PHYSICS 210

OVERVIEW OF PROGRAMMING

Programming Paradigms

- **Procedural / Imperative (our primary focus in this course)**
  - Programmer specifies sequence of steps needed to complete task
  - Two key components: Data (Structures) & Algorithms
  - Also typically involves decomposition of large program into smaller modules (procedures, functions)
  - Examples: C, Pascal, MATLAB, Fortran, …

- **Functional (our secondary focus)**
  - Treats programming as evaluation of mathematical functions
  - Role of data structures downplayed, functions should have no "side effects"; e.g. shouldn't *change* data per se
  - Examples: LISP, Scheme, Haskell, …
  - Will see some elements of functional programming in Maple, MATLAB

Programming Paradigms (cont.)

- **Object-Oriented (will not use in class / homework, but you can use it in your term project)**
  - Uses “objects” which combine data (datafields) and algorithms/procedures/functions (methods) which operate on the datafields
  - Relative to procedural & functional programming, focus is more on the data itself, rather than the processes which manipulate the data
  - Has become a very popular mode of programming since the 1990’s, particularly for large projects involving many programmers
  - Examples: C++, Java, Smalltalk, and many others that have support for objects (Python, Perl, …)
  - Arguably less intuitive than procedural programming, and can be “overkill” for many task in scientific computing

Basic Constructs for Procedural Programs

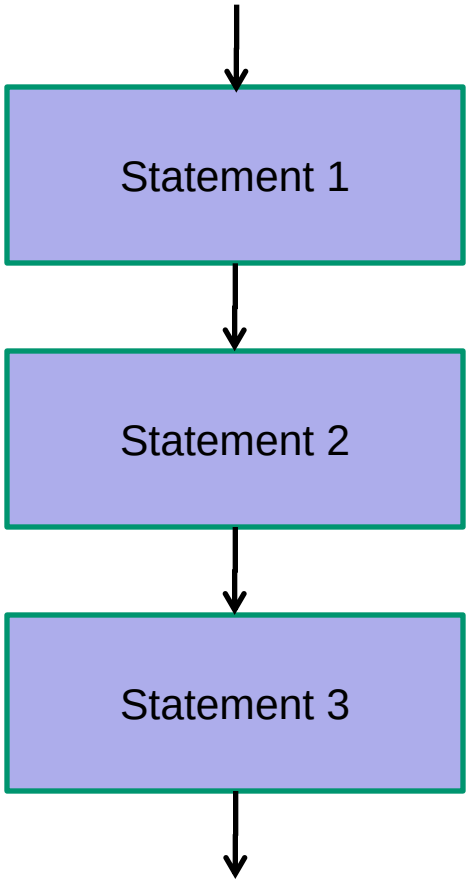
- Given the notion of a **statement** in a programming language, which we will take as an elementary operation (such as assignment of a value to a variable, or a call to a sub-program / function …), there are essentially only 3 fundamental constructs that a procedural programming language needs to have

1. **Sequence**
2. **Selection**
3. **Iteration**

Sequence

- Ability to execute an arbitrary number of statements, in the order in which they occur in the code
- Straightforward in *all* procedural languages
- **octave/maple** generic example

statement 1
statement 2
statement 3
.
.
.



