

SPARCworks/Ada Tutorial



A Sun Microsystems, Inc. Business

2550 Garcia Avenue
Mountain View, CA 94043
U.S.A

Part No.: 802-3472-10
Revision A, November 1995

© 1995 Sun Microsystems, Inc. All rights reserved.
2550 Garcia Avenue, Mountain View, California 94043-1100 U.S.A.

This product and related documentation are protected by copyright and distributed under licenses restricting its use, copying, distribution, and decompilation. No part of this product or related documentation may be reproduced in any form by any means without prior written authorization of Sun and its licensors, if any.

Portions of this product may be derived from the UNIX[®] and Berkeley 4.3 BSD systems, licensed from UNIX System Laboratories, Inc., a wholly owned subsidiary of Novell, Inc., and the University of California, respectively. Third-party font software in this product is protected by copyright and licensed from Sun's font suppliers.

RESTRICTED RIGHTS LEGEND: Use, duplication, or disclosure by the United States Government is subject to the restrictions set forth in DFARS 252.227-7013 (c)(1)(ii) and FAR 52.227-19.

The product described in this manual may be protected by one or more U.S. patents, foreign patents, or pending applications.

TRADEMARKS

Sun, the Sun logo, Sun Microsystems, Solaris, are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and certain other countries. UNIX is a registered trademark in the United States and other countries, exclusively licensed through X/Open Company, Ltd. OPEN LOOK is a registered trademark of Novell, Inc. PostScript and Display PostScript are trademarks of Adobe Systems, Inc. All other product names mentioned herein are the trademarks of their respective owners.

All SPARC trademarks, including the SCD Compliant Logo, are trademarks or registered trademarks of SPARC International, Inc. SPARCstation, SPARCserver, SPARCengine, SPARCstorage, SPARCware, SPARCcenter, SPARCclassic, SPARCcluster, SPARCdesign, SPARC811, SPARCprinter, UltraSPARC, microSPARC, SPARCworks, and SPARCcompiler are licensed exclusively to Sun Microsystems, Inc. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc.

The OPEN LOOK[®] and Sun[™] Graphical User Interfaces were developed by Sun Microsystems, Inc. for its users and licensees. Sun acknowledges the pioneering efforts of Xerox in researching and developing the concept of visual or graphical user interfaces for the computer industry. Sun holds a non-exclusive license from Xerox to the Xerox Graphical User Interface, which license also covers Sun's licensees who implement OPEN LOOK GUIs and otherwise comply with Sun's written license agreements.

X Window System is a trademark of the X Consortium.

THIS PUBLICATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT.

THIS PUBLICATION COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION HEREIN; THESE CHANGES WILL BE INCORPORATED IN NEW EDITIONS OF THE PUBLICATION. SUN MICROSYSTEMS, INC. MAY MAKE IMPROVEMENTS AND/OR CHANGES IN THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED IN THIS PUBLICATION AT ANY TIME.



Contents

Preface	v
1. Examining the <code>solve</code> Program	1-7
1.1 Listing Objectives and Tasks	1-7
1.2 Getting Started	1-8
1.3 Creating a New Library	1-9
1.4 Choosing Your Editor	1-12
1.5 Importing Units	1-13
1.6 Viewing Job Status	1-16
1.7 Changing the ADAPATH	1-19
1.8 Setting a LINK Directive	1-25
1.9 Linking <code>solve</code>	1-28
1.10 Running <code>solve</code> From AdaVision	1-29
1.11 Examining <code>solve</code> in AdaVision	1-32
1.12 Explaining the Program's Behavior: Two Hypotheses. . .	1-35
2. Debugging the <code>solve</code> Program	2-37

2.1	Starting AdaDebug From AdaVision.....	2-37
2.2	Setting the First Breakpoint.....	2-39
2.3	Executing the Program.....	2-41
2.4	Stepping Into <code>attack()</code>	2-41
2.5	Examining a Task.....	2-44
2.6	Setting a Breakpoint in the Task View.....	2-47
2.7	Stepping Into <code>munch()</code>	2-50
2.8	Stepping Through <code>munch()</code>	2-52
2.9	Stack Tracing.....	2-55
2.10	Setting a Breakpoint at <code>start_muncher()</code>	2-56
2.11	Evaluating Parameters.....	2-57
2.12	Fixing Bugs From AdaVision.....	2-60

Preface

This tutorial demonstrates SPARCworks/Ada features and facilities as they might be used when working with an actual program. The SPARCompiler Ada demonstration program is called `solve`.

In this tutorial, you use the tools to examine and fix a version of the `solve` program that does not run correctly.

Where `solve` is Located

The `solve` program supplied with SPARCompiler Ada is located in:

```
$ADAHOME/examples/X11_examples/xmaze
```

This directory contains the makefile and source files for two versions of `solve`: one that runs correctly (`maze_muncher_b.aok`), and one that contains a bug (`maze_muncher_b.deb`). You inherit the version with the bug.

Purpose of This Tutorial

Developers might use many strategies and tactics to solve the kinds of problems the `solve` program has. The SPARCworks/Ada tools themselves provide multiple ways of doing things. The approach outlined in this tutorial is just one of many ways of solving the given problems.

This tutorial demonstrates SPARCworks/Ada features and facilities. You are told to do a few things you might not ordinarily do to illustrate some things about SPARCworks/Ada.

Note – Use local execution when following the steps in the tutorial.

How This Tutorial is Organized

This tutorial consists of two chapters:

<i>Examining the solve Program</i>	<i>page 7</i>
<i>Debugging the solve Program</i>	<i>page 37</i>

Appendix A, “List of Steps for AdaDebug Tutorial,” is a step-by-step summary of the second chapter so that you can restart the AdaDebug tutorial and advance quickly to a particular step.

Examining the `solve` Program

1 

You are assigned to take over the broken version of the maze from its original author, who has left the development team. Although you are a veteran Ada programmer, you are fairly new to SPARCworks/Ada. The `solve` program is also something of a mystery.

1.1 Listing Objectives and Tasks

A list of your objectives and the tasks required to accomplish each of them might read like the following:

1. Copying the files in the `xmaze` directory to your own workspace and renaming it `my_broken_maze` to avoid confusion with the original
2. Building the SPARCompiler Ada library `my_broken_maze`
 - a. Starting AdaVision in the `my_broken_maze` directory
 - b. Creating a new SPARCompiler Ada library
 - c. Setting LINK directives to connect the C X11 archive
 - d. Compiling and linking `my_broken_maze`
3. Running `solve` to see what the problem looks like
4. Examining the structure of `solve` in AdaVision
5. Starting AdaDebug and trying to fix the program

1.2 Getting Started

1. Copy the `xmaze` files to a working directory.

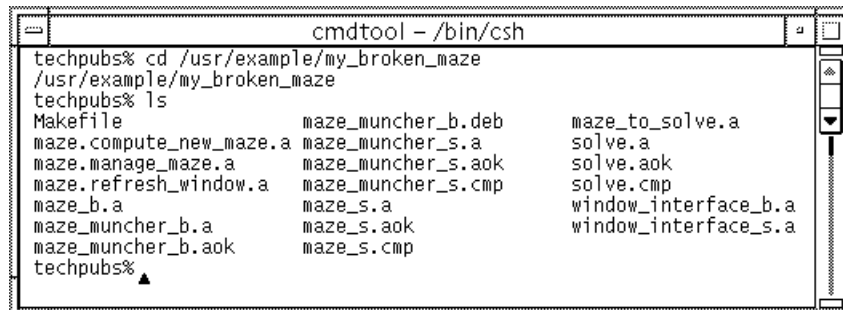
Note – `xmaze` is write-protected. Before you start the tutorial, copy the complete contents of the `$ADAHOME/examples/X11_examples/xmaze` directory to somewhere else in your file system. In this tutorial, the files are copied to `/usr/example/my_broken_maze`.

The maze tutorial directory contains a `.desc` file and some other `.xxx` files, so a simple `cp *` does not copy all of them. Add the option `-r` as follows:

```
% cd $ADAHOME/examples/X11_examples/xmaze
% cp -r . /usr/example/my_broken_maze
```

2. Work with the broken version of the maze and list the contents of the newly created directory as follows:

```
% cd /usr/example/my_broken_maze
% make broken
% ls
```

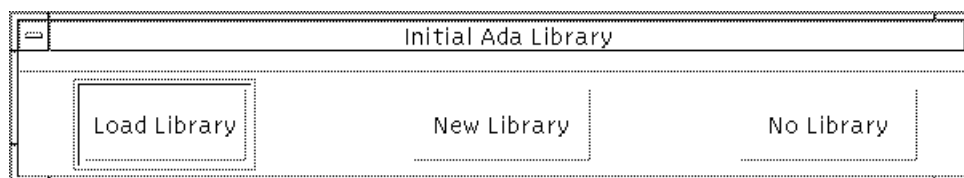


```
cmdtool - /bin/csh
techpubs% cd /usr/example/my_broken_maze
/usr/example/my_broken_maze
techpubs% ls
Makefile                maze_muncher_b.deb      maze_to_solve.a
maze.compute_new_maze.a  maze_muncher_s.a       solve.a
maze.manage_maze.a      maze_muncher_s.aok     solve.aok
maze.refresh_window.a   maze_muncher_s.cmp     solve.cmp
maze_b.a                maze_s.a               window_interface_b.a
maze_muncher_b.a        maze_s.aok             window_interface_s.a
maze_muncher_b.aok      maze_s.cmp
techpubs%
```

3. Start AdaVision from a command line by typing:

```
% adavision &
```

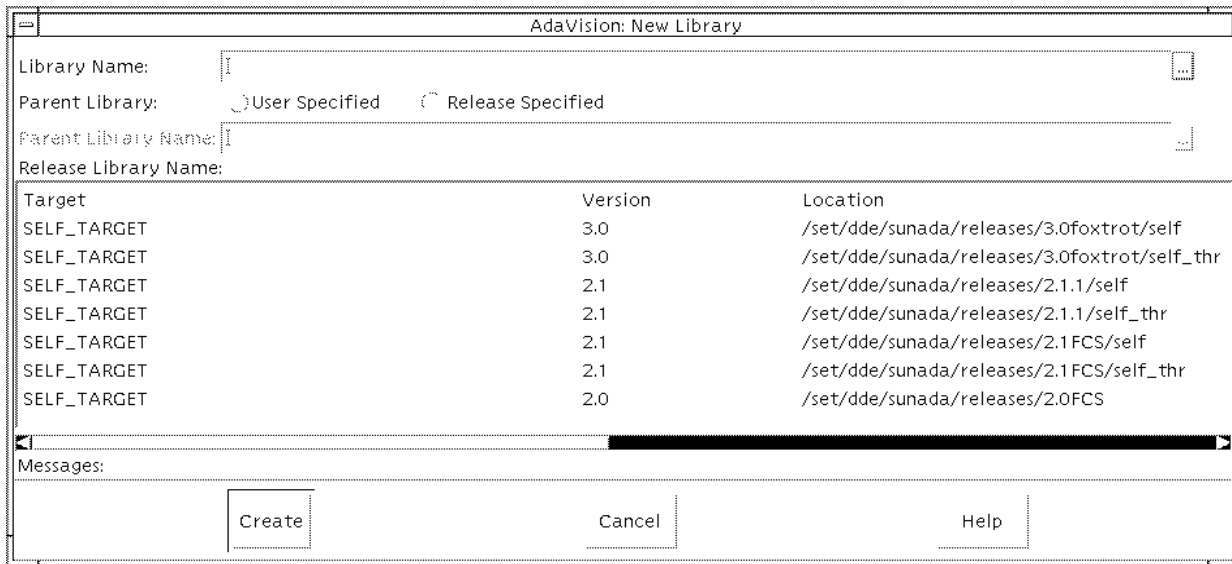

As you have not yet created an Ada library, the Initial Ada Library pop-up window appears:



4. Click on New Library.

1.3 Creating a New Library

In a few seconds, the empty AdaVision main window appears on the screen, along with the New Library pop-up window.



1. Enter the library name

`/usr/example/my_broken_maze/`

by either typing it directly into the text field or clicking on the ellipsis (. .) button to the right of the Library Name text field to open a file chooser from which you select the name.

Note - The target release(s) as shown depend on the entries in your `/etc/VADS` file. Parent marks the release pointed to by the first line in `/etc/VADS` and designates it the default library.

2. Choose Release Library:

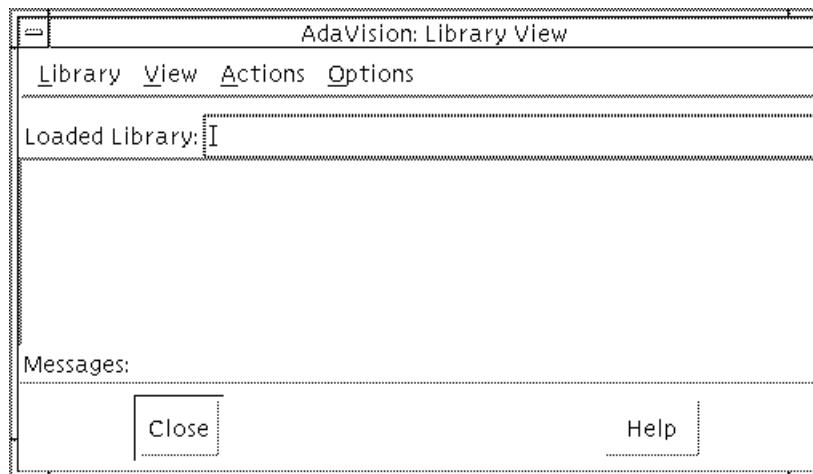
`SELF_TARGET 3.0 opt/SUNWspro/Ada3.0/self`

3. Click on the Create button to create a new SPARCCompiler Ada library named `my_broken_maze` with a parent library named:

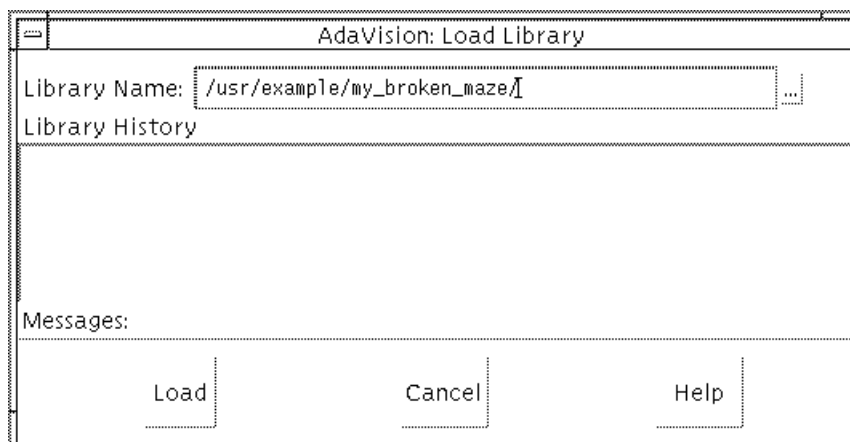
`/opt/SUNWspro/Ada3.0/self/verdixlib`

The New Library window then closes, and the Unit View message log indicates that the action is complete.

