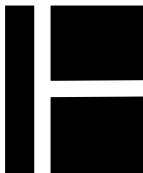


Development Tools

Restricted Rights Legend

When the Licensee is the U.S. Government or a duly authorized agency thereof, use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subdivision (b)(3)(II) of the Rights in Technical Data and Computer Software clause at 52.277.7013, dated Nov. 9, 1984.



Applied Logic Systems, Inc.

PO Box 175
Cambridge, MA 02140 USA
Email: info@als.com
WWW: <http://www.als.com>

A decorative rectangular border composed of repeating stylized motifs. The top and bottom horizontal sections feature a sequence of 'D'-like shapes followed by 'E'-like shapes, separated by small dots. The left and right vertical sections feature a sequence of 'U'-like shapes, also separated by small dots. The four corners are decorated with stylized human-like figures.

Development Environment

18 TTY Development Environment

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

18.0.1 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
?-
```

18.0.2 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 sig-

nifies 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.

18.1 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.

18.2 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.

18.3 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.

18.4 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:

18.4.1 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.
```

18.4.2 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 directories. See the User Guide for more information concerning loading files.

18.5 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 addition, you can add directories using the command-line switch `-S` at start-up time (see Section 18.7 (*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 described in the next section.

18.6 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.

18.7 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 mes-

sage. 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 sin-

gle 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 $M : (H_1, \dots, H_k)$, then each of H_1, \dots, H_k is asserted in module M . Thus,

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

would cause the two 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 (H_1, \dots, H_k) , then each of H_1, \dots, H_k is asserted in module `user`. Thus,

```
alspro -A `(jerry, ben)`
```

would cause the two 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.]

19 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).

19.1 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 get-

ting in and out of the procedure.

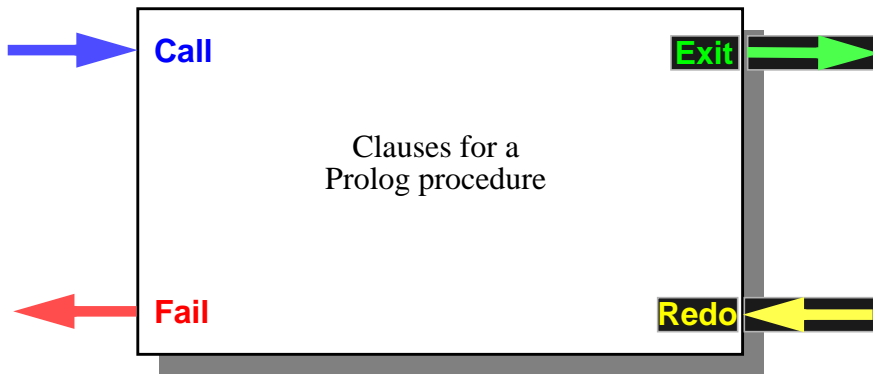


Figure 14. Generic Procedure Box.

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`.

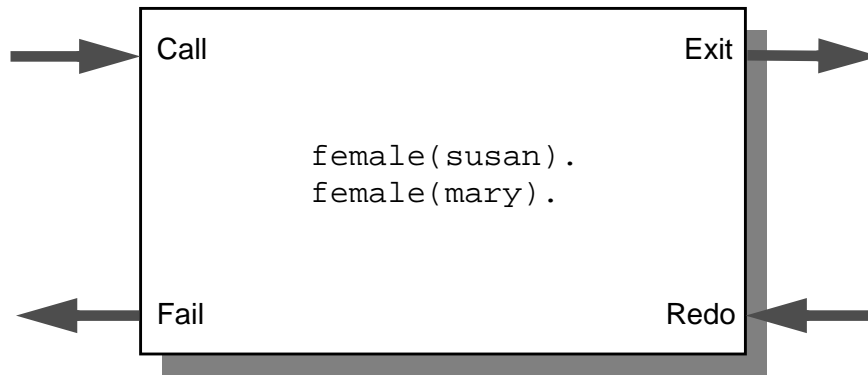


Figure 15. Procedure Box for `female/1`.

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)`.

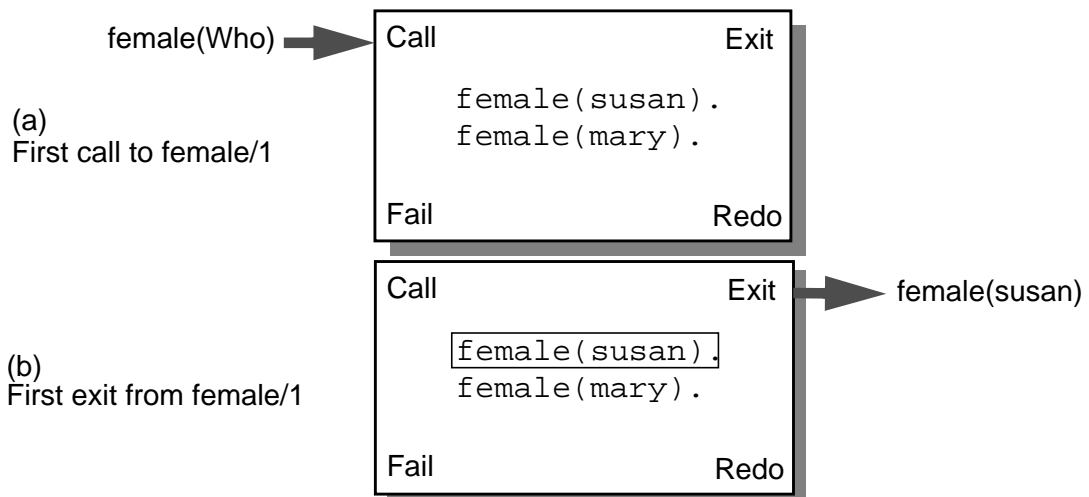


Figure 16. First call and exit tracing female/1.

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).

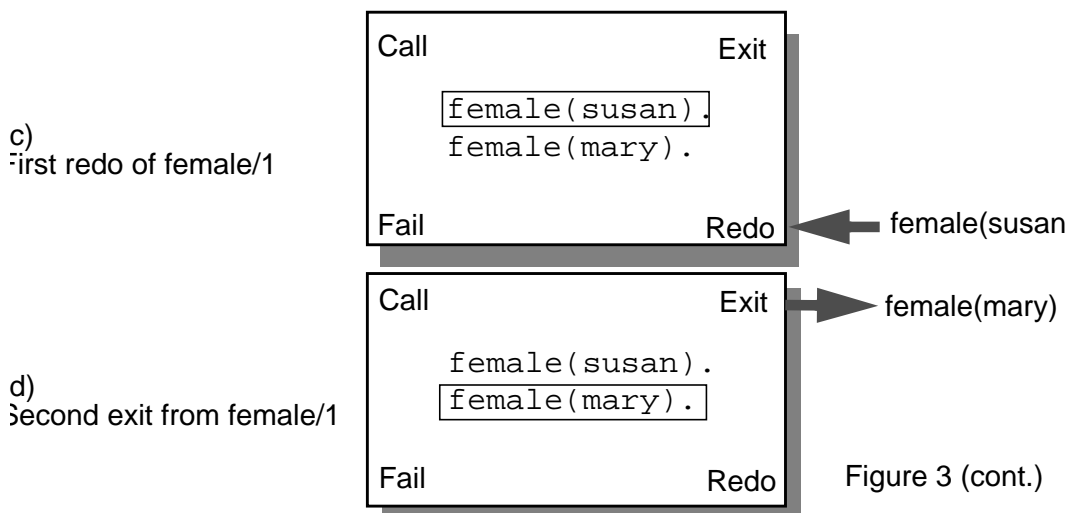


Figure 17. Redo and exit tracing female/1.

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.

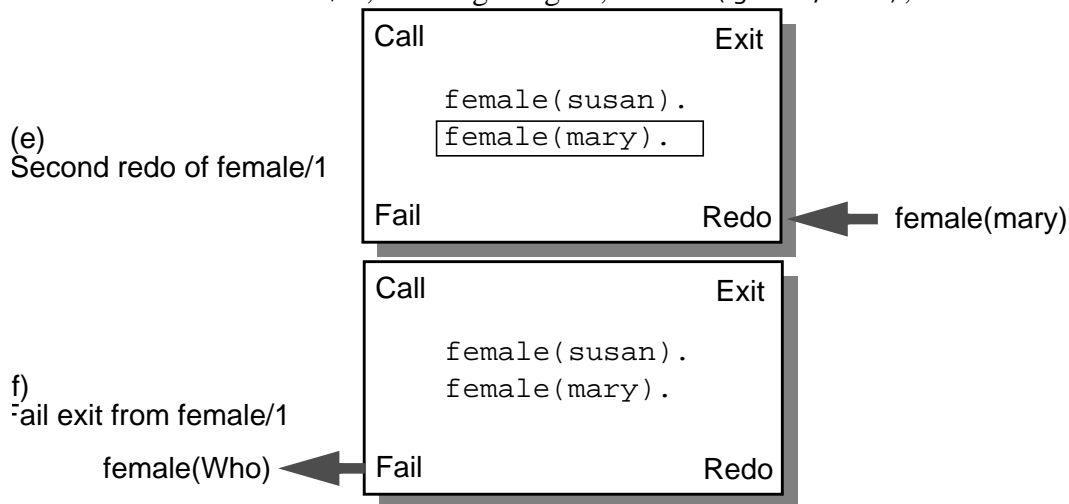


Figure 18. Redo and fail tracing `female/1`.

19.2 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.
```

19.3 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..

19.3.1 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.

19.3.2 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 than 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.

19.3.3 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`.

19.4 Changing the Leashing

After watching all of the ports during a trace, you might find that you 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.

19.5 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.

19.6 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 ? 1  
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) ?
```

19.7 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.
```

19.8 Getting Help

Typing 'h' at the ? prompt of the debugger gives a summary of the debugger commands.

19.9 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.

20 ALS Integrated Development Environment

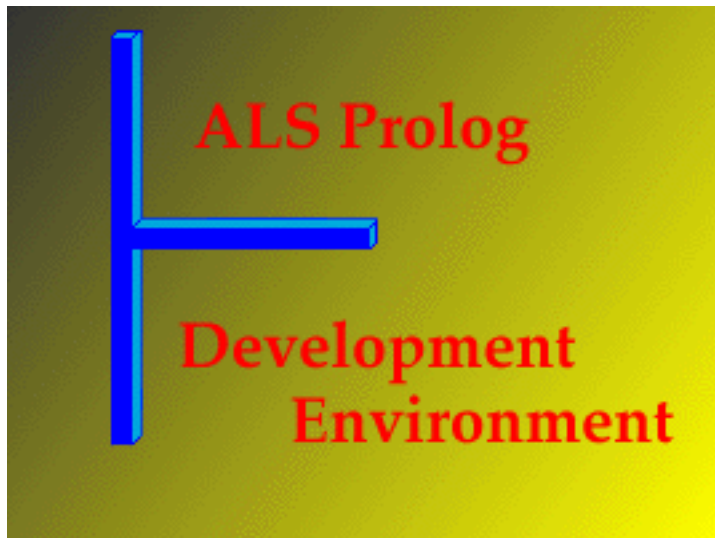
The ALS Integrated Development Environment (IDE) provides a GUI-based developer-friendly setting for developing ALS Prolog programs. Start the ALS IDE either by clicking on its icon (Windows or Macintosh, or CDE versions of Unix), or



by issuing

