A *FORTRAN* Primer

David A. Clarke

Saint Mary's University, Halifax NS, Canada

david.clarke@smu.ca

January 2002; revised 12/05, 1/11

Contents

Preface

These notes were prepared for use in my junior computational physics course for students who have never been exposed to any serious computing. When IBM made available its first version of *FORTRAN* in the late 1950s, it was the only real alternative to writing code in assembly language. Since then, even though many other "high-level languages" have been introduced all with their aficionados and detractors, *FORTRAN* remains one of the easiest and most intuitive computing languages to learn, resembling more than any other "abbreviated English". My contention is: if you know any other computing language and can speak some English, learning *FORTRAN* is less than a weekend's work. Even if the student has never done any computing before, learning *FORTRAN* is probably the fastest way to start computing. I am a *FORTRAN* programmer (*FORTRAN*77 in particular), and that isn't going to change. I tell my students that they can use any language they like for my course, but if they want me to help them with their coding, it'll have to be in *FORTRAN*! This is usually a good motivator for the student to learn the language, and I have found these notes to be helpful toward that end.

These pages represent about 80% of what I know about *FORTRAN*, and about 99% of what I use regularly. For everything else, a reader experienced with these notes will be able to look up the rest. Inasmuch as these notes may help others, the reader is free to use, distribute, and modify them as needed so long as they remain in the public domain and are passed on to others free of charge.

David Clarke
Saint Mary's University
January, 2011; revised 2016

Primers by David Clarke:

1. A *FORTRAN* Primer
2. A *UNIX* Primer
3. A DBX (debugger) Primer
4. A Primer on Tensor Calculus
5. A Primer on Magnetohydrodynamics
6. A Primer on *ZEUS-3D*

I also give a link to David R. Wilkins' excellent primer Getting Started with LaTeX, in which I have added a few sections on adding figures, colour, and HTML links.

A *FORTRAN* Primer

1 Introduction

FORTRAN is a FORmula TRANslator which translates formulæ written in the human language of mathematics to the machine language of binary code. *FORTRAN* allows the programmer to type in expressions at the keyboard reminiscent to how they appear on paper and, through the steps of pre-compiling, compiling, and linking, a binary code executable by the machine is generated.

FORTRAN was designed for scientific computing. Other languages like C, C^{++}, Basic, Cobol, PL1, *etc.* all have their proponents, the most militant of which will waste no time in telling you that as a computer language, *FORTRAN* is on its way out. Yet, at the time of this writing, 90% of all flops[1] "flopped" by scientific applications worldwide are "flopped" by code generated by a *FORTRAN* compiler[2]. Evidently, as Mark Twain might have put it: *The report of FORTRAN's demise has been greatly exaggerated!*[3].

In all likelihood, *FORTRAN* is the only scientific computer language you will ever *need* to learn. You may want to pick up C^{++} at some time, but it won't be necessary if you know *FORTRAN* thoroughly. The biggest complaint about *FORTRAN* by its detractors is that it leads to unstructured, undecipherable "spaghetti code" (*e.g.*, the all-dreaded and overly maligned `GOTO` statements). Untrue. *FORTRAN allows* the programmer to write "spaghetti code", but also allows highly structured coding and whether a program is written as "spaghetti code" or not is a function of the programmer's laziness, not of any inherent shortfalls in the language.

IBM released its first *FORTRAN* compiler in 1957, after which numerous versions including *FORTRAN* II, III, and IV were released all touting various improvements over the previous. However, as use spread and various other vendors produced their own version of the language, standardisation was lost. Thus, in 1967, the first version of the language to be endorsed by the *American Standards Association* (now ANSI) was released as *FORTRAN*66, the first high-level language to enjoy such wide-spread use in such a standardised form. By today's standards, *FORTRAN*66 was limited and, as demand for the language grew, new and easier-to-use features were designed and installed to create the next version, *FORTRAN*77 released in 1978. This was the first comprehensive version of the language, allowing for virtually all functions a scientific programmer could need to perform any computer task imaginable.

As "structured programming" became the buzz and demand for more abstract programming grew, Fortran90 was released (which, among other things, changed the naming

[1] Flops is "computerese" for FLoating point OPerationS, one of which is required for each add or subtract, four for each multiply, 10 or so for each divide, *etc.*

[2] To those C users who may doubt this statistic, look carefully at your C compiler. Many C compilers, especially in supercomputing environments, will actually translate some or all of your C to *FORTRAN*, then use a *FORTRAN* compiler which is better suited for optimisation!

[3] Actually, what Mark Twain wrote was *The report of my death was an exaggeration*, but the words *demise* and *greatly* are commonly added, as though Mark Twain's writings weren't colourful enough!

convention from all-caps to capitalised), but not until 1992. This long-awaited version underwent many unexpected revisions as the international committee charged with its content haggled over the details. In the 15 years between *FORTRAN*77 and Fortran90 and because of impatience among many programmers, many "non-standard" features were built into various versions of *FORTRAN*77 by various vendors. Thus, IBM, DEC, Cray, Sun, *etc.*, all offered *FORTRAN* compilers that contained standard *FORTRAN*77 plus their own special features upon which many programmers came to rely. However, standardisation was sacrificed and, as *FORTRAN* became less portable, some started to abandon the language.

Fortran90 was designed to include most of the specialised features invented by some of the largest vendors of *FORTRAN*77 as well as all the features of *FORTRAN*77 itself. Thus, programmers who used an extended version of *FORTRAN*77 suddenly found that their code was again portable, at least with very minor changes to conform to the new standards, and the use of the language began to grow again.

Three years after Fortran90 was released, Fortran95 came out which, for the most part, is considered a minor revision of Fortran90. Since then, Fortran2003 (a major revision) and Fortran2008 (another minor revision) have been released, with Fortran2015 anticipated some time in 2018.

Still, enjoying as it did a virtual monopoly for about a decade as the only standarised computer language available, *FORTRAN*77 built up a huge legacy, and many many coding projects were developed using it and continue to use it to this day. If your criterion for learning a computing language is to minimise the ratio of what has to be learned to what can be done, *FORTRAN*77 remains unsurpassed.

In this primer, a subset of standard *FORTRAN*77 is described. Any Fortran90 compiler can compile any program written in standard *FORTRAN*77 (*e.g.*, without any of the vendor-specific extensions used), and if your code is written with only the statements found in this primer, it can be compiled by any Fortran95 compiler as well. This primer assumes the reader has a rudimentary knowledge of *UNIX* and, in a *UNIX* environment, knows how to create files, directories, move files around, edit text files, and submit commands. No detailed knowledge of *UNIX* script language is assumed, though the more the reader is familiar with *UNIX*, the better.

The next section lists the typical steps required to design, develop, and execute a computer program (written in *FORTRAN* or otherwise), without reference to details of the computer language. Section III is the most important section in this primer, and should be read and understood thoroughly. It gives a sample *FORTRAN* program that the reader can use as a template for their first programs, and includes a careful description of the code line by line. Attention is paid to what the *FORTRAN* statements do, and to programming style. Finally, the last two sections, meant as a reference, give more details on the syntax of the language. Section IV discusses variables, variable types, arrays, *etc.*, while Section V gives descriptions of the most commonly used *FORTRAN* statements. These discussions are by no means exhaustive, and the more experienced programmer will want to consult one of the many comprehensive *FORTRAN* texts, vendor manuals, and websites available.

2 Steps to Computer Programming

In order to execute a computer program (code), one must:

1. Develop the algorithm (*e.g.*, with pen and paper);
2. Write the code (using your favourite text editor in your favourite computer language—*FORTRAN*!);
3. Precompile the code (optional);
4. Compile the code (`gfortran -c myprob.f`);
5. Link the code (`gfortran myprob.o mylib.a`);
6. Execute the code (`a.out`),

where `myprob.f` is the code being compiled, `mylib.a` is an external *library* (don't worry if you've no idea what that means), and `gfortran` is the "`GNU`" *FORTRAN* compiler available on virtually all our machines. It is a public domain compiler[4] which, with the `-c` "flag", generates what is called "object code", unreadable by humans and unexecutable by computers. It is an intermediate stage to which other object files can be linked (*e.g.*, mathematical or graphical libraries). The "link" step also uses `gfortran`. If there are no libraries to be linked, steps 4 and 5 can be combined into one step, namely:

```
gfortran myprob.f
```

in which case no object code is generated, only the executable.

FORTRAN files should end with the suffix '`.f`' or '`.f90`'. The corresponding object code, if any, will have the same name, but with suffix `.o`. Thus, the object code for `myprob.f` will be in a file called `myprob.o`. As given above, the link steps generate an "executable" file called `a.out`, which is the default name (*i.e.*, that given if none is specified). By typing `a.out` at your cursor, your program will execute. If you wish to call your executable something else, you can either rename the file once created (*e.g.*, `mv a.out myprob.x`), or use the `-o` flag in the `gfortran` command (*e.g.*, `gfortran -o myfile.x myfile.f`). If the compile and link steps are split as in the list above, specify the `-o` flag in the link step.

Over the course of the semester, you will find your programs becoming more and more complicated and you will likely find most of your time consumed by *debugging*, the process of eliminating errors in syntax (misuse of the *FORTRAN* language), logic (unintentional "zero-divides" or square roots of negative numbers), implementation (code does not match algorithm), and design (problems with the algorithm itself). Only the syntax errors are flagged by the compiler; everything else you will have to root out yourself. Most people start by inserting "write statements" at strategic places in their program, hoping the output will reveal the problem. While this strategy is fine for simple programs, a program need not get too complicated before the use of a proper *debugger* is essential.

I refer you to my DBX Primer, in which I introduce and promote the use of a debugger, and highlight one particular debugger, namely *DBX*. For here, suffice it to say that one can

[4] Other `GNU` compilers include `g77`, `g90`, `g95`, `gcc`, *etc.*, and can be downloaded from many sites worldwide.

run a program "through a debugger" (which itself is a program) provided your program has be properly prepared. For virtually all debuggers, this means the `-g` compiler flag is set as your code is compiled. Specifically, during the development phase of any code and depending on which debugger you intend to use, on `venus` you should compile your program as follows:

debugger	compile statement
Sun's *DBX*	`f90 -g -C -ftrap=common` *myprob.f*
GNU's `gdb`	`gfortran -g -fbounds-check -ffpe-trap=i,z,o` *myprob.f* `g95 -g -fbounds-check -ftrace=full` *myprob.f*
Intel's `idb`	`ifort -g -debug-parameters -check -fpe1 -traceback` *myprob.f*

Note that `f90` is Sun's compiler, and must be used with *DBX*. Likewise, Gnu's compilers `gfortran` and `g95` may be used with `gdb` and Intel's compiler `ifort` is used with `idb`. The compiler flags other than `-g` check to make sure arrays stay in bounds (*i.e.*, that you don't try to access `arr(11)` when `arr` is declared with only ten elements), overflows, divides by zero, and invalid operations (*e.g.*, $\sqrt{x < 0}$). These checks take up computer time—for large programs, a significant amount—and should be used only during code development.

Once you are certain the bugs are out, recompile your code replacing each set of compiler flags with `-O2` (that's the letter `O` before the '`2`', not zero). Thus, and for example,

`gfortran -O2` *myprob.f.*

This is an "optimisation" flag that will make your program run about as quickly as safe rearrangement of your code by the compiler can achieve.

Even with your program working perfectly, there remain the issues of optimisation, readability, and generality:

- Optimisation: modifying your code without changing the algorithm to take advantage of compiler and hardware features to make best use of your machine (speed and memory);
- Readability: writing your code so that others may read, understand, and use what you have created;
- Generality: designing your code to be as flexible and general as possible.

