Programming with Proofs

Verifying Imperative Programs

Reducing an Imperative Program to a Formula

```
{ true }
// Check that this implication is valid (negation is unsatisfiable)
{ (a ⇒ (b ⇒ (true ⇔ (a ⇒ b))) ∧ (¬b ⇒ (false ⇔ (a ⇒ b)))) ∨ (¬a ⇒ (true ⇔ (a ⇒ b)))  }
if (a) {
{ (b ⇒ (true ⇔ (a ⇒ b))) ∧ (¬b ⇒ (false ⇔ (a ⇒ b))) }
  if (b) {
{ true ⇔ (a ⇒ b) }
    res := true
{ res ⇔ (a ⇒ b) }
  } else {
{ false ⇔ (a ⇒ b) }
    res := false
{ res ⇔ (a ⇒ b) }
  }
{ res ⇔ (a ⇒ b) }
} else {
{ true ⇔ (a ⇒ b) }
  res := true
{ res ⇔ (a ⇒ b) }
}
{ res ⇔ (a ⇒ b) }
```

Hoare triples

For predicates P and Q and program S, the *Hoare triple*

states the following:

if S is started in any state that satisfies P,
then S will not crash (or do other bad things) and
will terminate in some state satisfying Q

Examples:

```
{ x == 1 } x := 20          { x == 20 }
{ x < 18 } y := 18 - x      { y >= 0 }
{ x < 18 } y := 5           { y >= 0 }
```

Non-example:

```
{ x < 18 } x := y { y >= 0 }
```

Forward reasoning

Constructing a postcondition from a given precondition

In general, there are many possible postconditions

Examples:

```
1. { x == 0 }  y := x + 3  { y < 100 }
2. { x == 0 }  y := x + 3  { x == 0 }
3. { x == 0 }  y := x + 3  { 0 <= x && y == 3 }
4. { x == 0 }  y := x + 3  { 3 <= y }
5. { x == 0 }  y := x + 3  { true }
```

Strongest postcondition

Forward reasoning constructs the **strongest** (i.e., most specific) postcondition

```
{ x == 0 }  y := x + 3  { 0 == x && y == 3 }
```

Def: A is *stronger* than B if A ==> B is a valid formula

Def: A formula is *valid* if it is true for any valuation of its free variables

Backward reasoning

Construct a precondition for a given postcondition

Again, there are many preconditions

Examples:

```
1.                { x <= 70 }  y := x + 3  { y <= 80 }
2.  { x == 65 && y < 21 }  y := x + 3  { y <= 80 }
3.                { x <= 77 }  y := x + 3  { y <= 80 }
4.  { x*x + y*y <= 2500 }  y := x + 3  { y <= 80 }
5.                  { false }  y := x + 3  { y <= 80 }
```

Weakest precondition

Backward reasoning constructs the weakest (i.e., most general) precondition

```
{ x <= 77 }  y := x + 3  { y <= 80 }
```

Def: A is *weaker* than B if B ==> A is a valid formula

Weakest precondition for assignment

Given $\{ \; ? \; \}\ x := E\ \{ \; Q \; \}$

we construct $?$ by replacing each x in Q with E (denoted by $Q[x \backslash E]$)

Weakest precondition for assignment

Given $\{Q[x \backslash E]\}\ x := E\ \{ Q \}$

Examples: `{ ? }  y := a + b  { 25 <= y }`

1. `{ 25 <= x + 3 + 12 } a := x + 3 { 25 <= a + 12 }`
2. `{ x + 1 <= y } x := x + 1 { x <= y }`
3. `{ 3*2*x + 5*y < 100 } x := 2*x  { 3*x + 5*y < 100 }`

Swap example

```
var tmp := x;

x := y;

y := tmp;
```

Swap example

```
{ x == X && y == Y }

var tmp := x;

x := y;

y := tmp;
{ x == Y && y == X }
```

The initial values of x and y are specified using **logical variables** X and Y

Swap example

```
{ x == X && y == Y }
{ ? }
var tmp := x;
{ ? }
x := y;
{ ? }
y := tmp;
{ x == Y && y == X }
```

The initial values of x and y are specified using **logical variables** X and Y

Swap example

```
{ x == X && y == Y }
{ ? }
var tmp := x;
{ ? }
x := y;
{ x == Y && tmp == X }
y := tmp;
{ x == Y && y == X }
```

Swap example

```
{ x == X && y == Y }
{ ? }
var tmp := x;
{ y == Y && tmp == X }
x := y;
{ x == Y && tmp == X }
y := tmp;
{ x == Y && y == X }
```

Swap example

```
{ x == X && y == Y }
{ y == Y && x == X }
var tmp := x;
{ y == Y && tmp == X }
x := y;
{ x == Y && tmp == X }
y := tmp;
{ x == Y && y == X }
```

Swap example

```
{ x == X && y == Y }
{ y == Y && x == X }
var tmp := x;
{ y == Y && tmp == X }
x := y;
{ x == Y && tmp == X }
y := tmp;
{ x == Y && y == X }
```

The final step is the *proof obligation* that

```
(x == X && y == Y) ==> (y == Y && x == X)
```

is valid

Program-proof bookkeeping

```
{ x == X && y == Y }

x := y - x;

y := y - x;

x := y + x;
{ x == Y && y == X }
```

Program-proof bookkeeping

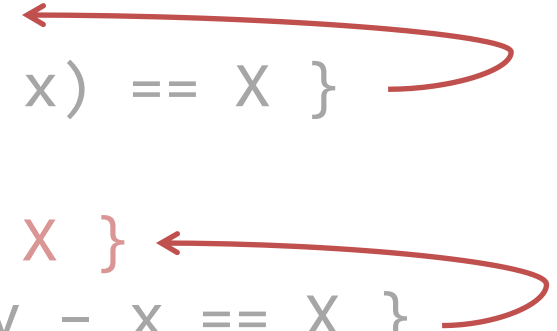
```
{ x == X && y == Y }
{ y - (y - x) + (y - x) == Y && y - (y - x) == X }
x := y - x;
{ y - x + x ==Y && y - x == X }
y := y - x;
{ y + x == Y && y == X }
x := y + x;
{ x == Y && y == X }
```

The constructed precondition simplifies to

```
y == Y && x == X
```

Program-proof bookkeeping

```
{ x == X && y == Y }
{ y == Y && x == X }
{ y == Y && y - (y - x) == X }
x := y - x;
{ y == Y && y - x == X }
{ y - x + x == Y && y - x == X }
y := y - x;
{ y + x == Y && y == X }
x := y + x;
{ x == Y && y == X }
```



We are also allowed to **strengthen** the conditions as we work backwards (but not weaken them!)