4. Choose View ► Libraries from the Unit View to open the Library View.

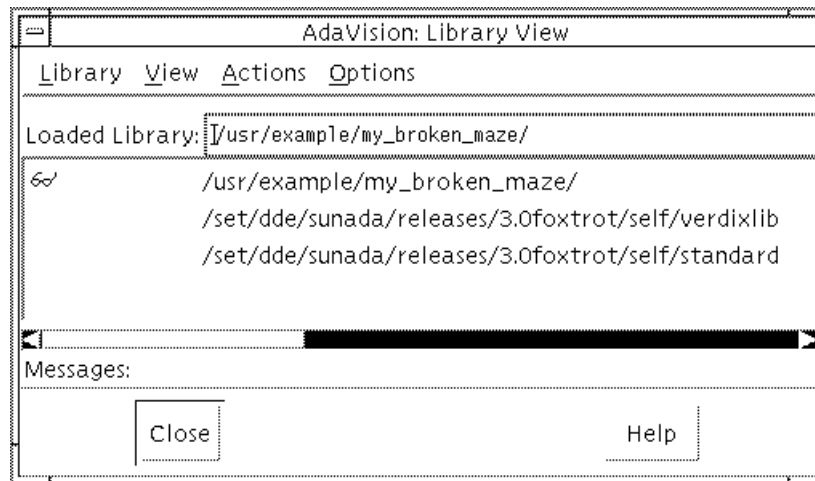


5. Choose Library ► Load from the Library View and type /usr/example/my_broken_maze/ in the library name text field (or select it using the file chooser).



6. Click Load to load the library.

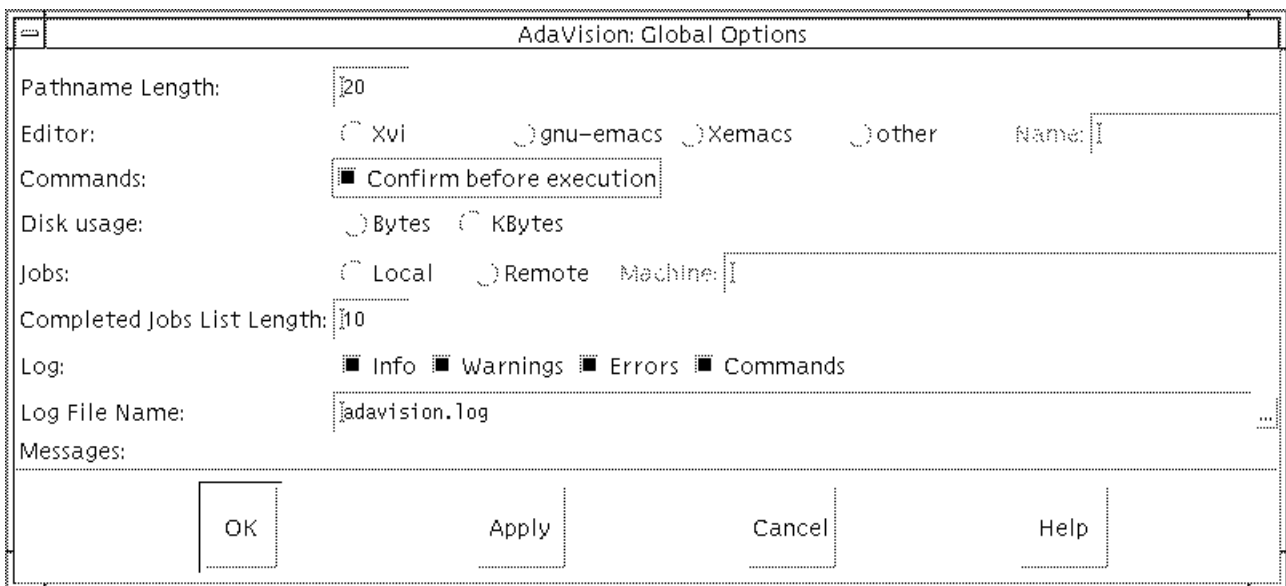
As soon as the library loads, the Load Library window closes and the Library View displays the new library (with the eyeglasses glyph) and all the libraries on its ADAPATH.



1.4 Choosing Your Editor

Open the Global Options window to determine your editor of choice. The default editor is vi.

1. Choose Options ► Global to open the Global Options window.



2. Select your editor of choice.
3. Click OK to quit the window.

1.5 Importing Units

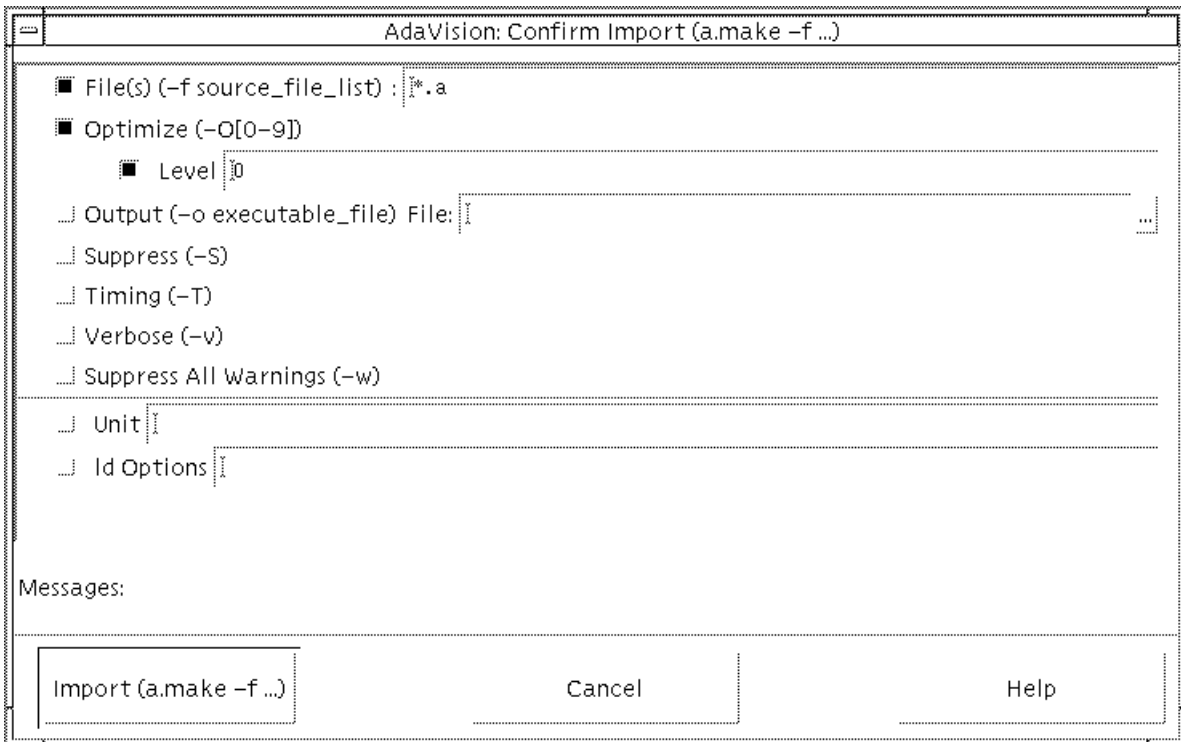
You are now ready to import the `my_broken_maze` units into AdaVision.

AdaVision is primarily an Ada unit-based system rather than a file-based system. When you import units, you instruct AdaVision to compile a list of Ada source files and display an object for each of the resulting units in the display pane. The unit objects are named and have graphically-coded icons.

To import:

1. Select `/usr/example/my_broken_maze` by clicking on it in the scrolling list in the Library View.
2. Choose **Actions** ► **Import**, which brings up a confirmation window.

3. Click on Optimize and enter the number 0 in the Level text field.

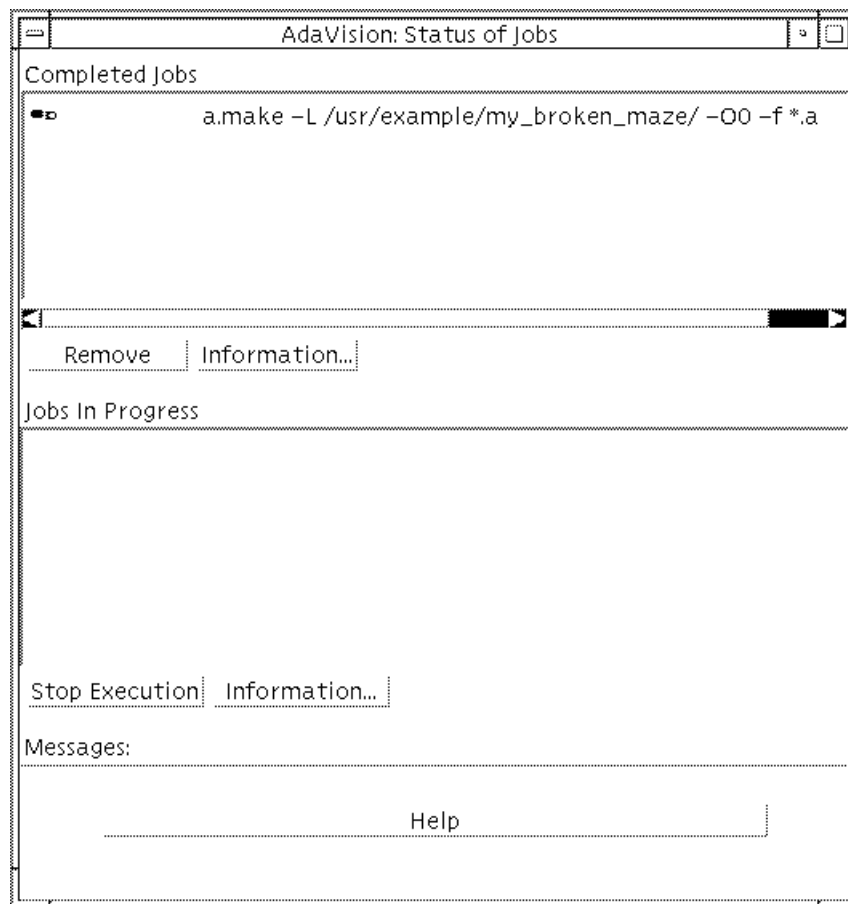


4. Click Import.

AdaVision starts an import job in the background. When the job begins, the confirmation window closes. To check on the status of the import job:

5. Select Status ► Jobs in the Unit View.

Job Status is displayed in the Status of Jobs window.



Unexpectedly, instead of completing a satisfactory compile, the job is displayed in the Completed Jobs pane with a broken glyph, indicating that the job terminated unsuccessfully.

1.6 *Viewing Job Status*

To discover what has happened:

1. **Select the unsuccessful job in the Status of Jobs window by clicking on it.**
2. **Click on the Information button under the Completed Jobs pane.**
3. **Read the message in the Job Output pane of the Job Information window.**

