments are set when a Prolog variable in the query string is bound. The prolog variables are matched to varNames in left-to-right depth first order.

Examples

prolog call builtins append -atom a -atom b -var x

Returns 1, and the Tcl variable x is set to {a b}.

prolog read_call “append(a, b, X)” x

Returns 1, and the Tcl variable x is set to {a b}.

7.5 Stand-Alone TCL

Normally Tcl/Tk is installed in a system independent of ALS Prolog. Typically the Tcl/Tk shared/dynamic libraries are stored in a system directory (/usr/local/lib on Unix, \winnt\system32 on Windows NT, and “System Folder:Extensions” on MacOS). Tcl/Tk support libraries are similarly stored in a global location.

When creating a stand alone Prolog/Tcl-Tk application, it is sometimes convienient to create a package which includes Tcl/Tk so that the application will run correctly even on systems without Tcl/Tk.

Basicly this is done by moving the Tcl/Tk shared/dynamic libraries and support libraries into the same directory as ALS Prolog. Here are sample directories for MacOS, Unix and Win32:

MacOS

- ALS Prolog
- Tcl8.0.shlb
- Tk8.0.shlb
- MWRuntimeLib
- tcl8.0 (support library folder)
- tk8.0 (support library folder)

Unix

- alspro
- libtcl8.0.so or libtcl8.0.sl
- libtk8.0.so or libtk8.0.sl
- lib (directory containing:)

  tcl8.0 (support library directory)

  tk8.0 (support library directory)

Win32

- alspro.exe
- tcl80.dll
- tk80.dll
- lib (directory containing:)

  tcl8.0 (support library directory)

  tk8.0 (support library directory)

On Solaris, unlike other Unix systems, the search path list for shared objects does not include the executable's directory. To ensure that the Tcl/Tk shared objects are found, the current directory '.' must be added to the LD_LIBRARY_PATH environment variable.

8 Packaging for Delivery

A typical complex ALS Prolog application involves the following elements:

- the ALS Prolog system
- various ALS Prolog source files
- various foreign C language source files

During development, the object versions of the foreign C language files may be dynamically loaded into a running ALS Prolog image (in the versions of ALS Prolog which support this), or may be statically linked with the ALS Prolog run-time library to create an extended ALS Prolog image. The total application under development is started by invoking either the basic ALS Prolog image or the extended image, loading the foreign C language object files if necessary, and dynamically consulting the ALS Prolog files. However, for delivery of application programs to users, it is desirable to be able to package all these elements of the program together in one single executable file. ALS Prolog provides packaging tools for achieving this goal. These tools are available on all platforms except the Macintosh.

The packaging tools are very straight-forward to use. Simply proceed through the following steps:

1. Start the image (either the basic ALS Prolog image, or an extended image);
2. Load the Prolog files constituting the application.
3. Invoke save_image/2.

Apart from the details necessary to flesh out step 3, this is all there is to it! The information necessary for step 3 is the following:

A. The name of the file in which the executable image is to be stored (e.g., my_app, or your_app.exe, etc.).

B. The name of a 0-ary Prolog predicate which is the entry point to the application (e.g., `start_my_app/0`).

C. The list of names of library files (if any) to be included in the application.

For example, suppose that the application is to be stored in the file *my_app*, that its entry point is the 0-ary predicate `start_my_app/0`, and that the list of library files which the application uses is

```
[cmdline, listutl1, listutl2, misc_db, misc_io,
  objs_run, strings]
```

Then, for step 3, simply issue the following goal:

```
?- save_image(my_app,
       [start_goal( start_my_app ),
        select_lib([cmdline, listutl1, listutl2,
                    misc_db, misc_io, objs_run,
                    strings] ) ] ).
```

You will see a number of cryptic messages, and eventually, the goal will succeed. If you exit the running prolog image, and perform a listing of your working directory, you will find a file *my_app*. Running this file is roughly equivalent to (i.e., will have the same effect as) the following ALS Prolog command-line:

```
alspro <my_app component files> -g start_my_app
```

or, if you are using an extended image,

```
my_ex_alspro <my_app component files> -g start_my_app
```

However, none of the component files need be consulted nor do the library files have to be loaded: they are all pre-packaged into the *my_app* image. This image is suitable for distribution (assuming your program `my_app` is ready for its debut to the outside world).

To explain how this works, let's assume that you are using a statically linked extended image `my_ex_alspro` as your starting point. Then, from start to finish, here is what happens.

