# CSSE3100 Crib Sheet

## Exam Format

The confirmed format of the exam is:

| | |
|---|---|
| Q1 | weakest precondition reasoning. |
| Q2 | method specification and loop invariants. |
| Q3 | recursion and termination metrics. |
| Q4 | classes and data structures. |
| Q5 | lemmas and functional programming |

This section will be removed before the exam

## Question 1

### Predicate Logic

| | |
|---|---|
| $A \wedge (A \vee B) \equiv A \equiv A \vee (A \wedge B)$ | (A.6) |
| $A \wedge (B \vee C) \equiv (A \wedge B) \vee (A \wedge C)$ | (A.7) |
| $A \vee (B \wedge C) \equiv (A \vee B) \wedge (A \vee C)$ | (A.8) |
| $\neg(A \wedge B) \equiv \neg A \vee \neg B$ | (A.18) |
| $\neg(A \vee B) \equiv \neg A \wedge \neg B$ | (A.19) |
| $A \vee (\neg A \wedge B) \equiv A \vee B$ | (A.20) |
| $A \wedge (\neg A \vee B) \equiv A \wedge B$ | (A.21) |
| $A \Rightarrow B \equiv \neg A \vee B$ | (A.22) |
| $A \Rightarrow B \equiv \neg(A \wedge \neg B)$ | (A.24) |
| $\neg(A \Rightarrow B) \equiv A \wedge \neg B$ | (A.25) |
| $A \Rightarrow B \equiv \neg B \Rightarrow \neg A$ | (A.26) |
| $C \Rightarrow (A \wedge B) \equiv (C \Rightarrow A) \wedge (C \Rightarrow B)$ | (A.33) |
| $(A \vee B) \Rightarrow C \equiv (A \Rightarrow C) \wedge (B \Rightarrow C)$ | (A.34) |
| $C \Rightarrow (A \vee B) \equiv (C \Rightarrow A) \vee (C \Rightarrow B)$ | (A.35) |
| $(A \wedge B) \Rightarrow C \equiv (A \Rightarrow C) \vee (B \Rightarrow C)$ | (A.36) |
| $A \Rightarrow (B \Rightarrow C) \equiv (A \wedge B) \Rightarrow C \equiv B \Rightarrow (A \Rightarrow C)$ | (A.37) |
| $(A \Rightarrow B) \wedge (\neg A \Rightarrow C) \equiv (A \wedge B) \vee (\neg A \wedge C)$ | (A.38) |
| $(\forall x \text{ s.t. } x = E \Rightarrow A) \equiv A[x \backslash E] \equiv (\exists x \text{ s.t. } x = E \wedge A)$ | (A.56) |
| $\forall x :: A \wedge B = (\forall x :: A) \wedge (\forall x :: B)$ | (A.65) |
| $\forall x :: A = A$ provided $x$ not free in $A$ | (A.74) |

### Rules to know

### Basic Function

```
method MyMethod(x: int) returns (y: int)
    requires x == 10
    ensures y >= 25
{
    {x == 10}
    {x + 3 + 12 == 25}
    var a := x + 3;
    {a + 12 == 25}
    var b := 12;
    {a + b == 25}
    y := a + b;
    {y >= 25}
}
```

### Loops

```
{J}
while B
        invariant J
{
        {B && J}
        ...
        {J}
}
{J && !B}

{y >= 4 && z >= x}
while z < 0
        invariant y >= 4 && z >= x
{
        {z < 0 && y >= 4 && z >= x}
        {y >= 4 && z + y >= x}
        z := z + y;
        {y >= 4 && z >= x}
}
{z >= 0 && y >= 4 && z >= x}
```

### Arrays

```
var a := new string[20];
# Type of a is array<string>
var m := new bool[3, 10];
# Type of m is array2<bool>
```

```
method LinearSearch<T>(a: array<T>, P: T -> bool)
returns (n: int)
ensures 0 <= n <= a.Length
ensures n == a.Length || P(a[n])
ensures n == a.Length ==>
forall i :: 0 <= i < a.Length ==> !P(a[i])
{
n := 0;
while n != a.Length
        invariant 0 <= n <= a.Length
        invariant forall i :: 0 <= i < n ==>
                                !P(a[i])
{ 0 <= n < a.Length &&
(!P(a[n]) ==> (forall i :: 0 <= i < n ==>
                                !P(a[i]))
                                        && !P(a[n])) }
{ (P(a[n]) ==> 0 <= n <= a.Length &&
(n == a.Length || P(a[n])) &&
(n == a.Length ==>
forall i :: 0 <= i < a.Length ==> !P(a[i]))) &&
(!P(a[n]) ==> (forall i :: 0 <= i < n ==>
                                !P(a[i])) && !P(a[n])) }
if (P(a[n])) {
        return;
}
{ (forall i :: 0 <= i < n ==> !P(a[i])) (A.56)
&& (forall i :: i == n ==> !P(a[i])) } (A.65)
{ forall i :: (0 <= i < n ==> !P(a[i])) &&
                                (i == n ==>
                                !P(a[i])) } (A.34)
{ forall i :: 0 <= i < n || i == n ==> !P(a[i])}
{ forall i :: 0 <= i < n + 1 ==> !P(a[i]) }
n := n + 1;
{ forall i :: 0 <= i < n ==> !P(a[i]) }
```