What about strongest postconditions?

Consider `{ w < x && x < y } x := 100 { ? }`

Obviously, `x == 100` is a postcondition, but it is **not** the strongest

Something more is implied by the precondition:

there exists an `n` such that `w < n && n < y`

which is equivalent to saying that `w + 1 < y`

In general:

```
{ P } x := E { exists n :: P[x\n] && x == E[x\n] }
```

WP and SP

Let P be a predicate on the pre-state of a program S and
let Q be a predicate on the post-state of S

WP [S, Q] denotes the weakest precondition of S wrt Q

SP [S, P] denotes the strongest postcondition of S wrt P

```
WP [x := E, Q] = Q[x\E]
```

```
SP [x := E, P] = exists n :: P[x\n] && x == E[x\n]
```

Control flow

Until now:

Assignment: `x := E`

Variable introduction: `var x`

Next:

Sequential composition: `S ; T`

Conditions: `if B { S } else { T }`

Method calls: `r := M(E)`

Later:

Loops: `while B { S }`

Sequential composition

S ; T { P } S { Q } T { R }

{ P } S { Q } and { Q } T { R }

Strongest postcondition

let Q = SP [S, P]

SP [S;T, P] = SP [T, Q] = SP [T, SP [S, P]]

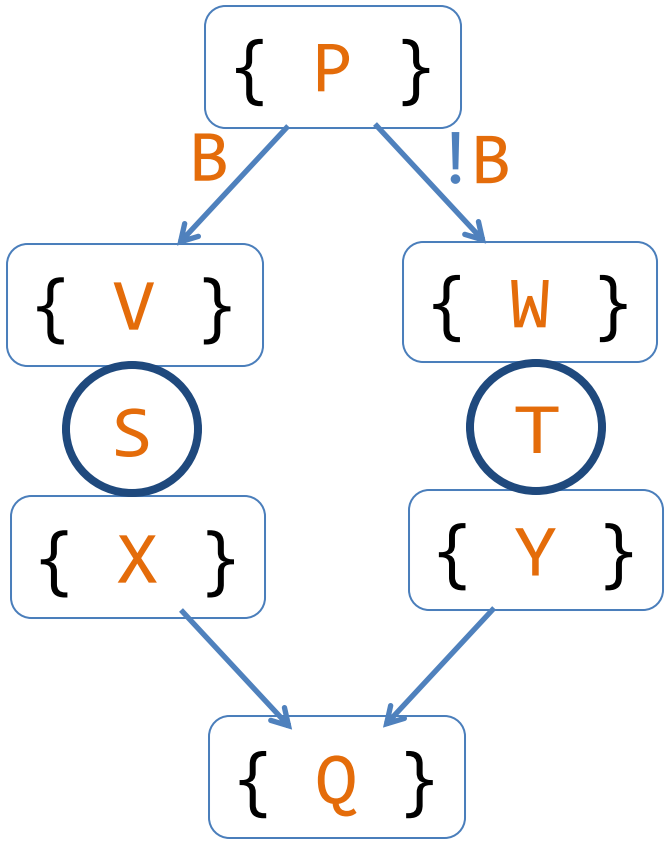
Weakest precondition

let Q = WP [T, R]

WP [S;T, R] = WP [S, Q] = WP [S, WP [T, R]]

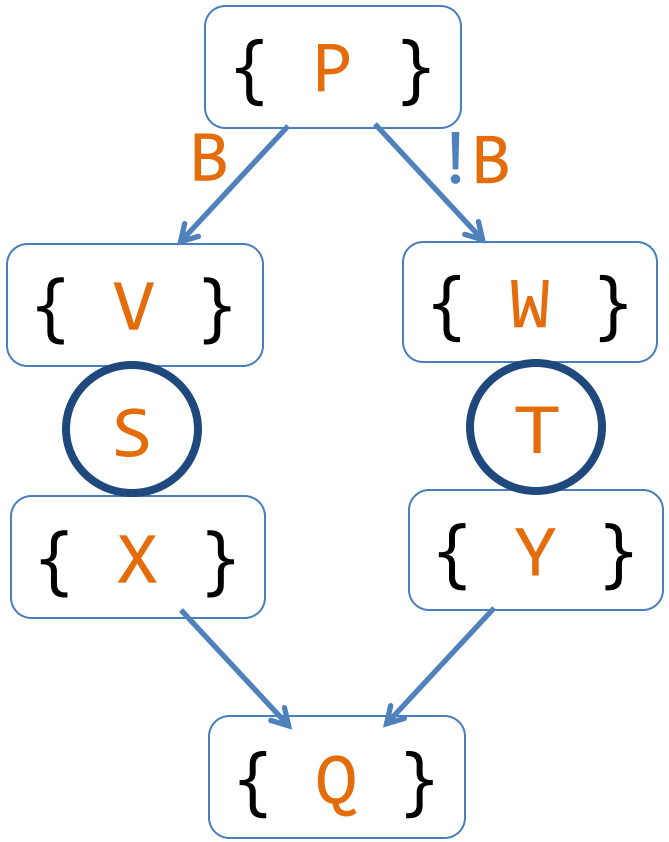
Conditional control flow

```
if B { S } else { T }
```



