CS1110 Introduction to Systematic Programming
Week 3 -- Practical Class 1

Compiler Error Messages and Compiler Listings

So far we have assumed that when you compile an Ada program everything goes smoothly. However suppose that the source file (`divide.adb`, say) contains programming errors. The Ada compiler will output various error messages to the VDU screen indicating the location at which each error was detected and also giving an indication of its nature. When a compilation fails because of errors in the source file, an executable file is not produced (nor are the binary relocatable file `divide.o` and the Ada library info. file `divide.ali`).

We are going to debug a program `divide.adb` located in the Practical Materials Folder of the CS1110 Web-site to your own Ada directory. First you must copy the program to your own Unix user area. So in Netscape go to the Practical Materials Folder and then right-press with the mouse on the link `divide.adb` and save the file in your own user area by selecting **Save Link As ..** from the pop-up menu that appears and then type ~/`divide.adb` in the Selection text box.

Compile `divide.adb`

The program `divide.adb` is a modified version of the program in Unit 3 of the ISP notes containing several deliberate errors. Now try to compile `divide.adb` in the standard way by typing the following command in a terminal window[1]:

```
gnatmake divide.adb
```

the compiler will output a number of **error messages** to the VDU screen:

```
divide.adb:10:22: missing ";"
divide.adb:18:28: "LOOP" expected
divide.adb:33:01: "END Divide;" expected
```

For example the first error message above indicates that a "missing semicolon" error was detected on line 10, column 22 of the file `divide.adb`.

Look at the error messages produced by the compiler to give you clues as to what is wrong. There are three fairly obvious errors, track these down and correct the errors using Emacs, save the file and then compile it again

Alas, a different error[2] appears this time! However don't give up! Last time the compiler found various **syntax errors** (for example, missing semicolons or missing Ada keywords etc.) and listed these. As the compiler found several syntax errors, it did not attempt to do any **semantics checking** (for example checking that variables are declared; that there are no type mismatches in arithmetic expressions and that I/O procedures are called with the correct number and type of parameters etc.). Now that you have corrected the basic syntax, the compiler is able check the semantics and may detect further errors such as undeclared variables.

You should see an error message of the form:

```
divide.adb:17:04: "Quotient" is undefined (more references follow)
divide.adb:17:04: possible misspelling of "Quoteint"
```

The error message indicates that an identifier (`Quotient`) on line 17 has not been properly declared in the variable declaration section of the program (and that other references to

---

[1] or by opening the file in Emacs and using the **Compile** or **Build** command from the Emacs **Ada** menu.

[2] This assumes you found and corrected all three syntax errors. If not, try again! If after several attempts, you are 'stuck', ask the lab. demonstrator for help.

`Quotient` appear later in the program). Correct this error using Emacs (you may need to look carefully at the declaration to spot the error), save the changes and compile the program again.

This time the program will compile -- well almost!! Try to find and correct the remaining error yourself.

Emacs Line-Number Mode
Emacs has a **line number mode** which displays the current line number of the cursor on the Emacs **status line** (that is the line in reverse video near to the bottom of the Emacs window). This enables users to see immediately the position of the cursor in the file and so avoid the need to count lines manually.

To move the cursor to a particular line number in small files it suffices to use the cursor up and down arrow keys whilst keeping an eye on the line number in Emacs' Status line. For larger files scroll to the required line using the line number displayed on the status bar as a guide.

Using the history list to save typing
As you type in UNIX commands in an `xterm` window, they are stored on a **history list**. You can recall previous commands[3] by pressing the up-arrow key to scroll up the history list. The down-arrow key can be used to scroll back down the list.

This can save retyping commands. Suppose you compile your program with the command

```
gnatmake divide.adb
```

but it contains compilation errors, then, after correcting the errors using Emacs, all you need do to recompile is to press the up-arrow key (with the focus in the `xterm` window) until the required `gnatmake` command reappears and press `<Return>`; the compilation command is repeated.

Note each separate `xterm` window has its own history list, which is initially empty when the window is created. So make sure you reuse the original `xterm` window; DON'T start a new one for each compilation.

The history mechanism can also be useful if you mis-type a longish command, say

```
gantmake divide.adb
```

The command name `gnatmake` has been mispelt. You need to correct the spelling but rather than retyping everything, just recall the offending command using the up-arrow key, then use the left-arrow key to position the cursor just after `an`, delete these two characters and type `na`, instead, then press `<Return>`[4] and the corrected command will be executed.