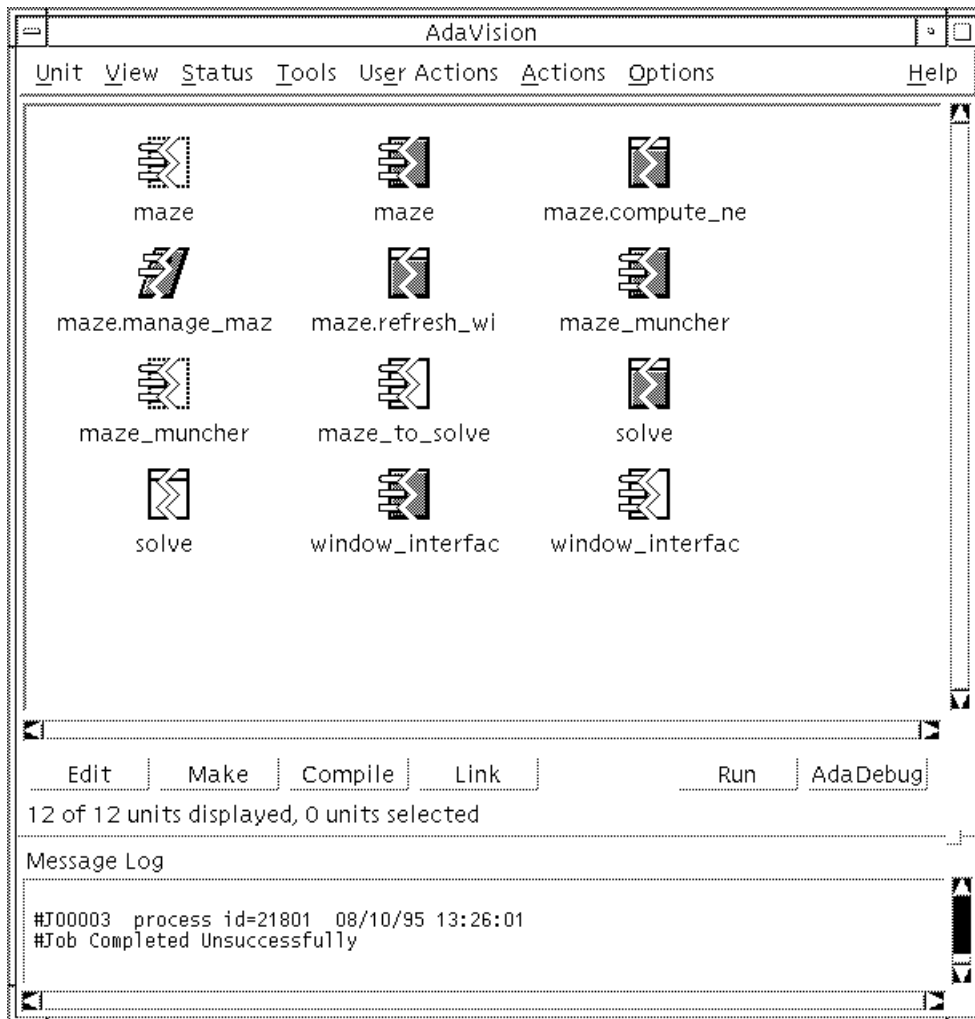

```
AdaVision: Job Information
Status: Failed
Process ID: 25769
Started: 08/08/95 12:21:44
Completed: 08/08/95 12:22:07
Command: a.make -L /usr/example/my_broken_maze/ -OO -f *.a
Job Output
  compilation of window_interface_s.a suppressed:
    unit xlib not found in searched libraries
  compilation of window_interface_b.a suppressed:
    unit v_semaphores not found in searched libraries
  compilation of maze_s.a suppressed:
    unit xlib not found in searched libraries
  compilation of maze_muncher_s.a suppressed:
    spec of maze, in file maze_s.a, needs to be recompiled
  compilation of maze_b.a suppressed:
    unit xtypes not found in searched libraries
  compilation of maze.refresh_window.a suppressed:
    body of maze, in file maze_b.a, needs to be recompiled
  compilation of maze.manage_maze.a suppressed:
    body of maze, in file maze_b.a, needs to be recompiled
  compilation of maze.compute_new_maze.a suppressed:
    body of maze, in file maze_b.a, needs to be recompiled
  compilation of maze_to_solve.a suppressed:
    spec of maze, in file maze_s.a, needs to be recompiled
  compilation of maze_muncher_b.a suppressed:
    spec of maze_muncher, in file maze_muncher_s.a, needs
    to be recompiled
  compilation of solve.a suppressed:
    spec of maze_muncher, in file maze_muncher_s.a, needs to be recompiled
Save Job Output File:
  Redo Command Print Job Output...
Messages:
  Close Help
```

The output indicates that maze references units in a library that is not on the my_broken_maze ADAPATH. publiclib was not assigned as the parent library when creating the new library. This particular program also needs

vads_exec and x11 on its ADAPATH.

Rather than starting over, you can go to the my_broken_maze Library Options window and change the ADAPATH. Leave the Job Information and Status of Jobs windows open for later.

Although the job completed unsuccessfully, the icons associated with it appeared in the Unit View as soon as the job completed.

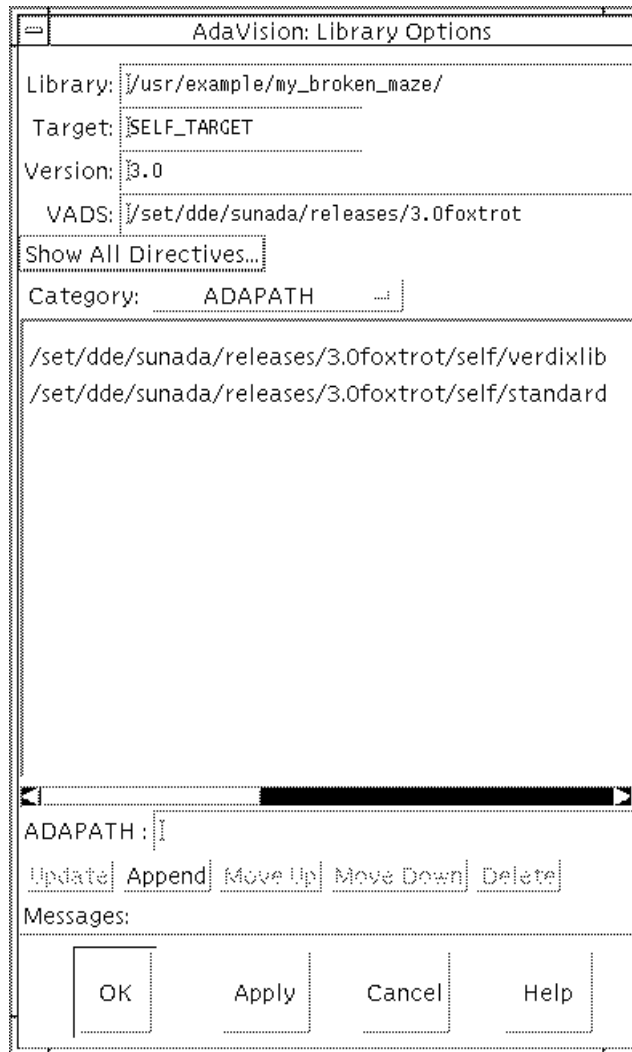


1.7 Changing the ADAPATH

- 1. Choose Options ► Library in the Library View to display the Library Options pop-up window.**

In addition to changing the ADAPATH, you might as well write the link directives for the C X11 archive while the Library Options window is open.

The ADAPATH is visible in the window's main display pane.



The ADAPATH control shows only the standard and verdirlib libraries. The vads_exec, X11, and publiclib libraries need to be added to the ADAPATH list.

The libraries should be in the following order:

- a. . . ./publiclib
- b. . . ./verdixlib
- c. . . ./X11
- d. . . ./vads_exec
- e. . . ./standard

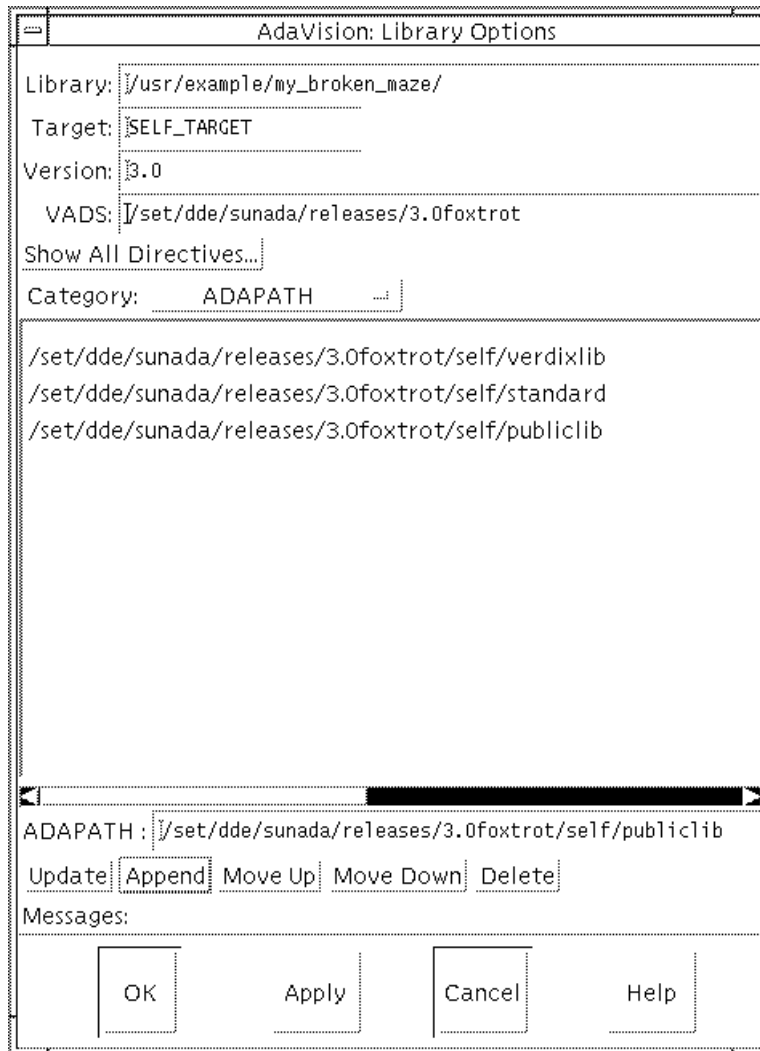
Begin by adding `publiclib`.

2. Click in the `ADAPATH` text field at the bottom of the window and type in the `ADAPATH` for `publiclib`, where `sunada_location` is the location of SPARCompiler Ada (and the value of the `$ADAHOME` environment variable):

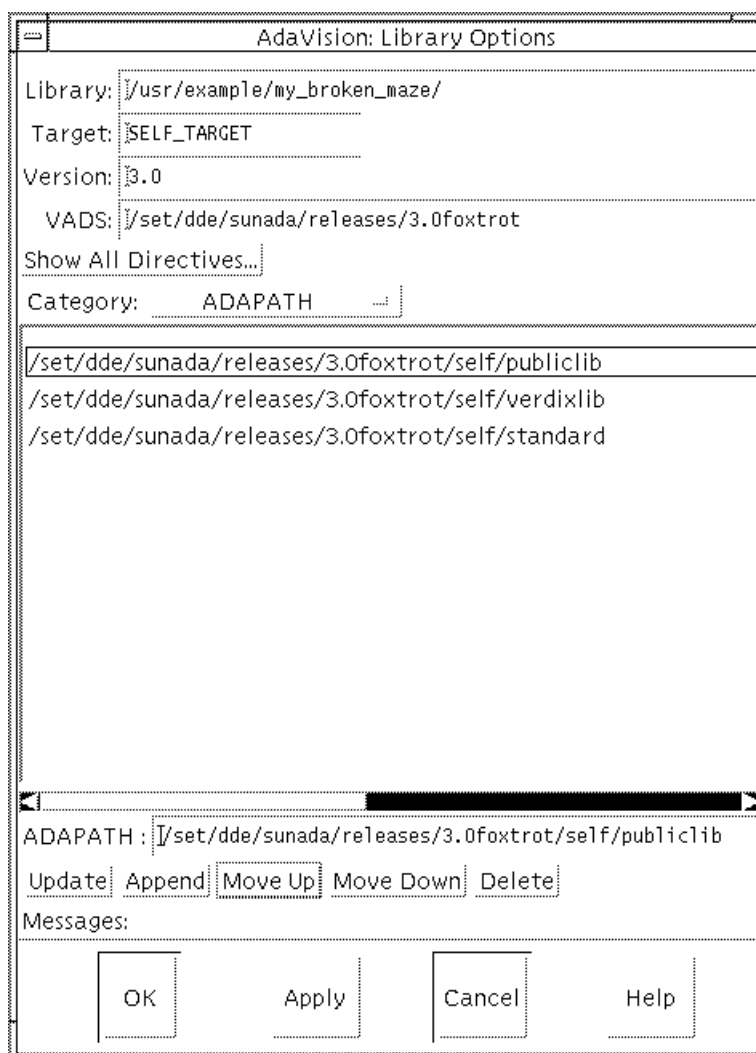
`sunada_location/self/publiclib`

3. Click Append to see the change in the scrolling list.

Note – Although the scrolling list updates, the actual ADAPATH does not change until you press Apply or OK, which you will do in a minute.

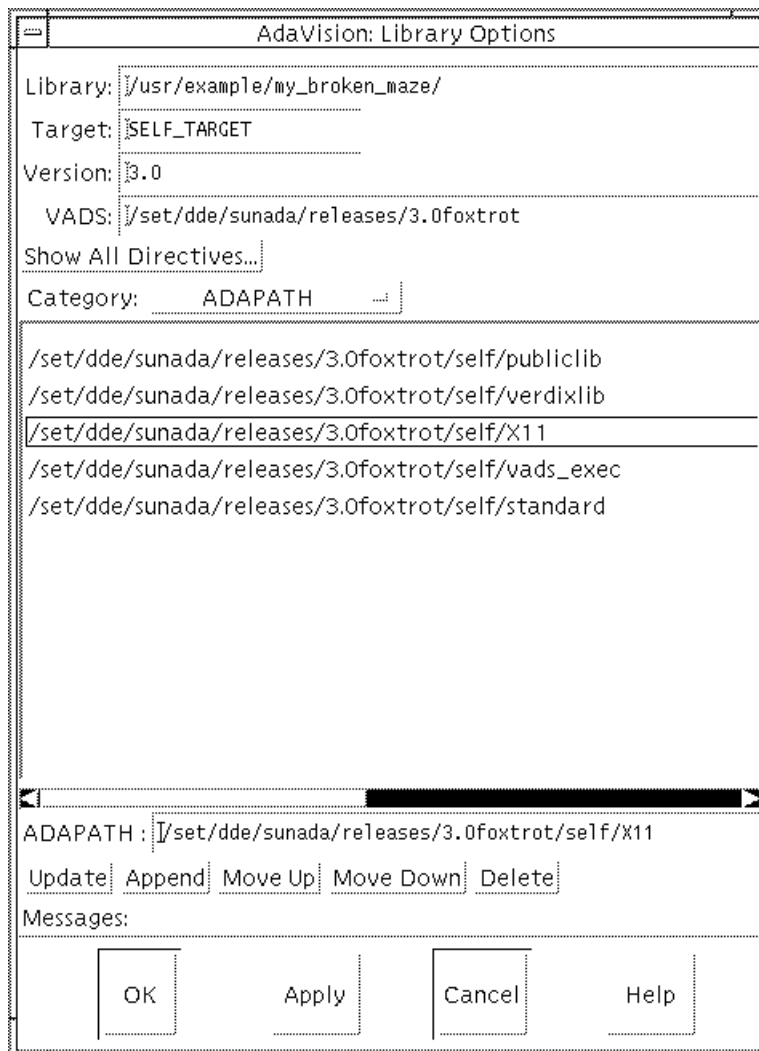


4. Select `publiclib` from the scrolling list and press the Move Up button until the library is at the desired location in the list.



