Using GNAT on Unix

Mark Allen Weiss
School of Computer Science
Florida International University
University Park
Miami, FL 33199

Abstract

This handout provides a brief overview of the Ada95 compiler *gnat* running under SunOS. It includes instructions for compiling, linking and running, as well as a brief summary of the basic differences between Ada83 and Ada95 that will be used to update the textbook *Data Structures and Algorithm Analysis in Ada*.

1 Using *gnat*

Before you can use *gnat*, you must make certain things visible in your search path. How this is done is very system dependent; in particular it depends on the name of the directory where the *gnat* compiler, linker, libraries, and other files are stored. Here's the setup you'll need at FIU:

```
setenv GCC_EXEC_PREFIX /homes/compsci/sun4/gnat/lib/gcc-lib/sparc-sun-sunos4.1.3/2.6.2/
setenv ADA_INCLUDE_PATH /homes/compsci/sun4/gnat/adainclude
setenv ADA_OBJECTS_PATH /homes/compsci/sun4/gnat/lib/gcc-lib/sparc-sun-sunos4.1.3/2.6.2/adalib
setenv LIBRARY_PATH /homes/compsci/sun4/gnat/lib/gcc-lib/sparc-sun-sunos4.1.3/2.6.2/adalib
setenv C_INCLUDE_PATH /homes/compsci/sun4/gnat/lib/gcc-lib/sparc-sun-sunos4.1.3/2.6.2/include
set path=(/homes/compsci/sun4/gnat/bin $path)
```

These commands should be placed in your `.cshrc` file. You'll need to logout and log back in for them to take effect (or else you can `source .cshrc`).

As is typical of compilers running under UNIX, *gnat* recognizes the type of file by its suffix. Figure 1 lists some of the basic suffixes. Suppose we want to compile the Ada program in Figure 2.

Because the compilation unit name is `FirstProgram`, the source must be placed in a file name `FirstProgram.adb` (a compiler warning will be issued otherwise). [1] Then we issue the command: `gcc -c FirstProgram.adb`

[1] If `F` is a separate routine in procedure `G`, then `F` should be placed in file `g-f.adb`.

.adb	An ada compilation unit (or package body)
.ads	A package specification
.ali	An ada library map for a compilation unit
.o	An object file for a compilation unit
.c	A C program

Figure 1: Suffixes used by *gnat*

```
-- Compilation Unit Name Is FirstProgram
with Ada.Text_IO; use Ada.Text_IO;
procedure FirstProgram is
begin
   Put_Line( "Testing: 1, 2, 3" );
end FirstProgram;
```

Figure 2: A simple Ada program

That's right! We use *gcc*. When the compilation is finished (and is successful), we'll have generated `FirstProgram.ali` and `FirstProgram.o`. We can link the compilation unit by supplying the command

`gnatbl -o FirstProgram FirstProgram.ali`

The `-o` option is used to give a meaningful name to the resulting executable file; otherwise `a.out` will be used. We can then run the program by issuing the command

`FirstProgram`

WARNING: If you accidentally type `gcc -o FirstProgram.adb`

or

`gnatbl -o FirstProgram.adb FirstProgram.ali`

then you will clobber your source code. It cannot be unclobbered. Do not use `-o` with `gcc`, and when you use `-o` with `gnatbl`, make sure that the filename that follows is the one you really want to use to store the executable.

When packages are involved, the procedures are similar, except that the package specification must be in a `.ads` file, and the body in a `.adb` file. The compilation order is unchanged from Ada83. Thus for the complex number package and test described in Figures 1.6 to 1.8 of the text, we now have three files:

`complex_numbers.ads`:	Package specification for `complex_numbers`
`complex_numbers.adb`:	Package body for `complex_numbers`
`complex_test.adb`:	Compilation unit to test the package

The order of compilation is the same as the order that the files are listed above. We then issue the `gnatbl` command with `complex_test.ali` as the compilation unit library file.

2 Simple Ada95 Changes That Are Used Throughout

Throughout the new code, we make use of two changes introduced in Ada95.

2.1 Text_IO is slightly renamed

As Figure 2 shows, the statements

```
with Text_IO; use Text_IO;              -- Ada83 style
```

have been replaced with

```
with Ada.Text_IO; use Ada.Text_IO;  -- Ada95 style
```

Other packages include `Ada.Integer_Text_IO`.

2.2 "=" Is Overloadable

In Ada83, "=" was not an overloadable operator. As a result, generic packages that needed an equality function required a generic parameter of the form

```
with function Equals( Left, Right: Element_Type ) return Boolean;
```

instead of the more natural

```
with function "="( Left, Right: Element_Type ) return Boolean;
```

This rule has been repealed in Ada95; "=" is now overloadable. This changes code for linked lists and hash tables in the text. Furthermore, when "=" returns a `Boolean`, "/=" is automatically defined consistently. For some reason, a similar rule still does not apply for "<" and ">".

3 A Quick Tour of Object-Oriented Additions That Are Used

Perhaps the fundamental change of Ada95 is the introduction of support for *object-oriented*, rather than *object-based* programming. The key feature of this is *inheritance*.

3.1 Object-Oriented Approach: Inheritance

Inheritance is used to *derive* new types from existing types. Both the existing type and new type are records, and the new type inherits all components from the existing type. It can then add additional components. Additionally, any function that was defined for the original type is automatically defined for the new type. The meaning of these functions can be altered by an explicit redefinition, and additional functions can be added. Inheritance is discussed in more detail in the handout *Using Inheritance for Heterogeneity*. Except as noted in Section 3.2, we do not directly use inheritance in the design of data structures. However we use inheritance to design type hierarchies that can be used to instantiate our existing data structures.

3.2 Construction and Destruction (Typical C++)

In the text, we have several data structures that use dynamically allocated memory (i.e. memory obtained via a call to `new`). When an instance of such a data structure exits scope, then unless arrangements are made, this memory is lost forever.

Consider the code in Figure 3. Procedure `Inner` declares a local stack, and then pushes 10 items. The `Stack_List` class implements this by dynamically allocating a record for each `Push`. Under Ada83, when a call to `Inner` terminates, the object `Stack` exits scope, but the dynamically allocated memory remains allocated (and now unreferenced); thus Ada83 will leak memory. To fix this, the user would need to add a call to `Make_Empty` (and this might still leak a little memory). Under Ada95, we can declare that a `Stack` is inherited from the type `Ada.Finalization.Limited_Controlled`. We can then write a procedure named `Finalize` that is automatically called whenever a `Stack` object exits scope. In this case, we would have `Finalize` deallocate the memory consumed by the `Stack` object. `Finalize` is the C++ equivalent of a destructor.

Similarly, the procedure `Initialize` is the C++ equivalent of a constructor. Because components of a record can be initialized under Ada83, we don't need to use `Initialize` as often as `Finalize`. A third routine, known as `Adjust`, is useful for allowing copying. For more details, see the handout *Using Finalization in Ada95*.

```
with Stack_List;

procedure Outer is
   package My_Stack is new Stack_List( Integer ); use My_Stack;

   -- Push 10 items onto the stack.
   -- With linked list, this makes 10 calls to new.
   -- This is a memory leak in Ada83.
   -- It is not a leak in Ada95 (using revised Stack_List).
   procedure Inner is
      S : Stack;
   begin
      for i in 1..10 loop
         Push( i, S );
      end loop;
   end Inner;
begin
   for i in 0..100_000 loop
      Inner;
   end loop;
end Outer;
```

Figure 3: Example of a memory leak in Ada83 that can be fixed in Ada95