CS:5810 Formal Methods in Software Engineering

Recursion and Termination

Recursive methods

```
method Double(x: int) returns (d: int)
  requires x >= 0
  ensures d == 2*x
{
  if x == 0
  {
    d := 0;
  } else {
    var y;
    y := Double(x - 1);
    d := y + 2;
  }
}
```

Recursive methods

```
method Double(x: int) returns (d: int)
  requires x >= 0
  ensures d == 2*x          { x >= 0 ==> (x != 0 ==> x > 0) }
{                           { x != 0 ==> x > 0 }
  if x == 0                 { (x == 0 ==> 0 == 2*x) &&
                              (x != 0 ==> x - 1 >= 0 ) }
  {                         { 0 == 2*x }
    d := 0;                 { d == 2*x }
  } else {                  { forall y :: x - 1 >= 0 && true}
    var y;                  { x - 1 >= 0 && forall r :: r == 2*(x - 1) ==>
                                                r + 2 == 2*x }
    y := Double(x - 1);     { y + 2 == 2*x }
    d := y + 2;             { d == 2*x }
  }                         { d == 2*x }
}
```

Recursive methods

```
method Double(x: int) returns (d: int)
  requires x >= 0
  ensures d == 2*x
{
  if x == 0
  {
    d := 0;
  } else {
    var y;
    y := Double(x - 1);
    d := y + 2;
  }
}
```

Recursive methods can be analyzed
like any methods that call other methods …

… if they terminate!

Problematic recursion

```
method BadDouble(x: int) returns (d: int)
  ensures d == 2*x
{
  var y := BadDouble(x - 1);
  d := y + 2;
}
```

Does not terminate!

```
method BadIdentity(x: int) returns (y: int)
  ensures y == x
{
  if x % 2 == 2
    { y := x; }
  else
    { y := BadIdentity(x); }
}
```

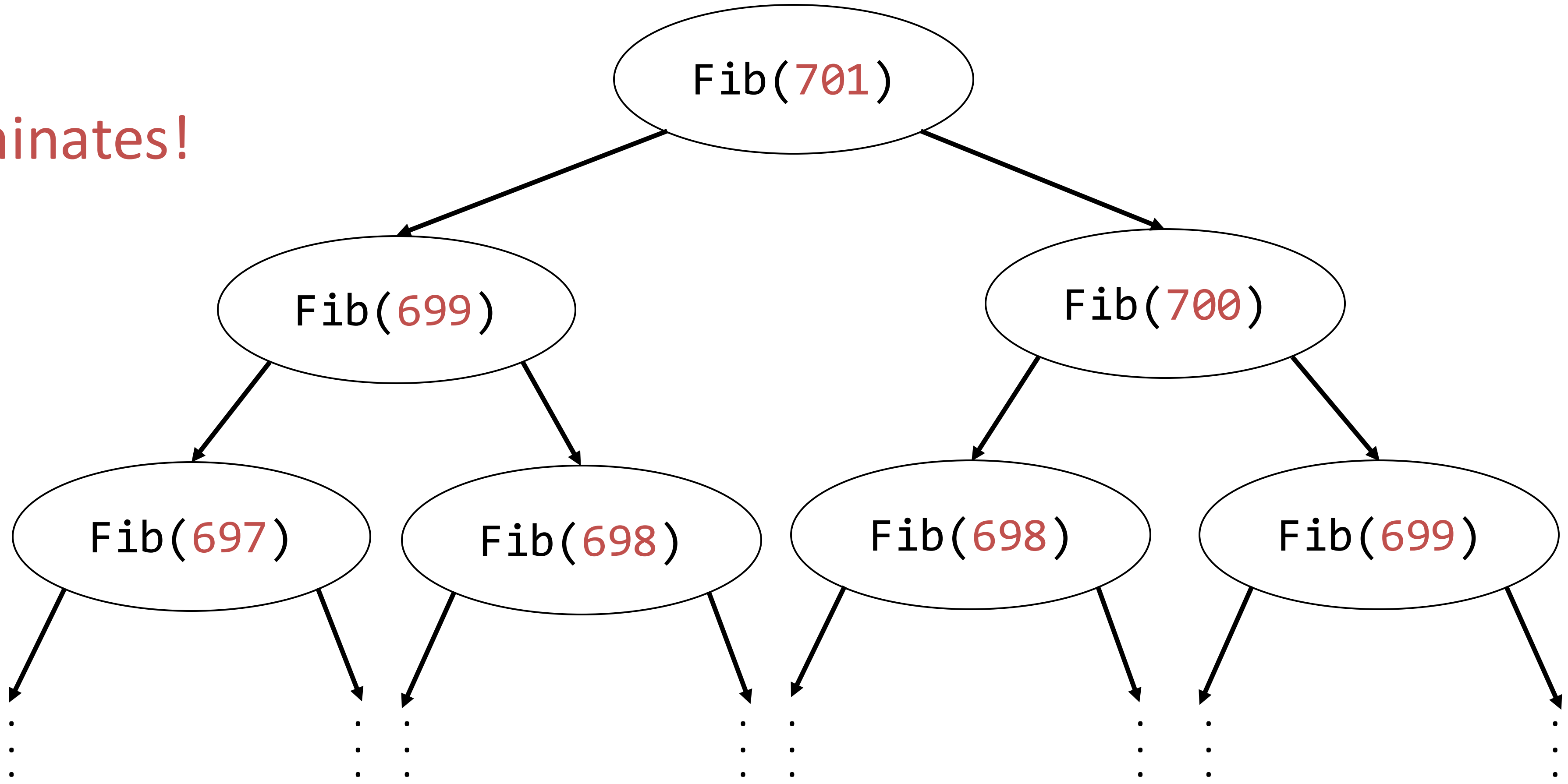
Does not terminate!