Conditional control flow

```
if B { S } else { T }
```

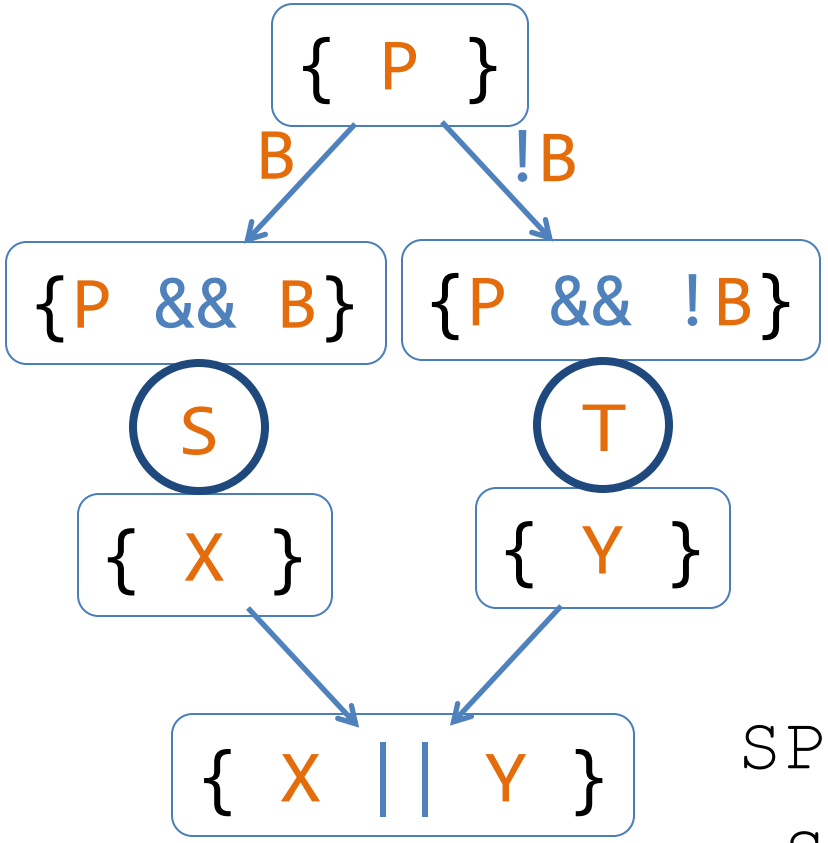


Floyd-Hoare logic tells us:

1. `P && B ==> V`
2. `P && !B ==> W`
3. `{ V } S { X }`
4. `{ W } T { Y }`
5. `X ==> Q`
6. `Y ==> Q`

Strongest postcondition

```
if B { S } else { T }
```



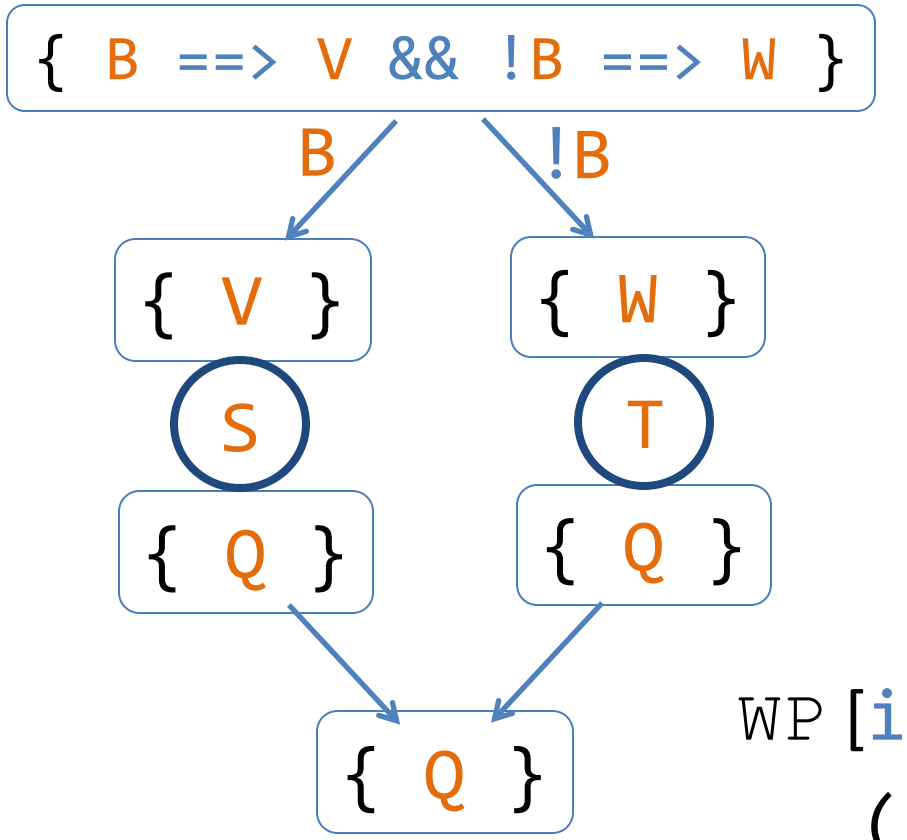

```
X = SP [S, P && B]

Y = SP [T, P && !B]
```

```
SP [if B { S } else { T }, P] =
  SP [S, P && B] || SP [T, P && !B]
```

Weakest precondition

```
if B { S } else { T }
```



$V = WP[S, Q]$

$W = WP[T, Q]$

$$WP[\texttt{if B \{ S \} else \{ T \}}, Q] = (B \Longrightarrow WP[S, Q]) \wedge (\neg B \Longrightarrow WP[T, Q])$$