Indeed, developing good software is as much an art as anything else, and your programming skills will be honed more by doing, rather than by pouring over programming texts. This primer is therefore designed to get you started with the least amount of details possible. As such, debugging and optimisation techniques are left beyond the scope of this primer. Instead, emphasis is placed on developing general-use *FORTRAN* code, a consistent and readable programming style, and ways of avoiding some of the more common traps often encountered by beginning programmers.

3 Sample Algorithm and *FORTRAN* Program

The mathematical problem of determining the number of prime numbers less than a given integer was known to the Greeks and, to this day, there is no solution other than by brute force. This section illustrates a computer "solution" to this problem. It is not general because one cannot ask for an arbitrarily high integer; the machine can only handle integers less than roughly 2 billion; anything bigger requires more than 31 bits (the 32nd bit in a four-byte integer is reserved for the sign).

However, this primer is not here to make history, but to illustrate algorithm design and its *FORTRAN* implementation for a "simple" (easy to pose, if not solve) problem.

3.1 Designing an Algorithm

Algorithm design can take on many forms. In a Computer Science class, your instructor may take great pains to convey to you what key words you should use, or whether a certain step should be encircled by a diamond, oval, rectangle, rounded rectangle, *etc.* We do none of that here. An algorithm starts off as an idea or a problem you wish to solve, and then consists of various stages in which the method of solution is fleshed out. At some point, there is enough meat on the bones that you can use the algorithm you have developed to generate code.

It is probably a good idea not to write your algorithm in the language of computer code (*e.g.*, *FORTRAN*), but instead in plain English. Invariably, though, as the algorithm develops, most people will start to pepper the English with *FORTRAN*- or C-type statements, mostly because the computer language is usually so much briefer than grammatically correct English!

In this section, I show you how I went about designing my algorithm to find out how many prime numbers there are below a certain integer. After this, I give you two versions of the associated *FORTRAN* code, both fully debugged (*i.e.*, they execute properly), but only one I would be happy to share with a colleague!

Step 1: Statement of the problem

Find all prime numbers less than a given integer.

Report each prime found.

Report the total number of prime numbers found.

Step 2: First refinement

For a given integer, n, one need only examine divisors greater than 1 and less than the square root of n (think about it!).

Check to see if each divisor divides evenly into n.

As soon as one divisor is found, n is not prime. Continue on to the next candidate.

If no divisors are found, n is prime. Report each prime found to both the screen and a disc file.

Step 3: Second refinement

Open disc file.

Ask user for maximum integer, $n_{\max}$.

Echo input integer to both screen and discfile.

For each integer n less than $n_{\max}$, do:

Find greatest integer less than $\sqrt{n_{\max}}$; call this $i_{\max}$.

For each i, $2 \leq i \leq i_{\max}$, do:

If i goes evenly into n, n is not a prime. Exit loop, and go onto $n + 1$.

i goes evenly into n if n/i truncated times i equals n.

If there are no even divisors of n, n is prime. List n as a prime integer, add one to the counter, and go onto $n + 1$.

Close disc file.

At this point, the algorithm is sufficiently refined to generate code, which starts on the next page.

You might wonder how you would go about solving this problem for numbers of truly arbitrary size; that is, how the algorithm presented above might be modified so that it is not limited by the internal representation of integers. This is not a trivial problem!

3.2 Converting the Algorithm to *FORTRAN* with good Programming Style

My *FORTRAN*77 implementation of this algorithm is written in my own personal style that I have developed over the past 25 years, and is full of comments, headers, vertical alignment, indentation of `do`-loops and `if-then-else-endif` structures, sensibly chosen variable names, *etc.*, all to make the code as readable as possible by a human. One of the nice things about *FORTRAN* is it largely reads like English; no crazy punctuation conventions or made-up words to throw you. After the code, I give some description of what each segment does, and after that I give another version of the code written without any regard to readability so that you can compare two extremes of coding style.

You do not need to adopt my programming style, although you may if you wish. However, for the purposes of assignments, no-style coding (*e.g.*, similar to the second version of the program below) will be given very low grades, regardless of whether it works or not! Note that strict adherence of the rules of *FORTRAN* requires *none* of these styles to be adopted; style is entirely self-imposed.

So why this emphasis on coding style? Two reasons. First, it makes finding errors (bugs) much easier. Second, the code is much more readable. By whom, you might ask? Primarily by you! Much of the coding I generated as a scientist in the last 20 years, I still use today, or at least some modified version of it. Try digging up some old coding you generated several years ago without comments, structure, or algorithm description, and see how long it takes

you to get it going again! In my case, there are well over 100,000 lines of my coding that are used by other scientists worldwide. These would have been of very little use to anyone had I not adopted a strict set of programming guidelines for myself when I started out. This is true of anybody who has made any impact on the scientific public domain. In this game, it is not at all a bad thing to be a little anally retentive!

As for the program listing which starts below, the first four columns are not part of the *FORTRAN* statement. I have added these columns to allow for line numbers so I can refer to various lines in my description following the code. Thus, the first character in each *FORTRAN* statement below actually starts in column 5 and so, if you are typing the program in at your terminal, the first column you should type for each line is column 5.

*FORTRAN*77 statements are never more than 72 columns wide. Any and all characters beyond the 72nd column are ignored. The first six columns are reserved for special "instruction" characters. For example, if the line is to be a comment, the first character must be a '`c`', '`C`', or a '`*`'. If the statement is a continuation of the previous line (because 72 characters wasn't enough), then the sixth character should be a non-zero digit (`1-9`), or the ampersand (`&`), or the period (`.`). Finally, if the statement is referred to by another statement (*e.g.*, by a `goto` statement), then the reference number (target) for that statement goes in the first five columns. There are numerous examples of this in the code below.

The actual *FORTRAN*77 statement must start in column 7. However, I always start in column 8, so that any continuation character in column 6 is not butt up against the leading character in my *FORTRAN* statement. Again, this rule is not *imposed* by *FORTRAN*77, but is *allowed*, and improves readability of the code by human eyes.

```
 1 c=====================================================================
 2 c
 3 c    \\\\\\\\\\         B E G I N   P R O G R A M         //////////
 4 c    //////////                 P R I M E                 \\\\\\\\\\
 5 c
 6 c=====================================================================
 7 c
 8       program prime
 9 c
10 c    abcd:prime <---------------------- lists all prime numbers .le. nmax
11 c                                                    september, 2001
12 c
13 c    written by: A Busy Code Developer
14 c    modified 1: by ADCD, October 2001; modularised program by placing
15 c                the coding that searches for divisors into a separate
16 c                subroutine.
17 c
18 c  PURPOSE:  This routine generates all prime numbers less than or equal
19 c  to the input integer, and reports these in a list printed both to the
20 c  screen and a datafile.
21 c
22 c  EXTERNALS:
23 c    DIVISOR
24 c
25 c---------------------------------------------------------------------
26 c
27       implicit      none
28 c
```

```
29        character*9   filename
30        logical       lflag
31        integer       nmax    , icount  , n
32        real          cpubeg  , cpuend  , cputot
33 c
34        external      divisor
35 c
36 c-----------------------------------------------------------------------
37 c
38 c      Start up CPU time counter.
39 c
40        call cpu_time ( cpubeg )
41 c
42 c      Open disc file.
43 c
44        filename = 'prime.txt'
45        open (10, file=filename, status='unknown')
46 c
47 c      Input parameter 'nmax', and write opening line to standard output
48 c  and 'filename'.
49 c
50        write( 6,2010)
51        read ( 5,   *) nmax
52        write( 6,2011) nmax, filename
53        write(10,2020) nmax
54 c
55 c      Loop over all integers .le. nmax, and report all the prime
56 c  numbers.
57 c
58        icount = 0
59        do 10 n=2,nmax
60          call divisor ( n, lflag )
61          if (lflag .eqv. .true.) then
62            icount = icount + 1
63            write(10,2030) n
64          endif
65 10     continue
66        write( 6,2040) icount, nmax
67        write(10,2040) icount, nmax
68 c
69        close ( 10 )
70 c
71 c      End CPU time counter, and report time used.
72 c
73        call cpu_time ( cpuend )
74        cputot = cpuend - cpubeg
75        write( 6,2050) cputot
76 c
77 c-----------------------------------------------------------------------
78 c---------------------- Write format statements ------------------------
79 c-----------------------------------------------------------------------
80 c
81 2010   format('PRIME   : Enter the maximum integer to be checked for '
82      1       ,'prime.')
83 2011   format('PRIME   : All prime numbers less than or equal to ',i4
84      1       ,' are listed in file ',/
85      2       ,'PRIME   : ''',a9,'''.')
86 2020   format('PRIME   : The following is a list of all prime numbers '
87      1       ,'.le. ',i8,'.',/)
```

```
 88 2030   format('PRIME   : ',i9)
 89 2040   format('PRIME   :',/
 90      1       ,'PRIME   : There are ',i8,' prime numbers less than or '
 91      2       ,'equal to ',i8,'.')
 92 2050   format('PRIME   :',/
 93      1       ,'PRIME   : Total cpu usage for this run is ',1pg12.5
 94      2       ,' seconds.')
 95 c
 96        stop
 97        end
 98 c
 99 c=====================================================================
100 c
101 c    \\\\\\\\\\          E N D   P R O G R A M          //////////
102 c    //////////                P R I M E                \\\\\\\\\\
103 c
104 c=====================================================================
105 c
106 c
107 c=====================================================================
108 c
109 c    \\\\\\\\\\     B E G I N   S U B R O U T I N E     //////////
110 c    //////////             D I V I S O R             \\\\\\\\\\
111 c
112 c=====================================================================
113 c
114        subroutine divisor ( number, lflag )
115 c
116 c    abcd:prime.divisor <---------- determines if input has any divisors
117 c                                                        october, 2001
118 c
119 c    written by: A Busy Code Developer
120 c    modified 1:
121 c
122 c  PURPOSE:  This routine searches the input number for any divisors
123 c  other than 1 and itself.
124 c
125 c  INPUT VARIABLES:
126 c    number       integer for which divisors are sought
127 c
128 c  OUTPUT VARIABLES:
129 c    lflag        = .false. if a divisor is found
130 c                 = .true.  if no divisors are found
131 c
132 c  EXTERNALS:  [none]
133 c
134 c---------------------------------------------------------------------
135 c
136        implicit      none
137 c
138        logical       lflag
139        integer       number  , imax    , i       , num
140        real          rnum    , sqrtnum , small   , qty
141 c
142        data          small    / 0.0001 /
143 c
144 c---------------------------------------------------------------------
145 c
146        lflag = .true.
```

```
147        if ( number .le. 3 ) return
148 c
149 c      The maximum divisor that needs to be checked is the greatest
150 c  integer less than or equal to the square root of the number (‘imax’).
151 c  The lowest divisor is 2.  Thus, check all divisors between 2 and
152 c  ‘imax’.  As soon as one divisor is found, set ‘lflag’ to .false.,
153 c  (i.e., "number" is not prime) and return to the calling program.
154 c
155        rnum    = real ( number )
156        sqrtnum = sqrt ( rnum )
157        imax    = int  ( sqrtnum + small )
158        do 10 i=2,imax
159          qty = rnum / real ( i )
160          num = i * int ( qty + small )
161          if ( num .eq. number ) then
162            lflag = .false.
163            return
164          endif
165 10     continue
166 c
167        return
168        end
169 c
170 c=====================================================================
171 c
172 c    \\\\\\\\\\        E N D   S U B R O U T I N E        //////////
173 c    //////////              D I V I S O R              \\\\\\\\\\
174 c
175 c=====================================================================
```

3.2.1 Delineating Program Modules

Lines 1–7 and 98–104 are how I delineate the beginning and end of the main program, while lines 107–113 and 170–175 delineate the subroutine. These are all “comment” statements, which can be placed anywhere in the code and, in *FORTRAN*77, must have the character ‘`c`’, ‘`C`’, or ‘`*`’ in the first column. There is nothing that makes reading a program more difficult than when you can’t tell when you’ve passed from one module to another. My style leaves little doubt!

