# CS 591: Formal Methods in Security and Privacy:
# An imperative programming language

Marco Gaboardi
gaboardi@bu.edu

Alley Stoughton
stough@bu.edu

# From the previous class

# Does the program comply with the specification?

Precondition: $x \geq 0$ and $y \geq 0$

```
Function Add(x: int, y: int) : int
{
  r = 0;
  n = y;
  while n != 0
  {
    r = r + 1;
    n = n - 1;
  }
  return r
}
```

Postcondition: r == x + y

# Does the program comply with the specification?

Precondition: x ≥ 0 and y ≥ 0

```
Function Add(x: int, y: int) : int
{
  r = 0;
  n = y;
  while n != 0
  {
    r = r + 1;
    n = n - 1;
  }
  return r
}
```

Postcondition: r == x + y

Fail to meet the specification

# How about this one?

Precondition: x ≥ 0 and y ≥ 0

```
Function Add(x: int, y: int) : int
{
  r = x;
  n = y;
  while n != 0
  {
    r = r + 1;
    n = n - 1;
  }
  return r
}
```

Postcondition: r == x + y

# How about this one?

Precondition: x ≥ 0 and y ≥ 0

```
Function Add(x: int, y: int) : int
{
  r = x;
  n = y;
  while n != 0
  {
    r = r + 1;
    n = n - 1;
  }
  return r
}
```

Postcondition: r == x + y

It meets the specification

How can we make this reasoning mathematically precise?

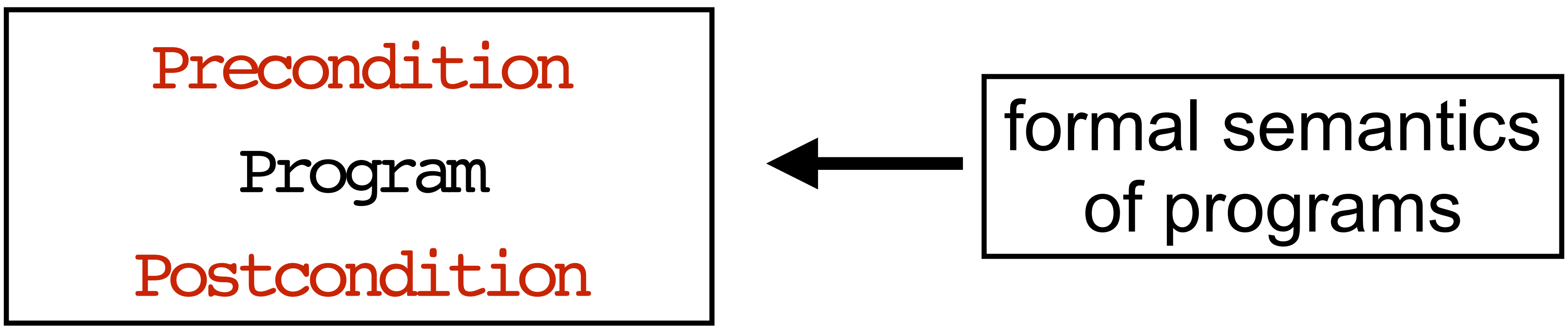
# Formal Semantics

We need to assign a formal meaning to the different components:

| |
|---|
| `Precondition` |
| `Program` |
| `Postcondition` |

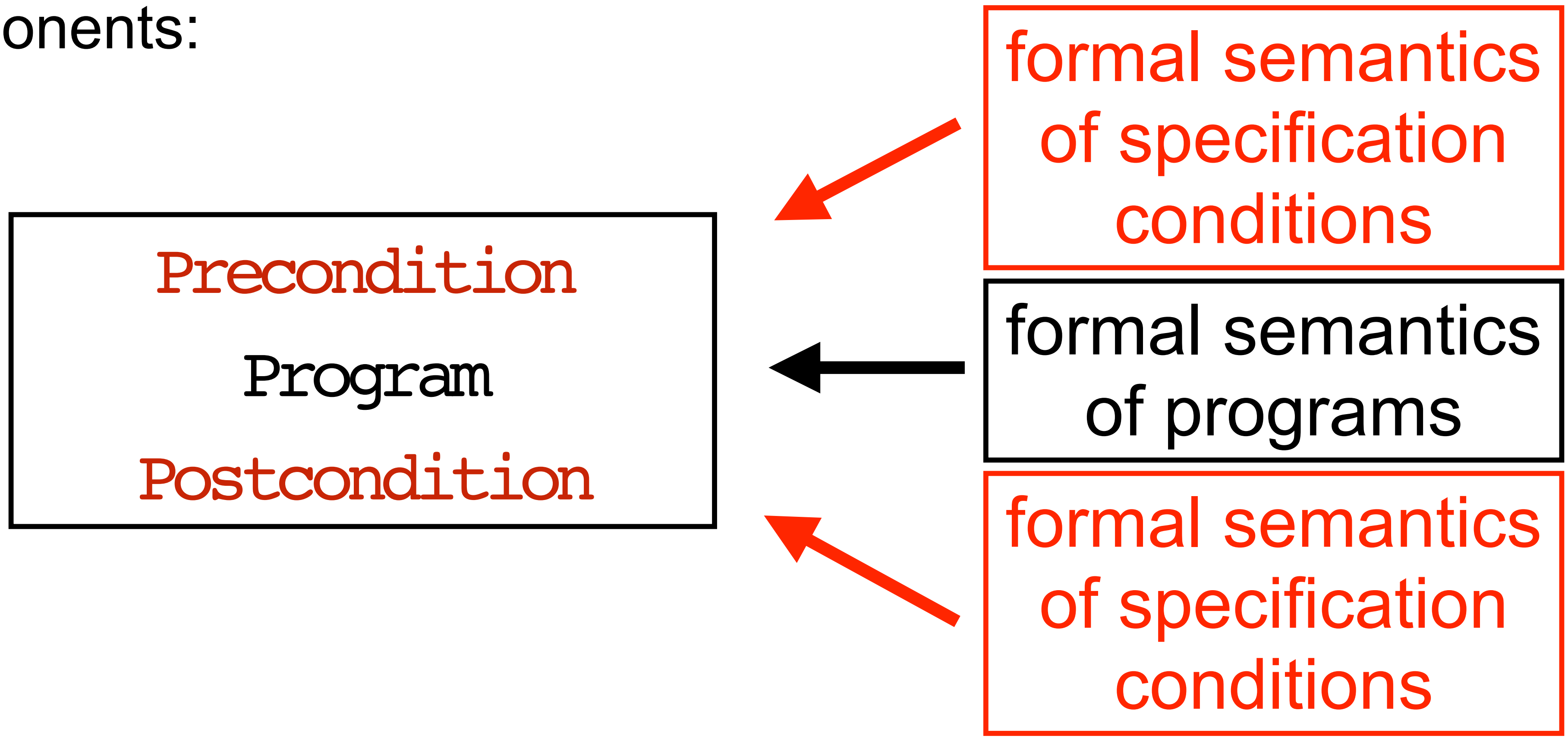
# Formal Semantics

We need to assign a formal meaning to the different components:



# Formal Semantics

We need to assign a formal meaning to the different components:



# Formal Semantics

We need to assign a formal meaning to the different components:

Precondition

Program

Postcondition

formal semantics of specification conditions

formal semantics of programs

formal semantics of specification conditions

We also need to describe the rules which combine program and specifications.

# Goal for today

- Formalize the semantics of a simple imperative programming language.

# A first example

```
FastExponentiation(n, k : Nat) : Nat
  n′:= n; k′:= k; r := 1;
  if k′ > 0 then
    while k′ > 1 do
      if even(k′) then
        n′ := n′ ∗ n′;
        k′ := k′/2;
      else
        r := n′ ∗ r;
        n′ := n′ ∗ n′;
        k′ := (k′ − 1)/2;
    r := n′ ∗ r;
  (* result is r *)
```

# Programming Language

```
c::= abort
   | skip
   | x:=e
   | c;c
   | if e then c else c
   | while e do c
```

x, y, z, ... program variables

$e_1, e_2, \ldots$ expressions

$c_1, c_2, \ldots$ commands

How would you describe the meaning of a program in a mathematically precise way?

# Expressions

We want to be able to write complex programs with our language.

```
e::= x
   | f(e1,…,en)
```

Where `f` can be any arbitrary operator.

Some expression examples

`x+5`  `x mod k`  `x[i]`  `(x[i+1] mod 4)+5`

# Types

In expressions we want to be able to use "arbitrary" data types.

```
t::= b
   | T(t₁,…,tₙ)
```

# Types

In expressions we want to be able to use "arbitrary" data types.

```
t::= b
   | T(t1,…,tn)
```

We assume a collection of base types `b` including

`Bool`    `Int`    `Nat`    `String`

We also assume a set of type constructors `T` that we can use to build more complex types, such as:

`Bool list`    `Int*Bool`    `Int*String -> Bool`

# Types

We also use types to guarantee that commands are well-formed.