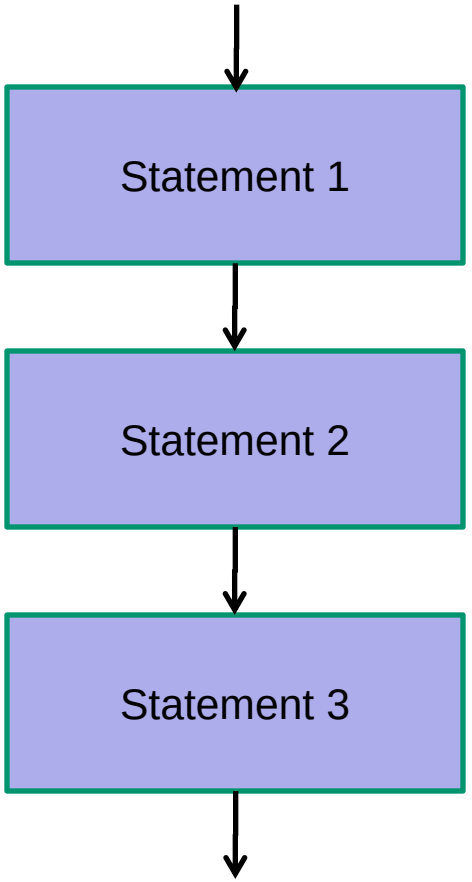
Sequence

- **maple** specific example

```
a := 2;
vec := [1 2 3 4];
cos(2);
          .
          .
```

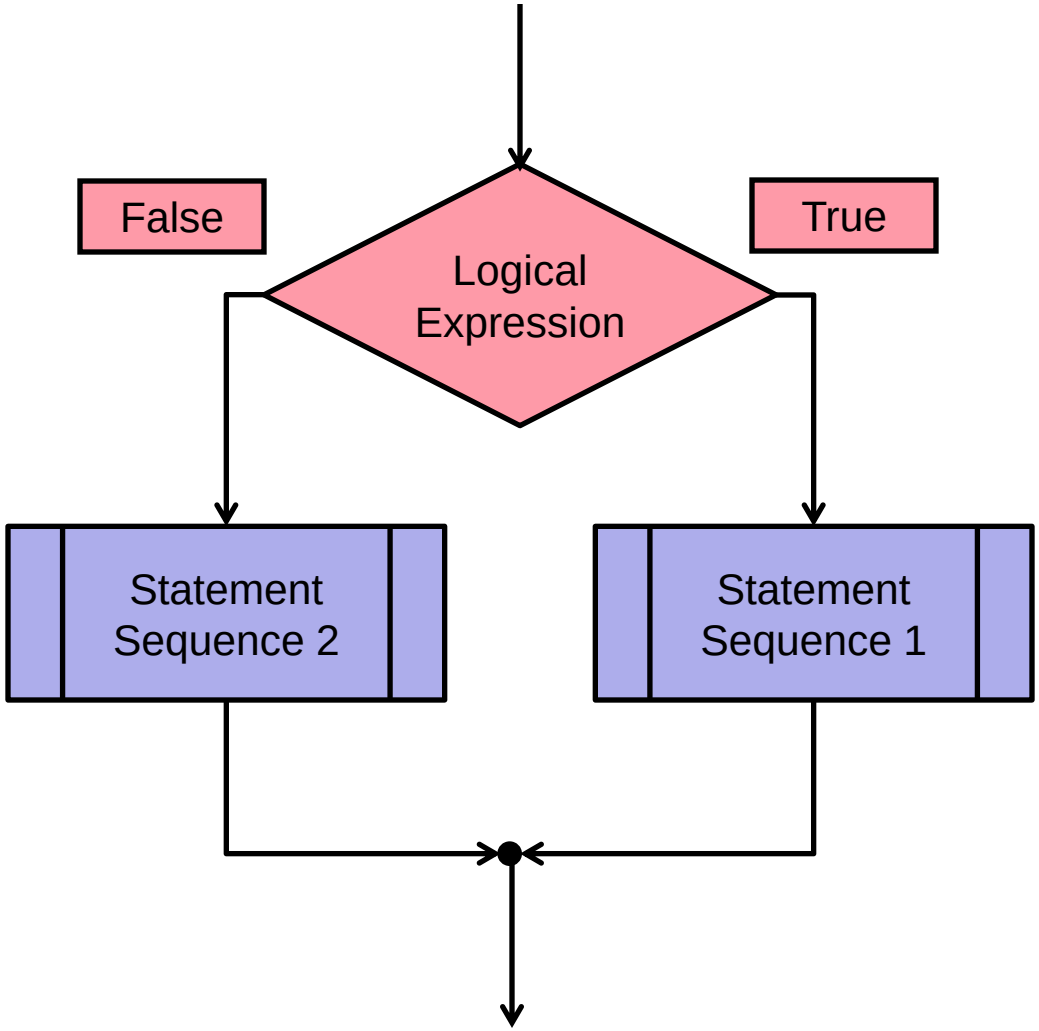
- **octave** specific example

```
a = 2
vec = [1 2 3 4]
cos(2)
          .
          .
```