5. Press Apply to update the Library View and have the change in the ADAPATH take effect.

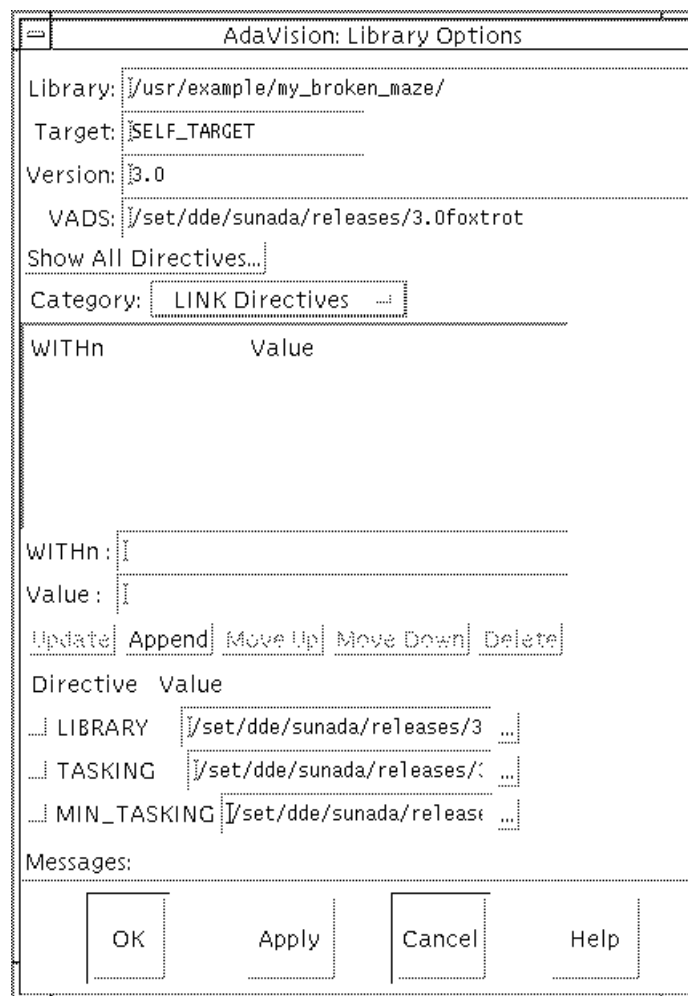
6. Repeat Step 2 through Step 5 for the vads_exec and X11 libraries.



1.8 Setting a LINK Directive

You are now ready to write the LINK directive for the C X11 archives. Still working in the Library Options window,

1. Select the LINK Directives setting from the Category menu to display the LINK directive controls.



No WITHn directives are set in my_broken_maze.

≡ 1

You will want to set a LINK directive in `my_broken_maze` to tell AdaVision to include the shared object (or archive) `libX11` in any executable linked in this Ada library.

To set `WITHn` LINK directives in the current library:

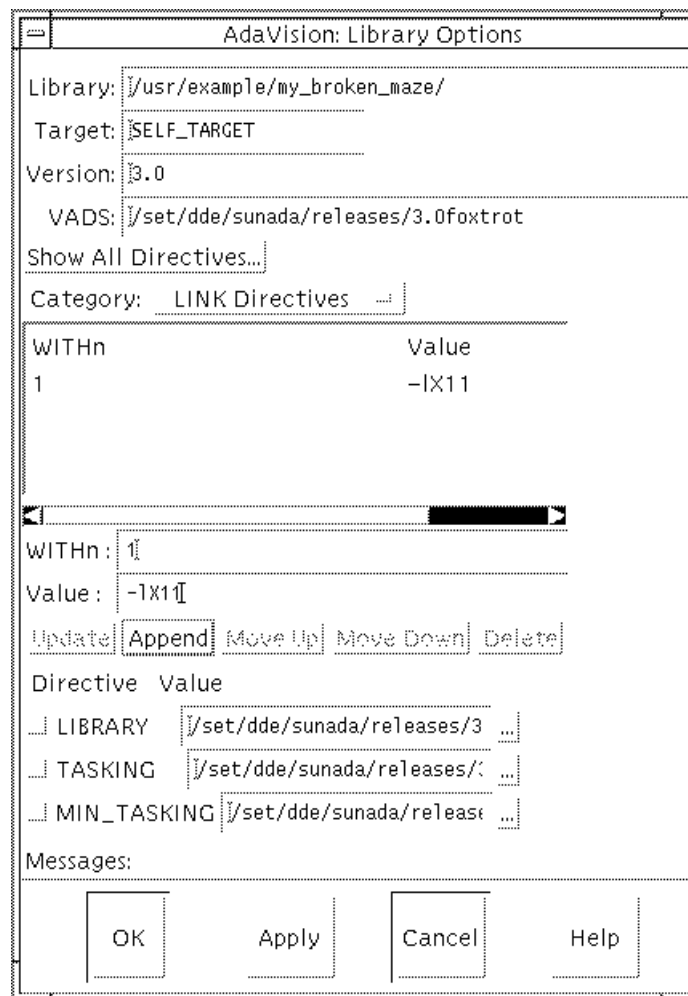
2. Click on the `WITHn` text field and type 1 to indicate the number of the WITH directive.

3. In the Value field, enter the name of the C X11 archive:

`-lX11`

Note – In the C X11 archive above, the hyphen is followed by a lowercase “l,” a capital “X,” and two ones.

4. Click Append to have the changes take effect in the scrolling list.



5. Click OK to update the Ada library. The Library Options window closes.

You are now ready to rerun the Import job.

6. Press the Redo Command button in the Job Information window, which was left open on the desktop.

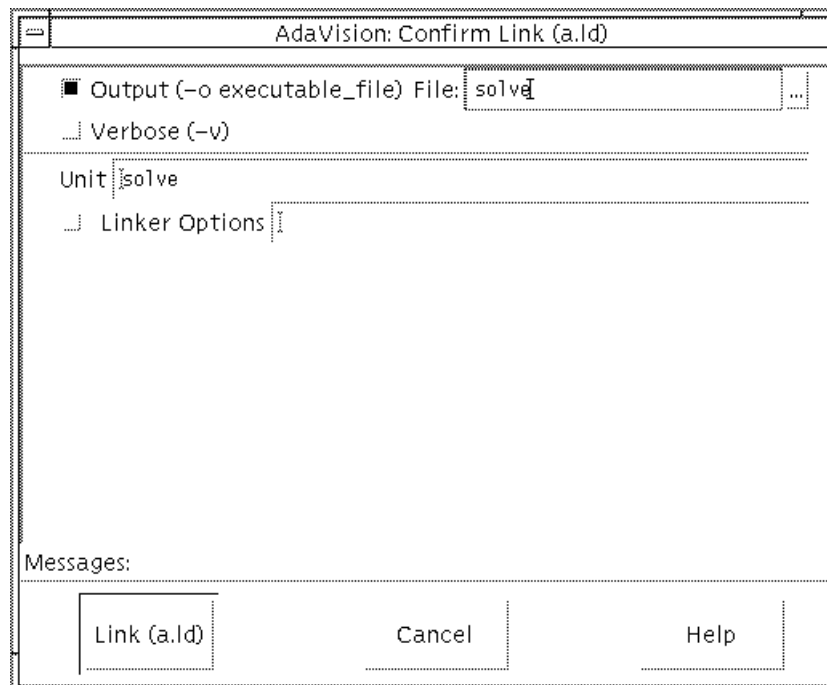
This time, the import is successful.

7. Close the Job Information window, the Status of Jobs window, and the Library View window.

All that remains before running `solve` is to link the executable.

1.9 Linking `solve`

1. Select the body of the unit `solve` by clicking on its icon in the Unit View.
2. Choose Actions ► Link (or click on the Link button).



3. Select Output File and type `solve` in the text field to make the file name of the executable image more descriptive than the default `a.out`.
4. Confirm the link action by pressing Link in the confirmation window.

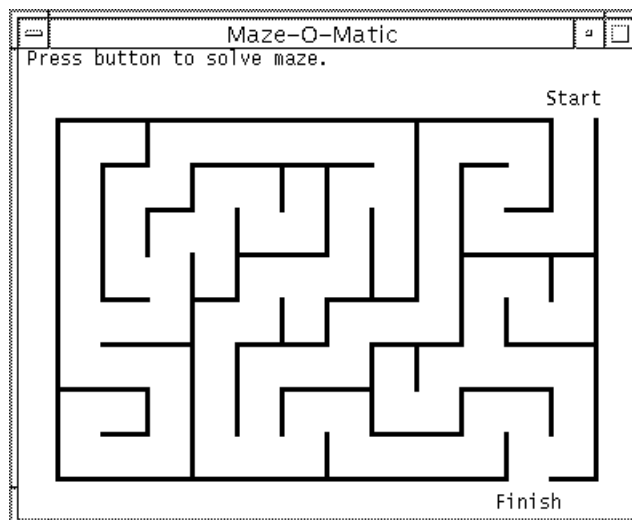
The link succeeds and the `solve` unit icon now has a Sun logo inside to indicate it is executable.

1.10 Running `solve` From *AdaVision*

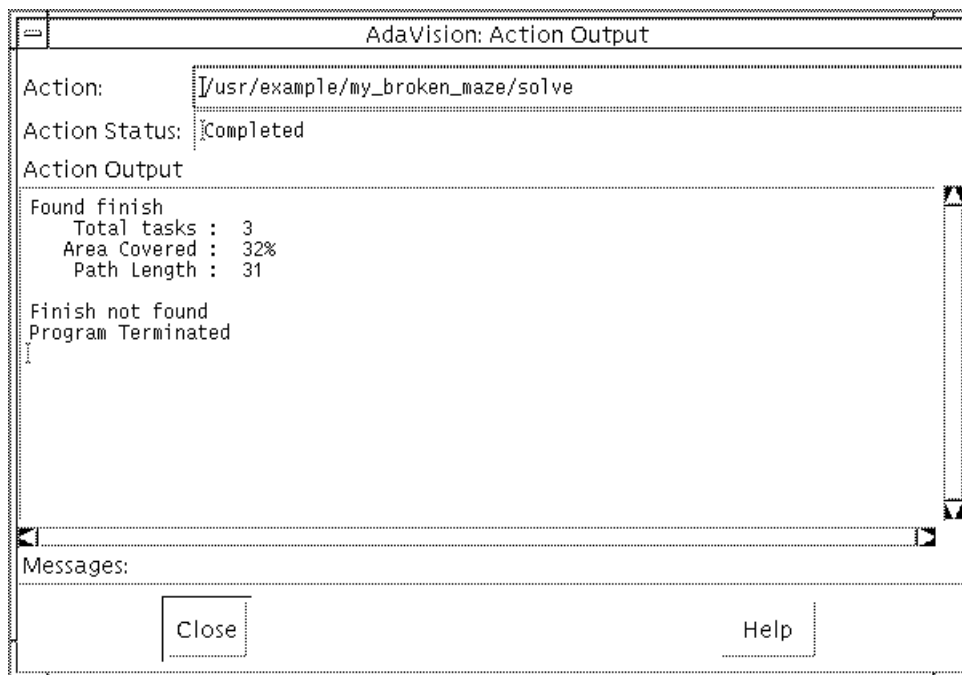
To start running `solve` from within *AdaVision*:

1. With the `solve` executable still selected in the Unit View, choose Actions ► Run (or click on the Run button) and click Run in the Run action confirmation window.

The `solve` program, called Maze-O-Matic for end users, is displayed on the screen.

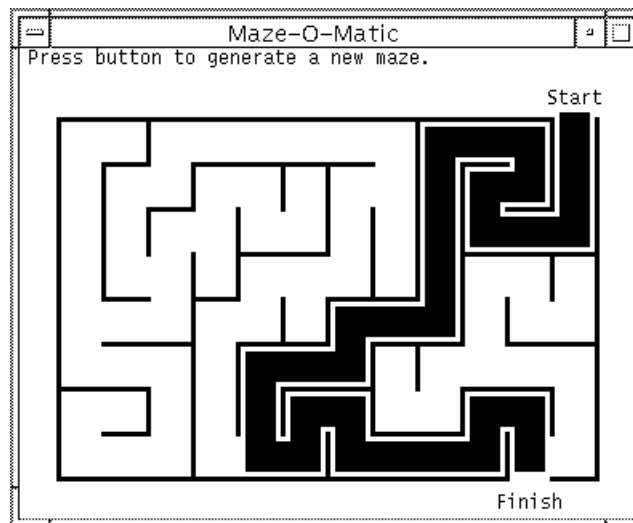


The Run action generates an Action Output window that contains details on the state of the action.



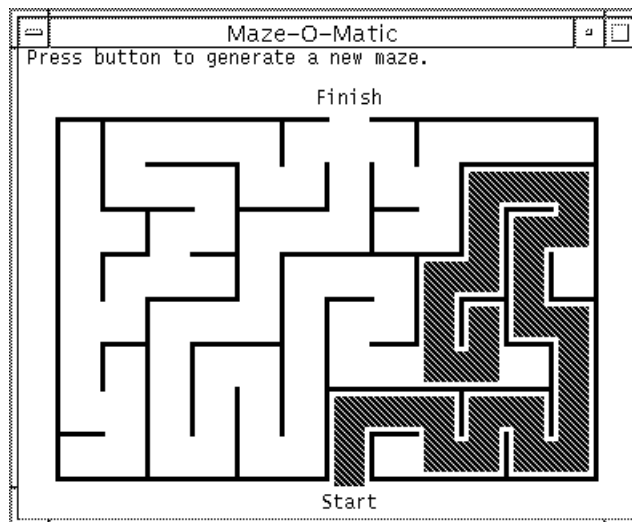
2. Put the cursor inside the maze display pane and click the mouse, as the message in the maze instructs users to do.

The first maze completes successfully.



3. Generate a new maze by clicking on the maze window once, then try to solve the new maze by clicking on it again.

Maze-O-Matic fails to work this time, terminating in a blind maze pathway.