```
alsdev
```

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

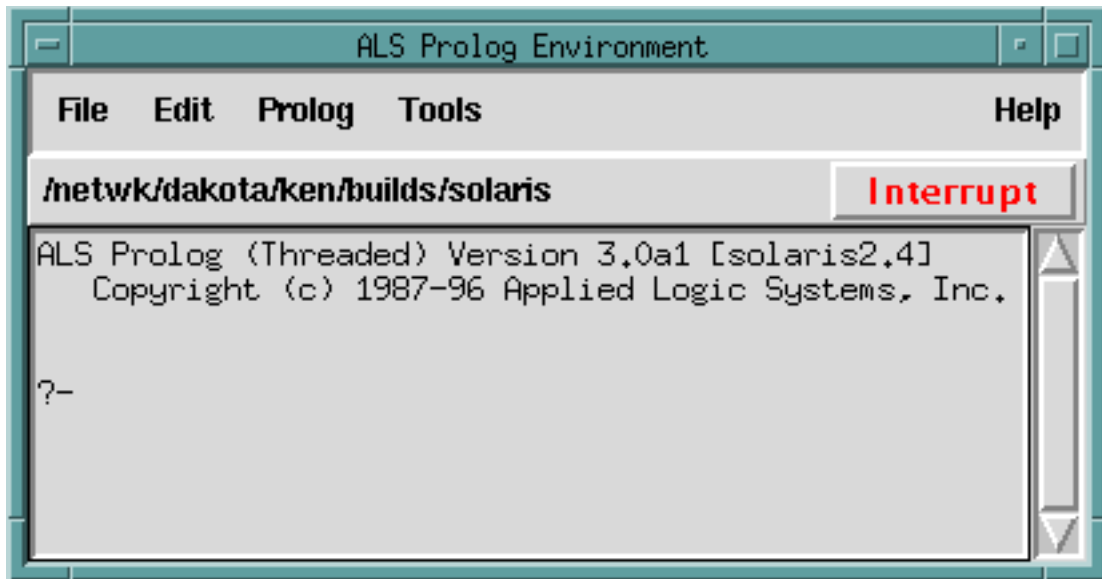


while it loads, and then replaces the splash screen with the main listener window.

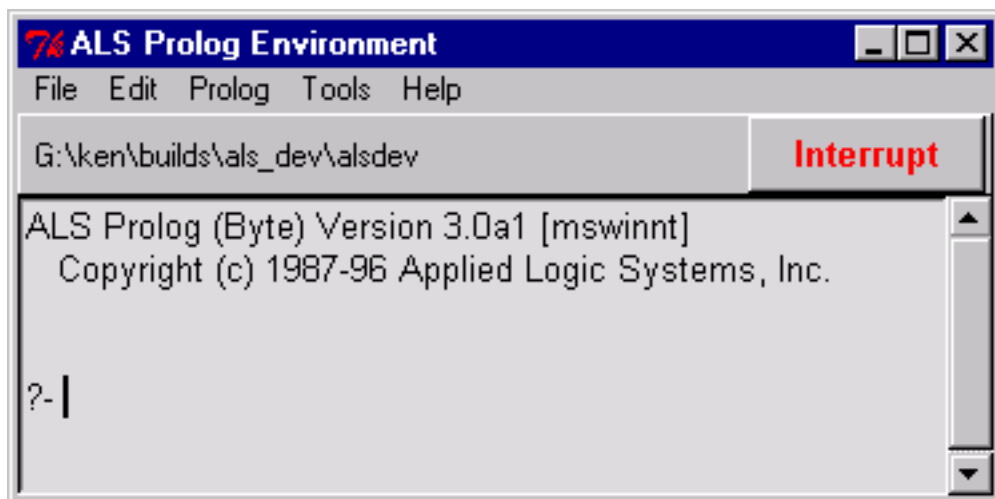
20.1 Main Environment Window

The details of the appearance of the ALS IDE windows will vary across the plat-

forms. Here is what reduced-size versions of the main window look like on Unix::



and on Windows:

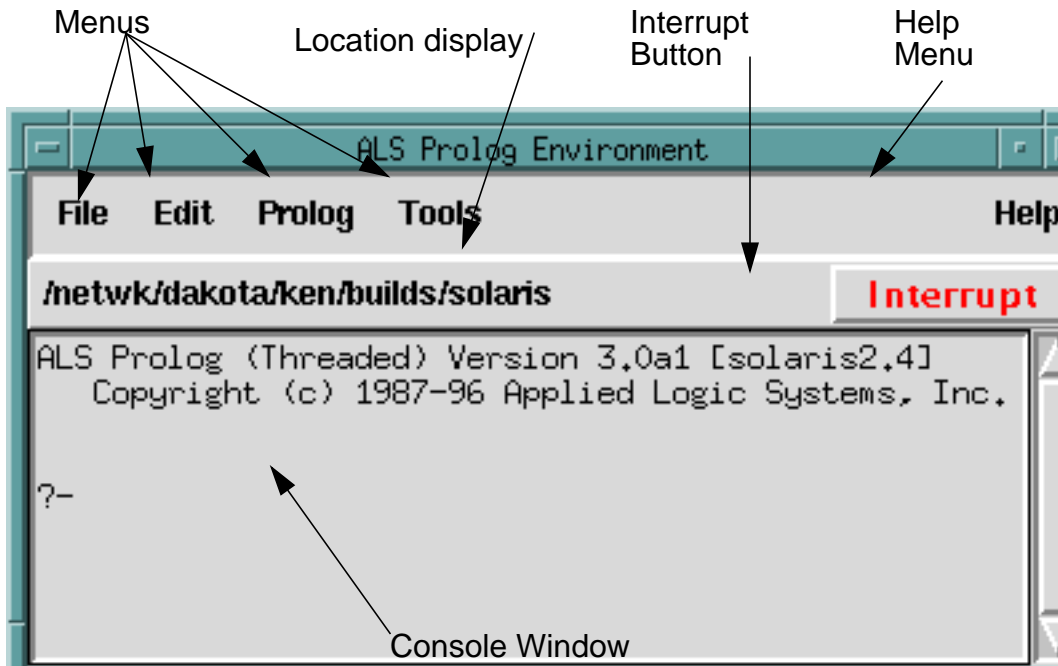


and on Macintosh:



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:



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.

20.2 Menus

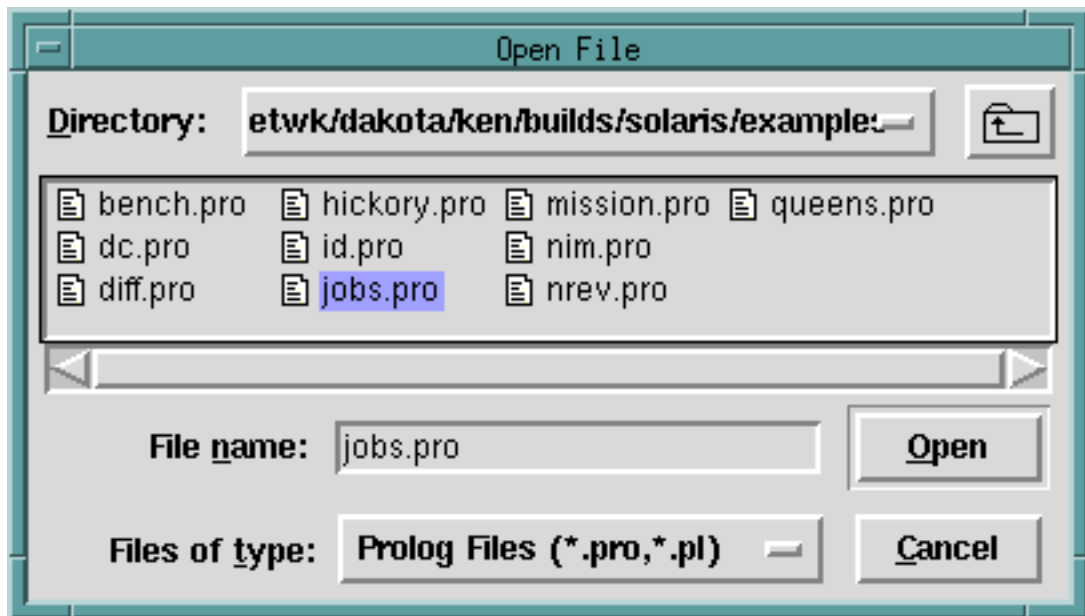
The ALS IDE is undergoing steady development. Some of the planned features are not yet implemented, and so menu items corresponding to them will show as grayed-out. The options indicated by the accelerator keys on the menus apply when the focus (insertion cursor) is located over the main listener window,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

20.2.1 File Menu

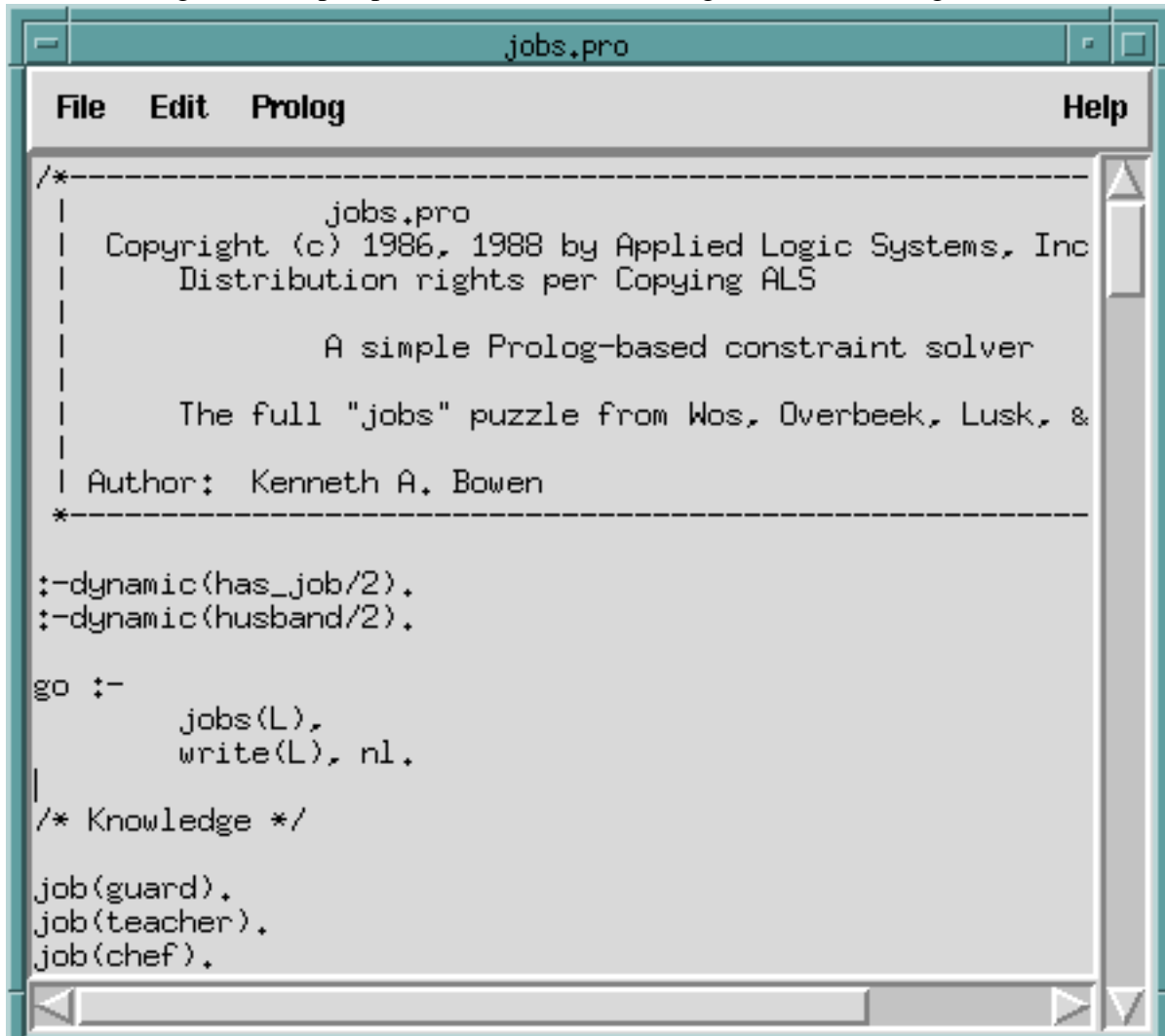


New, Open, Close, Save, and SaveAs apply to editor windows, and have their usual meanings. New opens a fresh editor window with no content, while Open al-

lows 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:

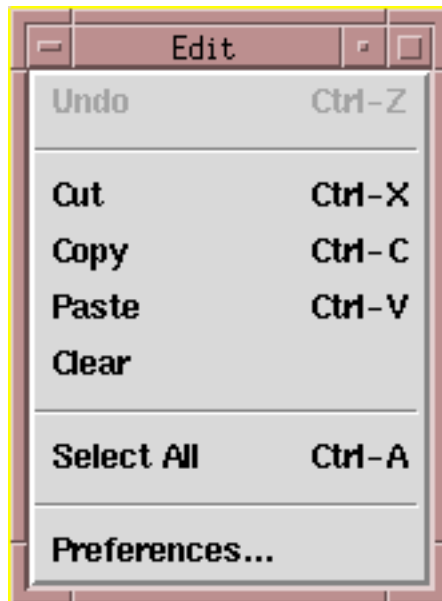


```
/*-----  
|           jobs.pro  
| Copyright (c) 1986, 1988 by Applied Logic Systems, Inc  
|   Distribution rights per Copying ALS  
|  
|           A simple Prolog-based constraint solver  
|  
|   The full "jobs" puzzle from Wos, Overbeek, Lusk, &  
|  
| Author: Kenneth A. Bowen  
|-----*/  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

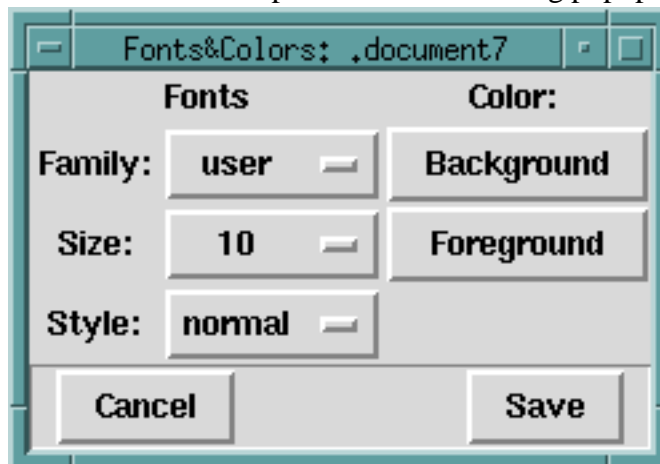
The Close, Save and SaveAs entries from the file menu apply to the edit window having the current focus.

Quit allows you to exit from the ALS Prolog IDE.

20.2.2 Edit Menu

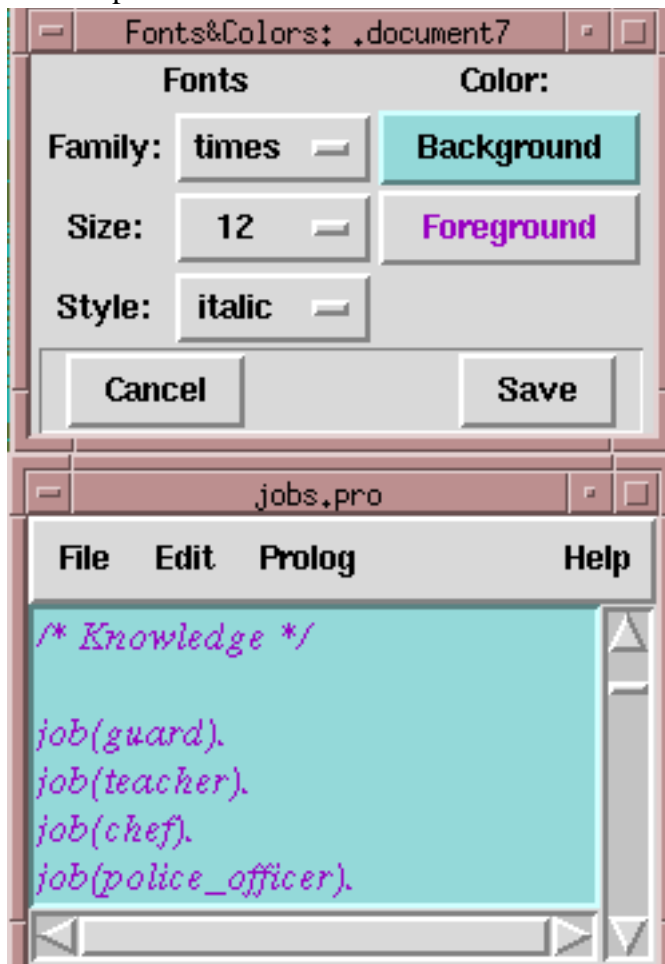


Cut, Copy, Paste, and Clear apply as usual to the current selection in an editor window. Copy applied in the main listener window, and Paste in the main window always pastes into the last line of the window. Select All only applies to editor windows. The Preferences choice produces the following popup window: Selec-



tions made using any of the buttons on this window immediately apply to the win-

dow (listener, debugger, or editor window) from which the Preferences button was pressed. For example:



Selecting **Cancel** simply removes the Fonts & Colors window without undoing any changes. Selecting **Save** records the selected preferences in the initialization 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 recording preferences. Preferences for the main listener window and the debugger window are save separately from the editor window preferences.

20.2.3 Prolog Menu



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:

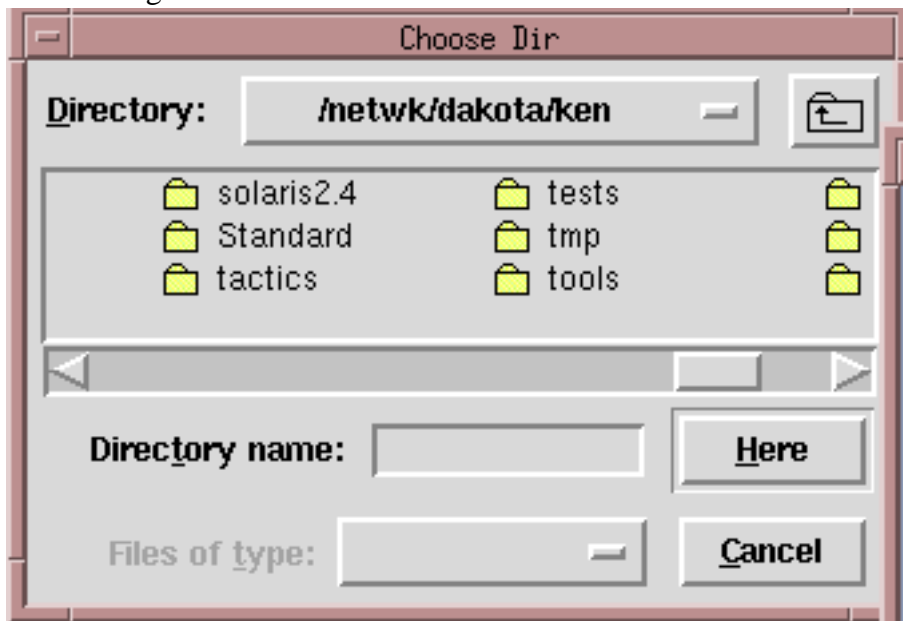


In contrast, if **Consult** is selected from an edit window (or the *Ctrl/<apple>-K* accelerator key is hit over that window), the file associated with that window is (re)consulted into the Prolog database; if unsaved changes have been made in the editor window, the file/window is first **Saved** before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

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

Set Directory ... allows you to change the current working directory. The Unix ver-

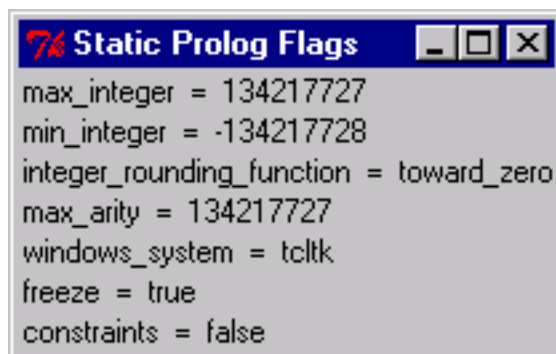
sion of this dialog is shown below:



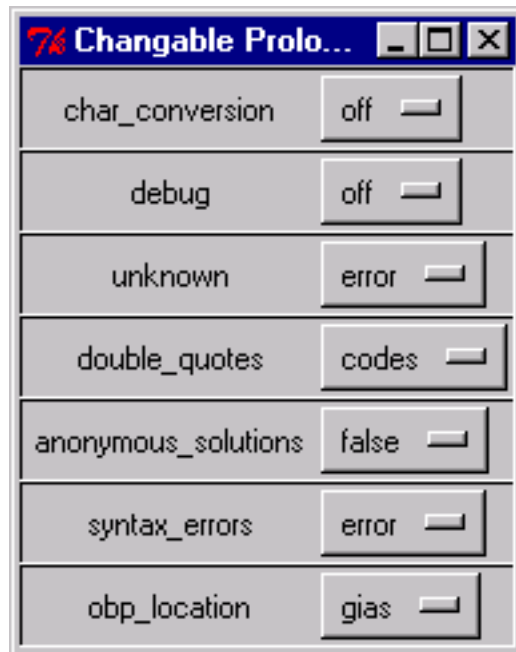
Clicking on **Here** in the display above would then cause the `get_cwd/1` predicate to behave as follows:

```
?- get_cwd(X).  
X = /netwk/dakota/ken
```

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

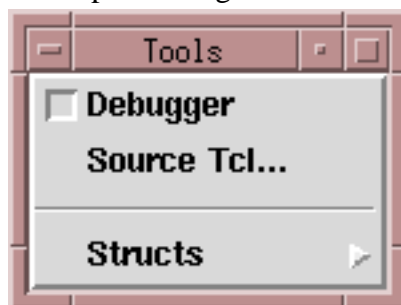


Similarly, selecting Dynamic Flags produces a popup window which displays the current values of all of the changable Prolog flags in the system, and allows one to reset any of those values:



20.2.4 Tools Menu

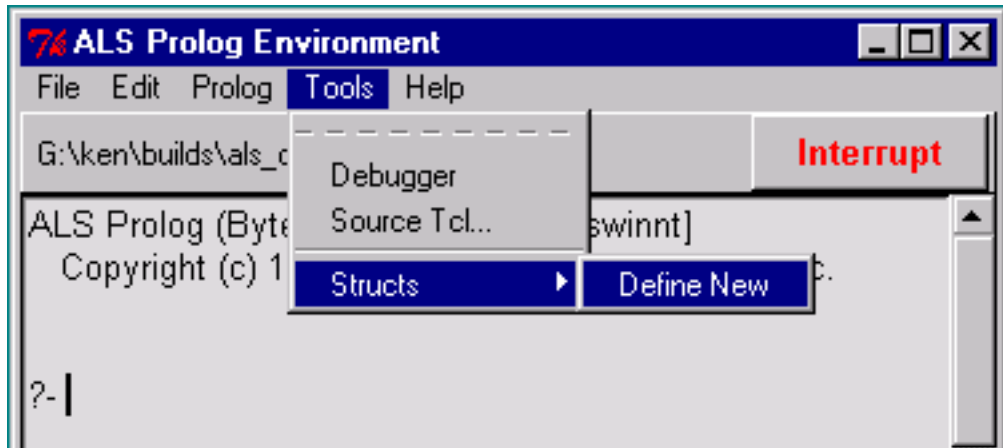
This menu is expected to grow with new entries in future releases. Currently:



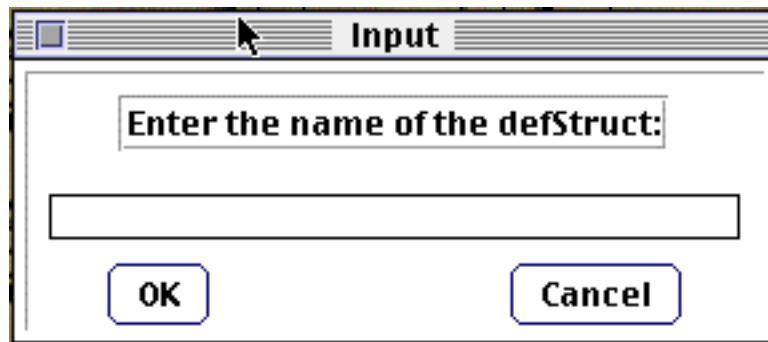
Selecting **Debugger** provides access to the GUI Debugger; this will be described in detail in Chapter 21 (*Using the ALS IDE Debugger*) .

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.

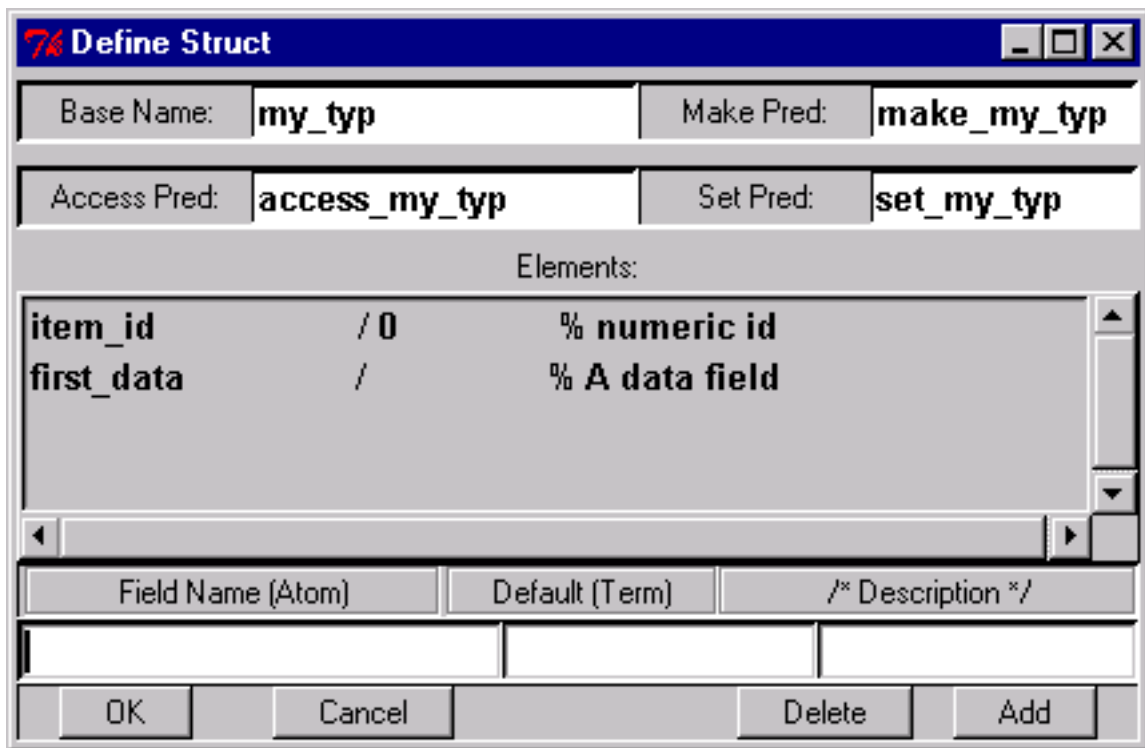
Structs provides a GUI interface to the creation (and later, editing) of DefStructs as described in Chapter 25 (*Abstract Data Types: Structure Definition*) . Select Structs > New;



and you are first prompted for a name for the type:



Enter the name a click OK, and the following window appears:



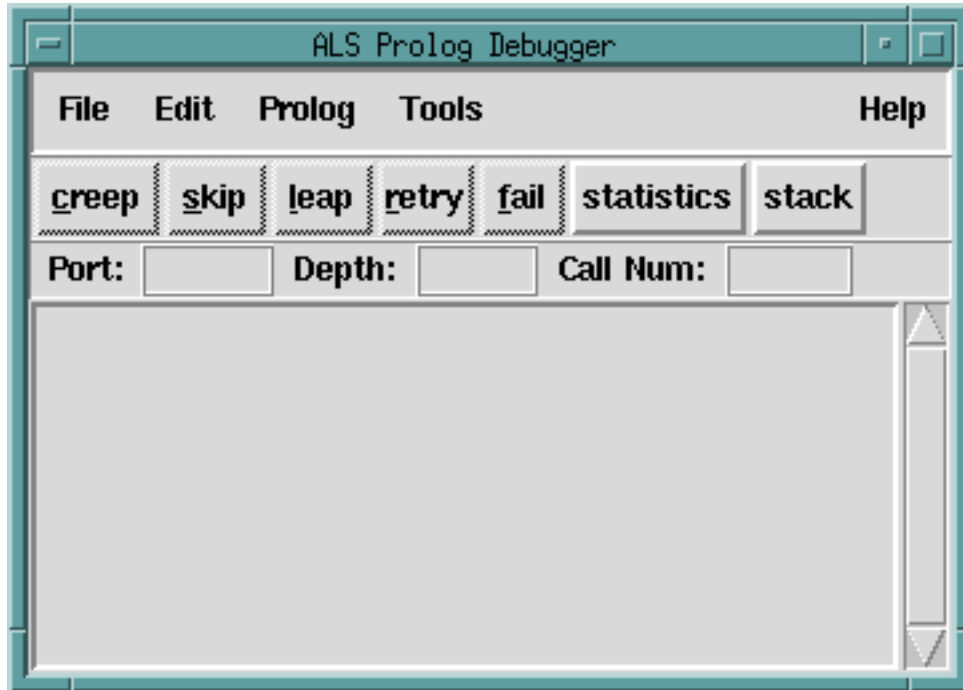
The image shows a dialog box titled "Define Struct" with a red logo on the left. It contains several input fields and a list of elements. The "Base Name" field is filled with "my_typ", "Make Pred" with "make_my_typ", "Access Pred" with "access_my_typ", and "Set Pred" with "set_my_typ". Below these is a section labeled "Elements:" containing a list with two entries: "item_id / 0 % numeric id" and "first_data / % A data field". At the bottom, there is a table with three columns: "Field Name (Atom)", "Default (Term)", and "/* Description */". Below the table are four buttons: "OK", "Cancel", "Delete", and "Add".

Base Name:	my_typ	Make Pred:	make_my_typ
Access Pred:	access_my_typ	Set Pred:	set_my_typ
Elements:			
item_id	/ 0	% numeric id	
first_data	/	% A data field	
Field Name (Atom)	Default (Term)	/* Description */	
OK	Cancel	Delete	Add

Initially, the Elements region will be empty, but the top for regions will be filled in. Making entries in the fields at the bottom and selecting **Add** causes lines to be entered in the Elements area. Selecting **OK** will prompt you for the name of a file (which is given extension *.typ*) in which to write the abstract type definition. The file you create, (for example, *my_typ.typ*) can simply be consulted. The ALS IDE recognizes the *.typ* extension, and first applies the comtype transformer to create the corresponding file *my_typ.pro*, which is then consulted.

21 Using the ALS IDE Debugger

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



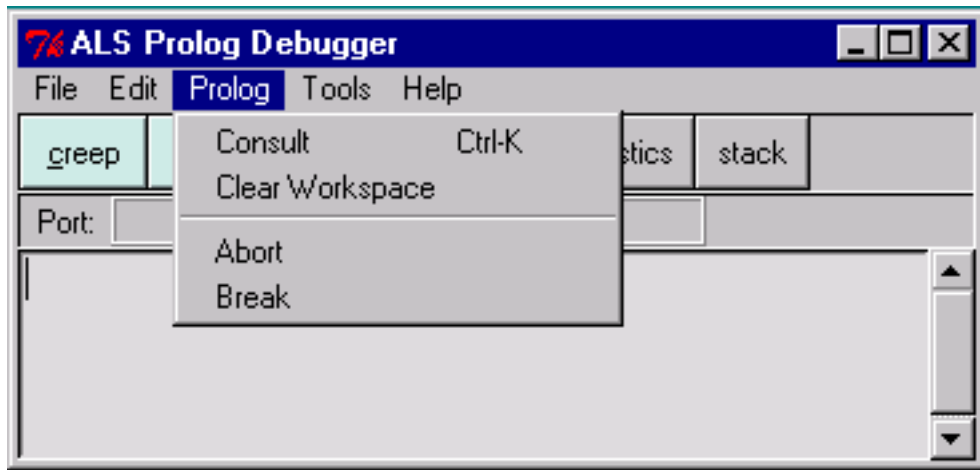
The debugger combines a traditional prolog four-port debugger (as described in Chapter 19 (*Using the Four-Port Debugger*)) with a source-code trace debugger. The details of the debugger action and the source trace will be described below. First we will examine the menus and buttons on the debugger window.

21.1 Debugger Window Menus.

The first two debugger menus, File and Edit, provide the same facilities as discussed for all other windows in Chapter 20 (*ALS Integrated Development Environment*).

21.1.1 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:

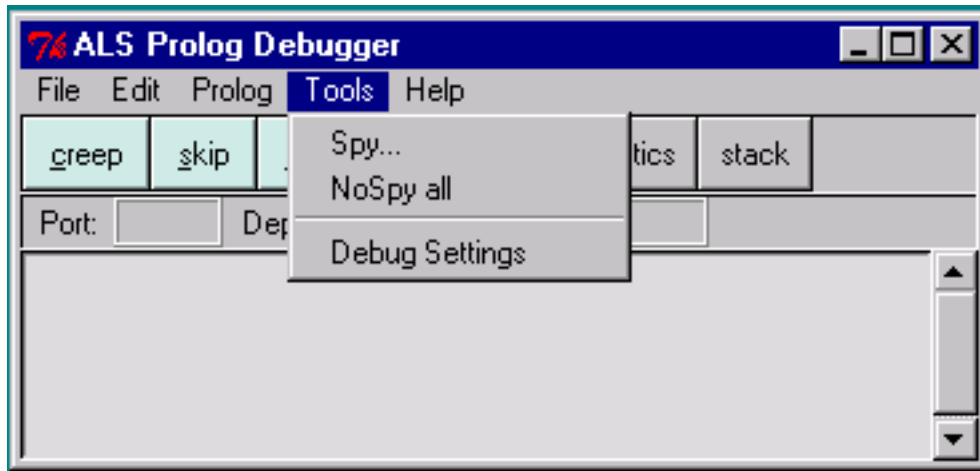
Abort -- choosing this causes the computation currently being traced to be aborted (effectively, `abort/0` is invoked).

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.

21.1.2 Tools Menu

The Tools Menu

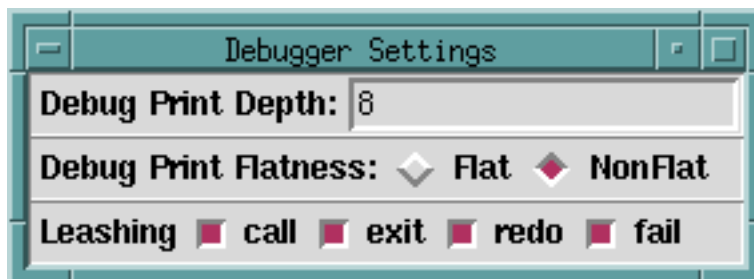


is completely specific to the debugger.

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

Choosing **NoSpy all** removes all spy points which are currently set.

