CS111 Introduction to Systematic Programming

An Overview of the Compilation Process
Computers cannot directly execute programs written in high-level languages such as Ada, C or Java. The program must first be translated into binary machine instructions that the computer can execute. This translation process is known as **compilation** and is peformed by a program known as a **compiler**. The particular compiler used in this course is called `gcc`.

The first stage of the compilation process is to check the program for **syntax errors** such as missing semi-colons, mis-spelt keywords etc.. If errors are found error messages are output to help in the debugging process and the compilation process terminates. If no syntax erros are found, the compiler goes on to check for **semantic errors** such as undeclared variables, type mis-matches in expressions and incorrect calls to library I/O procedures etc.. If errors are found in the semantics, error messages are again output to help in the debugging process and the compilation process terminates. If no errors are found, the compiler proceeds to generate machine code and stores it in an **object file**. The original high-level language program in Ada or C is often known as the **source file** to distiguish it from the binary program in the object file.

Although the object file contains binary machine instructions, it does not yet form a complete program that can be directly executed. First it must be **linked** to form a complete **executable program** by a utility called a **linker**. The linker combines the object file produced by the compiler with object files containing the libraries used by the program (for performing I/O etc.) and also with an object file containing some run-time support code. The purpose of the run-time support code is to pass control from the operating system (in our case UNIX) to our program when it is executed and to pass control back to the operating system when our program terminates. See Figure 1 overleaf.

Object code and executables are binary files and are not intended to be read by humans. If you display them using a pager program such as `more` or `less` or if you open the file with Emacs, the output will be incomprehensible (and may crash the terminal window). Do not attempt to print binary files on a line-printer as this will waste reams of paper and will probably crash the printer.

Machine Dependence of Executables and Object Files
When we write a program in a high-level language such as Ada, the source code we write is the same (more or less) whether we intend to run the program on a Sparc, a PC or a Macintosh computer. We say that the high-level language is **machine-independent** or **platform-independent**. The same is not true for object code and executable programs. Each make of machine has a different type of CPU chip and a different set of binary machine instructions. The machine code for, say, a Sparc processor cannot be run on a PC (which has a Pentium processor) nor on a Macintosh[1] which has yet another type of CPU (called a PowerPC). We say that the machines have different **architectures**.

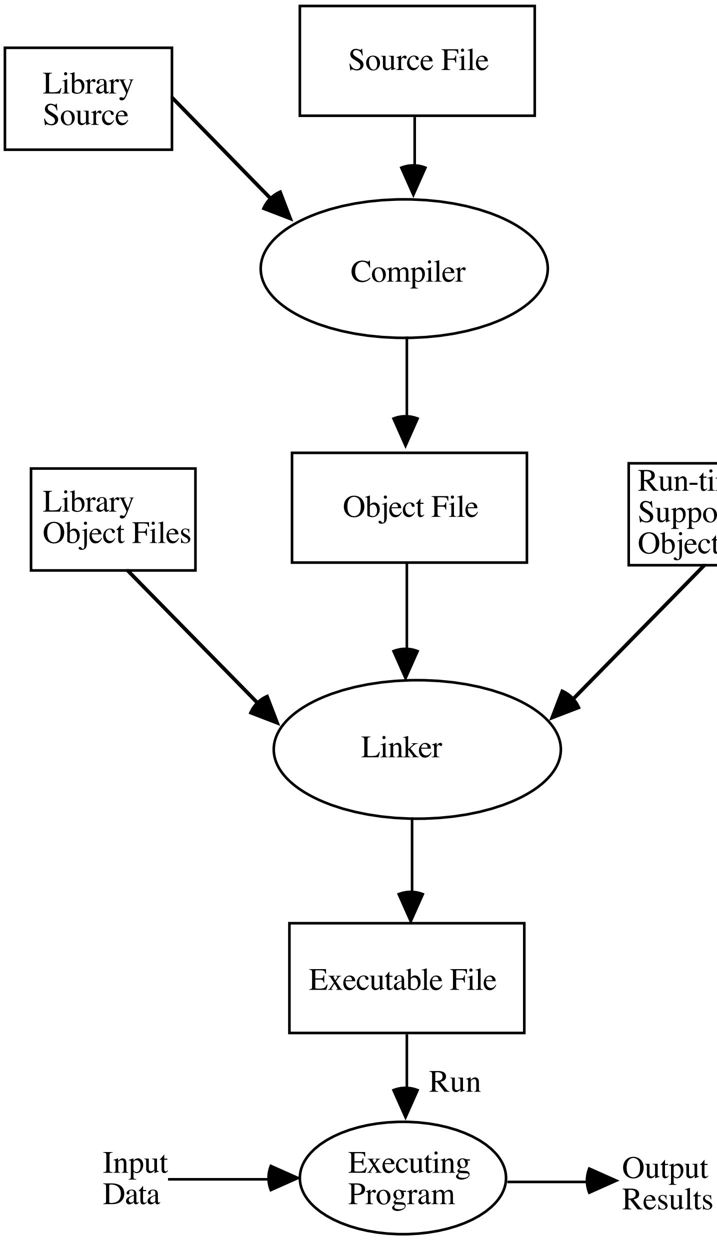
Executables and object code are **machine-dependent** and so it is not possible to run a program compiled on a Sparc running Solaris on a PC (even if it is also running under Solaris) or vice-versa. The program must be completely recompiled for the new machine.

