7. Compilers

Math3101/5305 Computational Mathematics

Version: April 10, 2017

Outline

Introduction

Our aim in this lesson is to gain an overview of how a program gets from a human-readable form to one that the CPU can execute, and hence understand something of what Julia does behind the scenes.

Four example programs — really four versions of the same program — will provide concrete illustrations of the key concepts. The first two programs are written in Fortran, the third in Julia, and the fourth in C. Each one evaluates an $N \times N$ quadratic form

$$\boldsymbol{v}^{\mathsf{T}} A \boldsymbol{v} = \sum_{i=1}^{N} \sum_{j=1}^{N} v_i a_{ij} v_j,$$

where the entries of the matrix A and the vector $\boldsymbol{v}$ are pseudo-random numbers from the unit interval.

Example 1 — Fortran

A compiler is a program that translates one or more source files into an executable file.

The command

```
$ gfortran example1.f08
```

compiles the Fortran 2008 source file **example1.f08** into the executable file `a.out`; to execute the latter, type

```
$ ./a.out
```

The program produces output like

```
1000-by-1000 quadratic form.
Fortran intrinsics: ans = 0.1240422463112OE+06
quadform function:  ans = 0.1240422463112OE+06
```

Whereas `example1.f08` is a text file, the executable `a.out` is a binary file so you cannot read it with a standard text editor. The command

```
$ ls -lh
```

shows that `example1.f08` is 1.1K whereas `a.out` is 14K. Thus, in this case the executable is more than 10 times larger than the source file.
The command

```
$ file a.out
```

describes `a.out` as an "ELF 64-bit LSB executable, x86-64". Here, ELF (Executable and Linkable Format) is a standard binary file format, and LSB (Least Significant Bit) indicates the byte order.

Makefiles

The make program is the traditional build tool for Unix. By writing a suitable makefile, a programmer can indicate how a given program should be built out of the files containing its source code as well as any necessary libraries. A makefile consists of one or more rules of the form

```
1 TARGET: PREREQUISITES
2         COMMAND
```

(The space before COMMAND *must* be a tab.) The rule tells make that the TARGET can be built if the PREREQUISITES exist by running the COMMAND.

For example, the rule

```
1 example1: example1.f08
2         gfortran -o $@ $<
```

tells **make** that it can build the executable **example1** using the command

```
1 $ gfortran -o example1 example1.f08
```

provided the file **example1.f08** exists. Notice that **make** recognises `$@` and `$<` as abbreviations for the `TARGET` and the first of the `PREREQUISTITES`, respectively. The **-o** option is used if you want to choose a name for the executable different from **a.out**.

Typing the command

```
1 $ make
```

causes **make** for look for a file called **makefile** in the current directory, and to attempt to build the first target it finds in that file. Also, **make** echos all of the commands it uses so you can see exactly what it does.

Example 2 — Fortran

Compare **example1.f08** and **example2.f08**. These programs do the same thing, but differ in the way they access the subroutine **quadform**. In the first case, all of the code is in the same file, but in the second the code for the subroutine resides in a separate module **mystuff.f08**.

The advantage of the second approach is that we can easily use **quadform** in several different programs, and can take advantage of separate compilation of source files.

Typing the command

```
$ make example2
```

causes **make** to execute

```
gfortran -c mystuff.f08
gfortran -c example2.f08
gfortran -o example2 example2.o mystuff.o
```

In the first of these three commands, the **-c** option causes **gfortran** to create a file **mystuff.o** instead of trying to create an executable. Doing

```
1 $ file mystuff.o
```

reveals that this file is an "ELF 64-bit LSB relocatable", commonly known as an object code file. After separately compiling the two source files, the third command creates the executable **example2** which we can run by typing