Selection

- Ability to evaluate logical (i.e. true/false / binary / boolean) expressions, and then follow one of two branches of code that eventually merge into a single branch



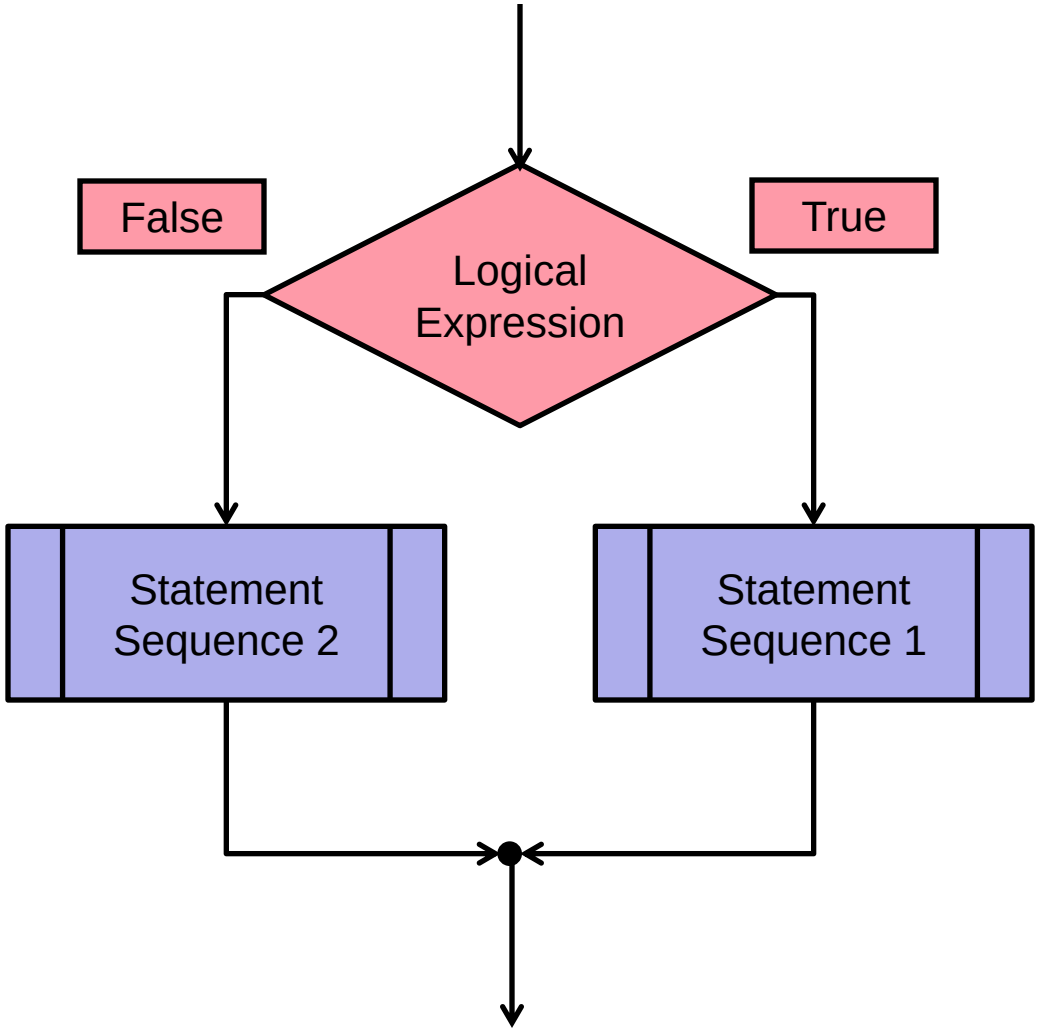
Selection

- **maple** generic example:

if *logical-expr* **then**
    statements 1
else
    statements 2
fi;