Note that the machines in MB357, MB202, MB268 and MB264/6 all run under Solaris and present an almost identical working environment. A user who creates a file on a computer in one room can later access it from a system in another room is accessible on all these system However the machines in 264/6 are Sparcs whereas those in the other room are PCs.

[1] Older Macintoshes have yet another kind of CPU chip called a Motorola 68000.

Figure 1

Compilation of a High-Level Language



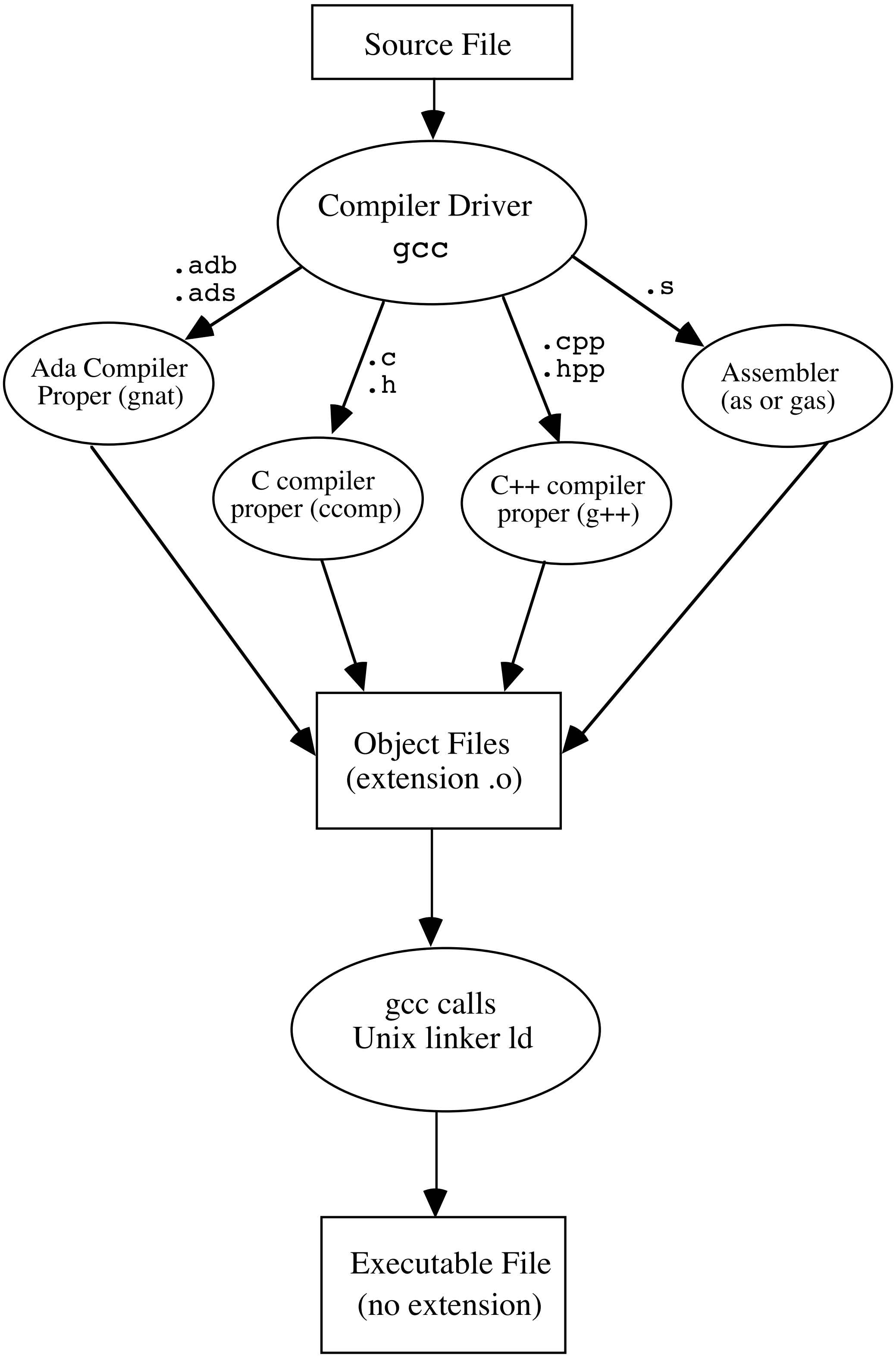
`gcc` as a Multi-language Compiler
Actually `gcc` can be used to compile programs written in the high-level languages[2] C and C++ as well as Ada programs. In fact `gcc` is not really a compiler but rather a **compiler driver**. It looks at the extension of the source file supplied to determine which language the file is written in and then calls the appropriate compiler for the language concerned. `gcc` can also be used to assemble (compile) assembly language programs and to link object code. It behaves as follows:

Type of File	**Extension**	**Action**
Ada source file	`.ads` or `.adb`	Call the Ada compiler proper `gnat`
C source file	`.h` or `.c`	Call the C compiler proper `ccomp`
C++ source file	`.hpp` or `.cpp`	Call the C++ compiler proper `g++`
Assembly Language	`.s`	Call the UNIX assembler `as` (or `gas`)
Object code	`.o`	Call the UNIX linker `ld`

[2] Extended versions are also available for compiling Fortran and Modula-2. although these are not installed at Aston.

Figure 2

gcc as a Multi-Language Compiler



Ada Library Information Files
When an Ada program is compiled with `gcc`, two files are actually produced: an object file (with extension `.o`) as described above plus an **Ada Library Information** file (with

extension `.ali`). The Ada Library Information file contains information about the libraries used by the Ada program and is used to control the linking process. More specifically it is used to ensure that

> that run-time support code is generated (a process called **binding**). This code ensures that all the libraries are correctly **elaborated** (initialised) at run-time.
>
> the object code for all the required libraries and the run-time support code is linked
>
> the correct versions of the library object code and run-time support code are linked

Note that the C and C++ compilers do not generate a Library Information file and so with these languages it is more difficult to ensure that all the required libraries are linked and it is sometimes possible to link incompatible object files (perhaps generated by an different versions of `gcc` or compiled from out-of-date library source code). Thus it is possible to form an executable that may crash at run-time even though the source code contains no errors.

This cannot occur with Ada as the library information files provide all the necessary information to automate the linking process and ensure that a consistent executable is produced.

The Gnat Ada Compiler
More on the compilation, binding and linking process
To compile an Ada program `somefile.adb` (say), we usually proceed as follows:

```
gnatmake somefile.adb
```

It is instructive to consider more fully what this involves (see also figure 3). First the Ada program is compiled

```
gcc -c somefile.adb
```

The call to `gcc` causes the Ada program to be compiled (converted into binary machine instructions or **object code**) to produce an **object file** `somefile.o` plus an **Ada library information file** `somefile.ali` which contains information about the Ada libraries imported (`WITH`'ed) by the file `somefile.adb`.

To convert the object file into an executable file requires two further stages of processing:

1. **Binding**
This is performed by the program `gnatbind`

```
gnatbind somefile.ali
```

Binding automatically generates an Ada package in files called `b~somefile.ads` and `b~somefile.adb`

The bind files contain an Ada function called `main`. The Ada code in `main` is generated automatically from information in the Ada Library Information file (`.ali` file).

When an Ada program is executed, UNIX does not call <u>your</u> main Ada procedure directly; instead it calls `main` in the bind package. The function `main` initialises the Ada units making up the program, that is the main Ada program and any Ada libraries that it uses, (a process called **elaboration**) and <u>then</u> it calls your main Ada procedure. When your Ada procedure terminates, control is returned to the function `main` which performs certain tidy-up operations (a process called **finalisation**) before control is returned to UNIX.

2. **Linking**
This is performed by the program `gnatlink`.

```
gnatlink somefile.ali
```

and actually consists of two sub-stages:

2a. Compile the bind package to produce an object file `b~somefile.o` (and a library information file `b~somefile.ali`)

```
gcc -c b~somefile.adb
```

2b. Link all the object files to form an executable called `somefile`

```
gcc -o somefile b~somefile.o somefile.o  (+ .o files of Ada libraries)
```

In the last step `gcc` calls the UNIX linker `ld` to link all the necessary object files (of the main Ada program, of any Ada packages required and of the bind file) to produce the final executable program called `somefile`.

