

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

$A \wedge (A \vee B) \equiv A \equiv A \vee (A \wedge B)$
 $A \wedge (B \vee C) \equiv (A \wedge B) \vee (A \wedge C)$
 $A \vee (B \wedge C) \equiv (A \vee B) \wedge (A \vee C)$
 $\neg(A \wedge B) \equiv \neg A \vee \neg B$
 $\neg(A \vee B) \equiv \neg A \wedge \neg B$
 $A \vee (\neg A \wedge B) \equiv A \vee B$
 $A \wedge (\neg A \vee B) \equiv A \wedge B$
 $A \Rightarrow B \equiv \neg A \vee B$
 $A \Rightarrow B \equiv \neg(A \wedge \neg B)$
 $\neg(A \Rightarrow B) \equiv A \wedge \neg B$
 $A \Rightarrow B \equiv \neg B \Rightarrow \neg A$
 $C \Rightarrow (A \wedge B) \equiv (C \Rightarrow A) \wedge (C \Rightarrow B)$
 $(A \vee B) \Rightarrow C \equiv (A \Rightarrow C) \wedge (B \Rightarrow C)$
 $C \Rightarrow (A \vee B) \equiv (C \Rightarrow A) \vee (C \Rightarrow B)$
 $(A \wedge B) \Rightarrow C \equiv (A \Rightarrow C) \vee (B \Rightarrow C)$
 $A \Rightarrow (B \Rightarrow C) \equiv (A \wedge B) \Rightarrow C \equiv B \Rightarrow (A \Rightarrow C)$
 $(A \Rightarrow B) \wedge (\neg A \Rightarrow C) \equiv (A \wedge B) \vee (\neg A \wedge C)$
 $(\forall x \text{ s.t. } x = E \Rightarrow A) \equiv A[x \backslash E] \equiv (\exists x \text{ s.t. } x = E \wedge A)$
 $\forall x :: A \wedge B = (\forall x :: A) \wedge (\forall x :: B)$
 $\forall x :: A = A$ provided x not free in A

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}
```

Arrays

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

idk what else to put here
```

- (A.6)
- (A.7)
- (A.8)
- (A.18)
- (A.19)
- (A.20)
- (A.21)
- (A.22)
- (A.24)
- (A.25)
- (A.26)
- (A.33)
- (A.34)
- (A.35)
- (A.36)
- (A.37)
- (A.38)
- (A.56)
- (A.65)
- (A.74)

Methods

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

```
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 5

Lemmas

```
lemma name(x1 : T, x2 : T, ..., xn : T)
  requires P
  ensures R
```

```
{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}
```

```
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]) }
  }
```

```
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 }
  }
```

```
{ }
```

Lemmas can be called in a method to **prove** the lemmas property from that point onwards.

Weakest Precondition

$\text{wp}(M(E), Q) = P[x \backslash E] \ \&\& \ (R[x \backslash E] \implies Q)$

Calc

To prove a lemma by hand, you can add a **calc** section into the lemmas body, where γ is the default transitive operator between lines.

```
calc  $\gamma$  {
  5 * (x + 3);
  == 5 * x + 5 * 3;
  == 5x + 15;
}
```

You can use any transitive operator between lines (e.g. \implies). 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*}, 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 c1
case c2
...
case cn
```

Discriminators

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`.

Intrinsic 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 intrinsically can lead to long verification times, so only define properties intrinsically if they will be required for all applications of the function.