- **octave** generic example:

if *logical-expr*
    statements 1
else
    statements 2
end



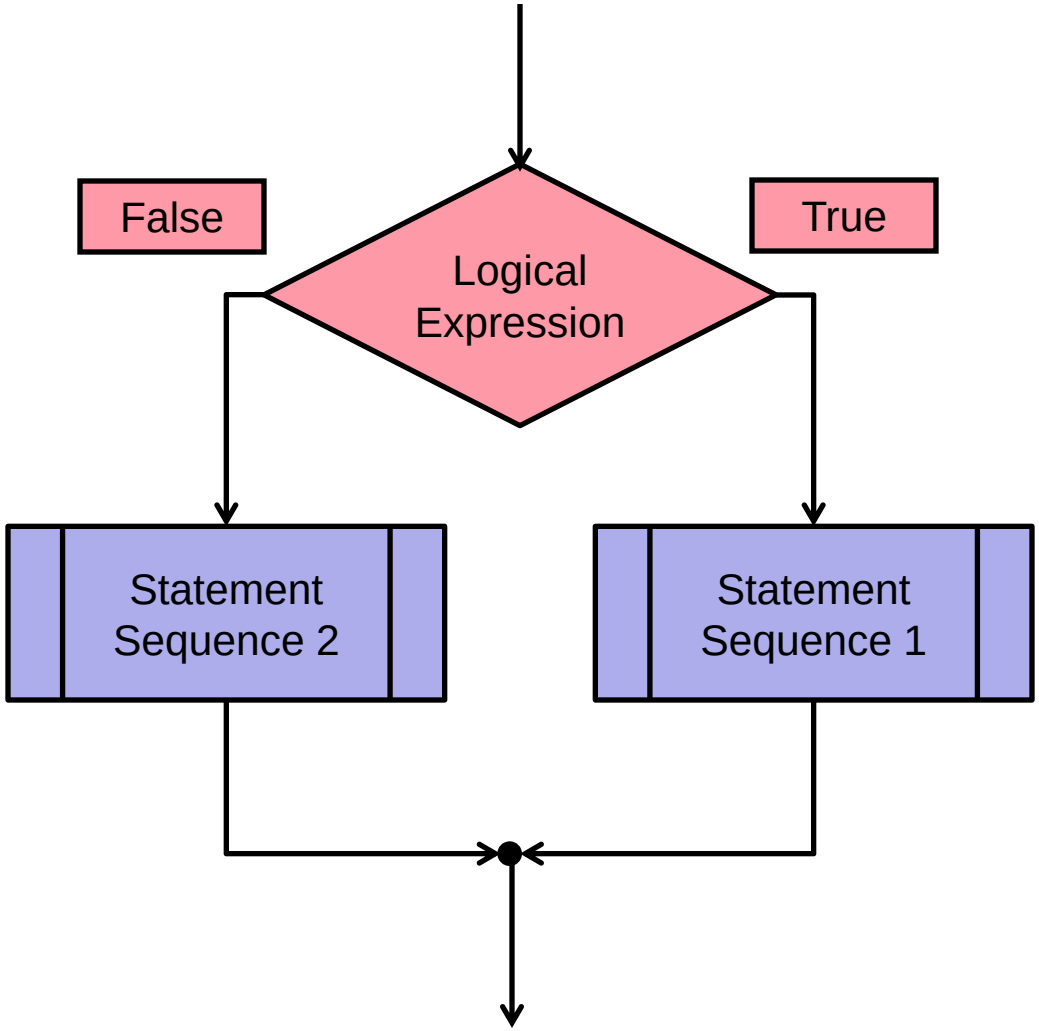
Selection

- **maple** specific example:

```
if n = 1 then
    vec := [1 2 3 4];
else
    vec := [4 3 2 1];
fi;
```