Choosing **Debug Settings** raise the following popup window:



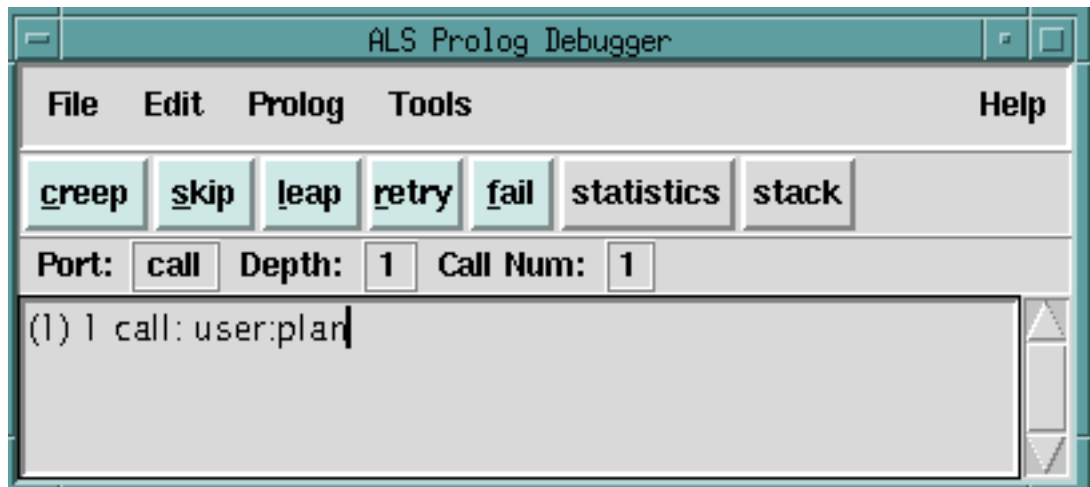
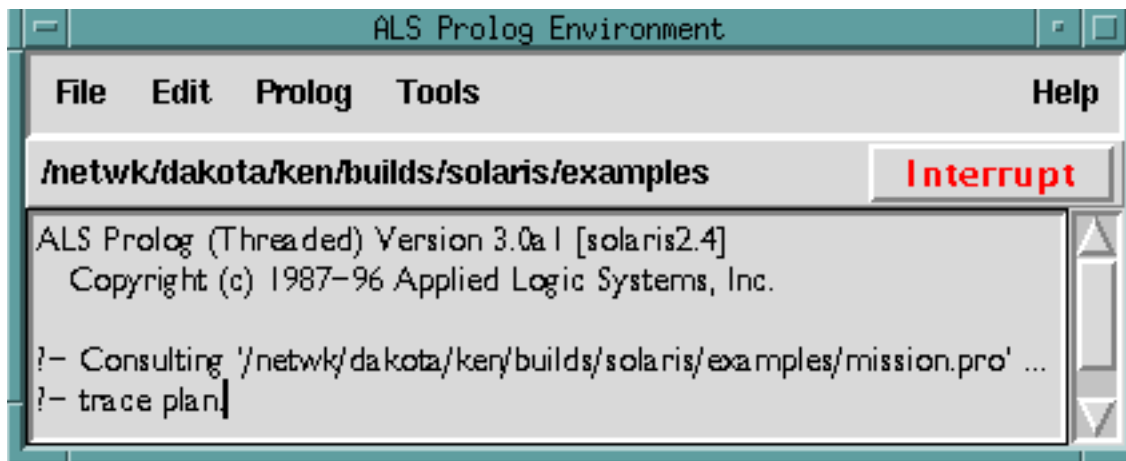
These settings control which debugger ports are shown, and also control the appearance of the lines printed for the ports which are show.

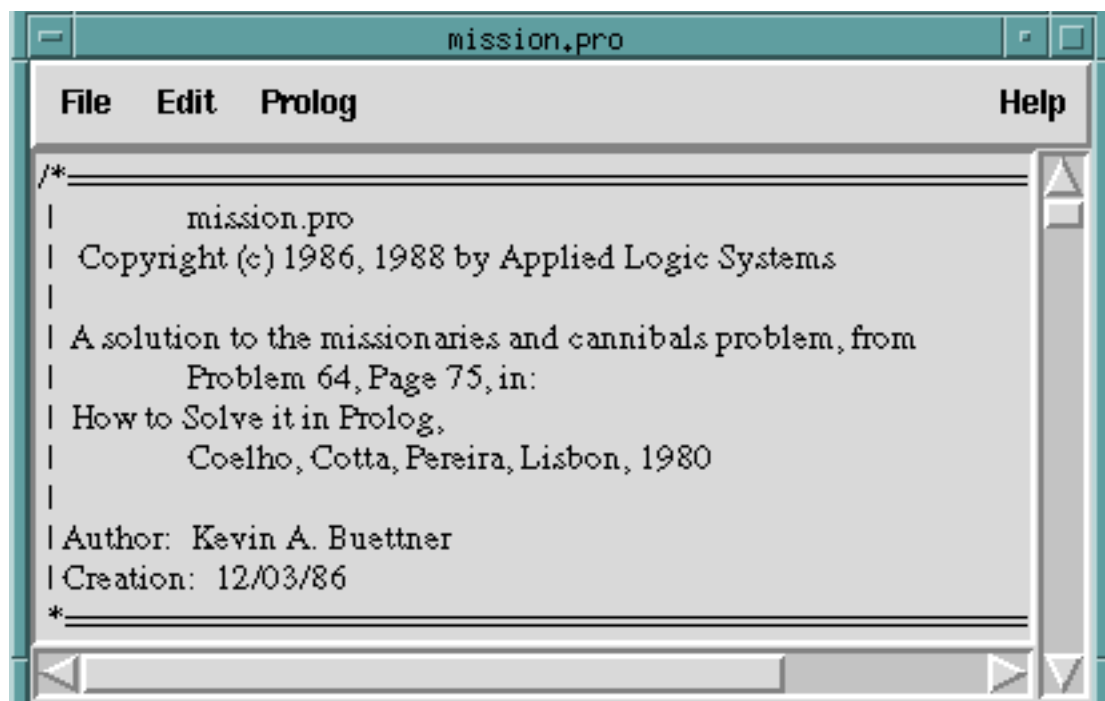
21.2 Tracing with the IDE Debugger.

It is necessary to raise the debugger window before consulting any of the files for which one wishes to see the source trace. Suppose we have raised the debugger, consulted the *mission.pro* example from the supplied example programs, and that we also open that program in the editor. Then begin tracing by typing

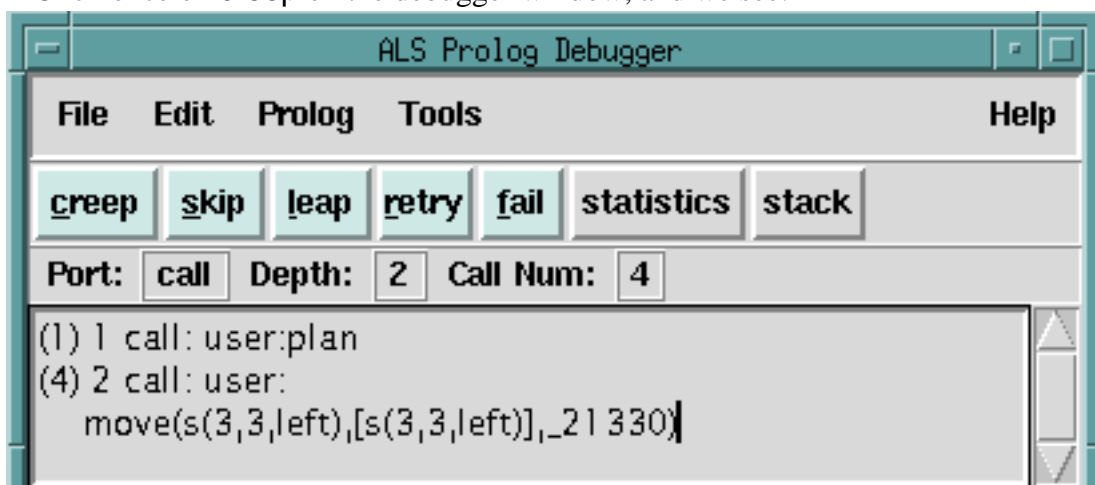
```
trace plan.
```

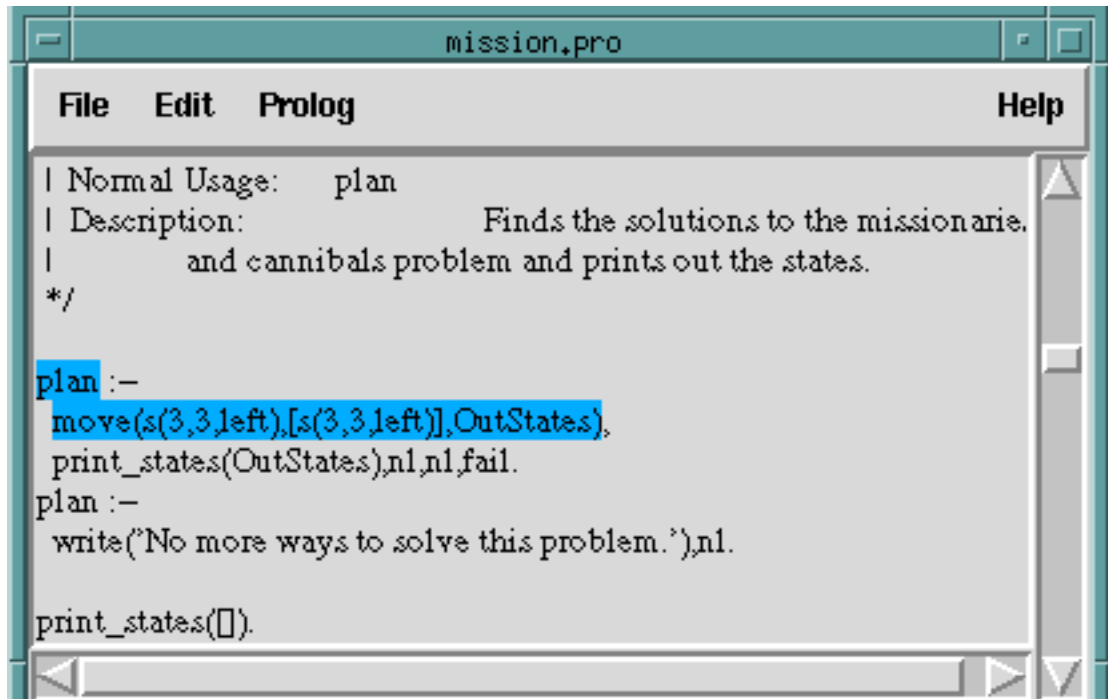
in the listener window:





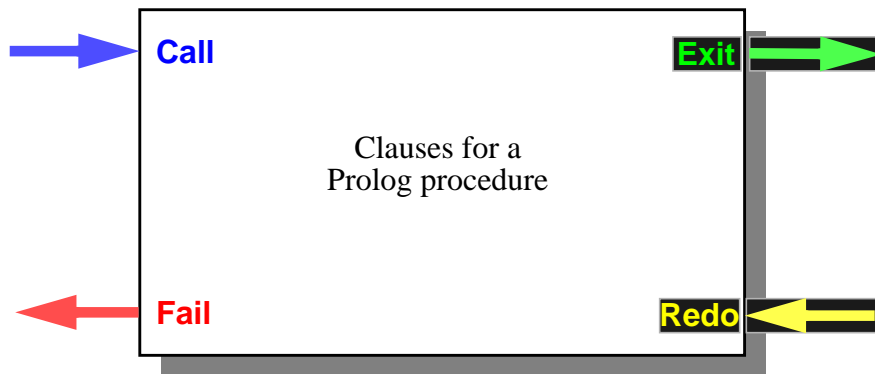
Click once on **creep** on the debugger window, and we see:



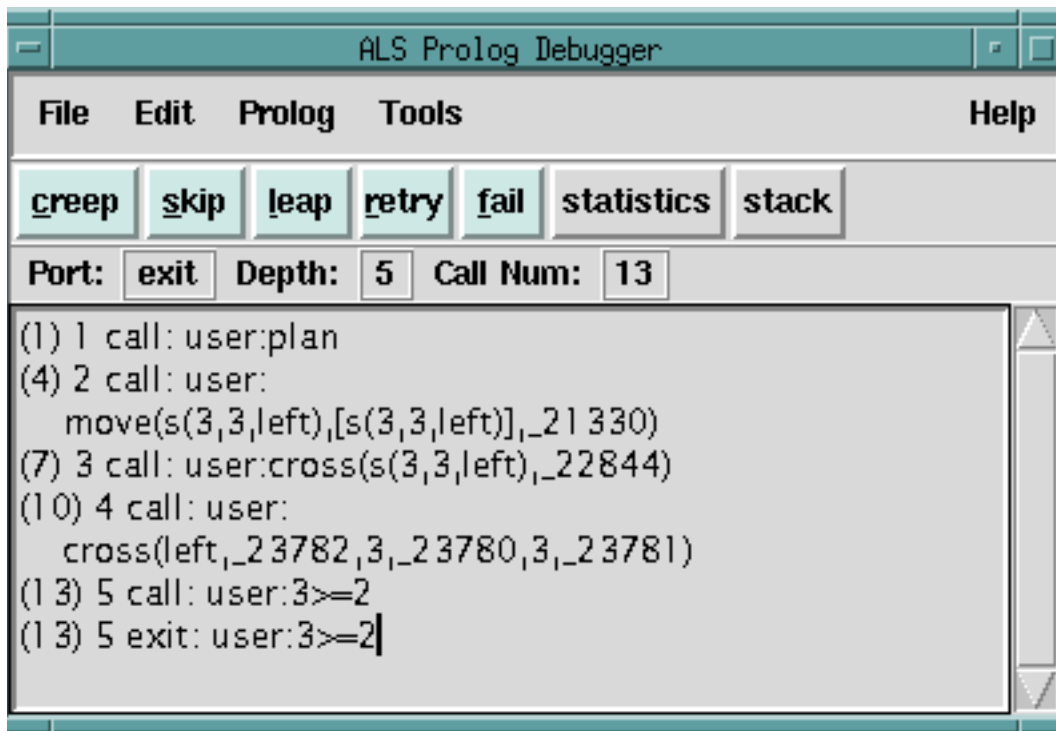


The call `move (. . .)` in the source code window which corresponds to the current call in the 4-port debugger window is highlighted in blue. In addition, the head of the clause in which the current call occurs is also highlighted in blue. Note that the window also automatically scrolled so that this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



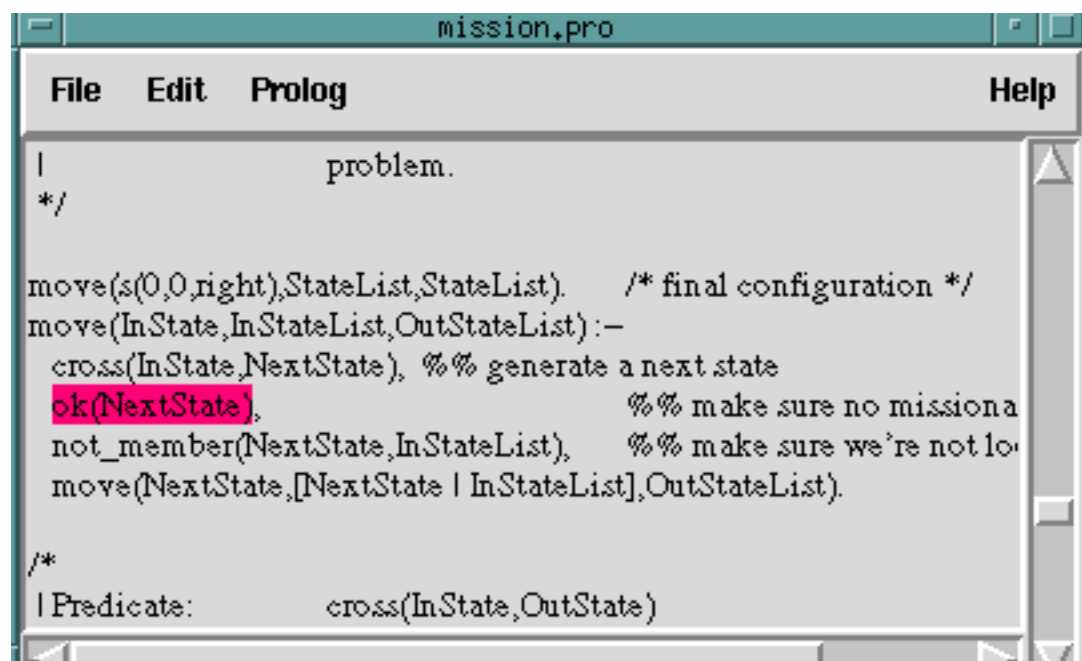
Click on creep four more times, and the situation is now:




```
File Edit Prolog Help
|
|      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 fro
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:

```
File Edit Prolog Tools Help
creep skip leap retry fail statistics stack
Port: fail Depth: 3 Call Num: 23
cross(left,_23782,3,_23780,3,_23781)
(1 3) 5 call: user:3>=2
(1 3) 5 exit: user:3>=2
(1 6) 5 call: user:_23780 is 3-2
(1 6) 5 exit: user:1 is 3-2
(1 0) 4 exit: user:cross(left,right,3,1,3,3)
(7) 3 exit: user:cross(s(3,3,left),s(1,3,right))
(2 3) 3 call: user:ok(s(1,3,right))
(2 3) 3 fail: user:ok(s(1,3,right))
```



```
mission.pro

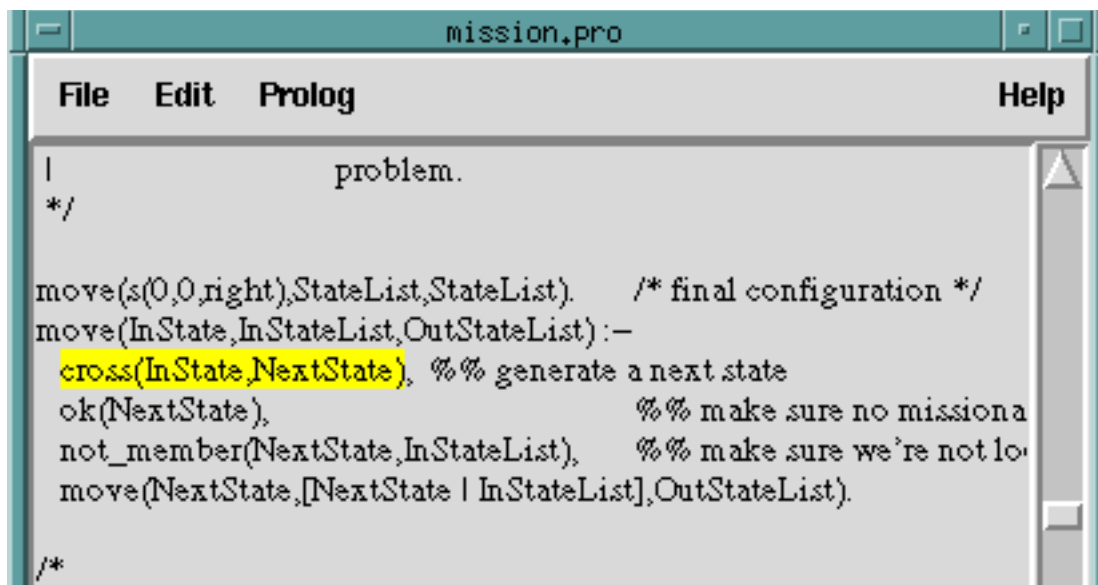
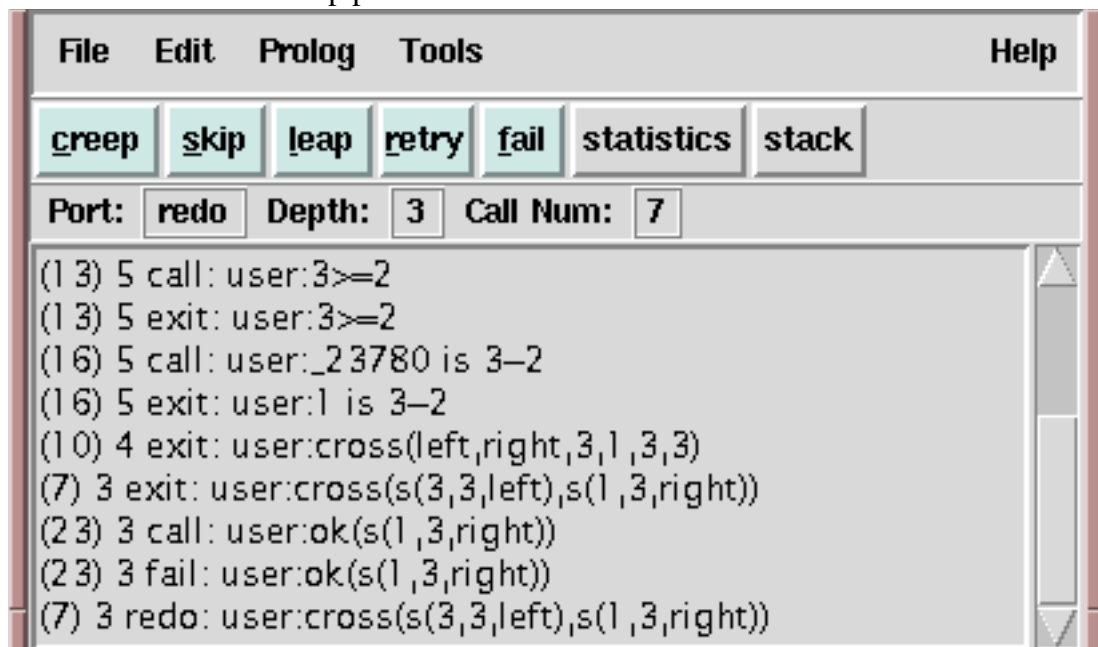
File Edit Prolog Help

|           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).

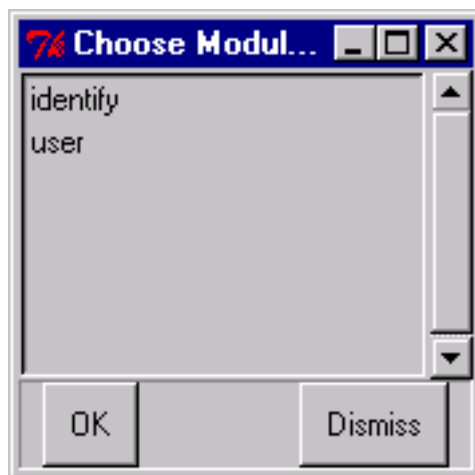
/*
| Predicate:      cross(InState,OutState)
```

One more click on creep produces:



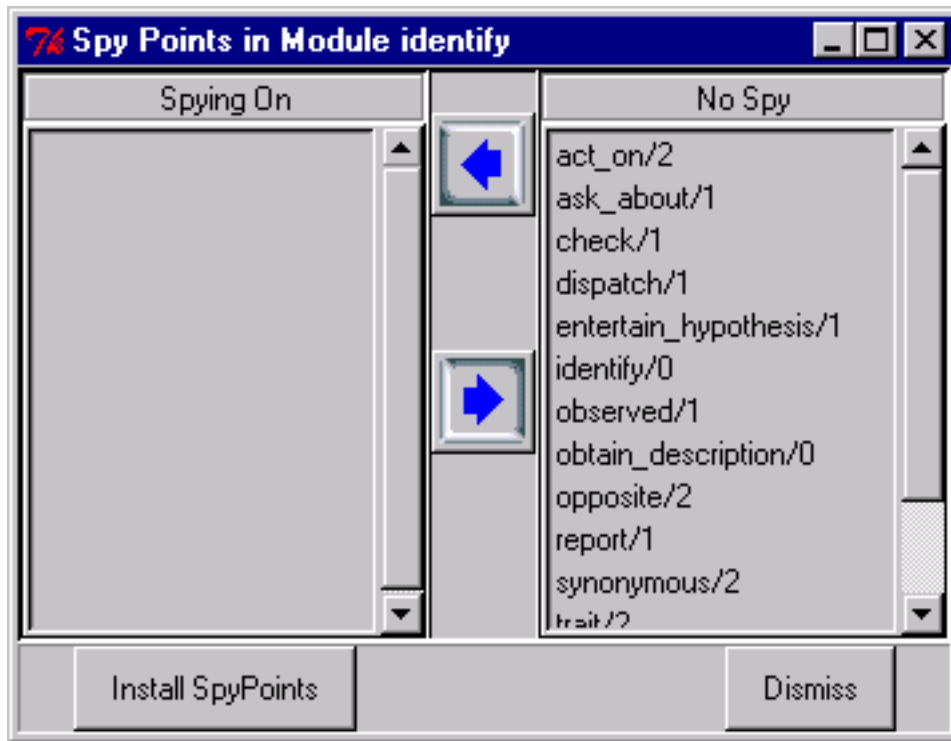
21.3 Spying with the ALS IDE

Choosing **Spy** from the **Debugger Tools** menu causes a popup scrolled list to appear. This list displays all the user/program-defined modules which have been created by consulting code, or possibly by direct program execution. For example, suppose that we have consulted the example file *hickory.pro*, which also causes the file *identify.pro* to be consulted. Then after selecting **Spy**, we would see:



Click on *identify*, and then click **OK**. A two-column spy/nospy choice window will

popup: :



Use the mouse to select items (more than one item can be selected at a time), and use the blue left and right arrows to move items between the columns. When you click on **Install SpyPoints**, all items in the left (**Spying On**) column will have spy points installed on them, and any items which were previously being spied upon, but are now in the righthand (**No Spy**) column will have their spy point removed. Several of these windows, for different modules, can be present on the screen at once, and can remain on the screen when carrying out a 4-port trace.



Development Tools

22 Using the GUI Library

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

22.1 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.

22.2 Dialogs.

22.2.1 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:



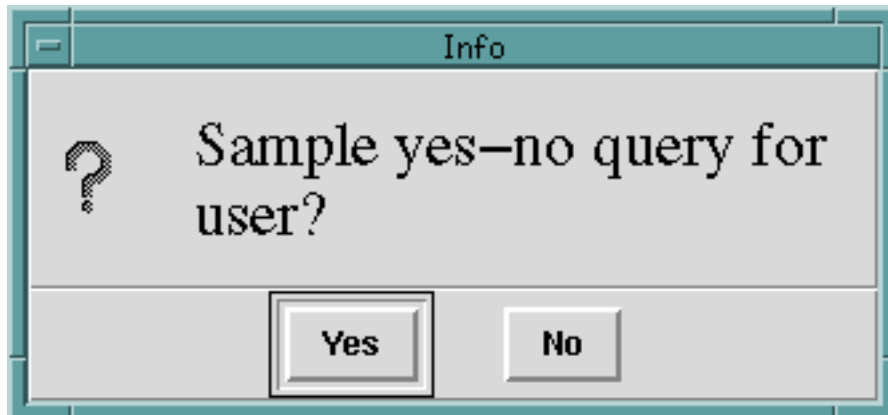
22.2.2 Yes-no dialogs.

```
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:



If the user clicks “Yes”, the result is

```
Answer = Yes
```

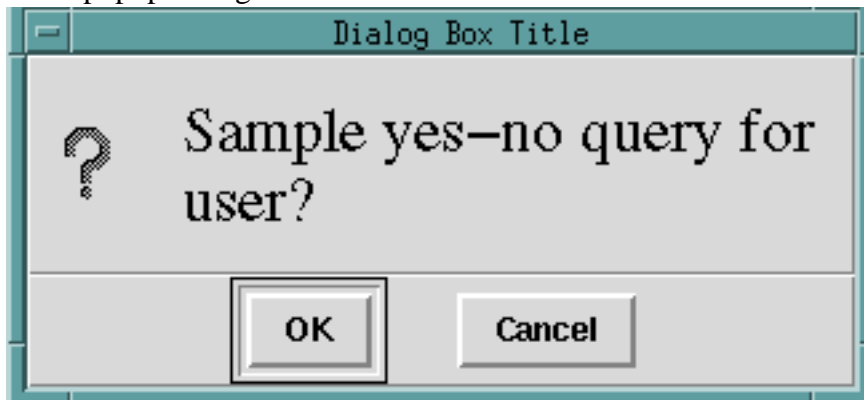
while clicking “No” yields

```
Answer = No.
```

The call

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

produces the popup dialog



Clicking “OK” yields

```
Answer = OK
```

while clicking “Cancel” yeilds

```
Answer = Cancel.
```

22.3 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).
```

produces the following popup:



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:

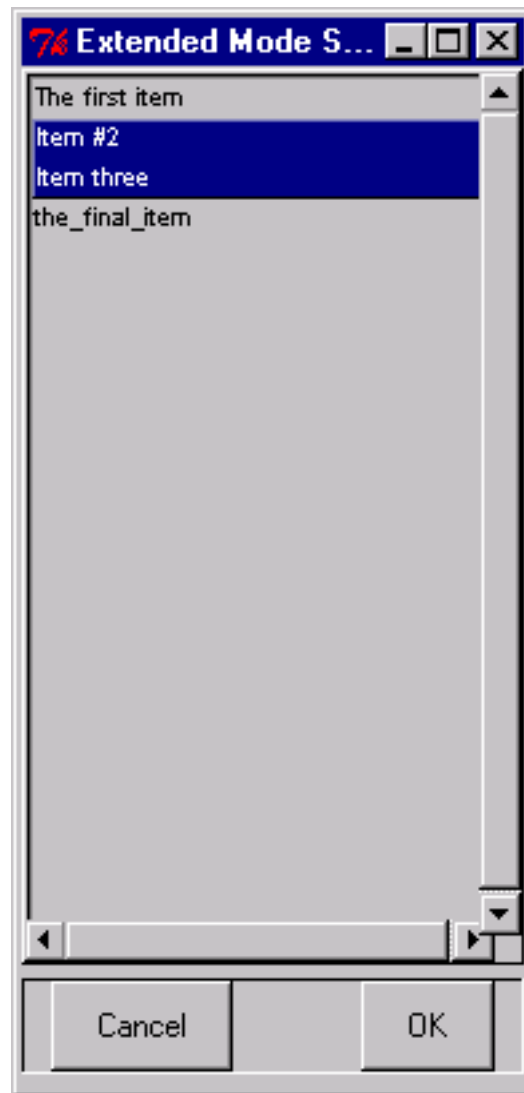
Selection = [Item three].

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 el-

ements, as seen here:



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 fol-

lows)::

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 discontinuous 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 elements between the anchor and the element under the mouse, inclusive.

22.4 Inputting atoms (answering questions).