FORTRAN programs are divided up into the “main” program, and various “subroutines” or “functions” that help modularise the program. Subroutines and functions are useful if a particular programming unit appears more than once in a program, if a program module needs to be used by more than one program, or if you wish to encapsulate logical units in separate modules. The latter use makes a program read more like the algorithm, and thus makes it more readable in general by a human. In this example, I have relegated the search for divisors to a separate subroutine for this very reason.

3.2.2 Program Statement and Selecting a Name for your Program

The program statement (line 8) is always the first non-comment statement in a program. The name of the program is entirely up to you, but keep it, and names of all subroutines and functions, to eight characters or less. *FORTRAN* is case insensitive, and I tend to use lower case for everything. You can mix lower and upper case as you like, the *FORTRAN* compiler

makes no distinction. Thus, program modules `PRIME` and `prime` are the same, variables `NMAX` and `nmax` are the same.

Usually, I name the file containing the program (main routine plus its subroutines and functions) the same as the main routine with a `.f` appended to it, though this is not necessary. Thus, the above program exists on my disc as a file named `prime.f`. If the compiler should find syntax errors (illegal *FORTRAN* usage) anywhere in your program, it will refer to the name of the program module (*e.g.*, `prime` or `divisor`), not the name of the file (*e.g.*, `prime.f`), when it tells you where the syntax errors are.

3.2.3 Program History

The comments in lines 9–24 provide a history of the module `prime`. This is not 'fluff'. If you ever use any of your software again after several months of non-usage, you will breathe a sigh of relief when you find some history information in the header. This will reassure you that you have found the right version of the code, that the module does what you think it does, *etc.*

3.2.4 Declaration Statements

I start and end my declaration sections (lines 25–36 and 134–144) with a single line clearly demarcating it from the header above and the body of the *FORTRAN*77 code below. The declarations are a critical part of the code, and must be easy to spot. There are several items of note here, all in the realm of coding style.

1. The `implicit none` statement (lines 27 and 136) forces you to declare all your variables as type character, integer, real, or logical. *FORTRAN* treats variables of these types differently, and so you cannot mix these types in the same statement without due care. Thus, if the variable `filename` is to contain the name of the file, it is a character string, and cannot be set, for example, to an arithmetical expression involving numbers. While this may seem obvious, what may not be so obvious is that integers and real numbers are treated differently too, and cannot be mixed either. Thus, if `nmax` is an integer, and `sqrtnmax` is real, you cannot set

   ```
   sqrtnmax = sqrt(nmax)
   ```

   because the intrinsic function[5] `sqrt` is expecting a real number, not an integer. Fortunately, there are intrinsic functions that will allow conversions between reals and integers, and so their arithmetic can be mixed when needed. There are examples of this in this program.

   Without `implicit none`, all variables beginning with the letter '`i`', '`j`', '`k`', '`l`', '`m`', and '`n`' are assumed to be integers, and all the rest are assumed to be real. Or, for example, you can use the `implicit` feature to tell the compiler that all variables starting with the letter '`c`' are character*8, and all variables starting with the letter '`l`' are logical.

[5] Intrinsic functions are operations that come with the compiler; that is you can use these without having to write a separate function or subroutine to do the operation. The most frequently-used intrinsic functions (such as square roots, trig functions, type conversion, *etc.*) are described in §5.9.

While this may be convenient, it is widely recognised as sloppy coding. Declaring `implicit none` forces you to declare each variable deliberately in the declarations list, and any variable found in the program not declared in the declaration list will generate a compiler error. Thus, you are safeguarded from misspelling a variable somewhere in your program which, had you used implicit declarations, would have been assumed to be a brand new variable, initialised accordingly (usually to zero or the null string), and computation would resume, usually incorrectly and without warning.

2. Enforcing vertical alignment means that I allow eight spaces for each variable name in the declaration lists, regardless of whether the variable has eight characters or not (hence the four spaces between `nmax` and the following comma in line 31). Frequently, you will have a program in which there are many similarly named variables, and having them all vertically aligned means that a quick scan can tell whether or not a variable has been forgotten, misspelled, *etc.*

3. External statements (line 34) list all subroutines and functions called by the module, and is used in the spirit that all variables be declared; it is simply good programming practise.

4. There is some restriction to the order in which statements must occur. The `program` statement is always first. Next, is the `implicit` statement, followed by the declarations of variables and parameters in any order (`character`, `integer`, `logical`, `real`, `real*8`, `parameter`). Next, `equivalence`, `common`, `data`, and `external` statements, in any order, are given, followed finally by the text of the program.

3.2.5 The Body of the Main Program

Lines 37–97 comprise the "body" of the "main" program. There are a number of things of note here as well:

1. Line 40 initialises the cpu[6] time counter so that lines 73–75 can report the cpu time required to execute the program. Note that the subroutine invoked to "glance at the cpu clock" (`cpu_time`) is *not* a subroutine intrinsic to *FORTRAN*, but a special extension of the *FORTRAN* compiler (in this case, `gfortran`). If you have a different compiler than `gfortran`, it will have an equivalent, but differently-named subroutine or function than `cpu_time`, and you will need to consult the relevant manuals to determine what it is, and how to use it.

2. Line 44 shows how a character variable is set. It is also possible to set a specific portion of the variable, leaving the rest as is. Thus, if later on I wished to change `filename` to `'prime.num'`, I could either set:

```
filename = 'prime.num'
```

which overwrites the entire value of the variable, including that portion which did not change, or I could reset only that portion I wish to change:

[6] Computer Processing Unit; the "smarts" of the computer.

```
filename(7:9) = 'num'
```

leaving the first 6 characters (`prime.`) as they were. Note that the variable `filename` was declared as a character*9 variable (line 29), and thus `filename` can contain no more than nine characters.

3. Again in line 44, note the single space on each side of the equals sign. This is not required in *FORTRAN*, but adds enormously to the readability of the program, particularly when there are a lot of equations. In addition, I typically surround `+ - * /` operators (plus, minus, multiply, divide) with single spaces to improve readability.

4. Line 45 shows how to open a file on disc. It makes a specific link between a disc file `filename` (set to `'prime.txt'` on line 44) and unit `10`, which is how the *FORTRAN* program will refer to this file until it is "closed" in line 69. So later, when I want to write something to the file `prime.txt`, the `write` statements in lines 53, 63, and 67 refer to the unit number rather than the file name. The `status` variable in the open statement indicates whether the file is `new`, (thus, a new file named `prime.txt` is created unless such a file already exists, in which case an error message is generated and execution aborted to protect you from unwittingly overwriting a file you may wish to keep), `old`, (thus an existing file named `prime.txt` is opened so the file may be modified, unless the file doesn't already exist, in which case an error message is generated and execution aborted), or `unknown` (in which case a file is created if none exists, or opened if one already exists, as in this example).

5. Lines 50–53 are read/write statements. For both, the first argument in parentheses refers to the unit. Since unit `10` is attached to the disc file `prime.txt`, the `write` statement to unit `10` (line 53) adds data to this file. For the `write` statement, unit `6` refers to "standard output" by default (normally the computer screen in front of you), and for the `read` statement in line 51, unit `5` refers to "standard input" by default (again, the computer screen in front of you, and in particular, the data you type at the cursor).

   What is read/written by a read/write statement are the values of the variables, if any, following the parentheses (*e.g.*, `nmax` in lines 51–53). *How* these data are written is determined by the "format statement" referenced by the second argument in the parentheses. In line 50, the second argument, 2010, refers to line 81 in which the first four columns contain the target number 2010. Since no variables are listed after the parentheses in line 50, format statement 2010 just provides the text (contained in single quotes) to be written to the specified unit, which in this case is 6, your computer screen. This particular write statement prompts you to enter the datum required by the program to do what it is designed to do, namely list off all the primes less than this input value.

   Line 51 is the read statement, and execution is paused until you enter the data required (*i.e.*, enough numbers of the right type to assign values to all variables listed after the parentheses, which in this case, is just `nmax`). With unit 5 (terminal screen) specified, you enter data right at the cursor, and hit `return` or `enter` when you are done. If the

read statement refers to a unit number attached to a disc file, that disc file would have had to be opened by a previous `open` statement, and would have to exist on disc with the data in the expected format.

When entering the value for `nmax` at the screen (unit 5), take care to enter an integer, since `nmax` is declared as an integer by line 31. Thus, if you want to enter one thousand, enter `1000`, not `1000.0` which by virtue of the '`.0`' is real. Commas are also not allowed. Don't enter `1,000` for example.

The second argument in the parentheses of the `read` statement on line 51 is an asterisk (`*`), which means you are using an "unformatted `read` statement". This instructs the computer to read in whatever you enter and deal with it in a "sensible" way. This means, of course, that the person who designed this portion of the compiler had to do a lot of programming (the compiler is a computer program too!) to allow for whatever you or I might have had a whim to do, and to generate sensible error messages if we make one of the many possible errors, each of which have to be specifically anticipated. So, for example, the compiler needs to be able to detect if you've entered in a character string when what is required is a real number, and it then has to generate an error message to tell you what you did wrong. What if you use a comma instead of a period for the decimal point? What if you enter too much data, or not enough? And so it goes.

Instead of the asterisk, you may specify the target for a `format` statement as you did for the `write` statement. However, if you do use what is called a "formatted `read` statement", the data you enter must appear *exactly* as specified by the `format` statement (see §5.8 for details).

Lines 52 and 53 are formatted `write` statements that write the contents of `nmax` according to the rules of `format` statements 2011 and 2020 (discussed below in point 6, and further in §5.7). The first writes to your screen (unit 6) and the second to the file `prime.txt` (unit 10). The latter will remain as the permanent copy of the results of executing the program long after the results flashed to your screen have scrolled out of sight.

Note the use of vertical alignment; extra spaces are inserted before the `*` in the `read` statement so that the close-parentheses will all align in the `read` and `write` statements. Obviously, this is hardly needed here. I do it as a matter of habit, without thinking so that when vertical alignment is needed (to make reading many lines of complicated formulæ a lot easier), it is always there.

6. Lines 58–65 comprise the main loop of the program. The variable `icount` will be the total number of prime numbers found. Line 58 initialises this counter to zero, then line 62 adds one to `icount` whenever another prime number is found. Note that line 62 makes no sense algebraically, if one thinks of `=` as a true 'equals sign'. If line 62 were a true algebraic statement, one could then subtract `icount` from each side, and arrive at the astounding result that $0 = 1$! Rather, in *FORTRAN*, `=` is an *assignment* symbol, and in a statement such as `icount = icount + 1`, we read "Replace the contents of the variable `icount` with `icount + 1`".

The `do`-loop structure is one of the most commonly used and powerful structures in *FORTRAN* (§5.1). Statement 60 states, in English, "Repeat all instructions from here to target `10` (line 65) as many times as required for the counter `n` to go from `2` to `nmax` when `n` is incremented by 1. If `n` is to be incremented by a number other than 1, the increment must be specified after `nmax`. Thus, `do 10 n=3,nmax,2` would mean "do for every `n` between 3 and `nmax` counting by twos". Thus, `n=3,5,7,...,nmax` or `nmax-1`, whichever is odd.