4. Quit Maze-O-Matic and close the Action Output window.

1.11 Examining `solve` in *AdaVision*

The main thing you know about `solve` is that it is a multitasking program that spawns tasks at each intersection in the maze, and that these tasks then follow all the pathways simultaneously to solve the maze.

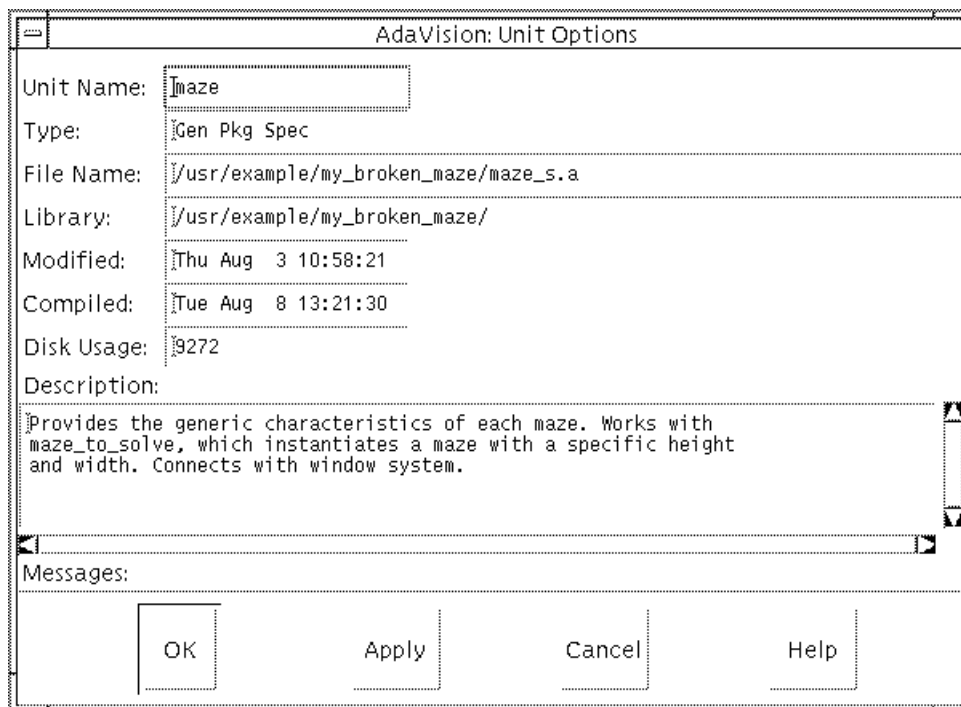
While watching the program fail, you noticed that it:

- Dead-ends
- Fails to make footprints on other pathways in each maze
- Seems to have trouble spawning tasks and sending them in each direction at an intersection

Before proceeding, you need to know more about how the program is assembled. What little documentation is available is either in comments in the source files or in the brief descriptions of each unit that a coworker wrote up early in the project.

The first thing to do is look quickly at the description for each unit provided by the previous programmer. To do this:

- 1. Select each unit in turn and choose Options ► Unit to display the Options window for each unit.**

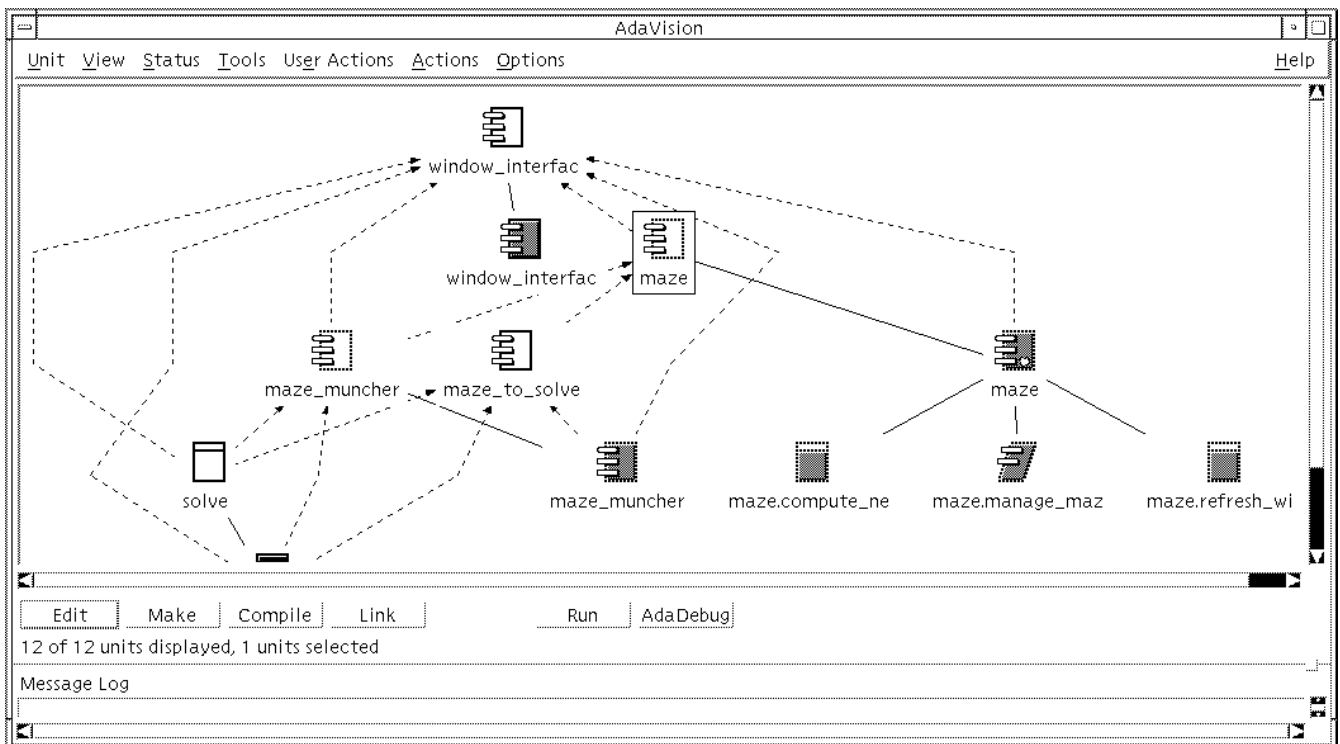


Now that you have a fair idea of how the program is assembled, it becomes important to see how the Ada units are themselves structured into specs, bodies, and subunits.

- 2. Click OK to close the Unit Options window.**

3. From the Unit View, choose View ► Graph.

AdaVision opens a base window that graphs the interdependencies among the displayed units (in this case, all of the units in the `solve` program). WITH dependencies appear as dashed lines, and parent-child relationships appear as solid lines.



4. Choose View ► Icon to return to the icon view.

1.12 *Explaining the Program's Behavior: Two Hypotheses*

If you run through the program again, you can see that the program is not spawning tasks in all possible directions at each intersection, as it is supposed to do. All that white space should be gray because task munchers should have gone down each path, flushing the entire maze. Also, in the second maze, a task did not head out to the left at the first intersection.

There are two plausible explanations for the program's behavior:

- Maybe only one task is running and the others are not being spawned successfully.
- Maybe tasks are spawned successfully, but they are dying before they get anywhere.

You can use the Debugger tool to further examine the program.

Note – If you want to redo the first chapter, be sure to type `make clean` in a command line before starting over.

Debugging the solve Program

2 

You are now ready to view and edit the source code. If you want to follow the AdaDebug section of the tutorial in more than one sitting (or if you miss a step that breaks the correct sequence), you can use the concise list of steps in Appendix A to return the program quickly to any particular place in the tutorial.

Note - If you skipped Chapter 1 of this tutorial, go back and perform Steps 1 and 2 from Section 1.2, “Getting Started.” Then, execute the `make all` command at the command line before continuing with this chapter.

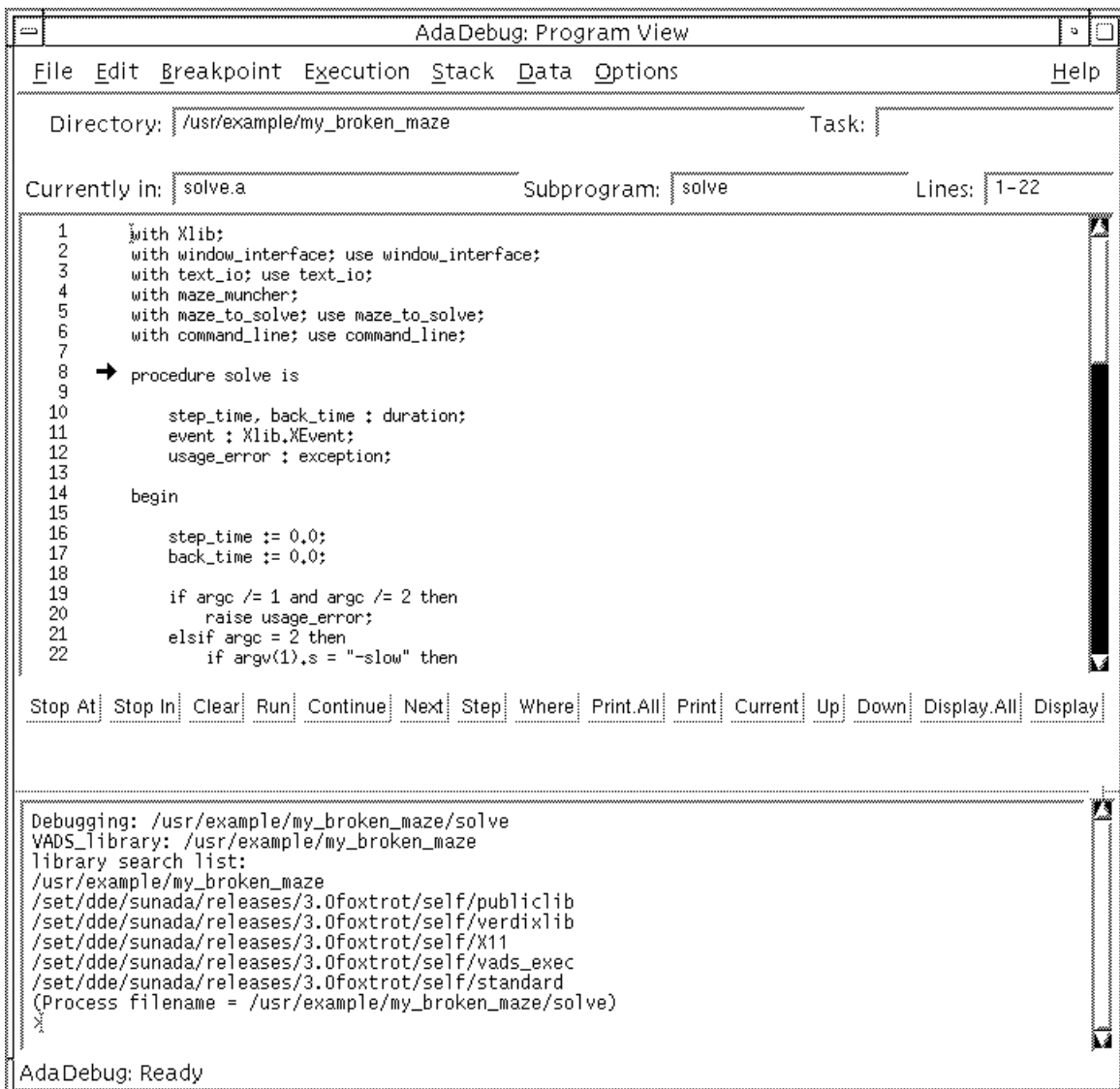
2.1 Starting AdaDebug From AdaVision

To start AdaDebug from within AdaVision:

- ◆ **Select the `solve` executable in the AdaVision Unit View and choose Tools ► Debug (or click on the AdaDebug button).**

Note - If AdaDebug is already running on the desktop, you can start a debugging session by choosing File ► Load Executable. If neither AdaVision nor the Debugger is running on your Desktop, you can type `adadebug solve &` in a Shell Tool window to start the Debugger.

You may want to close AdaVision to an icon for now.



At startup, AdaDebug displays the Program View window with the main program source code in the display pane. A small arrow marks the next line to be executed by the debugger—that is, the current position in the program.

2.2 *Setting the First Breakpoint*

You know that the program solves the first maze and therefore decide to find a place in the code where it passes the first maze but does not start the second one.

1. **Begin by scrolling halfway through the file to find parts of the code that might be relevant.**

Line 57 reads:

```
57      generate_new_maze
```

At first, this seems like a good place to put a breakpoint. But this statement probably comes too late to catch any of the action.

In the code that comes before this call to generate a new maze, line 41 is a call to the `attack()` procedure. From the description of `solve`, it is fairly obvious that the program uses `attack()` to launch the task munchers that solve the maze. A breakpoint at the call to the `attack()` procedure seems like a good idea.

2. **Double-click in line 41. The first word in the line is highlighted.**

3. Choose Breakpoint ► Stop At <selected line> (or click Stop At).

A stop sign glyph appears to the left of line 41, indicating that a breakpoint is set at that line.

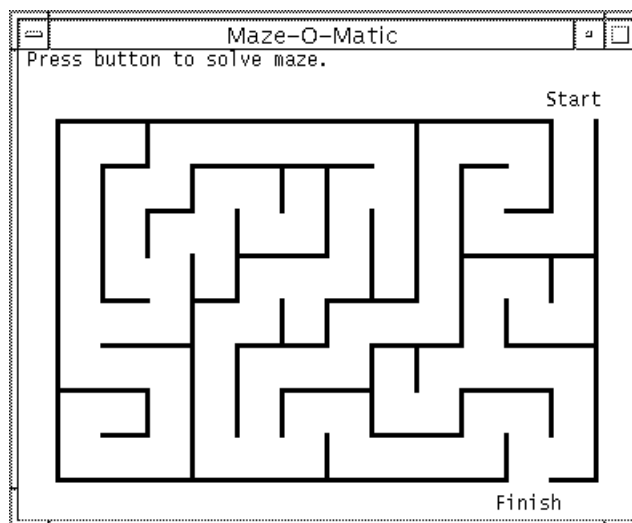


2.3 Executing the Program

Having set a breakpoint, start the program from the beginning.