```
atom_input_dialog(Msg, Atom)
:-
    atom_input_dialog(Msg, 'Input', Atom).
atom_input_dialog(Msg, Title, Atom)
:-
    atom_input_dialog(tcli, Msg, Title, Atom).
atom_input_dialog(Interp, Msg, Title, Atom)
```

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

```
?- atom_input_dialog('Please input something:',
    Atom).
```

will popup the following window:

If the user types

This is what I typed in ...

then the result would be

Atom = This is what I typed in ...

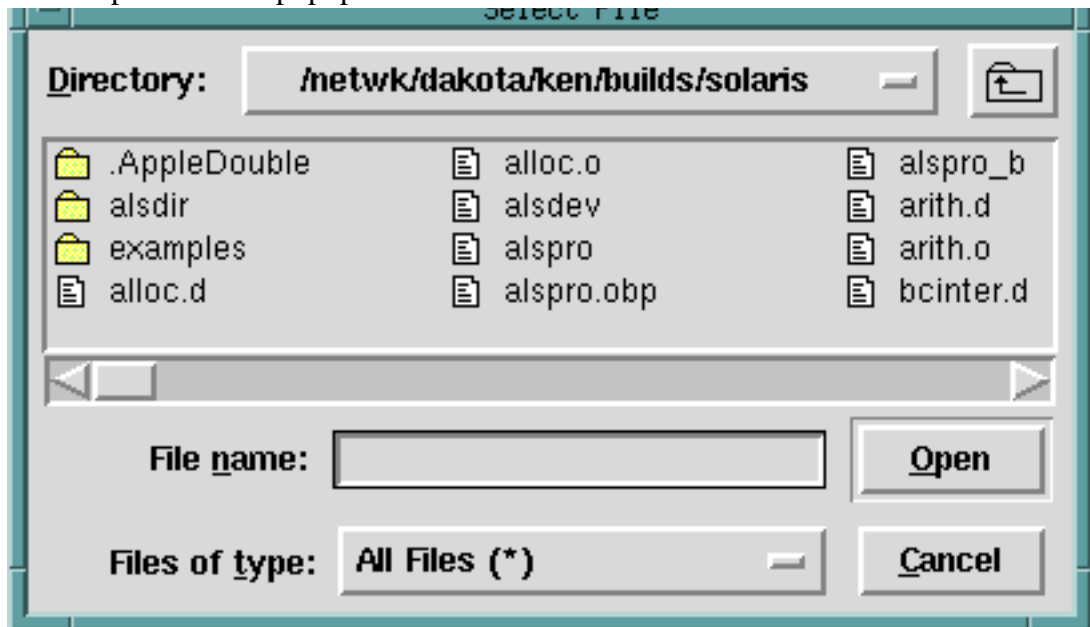
22.5 File selection dialogs

```
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).
```

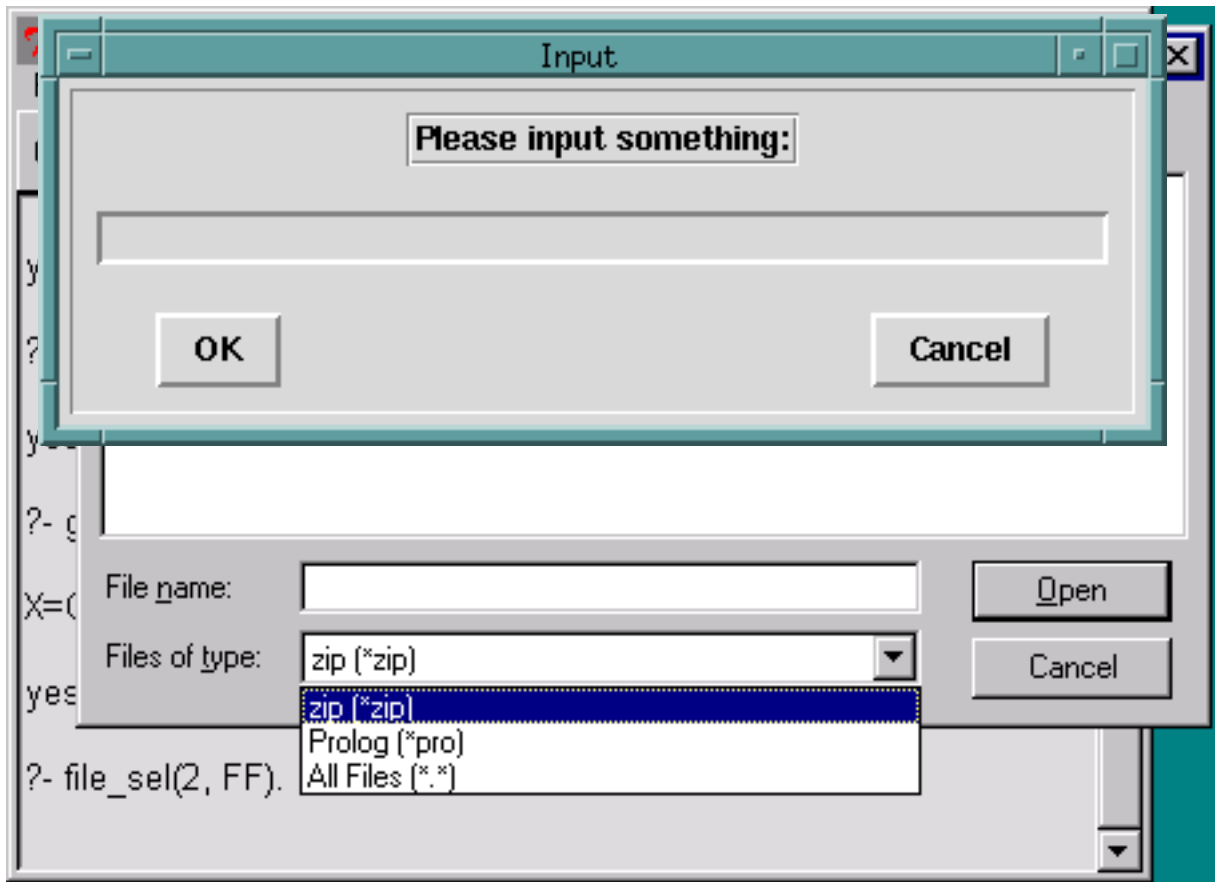
would produce this popup:



The call

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

would produce



22.6 Displaying Images.

These routines provide simply 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



22.7 Adding to the ALS IDE main menubar.

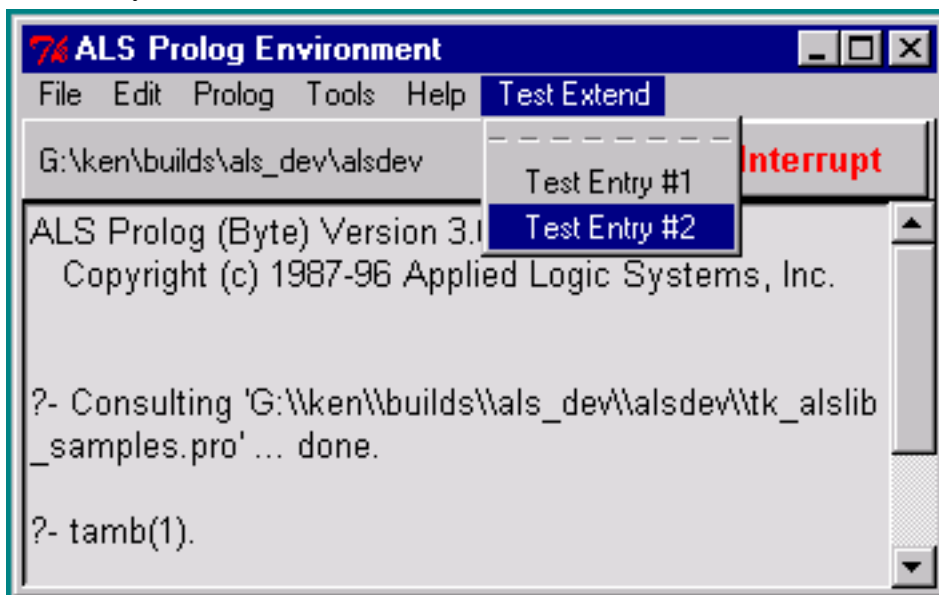
Simple additions to the main menubar are often useful. The general call to accomplish this is:

```
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:



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 + Behavior

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 choosing 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, `TclEntry` can be any Tcl/Tk expression for evaluation, 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 indices 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.

23 ALS Prolog - TCL/TK Interface

23.1 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 evaluation 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 variables to computed values.

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

23.2 Prolog to Tcl Type Conversion

23.3 Prolog to Tcl Interface Predicates

23.3.1 `tcl_new(?Interpreter)` `tk_new(?Interpreter)`

`Tcl_new` creates a new Tcl interpreter. If the `Interpreter` argument is a variable, then a unique name is generated for the interpreter. If `Interpreter` is an atom, the new Tcl interpreter is given that name.

`Tk_new` functions the same as `tcl_new`, except that the created Tcl interpreter is initialized with the Tk 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)

23.3.2 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)

23.3.3 **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)

23.3.4 **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.

23.4 Tcl Prolog Interface

23.4.1 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 variable. 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}.

23.5 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 convenient to create a package which includes Tcl/Tk so that the application will run correctly even on systems without Tcl/Tk.

Basically 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.



24 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. You 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_al spro` 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 appropriate 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_al spro <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`.

25 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 processed by using the appropriate pull down menu choice in the ALS development environment, or by using the following ALS Prolog command-line interaction (user input is shown in bold):

```

?-comptype.
Source file (*.typ, *. only) =wintypes.
yes.
?-
```

The predicates for invoking type compilation are treated as part of the ALS library, and so on systems supporting command-line shells or scripts, the following operating system command line will accomplish the same task:

```
alspro -g comptype_cl -p wintypes
```

(Of course, this can be packaged into an operating system shell script which makes its use even simpler, or included in a Makefile script.) Whatever the method, the result of this processing is a file **wintypes.pro** containing the following code:

```
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.

25.1 Specifying Structure Definitions

Structure definitions must occur in a file with a **.typ** extension. By default, the corresponding ALS Prolog code is placed in a file of the same name with a **.pro** extension. The general structure of such a file is as follows:

```
target_file(...)      % optional

module(...).

defStruct(...).
defStruct(...).
...
endmod.

module(...).

defStruct(...).
defStruct(...).
...
endmod.
...
```

The file begins with an optional target file declaration:

```
target_file(FileName).
```

specifying the file in which to place the generated code. `FileName` should be an atom; the complete file name will be `FileName.pro`. If the `target_file` declaration is missing, `FileName` will be taken to be identical with the name of the source file.

The remainder of the file is made up of one or more module blocks bracketed by the `module(Mod) . . . endmod.` declaration. This declaration specifies the module in which the code generated for the definitions within the block is to reside. Any definitions not contained within such a module declaration are taken to reside in module `user`.

Within a module block occur one or more structure definitions, or *defStructs*. 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:

25.1.1 `accessPred`

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

25.1.2 `setPred`

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

25.1.3 `makePred`

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

25.1.4 `structLabel`

The name of the functor of the structure defined.

25.1.5 propertiesList

This is a list of slot specifications. A *slot specification* is one 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.]

25.2 Compiling Structure Definitions

The compilation process uses the defStruct declarations to generate a file of ALS Prolog clauses implementing the access, change, and creation predicates specified. The predicate invoking this process is comptype/0 contained in the file **type-comp.pro**. It can be directly invoked from Prolog, as shown below for the **wintypes** example:

```
?-comptype.  
Source file (*.typ, *. only) =wintypes.  
yes.  
?-
```

On systems supporting command-line shells or scripts, the following operating system command line will accomplish the same task:

```
alspro -g comptype_cl -p wintypes
```

The great virtue of such operating system command lines is that they can be placed in makefiles.

25.3 Using Compiled 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=acpracpr(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=stprstpr(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=mkprmkpr(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=mkprmkpr(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 position corresponding to `SlotName`; these “local defaults” will override any “global defaults” specified in the `defStruct`.

`makePred=xmkprmkpr(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 `xmkprmkpr(Struct, SlotVars)` is equivalent to

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



26 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.

26.1 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. Multiple inheritance is supported. 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.

<i>slot name</i>	<i>slot value</i>
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. Each object is required to have a name which is used in sending messages to the object. However, essentially nameless objects are supported by allowing the system to generate a name for an object. An object's *methods* are determined by the class to which it belongs. For convenience, an object can also possess strictly local methods, which may override methods for its class (or the superclasses its class inherits from). In essence, such an object belongs to an implicit class having only one member.

A *class* is determined by three things:

- A local state-schemata 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.

The complete *state-schemata* 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. A restriction is imposed that the various local state-schemata must be disjoint in that no two slots in distinct local-schemata may have the same names. [The actual restriction is slightly weaker: no child class can inherit from parent classes with intersecting local-schemata, nor can a child class's local state-schemata intersect with a parent's local state schemata.] Classes are also required to have names -- these are principally used in defining objects. 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-instantiated message paradigm of Concurrent Prolog [Ref]. The ALS ObjectPro system is integrated with the module system of ALS Prolog. Any module which wishes to utilize the facilities of ALS ObjectPro must use the module `objects`. Class and object defini-

tions 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.

26.2 Defining Objects and Sending Messages

An object is defined by an expression of the form

```
defineObject(Arg)
```

where Arg is a list of *equations* of the form

```
Keyword = Value
```

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

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

The `instanceOf` keyword equation is the only required equation; if the name equation is omitted, the system generates a name for the object. Any atom can serve as the name of an object or a class. Here is an example of a simple object definition:

```
defineObject([name=engine1, instanceOf=iC_Engine ])
```

The equations appearing on a list which is the Value for a values equation are 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. 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.

Objects may also be defined by expressions of the form

```
defineObject(Arg) :- Condition.
```

In this case, the goal

```
:- Condition
```

is first run and the the `defineObject(Arg)` goal is executed.

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

```
send(ObjectName, Message)
```

where `ObjectName` is the 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).

Defining an object causes the code implementing the object to be generated, and also causes the global variable which will point to the object's state to be allocated. However, the term implementing the object's state is not constructed by the definition process, and so the object does not yet really exist. To bring the object into existence (i.e., to cause it's state term to be constructed and a pointer to it installed in the object's global variable), one must send the `initialize` message to the object:

```
send(ObjectName, initialize)
```

Not only is the object's state term constructed, but if the object definition includes a `values` line or the object's class includes an equational constraint (see the next section), these values will be installed in the appropriate slots. The (generated) code representing the object resides in whatever module the definition occupied. (Section 26.7 (*Using ObjectPro Tools*) for more detail.) Certain information about the object is stored in the distinguished module `objects`.

26.3 Defining Classes

A class is defined by an expression of the form

```
defineClass(Arg)
```

where `Arg` 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	- list of atoms (names of classes)
addl_slots	- list of atoms (names of local slots)
constrs	- list of constraint expressions
export	- yes or no
action	- atom

The name equation and the subclassOf equations are both required.

The ObjectsPro system pre-defines one top-level class named `genericObjects`; all classes are subclasses of the `genericObjects` class. `genericObjects` provides one visible slot, `myName`, which is always instantiated to the object's name. A class is said to be an *immediate subclass* of the classes named on the `subclassOf` list. The relation *subclass* is the transitive closure of the immediate subclass relation. The atoms on the `addl_slots` list name slots in structure defining the state of objects which are instances of this class. 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. For this reason, 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] ])
defineClass([name=landBasedPlane,
```

```
subclassOf=[wingedVehicle,wheeledVehicle])
```

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

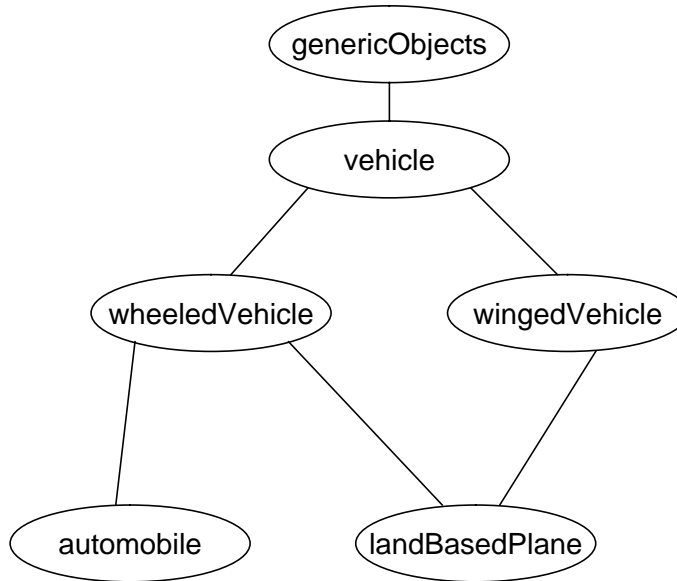


Figure 20. Example Class Inheritance Relations.