Weakest precondition (example)

```
if x < 3 {

   x, y := x + 1, 10;

} else {

   y := x;

}
{ x + y == 100 }
```

Weakest precondition (example)

```
if x < 3 {

   x, y := x + 1, 10;

} else {

   y := x;
   { x + y == 100 }
}
{ x + y == 100 }
```

Weakest precondition (example)

```
if x < 3 {

   x, y := x + 1, 10;

} else {

   { x + x == 100 }
   y := x;
   { x + y == 100 }
}
{ x + y == 100 }
```

Weakest precondition (example)

```
if x < 3 {

   x, y := x + 1, 10;

} else {
   { x == 50 }
   { x + x == 100 }
   y := x;
   { x + y == 100 }
}
{ x + y == 100 }
```

Weakest precondition (example)

```
if x < 3 {
    { x == 89 }
    { x + 1 + 10 == 100 }
    x, y := x + 1, 10;
    { x + y == 100 }
} else {
    { x == 50 }
    { x + x == 100 }
    y := x;
    { x + y == 100 }
}
{ x + y == 100 }
```

Weakest precondition (example)

```
{ (x < 3 ==> x == 89) && (x >= 3 ==> x == 50) }
if x < 3 {
   { x == 89 }
   { x + 1 + 10 == 100 }
   x, y := x + 1, 10;
   { x + y == 100 }
} else {
   { x == 50 }
   { x + x == 100 }
   y := x;
   { x + y == 100 }
}
{ x + y == 100 }
```

Weakest precondition (example)

```
{ x == 50 } { (x < 3 ==> x == 89) && (x >= 3 ==> x == 50) }
            if x < 3 {
               { x == 89 }
               { x + 1 + 10 == 100 }
               x, y := x + 1, 10;
               { x + y == 100 }
            } else {
               { x == 50 }
               { x + x == 100 }
               y := x;
               { x + y == 100 }
            }
            { x + y == 100 }
```

Refresher: Implication properties

```
A ==> B    equiv. to    !A || B
```

Hence,

A ==> true	equiv. to	true
A ==> false	"	! A
true ==> B	"	B
false ==> B	"	true

Useful law for simplifying predicates

```
A ==> (B ==> C)    equiv. to    (A && B) ==> C
```

Weakest precondition (example)

```
{ (x < 3 ==> x == 89) && (x >= 3 ==> x == 50) }
if x < 3 {
   x, y := x + 1, 10;
} else {
   y := x;
}
{ x + y == 100 }
```

Weakest precondition (example)

```
{ (x >= 3 || x == 89) && (x < 3 || x ==50) }
{ (x < 3 ==> x == 89) && (x >= 3 ==> x == 50) }
if x < 3 {
   x, y := x + 1, 10;
} else {
   y := x;
}
{ x + y == 100 }
```

Weakest precondition (example)

```
{ (x >= 3 && x < 3) || (x >= 3 && x == 50) ||
  (x == 89 && x < 3) || (x == 89 && x == 50) }
{ (x >= 3 || x == 89) && (x < 3 || x == 50) }
{ (x < 3 ==> x == 89) && (x >= 3 ==> x == 50) }
if x < 3 {
   x, y := x + 1, 10;
} else {
   y := x;
}
{ x + y == 100 }
```

Weakest precondition (example)

```
{ false || x == 50 || false || false }
{ (x >= 3 && x < 3) || (x >= 3 && x == 50) ||
  (x == 89 && x < 3) || (x == 89 && x == 50) }
{ (x >= 3 || x == 89) && (x < 3 || x == 50) }
{ (x < 3 ==> x == 89) && (x >= 3 ==> x == 50) }
if x < 3 {
   x, y := x + 1, 10;
} else {
   y := x;
}
{ x + y == 100 }
```

Weakest precondition (example)

```
{ x == 50 }
{ false || x == 50 || false || false }
{ (x >= 3 && x < 3) || (x >= 3 && x == 50) ||
  (x == 89 && x < 3) || (x == 89 && x == 50) }
{ (x >= 3 || x == 89) && (x < 3 || x == 50) }
{ (x < 3 ==> x == 89) && (x >= 3 ==> x == 50) }
if x < 3 {
   x, y := x + 1, 10;
} else {
   y := x;
}
{ x + y == 100 }
```

Method correctness

Given

```
method M(x: Tx) returns (y: Ty)
   requires P
   ensures Q
{
   B
}
```

we need to prove

```
   P ==> WP[B, Q]
```

Method calls

Methods are *opaque*, i.e., *we reason in terms of their specifications*, **not** their implementations

Given

```
method Triple(x: int) returns (y: int)
  ensures y == 3 * x
```

we expect to be able to prove, for instance, the following method call

```
{ true } v := Triple(u + 4) { v == 3 * (u + 4) }
```

Parameters

We need to relate the actual parameters (of the method call) with the formal parameters (of the method)

To avoid any name clashes, we first rename the formal parameters to fresh variables:

```
method Triple(x': int) returns (y': int)
  ensures y' == 3 * x'
```

Then, for a call `v := Triple(u + 1)` we have

```
x' := u + 1
v  := y'
```

Assumptions

The caller can assume that the method's postcondition holds

We introduce a new statement, `assume E`, to capture this

```
SP[assume E, P] = E && P

WP[assume E, Q] = E ==> Q
```

The semantics of `v := Triple(u + 1)` is then given by

```
var x'; var y';
x' := u + 1;
assume y' == 3 * x';
v := y'
```

```
method Triple(x': int)
returns (y': int)
  ensures y' == 3 * x'
```

Weakest precondition

```
WP [r := M(E), Q] = forall y':: R[x,y\E,y'] ==> Q[r\y']
```

where x is M's input, y is M's output, and R is M's postcondition

Example. Let Q be v == 48 for the method:

```
method Triple(x: int) returns (y: int)
  ensures y == 3 * x
```

```
{ u == 15 }
{ 3 * (u + 1) == 48 }
{ forall y' :: y' == 3 * (u + 1) ==> y' == 48 }
v := Triple(u + 1);
{ v == 48 }
```

