Outline

- Announcements:
  - Homework I on web, due Fri., 5PM by e-mail
  - Small error on homework
  - Wed and Fri in ACCEL, Attendance required
  - LAST DAY TO ADD/DROP!
- A Closer look at compilation
- What do I mean by "formal methods?"
- Specification
- Verification
- Practical formal methods

A Closer Look at Compilation

- Building an executable requires 2 steps
  - Compile--turn source code into low-level instructions (object files)
    - cc -w or "compile"
  - Link--merge object files to form executable
    - cc or "link"

A Closer Look at Compilation

- Why would you ever use -w?
  - Compilation is slow, linking is fast
  - Suppose program is split into several files: main.c, io.c, matrix.c
    - Suppose we know the code in io.c and matrix.c work, but we're unsure of main.c
    - cc -w *.c --creates main.o, io.o, matrix.o
    - If we make a change in main.c, we can just compile it:
      - cc-w main.c; cc *.o -omyapp

A Closer Look at Compilation

- What about simple programs?
  - cc -c firsttry.c produces firsttry.o
  - What's missing?
    - We called "printf" to write to the screen
    - The low-level instructions for printf are in a library somewhere on your system.
    - The linker gets those instructions and merges them with firsttry.o

Formal Methods

- As scientist we should be accustomed to precision
  - You must be able to describe the *exact* procedures used in your experiment/analysis
  - This is essential for reproducibility
- Reproducibility is equally important for computational work
  - Formal methods are a collection of techniques to describe *precisely* what a program should do

Importance of Formalism in Scientific Software Development

- A Scary Story:
  - You write a program to implement algorithm X (you think)
  - But, you actually implemented algorithm Y
  - It is possible that two similar algorithms can produce very different results (think chaos)
  - You publish a paper describing your results (from running algorithm Y), but in your methods you describe algorithm X
  - The results are spectacular. You get your Ph.D. and a tenure track job.
  - However, just as you're being reviewed for tenure, a grad student in Afghanistan tries to repeat your experiments. Based on your paper, she correctly implements algorithm X and gets very different results.
  - Your tenure is denied, no one will hire you, you walk around campus with a "Will code for food" sign while the Afghani student takes your position.

Formal Methods

- Formal methods can be divided into two steps:
  - Specification: a precise statement of what a program (or subroutine) should do
  - Verification: a demonstration that the actual program satisfies the specification

Specification Methods

- Math/Logic is the preferred method
  - rigorous
  - Precise
- But, English has its place
  - Some would say, English is not formal
  - My view: a good English spec. is better than nothing at all
    - You may actually write a spec in English
    - Can include in comments

Keys to Specification

- Describe the properties of the inputs and outputs
  - Data structures
  - Precision/type
  - Assumptions: sorted? symmetric? ≠0?
- Describe what will happen if assumptions violated
  - return error
  - return null value, NaN, -999
  - throw exception

Formal Specification

- $I \wedge P \Rightarrow O$
  - I=formal statement about input
  - O=formal statement about output
  - P=program
  - Says that if input conditions are given to program P, then the output conditions will be true
- This just says what a program should do, but says nothing about how it will get done
  - No details of P

Specification

- A specification can take many forms:
  - English: "Given a sorted array of integers, the routine will return the location of k in the array, where k is provided by the user"
  - Logic:

$$(v \in \mathbb{Z}^n \wedge \forall j : v(j) \leq v(j+1)) \wedge (binarysearch(v,k)) \implies$$
$$(k \in v \wedge v(j) = k) \vee (k \notin v \wedge j = -1)$$

Specification

- English: Given real-valued parameters a, b, and c, the routine returns the roots of the quadratic equation
- Math:

$$(a, b, c \in \mathbb{C} \wedge a \neq 0) \wedge (roots(a,b,c,x_0, x_1)) \implies$$
$$x_0 = \frac{b + \sqrt{b^2 - 4ac}}{2a},$$
$$x_1 = \frac{b - \sqrt{b^2 - 4ac}}{2a}$$

Verification

- Demonstrate (prove) that your program satisfies specification
- Keep the specification in mind as you develop your code
  - Lines or sections of code will establish different parts of the specification
  - Sometimes, it is easier to refine I (strengthen assumptions) than to satisfy it

Formal Verification

- There are rigorous approaches to verification
  - Special types of logic that describe what code does
- Theoreticians have built "theorem provers" that can be used to automate this process
  - Theorem: Specification
  - Proof: Logical equivalent of specification should show that specification is true

Formal Verification

- This is a really cool idea
  - See NuPRL web sit
  - http://www.cs.cornell.edu/Info/Projects/NuPrl/Intro/intro.html
- Not practical for most scientific programs
- Essential for software controlling costly or important actions
  - Airplanes
  - Space probes
  - Stock trades

Free Verification

- Typing is very simple form of verification
- In a strongly-typed language (Java)
  - X=Y is allowed only if X and Y are the same type
  - This is very helpful, but doesn't come close to guaranteeing that your program is correct:

```
void BadFunction(int[] big){
    int[] small=new int[5];
    for(int j=0;j<big.length;j++){
        small[j]=2*big[j];
    }
}
```

Practical Formal Methods

- Ideally we would all conduct rigorous specification/verification of our programs
- But who are we kidding?
- In the real world,
  - 1) Write a specification as formally as you can, and put an English approximation in the comments
  - 2) As you write your code, prove to yourself that you are actually solving the problem
    - You should include comments like "this line only works if X is true" and "this line makes sure X is true"

A Useful Technique

- A standard technique in formal methods is to search for invariant properties
  - An invariant is a property that doesn't change
  - If a line of code violates the invariant, next one should reestablish it

Specifying the Model Problem

- English: Given initial distribution of C defined on an evenly spaced grid of m points starting at 0 and ending at L (etc. for u, k and reaction), a time step dt, and an ending time T, RAD1d finds an approximate solution fo PDE at time T

Specifying RAD1d

- We know that C provided by user is an approximate solution at time 0. This suggests an invariant:
  - C is a solution for PDE a current time t
- We can use the invariant to help develop the code

```
Get C at time 0
t=0---establishes invariant at start of loop
while (t<T){
    Build A, b
    Solve AC=b--invariant violated, C is sol'n at t+dt
    t=t+dt --invariant reestablished
}
```