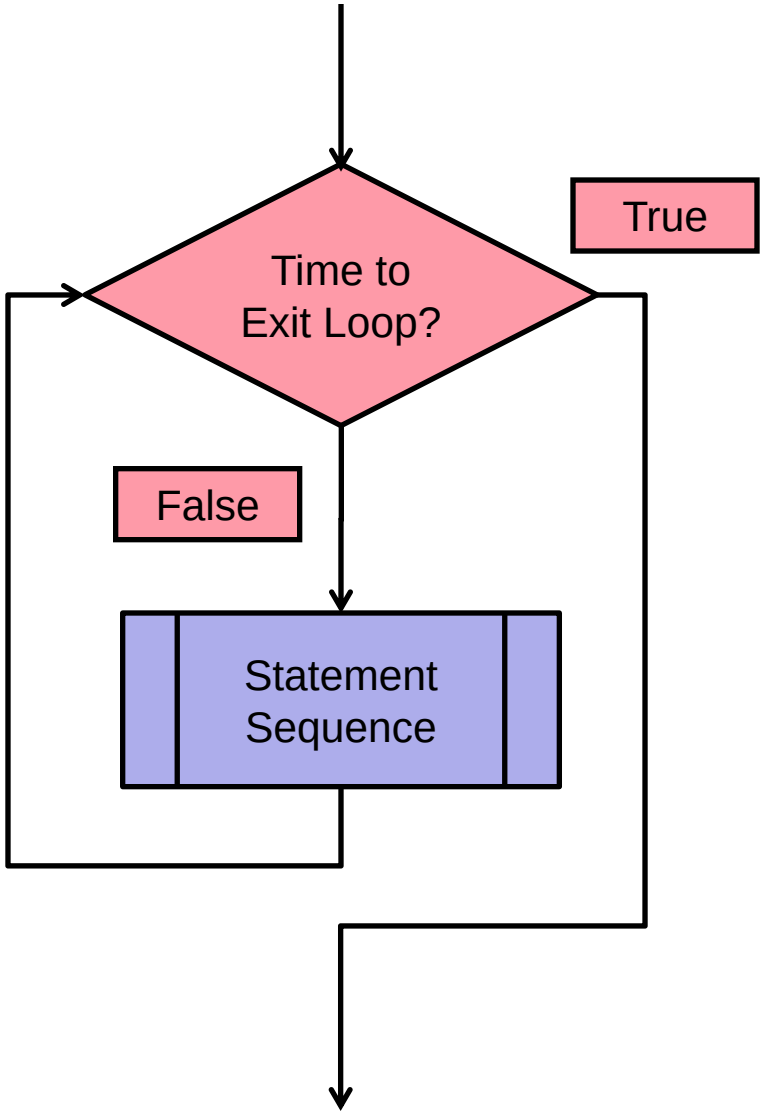
- **octave** specific example:

```
if n == 1
    vec = [1 2 3 4]
else
    vec = [4 3 2 1]
end
```



Iteration

- Ability to repeatedly execute some arbitrary statement sequence (looping):
  1. With some loop parameter ranging over specified values
  2. While some logical expression remains true
  3. Until some logical expression becomes true



