What is Frama-C?

- A set of extensions for the C language to permit formal specification and verification of code
- Usable with existing C libraries and interfaces
- Utilizes specifications embedded in comments using the ANSI C Specification Language (ACSL)
ACSL Basics

- Tag comments with `//@` or `/*@`
- Allows specification of:
  - Pre and post conditions
  - Logical functions
  - Predicates
  - Lemmas
  - Dependencies (sort of)
Let’s build a simple example: the `makeN` function.

- This function takes a pointer `i` to an integer, and a number `n`.
- If `n` is non-negative, it sets the value at `i` to `n` and returns `true`.
- Otherwise, it returns `false`.
- The function must only modify the value at `i` if `n` is non-negative.
- The function must not modify `i` itself or `n` under any conditions.
Preconditions

requires: What must be true when this function is called?

/*@
   requires \valid(i);
   requires n >= 0;
*/
bool makeN(int *i, int n) { return true; }
ensures: What will be true when this function terminates?

/*@ 
requires \valid(i)
requires n >= 0;

ensures *i == n;
*/
bool makeN(int *i, int n) {
    *i = n;
    return true;
}
assigns: What values does this function assign to?

```c
/*@ 
   requires \valid(i)
   requires n >= 0;

   assigns *i;

   ensures *i == n;
*/
bool makeN(int *i, int n) {
   *i = n;
   return true;
}
```
So what if we want to do something conditionally?

behavior: What is one possible outcome of this function?

assumes: What does this behavior assume?

```c
/*@ 

requires \valid(i)

behavior NonNegative:
  assumes n >= 0;

  assigns *i;

  ensures *i == n;
  ensures \result == true;

*/
```
Behaviors

/*@ 
behavior Negative:
    assumes n < 0;
    assigns \nothing;
    ensures \result == false;
    complete behaviors;
    disjoint behaviors;
*/
Behaviors

bool makeN(int *i, int n) {
    if (n >= 0) {
        *i = n;
        return true;
    }
    return false;
}
Frama-C allows you to specify logic functions and predicates which exist only in the prover, not in the source code. These functions and predicates operate at a certain state of the program.

```c
/*@ 
logic int getI{T}(int *i) = *i;

predicate isValidI{T}(int *i) = getI{t}(i) >= 0;
*/
```
Lemmas

Lemmas are results which should be true given the defined logic functions and predicates, but are not tied to a particular source code function.

```c
/*@ predicate IsEqual{A, B}(int *a, integer n, int *b) = 
   \forall integer i; 0 <= i < n ==> \at(a[i], A) == \at(b[i], B);

lemma EqualityReflexive{T} : 
   \forall int *i; IsEqual{T, T}(i, i);

lemma EqualityCommutative{S, T} : 
   \forall int *i, *j; 
   IsEqual{S, T}(i, j) ==> IsEqual{T, S}(j, i);
*/
```
We can use predicates in our pre- and post-conditions, too (\valid & others are just built-in predicates)

```c
/*@ ...
behavior Negative:
  assumes n < 0;
  assigns \nothing;
  ensures \result == false;
  ensures isEqual{Pre, Post}(i, i);

... */
```
Benefits

- Most programmers are familiar with C
- Can integrate easily with existing libraries
Detriments

- Most programmers are not familiar with ACSL
- Verbosity. Multiple slides of code even for trivial example like `makeN`
Questions?