Assertions

`assert E` does nothing when `E` holds,
otherwise it crashes the program

```
method Triple(x: int) returns (r: int) {
  var y := 2 * x;
  r := x + y;
  assert r == 3 * x;
}
```

```
WP[assert E, Q] = E && Q
```

```
SP[assert E, P] = P && E
```

Method calls with preconditions

Given

```
method M(x: X) returns (y: Y)
  requires P
  ensures R
```

The semantics of `r := M(E)` is

```
var x_E ; var y_r ;
x_E := E ;
assert P[x\x_E] ;
assume R[x,y\x_E,y_r] ;
r := y_r
```

```
WP[r := M(E), Q] = P[x\E] && forall y_r :: R[x,y\E,y_r] ==> Q[r\y_r]
```

Loops in Velvet

```
while G
  invariant J
  decreases M
  do
    Body
```

G: *loop guard*, Boolean expression

M: (optional) *termination measure*, expression whose value is expected to decrease at each loop iteration

J: loop invariant, condition expected to hold at each iteration

Hoare triples for loops

```
{ J }
while G
  invariant J
{ J && !G }
```

Example

```
r:= 0;
N := 104;
while (r+1)*(r+1) <= N
  invariant 0 <= r && r*r <= N
assert 0 <= r && r*r <= N < (r+1)*(r+1);
```

Floyd-Hoare logic for loop body

For a loop

```
while G
  invariant J
  do
    Body
```

we need to prove

```
{ J && G }
Body
{ J }
```

Quotient modulus

```
x := 0;
y := 191;
while !(y < 7)
  invariant 0 <= y && 7*x + y == 191
assert x == 191 / 7 && y == 191 % 7;
```

Quotient modulus

```
x := 0;
y := 191;
while !(y < 7)
invariant 0 <= y && 7*x + y == 191
do
```

```
assert x == 191 / 7 && y == 191 % 7;
```

Quotient modulus

```
x := 0;
y := 191;
while !(y < 7)
invariant 0 <= y && 7*x + y == 191
do
  { 0 <= y && 7*x + y == 191 && 7 <= y }




  { 0 <= y && 7*x + y == 191 }

assert x == 191 / 7 && y == 191 % 7;
```

Quotient modulus

```
x := 0;
y := 191;
while !(y < 7)
invariant 0 <= y && 7*x + y == 191
do
  { 0 <= y && 7*x + y == 191 && 7 <= y }


  x := x + 1;
  { 0 <= y && 7*x + y == 191 }

assert x == 191 / 7 && y == 191 % 7;
```

Quotient modulus

```
x := 0;
y := 191;
while !(y < 7)
invariant 0 <= y && 7*x + y == 191
do
  { 0 <= y && 7*x + y == 191 && 7 <= y }



  { 0 <= y && 7*(x + 1) + y == 191 }
  x := x + 1;
  { 0 <= y && 7*x + y == 191 }

assert x == 191 / 7 && y == 191 % 7;
```

Quotient modulus

```
x := 0;
y := 191;
while !(y < 7)
invariant 0 <= y && 7*x + y == 191
do
  { 0 <= y && 7*x + y == 191 && 7 <= y }



  { 0 <= y && 7*x + 7 + y == 191 }
  { 0 <= y && 7*(x + 1) + y == 191 }
  x := x + 1;
  { 0 <= y && 7*x + y == 191 }

assert x == 191 / 7 && y == 191 % 7;
```

Quotient modulus

```
x := 0;
y := 191;
while !(y < 7)
invariant 0 <= y && 7*x + y == 191
do
  { 0 <= y && 7*x + y == 191 && 7 <= y }

  y := y - 7;
  { 0 <= y && 7*x + 7 + y == 191 }
  { 0 <= y && 7*(x + 1) + y == 191 }
  x := x + 1;
  { 0 <= y && 7*x + y == 191 }

assert x == 191 / 7 && y == 191 % 7;
```

Full program

```
x := 0;
y := 191;
while !(y < 7)
invariant 0 <= y && 7*x + y == 191
do
  { 0 <= y && 7*x + y == 191 && 7 <= y }
  { 0 <= y - 7 && 7*x + 7 + (y - 7) == 191 }
  y := y - 7;
  { 0 <= y && 7*x + 7 + y == 191 }
  { 0 <= y && 7*(x + 1) + y == 191 }
  x := x + 1;
  { 0 <= y && 7*x + y == 191 }

assert x == 191 / 7 && y == 191 % 7;
```

Leap to the answer

```
x := 0;
y := 191;
while !(y < 7)
invariant 0 <= y && 7*x + y == 191
do
  { 0 <= y && 7 * x + y == 191 && 7 <= y }


  x := 27
  y := 2
  { 0 <= y && 7 * x + y == 191 }

assert x == 191 / 7 && y == 191 % 7;
```

Leap to the answer

```
x := 0;
y := 191;
while !(y < 7)
invariant 0 <= y && 7*x + y == 191
do
  { 0 <= y && 7 * x + y == 191 && 7 <= y }
  { true }
  { 0 <= 2 && 7 * 27 + 2 == 191 }
  x := 27
  y := 2
  { 0 <= y && 7 * x + y == 191 }

assert x == 191 / 7 && y == 191 % 7;
```

Computing sums

```
while n != 33
  invariant s == n * (n - 1) / 2
  do
  { s == n * (n - 1) / 2 && n != 33 }
```

```
  { s == n * (n - 1) / 2 }

assert s == 33 * 32 / 2;
```