1. You issue the command to run `my_ex_alspro` , and it is loaded and running.
2. You issue a consult to load the files making up your application `my_app`.
3. Your submit the goal

```
?- save_image( my_app,
                 [start_goal( start_my_app ),
                  select_lib([cmdline, listutl1,
   listutl2,
                              misc_db, misc_io,
   objs_run,
                               strings] ) ] ).
```

4. ALS Prolog loads the library files

```
[cmdline, listutl1, listutl2, misc_db, misc_io,
  objs_run, strings]
```

5. ALS Prolog changes the usual Prolog shell startup to run your goal `start_my_app` instead. For the curious, the definition of the 0-ary predicate `'$start'/0` at the end of the builtins file *builtins.pro* is retracted, and the following clause is asserted in module `builtins`:

```
'$start' :- start_my_app.
```

6. The entire (Prolog) code space is copied out in appropriate format to a temporary file, call it *TF.*
7. The file *my_app* is opened, and the original image `my_ex_alspro` is copied into the file *my_app,* which is left open.
8. The entire temporary file *TF* is copied onto the end of the file *my_app.*
9. A fixup to a certain global variable (call it G) is made and *my_app* is closed.

The 'copying' which is carried out is really writing the various code and data elements in the executable object file format appropriate to the given machine and operating system (e.g., a.out, coff, elf, etc.) So the final file *my_app* is really an executable file whose initial segment is the original image `my_ex_alspro`, followed by some 'extra stuff'. Moreover, the entry point for `my_app` is the original entry point for `my_ex_alspro`, which is just a version of the ALS Prolog image.

When any such image (be it plain ALS Prolog or an extended image) starts up, at a point very early in its initialization, it looks at the global variable G. In the plain ALS Prolog image, or one which has simply been statically linked with some additional C code, G = 0. But in a 'packaged application' such as `my_app`, G contains a number effectively telling the system where the end of the image `my_ex_alspro` lies, and where the 'extra stuff', the packaged Prolog code, starts. The system uses this to appropriately allocate memory, and then to load the packaged ALS Prolog code from the application, from the builtins, and from the loaded library files, into the approprate code area. (This is a block move, so it happens very quickly. The difference in the startup times for alspro_b and alspro reflects the difference between loading the builtins *.obp files from disk, and this simple block move of the builtins code. Why? Because alspro is just a packaged version of alspro_b, packaging the builtins.)

Note that the 3-step process of building the application can be combined into a single-step operating system shell command-line action:

```
my_ex_alspro <my_app component files> \
  -g save_image( my_app, [start_goal( start_my_app ),
  \
  select_lib([cmdline, listutl1, listutl2, \
  misc_db, misc_io, objs_run, strings] ) ] ).
```

Such approaches are especially suitable for use in makefiles. (Note the continuation characters '\' at the end of each line except the final one.)

Like the ALS Prolog environments themselves, some packaged applications will want to accept command line arguments. Two predicates can be used in this regard. To obtain the complete original command line, including the name of the program being run, use `pbi__get_command_line/1`. For example, if the packaged application is named foo, then issuing the following operating system command line,

```
foo zipper -f zap
```

would cause any call

```
pbi__get_command_line(X)
```

to yield the value

```
X = [foo,zipper, '-f', zap].
```

If you want the packaged version ("foo") of the application to co-ordinate with the "unpackaged version" developed under the ALS environment, you can use `setup_cmd_line/0`. If the packaged application "foo" starts with `start_my_app/0`, simply ensure that `start_my_app/0` makes a call on `setup_cmd_line/0` any time before any call is made on `command_line/1`.

9 Abstract Data Types: Structure Definition

One powerful modern programming idea is the use of abstract data types to hide the inner details of the implementation of data types. The arguments in favor of this technique are well-known (cf. [Ref: Liskov]). Of course, it is possible to use the abstract data type idea ’by hand’ as a matter of discipline when developing programs. However, like many other things, programming life becomes easier if useful tools supporting the practice are available. In particular, good tools make it easy to modify abstract data type definitions while still maintaining efficient code.

The *defStruct* tool provides such support for a common construct: the use of Prolog structures (i.e., compound terms) which must be accessed for values and may be (destructively) updated. For example, the implementation of a window system often passes around structures with many slots representing the various properties of particular windows. When programming in C, one would use a C struct for the entity. The analogue in Prolog is a flat compound term.