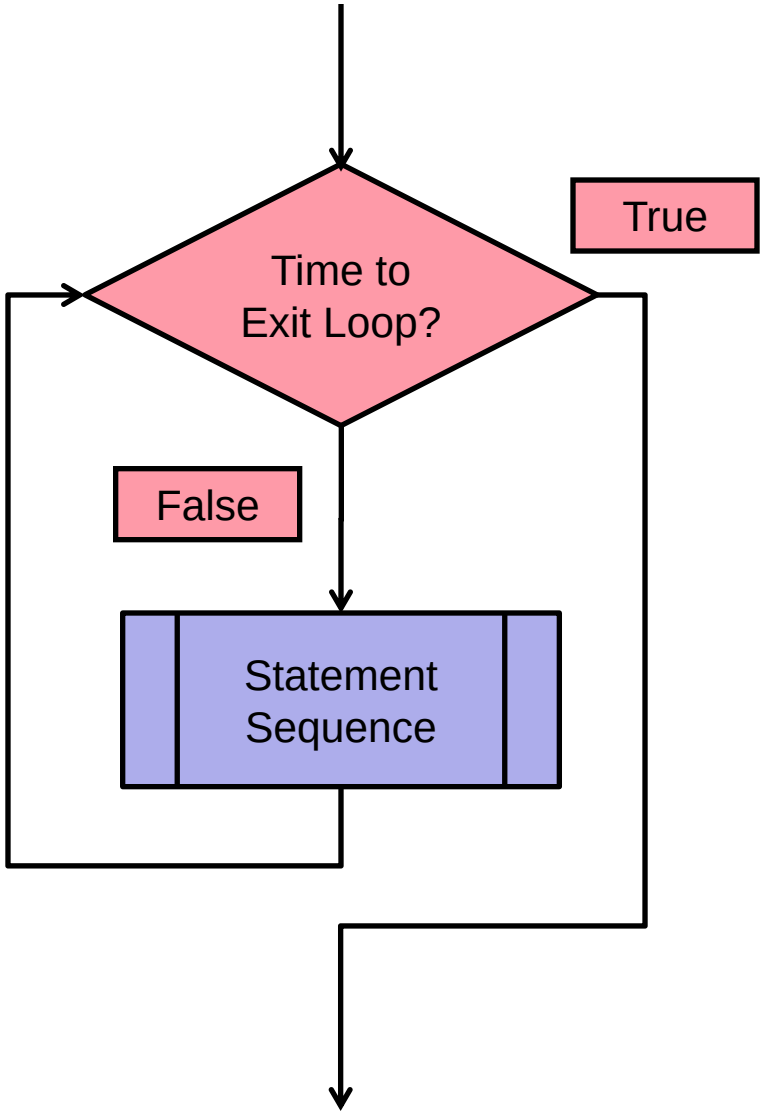
Iteration

- **maple** generic example (type 1)

  for *var* **from** *start* **to** *finish* **do**
  commands
  od;

- **octave** generic example (type 1)