For example, in the commands

```
while e do c        if e then c1 else c2
```

We require that `e` is of type `Bool`.

# Types

We also use types to guarantee that commands are well-formed.

For example, in the commands

`while e do c`        `if e then` $c_1$ `else` $c_2$

We require that `e` is of type `Bool`.

We omit the details of the type system here but you can find them in the notes by Gilles Barthe

# Values

Values are atomic expressions whose semantics is self-evident and which do not need a further analysis.

For example, we have the following values

```
true     5        [1,2,3,4]       "Hello"
```

The following are not values:

```
not true      x+5       [x,x+1]        x[1]
```

# Values

Values are atomic expressions whose semantics is self-evident and which do not need a further analysis.

For example, we have the following values

`true`    `5`    `[1,2,3,4]`    `“Hello”`

The following are not values:

`not true`    `x+5`    `[x,x+1]`    `x[1]`

We could define a grammar for values, but we prefer to leave this at the intuitive level for now.

# How can we give semantics to expressions and commands?

```
FastExponentiation(n, k : Nat) : Nat
  n′:= n; k′:= k; r := 1;
  if k′ > 0 then
   while k′ > 1 do
      if even(k′) then
        n′ := n′ ∗ n′;
        k′ := k′/2;
     else
       r := n′ ∗ r;
       n′ := n′ ∗ n′;
       k′ := (k′ − 1)/2;
   r := n′ ∗ r;
 (* result is r *)
```

# Memories

We can formalize a memory as a map $\mathtt{m}$ from variables to values.

$$\mathtt{m} = [\mathtt{x}_1 \longmapsto \mathtt{v}_1, \ldots, \mathtt{x}_n \longmapsto \mathtt{v}_n]$$

We consider only maps that respect types.

# Memories

We can formalize a memory as a map $\mathtt{m}$ from variables to values.

$$\mathtt{m} = [\mathtt{x}_1 \longmapsto \mathtt{v}_1, \ldots, \mathtt{x}_n \longmapsto \mathtt{v}_n]$$

We consider only maps that respect types.

We want to read the value associated to a particular variable:

$$\mathtt{m}(\mathtt{x})$$

We want to update the value associated to a particular variable:

$$\mathtt{m}[\mathtt{x} \leftarrow \mathtt{v}]$$

This is defined as

$$\mathtt{m}[\mathtt{x} \leftarrow \mathtt{v}](\mathtt{y}) = \begin{cases} \mathtt{v} & \text{If } \mathtt{x} = \mathtt{y} \\ \mathtt{m}(\mathtt{y}) & \text{Otherwise} \end{cases}$$

# Semantics of Expressions

What is the meaning of the following expressions?

`x+5`        `x mod k`        `x[i]`        `(x[i+1] mod 4)+5`

# Semantics of Expressions

What is the meaning of the following expressions?

`x+5`      `x mod k`      `x[i]`      `(x[i+1] mod 4)+5`

We can give the semantics as a relation between expressions, memories and values.

`Exp * Mem * Val`

We will denote this relation as:

$\{e\}_m = v$

# Semantics of Expressions

What is the meaning of the following expressions?

`x+5`      `x mod k`      `x[i]`      `(x[i+1] mod 4)+5`

We can give the semantics as a relation between expressions, memories and values.

```
Exp * Mem * Val
```

We will denote this relation as:

$\{e\}_m = v$

This is commonly typeset as:

$$[\![e]\!]_m = v$$

# Semantics of Expressions

This is defined on the structure of expressions:

$$\{x\}_m = m(x)$$

$$\{f(e_1,\ldots,e_n)\}_m = \{f\}(\{e_1\}_m,\ldots,\{e_n\}_m)$$

where $\{f\}$ is the semantics associated with the basic operation we are considering.