While it is not necessary to indent `do`-loops and `if-then-else-endif` structures, it vastly improves readability, so we do it. Again, I indent by habit. That way, all my software has the same readability. You may argue, "Well, for this one eensy-weensy program, I don't need to bother with indenting." But what if this program then makes the core of a larger one, that is then expanded again to make a still-larger one? When *do* you start to indent? And then do you have to go back to all the old coding, and retro-indent all that once you have decided the code has grown long enough? What a pain! Like most things in life, it usually takes no more time to do a job well than to do it poorly, so you may as well get into the "indenting, vertical alignment, extra spaces around arithmetical operators habit" right away!

Line 60 is a `call` to the subroutine `divisor` which appears after the main program (lines 107–175). Logically, you can think of line 60 as being replaced by the "guts" of the subroutine `divisor`. The use of subroutines (and functions) is strictly for making your program more useful and readable to the human, not for the computer. We use a subroutine (or a function) for the following reasons:

- If a module appears more than once in a program, the subroutine allows you to write the logic out only once and then `call` it as often as needed.
- If the module is used by several programs, it makes sense to put it into a separate subroutine in a separate file (library) so that you don't have to have several copies of the same module. This is particularly critical if you ever have to update the algorithm in the affected subroutine. Wouldn't it be a pain if you had to update the same subroutine in twelve different programs, then check and debug each to make sure you updated each right, only to realise months later that there were four more copies of the same subroutine in four other programs that you don't use very often and forgot about?
- To segregate logically separate modules or ideas of the program from each other. Often, a subroutine call represents a single statement or paragraph of a well-designed algorithm, and thus relegating each logical unit to a subroutine makes the main program read like the algorithm itself; a laudable goal for any good programmer.

Line 60 is typical of a `call` to a subroutine. The arguments listed parenthetically beside the subroutine name are variables passed between the calling routine (`prime`) and the subroutine (`divisor`). These variables could be data needed by the subroutine to do its task or data generated by the subroutine needed by the calling routine to continue its task. In this case, the variable `n` is passed to `divisor` (the quantity to be checked

for divisors), and the logical variable `lflag` is returned to the calling routine to indicate whether `n` has a divisor nor not.

There must be a one-to-one correspondence between the variable types (integer, real, *etc.*) in the calling statement (line 60) and the subroutine declaration statement (line 114). However, the actual variable names need not be the same. In this case, I have elected to call the input integer `number` in the subroutine `divisor` but `n` in the calling routine `prime`. Conversely, the logical variable `lflag` has the same name in both `prime` and `divisor`.

A logical variable (*e.g.*, `lflag`) is different from a real or integer variable. It occupies much less memory (usually only one byte rather than four or eight bytes for a real or integer) and can take on only one of two values, either `.true.` or `.false.` (*e.g.*, as set in line 146). Note the periods before and after the variable value; these are as much a part of the value as the digit 1 is part of the real value 1.25. Line 61 then tests the value of `lflag` as set by the subroutine `divisor`. In English, we would read "If `lflag` is true, then perform the statements up to and including the next `endif` statement. Otherwise, skip on past the next `endif` statement, and carry on from there." Note the use of `.eqv.` instead of `=`, again with the periods. This is interpreted by the compiler as a "logical equivalent" rather than an "assignment" as specified by '`=`'. The opposite of `.eqv.` is `.neqv.` Thus, the following three statements are synonymous:

```
if (lflag .eqv.  .true.) then
if (lflag .neqv. .false.) then
if (lflag) then
```

While the following three equivalent statements are opposite to those above:

```
if (lflag .eqv.  .false.) then
if (lflag .neqv. .true.) then
if (.not. lflag) then
```

In each set of three statements above, the last statement [`if (lflag) then` and `if (.not. lflag) then`] are carry-overs from *FORTRAN*66. In general, it is better practise to avoid these.

7. Lines 81–94 are the format statements. These may be placed *anywhere* in the program, including before the `write` statement that calls them. I prefer to place them all at the end of the program module rather than cluttering up the program logic by placing them directly after the `write` or `read` statement that first references them.

   `Format` statements contain explicit instructions on how the output is to be written, or how the input is to be read. The few statements here provide some examples. Each instruction, or "format code" within a `format` statement is separated by commas. For example, lines 86–87 comprise one statement that is too long for one line. Thus, column 6 in line 87 contains the digit 1, indicating this is a continuation from line 86. In this `format` statement, there are five separate format instructions. The first instruction, which fills most of line 86, is to write the text "`PRIME : The following is a list of all prime numbers `". The second instruction, which starts off continuation line

87, is to write the text "`.le.  `" immediately following the text written by the instruction on line 86. The third instruction is to write the integer variable `nmax` (line 53), as an eight-digit integer (`i8`). The fourth instruction is to write a period, and finally the fifth instruction is to add a blank line afterwards (`/`). The first four instructions together form a complete sentence, while the last instruction is effectively a "carriage return", generating an extra blank line to make the output more readable.

More complete syntax rules for `format` statements are given in §§5.7 and 5.8.

8. Line 96 is the `stop` statement, which flushes the program out of computer memory and releases the computer chip for the next task. Generally, if there is no `stop` statement in the program (in either the main routine or any of the subroutines), the compiler will issue an error message. However, it is possible to outsmart the compiler by making it impossible logically for execution to reach a `stop` statement. In this case, the computer will be left hanging after execution is complete, until some abort signal is sent to the cpu (*e.g.*, `control-c`).

   While the `stop` statement need not be the second last line of the main routine, it often is. The last line of a module, lines 97 and 168, is always an `end` statement which tells the compiler to cease reading statements as part of the current module.

3.2.6 The Subroutine

Lines 107–175 comprise the subroutine `divisor`. Many of these statements have been described or addressed already, but a few additional points of interest remain:

1. The `data` statement (line 142) is a useful way to set constants that are to remain the same throughout execution (though they could be changed by subsequent assignment statements if desired). In this case, the value for `small` is set to 0.0001. If there were other variables to set by the same data statement, one would add them after `small` separated by commas, and their corresponding values would be listed between the slashes, also separated by commas. For more than one value in the data statement, I often list them "stacked" (with the variable name appearing directly above its value), and/or leave enough blanks between each variable name so that each variable name takes up eight characters. Again, this is strictly for ease of reading by a human. Thus, suppose there were four variables instead of one, namely `tiny`, `small`, `big`, and `huge`. My data statement would then look like this:

```
      real          tiny    , small   , big     , huge
      data          tiny    , small   , big     , huge
     1             / 1.0e-32 , 0.0001  , 1000.0  , 1.0e32 /
```

   The vertical alignment allows for no mistaking what value is assigned to each variable. Note also how scientific notation is expressed for single-precision `real` variables (`1.0e-32` $= 1.0 \times 10^{-32}$). For double-precision (`real*8`) variables, one uses a `d` instead of the `e`. Thus, 1.0×10^{99} = `1.0d99`.

2. Line 147 acknowledges the fact that 2 and 3 are already prime, and so there is no need to go through the motions to see if these have any divisors. Thus, execution is returned straight away with the logical variable `lflag` set to `.true.` (*i.e.*, `number` is prime).

3. Still on line 147, it (as well as line 161) is an example of an "if test" (§5.2) done on numerical values. Unlike logical variables which can either be `.eqv.` or `.neqv.` (line 61), numerical and character values can be one of `.eq.`, `.ne.`, `.gt.`, `.lt.`, `.ge.`, and `.le.`. Note the use of `.eq.` instead of `=`, again with the periods. This is interpreted by the compiler as a "logical equal" rather than an "assignment" as specified by '`=`' and different again from the logical equivalent (`.eqv.`) reserved for logical variables.

4. Note the vertical alignment in lines 155–157. I want the square root of `number`, but `number` is an integer, and I cannot take the square root of an integer because the square root intrinsic function `sqrt` expects a real number (single or double precision) as an argument (§5.9.2). Thus, by using the intrinsic function `real` (§5.9.13), line 155 converts the value of the integer `number` to real, and assigns the real variable `rnum` that value. Thus, if `number=1000`, line 155 would assign `rnum=1000.000`, seven digits of precision for a four byte ("single precision") real number.

   Now, one thing one has to realise about a computer is it is not always as precise as you might expect. On some compilers, `1000` might get converted to `999.9999` or `1000.001`, rather than `1000.000`. These imprecisions are usually infrequent and innocuous, though on some rare occasions, they can lead to very wrong answers. If high precision is required, one can use so-called "double-precision" real numbers (declared as `real*8` instead of `real`), or one can make some minor alterations to the algorithm to account for such potential problems, as done in line 157.

   Line 157 converts the square root of `rnum` back to an integer, using the intrinsic function `int` (opposite operation of `real`). This conversion is done by truncation, not by rounding. Thus, `6.928` (the square root of `48`) gets truncated back to `6` when converted to an integer, not rounded up to `7`.

   Here is where the addition of `small` comes in in line 157. Because integer-real (in computerese, "fixed-float") conversions are not 100% accurate, it is quite possible that `number=100` gets converted to `rnum=100.0000`, and then `sqrtnum=9.999999` because of floating truncation errors peculiar to the compiler. Thus, because `int` truncates always, `int(sqrtnum)=9`, and not `10` as it should be. Since these truncation errors are always small (1 in the last digit of precision at the very most), adding just a very small number (*e.g.*, `small=0.0001`) to `sqrtnum` before converting it back to an integer means that the number `10.000099` gets truncated by `int` to `10` rather than `9.999999` getting truncated to `9`. Note that in all other situations, adding `small` to `sqrtnum` would not be enough to change the value of `imax`.

5. In lines 158–165, the main loop of the subroutine checks the input integer `number` for all possible divisors, with the counter of the do-loop being used as the candidate divisor. Lines 159 and 160 are the guts of the algorithm, which use the `real` and `int` intrinsic functions to determine if `i` divides evenly into `number`. Note the use of `small` again to avoid unfortunate truncation errors in the conversions.

   Line 161 performs the salient test. Should the test be true, then an exact divisor of `number` has been found, the logical variable `lflag` is set to `.false.` (*i.e.*, `number` is

not a prime), and execution is returned to the main program even before `do`-loop 10 is finished, since there is no need to look further for divisors.

On the other hand, should execution get to line 167, then `number` was found to have no divisors. Execution is returned to the calling routine with `lflag` left as `.true.` (yes, `number` is a prime) by virtue of line 146 and because line 162 was never reached to reset it to `.false.`.

3.3 Poor Programming Style

I will now give you the same program but without any indentation, comments, *etc.* It is logically identical to the program given at the top of this section (with no cpu-counter), and almost a factor of five shorter! But which would you rather have to face for the first time, and be asked to tell your boss what it does in ten minutes?

```
 1       program main
 2       open(10,file='main.txt',status='unknown')
 3       write(6,8)
 4 8     format('Enter nmax')
 5       read(5,*)nmax
 6       m=0
 7       do 9 n=1,nmax
 8       ii=1
 9       if(n.le.3)goto 9
10       r=real(n)
11       s=sqrt(r)
12       imax=int(s+.0001)
13       do 1 i=2,imax
14       q=r/real(i)
15       num=i*int(q+.0001)
16       if(num.eq.n)then
17       ii=0
18       goto 1
19       endif
20 1     continue
21       if(ii.eq.1)then
22       m=m+1
23       write(6,17)n
24       write(10,17)n
25 17    format(i9)
26       endif
27 9     continue
28       write(6,3)m
29       write(10,3)m
30 3     format(/,'m=',i4)
31       close(10)
32       stop
33       end
```

4 Data Types and Structures

4.1 Variables

Variables play the same role in *FORTRAN* as they do in algebraic equations; they represent quantities whose values are to be determined by the calculation. In general, most variables you use will be temporary, and represent intermediate steps in what may be a rather complex calculation. A few of the variables will contain the actual data needed, and you as the programmer will have to arrange to have them printed to the screen, disc, or presented in a graphical format.