### Methods

```
wp(t := M(E), Q )
  = P[x\E]
    &&  forall y' ::
      R[x,y\E, y']
        ==> Q[t\y']
```

```
Given:
method Triple(x: int) returns (y: int)
requires x >= 0
ensures y == 3*x {}


{ u == 15}
{ u + 3 >= 0 &&
        3*(u + 3) == 54 } (A.56)
{ u + 3 >= 0 &&
        forall y' :: y' == 3*(u + 3)
                ==> y' == 54 }
t := Triple(u + 3);
{ t == 54 }
```

```
function SeqSum(s: seq<int>, lo: int, hi: int): int
requires 0 <= lo <= hi <= |s|
decreases hi - lo
{
        if lo == hi then 0 else s[lo] +
            SeqSum(s, lo + 1, hi)
}
```

## Question 2

### Loop Design Techniques

### Look in the postcondition.

For a postcondition A && B, choose the invariant to be A and the guard to be !B.

```
method SquareRoot(N: nat) returns (r: nat)
ensures r*r <= N && N < (r + 1)*(r + 1)
    { { 0 <= N }
    { 0*0 <= N}
    r := 0;
    { r*r <= N }
    while (r + 1)*(r + 1) <= N
    invariant r*r <= N
    {
        { (r + 1)*(r + 1) <= N
                && r*r <= N } (strengthen)
        { (r + 1)*(r + 1) <= N }
        r := r + 1;
        { r*r <= N }
    }
}
```

### Programming by wishing

If a problem can be made simpler by having a precomputed quantity Q, then introduce a new variable q with the intention of establishing and maintaining the invariant q == Q

```
method SquareRoot(N: nat) returns (r: nat)
ensures r*r <= N < (r + 1)*(r + 1)
{
    r := 0;
    var s := 1;
    while s <= N
    invariant r*r <= N
    invariant s == (r + 1)*(r + 1)
    {
        s := s + 2*r + 3;
        r := r + 1;
    }
}
```

### Replace a constant by a variable

For a loop to establish a condition P(C), where C is an expression that is held constant throughout the loop, use a variable k that the loop changes until it equals C, and make P(k) a loop invariant.
For example, Min method (Week 4) had postcondition

```
    ensures forall i :: 0 <= i < a.Length ==>
                                    m <= a[i]
```

and invariant

```
    invariant forall i :: 0 <= i < n ==> m <= a[i]
```

### What's yet to be done

. If you're trying to solve a problem of the form p == F(n), replacement of a constant by a variable results in a what-has-been-done invariant

```
    invariant p == F(i)
```

Alternatively, you may use a what's-yet-to-be-done invariant

```
    invariant p @ F(n { i) == F(n)
```

where @ is some kind of combination operation.

### Use the postcondition

To establish a postcondition Q, make Q a loop invariant.
For the Min example, to ensure the postcondiVon

```
ensures exists i :: 0 <= i < a.Length && m == a[i]
```

we used the invariant

```
invariant exists i :: 0 <= i < a.Length && m == a[i]
```

## Question 3

### Termination Metrics

Any set of values which have a *well-founded* order can be used as a termination metric.
An order $\succ$ is well-founded when

- $\succ$ is irreflexive: a $\succ$ a never holds
- $\succ$ is transitive:
  a $\succ$ b && b $\succ$ c $\implies$ a $\succ$ c
- there is no infinite descending chain
  $a_1 \succ a_2 \succ a_3 \succ \ldots$

We write X decreases to x as $X \succ x$.
For integers, $X \succ x$ when X ¿ x && X ¿= 0.
For booleans, $X \succ x$ when X && !x.
A termination metric for a recursive function is a metric that can be proven to decrease every iteration.
E.g. for the function;

```
    function F(x: int): int
    {
        if x < 10 then x else F(x { 1)
    }
```

the termination metric would be x since $x \succ x - 1$.

### Lexicographic tuples

A lexicographic order is a component-wise comparison where earlier components are more significant.
$\{a_0, a_1, a_2, \ldots, a_n\} \succ \{b_0, b_2, b_3, \ldots, b_n\}$ if and only if
$a_0 \succ b_0$ || ($a_0 == b_0$ && $a_1 \succ b_1$) ||
($a_0 == b_0$ && $a_1 == b_1$ &&
$a_2 \succ b_2$) || ... ||
($a_0 == b_0$ && $a_1 == b_1$ && ... &&
$a_{n-1} == b_{n-1}$ && $a_n \succ b_n$)
A lexicographic ordering allows tuples to be used as termination metrics.