Computing sums

```
while n != 33
  invariant s == n * (n - 1) / 2
  do
  { s == n * (n - 1) / 2 && n != 33 }
```

```
  n := n + 1;
  { s == n * (n - 1) / 2 }

assert s == 33 * 32 / 2;
```

Computing sums

```
while n != 33
  invariant s == n * (n - 1) / 2
  do
  { s == n * (n - 1) / 2 && n != 33 }
```

```
  { s == (n + 1) * (n + 1 - 1) / 2 }
  n := n + 1;
  { s == n * (n - 1) / 2 }

assert s == 33 * 32 / 2;
```

Computing sums

```
while n != 33
  invariant s == n * (n - 1) / 2
  do
  { s == n * (n - 1) / 2 && n != 33 }




  { s == (n + 1) * n / 2 }
  { s == (n + 1) * (n + 1 - 1) / 2 }
  n := n + 1;
  { s == n * (n - 1) / 2 }

assert s == 33 * 32 / 2;
```

Computing sums

```
while n != 33
  invariant s == n * (n - 1) / 2
  do
  { s == n * (n - 1) / 2 && n != 33 }









  { s == (n*n + n) / 2 }
  { s == (n + 1) * n / 2 }
  { s == (n + 1) * (n + 1 - 1) / 2 }
  n := n + 1;
  { s == n * (n - 1) / 2 }

assert s == 33 * 32 / 2;
```

Computing sums

```
while n != 33
  invariant s == n * (n - 1) / 2
  do
  { s == n * (n - 1) / 2 && n != 33 }




  { s == (n*n - n + 2*n) / 2 }
  { s == (n*n + n) / 2 }
  { s == (n + 1) * n / 2 }
  { s == (n + 1) * (n + 1 - 1) / 2 }
  n := n + 1;
  { s == n * (n - 1) / 2 }

assert s == 33 * 32 / 2;
```

Computing sums

```
while n != 33
  invariant s == n * (n - 1) / 2
  do
  { s == n * (n - 1) / 2 && n != 33 }



  { s = (n*n – n) / 2 + 2*n / 2 }
  { s == (n*n – n + 2*n) / 2 }
  { s == (n*n + n) / 2 }
  { s == (n + 1) * n / 2 }
  { s == (n + 1) * (n + 1 - 1) / 2 }
  n := n + 1;
  { s == n * (n - 1) / 2 }

assert s == 33 * 32 / 2;
```

Computing sums

```
while n != 33
  invariant s == n * (n - 1) / 2
  do
  { s == n * (n - 1) / 2 && n != 33 }


  { s = n * (n - 1) / 2 + n }
  { s = (n*n – n) / 2 + 2*n / 2 }
  { s == (n*n – n + 2*n) / 2 }
  { s == (n*n + n) / 2 }
  { s == (n + 1) * n / 2 }
  { s == (n + 1) * (n + 1 - 1) / 2 }
  n := n + 1;
  { s == n * (n - 1) / 2 }

assert s == 33 * 32 / 2;
```

Computing sums

```
while n != 33
  invariant s == n * (n - 1) / 2
  do
  { s == n * (n - 1) / 2 && n != 33 }

  s := s + n;
  { s = n * (n - 1) / 2 + n }
  { s = (n*n – n) / 2 + 2*n / 2 }
  { s == (n*n – n + 2*n) / 2 }
  { s == (n*n + n) / 2 }
  { s == (n + 1) * n / 2 }
  { s == (n + 1) * (n + 1 - 1) / 2 }
  n := n + 1;
  { s == n * (n - 1) / 2 }

assert s == 33 * 32 / 2;
```

Computing sums

```
while n != 33
  invariant s == n * (n - 1) / 2
  do
  { s == n * (n - 1) / 2 && n != 33 }
  { s == n * (n - 1) / 2 }
  s := s + n;
  { s = n * (n - 1) / 2 + n }
  { s = (n*n – n) / 2 + 2*n / 2 }
  { s == (n*n – n + 2*n) / 2 }
  { s == (n*n + n) / 2 }
  { s == (n + 1) * n / 2 }
  { s == (n + 1) * (n + 1 - 1) / 2 }
  n := n + 1;
  { s == n * (n - 1) / 2 }

assert s == 33 * 32 / 2;
```

Full program

Need to choose initial values of s and n to establish invariant

```
s := 0;
n := 0;
while n != 33
  invariant s == n * (n - 1) / 2
  do
   s := s + n;
   n := n + 1;
```

Reasoning about Termination

Loop termination

For a loop

```
while G
   invariant J
   decreasing D
   do
     Body
```

we need to prove

```
{ J && G }
ghost var d := D;
Body
{ D < d && 0 <= D }
```

Ghost variables are for reasoning only. They are not part of the compiled code.

Termination of quotient modulus

```
x, y := 0, 191;
while 7 <= y
  invariant 0 <= y && 7 * x + y == 191
  decreasing y
  do
   y := y - 7;
   x := x + 1;
```

```
{ 0 <= y && 7 * x + y == 191 && 7 <= y }
ghost var d := y;
y := y - 7;
x := x + 1;
{ d > y && y >= 0 }
```

- `y < d` follows from `y := y - 7`
- `0 <= d` follows from `0 <= y` in invariant

Quick body

```
x, y := 0, 191;
while 7 <= y
  invariant 0 <= y && 7 * x + y == 191
  decreasing y
  do
   y := 2;
   x := 27;
```

```
{ 0 <= y && 7 * x + y == 191 && 7 <= y }
ghost var d := y;
y := 2;
x := 27;
{ d > y && y >= 0 }
```

- `y < d` follows from `7 <= y` in invariant
- `0 <= d` follows from `0 <= y` in invariant

Complete loop rule

```
{ J }
while G
   invariant J
   decreasing D
   do
    Body

{ J && !G }
```

```
{ J && G }
ghost var d := D;
Body
{ J && d > D && D >= 0 }
```