Unlike variables in algebraic equations, the computer does not actually manipulate the symbols themselves[7], but instead stores their numerical values. The name of the variable you assign is actually mapped by the compiler to a specific area of memory which will be used to contain the current value of that variable as the calculation proceeds.

Also, unlike algebraic equations, variable names are restricted to letters (`a`–`z`, `A`–`Z`), digits (`0`–`9`), and the underscore (`_`), with the proviso that all variables shall start with a letter. No other punctuation (`!`, `@`, `#`, *etc.*) may be used. Thus, variable names can contain no Greek letters, no subscripts, no parentheses, *etc.* You may use up to eight characters for each variable name[8]. You may use `UPPER CASE` or `lower case` or both, knowing that *FORTRAN* compilers make no distinction; they are *case-insensitive*. Thus `NMAX`, `Nmax`, and `nmax` would all be treated as the same variable.

When selecting names for your variables, make sure each name describes what the variable is. Thus, if you have a program which deals with the ideal equation of state;

$$PV = NRT; \quad T = \frac{P}{R}\frac{V}{N} = \frac{P}{R\rho_m}, \tag{1}$$

where $\rho_m = N/V$ is the molar density (moles per unit volume), you might represent this in a *FORTRAN* code as:

```
temp = pressure / ( gasconst * mol_den )
```

which is preferable to:

```
T = p / ( R * rho )
```

since it is considered better programming style to avoid single-character variable names. Ideally, you would like your variable name to be a (nearly) unique combination of characters so that an electronic search in your program file for a particular variable picks up only the variable you are looking for, and not all occurrences of the letter `p`, for example, for which there may be hundreds of occurrences in a long, well-commented program. Still, this is much better than:

```
a=b/c/d
```

which is perfectly legal in *FORTRAN* and would still give you the right answer so long as `a` were temperature, `b` were pressure, `c` were the gas constant, and `d` were the molar density, but where nothing about the variable name is a mnemonic for what the variable is.

[7] There are so-called *symbolic manipulating* programs, such as *Mathematica* and *Maple*, but we're not talking about these here.

[8] This is a *FORTRAN*77 restriction. In fact, *FORTRAN*90 and *FORTRAN*95 allow for unlimited variable name length, but this is a practise I strongly discourage.

4.2 Variable types

Variables come in many types, the most common of which are `integer`, `character`, `logical`, `real`, and `real*8`. Other types include `complex`, `real*16`, `integer*2`, *etc.*

Variable type is declared using declaration type statements:

```
integer       ivar
character*n   cvar
logical       lvar
real          rvar
real*8        dvar
```

where *ivar* are the integer variables (separated by commas), *cvar* are the character variables, n is an integer greater than zero to indicate the number of characters in each of the variables listed in the character declaration statement, *lvar* are the logical variables, *rvar* are the real variables, and *dvar* are the double-precision real variables. Double precision variables may also be declared as:

```
double precision dvar
```

but I shall stick with the `real*8` syntax in this primer.

4.2.1 Integer Type

Integer variables consist of a sign (+ or −) and digits, nothing else. `-245` and `4609` are valid integers, `-245.0` and `4,609` are not (no decimal points and no commas allowed). Computer manuals refer to integers as *fixed* values, and integer arithmetic as *fixed* arithmetic. Fixed arithmetic is exact; no truncation errors are committed during fixed operations, which include addition, subtraction and multiplication. Division is, strictly speaking, *real* arithmetic, as there is the potential for the result not to be an integer. However, one can still divide integers, with the result being truncated to the largest integer less than or equal to the actual ratio. Thus, $3/2 = 1$ in fixed arithmetic.

An integer is stored with four bytes, or 32 bits. The bit is the fundamental unit of computer memory, and is how all computers store data. A bit is either "on" (1) or "off" (0). The first bit of an integer stores the sign of the number (+ or −), leaving 31 bits for the number itself. Since a bit can only have one of two values, the computer stores numbers in "binary code" and, with 31 bits, the highest number the computer can store is $2^{31} =$ 2,147,483,648.

Finally on integers, if you really had to store an integer much bigger than two billion, you could declare a variable to be `integer*8`, reserving eight bytes for that integer, thus allowing you to access integers to almost *ten pentillion* (10^{19}), but even this is finite! Note that `integer*8` and `integer*4` variables are different types, as different as `character` and `logical`.

4.2.2 Character Type

Character variables contain characters only, no numerical values. They may contain the text of the title you wish to put on a plot, or text to be printed to the computer screen or a line from a text file.

Each character in a character variable takes one byte, or eight bits thus allowing for as many as $2^8 = 256$ characters. Indeed, the standard "ASCII character set" contains 256 characters, including all lower and upper case Roman letters (52), all digits (10), all punctuation on your keyboard (32), and the blank (1). The remaining 161 characters consist of characters that are not on the standard `QWERTY` keyboard, including many accented characters used world-wide, other characters used in other alphabets, as well as some characters not found in any alphabet. Consult any comprehensive *FORTRAN* manual if you need to access the full ASCII set of characters.

Because each character takes one byte of storage, storing the number 2000000000 (two billion) as a character variable would require ten bytes (one for each character). Clearly, the integer type, requiring just four bytes, is a more efficient way of storing numbers than character type.

4.2.3 Logical Type

Logical variables are one byte long, and can have only one of two values: `.true.` and `.false.`. Note that the periods are a necessary part of the value, as necessary as the `1` is in the real value `1.25`. These values are equivalent to "on" (1) and "off" (0), and in principle should only need one bit, not one byte. However, most computer architecture is such that memory access has one byte resolution, and some hardwares such as some Cray supercomputers can only access its memory with eight-byte resolution. Thus, logical variables consist largely of "unused memory". Still, if the sole purpose of a variable is to act as a toggle, logical type is the most efficient way to go.

4.2.4 Real Type (Single and Double Precision)

Real variables are also referred to as *floating* variables, and real arithmetic as *floating* arithmetic. Like an integer, a real variable uses four bytes of memory, but unlike an integer, a real number is not exact. `1.000000` is the value of one to seven significant digits. What the eighth digit is is completely unknown (there is only a one in ten chance of it being a zero), and thus, the real value `1.000000` is not *exactly* one, while the integer `1` is.

The proper syntax of the real number 1.538702×10^{-17} is `1.538702e-17`, where the exponent portion may be omitted if it is zero. In general, the first bit of a 32-bit real number is reserved for the sign, the next 25 bits for the mantissa (allowing for seven or eight significant figures), the next bit for the sign of the exponent, and the last 5 bits for the exponent itself[9] (giving a maximum exponent of $2^5 = 32$). Thus, the greatest number that can be stored as a real number is 3.3554432×10^{32}, or `3.3554432e32` ($2^{25} = 33554432$). Anything larger may generate "overflow" error messages and cause the program to stop, or even worse, generate no messages and continue erroneously!

So, if 3.3554432×10^{32} is not big enough for your application (the mass of the sun in grams is already too large!), you will have to declare "double precision" (`real*8`) variables which use eight bytes, or 64 bits per number. In a double precision real value, the first bit is reserved for the sign, the next 52 bits for the mantissa (giving 15 digits of precision), the next bit for the exponent sign, and the last 10 bits for the exponent. The largest double precision

[9] Some compilers reserve 6 and even 7 bits for the exponent at the expense of the mantissa

number you can assign is $4.503599627370496 \times 10^{1024}$ before overflow problems start. If this is still not large enough, redesign your computer program!

These days, most processors use 64-bit technology, making it easier for these machines to process 64-bit numbers than 32-bit numbers. Thus, you may find that double-precision calculations actually run *faster* on 64-bit machines than single-precision calculations. There will come a time when 64-bit numbers are considered "single precision"[10], and 32-bit numbers "half-precision", just as once upon a time, 2-byte integers were considered "full integers" instead of "half-integers" as they are today. But for now, 32-bit real numbers are still considered single-precision (and therefore the default) and 64-bit real numbers as "double-precision".

When truly precise (32 digits of precision) or truly colossal (up to 10^{10^6}) numbers are required, there is "quad-precision" (`real*16`). However, I have never come across anyone who has had a legitimate need for these in practise.

Floating arithmetic includes all operations such as square roots (`sqrt`), trigonometry functions (`sin` and `cos`), exponentiation (`exp`), logarithms (`log`), *etc.* (§5.9). Floating point arithmetic is approximate, and round-off errors will frequently occur in the last significant digit. This can have annoying consequences when doing logical comparisons. For example, in the following snippet of coding:

```
1       x    = 1.0
2       xby4 = x / 4.0
3       if ( xby4 .eq. 0.25 ) then
4         instruction set 1
5       else
6         instruction set 2
7       endif
```

It is quite possible that on some compilers, `xby4` might actually be stored as `0.2499999`, and the logical test of line 3 would test negative, invoking the second set of instructions (line 6) rather than the intended first (line 4). As it turns out, it is unusual for an algorithm to rely on the exact equality of a real expression; normally one uses inequalities (`.lt.`, *etc.*). However, should an equality such as this be needed, the only fool-proof way of doing it would be to replace line 3 with something like:

```
3       if ( ( xby4 .lt. 0.2500003 ) .and. ( xby4 .gt. 0.2499997 ) ) then
```

where $\pm$ 3 in the seventh digit is just from my own personal experience. $\pm$ 2 is probably fine, while $\pm$ 1 may be pushing your luck. Note the introduction of the logical connector `.and.`. As you might expect, `.or.` is also available to create more complex logical tests.

4.3 Arrays and Parameters

It is often useful to declare variables analogous to vectors (1-D arrays), matrices (2-D arrays), and beyond (3-D, 4-D, *etc.*)

[10] Cray has treated 64-bit (8-byte) numbers as single precision since the 1980s, but at the time of this writing, they remain the only vendor to do so.

For example, suppose in a computer program that predicts the weather you wished to store the temperature at many uniformly spaced points over the surface of the Earth. If your resolution were to be 1 (angular) degree of latitude by 1 degree of longitude, then there would be 180 different latitudes and 360 different longitudes. You could declare $180 \times 360 = 64{,}800$ different temperature variables, one for each point on the Earth (but don't!), or you could declare a single 2-D temperature array:

```
real          temp    ( 180, 360)
```

This tells the compiler to reserve enough memory to store 64,800 different real numbers, and that in the program, the variable `temp` will be treated as an array of 180 columns by 360 rows. Each element of the array, corresponding to a temperature at some point on the Earth, can be addressed individually by specifying `temp(i,j)`, where `i` is an integer between 1 and 180, and `j` is an integer between 1 and 360.