For speed of access to the slot values, one wants to use the arg/3 builtin. For destructively updating the slot values, one uses the companion mangle/3 builtin. The difficulty with using these builtins is that both require the slot *number* as an argument. As is well-known, hard-coding such numbers leads to opaque code which is difficult to change. The *defStruct* approach allows one to assign symbolic names to the slots, with the corresponding numbers being computed once and for all at compile time. Instead of making calls on arg/3 and mangle/3, the programmer makes calls on access predicates which are defined in terms of arg/3 and mangle/3. (These calls can themselves be macro-processed to replace the access predicate calls by direct calls on arg and mangle, thus making it possible to utilize good coding practice with no loss in performance. See Section [Ref: Macros] for more information.)

Consider the following example which is a simplified version of a defStruct used in an early ALS windowing package. The definition of the structure is declaratively specified by the following in a file with extension ***.typ,*** say ***wintypes.typ*** :

```
:- defStruct(windows,
        [
      propertiesList =
         [windowName,       % name of the window
```

```
            windowNum,         % assigned by window sys
            borderColor/blue,  % for color displays
            borderType/sing,   % single or double lines
            uLR, uLC,          % coords(Row,Col) of
                               % upper Left corner
            lRR, lRC,          % coords(Row,Col) of
                               % lower Right corner
            fore/black,        % foreground/background
            back/white         % text attribs
           ],
        accessPred  = accessWI,
        setPred     = setWI,
        makePred    = makeWindowStruct,
        structLabel = wi
       ]
      ).
```

We will discuss the details of this specification below. It can be placed anywhere in a source file, and acts like a macro, generating the following code in its place:

```
   export accessWI/3.
   export setWI/3.
   accessWI(windowName,_A,_B) :- arg(1,_A,_B).
   setWI(windowName,_A,_B) :- mangle(1,_A,_B).

   accessWI(windowNum,_A,_B) :- arg(2,_A,_B).
   setWI(windowNum,_A,_B) :- mangle(2,_A,_B).
   ...

   accessWI(back,_A,_B) :- arg(10,_A,_B).
   setWI(back,_A,_B) :- mangle(10,_A,_B).

   export makeWindowStruct/1.
   makeWindowStruct(_A) :-
     _A=..[wi,_B,_C,blue,sing,_D,_E,_F,_G,black,white].
```

```
export makeWindowStruct/2.
makeWindowStruct(_A,_B) :-
       struct_lookup_subst(
            [windowName,windowNum,borderColor,
             borderType,uLR,uLC,lRR,lRC,fore,back],
            [_C,_D,blue,sing,_E,_F,_G,_H,
             black,white],_B,_I),
        _A=..[wi|_I].

export xmakeWindowStruct/2.
xmakeWindowStruct(wi(_A,_B,_C,_D,_E,_F,_G,_H,_I,_J),
                 [_A,_B,_C,_D,_E,_F,_G,_H,_I,_J]).
```

Now let us examine the details.

9.1 Specifying Structure Definitions

defStructs directives are simply expressions of the form:

```
 :-defStruct(Name, EqnsList).
```

These are simply binary Prolog terms whose functor is `defStruct`. The first argument is an atom functioning as an identifying name for the type (it has no other use at present). The second argument is a list of *equality statements* providing the details of the definition. An *equality statement* is an expression of the form:

```
Left = Right
```

For defStructs, the left component of the equality statements must be one of the following atoms:

- propertiesList
- accessPred
- setPred
- makePred
- structLabel

The right sides of the defStruct equality statements are Prolog terms whose structure depends on the left side entry. The right side corresponding to 'propertiesList' is a list of atoms which are the symbolic names of the properties or slots of the structure being defined. For all of the rest of the equality statements, the right side is a single atom. The roles of these right side atoms are described below:

9.1.1 accessPred

The name of the ternary (3-argument) predicate to be used for accessing the values of the slots in the structure.

9.1.2 setPred

The name of the ternary (3-argument) predicate to be used for setting or changing the values of the slots in the structure.

9.1.3 makePred

The name of the unary predicate used for obtaining a fresh structure of the defined type.

9.1.4 structLabel

The name of the functor of the structure defined.