Integer square root

```
method SquareRoot(N: Nat) return (r: Nat)
  ensures r*r <= N < (r+1)*(r+1)
```

Integer square root

```
method SquareRoot(N: Nat) return (r: Nat)
  ensures r*r <= N && N < (r+1)*(r+1)
```

Loop design pattern
For a postcondition `A && B`, use
A as the invariant and `!B` as the guard

```
do

  while (r+1)*(r+1) <= N
    invariant r*r <= N
```

Integer square root

```
method SquareRoot(N: Nat) return (r: Nat)
  ensures r*r <= N && N < (r+1)*(r+1)
```

Loop design pattern

For a postcondition `A && B`, use

A as the invariant and `!B` as the guard

```
do
 let mut r := 0;
 while (r+1)*(r+1) <= N
   invariant r*r <= N
   do
     r := r + 1
```

A more efficient algorithm

Rather than calculate `(r + 1)*(r + 1)` on each iteration, add a new variable `s` and maintain invariant

```
s == (r + 1)*(r + 1)
```

A more efficient algorithm

Rather than calculate `(r + 1)*(r + 1)` on each iteration add a new variable `s == (r + 1)*(r + 1)`

Then we have `s` initially `1`, loop guard `s <= N` and invariant `s == (r + 1)*(r + 1)`

```
{ s == (r + 1)*(r + 1) }
```

```
{ s == (r + 1 + 1)*(r + 1 + 1)}
r := r + 1
{ s == (r + 1)*(r + 1) }
```

A more efficient algorithm

Rather than calculate `(r + 1)*(r + 1)` on each iteration add new variable `s == (r + 1)*(r + 1)`

Then we have `s` initially `1`, loop guard `s <= N` and invariant `s == (r + 1)*(r + 1)`

```
{ s == (r + 1)*(r + 1) }
```

```
{ s == r*r + 4*r + 4}
{ s == (r + 1 + 1)*(r + 1 + 1)}
r := r + 1
{ s == (r + 1)*(r + 1) }
```

A more efficient algorithm

Rather than calculate `(r + 1)*(r + 1)` on each iteration add new variable `s == (r + 1)*(r + 1)`

Then we have `s` initially `1`, loop guard `s <= N` and invariant `s == (r + 1)*(r + 1)`

```
{ s == (r + 1)*(r + 1) }
```

```
{ s == r*r + 2*r + 1 + 2*r + 3 }
{ s == r*r + 4*r + 4}
{ s == (r + 1 + 1)*(r + 1 + 1)}
r := r + 1
{ s == (r + 1)*(r + 1) }
```

A more efficient algorithm

Rather than calculate `(r + 1)*(r + 1)` on each iteration add new variable `s == (r + 1)*(r + 1)`

Then we have `s` initially `1`, loop guard `s <= N` and invariant `s == (r + 1)*(r + 1)`

```
{ s == (r + 1)*(r + 1) }
```

```
{ s == (r + 1)*(r + 1) + 2*r + 3}
{ s == r*r + 2*r + 1 + 2*r + 3 }
{ s == r*r + 4*r + 4}
{ s == (r + 1 + 1)*(r + 1 + 1)}
r := r + 1
{ s == (r + 1)*(r + 1) }
```

A more efficient algorithm

Rather than calculate `(r + 1)*(r + 1)` on each iteration add new variable `s == (r + 1)*(r + 1)`

Then we have `s` initially `1`, loop guard `s <= N` and invariant `s == (r + 1)*(r + 1)`

```
{ s == (r + 1)*(r + 1) }

s := s + 2*r + 3;
{ s == (r + 1)*(r + 1) + 2*r + 3}
{ s == r*r + 2*r + 1 + 2*r + 3 }
{ s == r*r + 4*r + 4}
{ s == (r + 1 + 1)*(r + 1 + 1)}
r := r + 1
{ s == (r + 1)*(r + 1) }
```

A more efficient algorithm

Rather than calculate `(r + 1)*(r + 1)` on each iteration add new variable `s == (r + 1)*(r + 1)`

Then we have `s` initially `1`, loop guard `s <= N` and invariant `s == (r + 1)*(r + 1)`

```
{ s == (r + 1)*(r + 1) }
{ s + 2*r + 3 == (r + 1)*(r + 1) + 2*r + 3 }
s := s + 2*r + 3;
{ s == (r + 1)*(r + 1) + 2*r + 3}
{ s == r*r + 2*r + 1 + 2*r + 3 }
{ s == r*r + 4*r + 4}
{ s == (r + 1 + 1)*(r + 1 + 1)}
r := r + 1
{ s == (r + 1)*(r + 1) }
```

Full program

```
method SquareRoot(N: Nat) return (r: Nat)
   ensures r*r <= N < (r+1)*(r+1)
   do
   let mut r := 0;
   var s := 1;
   while s <= N
      invariant r*r <= N
      invariant s == (r+1)*(r+1)
      do
       s := s + 2*r + 3;
       r := r + 1;
```

More examples

Iterative Fibonacci

```
def Fib (n: Nat) :=  match n with
  | 0 => 0
  | 1 => 1
  | n + 2 => Fib (n + 1) + Fib n

method ComputeFib(n: nat) returns (x: nat)
  ensures x = Fib(n)
  do
  x := 0;
  var i := 0;
  while i != n
    invariant 0 <= i <= n
    invariant x = Fib(i)
```

Iterative Fibonacci

Loop design technique (*Replace a constant by a variable)*
For a loop to establish a condition P[E], where E is an expression that maintains a constant value throughout the loop,

- use a variable i that the loop changes until it equals E, and
- make P[i] a loop invariant

Example: to establish `x = Fib(n)` introduce `i` and

```
invariant x = Fib(i)
```