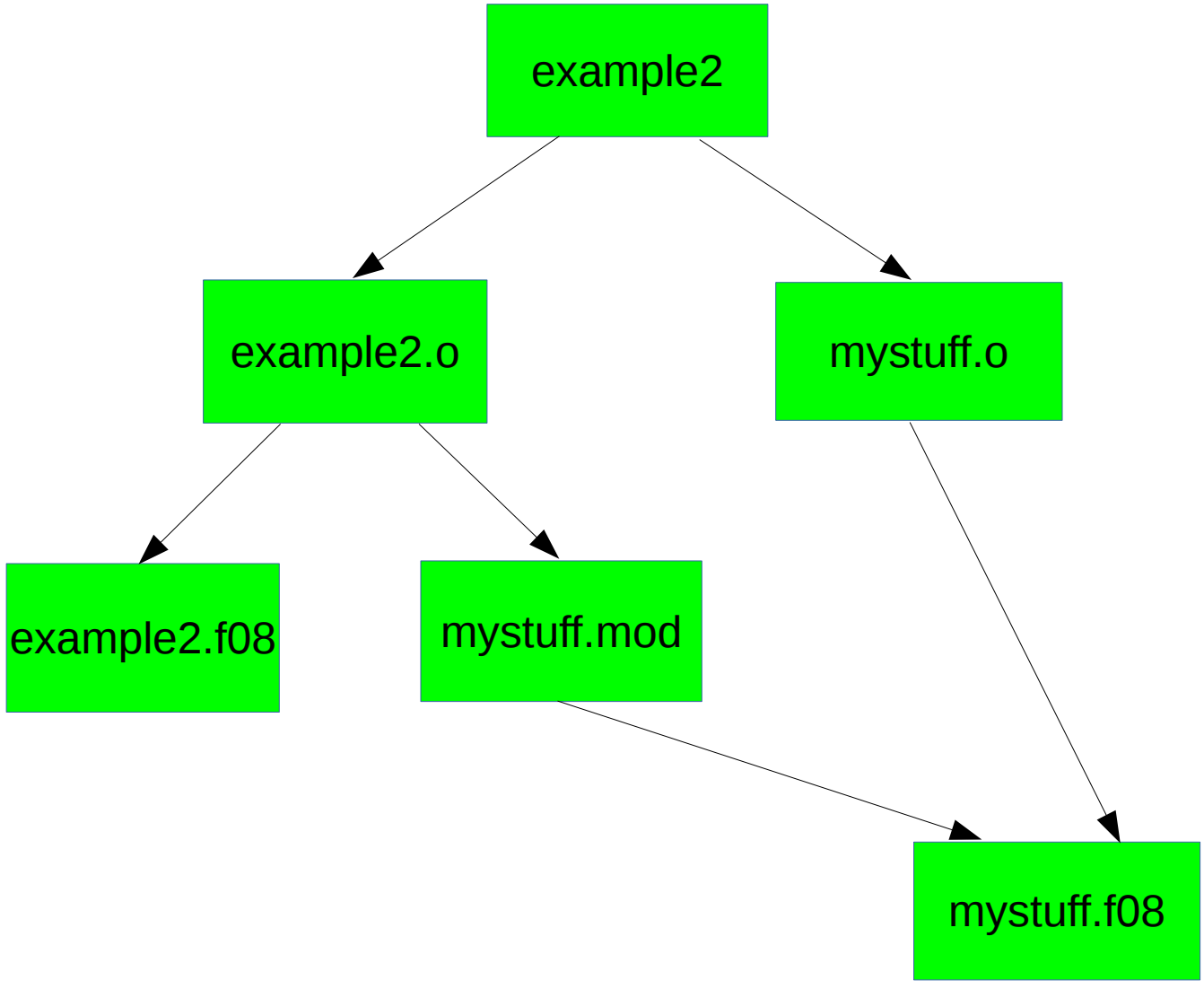
```
1 $ ./example2
```

You might have noticed that the first command actually created two files: **mystuff.o** and **mystuff.mod**. The latter tells the compiler the list of argument types for **quadform**, and is needed when **example2.f08** is compiled.

The three rules

```
1 example2: example2.o mystuff.o
2         gfortran -o $@ $^
3
4 example2.o: example2.f08 mystuff.mod
5         gfortran -c $<
6
7 mystuff.o mystuff.mod: mystuff.f08
8         gfortran -c $<
```

tell make how to contruct a complete dependency graph for the executable file **example2**. In this way, **make** knows that it must first compile **mystuff.f08**, then **example2.f08**, and finally link the two to create **example2**.


example2
example2.o
mystuff.o
example2.f08
mystuff.mod
mystuff.f08

Typing a second time

```
$ make example2
```

produce

```
make: `example2' is up to date.
```

If a target already exists, **make** compares its modification timestamp with those of the prerequisites. If no prerequisite has changed since the target was built, them **make** does nothing.