Exercise Copy the file `factors.adb` from the Practical Material folder to your own directory .

Full Compiler Listings with Errors
Although the error listing of the form described above is useful for debugging relatively short programs, the GNAT Ada compiler can be made to produce a **full compiler listing** of the source file with line numbers and with any compilation errors indicated at the appropriate points in the program. The extra context information often makes it easier to spot bugs particularly in long programs.

After copying the file `factors.adb` to your own directory compile it as follows by typing in an xterm window:

---

[3] Only the last 20 or so commands are stored on the list.
[4] There is no need to move the cursor back to the end of the line before pressing `<Return>`

```
gnatmake factors.adb -cargs -gnatl
```

The file `buggy.adb` is a buggy version of the example program in Unit 4 of the ISP notes. A compilation listing of the source program is output to the VDU screen complete with error messages. The option `-gnatl` (the final character is lowercase 'ell' not a 'one') tells the compiler to produce a **l**ong error listing.

Alternatively, instead of outputting a listing to the VDU screen, you can arrange for the listing to be sent to a file (`error.lst`, say) by using UNIX I/O re-direction as follows:

```
gnatmake factors.adb -cargs -gnatl > error.lst
```

In general any UNIX command followed by `>`*`filename`* causes any output produced by the command to be sent to the specified file rather than to the VDU screen.

Experiment with this and view the resulting listing file, `error.lst`, using Emacs. Try debugging the program `factors.adb` with the aid of the full compiler listing. Use Emacs to correct the errors, save the corrected version of the file and then recompile.

As before when you have corrected the basic syntax errors, the compiler will be able detect some semantic errors. **Delete the old error listing file** (use the command `rm error.lst`) and then generate another full compiler listing and use it to track down and correct these semantic errors.

Hint: Missing or incorrect `WITH` and `USE` clauses can cause the compiler to generate errors regarding incorrect parameters in calls of the I/O procedures `Put` and `Get`.

Some Debugging Hints

1) A stitch in time...; debugging can be a time-consuming process and is best avoided or at least reduced to a minimum! Thus when programming always adopt a careful methodical approach -- check each line as you type it in. It is often easier to spot typos at this stage rather than in the completed program. As you use an identifier check that you have declared it. If you don't put bugs into your programs, then you won't need to remove them!

   Don't attempt to take short-cuts on program layout or on the use of meaningful identifiers. It is much easier to find bugs in programs which are well-laid out. **Never say (or think)** "I'll get it working and then fix the layout".

2) Don't correct one error at a time. Use the error listing and try to correct a number of errors before recompiling.

3) **Look carefully what you have actually typed** in the source program, NOT what you think you have typed!

4) If you can't spot an error on the line indicated, try inspecting the previous few lines for errors. Sometimes the Ada compiler can't detect an error until a few lines after it occurs.

5) Read the error message carefully and try to understand it! It should give you vital clues as to the nature of the error. If you don't understand fully what a compiler error message means, you can still inspect the indicated line carefully to try to spot the error. Next time you will know what to look for when you see an error message of this type.

6) Check that zero is not typed as the capital letter O (or vice-versa) and that one is not typed as the lower-case letter l (or vice-versa).

7) Check carefully that the spelling of identifiers (e.g. variable and procedure names) is consistent throughout the program. Transpositions of characters is a common typo which is sometimes difficult to spot, for example `Quotient` and `Quoteint`.

8) If there are a lot of error messages relating to I/O steps, check that you have `WITH`ed and `USE`d the correct I/O libraries. In particular error messages suggesting possible missing instantiations of `Ada.Text_IO.Integer_IO` or `Ada.Text_IO.Float_IO` probably mean you have forgotten to import `CS_Int_IO` or `CS_Flt_IO` respectively[5]. Such errors are also likely to give rise to error messages regarding the parameters of `Put` and `Get`.

If you are using format parameters in I/O steps, check that you are using the appropriate formal parameter names: `Width` for `Integer` output, `Fore` and `Aft` for `Float` output.

[5] The CS I/O libraries are not a standard feature of Ada. They were specially written for the ISP course and the GNAT compiler does not 'know' about them.