Note the bind files `b~somefile.ads`, `b~somefile.o` etc. are usually deleted automatically by `gnatmake` after the program is linked unless the program is compiled with the debug option `-g`. In this case the bind files **are** retained as they need to be accessed by the debugger `gdb`. Occasionally bind files may not be deleted if the binding/linking process fails unexpectedly (for example due to lack of sufficient disk space). See Figure 3 overleaf.

Calling the Compiler and Binder/Linker separately
Rather than invoking the compiler, binder and linker 'in one go' by calling `gnatmake`.

```
gnatmake somefile.adb
```

it is possible to compile the program in two stages:

```
gcc -c somefile.adb
gnatbl  somefile.ali
```

The first command invokes `gcc` to compile the program `somefile.adb`. The second command `gnatbl` binds and links the program by calls to `gnatbind` and `gnatlink` to produce the executable `somefile` (plus object and Ada Library information files `somefile.o` and `somefile.ali`.

Compiling with the switch `-g`
If a program is compiled with a command of the form

```
gnatmake -g somefile.adb
```

the object files produced by the compiler, namely `somefile.o` and `b~somefile.o`, contain extra information embedded in the machine code (this information includes variable names and line number information from the source file). When a program is run under the control of a debugger utility (such as `gdb`, `gdbtk` or `gvd`), this extra information is read by the debugger utility and used to display useful debugging information.

Note if only `gcc` is invoked with the switch `-g` as in

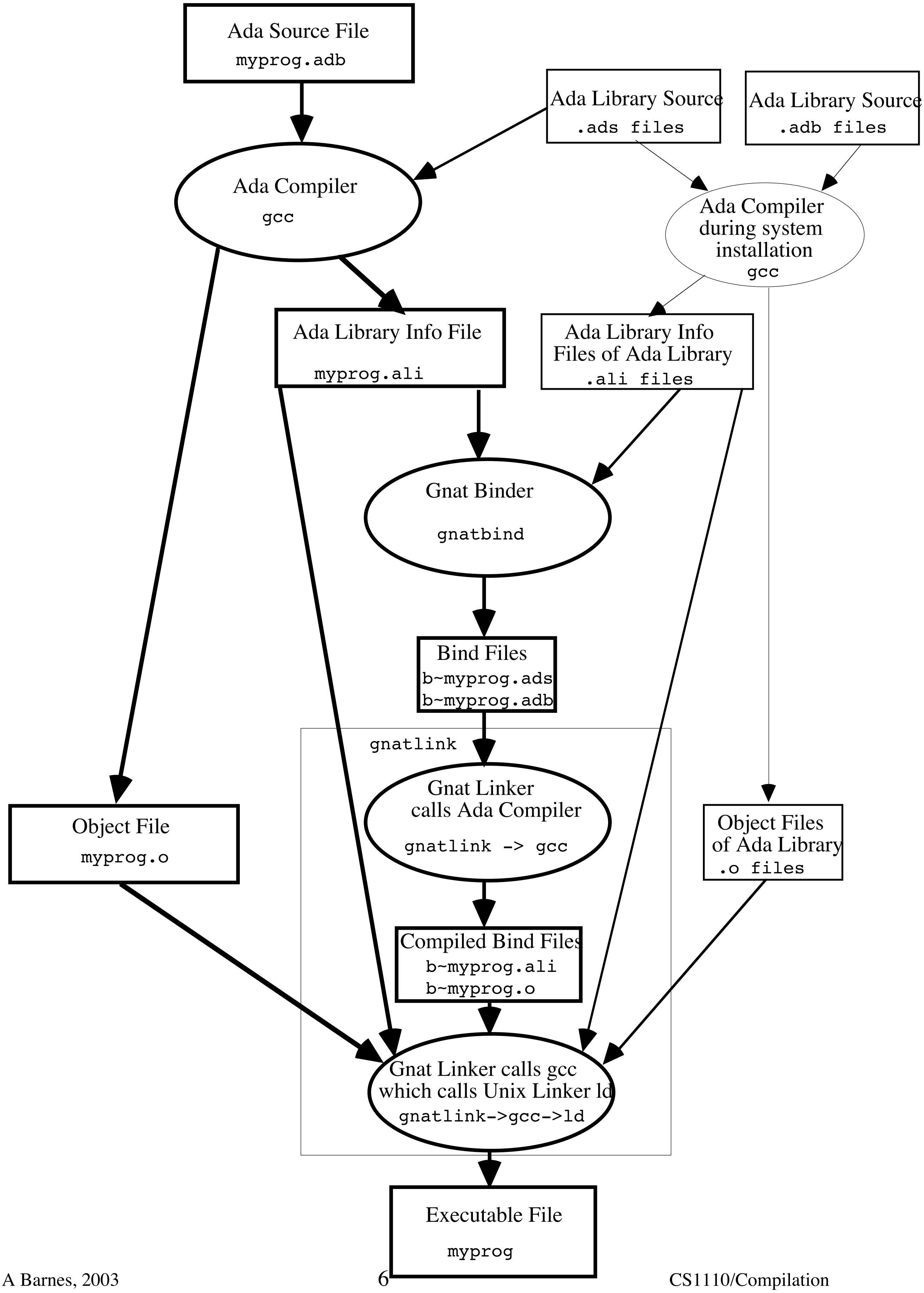
```
gcc -c -g somefile.adb
gnatbl  somefile.ali
```