Exercise: explain what happens when you type

```
$ touch example2.f08
$ make example2
```

Assembly language

Strictly speaking, the compiler proper translates source code into assembly language. A separate program, an assembler, then translates the assembly language into object code.

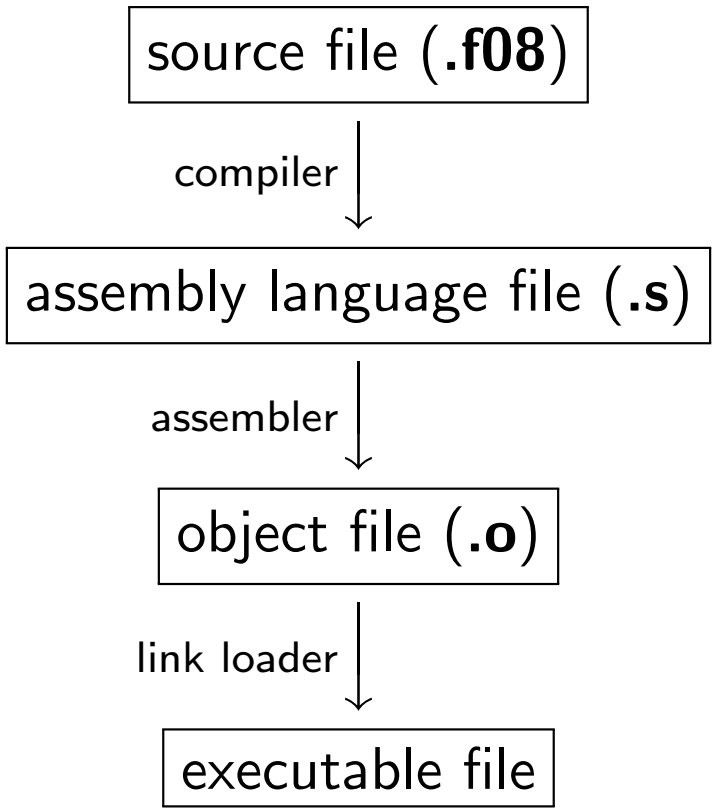
Use the **-S** option as follows

```
1 $ gfortran -S mystuff.f08
```

to tell **gfortran** to create a file **mystuff.s** containing the assembly language version of **mystuff.f08**. Essentially, by looking in the plain text file **mystuff.s** you can see the CPU instructions coded in the binary file **mystuff.o**.

Also, it is not really the compiler that creates the executable out of the object file(s). That task is performed by a separate program called the link loader.

Thus, a more complete picture is as shown below.



Example 3 — Julia

The Julia source files **example3.jl** and **MyStuff.jl** carry out the same computation as the Fortran source files **example2.f08** and **mystuff.f08** (but with a different random number generator). You must first include `MyStuff.jl` and then `example3.jl`.

The **code_native** function shows you the assembly language instructions that Julia generates for a given function:

```
code_native(quadform,
            (Matrix{Float64}, Vector{Float64}))
```

Alternatively, you can write to a file:

```
open("quadform_julia.s", "w") as output
    code_native(output, quadform,
                (Matrix{Float64}, Vector{Float64}))
end
```

Example 4 — C

Our final example carries out the quadratic form computation using C. The **makefile** rules are

```
example4: example4.o quadform.o
	gcc -o $@ $^ -lm

example4.o: example4.c quadform.h
	gcc -c $

quadform.o: quadform.c
	gcc -c $<
```

The file **quadform.c** defines the **quadform** function, and corresponds to the Fortran file **mystuff.f08** or the Julia file **MyStuff.jl**. C does not have a module construct so we can just put the function in a file by itself. The header file **quadform.h** plays the role of **mystuff.mod** in Fortran.