9.1.5 propertiesList

This is a list of slot specifications. A ***slot specification*** is on of the following:

- an atom, which is the name of the particular slot, or
- an expression of the form

  ```
  SlotName/Term,
  ```

  where `SlotName` is an atom serving as the name of this slot, and `Term` is an arbitrary Prolog term which is the default value of this particular slot, or
- an ***include*** expression which is a term of the form

  ```
  include(File, Type)
  ```

  where `File` is a path to a file, and `Type` is the name of a defStruct which appears in that file; if `File` can be located, and if the defStruct `Type` appears in `File`, the elements of `propertiesList` for `Type` are interpolated at the point where the ***include*** expression occurred; ***include*** expressions may be recursively nested. [Note: The typecomp compiler does not change its directory location when handling include expressions. Thus, if you utilize relative paths in recursive includes, these paths must always be valid from the directory in which the compiler was invoked.]

9.2 Using Structure Definitions

As can be seen from the generated code for the wintypes example at the beginning of this section, the atoms on the right sides of the accessPred and setPred equality statements become names for ternary predicates which are surrogates for arg/3 and mangle/3, respectively. And the atom on the right side of the makePred equality statement becomes the name of a unary predicate producing a new instance of the structure when called with a variable as its argument. Formally:

accessPred=*acpr* `acpr(Slot_name,Struct,Value)` succeeds precisely when `Slot_name` is an atom occurring on the propertiesList in the defStruct, `Struct` is a structure generated by the makePred of the defStruct, and `Value` is the argument of Struct corresponding to the the slot `Slot_name`.

setPred=*stpr* `stpr(Slot_name,Struct,Value)` succeeds precisely when `Slot_name` is an atom occurring on the propertiesList in the defStruct, `Struct` is a structure generated by the makePred of the defStruct, and `Value` is any legal Prolog term; as a side-effect, the argument of Struct corresponding to `Slot_name` is changed to become `Value`.

makePred=*mkpr* `mkpr(Struct)` creates a new structured term whose functor is the right side of the `structLabel` equality statement of the defStruct and whose arity list the length of the propertiesList of the defStruct, and unifies `Struct` with that newly created term; as a matter of usage, `Struct` is normally an uninstantiated variable for this call.

Thus, the goal

```
makeWindowStruct(ThisWinStruct)
```

will create a `wi(...)` structure with default values and bind it to `ThisWin-Struct`. Besides the unary generated 'make' predicates, two other construction predicates are created:

makePred=*mkpr* `mkpr(Struct, ValsList)` creates a new structured term whose functor is the right side of the `structLabel` equality statement of the defStruct and whose arity list the length of the propertiesList of the defStruct, and unifies `Struct` with that newly created term; `ValsList` should be a list of equations of the form `SlotName = Value`, where `SlotName` is one of the slots specified on `PropertiesList`; the newly created structured term will have value `Val` at the postion corresponding to `SlotName`; these "local defaults" will override any "global defaults" specified in the `defStruct`.

makePred=x*mkpr*`mkpr(Struct,SlotVars)` creates a new structured term whose functor is the right side of the `structLabel` equality statement of the defStruct and whose arity list the length of the propertiesList of the defStruct, and unifies `Struct` with that newly created term; no defaults are installed, and SlotVars is a list of the variables occurring in Struct. This binary predicate x*mkpr* `mkpr(Struct,SlotVars)` is equivalent to

```
Struct =.. [ StructLabel | SlotVars].
```

10 ObjectPro: Object-Oriented Programming

ALS ObjectPro is an object-oriented programming toolkit fully integrated with ALS Prolog. Unlike some other approaches to object-oriented programming in Prolog, it is not implemented as a system on top of Prolog. Instead, it is seamlessly integrated with Prolog: object-oriented facilities can be smoothly accessed from ordinary Prolog programs, and the full power of Prolog can be used in the definition of object methods.

10.1 Overview of ObjectPro

The *objects* of ObjectPro are frame-like entities possessing state which survives backtracking. Each object belongs to a class from which it obtains its methods. Classes are arranged in a hierarchy, with lower classes inheriting methods from parent classes. The behavior of an object is determined by two aspects:

- The object's state, and
- The object's methods.

An object's *state* is a frame-like object consisting of named slots which can hold values. Figure 19 (*Illustration of an Object's State.*) illustrates the states of some simple objects.

slot name	*slot value*
myName	
locomotionType	
powerSource	
numWheels	
engine	
autoClass	
manufacturer	

Figure 19. Illustration of an Object's State.

Changes to the object's state amount to changes in the values of one or more slots.