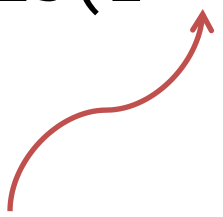
Iterative Fibonacci

```
method ComputeFib(n: nat) returns (x: nat)
  ensures x == Fib(n)
  do
  let mut x := 0
  let mut y := 1
  let mut i := 0
  while i != n
    invariant 0 <= i && i <= n
    invariant x = Fib(i)
    do
    ...
    i := i + 1;

}
```

Iterative Fibonacci

```
method ComputeFib(n: nat) returns (x: nat)
  ensures x == Fib(n)
  do
  let mut x := 0
  let mut y := 1
  let mut i := 0
  while i != n
    invariant 0 <= i && i <= n
    invariant x = Fib(i)
    invariant x == Fib(i) && y == Fib(i + 1)
    do
    ...
     i := i + 1;
```



Cannot use `y == Fib(i-1)`
as not defined when `i == 0`

Loop body

```
{ 0 <= i <= n  && x == Fib(i)
               && y == Fib(i+1) &&  i != n}
```

```
i := i + 1;
{ 0 <= i <= n && x == Fib(i) && y == Fib(i+1) }
```

Loop body

```
{ 0 <= i <= n  && x == Fib(i)
               && y == Fib(i+1) &&  i != n}
```

```
{ 0 <= i+1 <= n && x == Fib(i+1)
                && y == Fib(i+1+1) }
i := i + 1;
{ 0 <= i <= n && x == Fib(i) && y == Fib(i+1) }
```

Loop body

```
{ 0 <= i <= n  && x == Fib(i)
               && y == Fib(i+1) &&  i != n}
```

```
{ 0 <= i+1 <= n && x == Fib(i+1) && y == Fib(i+2)}
{ 0 <= i+1 <= n && x == Fib(i+1)
                && y == Fib(i+1+1) }
i := i + 1;
{ 0 <= i <= n && x == Fib(i) && y == Fib(i+1) }
```

Loop body

```
{ 0 <= i <= n  && x == Fib(i)
               && y == Fib(i+1) &&  i != n}
```

```
{ 0 <= i+1 <= n && x == Fib(i+1)
                && y == Fib(i) + Fib(i+1) }
{ 0 <= i+1 <= n && x == Fib(i+1) && y == Fib(i+2)}
{ 0 <= i+1 <= n && x == Fib(i+1)
                && y == Fib(i+1+1) }
i := i + 1;
{ 0 <= i <= n && x == Fib(i) && y == Fib(i+1) }
```

Loop body

```
{ 0 <= i <= n  && x == Fib(i)
               && y == Fib(i+1) &&  i != n}
```

```
let tmp := x; x := y; y := tmp + y
{ 0 <= i+1 <= n && x == Fib(i+1)
                && y == Fib(i) + Fib(i+1) }
{ 0 <= i+1 <= n && x == Fib(i+1) && y == Fib(i+2)}
{ 0 <= i+1 <= n && x == Fib(i+1)
                && y == Fib(i+1+1) }
i := i + 1;
{ 0 <= i <= n && x == Fib(i) && y == Fib(i+1) }
```

Loop body

```
{ 0 <= i <= n  && x == Fib(i)
               && y == Fib(i+1) &&  i != n}


{ 0 <= i+1 <= n && y == Fib(i+1)
               && x+y == Fib(i) + Fib(i+1) }
let tmp := x; x := y; y := tmp + y
{ 0 <= i+1 <= n && x == Fib(i+1)
               && y == Fib(i) + Fib(i+1) }
{ 0 <= i+1 <= n && x == Fib(i+1) && y == Fib(i+2)}
{ 0 <= i+1 <= n && x == Fib(i+1)
               && y == Fib(i+1+1) }
i := i + 1;
{ 0 <= i <= n && x == Fib(i) && y == Fib(i+1) }
```

Loop body

```
{ 0 <= i <= n  && x == Fib(i)
               && y == Fib(i+1) &&  i != n}

{ 0 <= i+1 <= n && x == Fib(i) && y == Fib(i+1) }
{ 0 <= i+1 <= n && y == Fib(i+1)
                && x+y == Fib(i) + Fib(i+1) }
let tmp := x; x := y; y := tmp + y
{ 0 <= i+1 <= n && x == Fib(i+1)
                && y == Fib(i) + Fib(i+1) }
{ 0 <= i+1 <= n && x == Fib(i+1) && y == Fib(i+2)}
{ 0 <= i+1 <= n && x == Fib(i+1)
                && y == Fib(i+1+1) }
i := i + 1;
{ 0 <= i <= n && x == Fib(i) && y == Fib(i+1) }
```

Loop body

```
{ 0 <= i <= n  && x == Fib(i)
               && y == Fib(i+1) &&  i != n}
{ 0 <= i <= n   && x == Fib(i) && y == Fib(i+1) }
{ 0 <= i+1 <= n && x == Fib(i) && y == Fib(i+1) }
{ 0 <= i+1 <= n && y == Fib(i+1)
                && x+y == Fib(i) + Fib(i+1) }
let tmp := x; x := y; y := tmp + y
{ 0 <= i+1 <= n && x == Fib(i+1)
                && y == Fib(i) + Fib(i+1) }
{ 0 <= i+1 <= n && x == Fib(i+1) && y == Fib(i+2)}
{ 0 <= i+1 <= n && x == Fib(i+1)
                && y == Fib(i+1+1) }
i := i + 1;
{ 0 <= i <= n && x == Fib(i) && y == Fib(i+1) }
```

Full program

```
method ComputeFib(n: Nat) returns (x: Nat)
  ensures x = Fib(n)
  do
  let mut x := 0
  let mut y := 1
  let mut i := 0;
  while i != n
    invariant 0 <= i && i <= n
    invariant x = Fib(i)
    invariant y = Fib(i + 1)
    do
      let tmp := x; x := y; y := tmp + y
      i := i + 1
```