# Semantics of Expressions

Suppose we have a memory

$$\texttt{m=[i}\longmapsto\texttt{1,x}\longmapsto\texttt{[1,2,3],y}\longmapsto\texttt{2]}$$

That `{mod}` is the modulo operation and `{+}` is addition, we can derive the meaning of the following expression:

# Semantics of Expressions

Suppose we have a memory

$$\mathtt{m=[i\longmapsto 1,x\longmapsto [1,2,3],y\longmapsto 2]}$$

That $\{\mathtt{mod}\}$ is the modulo operation and $\{+\}$ is addition, we can derive the meaning of the following expression:

$\{\mathtt{(x[i+1]\ mod\ y)+5}\}_{\mathtt{m}}$

# Semantics of Expressions

Suppose we have a memory

$$\mathtt{m=[i \longmapsto 1, x \longmapsto [1,2,3], y \longmapsto 2]}$$

That $\{\mathtt{mod}\}$ is the modulo operation and $\{+\}$ is addition, we can derive the meaning of the following expression:

$\{\mathtt{(x[i+1]\ mod\ y)+5}\}_{\mathtt{m}}$

$\quad = \{\mathtt{(x[i+1]\ mod\ y)}\}_{\mathtt{m}}\{+\}\{\mathtt{5}\}_{\mathtt{m}}$

# Semantics of Expressions

Suppose we have a memory

$$\texttt{m=[i}\longmapsto\texttt{1,x}\longmapsto\texttt{[1,2,3],y}\longmapsto\texttt{2]}$$

That {mod} is the modulo operation and {+} is addition, we can derive the meaning of the following expression:

$\texttt{\{(x[i+1] mod y)+5\}}_{\texttt{m}}$

$= \texttt{\{(x[i+1] mod y)\}}_{\texttt{m}}\texttt{\{+\}\{5\}}_{\texttt{m}}$

$= \texttt{(\{x[i+1]\}}_{\texttt{m}}\ \texttt{\{mod\}}\ \texttt{\{y\}}_{\texttt{m}}\texttt{)\{+\}\{5\}}_{\texttt{m}}$

# Semantics of Expressions

Suppose we have a memory

$$m=[i \longmapsto 1, x \longmapsto [1,2,3], y \longmapsto 2]$$

That $\{mod\}$ is the modulo operation and $\{+\}$ is addition, we can derive the meaning of the following expression:

$$\begin{aligned}
&\{(x[i+1] \text{ mod } y)+5\}_m \\
&\qquad = \{(x[i+1] \text{ mod } y)\}_m\{+\}\{5\}_m \\
&\qquad = (\{x[i+1]\}_m \ \{mod\} \ \{y\}_m)\{+\}\{5\}_m \\
&\qquad = (\{x\}_m[\{i\}_m\{+\}\{1\}_m] \ \{mod\} \ \{y\}_m)\{+\}\{5\}_m
\end{aligned}$$

# Semantics of Expressions

Suppose we have a memory

$$m=[i\longmapsto 1, x\longmapsto [1,2,3], y\longmapsto 2]$$

That {mod} is the modulo operation and {+} is addition, we can derive the meaning of the following expression:

$$\{(x[i+1] \text{ mod } y)+5\}_m$$

$$= \{(x[i+1] \text{ mod } y)\}_m\{+\}\{5\}_m$$

$$= (\{x[i+1]\}_m \; \{\text{mod}\} \; \{y\}_m)\{+\}\{5\}_m$$

$$= (\{x\}_m[\{i\}_m\{+\}\{1\}_m] \; \{\text{mod}\} \; \{y\}_m)\{+\}\{5\}_m$$

$$= (\{x\}_m[1\{+\}1] \; \{\text{mod}\} \; 2)\{+\}5$$

# Semantics of Expressions

Suppose we have a memory

$$\texttt{m=[i}\longmapsto\texttt{1,x}\longmapsto\texttt{[1,2,3],y}\longmapsto\texttt{2]}$$

That $\texttt{\{mod\}}$ is the modulo operation and $\texttt{\{+\}}$ is addition, we can derive the meaning of the following expression:

$$
\begin{aligned}
&\texttt{\{(x[i+1] mod y)+5\}}_{\texttt{m}}\\
&\qquad = \texttt{\{(x[i+1] mod y)\}}_{\texttt{m}}\texttt{\{+\}\{5\}}_{\texttt{m}}\\
&\qquad = \texttt{(\{x[i+1]\}}_{\texttt{m}}\texttt{ \{mod\} \{y\}}_{\texttt{m}}\texttt{)\{+\}\{5\}}_{\texttt{m}}\\
&\qquad = \texttt{(\{x\}}_{\texttt{m}}\texttt{[\{i\}}_{\texttt{m}}\texttt{\{+\}\{1\}}_{\texttt{m}}\texttt{] \{mod\} \{y\}}_{\texttt{m}}\texttt{)\{+\}\{5\}}_{\texttt{m}}\\
&\qquad = \texttt{(\{x\}}_{\texttt{m}}\texttt{[1\{+\}1] \{mod\} 2)\{+\}5}\\
&\qquad = \texttt{(\{x\}}_{\texttt{m}}\texttt{[2] \{mod\} 2)\{+\}5}
\end{aligned}
$$