  for *var* **=** *start* **:** *finish*
  commands
  end



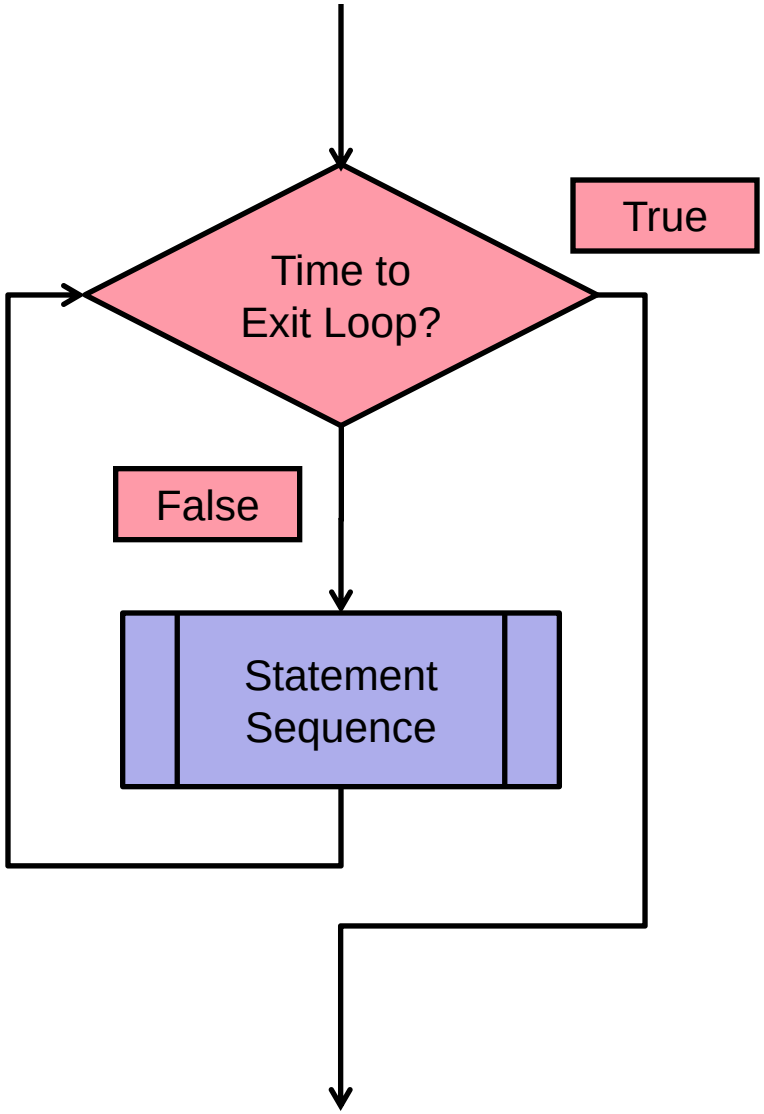
Iteration

- **maple** specific example (type 1)

```
for i from 1 to 10 do
    vec[i] := i^2;
od;
```

- **octave** specific example (type 1)

```
for i = 1 : 10
    vec(i) = i^2
end
```



Golden Rules of Programming

1. DESIGN IS IMPORTANT!

- Think carefully about what you need to do, and how you are going to do it, **before** you start programming ("coding")
- Resist the urge to "make it up as you go along" while sitting in front of the computer (except for very simple programs, or if you need to experiment)

2. IMPLEMENT INCREMENTALLY

- If possible, adopt a top-down approach, get "skeleton" of program working, then add to it gradually
- Avoid temptation to write large blocks of code in one go, and then debug
- Difficulty in debugging tends **not** to be linear in the number of bugs (i.e. more than twice as hard to find and fix two bugs than to find and fix one, etc.)

Golden Rules of Programming

3. MODULARIZE LARGE CODES

- When a code gets lengthy (more than a 100 lines or so), try to decompose into a set of sub-programs (routines, functions, procedures), each of which can be tested individually
- Try to identify commonality in sections of code, or important basic operations that are used repeatedly on your data (structures) and implement them as functions/procedures …

4. ERROR CHECKING & (EXHAUSTIVE) TESTING ARE VITAL

- Ensure that program does the **correct** thing with (all) **valid** input
- Ensure that program handles **invalid** input **gracefully**, e.g. exit with an appropriate error message
  - **Worst possible situation:** invalid input not detected, and program proceeds to compute something (GIGO principle – "garbage in, garbage out")

Golden Rules of Programming

5. LEARN HOW TO DEBUG EFFECTIVELY

- Debugging is a rather unique type of mental activity
  - First and foremost, you must accept that you **have** made a mistake; especially for novices it is natural/easy to assume that "the computer" is at fault, but 99.999…% of the time, **the fault will be yours!**
- **Look** at the values of your data as the program proceeds, i.e. by outputting the data to the terminal (or files if it will be convenient, e.g., to plot the data) at key points in the code – i.e. use **tracing** statements
- Put the tracing statements into your code **as you write it**; this adds a little more time to the coding, but, counter-intuitively, tends to save time in the end
- Code tracing statements so that they can be enabled and disabled easily (i.e. without; removing / re-inserting / commenting-out the statements); e.g. put the tracing statements within **if** *debug-on* **then** *tracing-statements* **end if** constructs

Golden Rules of Programming

6. DOCUMENT YOUR CODE

- Add **comments** to your code, but don't go overboard!
- **Do not** add comments which explain what one or two lines are doing, unless the code is tricky; document **blocks** of code
- **Do** document purposes of main program and sub-programs (functions, procedures …) including
  - inputs
  - outputs
  - main data (structures) used