1. Choose Execution ► Run (or click Run).

The Maze-O-Matic window opens with a message prompting the user to click the mouse button to solve the maze.



2. Click in the maze window to start the program solving the first maze.

The program runs until it hits the breakpoint at the call to `attack()`. The arrow indicating the next line to be executed points at line 41, where the breakpoint is set. Nothing has happened yet in the Maze-O-Matic window.

2.4 Stepping Into `attack()`

Recall that the program always solves the first maze, even though it does not work exactly as it is supposed to work because it doesn't send out munchers to explore the opposite path when it reaches an intersection. Work with the first maze for a while.

Step into the `attack()` procedure to see what happens when the program calls it.

- 1. Choose Execution ► Step (or click Step) to step into the `attack()` procedure.**

The Program View window now displays the `attack()` procedure. AdaDebug positions the current line arrow at line 191, the first line of `attack()`.

```

180         begin
181             accept start ( x : in maze_x; y : in maze_y; dir : in footprint ) do
182                 start_x := x; start_y := y; start_dir := dir;
183             end;
184             end_found := munch( start_x, start_y, start_dir );
185             accept finish ( found_end : out boolean ) do
186                 found_end := end_found;
187             end;
188         end;
189     end;
190
191     → procedure attack( step_time, back_time : duration ) is
192
193         first_muncher : access_maze_muncher;
194         start_x : maze_x;
195         start_y : maze_y;
196         start_dir : footprint;
197
198         begin
199             muncher_step_time := step_time;
200             muncher_back_time := back_time;
201             done_yet.restart;

```

Stop At Stop In Clear Run Continue Next Step Where Print.All Print Current Up Down Display.All Display

```

>b 41
b 411
>r
r1
solve
Resume...
41      attack( step_time, back_time );
>s
s1
191     procedure attack( step_time, back_time : duration ) is

```

AdaDebug: Ready

At this point, assume that the program will behave properly and begin stepping through `attack()` line by line.

2. Choose Execution ► Next (or click Next).

- 3. Repeat Next eight more times, advancing to line 206 and reading ahead a line or two before each step to analyze what is happening.**

With the current line arrow pointing at line 206, nothing has happened in the Maze-O-Matic window.

Line 206 reads:

```
206    first_muncher:=start_muncher(start_x,start_y,start_dir);
```

And the next line, 207, reads:

```
207    first_muncher.finish(found_finish);
```

Remember, the munchers' progress results in footprints on the maze. Don't allow the program to step over line 206, or the action involved in drawing the first footprint will be lost. Step into `start_muncher`.

- 4. Choose Execution ► Step (or click Step) to step into the `start_muncher` function at line 21.**

AdaDebug scrolls the display to point at line 21, where `solve` declares the `start_muncher()` function. This is a short but important routine.

- 5. Choose Execution ► Next (or click Next) four times until reaching line 26.**

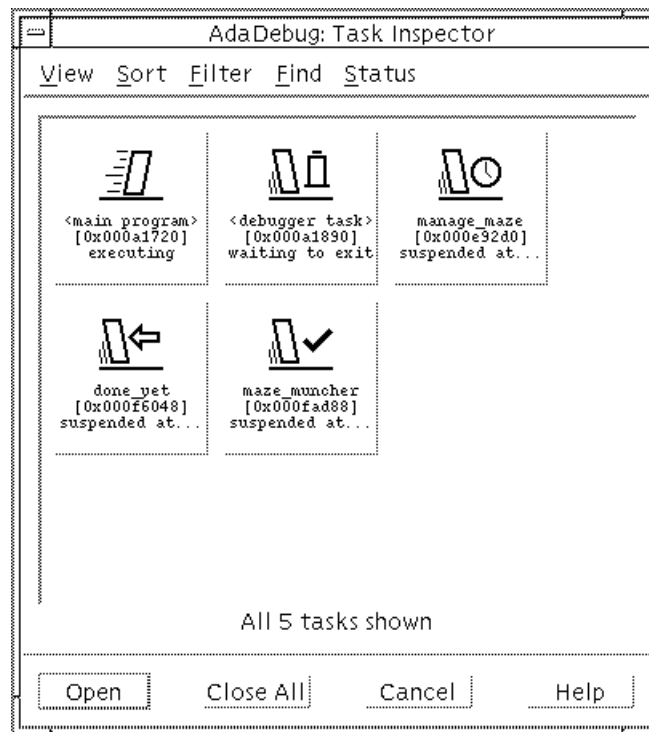
Line 25, the second line in `start_muncher()`, allocates a task of type `maze_muncher`. In stepping over line 25, notice that `solve` has just created a new task. Note also that the next line, 26, is a task rendezvous call:

```
26    new_muncher.start(x, y, dir);
```

Recall that the `maze_muncher` unit uses `attack()` to control the munchers that travel the maze. Now it should be clear that each muncher is a task.

2.5 Examining a Task

- 1. Choose Execution ► Task Inspector to display each activated task in the Task Inspector window.**



As expected, the icon for the new muncher task that `start_muncher` just created is visible. It is named `maze_muncher`.

From the Task Inspector, the task opens into a Task View window, where a task-specific breakpoint can be set at the accept statement for this muncher.

2. Select the `maze_muncher` task icon in the Task Inspector and click Open to open the task in a Task View window.

The Task View window looks similar to the Program View window.

Note – The important difference between the Task View window and the Program View window is functional. When you choose a menu item from within a Task View window, it applies only to the task identified in the Task View title bar.

AdaDebug: Task View

File Edit Breakpoint Execution Stack Data Options Help

Directory: /usr/example/my_broken_maze Task: maze_muncher [0xfad88]

Stopped in: Subprogram: _A_+ts_accept.38S11.ts Line:

Currently in: maze_muncher_b.a Subprogram: maze_muncher Lines: 170-192

```

170         return false;
171     end if;
172     end;
173
174     begin
175     declare
176         start_x : maze_x;
177         start_y : maze_y;
178         start_dir : footprint;
179         end_found : boolean;
180     begin
181     => accept start ( x : in maze_x; y : in maze_y; dir : in footprint ) do
182         start_x := x; start_y := y; start_dir := dir;
183     end;
184         end_found := munch( start_x, start_y, start_dir );
185     accept finish ( found_end : out boolean ) do
186         found_end := end_found;
187     end;
188     end;
189     end;
190
191     procedure attack( step_time, back_time : duration ) is
192

```

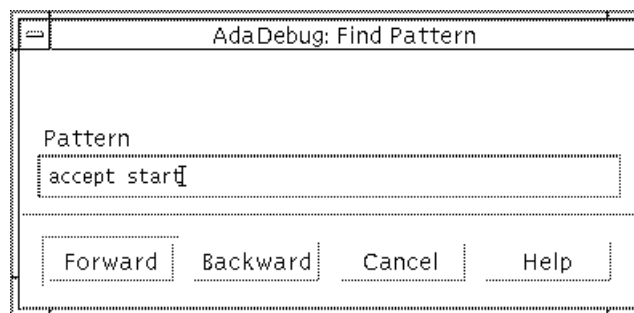
Stop At Stop In Clear Run Continue Next Step Where Print.All Print Current Up Down Display.All Display

2.6 Setting a Breakpoint in the Task View

By setting a breakpoint in the Task View window, the program reaches the breakpoint only if this particular task attempts to execute this line of code.

Find the `accept start` statement using the search facility:

1. Choose **Edit ► Find in the Task View**.



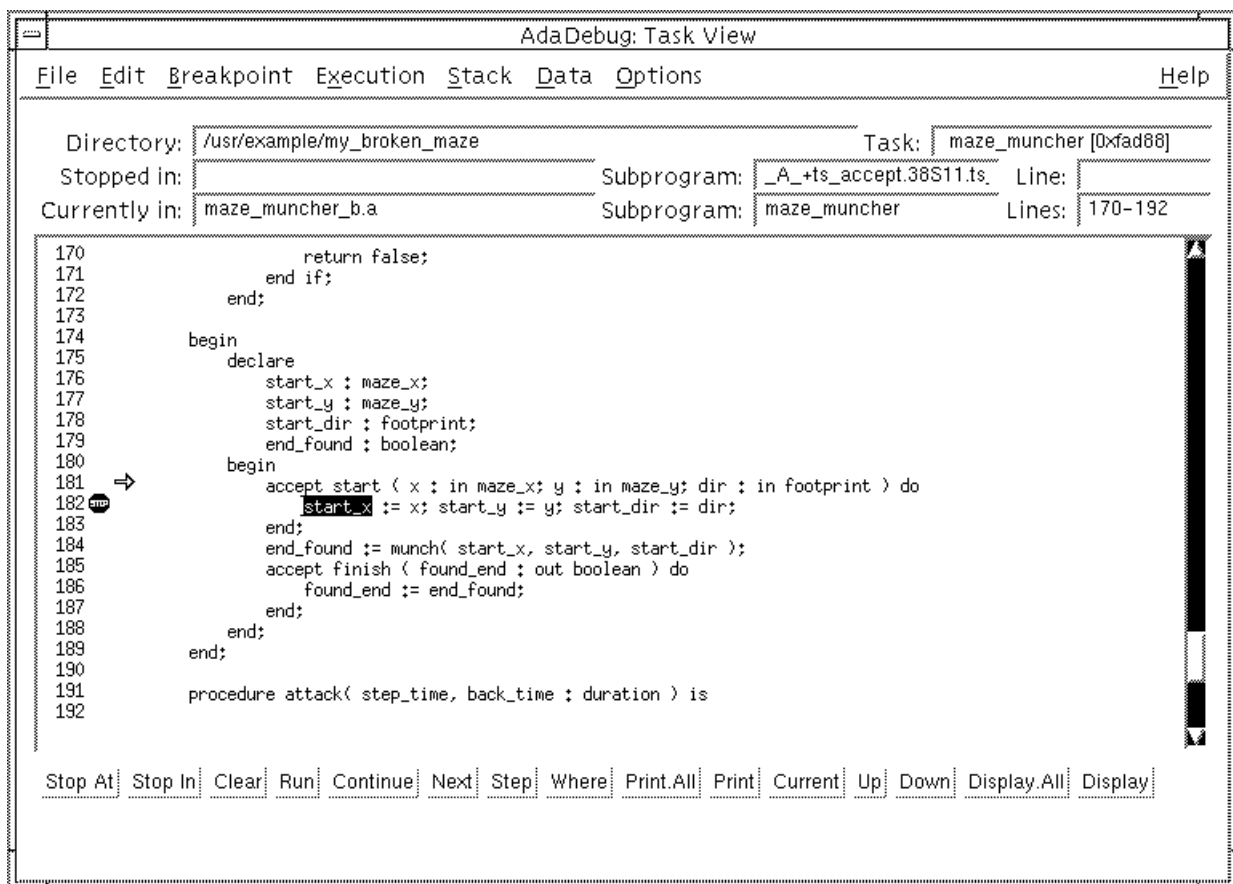
2. Type `accept start` in the text field and click **Forward**.

The code in the Task View pane highlights the `accept start` statement into view at line 181.

A breakpoint at line 181 is too early. The task is already suspended at the `accept start` rendezvous statement. The task-specific breakpoint goes on line 182.

3. Close the **Find Pattern** window.

4. Double-click on line 182 in the *Task View* window and choose Breakpoint ► Stop At <selected line> (or click Stop At).



The hollow arrow at line 181 shows where the task suspended its execution. Don't confuse the hollow arrow with the solid arrow, which shows the next statement to be executed when you continue the program.

5. Close the Task View window, but leave the Task Inspector open for later.

6. Choose Execution ► Continue (or click Continue) from the Program View.

The screenshot shows the AdaDebug: Program View window. The title bar reads "AdaDebug: Program View". The menu bar includes "File", "Edit", "Breakpoint", "Execution", "Stack", "Data", "Options", and "Help". The status bar shows "Directory: /usr/example/my_broken_maze" and "Task: maze_muncher [0xfad88]". Below this, it indicates "Stopped in: maze_muncher_b.a" and "Subprogram: maze_muncher" with "Line: 182". It also shows "Currently in: maze_muncher_b.a" and "Subprogram: maze_muncher" with "Lines: 171-192".

The main area displays the source code for the `maze_muncher` task. The code is as follows:

```

171         end if;
172     end;
173
174     begin
175         declare
176             start_x : maze_x;
177             start_y : maze_y;
178             start_dir : footprint;
179             end_found : boolean;
180         begin
181             accept start ( x : in maze_x; y : in maze_y; dir : in footprint ) do
182                 start_x := x; start_y := y; start_dir := dir;
183             end;
184             end_found := munch( start_x, start_y, start_dir );
185             accept finish ( found_end : out boolean ) do
186                 found_end := end_found;
187             end;
188         end;
189     end;
190
191     procedure attack( step_time, back_time : duration ) is
192

```

Below the code is a control panel with buttons: "Stop At", "Stop In", "Clear", "Run", "Continue", "Next", "Step", "Where", "Print.All", "Print", "Current", "Up", "Down", "Display.All", and "Display".

The bottom section is a command window with the following text:

```

>a
a1
26         new_muncher.start( x, y, dir );
>lt all
>b 182 in 0fad88
Warning: breakpoint set in generic instantiation
>g
g1
Resume...
182         start_x := x; start_y := y; start_dir := dir;
x:

```

The status bar at the bottom of the window reads "AdaDebug: Ready".

The program runs until the task named `maze_muncher` hits this task-specific breakpoint. Still, nothing has happened in the Maze-O-Matic window.

Clear the breakpoint at line 182, since it has already served its purpose.

7. **Double-click on line on 182 in the Program View and choose Breakpoint ► Clear At <selected line> (or click Clear).**

2.7 *Stepping Into* `munch ()`

By stepping into the `munch ()` procedure at line 184, you can step through the entire process by which `solve` creates task munchers.