Such changes are permanent and survive backtracking. The values which appear in slots can be any Prolog entity, including (the state of) other objects. Messages are sent to objects by calls of the form

```
    send(Object, Message).
```

In general, objects are created, held in variables, passed around among routines, and sent messages in the stype above. When necessary, an object can be assigned a global name when it is created which can be used for sending messages to the object. An object's *methods* are determined by the class to which it belongs.

A *class* is determined by three things:

- A local *state-schema* which describes the structure of part of the state of any object belonging to the class;
- The methods directly associated with the class;
- The classes from which this class inherits.

Classes are also required to have names -- these are principally used in defining objects. The complete *state-schema* for a class C is a structure whose collection of slots is the union of all of the slots appearing in the local state-schemata of classes from which C inherits, together with the slots from the local state-schemata of C. Slots in child classes must be distinct from slots in all ancestor classes. The methods associated with a class are defined by Prolog clauses which can utilize various primitive predicates for manipulating objects, as well as any ordinary Prolog predicates.

Objects are activated by sending them *message* . The methods of the class to which the object belongs (or from which its class inherits) determine the object's reaction to the message. A message can be an arbitrary Prolog term which may include uninstantiated variables, thus implementing the partially-instatiated message paradigm of Concurrent Prolog [Ref]

The ALS ObjectPro system is integrated with the module system of ALS Prolog, in that class adefinitions in ALS ObjectPro may be exported from their defining modules so as to be visible in other modules, or may be left unexported, rendering them local to the defining module. However, each object 'knows' the module of its defining class, so that if one has hold of the object in a variable `Object`, then the call

```
send(Object, Message)
```

can be made from the context of any module.

10.2 Defining Objects and Sending Messages

An object is defined by an expression of the form

```
create_object(Eqns, Obj)
```

where `Obj` is an uninstantiated variable which will be bound to the new object, and `Eqns` is a list of *equations* of the form

```
Keyword = Value
```

The acceptable keywords, together with their associated `Value` types, are the following:

```
instanceOf  - atom (name of a class)
values      - list of equations
```

The `instanceOf` keyword equation is the only required equation; the value on the right side of this equation must be an atom which is the name of class which is visible from the module in which the `create_object` call is made. Here is an example of a simple object definition, where `iC_Engine` must be the name of class:

```
create_object([instanceOf=iC_Engine ], Obj)
```

The equations appearing on a list which is the right side of a

```
values =ValuesList
```

equation are expressions of the form

```
SlotName = SlotValue
```

where `SlotName` is one of the named slots in the structure defining the object's state. These slots are determined by the class to which the object belongs, and may be slots from the state-schema of the immediate class parent, or may also be slots from any of the state-schemata of ancestor classes. The intent of the `values` equation is to enable the programmer to prescribe initial values for some of the object's slots when it is created.

When a global atomic name for the object is required, one includes an equation of the form

```
name  =  <atom> .
```

A message is sent to an object with a call of the form

```
send(Object, Message)
```

where `Object` is the target object (or an atom naming the object), and `Message` is an arbitrary Prolog term. The `Message` may include uninstantiated variables which might be instantiated by the object's method for dealing with `Message`. Such calls to `send/2` can occur both in ordinary Prolog code, and in the code defining methods of classes (and hence objects). For convience, or conceptual emphasis, a call

```
send_self(Object, Message)
```

is provided. This is merely syntactic sugar for

```
send(Object, Message)
```

That is, the implementation makes no attempt to verify that a `send_self` message is being truly sent from an object to itself.

10.3 Defining Classes

A class is defined by a directive of the form

```
:- defineClass(Eqns).
```

Here `Eqns` is a list of *equations* of the form

```
Keyword = Value
```

The acceptable keywords, together with their associated `Value` types, are the following:

```
name       - atom
subclassOf - atom (name of a (parent) classe)
addl_slots - list of atoms (names of local slots)
defaults   - list of default values for slots
constrs    - list of constraint expressions for slots
export     - yes or no
```

```
   action     - atom
```

The `name` equation and the `subclassOf` equation are both required.

The ObjectPro system pre-defines one top-level class named `genericObjects`; all classes are ultimately subclasses of the `genericObjects` class. `genericObjects` provides one visible slot, `myName`, which is always instantiated to the object's name. Several other slots, normally non-visible, are also provided.

A class is said to be an *immediate subclass* of the (parent) class named in the `subclassOf` equation. The relation *subclass* is the transitive closure of the *immediate subclass* relation.