Any variable type (integer, logical, *etc.*) can be declared as an array. Arrays are particularly useful when used inside a `do`-loop (§5.1). Thus, in programming equation (1), you might have:

```
 1        implicit      none
 2 c
 3        integer       idim    , jdim
 4        parameter     ( idim=180, jdim=360 )
 5 c
 6        integer       i       , j
 7        real          gasconst
 8 c
 9        real          pressure(idim,jdim), mol_den (idim,jdim)
10      1              , temp    (idim,jdim)
11 c
12        external      calcp   , calcd
13 c
14        data          gasconst / 8.3147 /
15 c
16 c-------------------------------------------------------------------------
17 c
18        call calcp ( pressure, idim, jdim )
19        call calcd ( mol_den , idim, jdim )
20        do 20 j=1,jdim
21          do 10 i=1,idim
22            temp(i,j) = pressure(i,j) / ( gasconst * mol_den(i,j) )
23 10       continue
24 20     continue
```

Given that variable arrays `pressure` and `mol_den` are computed by subroutines `calcp` and `calcd` respectively (lines 18 and 19), line 22 calculates the temperature for each `(i,j)`. Note that arrays can be passed as subroutine arguments as well; the only stipulation is that in `calcp`, the variable `pressure`, or whatever it is called there, must be of the same type (real) and have the same dimensions `(idim,jdim)` as in the calling routine. As in lines 18 and 19, the array dimensions may be passed to the subroutine in the calling list.

This example also illustrates the use of the `parameter` statement (line 4). Like variables, parameters may have any type (integer, logical, real, character which must be declared in an `implicit none` environment) and may be passed as subroutine arguments and used in the

right hand side of assignment statements just as constants are used. However, parameters differ from variables in that they may not be arrays, and once assigned by a parameter statement, they may not be re-assigned a value by a regular assignment statement in the program (*i.e.*, they may not appear on the left-hand side of an assignment statement). The nice thing about parameters is that integer parameters may be used to dimension arrays. Thus, should I want to redo this problem but at twice the resolution, I need only change the value of `idim` and `jdim` once in the parameter statement (line 4), and not everywhere they may appear in the declarations (lines 9 and 10).

4.4 Variable Assignments

In the body of the program, variables are assigned values using the "equals sign", and the syntax of the constants depends on the variable type. Examples of such assignments are included in the program "snippet" below.

```
 1        integer       ivar    , i       , iarray   (   5)
 2        character*12  filename
 3        logical       lvar
 4        real          rvar    , r_one   , rarray  (   5)
 5        real*8        dvar    , d_one   , darray  (   5)
 6 c
 7 c-----------------------------------------------------------------------
 8 c
 9        ivar           = 12
10        filename       = 'output.data     '
11        filename(8:11) = 'text'
12        lvar           = .true.
13        rvar           = 1.3807e-23
14        r_one          = 1.0
15        dvar           = 6.6261d-34
16        d_one          = 1.0d0
17        do 10 i=1,5
18          iarray(i)    = 2 * i
19          rarray(i)    = 2.0 * real ( i ) - 1.0
20          darray(i)    = 2.0d0 * dble ( i ) - 1.0d0
21 10     continue
```

Lines 1 through 5 declare the variables and lines 9–16 assign values to each of the variables. Note that double precision variables use the letter `d` (*e.g.*, line 15) rather than `e` (*e.g.*, line 13) to designate the exponent. Line 14 shows that when the exponent is zero, it is not necessary to include it as part of the expression for a single precision (`real`) value. However, as shown in line 16, the exponent must be included when assigning double precision (`real*8`) variables, even if it is zero, so that they are distinguishable from single precision (`real`) values. Note that `1.0` is a single precision value, while `1.0d0` is a double precision value. Thus, the statement `d_one = 1.0` when `d_one` is `real*8` is technically mismatched, and on some compilers may cause error messages or worse, unpredictable results elsewhere in the program.

Lines 18, 19, and 20 show simple integer, real, and real*8 arithmetic to assign the five elements in the integer array `iarray` to the even integers up to 10, and the five elements of the real array `rarray` and real*8 array `darray` to the real equivalents of the odd integers

up to 9. Note the use of the intrinsic functions `real` and `dble` to convert the integer `i` to real and real*8 respectively, to prevent mixed variable type arithmetic in the assignment statements. The intrinsic function `real` will also accept a real*8 argument and convert it to real, while the intrinsic function `dble` will accept a real argument and convert it to real*8. Finally, note that both constants in line 19 end in `.0` (so that they are real and thus avoid mixed variable type arithmetic, while the constants in line 20 end in `.0d0` for the same reason.

Many novice programmers will forget to include the conversions (*e.g.*, `real`) explicitly, or even to put the `.0` at the end of real numbers. For example, one might be tempted to write line 19 as:

```
19          rarray(i)    = 2 * i - 1
```

Indeed, you *may* be able to get away with this, as many compilers will perform the conversions automatically, but this is both careless and dangerous. The conversion rules may not be exactly as you expect, or the compiler may do something entirely stupid like interpret `rarray` as an integer from this point on, yielding unpredictable consequences later on in the computation. If you're doing floating arithmetic, then do floating arithmetic; anything else is sloppy. In line 19, this means using `2.0`, not `2`; `real(i)`, not `i`; and `1.0`, not `1`.

5 Common *FORTRAN* Statements

In this glossary, only the most commonly used *FORTRAN*77 statements are described. For each statement listed, a generic example of its usage in a *FORTRAN* context is given, followed by a description of the program element, as appropriate. In the generic example, key words that are to be used as shown are written in `typewriter font`. Items appearing in *italics* are intended to be substituted for whatever is needed at that place in the program. Items appearing between square brackets, `[ ]`, are optional items which, if left out, will invoke sensible defaults.

5.1 `do`

```
1         do n i=ibeg,iend,[iint]
2           FORTRAN statement(s)
3 n       continue
```

where n (the target) is an integer between 1 and 99999 which must appear immediately after the keyword `do` (separated by at least one blank), as well as in the first five columns of the last line of the `do`-loop, normally a `continue` statement. The `continue` statement is a "no-op" (performs no operation), and is used in *FORTRAN* primarily as targets.

i is an integer variable set by the internal counter of the `do`-loop, and may not be assigned a value by any statement or operation inside the `do`-loop itself. It may be *used* inside the `do`-loop (*i.e.*, it may appear on the right hand side of an assignment statement), but not on the left hand side where variables are assigned. Outside the `do`-loop, you may treat i as any other variable and, with `implicit none` set, must be declared an integer.

During the first pass of the `do`-loop, the value of i is *ibeg* and during the last pass, *iend*. At each pass, i is incremented by *iint* which, if left out, is assumed to be 1. For $iint > 0$, *iend* must be $>$ *ibeg* and for $iint < 0$, *iend* must be $<$ *ibeg*. Otherwise the compiler will generate an error message.

The target need not be a `continue` statement; instead, one could place n somewhere in the first five columns of the last *FORTRAN*77 *statement* in the loop. This line would then perform its intended function *plus* act as the target for the `do`-loop.

One may dispense with the need of a target statement altogether by replacing the `do-continue` structure with the `do-enddo` structure (non-standard *FORTRAN*77, but standard *FORTRAN*90). Thus, the following example is identical to the preceding example should your *FORTRAN*77 compiler support `do-enddo`:

```
1         do i=ibeg,iend,[iint]
2           FORTRAN statement(s)
3         enddo
```

Sticking with standard *FORTRAN*77, I shall use the "targeted `do` statements" from now on, and use `continue` statements for all targets.

`Do`-loops may be nested as needed, so long as the "target paths" don't cross. Thus,

```
1         do 20 j=jbeg,jend
2           do 10 i=ibeg,iend
3             FORTRAN statement(s)
```

```
4 10       continue
5 20     continue
```

is correct syntax, as is:

```
1        do 10 j=jbeg,jend
2          do 10 i=ibeg,iend
3            FORTRAN statement(s)
4 10     continue
```

where both loops end at the same statement. However,

```
1        do 20 j=jbeg,jend
2          do 10 i=ibeg,iend
3            FORTRAN statement(s)
4 20       continue
5 10     continue
```

is illegal, since the paths of the loops cross. Loop 10 begins inside loop 20, but ends outside, which makes no sense logically. Finally, both the following:

```
1        do 20 j=jbeg,jend
2          do 10 j=j1,j2
3            FORTRAN statement(s)
4 10       continue
5 20     continue
```

and,

```
1        do 20 j=jbeg,jend
2          do 10 i=ibeg,iend
3            j = i + 1
4            FORTRAN statement(s)
5 10       continue
6 20     continue
```

are illegal, since the value of `j` is reset inside loop 20 which uses `j` as its counter. However,

```
1        do 20 j=jbeg,jend
2          i = j + 1
3          do 10 i=ibeg,iend
4            FORTRAN statement(s)
5 10       continue
6 20     continue
```

is perfectly OK, since line 2 assigns a value to `i` *outside* loop 10.

5.2 `if`

```
1        if ( logical expression ) single executable statement
```

where the *single executable statement* is executed only if the *logical expression* is true. The *single executable statement* can be any legal *FORTRAN* statement that only requires one

line (possibly with one or more continuations) to execute. This includes an assignment statement, a `read` statement, a call to a subroutine, *etc.* It could not be a `do`-statement, since the do will require a target line (or an `enddo`), and other statements in between. Thus, `do`-loops require the use of the `if-then-else-endif` structures described next.

The *logical expression* consists of a left hand side, a right hand side, and a comparator. If the two values being compared are numbers (integer or real) or characters, the comparators can be `.eq.`, `.ne.`, `.lt.`, `.le.`, `.gt.`, or `.ge.`. If the two values are logical variables, the comparators can be `.eqv.` or `.neqv.` only. More complicated logical expressions can be built up from smaller ones by use of the logical connectors `.or.` and `.and.`. Parentheses should be used to group simpler logical expressions where there may be ambiguity. Thus, examples of legal logical expressions include:

```
1        if ( niib(j,k) .eq. 4 ) dv(ism1) = dv(ie)
2        if ( wctot .le. 0.0 ) wctot = amax1 ( cputot, tiny )
3        if ( dt .gt. 4.99*dtmin ) return
4        if ( filename(10:12) .eq. 'new' ) write ( 10, 2010 ) newline(i)
5        if ( ( iord .lt. 1 ) .or. ( iord .gt. 3 ) ) iord = 2
6        if ( ( ( x1fac .ne. 0.0d0 ) .and. ( nx1z .gt. 1 ) ) .or.
7     1      ( lflag .eqv. .true. ) ) call newx1
```

5.3 if-then-else-endif

```
1        if ( logical expression ) then
2          instruction set 1
3[       else
4          instruction set 2]
5        endif
```

The syntax for the *logical expression* is identical to that described in the `if` section.

Line 2 represents as many *FORTRAN* statements (executables, comments, no-ops, *etc.*) as needed, and represent logic that is to be executed provided the *logical expression* in line 1 is true.

Lines 3 and 4 are optional, and are omitted if there is nothing to do in the event the *logical expression* is false. Otherwise, line 4 represents as many *FORTRAN* statements as may be needed in the event that the *logical expression* in line 1 is false.

Line 5 is a no-op that closes off the `if-then-else-endif` structure.

5.4 go to; goto

```
1        go to n
2        FORTRAN statement(s)
3 n      continue
```

where n (the target) is an integer between 1 and 99999 which must appear immediately after the key word `to` (separated by at least one blank), as well as somewhere in the first five columns of the intended target line (*e.g.*, a `continue` statement). The `go to` (`goto` is equivalent) statement then redirects execution to the indicated target. A `go to` statement can redirect execution to a line *after* it (as in the example above), or *before* it (as in the

example below). However, `go to` statements cannot redirect execution outside the program module (*i.e.*, to another subroutine).