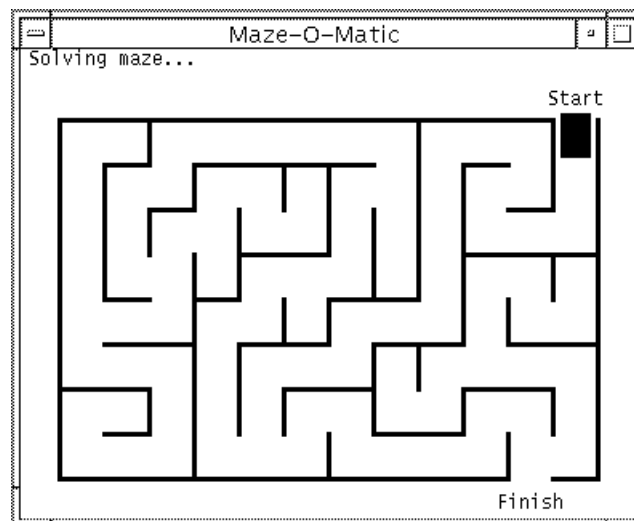
1. **Choose Execution ► Next (or click Next) twice to advance the current line pointer to line 184.**
2. **Choose Execution ► Step (or click on Step) at line 184 to step into the start of the `munch ()` function at line 70.**
3. **Scroll or click Next a few times to advance through the code, noticing the loop from lines 106 to 116.**

Just after the loop, at line 123, is an `if . . . then . . . else` statement that seems to be significant.

4. **Double-click on line 123 and choose Breakpoint ► Stop At <selected line> (or click Stop At).**

5. Choose **Execution ► Continue** (or click **Continue**) to have the program advance to the breakpoint you just set at line 123.

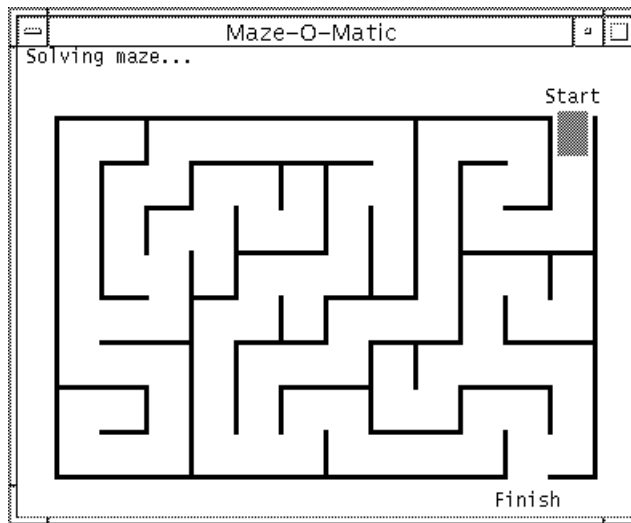
The program advances to the new breakpoint. In the Maze-O-Matic window, the program draws a single black footprint in the maze.



2.8 Stepping Through `munch()`

Step through `munch()` one line at a time to see what happens next.

1. Click Next five times. The fifth step executes the `make_print(x, y, dir Grey)` procedure (line 128), causing the first footprint to turn gray.



2. Click Next again to execute line 129.

Something unexpected happens: the program returns to line 123, even though the program is not in a loop. Also, the hollow arrow at line 129 indicated a different stack frame was executing.

```

116         end loop;
117
118         -- if only one way to go, then go there else if at
119         -- intersection spawn children to investigate for
120         -- you and wait for those children to finish else
121         -- at dead end
122
123     if num_poss = 0 then
124         end_found := false;
125     elsif num_poss = 1 then
126         xx := x; yy := y;
127         move_dir( xx, yy, possibilities(1) );
128         make_print( x, y, dir, Grey );
129         end_found := munch( xx, yy, possibilities(1) );
130     else
131         if not is_there( x, y, start ) then
132             make_print( x, y, dir, Grey );
133         end if;
134         for i in 1 .. num_poss loop
135             xx := x; yy := y;
136             move_dir( xx, yy, possibilities(1) );
137             munchers(i) := start_muncher( xx, yy, possibilities(i) );

```

Stop At | Stop In | Clear | Run | Continue | Next | Step | Where | Print.All | Print | Current | Up | Down | Display.All | Display

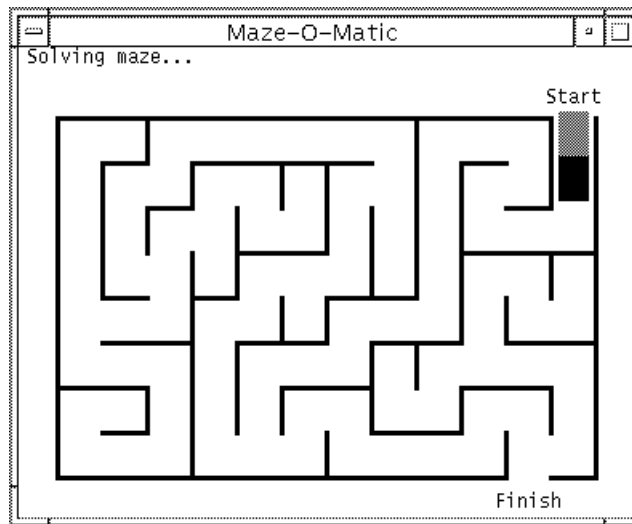
```

127         move_dir( xx, yy, possibilities(1) );
>a
a|
128         make_print( x, y, dir, Grey );
>a
a|
129         end_found := munch( xx, yy, possibilities(1) );
>a
a|
123         if num_poss = 0 then
x|

```

AdaDebug: Ready

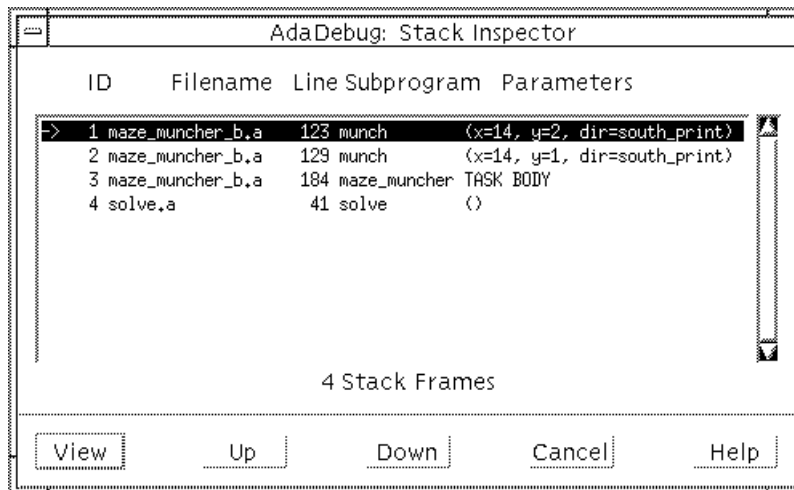
Also, `solve` draws another footprint in the maze.



`munch()` must be calling itself recursively at line 129. To check this hypothesis, bring up the Stack Inspector window.

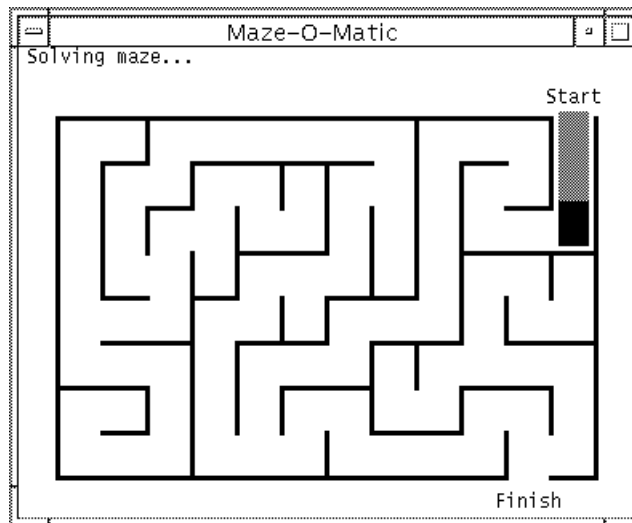
2.9 Stack Tracing

1. Choose Stack ► Inspector.



The Stack Inspector window display provides you with some key pieces of information. First, the top two entries on the stack are calls to `munch()`. Looking at the values of the parameters and comparing them to the location of the two footprints with respect to the maze—which is a 12 x 8 matrix—shows that the coordinates and the direction match perfectly.

2. Click Next six times to step through the recursive lines of code again (lines 123 through 129, returning to 123) and see the third footprint appear in the maze.



The new call also appears in the Stack Inspector.

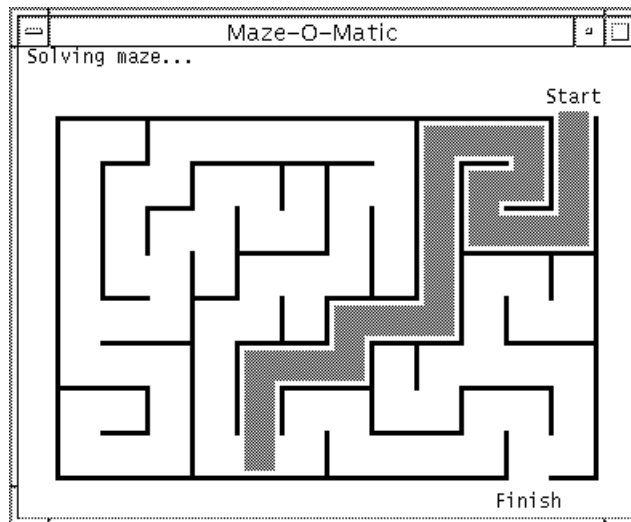
3. Select a word in line 123 and choose Breakpoint ► Clear At <selected line> (or click Clear) to remove the breakpoint at line 123.
4. Close the Stack Inspector window by clicking Cancel.

2.10 Setting a Breakpoint at `start_muncher()`

Maze footprints are not exploring all avenues (or “possibilities,” as the code calls them) when coming to an intersection. To discover why, set the next breakpoint where the `start_muncher()` procedure begins (line 137).

1. Double-click on a word in line 137 and choose Breakpoint ► Stop At <selected line> (or click Stop At).
2. Choose Execution ► Continue (or click Continue).

The maze takes off this time, drawing footprints one after another until it reaches the first two-way intersection in the maze and stops.



`solve` must spawn two new maze muncher tasks here, sending one in each direction. The breakpoint set at the line containing `start_muncher()` has stopped the program just before it creates the new task munchers.

The maze fails here because when it reaches intersections it sends a muncher in one direction but not the other.

2.11 Evaluating Parameters

Evaluate the parameters of the `start_muncher()` procedure: `xx`, `yy`, and `possibilities(i)`.

1. Drag the mouse to highlight the `xx` parameter in `start_muncher()` at line 137, then choose **Data** ► **Evaluate** <selected expr>.

AdaDebug displays the value 6 in the message pane. Count six cells from the left. The `xx` coordinate is correct, but how is the `yy` coordinate?

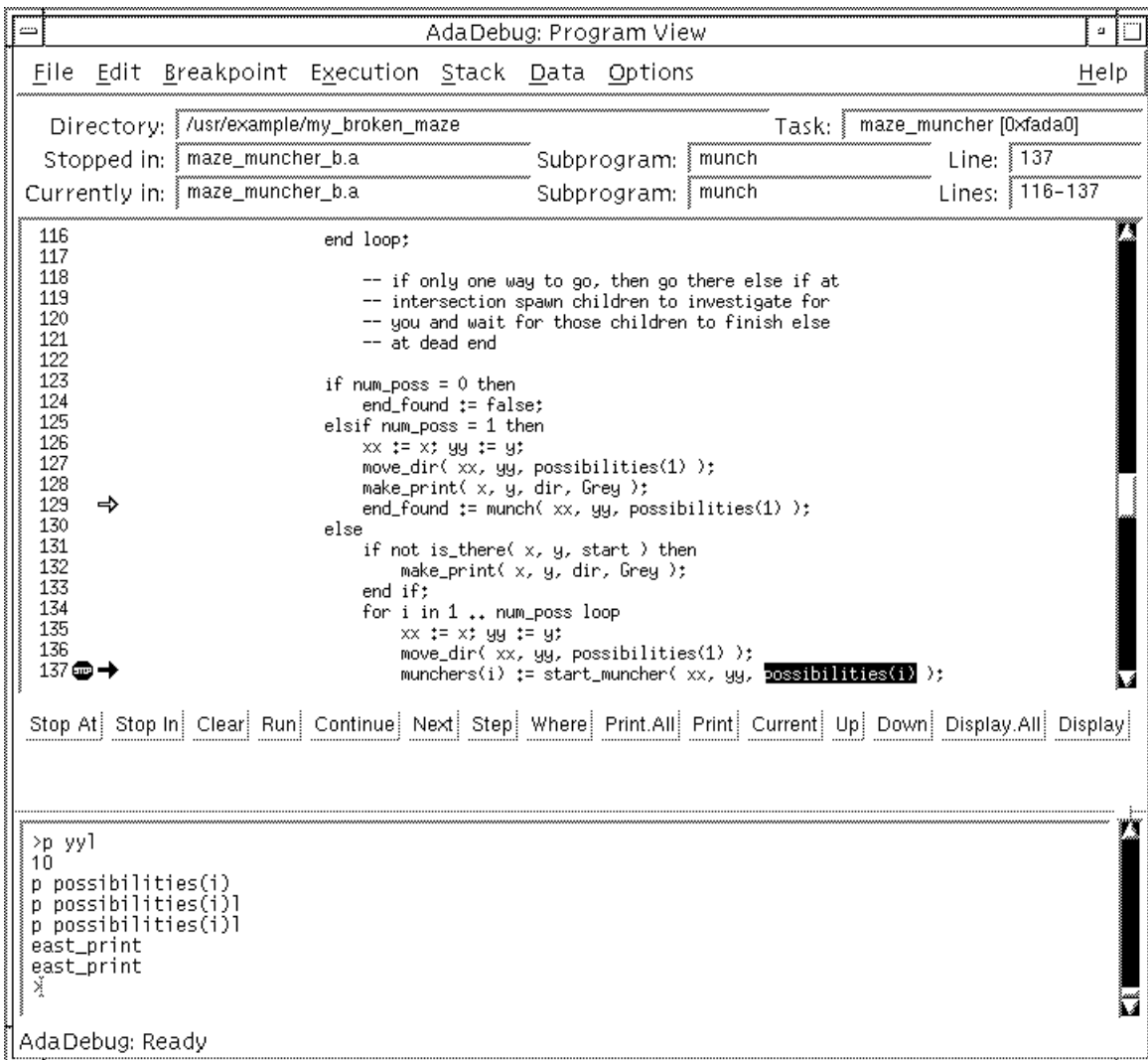
2. Select the `yy` parameter, then choose **Data** ► **Evaluate** <selected expr>.