The atoms on the `addl_slots` list specify slots in the structure defining the state of objects which are instances of this class. These new slot names must not be slot names in any of the ancestor classes from which the new class inherits; hence the nomenclature "addl_slots". The *state-schema* of a class is the union of the `addl_slots` of the class with the `addl_slots` of all classes of which the class is a subclass. Reiterating, it is required that the slot names occurring on all these `addl_slot` lists be distinct.

Here are several examples of simple class definitions:

```
   :- defineClass([name=vehicle,
            subclassOf=genericObjects,
           addl_slots=[locomotionType, powerSource] ]).
   :- defineClass([name=wheeledVehicle,
            subclassOf=vehicle,
            addl_slots=[numWheels] ]).
   :- defineClass([name=automobile,
            subclassOf=wheeledVehicle,
         addl_slots=[engine,autoClass,manufacturer] ])
   :- defineClass([name=wingedVehicle,
            subclassOf=vehicle,
            addl_slots=[numWings] ]).
```

The inheritance relations among these clesses is shown in Figure 20 .

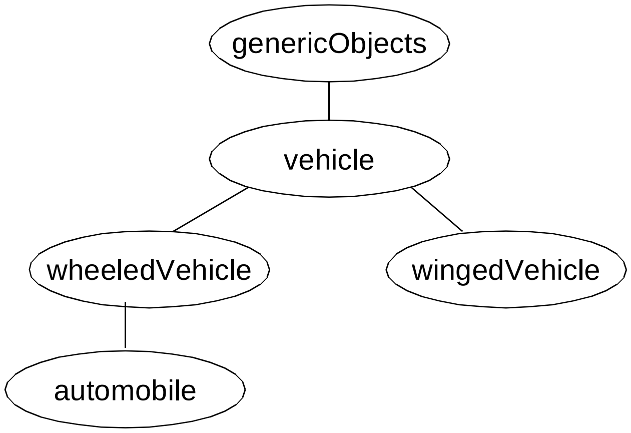


Figure 20. Example Class Inheritance Relations.

The state-schemata (not including the slots provided by `genericObjects`) for each of these classes are shown below:

```
vehicle        - [locomotionType, powerSource]
wheeledVehicle - ]locomotionType, powerSource,
                  numWheels]
automobile     - [locomotionType, powerSource,
                  numWheels,
                  engine,autoClass,manufacturer]
wingedVehicle  - [locomotionType, powerSource,
                  numWings]
```

An object which is instance of a class has a slot in its state structure corresponding to each entry in the state-schema for the class.

A class definition can supply default values for slots using the equation:

```
defaults = list of default values for slots
```

More specifically, the expression on the right should be a (possibly empty) list of equation pairs

<SlotName> = <Value>,

where <SlotName> is any one of the slotnames from the complete state schema of the class, and <Value> is any appropriate value for that slot. Omitting this keyword in a class definition is equivalent to including

```
    defaults = []
```

If an `export = yes` equation appears on the `Eqns` list of a class definition, the class methods and other information concerning the class are exported from the module in which the definition takes place.

Of course, the call could also fail if `C's` method code for `Message` fails. The `action=Name` equation is used to override the default name for the methods predicate of the class. If such an equation is present, the methods predicate will be `Name/2` instead of the default indicated above.

The `constraints` equation allows the programmer to impose constraints on the values of particular slots in the states of objects which instances of the class. The general form of a constraint specification is

```
   constrs = list of constraint expressions
```

Three types of constraint expressions are supported:

- `slotName = value`
- `slotName < valueList`
- `slotName - Var^Condition`

The first two cases are special cases of the third, and are provided for convenience. In all three cases, the left side of the expression is the name of a slot occurring in the complete state-schema of the class being defined (i.e., it is either the name of a slot on the `addl_slots` list of the class, or is a slot in the schema of a superclass from which the class being defined inherits). In the case of `slotName = value`, `value` is any Prolog term. This constraint expression indicates that any instance of the class being defined must have the value of slot `slotName` set equal to `value`. The generated code ensures that when instances of the class are initialized (via the call `send(Object, initialize)`), the value of `slotName` is set to `value`. The constraint expression `slotName < valueList` requires that the values of `slotName` be among the Prolog terms appearing on the list `valueList`. Here '<' is a short hand for 'is an element of'. The generated code for the class methods applies a test to any attempted update of the value of `slotName` to ensure that the new value is on the list `valueList`.