If a `go to` statement is performed *unconditionally*, then if execution is redirected after the `go to` statement, the coding in between the `go to` and its target will never be executed (as in the example above). On the other hand, if execution is redirected before the `go to` statement, the coding in between the `go to` and its target will be repeated over and over again without break, forming what is called an *infinite loop*. To avoid either of these scenarios, `go to` statements are normally used in conjunction with `if` statements, as in the following example:

```
1        i = 0
2 10     continue
3        FORTRAN statement(s)
4        i = i + 1
5        if ( i .le. imax ) go to 10
6 20     continue
```

Line 5 will cause the execution to return to line 2 (target `10`) so long as `i` remains less than or equal to `imax`. Once `i` exceeds `imax` (as it will eventually by virtue of line 4), execution skips on to line 6 and then on to whatever follows. Notice that without the `if`-test in line 5, this would be an infinite loop, with nothing in the logic to break the cycle.

Note also that the above example is logically equivalent to the following `do`-loop:

```
1        do 10 i=1,imax
2          FORTRAN statement(s)
3 10     continue
```

Indeed, most uses of the `go to` statement can be replaced with other, better structured *FORTRAN* such as `do`-loops and `if-then-else-endif` structures. Because `go to` statements can make the code difficult to read and the logic difficult to follow, they should be avoided where ever possible. Still, there are times when a `go to` statement is unavoidable.

For the more advanced programmer, there is a *computed* `go to` statement, which allows execution to be directed to one of many places depending on which of many values a variable may have. *Computed* `if` statements also exist, and the interested reader should consult a comprehensive *FORTRAN* manual for their descriptions. These structures can truly lead to "spaghetti code", and should be avoided in all but the most extenuating of circumstances.

5.5 open

```
1        open ( lunit, file='filename', status='filestat'[, err=target] )
```

The `open` statement opens a disc file and connects it to the logical unit *lunit* (an integer) which is specified as the first argument in the `open` parameter list. The name of the file *filename* is known only to the `open` statement; from this point on, the program will know this file only by its unit number *lunit*, and all `read`s, `write`s, `rewind`s, *etc.*, will be done to unit *lunit*. *filename* should be the complete name typed exactly as you would if you were to access the file from the directory in which the program was launched. Thus, if the name of the file is `output`, and it exists or is to be created in the same directory in which the program

is started, *filename* should be set to `output`. However, if the file `output` actually exists or is to be created in one directory up from where the program is launched, then *filename* should be set to `../output`. Thus, directory paths, as complicated as they need to be, may be included as part of *filename*.

The status of the file, *filestat*, can be one of `new`, `old`, or `unknown`, and one of these three words must be enclosed in quotes as indicated. Status `new` means no file of name *filename* should already exist. If one does, an error message is generated and if no *target* is specified for `err`, execution is aborted. Status `old` means the file *filename* should already exist on disc, and you wish to open it for modification. If no such file is found, an error code is generated, and again, if no *target* is specified for `err`, execution is aborted. Finally, status `unknown` means the file *filename* may not already exist, in which case a new one is created, or it may exist, in which case the old one is opened.

Finally, the parameter `err` allows you to redirect execution elsewhere in case an error message is generated. You may want to allow for the possibility that a file does not already exist without having to create a new one (as `unknown` would do). In this case, if you specify the target to some other line in the code (just as a `go to` or a `do`-loop would), then you could carry on execution without having to open up a new file should the one you tried to open not already exist. Note that the `err` parameter is optional; you can omit it altogether as many examples in this primer have done.

There are numerous other, less-used parameters associated with the `open` statement, and the interested reader is referred to any comprehensive *FORTRAN* manual for details.

5.6 close

```
1       close ( lunit )
```

The `close` statement closes the file attached to the logical unit *lunit* (an integer) by an earlier `open` statement. If you forget to close a file, the compiler closes it for you upon completion of the program. `close` statements are really only needed if you should want to try to re-open an existing file later on. Should it be still open, it can't be re-opened, and attempting to do so would generate an error message.

Like `open`, there are a variety of parameters you could set besides *lunit*, but these are rarely used.

5.7 write/format

```
1       write ( lunit, wformat ) [varlist]
```

The `write` statement writes text and/or the values of the variables listed in *varlist* to the device specified by the "logical unit" *lunit* (which could be the terminal screen, a disc file, a printer, even a character variable). How the text is written is specified by *wformat*, which can be an asterisk (`*`) meaning "unformatted" text, or an integer between 1 and 99999 which targets a `format` statement elsewhere in the program module. It is important to realise that the `write` statement converts everything to character strings before writing it to the output device. By writing the value of a real number, for example, you never actually see the real

variable itself, but instead a character representation of that real number. To examine the actual real number would mean you would be looking at bits (`0`'s and `1`'s).

If *lunit* is 6, output is directed to the computer terminal (often referred to as "standard output"). The statement:

```
1        write ( 6, * ) 'Hello!'
```

will write the exclamation `Hello!` (which must be enclosed by quotes in the `write` statement so that the compiler knows that `Hello!` is a character string and not a variable name) to the computer screen by virtue of the integer `6` appearing in the slot where the logical unit goes. The asterisk implies the `write` statement is "unformatted", which really means that some default formatting that depends on what is being output (in this case, the word `Hello!`) is invoked.

If *lunit* is an integer other than 5 or 6, the `write` statement will direct its output to the disc file to which the logical unit has been attached by a previous `open` statement.

```
1        open ( 10, file='output', status='unknown' )
2        FORTRAN statement(s)
3        write ( 10, * ) 'Hello!'
```

In this example, the exclamation `Hello!` is written to the next line of the opened disc file `output`, which was attached to logical unit 10 by virtue of the `open` statement (§5.5) in line 1.

Finally for *lunit*, if *lunit* is a declared character variable instead of an integer, the write statement will direct its output to the variable named. In this way, a real variable can be converted to a character variable, should such a conversion be required. The short program `testwr` below illustrates this feature.

```
 1        program testwr
 2 c
 3        character*10  cvar
 4        real          rvar
 5 c
 6 c  Boltzmann's constant...
 7 c
 8        rvar = 1.3807e-23
 9        write ( cvar, 2010 ) rvar
10        write (    6, 2020 ) rvar, cvar
11 c
12 2010   format ( 1pg10.4 )
13 2020   format ( 'rvar=', 1pg10.4, ', cvar=', a10)
14 c
15        stop
16        end
```

Line 9 writes the character representation of the real variable `rvar` to the character variable `cvar` according to the format specified in format statement `2010` (line 12). The format code `1pg10.4` means the number will be written with ten characters (including the sign, mantissa, decimal point, the character `e` to herald the exponent, the sign for the exponent, and two digits for the exponent) with four digits appearing to the right of the decimal point (*i.e.*, four decimal places). The `1p` portion of the format code indicates that one non-zero digit will appear to the left of the decimal place.

Line 10 then writes both variables to the screen so the reader can see that the conversion has been performed correctly. Since `cvar` is a character variable, it requires a different format code to write than the real variable `rvar`. Thus, in format statement `2020` (line 13), the real variable is written with the same format code (`1pg10.4`) as in format statement `2010`, whereas the character variable `cvar` is written with the format code `a10`. The `a` means ASCII text, the `10` means ten characters. As given, the screen output as determined by line 13 looks like:

```
rvar=1.3807E-23, cvar=1.3807E-23
```

If we dropped the `1p` from the format code in line 13 (but kept it in line 12), the output would change to:

```
rvar=0.1381E-22, cvar=1.3807E-23
```

Finally, if the format code `1pg9.4` were used in line 13 rather than `1pg10.4`, the output would look like:

```
rvar=*********, cvar=1.3807E-23
```

since 9 characters in total is not enough to display `1.3807E-23`. In fact, ten characters would not be enough if `rvar` were negative, since the negative sign would require an eleventh character. Thus, as a rule of thumb, **always specify 7 more characters in total than the number of decimal places you want**. That way, `1pg11.4` will always have enough characters to display the full real number with four decimal places. If you want five decimal places, specify `1pg12.5`, *etc.*

As we've seen, the `format` statement dictates how the output appears by use of various format codes enclosed by parenthesis after the keyword `format`. Format codes may be snippits of text to be written between the variables, or special codes which dictate how the variables listed in the *varlist* of the targeting `write` statement are to appear. All format codes listed in a `format` statement must be separated by commas. Extra spaces are optional. The following is a more comprehensive example of a `write` and `format` statement combination.

```
1        write ( 6, 2010 ) i, j, k, density(i,j,k), 'measured at (x,y,z) = '
2      1                 , eks(i), why(j), zed(k)
3 2010   format ( 'd(', i2, ',', i2, ',', i2, ') = ', 1pg12.5, /
4      1        , 10x, a22, 1p3g12.5)
```

If `i=5`, `j=12`, `k=110`, `den(i,j,k)=0.1025`, `eks(5)=0.5`, `why(12)=1.2`, and `zed(k)=11.0`, these lines of *FORTRAN*77 would write to your screen the output:

```
5 d( 5,12,**) =  1.02500e-01
6           measured at (x,y,z) =  0.50000     1.20000     11.0000
7 c===+====1====+====2====+====3====+====4====+====5====+====6====+====7==
```

Line 7 (a number line) has been added so you can count columns better. It is not part of the output.

Lines 1 and 3 illustrate that text can be specified in the *varlist* of the `write` statement (*e.g.*, `'measured at (x,y,z) = '`), or directly in the format statement (*e.g.*, `'d('`). If

given in the *varlist*, there must be a corresponding character format code in the format statement (in this case, `a22`) which indicates 22 characters (the number of characters in `'measured at (x,y,z) = '`, including all blanks) will be written.

In this example, the *varlist* consists of three integers, followed by one real, one character string, and three reals in the continuation line. Thus, the format codes in the format statement must be given in the same order, namely three integers codes (`i2` appearing three times), one real code (`1pg12.5`), one character code (`a22`), and finally three real codes (`1p3g12.5`, where the `3` appearing between the `p` and `g` indicates that this code is to be applied three times). In between each format code, you are free to insert whatever text and spacing codes you like. Thus, the format statement in line 3 will lead the first three integers with `d(`, then place commas between each integer, closing the three integers with `) =`, giving the very readable output `d( 5,12,**) = 1.02500e-01` in line 5.

Note that the format code `i2` was given for each of the three integers. Now, the first integer is `5`, requiring only one character. Since `i2` allows for two characters, a blank is inserted before the digit 5 in the output (line 5). If you wanted to have two *bone fide* digits rather than a blank and one digit, you could use the code `i2.2`, which forces two digits to be printed (the integer after the dot is the number of forced digits). This would then generate the output `d(05,12,**)` rather than `d( 5,12,**)`. The second integer is `12`, which uses up all two characters assigned to this field by the code `i2`. The third integer is `110`, which cannot be accommodated by two digits, thus generating `**` in the output. To correct this you would have to replace the third `i2` with `i3`.

The `1pg12.5` code has been explained above, as has the `a22`. The forward slash (`/`) forces a line break in the output. The code `10x` forces ten blank spaces.

Formatted I/O (Input/Output) is one of the trickier things to get used to in *FORTRAN*, and is one of the perennial complaints about the language. Often error messages resulting from incorrect formatting are cryptic at best and finding the cause of the error message can be frustrating. Taking some defensive measures such as adding a blank after each format code in the `format` statement (as has been done in lines 3 and 4) will help you to scan the format statement by eye for syntax errors. Correct whatever errors you find regardless of what you think the error message may be telling you, and the error message may well go away.

For example, which line is easier to read: lines 3 and 4 above, or the following:

```
8 2010   format ('d(',i2,',',i2,',',i2,') = ',1pg12.5,/,10x,a22,1p3g12.5)
```

Both give identical output, but line 8, with all blanks between format codes and character strings removed, is now a sea of commas and quotes, and it is hard to tell which commas separate format codes and which are meant to be part of the text in the output.