Fibonacci function

nat: non-negative integers

```
function Fib(n: nat): nat {
  if n < 2 then n else Fib(n - 2) + Fib(n - 1) }
```

Terminates!



How to prove termination?

```
function Fib(n: nat): nat
{
  if n < 2 then n else Fib(n - 2) + Fib(n - 1)
}

function Ack(m: nat, n: nat): nat
{
  if m == 0 then n + 1
  else if n == 0 then Ack(m - 1, 1)
       else Ack(m - 1, Ack(m, n - 1))
}
```

Also terminates!

Termination metric

```
function Fib(n: nat): nat
  decreases n
{
  if n < 2 then n else Fib(n - 2) + Fib(n - 1)
}
```

Suggestion for Dafny

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

Termination metric

Termination metrics do not have to be natural numbers

Any set of values with a *well-founded order* can be used

An order $\succ$ is well-founded when

- $\succ$ is *irreflexive*: $a \succ a$ never holds
- $\succ$ is *transitive*: if $a \succ b$ and $b \succ c$ then $a \succ c$
- there is no infinite descending chain: $a_0 \succ a_1 \succ a_2 \succ \ldots$

Well-founded orders in Dafny

type	$X \succ Y$ ("X decreases to Y") **iff**	
bool	`X && !Y`	true decreases to false
int	`X > Y && X >= 0`	negative ints not ordered
real	`X - 1.0 >= Y && X >= 0.0`	
set<T>	X is a proper superset of Y	$\supset$, not $\supseteq$
seq<T>	X strictly contains Y	e.g., [a, b, c] $\succ$ [b, c]
datatypes	X structurally includes Y	e.g., ((a, b), (c, d)) $\succ$ (a, b)

Lexicographic tuples

A *lexicographic order* orders tuples of values

It does component-wise comparison,
where earlier components are more significant

Examples:

- `4, 12` $\succ$ `4, 11` $\succ$ `4, 2` $\succ$ `3, 5260` $\succ$ `2, 0`
- `4, 12` $\succ$ `4, 12, 365, 0`
- `12, true, 1.9` $\succ$ `12, false, 57.3`

Lexicographic tuples

A *lexicographic order* orders tuples of values

It does component-wise comparison,
where earlier components are more significant

Theorem: A lexicographic order is well founded whenever all the component orders are well founded

Remaining study

The following method simulates your time until graduation, from when you have h hours of study left in course c

```
method Study(c: nat, h: nat)
  decreases c, h
{
  if h != 0 { Study(c, h - 1); }
  else if c == 0 { } // graduation!
  else { var h1 := ReqStudyTime(c - 1);
         Study(c - 1, h1);
       }
}
```

$c, h \succ c, h\text{-}1$

$c, h \succ c\text{-}1, h1$

Ackermann function

```
function Ack(m: nat, n: nat): nat
  decreases m, n
{
  if m == 0 then n + 1
  else if n == 0 then Ack(m - 1, 1)
       else Ack(m - 1, Ack(m, n - 1))
}
```

Mutually recursive functions

```
method StudyPlan(c: nat)
   requires c <= 40
   decreases 40 - c
{
  if c != 40 { var h := ReqStudyTime(c); Learn(c, h); }
}
```

40-c ≻ 40-c, h

```
method Learn(c: nat, h: nat)
   requires c < 40
   decreases 40 - c, h
 {
   if h == 0 { StudyPlan(c + 1); } else { Learn(c, h - 1); }
 }
```

40-c, h ≻ 40-(c+1)

40-c, h ≻ 40-c, h-1