As indicated, the third constraint expression subsumes the first two. `Var` is a Prolog variable, and `Condition` is an arbitrary Prolog call in which `Var` occurs. `Conditon` expresses a condition which any potential value for `slotName` in an instance of the class must meet in order to be installed. The generated code imposes this test on all attempts to update the value of `slotName`. The test is imposed by binding the incoming candidate value to the variable `Var` , and then calling the test `Conditon`.

Here is a class specification inclulding a constraint:

```
defineClass([name=engine,
             subclassOf=[genericObjects],
             addl_slots=
                [powerType,fuel,engineClass,
                 cur_rpm,running,temp],
             constrs=
                [engineClass<
               [internalCombustion,steam,electric]]
            ])
```

10.4 Specifying Class Methods

To specify the methods of a class, the programmer must define a two argument predicate which will specify the reactions of instances of the class to various messages. The default name of this action predicate is

<class name>Action

However, the name of the predicate can be specified by using a line

```
action = <atom>
```

in the class definition. Thus, using the default, the head of the clauses for the action predicate will be of the form:

```
<ClassName>Action(Message,State)
```

The clauses for this predicate specify the methods which the class objects will use for responding to the various messages they are prepared to accept. The `Message` argument can be any Prolog term, and may include uninstantiated variables. The `State` argument will be instantiated at execution time to the state of the object which is using this method to respond to `Message`. The programmer has no knowledge of the detailed structure of `State`. However, access to the slots of State is provided by two predicates:

```
   setObjStruct(SlotDescrip, State, Value)
   accessObjStruct(SlotDescrip, State, VarOrValue)
```

The first call

```
   setObjStruct(SlotName, State, Value)
```

destructively updates the slot `SlotName` of `State` to contain `Value`, which cannot be an uninstantiated variable. However, Value can contain uninstantiated variables. Any constraints imposed on this slot by the class must be satisfied by the incoming `Value`. The second call

```
   accessObjStruct(SlotName, State, Value)
```

accesses the slot `SlotName` of `State` and unifies the value obtained with `VarOrValue`.

The value of `SlotDescrip` above is a *slot description* , which is either a slot name, or an expression of the form

```
   SlotName^SlotDescrip
```

The latter is used in cases of compound objects in which the value installed in a slot may be the state of another object. Thus if the contents of SlotName in State is another object O2, then

```
   accessObjStruct(SlotName^SlotDescrip, State, V)
```

effectively performs

```
   accessObjStruct(SlotDescrip, O2, V) .
```

Two convenient alternatives for these predicates are supplied as “syntactic sugar”:

```
   State^SlotDescrip := Value
```

for

```
   setObjStruct(SlotDescrip, State, Value)
```

and

```
   VarOrValue := State^SlotDescrip
```

for

```
   accessObjStruct(SlotDescrip, State, VarOrValue)
```

Besides these two constructs, calls on `send/2` can be used in the clauses defining methods. The code for the action predicate should be defined in the same module as the definition of the class. (But it can reside in separate files.)

Consider the class `engine` specified in the preceeding section. Simple `start` and `stop` methods can be implemented for this class by the following clauses:

```
   engineAction(start,State) :-
               State^running := yes.
   engineAction(stop, State) :-
               State^running := no.
```

A method to query the status of an engine is given by:

```
   engineAction(status(What),State) :-
           What := State^running.
```

The `genericObjects` class provides three pre-defined methods, effectively defined as follows:

```
    genericObjectsAction(get_value(SlotDesc,Value),State)
            :-
            accessObjStruct(SlotDesc,State,Value).
    genericObjectsAction(set_value(SlotDesc,Value),State)
            :-
            setObjStruct(SlotDesc,State,Value).
```

10.5 Examples

The first simple example implements an elementary stack object:

```
   :- defineClass([name=stacker,
                subclassOf=[genericObjects],
                addl_slots=[theStack, depth]
               ]).

   :- defineObject([name=stack,
                 instanceOf=stacker,
                 values=[theStack=[], depth=0]
                ]).


   stackerAction(push(Item),State)
     :-
     accessObjStruct(theStack, State, CurStack),
     setObjStruct(theStack, State, [Item | CurStack]),
     accessObjStruct(depth, State, CurDepth),
     NewDepth is CurDepth + 1,
     setObjStruct(depth, State, NewDepth).

   stackerAction(pop(Item),State)
     :-
     accessObjStruct(theStack, State, [Item |
     RestStack]),
     setObjStruct(theStack, State, RestStack),
     accessObjStruct(depth, State, CurDepth),
     NewDepth is CurDepth - 1,
     setObjStruct(depth, State, NewDepth).

   stackerAction(cur_stack(Stack),State)
     :-
     accessObjStruct(theStack, State, Stack).

   stackerAction(cur_depth(Depth),State)
     :-
     accessObjStruct(depth, State, Depth).
```