The message pane shows that the value of `yy` is 8. Eight is the bottom row in the matrix, so this value, too, is correct. What are the `possibilities(i)`?

3. Highlight possibilities(i) and choose Data ► Evaluate <selected expr> again.

```
munchers(i) := start_muncher( xx, yy, possibilities(i) );
```

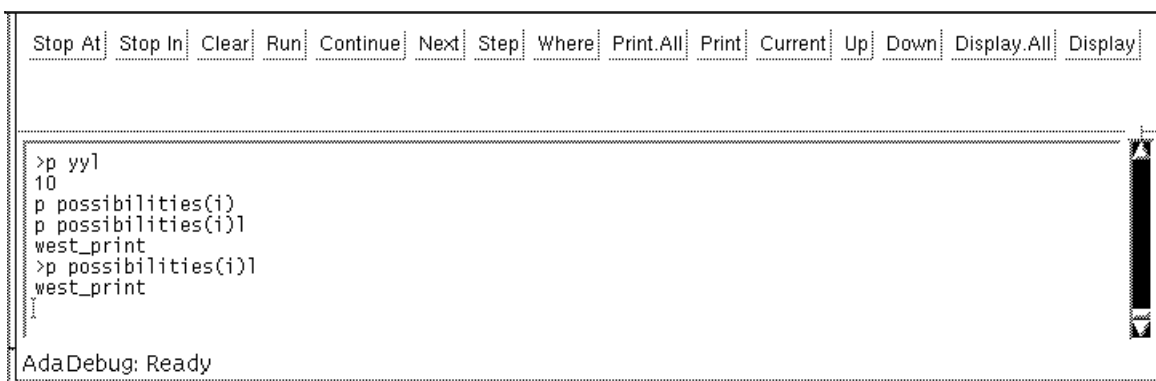
The message at the bottom of the window reports east_print. This value is also correct.



4. Choose Execution ► Continue (or click Continue).

Choosing Continue here at an intersection does not advance the maze. Instead, the program starts up a second maze muncher, and a second `maze_muncher` appears in the Task Inspector. This is the muncher that should go in the other direction, west. Test this by evaluating the arguments to `start_muncher()` for this second task.

5. Close the Task Inspector and repeat Step 1 through Step 4 to evaluate the `start_muncher()` parameters `xx`, `yy`, and `possibilities(i)`. The results are shown as follows:



```

Stop At | Stop In | Clear | Run | Continue | Next | Step | Where | Print.All | Print | Current | Up | Down | Display.All | Display
-----
>p yy|
10
p possibilities(i)
p possibilities(i)|
west_print
>p possibilities(i)|
west_print
...
AdaDebug: Ready

```

The direction is not correct: `west_print` coordinates are the same for the first muncher. It appears that the second muncher is preparing to follow the first one. Why?

Look at the line above the `start_muncher()` call, to where the program calls `move_dir()`:

```
136    move_dir( xx, yy, possibilities(1) );
```

The parameter should be `possibilities(i)`, not `possibilities(1)`. There are four possible directions, not one. No wonder the task munchers travel in one direction when they come to an intersection: the `possibilities` variable in the `mov_dir` call was coded mistakenly as a constant.

◆ **Change 1 to i, and the program behaves correctly.**

2.12 Fixing Bugs From AdaVision

Shift back to AdaVision to edit and then re-make the library.

1. Return to the AdaVision Unit View.

2. Double-click the icon for the `maze_muncher` body.

AdaVision opens the body of the `maze_muncher` unit in your editor of choice.

3. Go to line 136, which contains the call to `mov_dir()`, and edit the variable, changing `l` to `i` in the `possibilities` parameter.

4. Save the changes and quit the editor window.

5. Return to the AdaVision Unit View window and click in a blank area of the display pane to *deselect* the `maze_muncher` icon.

6. Choose Actions ► Make (or click Make) in the Unit View.

AdaVision runs `make` on *all* of the units in the library. The job completes successfully.

7. Select the `solve` executable by clicking it and then choose Actions ► Run (or click Run).

In a moment Maze-O-Matic opens, along with a Program I/O window.

8. Click in the Maze-O-Matic window to solve the maze.

9. After the program solves the first maze, click to generate a new maze, then click again to solve that maze.

You have corrected the error and `solve` now solves each of the mazes correctly.

List of Steps for AdaDebug Tutorial



This appendix contains a list of the steps in the AdaDebug portion (Chapter 2) of the tutorial. Having the steps in this format should make it easier to restart the AdaDebug tutorial session if you want to take the tutorial in more than one sitting or if you accidentally skip one of the steps.

≡ A

- 1 Select the `solve` executable in the Unit View and choose Tools ► Debug (or click on the AdaDebug button).
 - 2 Begin by scrolling halfway through the file to find parts of the code that might be relevant.
 - 3 Double-click in line 41. The first word in the line is highlighted.
 - 4 Choose Breakpoint ► Stop At *<selected line>* (or click Stop At).
 - 5 Choose Execution ► Run (or click Run).
 - 6 Click in the maze window to start the program solving the first maze.
 - 7 Choose Execution ► Step (or click Step) to step into the `attack()` procedure.
 - 8 Choose Execution ► Next (or click Next).
 - 9 Repeat Next eight more times, advancing to line 206 and reading ahead a line or two before each step to analyze what is happening.
 - 10 Choose Execution ► Step (or click Step) to step into the `start_muncher` function at line 21.
 - 11 Choose Execution ► Next (or click Next) four times until reaching line 26.
 - 12 Choose Execution ► Task Inspector to display each activated task in the Task Inspector window.
 - 13 Select the `maze_muncher` task icon in the Task Inspector and click Open to open the task in a Task View window.
 - 14 Choose Edit ► Find in the Task View.
 - 15 Type `accept start` in the text field and click Forward.
 - 16 Close the Find Pattern window.
 - 17 Double-click on line 182 in the *Task View* window and choose Breakpoint ► Stop At *<selected line>* (or click Stop At).
 - 18 Close the Task View window, but leave the Task Inspector open for later.
 - 19 Choose Execution ► Continue (or click Continue) from the Program View.
 - 20 Double-click on line 182 in the Program View and choose Breakpoint ► Clear At *<selected line>* (or click Clear).
 - 21 Choose Execution ► Next (or click Next) *twice* to advance the current line pointer to line 184.
 - 22 Choose Execution ► Step (or click Step) at line 184 to step into the start of the `munch()` function at line 70.
 - 23 Scroll or click Next a few times to advance through the code, noticing the loop from lines 106 to 116.
 - 24 Double-click on line 123 and choose Breakpoint ► Stop At *<selected line>* (or click Stop At).
 - 25 Choose Execution ► Continue (or click Continue) to have the program advance to the breakpoint you just set at line 123.
 - 26 Click Next five times. The fifth step executes the `make_print(x, y, dir Grey)` procedure (line 128), causing the first footprint to turn gray.
-

-
- 27 Click Next again to execute line 129.
 - 28 Choose Stack ► Inspector.
 - 29 Click Next six times to step through the recursive lines of code again (lines 123 through 129, returning to 123) and see the third footprint appear in the maze.
 - 30 Select a word in line 123 and choose Breakpoint ► Clear At *<selected line>* (or click Clear) to remove the breakpoint at line 123.
 - 31 Close the Stack Inspector window by clicking Cancel.
 - 32 Double-click on a word in line 137 and choose Breakpoint ► Stop At *<selected line>* (or click Stop At).
 - 33 Choose Execution ► Continue (or click Continue).
 - 34 Drag the mouse to highlight the *xx* parameter in `start_muncher()` at line 137, then choose Data ► Evaluate *<selected expr>*.
 - 35 Select the *yy* parameter, then choose Data ► Evaluate *<selected expr>*.
 - 36 Select `possibilities(i)` and choose Data ► Evaluate *<selected expr>* again.
 - 37 Choose Execution ► Continue (or click Continue).
 - 38 Close the Task Inspector and repeat the last four steps to evaluate the `start_muncher()` parameters *xx*, *yy*, and `possibilities(i)`.
 - 39 Return to the AdaVision Unit View.
 - 40 Double-click the icon for the `maze_muncher` body.
 - 41 Go to line 136, which contains the call to `mov_dir()`, and edit the variable, changing `1` to `i` in the `possibilities` parameter.
 - 42 Save the changes and quit the editor window.
 - 43 Return to the AdaVision Unit View window and click in a blank area of the display pane to *deselect* the `maze_muncher` icon.
 - 44 Choose Actions ► Make (or click Make) in the Unit View.
 - 45 Select the `solve` executable by clicking it and then choose Actions ► Run (or click Run).
 - 46 Click in the Maze-O-Matic window to solve the maze.
 - 47 After the program solves the first maze, click to generate a new maze, then click again to solve that maze.
-

Copyright 1995 Sun Microsystems, Inc., 2550 Garcia Avenue, Mountain View, Californie 94043-1100 U.S.A.

Tous droits réservés. Ce produit ou document est protégé par un copyright et distribué avec des licences qui en restreignent l'utilisation, la copie, et la décompilation. Aucune partie de ce produit ou de sa documentation associée ne peuvent Être reproduits sous aucune forme, par quelque moyen que ce soit sans l'autorisation préalable et écrite de Sun et de ses bailleurs de licence, s'il en a.

Des parties de ce produit pourront être dérivées du système UNIX[®], licencié par UNIX System Laboratories Inc., filiale entièrement détenue par Novell, Inc. ainsi que par le système 4.3. de Berkeley, licencié par l'Université de Californie. Le logiciel détenu par des tiers, et qui comprend la technologie relative aux polices de caractères, est protégé par un copyright et licencié par des fournisseurs de Sun.

LEGENDE RELATIVE AUX DROITS RESTREINTS: l'utilisation, la duplication ou la divulgation par l'administration américaine sont soumises aux restrictions visées à l'alinéa (c)(1)(ii) de la clause relative aux droits des données techniques et aux logiciels informatiques du DFARS 252.227-7013 et FAR 52.227-19. Le produit décrit dans ce manuel peut être protégé par un ou plusieurs brevet(s) américain(s), étranger(s) ou par des demandes en cours d'enregistrement.

MARQUES

Sun, Sun Microsystems, le logo Sun, Solaris sont des marques déposées ou enregistrées par Sun Microsystems, Inc. aux États-Unis et dans certains autres pays. UNIX est une marque enregistrée aux États-Unis et dans d'autres pays, et exclusivement licenciée par X/Open Company Ltd. OPEN LOOK est une marque enregistrée de Novell, Inc. PostScript et Display PostScript sont des marques d'Adobe Systems, Inc.

Toutes les marques SPARC sont des marques déposées ou enregistrées de SPARC International, Inc. aux États-Unis et dans d'autres pays. SPARCcenter, SPARCcluster, SPARCcompiler, SPARCdesign, SPARC811, SPARCengine, SPARCprinter, SPARCserver, SPARCstation, SPARCstorage, SPARCworks, microSPARC, microSPARC-II, et UltraSPARC sont exclusivement licenciées à Sun Microsystems, Inc. Les produits portant les marques sont basés sur une architecture développée par Sun Microsystems, Inc.

Les utilisateurs d'interfaces graphiques OPEN LOOK[®] et Sun[™] ont été développés par Sun Microsystems, Inc. pour ses utilisateurs et licenciés. Sun reconnaît les efforts de pionniers de Xerox pour la recherche et le développement du concept des interfaces d'utilisation visuelle ou graphique pour l'industrie de l'informatique. Sun détient une licence non exclusive de Xerox sur l'interface d'utilisation graphique, cette licence couvrant aussi les licenciés de Sun qui mettent en place OPEN LOOK GUIs et qui en outre se conforment aux licences écrites de Sun.

Le système X Window est un produit du X Consortium, Inc.

CETTE PUBLICATION EST FOURNIE "EN L'ÉTAT" SANS GARANTIE D'AUCUNE SORTIE, NI EXPRESSE NI IMPLICITE, Y COMPRIS, ET SANS QUE CETTE LISTE NE SOIT LIMITATIVE, DES GARANTIES CONCERNANT LA VALEUR MARCHANDE, L'APTITUDE DES PRODUITS À RÉPONDRE À UNE UTILISATION PARTICULIÈRE OU LE FAIT QU'ILS NE SOIENT PAS CONTREFAISANTS DE PRODUITS DE TIERS.

CETTE PUBLICATION PEUT CONTENIR DES MENTIONS TECHNIQUES ERRONÉES OU DES ERREURS TYPOGRAPHIQUES. DES CHANGEMENTS SONT PÉRIODIQUEMENT APPORTÉS AUX INFORMATIONS CONTENUES AUX PRÉSENTES. CES CHANGEMENTS SERONT INCORPORÉS AUX NOUVELLES ÉDITIONS DE LA PUBLICATION. SUN MICROSYSTEMS INC. PEUT RÉALISER DES AMÉLIORATIONS ET/OU DES CHANGEMENTS DANS LE(S) PRODUIT(S) ET/OU LE(S) PROGRAMME(S) DÉCRITS DANS CETTE PUBLICATION À TOUTS MOMENTS.



Adobe PostScript