then only the main object file (`somefile.o`) contains debugging information; the bind file (`b~somefile.o`) does not.

Saving Disk Space
Object files and Ada Library Information take up valuable disk space and periodically such unwanted files should be deleted. This can be done by issuing the UNIX command `clean` in the directory where the files are located. Unwanted executable files and `core` files (sometimes formed when a program crashes) should also be deleted periodically (using the UNIX command `rm`) to save disk.space. Provided the Ada source files are retained the executables can be regenerated simply by recompiling.

Figure 3

The Gnat Ada Compilation Process



File naming Conventions

With the Gnat Ada the following file naming conventions are used[3]:

The source files for Ada programs must have the extension `.adb`

The source files for Ada libraries must have the extensions `.ads` and `.adb.`

Object file have the same basename as their source files but with the extension `.o`

Ada Library Information files have the same basename as their source files but with the extension `.ali`

Executable files have the same basename as their source files but with no extension (or with the extension `.exe` on Windoze)

The basename of the file containing the program should be the same as the name of the main Ada procedure, but with all letters converted to lowercase.

Both `gnatmake` and `gcc` issue **warning** messages if the main program is stored in a file with a basename that is not the same as the **unit name** (the name of the main procedure converted to lowercase). However these are only warning messages and the program will be compiled and linked even though the the main procedure name does not match the name of the source file.

How `gcc` finds the Ada Libraries

During compilation of an Ada program the compiler needs to be able to locate the source code of the Ada Libraries imported (`WITH`'ed). It does this by looking in the current directory and if the required files can't be found there it inspects the environment variable `ADA_INCLUDE_PATH`. This variable specifies a list of directories to search for the Ada library package specifications (`.ads` files). If the required files can't be found in any of the directories on `ADA_INCLUDE_PATH` the compiler looks for the file in a directory containing the standard Ada libraries (the location of this directory is decided when the compiler is installed).

During the binding and linking process the object files holding the compiled versions of the libraries (`.o` and `.ali` files) need to be located. Again this is done by first looking in the current directory. If the required files are not to be found there, the compiler inspects the environment variable, `ADA_OBJECTS_PATH` in this case, which specifies a list of directories which the binder `gnatbind` should search for Ada Library Information files (`.ali` files) and that the linker `gnatlink` should search for Ada Library Information files (`.ali` files) and object files (`.o` files). If the required files can't be found in any of the directories on `ADA_OBJECTS_PATH`, the compiler looks for the file in a directory containing the standard Ada libraries (the location of this directory is decided when the compiler is installed).

These environment variables are set to appropriate values as part of the login process and so it is not necesary for novice users of the system to bother about these details. To inspect the current values of the Ada include and object paths issue the commands (in a terminal window)

```
printenv ADA_INCLUDE_PATH
printenv ADA_OBJECTS_PATH
```

Location of the Ada Libraries

The source and object files for the special CS libraries (`CS_Int_IO`, etc.) used in the ISP course may be found in the directory:

```
/usr/local/staffstore/CSAdaLib/
```

The source files for the standard Ada libraries on Sparc systems (`Ada.Text_IO`, etc.) may be found in the directory:

```
/usr/local/gnat3.12p/lib/gcc-lib/i386-pc-solaris2.6/2.8.1/adainclude/
```

The Ada Library Information files for standard Ada libraries may be found in the directory:

```
/usr/local/gnat3.12p/lib/gcc-lib/i386-pc-solaris2.6/2.8.1/adalib/
```

The corresponding object files are in a UNIX object archive file `libgnat.a` in this directory.

---

[3] For ways of avoiding these restrictions consult the Gnat User Guide in the file `/usr/local/gnat3.12p/gnat.3.12p-docs/gnat_ug.txt`