We can create a small loop to exercise an object of this class as follows:

```
run_stack :-
  create_object([instanceOf=stacker], Obj),
  rs(Obj).

rs(Obj)
  :-
  write('4stack:>'),flush_output,read(Msg),
  rs(Msg, Obj).

rs(quit, _).
rs(M, Obj)
  :-
  send(Obj, M),
  printf('Msg=%t\n', [M]),
  flush_output,
  rs(Obj).
```

Here is a sample session using this code:

```
?- [stacker].
Attempting to consult stacker...
... consulted /apache/als_dev/tools/objects/new2/
  stacker.pro

yes.
?- run_stack.
4stack:>push(2).
Msg=push(2)
4stack:>push(rr(tut)).
Msg=push(rr(tut))
4stack:>cur_stack(X).
Msg=cur_stack([rr(tut),2])
4stack:>pop(X).
Msg=pop(rr(tut))
4stack:>quit.
```

```
yes.
```

Our second example, the vehicles sketched earlier, illustrates the construction of compound objects.   First, here are the class defintions:

```
:- defineClass([name=vehicle,
       subClassOf=genericObjects,
       addl_slots=[locomotionType, powerSource] ]).
:- defineClass([name=wheeledVehicle,
       subClassOf=vehicle,
       addl_slots=[numWheels] ]).
:- defineClass([name=automobile,
       subClassOf=wheeledVehicle,
       addl_slots=[engine,autoClass,manufacturer]
        ]).
:- defineClass([name=engine,
       subClassOf=genericObjects,
       addl_slots=[powerType,fuel,engineClass,
                 cur_rpm,running,temp],
       constrs=[
       engineClass<
            [internalCombustion,steam,electric]]
       ]).
:- defineClass([name=iC_Engine,
       subClassOf=engine,
       addl_slots=[manuf],
       constrs = [engineClass = internalCombustion]
       ]).
```

Now here are the methods:

```
engineAction(start,State)
  :-
  State^running := yes.

engineAction(stop, State)
  :-
  State^running := no.
```

```
automobileAction(start,State)
  :-
  send((State^engine),start).

automobileAction(stop,State)
  :-
  send(State^engine,stop).

automobileAction(status(Status),State)
  :-
  send(State^engine,
       get_value(running,EngineStatus)),
  (EngineStatus = yes ->
       Status = running;
       Status = off
  ).
```

As in the stack example, we can create a simple loop to exercise this code:

```
run_vehicles
  :-
  set_prolog_flag(unknown, fail),
  create_object([instanceOf=iC_Engine ], Engine1),
  create_object([instanceOf=automobile,
        values=[engine=Engine1] ], Auto1),
  create_object([instanceOf=iC_Engine ], Engine2),
  create_object([instanceOf=automobile,
       values=[engine=Engine2] ], Auto2),

  run_vehicles(a(Auto1, Auto2)).

run_vehicles(Autos)
  :-
  printf('::>', []), flush_output,
  read(Cmd),
  disp_run_vehicles(Cmd, Autos).

disp_run_vehicles(quit, Autos) :-!.
disp_run_vehicles(Cmd, Autos)
  :-
  exec_vehicles_cmd(Cmd, Autos),
  run_vehicles(Autos).

exec_vehicles_cmd(Msg > N, Autos)
  :-
  arg(N, Autos, AN),
  send(AN, Msg),
  !,
  printf('%t-|| %t\n', [N,Msg]).

exec_vehicles_cmd(Cmd, Autos)
  :-
  printf('Can\'t understand: %t\n', [Cmd]).
```

And here is a trace of an execution of this code:

```
  ?- run_vehicles.
  ::>start > 1.
  1-|| start
  ::>status(A1) > 1.
  1-|| status(running)
  ::>start > 2.
  2-|| start
  ::>status(A2) > 2.
  2-|| status(running)
  ::>stop > 2.
  2-|| stop
  ::>status(X) > 2.
  2-|| status(off)
  ::>quit.
```

yes.