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]
landBasedPlane   - [locomotionType,
  powerSource,numWings,
                    numWheels]
```

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. If an `export = yes` equation appears on the `Arg` list of a class definition, the class (and the objects which are instances of that class) are exported from the module in which the definition takes place. This means that a `send` to an instance of this class can be carried out from any module which uses both the module `objects` and the module in which the class definition took place. Practically, the effect of the `export=yes` declaration is that the generated code for both the class and its object instances is exported from the defining module `M`. [Impl fix: export the object code.] ??????

Similarly to the situation for objects, classes may also be defined by expressions of the form

```
defineClass(Arg) :- Condition.
```

In this case, the goal

```
:- Condition
```

is first run and then the `defineClass(Arg)` goal is executed.

A call `send(Object, Message)` attempted in another module `N` will fail under any of the following conditions:

1. `N` does not use `objects`;
2. `N` does not use `M`;
3. `Object` is an instance of class `C` which was defined in `M`, but was not exported.

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 are 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. `Condition` 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 `Condition`.

Here is a class specification including a constraint:

```
defineClass([name=engine,
            subclassOf=[genericObjects],
            addl_slots=
                [powerType,fuel,engineClass,
                cur_rpm,running,temp],
            constra=
                [engineClass<
```

```
        [internalCombustion, steam, electric]]  
    ] )
```

The (generated) code representing a defined class resides in whatever module the definition occupied (see Section 26.7 - Using ObjectPro Tools - on page 343-343. for more detail.). Certain information about the class is stored in the distinguished module `objects`.

26.4 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 name of the predicate can be specified by using the `action` line in the class definition. If the `action` line is omitted, the name of the predicate is assumed to be:

```
<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 structure of `State`, but two pairs of constructs provide for access to the slots:

```
StateOrName^SlotDescrip := Value  
VarOrValue := StateOrName^SlotDescrip  
  
setObjStruct(SlotDescrip, State, Value)  
accessObjStruct(SlotDescrip, State, VarOrValue)
```

In both pairs, the first call destructively updates the slot `SlotName` of `State` to have value `Value`. The second call of each pair accesses the slot `SlotName` of `State` and unifies the value obtained with `VarOrValue`. `StateOrName` is either the name of an object or an object state. In the latter case, the object state is either supplied automatically by the ObjectPro implementation of `send`, or is obtained via the `your_state` method described below. `Value` should be a ground

Prolog term, while `VarOrValue` can really be any Prolog term. The value of `SlotDescrip` 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. The second pair of calls is the more primitive: the procedure `:=` is defined in terms of them.

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).
genericObjectsAction(your_state(State), State).
```

The `your_state` method is provided primarily for complex situations involving

compound objects.

26.5 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).
```

Here is a session using this object at the ALS Prolog command line:

```
?- [-oopex_stack].
Reconsulting oopex_stack ...
oopex_stack reconsulted
yes.

?- send(stack, initialize).
yes.

?- send(stack, push(a)).
yes.

?- send(stack, push(dd)).
yes.

?- send(stack, cur_depth(X)).
X = 2
yes.

?- send(stack, cur_stack(X)).
X = [dd, a]
yes.

?- send(stack, pop(X)).
X = dd
yes.

?- send(stack, cur_stack(X)).
X = [a]
```

yes.

?- send(stack,cur_depth(X)).

X = 1

yes.

The second example illustrates the construction of compound objects:

```
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],
            constra=
                [engineClass<

                [internalCombustion,steam,electric]]
            ] ).
defineClass([name=iC_Engine,
            subclassOf=[engine],
            addl_slots=[manuf],
            constra = [engineClass =
            internalCombustion]
            ] ).
```

```

defineObject([name=engine1, instanceOf=iC_Engine]).
defineObject([name=auto1,
               instanceOf=automobile,
               values=[engine=$object(engine1)]
              ]).
defineObject([name=auto2,
               instanceOf=automobile,
               values=[engine=

               defineObject([instanceOf=iC_Engine]])
              ]).

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
    ).
```

Here are some simple command-line interactions with these objects:

```
?- send(auto1,initialize).
yes.
```

```
?- send(auto1,status(X)).
X = off
yes.
```

```
?- send(engine1,get_value(running,X)).
X = nil
yes.
```

```
?- send(auto1,start).
yes.
```

```
?- send(auto1,status(X)).
X = running
yes.
```

```
?- Y := auto1^engine^running.
Y = yes
yes.
```

```
?- engine1^running := no.
yes.
```

```
?- send(auto1,status(X)).
X = off
yes.
```

Notice that the engine component of `auto2` is an anonymous object: the definition does not provide a name. Instead, the system generates a name for internal use.

26.6 OOP_Event Queues

The one-way connection provided by `send/2` between sender and recipient is quite direct: the sender must know the name of the object to which the message is directed. The connection is one-way in that the recipient need know nothing about the sender, but only needs to have a method to deal with the incoming message.

There are circumstances under which such a tight connection is awkward, and what would be useful would be a modified broadcast mechanism. Specifically, the 'sender' of the message need not know to whom the message is to be sent, simply that it is to be broadcast. In compensation for the weakening of the requirements on the sender, we will impose a slight requirement on the recipients, namely that the recipient's *interest* in the message be established in some way.

The ObjectPro mechanism providing these facilities is the OOP_Event Queue mechanism. By definition, *oop events* are simply oop messages. And *oop event forms* are also messages, but they may be partially instantiated versions of particular messages. For example, if a particular message normally is a ground term of the form `change_dir(TgtDir)`, where `TgtDir` is an atom, then `change_dir(TgtDir)` when `TgtDir` is an uninstantiated variable would be an acceptable oop event form. Using this mechanism, any routine which changes the current working directory can broadcast the fact of a change to any routines interested. The latter, for example, could be concerned with maintenance of a window showing the current working directory.

Oop event queues are associated with oop event forms: Objects express interest in message events of a given form, and other objects wishing to broadcast messages of a given form place them in the appropriate queue. There are two predicates for manipulating the queues.

`insert_oop_event_request/2`

`insert_oop_event_request(EventForm, Object)`

`insert_oop_event_request(+, +)`

This predicate causes a request to be entered from `Object` to be sent all broadcasts of oop message events of the form `EventForm`. Note that this does not cause all messages of the form `EventForm` to be directed to `Object`, only those which are

broadcast (using `queue_oop_event/1`). Also note that since this is a Prolog predicate, it can be invoked either from within an object method, or can be invoked directly in Prolog code.

`queue_oop_event/1`
`queue_oop_event(Message)`
`queue_oop_event(+)`

This predicate causes the oop Message to be placed on all oop event queues which it matches, with the effect that Message is sent (via `send/2`) to all objects for which an expression of interest has been made (via `insert_oop_request/2`). Note that, for convenience, both of the following are equivalent to `queue_oop_event(Message)` :

```
send(anyone, Message),  
send(any_object, Message).
```

26.7 Using ObjectPro Tools

The ALS ObjectPro generators are used by grouping class and object definitions in special files with extension `.oop`, and then processing these files with the generators. The ObjectPro generators (as well as the necessary runtime support routines) are installed as elements of the ALS Library. In essence, they are always available. Note that since the ObjectPro tools are configured as ALS Library components, classes and objects can be defined on the fly in program, and then processed with the generators.

If one has grouped class and object definitions in one or more files with extension `.oop`, one simply invokes the predicate `objectProcessFile/1` on a file name or a list of file names (in both cases, just the file name without the `'oop'` extension). This call generates a file or files with extension `.pro` containing the Prolog code implementing the class and object definitions contained in the source files. Any code in the source files which is not an object or class definition is passed through to the generated file unchanged. Thus class methods can be included in a `.oop` file, or they can be placed in a separate file. The object definitions for a given class can be included in the same file as the class definitions, or can be placed in a separate file. Within a `.oop` file, the class and object definitions appear as terms, as in all the examples given earlier.

There are a number of predicates which invoke and control object and class definition processing.

objectProcessFiles/1

objectProcessFiles(FileNames)

objectProcessFiles(+)

Recursively applies `opf/1` (or, `objectProcessFile/2`, ignoring the second argument) to each file name on the list `FileNames`.

opf/1

opf(File)

opf(+)

Defined by:

```
opf(File) :- objectProcessFile(File, _).
```

objectProcessFile/2

objectProcessFile(FileName, Records)

objectProcessFile(+, -)

`FileName` should be a filename without any extension. This predicate first constructs two derivative filenames:

```
SourceFile = FileName.oop and TargetFile =  
    FileName.pro
```

Then the following call is made:

```
objectProcessFile(SourceFile, TargetFile, Records).
```

objectProcessFile/3

objectProcessFile(SourceFile, TargetFile, Records)

objectProcessFile(SourceFile, TargetFile, Records)

Reads the terms occurring in `SourceFile` into a list `SourceTerms`, invokes the call

```
do_objectProcess(SourceTerms, SinkList, [],  
    RecordInfo),
```

and then uses `write_clauses/3` to write `SinkList` into `TargetFile`, after writing some initial header information into the file..

do_objectProcess/4

do_objectProcess(SourceList,SinkList,SinkListTail,RecordInfo)

do_objectProcess(+,?,?,?)

do_objectProcess/5

do_objectProcess(SourceList,Module,SinkList,SinkListTail,RecordInfo)

do_objectProcess(+,+,?,?,?)

Both predicates recursively process `SourceList` to produce the pair

`(SinkList, SinkListTail),`

where `SinkListTail` is the (normally uninstantiated) tail of `SinkList`. In addition, the list `RecordInfo` of (sometimes) useful information is produced.

set_object_messages/1

set_object_messages(Value)

set_object_messages(+)

`Value` should be one of the atoms `on` or `off`. Turns on or off a simple message facility providing information about the definitions of objects and classes as they are processed.

Besides these basic object processing tools, there are several predicates which are useful for ObjectPro development, allowing one to examine various aspects of the states of objects, etc.

mods_with_objs/1

mods_with_objs(ModsList)

mods_with_objs(?)

Unifies `ModsList` with a list of all modules which currently contain the definition of some object.

mods_with_classes/1

mods_with_classes(ModsList)

mods_with_classes(?)

Unifies `ModSList` with a list of all modules which currently contain the definition of some class.

objs_in_mod/2

objs_in_mod(Mod, ObjsList)

objs_in_mod(+, ?)

Unifies `ObjsList` with the sorted list of all objects defined in `Mod`.

classes_in_mod/2

classes_in_mod(Mod, ClassesList)

classes_in_mod(+, ?)

Unifies `ClassesList` with the sorted list of all classes defined in `Mod`.

don/1

don(ObjName)

don(+)

If `ObjName` is the name of an object, obtains the internal state `S` of the associated object, and then calls

`dpos(S).`

dpos/1

dpos(ObjState)

dpos(+)

If `ObjState` is the internal state of an object, prints out on the terminal or worksheet a list of equations consisting of the tag names for the slots of the object, paired with the current values in those slots. Since for a large object, this dump can consist of much data, the next two predicates are quite useful.

sdon/1

sdon(ObjName)

sdon(+)

Obtains the internal state `S` of the object named by `ObjName`, and calls `sdos(S).`

sdos/1

sdos(ObjState)

sdos(+)

If `ObjState` is the internal state of an object, first obtains the name `ObjName` of the object, then makes a call

```
objects:slots_profile_for(ObjectName,  
    SlotProfileList)
```

to obtain `SlotProfileList`, which is a list of slot names (tags), and then dumps, on the terminal or worksheet, the slot-value equations for just those slots whose tags occur on `SlotProfileList`.

set_obj_profile/2

set_obj_profile(SlotNameList, ObjName)

set_obj_profile(+, +)

If `ObjName` is the name of an object, and `SlotNameList` is a list of names of slots in the state of the object, then this list is installed as the `SlotProfileList` for `ObjName` for use with `sdos/1`.

obj_slots/2

obj_slots(ObjName, SlotsList)

obj_slots(+, -)

If `ObjName` is the name of an object, unifies `SlotsList` with the sorted list of tags (names) of slots in the state of the object.

26.8 Command Line Execution

27 Cross Reference Analysis

ProCref is an adaptation of the widely-used "cref" concept of simple program analysis to the setting of ALS Prolog. It provides a moderately powerful, and quite flexible, tool for developing information about program structure. It normally operates in two phases: An initial analytic phase during which basic calling information about a program is developed, and later interactive phase during which further analysis can be requested, program information can be viewed on the screen, and reports can be generated for printing. In addition, the tool is extensible in several ways which will be described below.

A program is referred to as a *suite*; it can consist of one or more files. When the program consists of a single file, the suite name is normally identified with the file name (sans extension). When the program consists of more than one file, the suite is usually defined by a suite specification file. This is a file whose name is the name of the suite, and whose extension is *'crf'*; for example, *my_prog.crf*. The file must contain an entry of the form

```
files = [file1, file2, ....].
```

and may possibly contain an entry of the form

```
dir = <directory path>.
```

Both of these entries must be Prolog terms, including terminal periods and appropriate quoting. Here is a sample contents for the suite specification file *my_prog.crf*:

```
files = ['foobar.pro', 'zipit.pro', 'silly.pro'].
dir = '/frege/tests/my_prog'.
```

The directory path attached to *dir* should be a path to the directory in which the files occurring in the list attached to *files* reside.

When performing an program analysis for the first time, one normally starts *cref* with one of the following two goals:

```
?- cref( <SuiteName> ).
?- cref( <SuiteName>, <Options> ).
```

<Options> is a list which will be described later. Once started, *cref* builds an

in-memory avl tree containing basic call information and other data, keeping you posted about its progress with brief messages. Once the tree is complete, control is passed to `cref_shell`, which is an interactive shell for exploring the tree and developing further information. Among the facilities offered by the shell is one which is quite useful: the ability to save the analysis tree to a file, and resume the work at a later time. Another useful feature is the fact that the tree is actually stored in a global ALS Prolog variable. Consequently, it is possible to exit from the `cref_shell` to the ALS Prolog shell, perform some actions, and then re-enter the `cref_shell`.

Some of the details in the following will be modified or extended as ALS Prolog is adapted to the ISO Prolog standard.

27.1 Starting and stopping `cref_shell`.

As noted above, `cref/1` and `cref/2` automatically start `cref_shell` when basic analytical processing is complete. The commands below allow one to directly start and stop `cref_shell`.

27.1.1 Exiting from `cref_shell`.

Each of the following causes `cref_shell` to exit to the normal Prolog shell.

`quit.`

`halt.`

`exit.`

Typing the end-of-file character.

27.1.2 Re-entering `cref_shell`.

`restart_cref_shell/0`

`rcl/0` Restarts `cref_shell` exactly in the state when it was last exited.

27.1.3 Directly starting `cref_shell`.

`cref_shell/0.`

Starts `cref_shell` using a new shell history structure. If an analysis tree, with its accompanying miscellaneous information structure, is present in the appropriate global variables, this tree and information structure are utilized. Otherwise, an empty tree and information structure are created.

Executing a single `cref_shell` command from Prolog.

`cx/1`

`cx(Cmd)` Calling `cx(Cmd)` from Prolog is equivalent to the following sequence of three interactive commands:

```
?- cref_shell.  
cref_shell(0): Cmd.  
cref_shell(1): quit.
```

27.2 Basic `cref_shell` Commands.

27.2.1 Getting simple statistics.

`stats` Prints out the number of nodes in the tree and the depth of the tree.

27.2.2 Getting help.

`help` Prints out a brief synopsis of all commands.

27.2.3 History.

The `cref_shell` maintains a history of the commands which have been issued, together with the outputs, if any, which resulted from each command. The number of each interaction is shown in the prompt, as follows:

```
cref(12):
```

The following commands allow one to explore the history.

`hist(N)` Prints out the history back to (including) command number `N`.

`hist` Equivalent to `hist(1)`.

`out(N)` Types out (again) the output of command number `N`.

In addition, one can transfer the output of the last command executed to the Prolog database for later processing, using the following:

`store` If the last command executed was number `N`, and if `OUTPUT` was the output of that command, then `scrfl(N, OUTPUT)` is asserted in module `user`.

`store(N)` If `OUTPUT` was the output of command `N`, then `scrfl(N, OUTPUT)` is asserted in module `user`.

27.2.4 Showing the files processed.

`files` Prints out the list of files which were processed to produce the tree.

27.3 Displaying information from the tree.

27.3.1 Displaying the entire tree.

There are two commands which will print out the entire tree, in rather different formats:

`inorder(also, write)`

 Makes an in-order traversal of the tree, printing out information from each node.