5.8 read/format

```
1        read ( lunit, rformat ) [varlist]
```

The `read` statement is similar to the `write` statement, and reads the values of the variables listed in *varlist* from the device linked to logical unit *lunit* in the format specified by *rformat*.

`Read` statements can be formatted or unformatted. If a `read` is used to initialise a variable from the terminal screen, *lunit* should be 5 (often referred to as "standard input"), and *rformat* should be `*`, *i.e.*, an "unformatted read". When execution encounters a `read` from the screen, execution is paused until the user has typed the desired value, and hit the `Enter` or `Return` key. Thus, before any `read` from the screen, it is wise to use a `write` to the screen to prompt the user for the desired value, as in the following example:

```
 1        write ( 6, 2010 )
 2 2010   format('Enter a real value for the radius.')
 3        read  ( 5,    * ) radius
```

Without the prompt, execution may remain paused for a very long time as the user wonders what is taking the program so long to execute, possibly unaware that the program is waiting for the user!

Reads may be formatted, in the same fashion `write` statements are formatted (§5.7), but this is usually reserved for `read`s from disc files, in which the exact format of the file is known. If the format rules in the format statement don't match *exactly* the format in the disc file from which the data are read, your data entry can go terribly wrong. For example, the following program reads the values of three variables: `ivar`, an integer; `rvar`, a real; and `date`, a character string (ten characters) from the discfile `indata`.

```
 1        program testread
 2 c
 3        character*10 date
 4        integer      ivar
 5        real         rvar
 6 c
 7        open  ( 10, file='indata', status='unknown' )
 8        read  ( 10, 1010 ) i, rvar, date
 9        write (  6, 2040 ) i, rvar, date
10        close ( 10 )
11 c
12 1010   format( i4, 1pg12.5, a10 )
13 2040   format('i=', i4, ', rvar=', 1pg12.5, ', date=', a10 )
14
15        stop
16        end
```

The `read` statement in line 8 refers to `format` statement `1010` (line 12) which specifies that the first 4 characters (character 1 through 4) on the input line shall contain the integer variable, the next 12 characters (characters 5 through 16) shall contain the real variable, and the last 10 characters (characters 17 through 26) shall contain the character string. Thus, if `indata` were a disc file with the single line (and note that a blank precedes the '`100`'):

```
 100 1.25000e-03November 1
```

then the output from the program `testread` would look like:

```
i= 100, rvar= 1.25000E-03, date=November 1
```

and everything is fine. However, just one slightest glitch in `indata` such as putting two spaces between the `100` and the `1.25` instead of one space:

```
 100  1.25000e-03November 1
```

can corrupt the output from the program `testread` completely:

```
i= 100, rvar=  1.2500 , date=3November
```

This is because the formatted `read` was expecting characters 5 through 16 to contain the real number. With the real shifted over by one character, the last digit of the exponent (`3`) is now the 17th character, and is not read as the last character of the real number (which is therefore reported as `1.2500`), but instead as the first character of the character string, which now loses its last intended character, the digit `1`.

But things could be much worse. Look what happens if the blank in front of the `100` is omitted, causing the field for the real number (characters 5 through 16) to include what was intended as the first letter of the character string (`N`). Thus, if `indata` is entered by mistake as:

```
100 1.25000e-03November 1
```

the program aborts, giving the most cryptic output:

```
dofio: [1015] read unexpected character
logical unit 10, named 'indata'
lately: reading sequential formatted external IO
part of last data: .25000e-03No|vembe
Abort
```

All that is obvious to me is the last line: `Abort`. The rest is gobbledigook, except maybe to more "seasoned" programmers.

You may have already come to the conclusion that formatted read statements are completely impractical for large quantities of data entry, particularly if those data are to be typed in by a human. Should another computer program generate the data, fine, but if a human has to type in hundreds of exact numbers in precise fields, forget formatted reads. For most beginning programmers, unformatted reads from the computer screen are likely the best way to input data to start off a program run. But even this can get unwieldy if the program evolves to one which requires dozens of input parameters for each run. In this case, the more advanced programmer may want to look into the `namelist` feature, which is supported by many *FORTRAN*77 compilers, and is standard to *FORTRAN*90, but beyond the scope of this primer.

Finally, as `write` statements can be used to convert a real variable to a character string, a `read` statement can be used to convert a character string to real. Thus, line 4 in the following example:

```
1       character*10 cvar
2       real         rvar
3       cvar = '1.3807e-23'
4       read ( cvar, * ) rvar
```

assigns to `rvar` the real value equivalent to the character string assigned to `cvar` in line 3.

5.9 Intrinsic Functions

Intrinsic functions are keywords recognised by the *FORTRAN* compiler when included as part of the right hand side of an assignment statement that invoke mathematical operations. With the exception of the double asterisk (§5.9.1), intrinsic functions have the format

function `(` *parameter list* `)`

where, for the most part, the parameter list consists of just one variable, constant, or arithmetic expression.

5.9.1 `**`

```
y = x**n
```

raises `x` to the `n`th power, and assigns that value to `y`. Both `x` and `y` must be declared to have the same data type (integer, real, or double precision), otherwise a type mismatch error will occur. If `x` and `y` are integer, `n` must be integer too. To raise an integer to a real power, you must first convert all integers to real. If `x` and `y` are real (double precision), `n` may be integer or real (double precision).

If `n` is a positive integer, the compiler interprets the above expression as:

$$y = x \times x \times \cdots \times x \quad \text{(where } x \text{ appears } n \text{ times)}.$$

If n is a negative integer, the compiler computes:

$$y = \frac{1}{x \times x \times \cdots \times x} \quad \text{(where } x \text{ appears } n \text{ times)} >$$

Finally, if n is real, the compiler computes:

$$y = e^{n \ln x}.$$

Thus, numerically, `x**2` ($x \times x$) and `x**2.0` ($e^{2 \ln x}$) may give slightly different results because of "round-off errors". Since raising a real number by a real power invokes the log and exponentiation functions, it is considerably more costly in computer time than simply multiplying a number by itself n times. Thus, it is always more desirable to do `x**`n than `x**`n`.0` when n is an integer. However, when n is not an integer, there is no way of avoiding the invocation of the log-exponentiation algorithm.

5.9.2 `sqrt`

```
y = sqrt ( x )
```

assigns to `y` the square root of `x`, and may be used for both single and double precision variables. `x` and `y` must be declared with the same data type, otherwise a type mismatch error will result.

Square roots are expensive computationally, and should be avoided where possible. The following is an example of where you might think a square root is necessary, but in fact can be avoided. Suppose you want to perform one set of calculations inside a sphere of radius r_0, and another set outside that radius. If you are doing the calculation in 3-D Cartesian coordinates, the distance r between the origin (centre of the sphere) and a point (x, y, z) is given by $r = \sqrt{x^2 + y^2 + z^2}$, and thus the relevant snippet of coding might look like:

```
 1        do 30 k=1,kmax
 2          zsq = zed(k)**2
 3          do 20 j=1,jmax
 4            ysq = why(j)**2
 5            do 10 i=1,imax
 6              radius = sqrt ( eks(i)**2 + ysq + zsq )
 7              if ( radius .gt. r0 ) then
 8                instruction set 1
 9              else
10                instruction set 2
11              endif
12 10         continue
13 20       continue
14 30     continue
```

First, note that we can avoid doing many of the squares by computing z^2 once per loop on k (line 2), and y^2 once per loop on j (line 4). Note that the square root on line 6 can be omitted altogether if we just do the `if`-test on `radius**2` rather than on `radius`. Thus, by replacing lines 6 and 7 with:

```
6              radsq = eks(i)**2 + ysq + zsq
7              if ( radsq .gt. r0sq ) then
```

(where the line `r0sq = r0**2` needs to be inserted somewhere before `do`-loop 30 starts) causes a logically-equivalent `if`-test to be performed, but without the `sqrt` function.

5.9.3 exp

```
y = exp ( x )
```

performs an exponentiation ($y = e^x$), and may be used for both single and double precision variables. `x` and `y` must be declared with the same data type, otherwise a type mismatch error will result.

5.9.4 log

```
y = log ( x )
```

performs a natural logarithm ($y = \ln x$), and may be used for both single and double precision variables. `x` and `y` must be declared with the same data type, otherwise a type mismatch error will result.

5.9.5 log10

```
y = log10 ( x )
```

performs a logarithm to the base 10 ($y = \log_{10} x$), and may be used for both single and double precision variables. `x` and `y` must be declared with the same data type, otherwise a type mismatch error will result.

5.9.6 sin

```
y = sin ( x )
```

performs the function $y = \sin x$, where x is assumed to be in radians. This function may be used for both single and double precision variables. `x` and `y` must be declared with the same data type, otherwise a type mismatch error will result.

5.9.7 cos

```
y = cos ( x )
```

performs the function $y = \cos x$, where x is assumed to be in radians. This function may be used for both single and double precision variables. `x` and `y` must be declared with the same data type, otherwise a type mismatch error will result.

The tangent function `tan`, inverse trig functions (`asin`, `acos`, `atan`), and all hyperbolic trigonometric functions (`sinh`, `cosh`, `tanh`) exist as well and are used the same way as `sin` and `cos`. Inverse hypertrig functions (`asinh`, `acosh`, `atanh`) may exist as extensions of your compiler, but are not part of ANSI (American National Standard Institute) *FORTRAN*77.

5.9.8 min

```
y = min ( x1, x2[, x3, ..., xn] )
```

sets `y` to the minimum value among those listed in the argument list. The argument list must have at least two elements in it, and each element must have the same type (*i.e.*, all integer, all real, or all double precision). `y` must be the same type (integer, real, or double precision) as the elements in the argument list (`x1`, `x2`, *etc.*). If any one of `y`, `x1`, *etc.* have a different type from any of the others, a type mismatch error will occur.

5.9.9 max

```
y = max ( x1, x2[, x3, ..., xn] )
```

sets `y` to the maximum value among those listed in the argument list. The argument list must have at least two elements in it, and each element must have the same type (*i.e.*, all integer, all real, or all double precision). `y` must be the same type (integer, real, or double precision) as the elements in the argument list (`x1`, `x2`, *etc.*). If any one of `y`, `x1`, *etc.* have a different type from any of the others, a type mismatch error will occur.

5.9.10 abs

```
y = abs ( x )
```

sets y to the absolute value of x (by setting the first bit of the number to 1, *i.e.*, positive). This function may be used for integer, real, and double precision variables. x and y must be declared with the same data type, otherwise a type mismatch error will result.

5.9.11 sign

```
y = sign ( x1, x2 )
```

(transfer of sign) sets y to x1 if x2 is greater than or equal to zero, and to -x1 if x2 is less than zero. This function may be used for integer, real, and double precision variables, but all variables must have the same type, otherwise a type mismatch error will occur. Thus, if x1 and x2 are real, y must be declared real, *etc.*

5.9.12 int

```
i = int ( x )
```

converts a real or double precision value (x) to an integer by truncation, and assigns that integer to i. If i is not declared to be an integer, a type mismatch error will occur.

Truncation means that the decimal portion of the real number is simply cut off. Thus, 1.2 is truncated to 1; 3.993 is truncated to 3, -5.27 is truncated to -5, and -12.75 is truncated to -12.

5.9.13 real

```
y = real ( x )
```

converts an integer or double precision variable (x) to real, and assigns that value to y. If y is not declared real, a type mismatch error will occur.

5.9.14 dble

```
y = dble ( x )
```

converts an integer or real variable (x) to double precision, and assigns that value to y. If y is not declared double precision, a type mismatch error will occur.