## Mutually Recursive Functions

Tuples can be used to provide termination metrics for mutually recursive functions since you can provide multiple values that the functions may reduce on. E.g. for the following methods;

```
method F(i: nat) returns (r: nat) {
    if i <= 2 { r := 1; }
    else {
        var h := H(i - 2);
        r := 1 + h;
    }
}

method H(i: nat) returns (r: nat) {
    if i == 0 { r := 0; }
    else {
        var f := F(i);
        var h := H(i - 1);
        r := f + h;
    }
}
```

the termination matrix would be {i, 1} for H and {i, 0} for F since the call F(i) in H will reduce on $1 \succ 0$.

# Question 4

## Classes

Ghost variables can be used for specification and reasoning only.

```
ghost var d: T
```

### Simple Classes

A simple class consits of only simple object, (i.e. objects that are not stored on the heap).
The specification for a simple class consists of:

- ghost variables for abstract state
- have class invariant, **ghost predicate Valid()**
- Valid() and functions have **reads this**
- constructor has **ensures Valid()**
- methods have **requires Valid()**, **modifies this**, **ensures Valid()**

Concrete states that consist of only simple objects are created and are related to the abstract state in **valid()**.
The constructor, methods, and functions must satisfy the class specification and will require both concrete and abstract state to be updated.

## Complex Classes

Complex classes consist of any combination of simple and complex objects, (i.e. objects that are stored on the heap).
Complex classes require a representation set,

```
ghost var Repr: set<object>
```

### Invariant

The invariant valid will consist of the following, where a, a0, a1 are non-composite objects or arrays and b, b0, b1 are composite objects.

```
ghost predicate Valid()
    reads this, Repr
    ensures Valid() ==> this in Repr
{
    this in Repr && ...
}
```

For a non-composite object or array **a**, include;

```
a in Repr && a.Valid()
```

For a non-composite objects or arrays **a0, a1**, include;

```
a0 != a1
```

For a composite object **b**, include;

```
b in Repr && b.Repr <= Repr &&
this !in b.Repr && b.Valid()
```

For a composite objects **b0, b1** and non-composite objects and arrays **a0, a1**, include;

```
{a0, a1} !! b0.Repr !! b1.Repr
```

### Constructor

For a non-composite array or object **a** and a composite object **b**.

```
constructor()
    ensures Valid() && fresh(Repr)
{
    ... (initialise concrete and abstract state)
    new;
    Repr := {this, a, b} + b.Repr;
}
```

### Functions

```
function F(x:X): Y()
    requires Valid()
    reads Repr
    ensures F(x) == ...
```

### Methods (Mutating)

```
method M(x:X) returns Y()
    requires Valid()
    modifies Repr
    ensures Valid() && valid(Repr - old(Repr))
```

# Question 5

## Lemmas

**lemma** $name(x_1 : T, x_2 : T, \ldots, x_n : T)$
  **requires** P
  **ensures** R
{ }
Lemmas can be called in a method to **prove** the lemmas property from that point onwards.

### Weakest Precondition

**wp**(M(E), Q) = `P[x\E] && (R[x\E] ==> Q)`

## Calc

To prove a lemma by hand, you can add a **calc** section into the lemmas body, where $\gamma$ is the default transitive operator between lines.
**calc** $\gamma$ {
  $5 * (x + 3)$;
  $== 5 * x + 5 * 3$;
  $== 5x + 15$;
}
You can use use any transitive operator between lines (e.g. ==>). If no default operator is specific, the default is ==.
The **calc** statements can also be added inline within a method instead of creating and calling a lemma.

## Induction

Lemmas can also be used to prove using induction by recursively calling the lemma in the body. E.g.
**lemma** SumLemma(a: *array¡int¿*, i: *int*, j: *int*)
  **requires** P
  **ensures** R
{
  if i == j {} // base case: Dafny can prove
  else {
    SumLemma(a, i+1, j); // inductive case
  }
}

## Functional Programming

Key features:

- Program structures as mathematical functions
- Data is immutable (i.e. no heap, no side effects)

## Match

**Match** is dafny's version of a switch statement, but it must cover all cases.
  **match** $x$
  **case** $c_1$
  **case** $c_2$
  ...
  **case** $c_n$

## Descriminators

Discriminators can be used to check if a variable is a given type. E.g. xs.Nil**?** checks if xs is type Nil.

## Destructors

Destructors are used to access data in a composite datatype. E.g. for a variable xs of the datatype **datatype** `List<T>` = Nil — Cons(head: T, tail: `List<T>`),
head can be accessed using xs.head. Similarly tail can be accessed using xs.tail.

## Instrinsic vs Extrinsic Property

- An intrinsic property is a property defined within a specification.
- An extrinsic property is a property defined externally using a lemma.
- Methods in Dafny are opaque, so all properties in the specification are intrinsic.
- Functions are transparent, so properties can be intrinsic or extrinsic.
- Intrinsic properties are available every time we apply a function, whereas extrinsic properties are only available if we call the lemma.
- Having all properties exposed instrinsicly can lead to long verification times, so only define properties intrinsicly if they will be required for all applications of the function.