`writetree`

 Traverses the tree, printing out each node in internal form, using indentation to suggest tree structure

27.3.2 Displaying individual nodes.

Individual nodes in the tree are identified by keys which are predicate identifiers of the form `P/N`, where `P` is the predicate name, and `N` is the arity of the predicate.

`search(P/N)`

 Searches the tree for the node with key `P/N`. If such a node is found, prints out the information contained in the node.

27.4 Saving and restoring Trees; Contexts.

Several commands allow one to store the analysis tree in a file. Both the tree and the accompanying miscellaneous information structure are stored. At a later time, perhaps after having exiting and re-entered ALS Prolog, one can re-load the tree and pursue further analysis.

`savetree(FileName)`

`FileName` must be an atom. Saves the tree and information structure in the file *FileName.cft*.

`savetree`

Equivalent to `savetree(SuiteName)`.

`loadtree(FileName)`

If `FileName` is an atom, and if the file *FileName.cft* exists and contains an analysis tree and related information, the tree and information are read from the file, and the newly read information becomes the current analytical context for `cref_shell`.

As noted earlier, the analysis tree and its related information are stored in global ALS Prolog variables. The values of these variables are not simply discarded when a new tree is read in. Instead, `cref_shell` maintains a stack of contexts (consisting of a tree and its related information). The tree which was the current context when the `loadtree` command was issued (i.e., which was in the global variables) is pushed onto this stack, and the new tree is then loaded into the global variables. The entries on the stack are identified by their suite names. Several commands allow one to examine and manipulate the stack of contexts.

`ctxs` Prints out the names of the stored contexts, in order from most recent to oldest.

`restore(Name)`

If `Name` is the name of a context stored on the context stack, then the context NC associated with `Name` is accessed, the association be-

tween Name and NC is removed from the stack, leaving a smaller stack, IS, the current context (in the global variables) is pushed onto IS yielding a new stack NS, and the extracted context NC is loaded into the global variables.

27.5 Requesting further analysis.

When the analysis tree is initially built by `cref/[1.2]`, the only information developed is which procedures each program procedure immediately calls. `cref_shell` allows one to develop further information. The simplest way to do this is to use the command `xall`, which simply runs all of the following specific commands:

`depends` Computes, for each predicate P/N in the tree, the closure under the 'calls' relation. This closure is stored as the 'dependson' list, and consists of all procedures called by P/N, or called by a procedure called by P/N, etc., to any depth.

`exports` Computes the list of all pairs M-L, where M is a module occurring in the program, and L is the list of all procedures which are defined in M and exported from M.

`calledby`

Computes the converse of the 'calls' relation.

`show_undefs`

`undefs` Computes, and prints on the terminal, the list of all procedures which are called somewhere in the program, but for which there is no definition in the program. This list can be saved using the `store` command.

The `write` and `inorder` commands print out all of the information calculated by the above commands. The information is labelled as follows:

`mod` = the module in which the predicate is defined

`exported` = true/false, according as the predicate is exported from its defining

module.

`files` = the list of files in which one or more defining clauses for the predicate occur.

`clausecount` = the number of non-fact clauses occurring in the definition of the procedure.

`factcount` = the number facts occurring in the definition of the procedure.

`calls` = the list of predicates which are called in one or more clauses of the procedure.

`calledby` = the list of all procedures in the program which call this predicate.

`dependson` = the list of all procedures ultimately called by this procedure (i.e., the image of the current procedure under the closure of the 'calls' relation).

`basis` = the subset of the dependson list for this procedure which are undefined in the program.

Some of this information can be viewed in different formats, as follows:

`mods` Prints out the list of all modules, other than `user`, occurring in the program, showing for each module, the list of all modules which it uses, and the list of all files in which defining clauses for the module occur.

`exp(Name)`

If `Name` is the name of a module in the program, prints out the list of all procedures which are exported from module `Name`.

`called_in(Name)`

If `Name` is the name of a module in the program, prints out the list of all procedures which are called in the module `Name`.

`called_in_file(Name)`

If Name is the name of a file making up the program, prints out the list of all procedures which are called in the file Name .

`def_in_file(Name)`

If Name is the name of a file making up the program, prints out the list of all procedures which are defined in the file Name .

`mis` Dumps the miscellaneous data structure on the terminal.

27.6 Directing Output.

Sometimes it is desirable to direct output from commands to files or other sinks instead of the terminal. This is easily done, since `cref_shell` supports a re-direction construct similar to many operating system shells. If `Cmd` is a `cref_shell` command which produces output, and if `Tgt` is either an open stream, an alias for an open stream, or a sink descriptor appropriate to `open/4`, then output is redirected by submitting the following to `cref_shell`:

`Cmd > Tgt.`

If `Tgt` is an open stream or an alias for an open stream, the output is simply written to `Tgt`. If `Tgt` is a sink descriptor appropriate to `open/4`, then before `Cmd` is executed, the goal

`open(Tgt, write, TgtS, [])`

is called, `Cmd` is then executed with output going to `TgtS`, and finally, `TgtS` is closed. Output from multiple commands can be directed to, say a file, by exploiting the fact that `cref_shell` allows one to group commands using parentheses and commas. Thus to execute the commands `mods`, `files`, and `inorder`, with output going to the file *my_test*, one would execute

`(mods, files, inorder) > my_test.`

27.7 User-defined processing.

It is fairly easy to extend the repertoire of processing which `cref_shell` can provide. `cref_shell` provides a method whereby a user-defined node processing predicate is "walked across" the tree so that the details of the tree can be largely ig-

nored. The command to invoke this is `walk1(PredName)`, where `PredName` is the name of an appropriately defined 6-ary predicate. The required arguments format of such a predicate is as follows:

```
<Pred>(Node, TopNode, Key, MIS, ResultList,
        ResultTail)
```

The arguments are as follows:

`Node` = a node from the analysis tree

`TopNode` = the top nodes of the tree

`Key` = the Key for node (normally the P/N of the node)

`MIS` = the (global) miscellaneous data structure

`ResultList` = if a result list is being accumulated (to be returned), this is the input (variable) for the list yet to be computed.

`ResultTail` = if a result list is being accumulated (to be returned), this is the variable for the tail of the list following computation at this node.

For example, the command `undefs` is executed by internally executing `walk(gundefs)`, where `gundefs/6` is defined by

```
gundefs(Node, _, Key, MIS, [Key | ResultTail],
        ResultTail)
:-
    accessCRF(factcount, Node, 0),
    accessCRF(clausecount, Node, 0),
    !.

gundefs(_, _, _, _, ResultTail, ResultTail).
```

Note that submitting the goal `walk(gundefs)` won't work, since `cref_shell` will look for a definition of `gundefs/6` in module `user`, instead of in module `cref`, which is where it looks for internally submitted walk goals. However, if

you assert

```
myundefs(A,B,C,D,E,F) :- cref:gundefs(A,B,C,D,E,F).
```

and then submit, `walk(myundefs)`, things will work as advertised. If you wish to get your hands on the accumulated output (e.g., when making a `cx` call into `cref_shell`) then use the two argument version of `walk`:

```
walk(<Pred>, Output).
```

For example, if you have defined `myundef/6` as above, then consider:

```
?- cx(walk(myundefs, Output)).  
Output = [.....]
```

If you are only doing some processing at each node (e.g., some printing) and don't want to accumulate any output, simply identify the last two arguments of your predicate, as in the second clause for `gundefs` above.

Finally, in order to create interesting predicates such as `gundefs`, one needs to know how to access the elements of a `Node` and the elements of `MIS`. Both of these structures are `defStructs` specified in the file `crefstr.typ`, which should be found either in *alsdir/builtins* or *alsdir/library*, or possibly another subdir of *alsdir*. Here are the specifications of each:

```
defStruct(crefStructs,  
  [propertiesList = [  
    pred,          %%  
    arity,         %%  
    mod/user,      %% Mod where defined  
    files/[],      %% Files where defined  
    exported/false,%% true/false  
    calls/[],      %% Other preds called by pred/arity  
    basis/nil,     %% Ultimate preds called by pred/  
arity  
    calledby/[],   %% Other preds calling this pred/  
arity  
    calledcount/0, %% Number of calls to this pred  
    factcount/0,  %% Number of facts  
    clausecount/0,%% Number of clauses
```

```

    depthcount/0,%%
    callinmod/[],%% Mods where called
    importto/[],%% Mods where imported
    dependson/nil,%%
    whereasserted/[]%%
  ],
  accessPred =accessCRF,
  setPred =setCRF,
  makePred = makeCRF,
  structLabel = crfCRF
] ).

defStruct(mi,
  [propertiesList = [
    files/[], %% files encountered
    files_mods/[],%% File+[ModsInFile]
    files_d_preds/[],%% File+[P/N DefInFile]
    files_c_preds/[],%% File+[P/N CalledInFile]
    mods/[], %% mods encountered
    mods_files/[],%% Mod+[FilesModOccursIn]
    mods_use/[],%% Mod+UseList
    mods_d_preds/[],%% Mod+[P/N DefInMod]
    mods_c_preds/[],%% Mod+[P/N CalledInMod]
    mods_exp_preds/[],%% Mod+[exported P/N]
    mods_imp_preds/[]%% Mod+[actually imported P/
N]
  ],
  accessPred =accessMI,
  setPred =setMI,
  makePred = makeMI,
  structLabel =crfMI
] ).

```

As seen in the definition of gundefs, components of a node are access by calls of the form

```
accessCRF(SlotName, Node, Val).
```

Access to the slots of the miscellaneous info structure MIS is via calls of the form

```
accessMI(SlotName, MIS, Val).
```

28 MacroPro: A Macro Processor¹ for ALS-Prolog

MacroPro provides powerful macro expansion facilities for ALS Prolog. These facilities allow the construction of macros ranging from simple aliases for constants to sophisticated tools such as 'mapcar' macros.

28.1 Introduction and Examples

A *macro specification* is a Prolog expression of the following form:

```
<pattern> => <expansion>
    [ when <goals> ]
    [ where <goals> ]
    [ with <list of clauses> ]
    [ if <expansion-time condition> ]
```

Figure 21. The Form of Macro Specifications.

Any items inside square brackets ([. . .]) are optional. If any of the optional *when*, *where*, *with*, or *if* parts occur, they must appear in the order indicated in Figure 21 (*The Form of Macro Specifications*).

Example 1.

The macro specification

```
pi => 3.1416
```

will cause all occurrences of the symbol *pi* to be replaced by the number 3.1416.

Example 2.

Given the operator declaration

```
:- op(100, fx, ++).
```

-
1. MacroPro is directly based on the work of Sei-Ich Kondoh and Takashi Chikayama, described in ***Macro Processing in Prolog***, in **Logic Programming, Proceedings of the Fifth International Conference and Symposium**, R.A. Kowalski and K.A. Bowen, eds, MIT Press, Cambridge, MA, 1988, pp 466-480.

the macro specification

```
++X => X+one
```

will cause all expressions of the pattern ++X to be replaced by X+one .

Macro specifications are entered into the system by making a call of the form:

```
define_macro((Macro_specification))
```

where Macro_specification is as specified above. Such expressions are called **macro definitions**. Note that the extra parentheses () are needed since "=>", "when", "where", "if" are defined as operators with precedences 1200, 1190, 1180, 1160. Typically, a macro definition is made as a goal in a program file:

```
:- define_macro((Macro_specification)).
```

The macro expansion is performed by making a call

```
expand( Clause , ExpandedClause )
```

where ExpandedClause is the clause which results from macro expansion based on definitions currently in effect.

28.2 Mode of Operation of MacroPro

Each clause is processed individually by MacroPro. Each individual clause is processed using the macro specifications in the order in which they were recorded.

The following major steps are performed when macro processing a clause:

1. Expand any macros appearing in the head of the clause.
2. Expand any macros appearing in the body of the clause. Note that even though the original clause may not have contained any expressions directly amenable to macro expansion, Step 1 may have introduced some goals in the body which contain expressions which may be macro processed.
3. The expanded body typically has unnecessary occurrences of the goal true, and may exhibit unnecessary parenthetical nesting. Flatten the expanded body to remove unnecessary nesting, and remove unnecessary true

goals.

Within the head and body processing, the steps are described below:

28.2.1 Macro expansion in the clause head.

In expanding macros in the clause head, the following steps are performed:

1. A subterm SUB of the clause head is unified with `<pattern>` from the macro specification; if no unification is possible, the next recorded macro specification is considered.
2. If Step 2.1.1 succeeded, then the `<expansion-time condition>` (the `if` part of the macro specification) is executed (the default `<expansion-time condition>` is the goal `true`). If the `<expansion-time condition>` fails, then the next macro definition is considered.
3. If `<pattern>` unified with a subterm SUB of the clause head (Step 2.1.1) and Step 2.1.2 succeeded, then the subterm SUB in the clause head is replaced by `<expansion>` (including bindings produced by the unification). Moreover, the clause body is replaced by the conjunction of the following:
 - (i) the `<goals>` preceded by `where` in the macro specification.
 2. the original body.
 3. the `<goals>` preceded by `when` in the macro specification.

Thus the new body will have the general form

`<where> <original body> <when>`

4. Each clause in the `<list of clauses>` following with in the macro specification is asserted.

28.2.2 Macro expansion in the clause body.

To expand macros in a clause body, the following steps are performed:

1. A subterm SUB of one of the subgoals of the clause body is unified with `<pattern>` from the macro specification; if no unification is possible, the

-
- next recorded macro specification is considered.
2. If Step 2.2.1 succeeded, then the `<expansion-time condition>` (the `if` part of the macro specification) is executed (the default `<expansion-time condition>` is the goal `true`). If the `<expansion-time condition>` fails, then the next macro definition is considered.
 3. The subgoal containing the matching `SUB` is replaced by the conjunction of the following
 4. the `<goals>` preceded by `when` in the macro specification.
 5. The original subgoal with the matching pattern `SUB` replaced by the `<expansion>` expression.
 6. the `<goals>` preceded by `when` in the macro specification.
 4. Each clause in the `<list of clauses>` following with in the macro specification is asserted.

28.2.3 Suppressing macro expansion.

To prevent the MacroPro from expanding specified structures, one makes use of the single and double backquote operators ``` or ````. Thus if we specify

```
X+1 => Z when add(X,1,Z)
```

then

```
`(X+1)
```

is not expanded by the macro-processor. Similarly,

```
``(X+1)
```

also is not expanded by the macro-processor. The difference is that ``` applies only for the top-level of the pattern (i.e., the outermost expression matching the pattern), while ```` suppresses expansion at all levels inside the expression to which it is applied.

28.3 Command Line Invocation of Macro Processing.

Macro processing of a file can be invoked at an operating system shell command line as follows:

```
alspro -g mx_cl -p -s Source -t Target -m Macs
```

A Source must be present. Defaults are supplied as follows. If Source is of the form <Name>.pro, then the default Target is <Name>.ppo; otherwise, the default Target is <Name>.pro. And the default MacroFile is <Name>.mac.

28.4 Predicates for Controlling Macro Processing.

mx/0

macro_expand/0

mx/0 is a synonym for macro_expand/0. The latter prompts for a source and target file, and processes the source file into the target file, using the macros currently present in module macroxp.

macro_expand_files/2

macro_expand_files(Source, Target)

macro_expand_files(+, +)

Processes the Source file into the Target file, using the macros currently present in module macroxp.

macro_expand_files/3

macro_expand_files(SourceFile, TargetFile, MacroFile)

macro_expand_files(+, +, +)

First removes any macros currently loaded into module macroxp, then loads the macros in MacroFile into module macroxp, then processes the SourceFile into the TargetFile, using the macros currently present in module macroxp., and finally removes all macros present in module macroxp.

mx_cl/0

Processes a command line, as described in Section 28.3 (*Command Line Invocation of Macro Processing.*) to extract the files, and then invokes

macro_expand_files/3.