# Semantics of Expressions

Suppose we have a memory

$$m=[i\longmapsto 1, x\longmapsto [1,2,3], y\longmapsto 2]$$

That {mod} is the modulo operation and {+} is addition, we can derive the meaning of the following expression:

```
{(x[i+1] mod y)+5}m
        = {(x[i+1] mod y)}m{+}{5}m
        = ({x[i+1]}m {mod} {y}m){+}{5}m
        = ({x}m[{i}m{+}{1}m] {mod} {y}m){+}{5}m
        = ({x}m[1{+}1] {mod} 2){+}5
        = ({x}m[2] {mod} 2){+}5
        = (2 {mod} 2){+}5 = 0 {+} 5 = 5
```

# Operational vs Denotational Semantics

The style of semantics we are using is denotational, in the sense that we describe the meaning of an expression by means of the value it denotes.

A different approach, more operational in nature, would be to describe the meaning of an expression by means of the value that the expression evaluates to in an abstract machine.

# Semantics of Commands

What is the meaning of the following command?

```
k:=2; z:=x mod k; if z=0 then r:=1 else r:=2
```

# Semantics of Commands

What is the meaning of the following command?

```
k:=2; z:=x mod k; if z=0 then r:=1 else r:=2
```

We can give the semantics as a relation between command, memories and memories or failure.

```
Exp * Mem * Mem
```

# Semantics of Commands

What is the meaning of the following command?

```
k:=2; z:=x mod k; if z=0 then r:=1 else r:=2
```

We can give the semantics as a relation between command, memories and memories or failure.

```
Exp * Mem * Mem
```

Would this work?

# Semantics of Commands

What is the meaning of the following command?

```
k:=2; z:=x mod k; if z=0 then r:=1 else r:=2
```

# Semantics of Commands

What is the meaning of the following command?

```
k:=2; z:=x mod k; if z=0 then r:=1 else r:=2
```

We can give the semantics as a relation between command, memories and memories or failure.

$$\texttt{Exp * Mem * (Mem | }\bot\texttt{)}$$

We will denote this relation as:

$\{c\}_m = m'$ Or $\{c\}_m = \bot$

# Semantics of Commands

What is the meaning of the following command?

```
k:=2; z:=x mod k; if z=0 then r:=1 else r:=2
```

We can give the semantics as a relation between command, memories and memories or failure.

```
Exp * Mem * (Mem | ⊥)
```

We will denote this relation as:

$\{c\}_m = m'$ Or $\{c\}_m = \bot$

This is commonly typeset as:

$$[\![c]\!]_m = m'$$

# Semantics of Commands

This is defined on the structure of commands:

# Semantics of Commands

This is defined on the structure of commands:

$\{\texttt{abort}\}_m = \bot$

# Semantics of Commands

This is defined on the structure of commands:

$$\{\texttt{abort}\}_m = \bot$$

$$\{\texttt{skip}\}_m = m$$

# Semantics of Commands

This is defined on the structure of commands:

$$\{\texttt{abort}\}_m = \bot$$

$$\{\texttt{skip}\}_m = m$$

$$\{x{:=}e\}_m = m[x \leftarrow \{e\}_m]$$

# Semantics of Commands

This is defined on the structure of commands:

$\{\texttt{abort}\}_m = \bot$

$\{\texttt{skip}\}_m = m$

$\{\texttt{x:=e}\}_m = m[x \leftarrow \{e\}_m]$

$\{c;c'\}_m = \{c'\}_{m'}$ If $\{c\}_m = m'$

# Semantics of Commands

This is defined on the structure of commands:

$\{\texttt{abort}\}_m = \bot$

$\{\texttt{skip}\}_m = m$

$\{\texttt{x:=e}\}_m = m[x \leftarrow \{e\}_m]$

$\{c;c'\}_m = \{c'\}_{m'}$ If $\{c\}_m = m'$

$\{c;c'\}_m = \bot$ If $\{c\}_m = \bot$

# Semantics of Commands

This is defined on the structure of commands:

$\{\texttt{abort}\}_m = \bot$

$\{\texttt{skip}\}_m = m$

$\{\texttt{x:=e}\}_m = m[x \leftarrow \{e\}_m]$

$\{c;c'\}_m = \{c'\}_{m'}$ If $\{c\}_m = m'$

$\{c;c'\}_m = \bot$ If $\{c\}_m = \bot$

$\{\texttt{if e then } c_t \texttt{ else } c_f\}_m = \{c_t\}_m$ If $\{e\}_m = \texttt{true}$

# Semantics of Commands

This is defined on the structure of commands:

$\{\texttt{abort}\}_m = \bot$

$\{\texttt{skip}\}_m = m$

$\{\texttt{x:=e}\}_m = m[x \leftarrow \{e\}_m]$

$\{c;c'\}_m = \{c'\}_{m'}$ If $\{c\}_m = m'$

$\{c;c'\}_m = \bot$ If $\{c\}_m = \bot$

$\{\texttt{if e then } c_t \texttt{ else } c_f\}_m = \{c_t\}_m$ If $\{e\}_m = \texttt{true}$

$\{\texttt{if e then } c_t \texttt{ else } c_f\}_m = \{c_f\}_m$ If $\{e\}_m = \texttt{false}$

# Semantics of While

What about while

# Semantics of While

What about while

$$\{\texttt{while e do c}\}_m = \texttt{???}$$

# Semantics of While

If $\{e\}_m = \text{false}$ Then

$$\{\texttt{while e do c}\}_m = m$$

# Semantics of While

If $\{e\}_m$=false Then

$\{\text{while e do c}\}_m = m$

What about when $\{e\}_m$=true ?

# Semantics of While

If $\{\texttt{e}\}_{\texttt{m}}$=`true` Then we would like to have:

$$\{\texttt{while e do c}\}_{\texttt{m}} = \{\texttt{c;while e do c}\}_{\texttt{m}}$$

# Semantics of While

If $\{e\}_m = true$ Then we would like to have:

$$\{\texttt{while e do c}\}_m = \{\texttt{c;while e do c}\}_m$$

Is this well defined?

# Approximating While

We could define the following syntactic approximations of a While statement:

$$\texttt{while}^{n}\ \texttt{e do c}$$

# Approximating While

We could define the following syntactic approximations of a While statement:

$$\texttt{while}^{n}\ \texttt{e do c}$$

This can be defined inductively on n as:

$$\texttt{while}^{0}\ \texttt{e do c = skip}$$

$$\texttt{while}^{n+1}\ \texttt{e do c =}$$
$$\texttt{if e then (c;while}^{n}\ \texttt{e do c) else skip}$$

# Semantics of While

We could go back and try to define the semantics using the approximations:

$\{\texttt{while e do c}\}_m = \{\texttt{while}^n \texttt{ e do c}\}_m$

# Semantics of While

We could go back and try to define the semantics using the approximations:

$$\{\texttt{while e do c}\}_m = \{\texttt{while}^n\ \texttt{e do c}\}_m$$

How do we find the n?

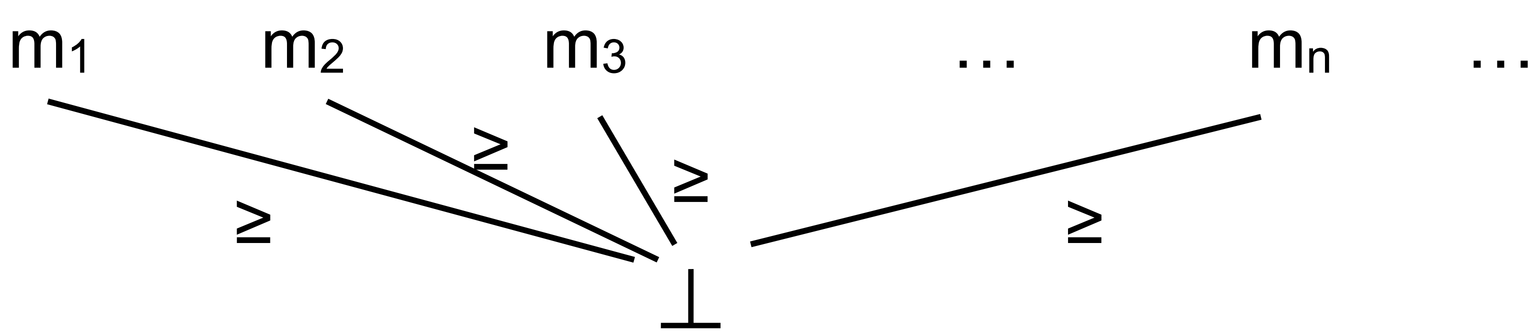
# Information order

Dana Scott

An idea that has been developed to solve this problem is the idea of information order.

This corresponds to the idea of order different possible denotations in term of the information they provide.

In our case we can use the following order on possible outputs:



# Semantics of While

Using fixpoint theorems on lattices we can try now to define the semantics using the approximations and a sup operation:

$$\{\texttt{while e do c}\}_m = \sup_{n \in Nat} \{\texttt{while}^n \texttt{ e do c}\}_m$$

# Semantics of While

Using fixpoint theorems on lattices we can try now to define the semantics using the approximations and a sup operation:

$$\{\texttt{while e do c}\}_m = \sup_{n \in Nat} \{\texttt{while}^n \texttt{ e do c}\}_m$$

Will this work?

# Semantics of While

Using fixpoint theorems on lattices we can try now to define the semantics using the approximations and a sup operation:

$$\{\texttt{while e do c}\}_m = \sup_{n \in Nat} \{\texttt{while}^n \texttt{ e do c}\}_m$$

Will this work?

We are missing the base case.

# Approximating While Revisited

We could define the following lower iteration of a While statement:

```
while_n e do c
```

This can be defined using the approximations as:

```
while_n e do c =
    while^n e do c;if e then abort else skip
```

# Semantics of While

We now have all the components to define the semantics of while:

$$\{\texttt{while e do c}\}_m = \sup_{n \in Nat} \{\texttt{while}_n \texttt{ e do c}\}_m$$

# Semantics of Commands

This is defined on the structure of commands:

$\{abort\}_m = \bot$

$\{skip\}_m = m$

$\{x:=e\}_m = m[x \leftarrow \{e\}_m]$

$\{c;c'\}_m = \{c'\}_{m'}$ If $\{c\}_m = m'$

$\{c;c'\}_m = \bot$ If $\{c\}_m = \bot$

$\{if\ e\ then\ c_t\ else\ c_f\}_m = \{c_t\}_m$ If $\{e\}_m = true$

$\{if\ e\ then\ c_t\ else\ c_f\}_m = \{c_f\}_m$ If $\{e\}_m = false$

$\{while\ e\ do\ c\}_m = sup_{n \in Nat}\{while_n\ e\ do\ c\}_m$

where

$while_n\ e\ do\ c = while^n\ e\ do\ c;if\ e\ then\ abort\ else\ skip$

and

# Semantics of Commands

This is defined on the structure of commands:

$\{\texttt{abort}\}_m = \bot$

$\{\texttt{skip}\}_m = m$

$\{\texttt{x:=e}\}_m = m[x \leftarrow \{e\}_m]$

$\{c;c'\}_m = \{c'\}_{m'}$ If $\{c\}_m = m'$

$\{c;c'\}_m = \bot$ If $\{c\}_m = \bot$

$\{\texttt{if e then } c_t \texttt{ else } c_f\}_m = \{c_t\}_m$ If $\{e\}_m = \texttt{true}$

$\{\texttt{if e then } c_t \texttt{ else } c_f\}_m = \{c_f\}_m$ If $\{e\}_m = \texttt{false}$

$\{\texttt{while e do c}\}_m = \sup_{n \in Nat}\{\texttt{while}_n \texttt{ e do c}\}_m$

where

$\texttt{while}_n \texttt{ e do c} = \texttt{while}^n \texttt{ e do c;if e then abort else skip}$

and

$\texttt{while}^0 \texttt{ e do c} = \texttt{skip}$

$\texttt{while}^{n+1} \texttt{ e do c} = \texttt{if e then (c;while}^n \texttt{ e do c) else skip}$

# Example

What is the semantics of the following program:

```
n:=2;
r:=1;
while n ≥ 1 do
  r := n * r;
  n := n-1;
```

# Example

What is the semantics of the following program:

```
Fact(n: Nat) : Nat
  r:=1;
  while n > 1 do
      r := n * r;
      n := n-1;
```