Formal Methods and Functional Programming

Introduction to Part II

Peter Müller

Chair of Programming Methodology
ETH Zurich

Organization

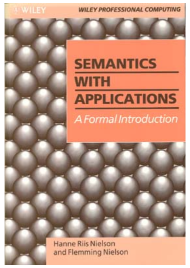
- Most aspects do not change (lecture times, web page, homework)
- In general, please attend the same exercise session
- Some details have changed:
  - Tuesday 13-15, ETZ G91 (Alex Summers, English)
  - Tuesday 13-15, NO D11 (Milos Novacek, English)
  - Tuesday 13-15, NO E11 (Uri Juhasz, English)
  - Wednesday 15-17, IFW A34 (Alex Viand, German)
  - Wednesday 15-17, IFW C33 (Cyril Steimer, German)
  - Students previously attending Ralf Sasse's exercise class (CAB G57) can choose either the class in ETZ G91 or any of the Wednesday classes.
  - Students in Joshua Schneider's class (ETZ G91) who wish to remain taught in German can choose either of the Wednesday classes.
- For all organizational issues, please email Alex Summers (alexander.summers@inf.ethz.ch)

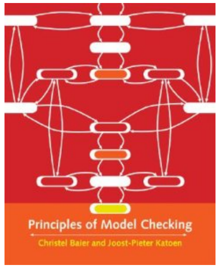
Homeworks and Exam

- Homework can be submitted in one of two ways:
  - By email to the appropriate tutor (see course website)
  - By hand in the appropriate box outside room CAB F53.1

  Solutions must be received by 11:00 on the Monday after the exercise is published, in order to receive feedback.
- The exam will take place in the exam session
  - See web page for details (coming soon)
- Please check the course website regularly, for announcements
  - `http://www.infsec.ethz.ch/education/ss2014/fmfp`

Recommended Books





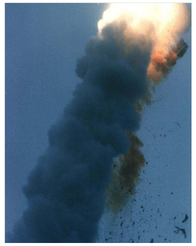
- Hanne Riis Nielson and Flemming Nielson: *Semantics with Applications: A Formal Introduction*
  - Available from `http://www.daimi.au.dk/~bra8130/Wiley_book/wiley.pdf`
- Christel Baier and Joost-Pieter Katoen: *Principles of Model Checking*

Software Errors Cost Large Amounts of Money

- Software errors cost US economy \$59.5 billion annually (estimate by Department of Commerce's National Institute of Standards and Technology, 2002)

- Software bugs in baggage handling system of the airport of Denver led to damage of around \$1 million per day (for almost a year)

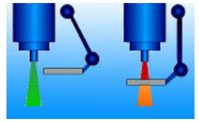
- Explosion of Ariane 5 destroyed satellites worth \$500 million



- In comparison: famous hardware bugs:
  - Pentium bug cost Intel \$500 million
  - Xbox bug cost Microsoft \$1 billion

Software Errors May Cost Lives

- Software error in Therac-25 medical linear accelerator led to overdose, which killed six people

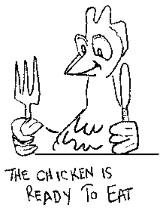


- Rounding error caused Patriot Missile system to ignore an incoming Scud missile; 28 soldiers died

- Many other safety critical systems
  - Controllers in airplanes, cars, trains, etc.
  - Air traffic control systems
  - Nuclear reactor control systems

Traditional Software Engineering

- Describes expected behavior using natural language or semi-formal notations
  - Ambiguities
  - Contradictions
  - Incompletenesses



- Relies on testing to ensure quality
  - *Testing can show the presence of errors, but not their absence.* [E. Dijkstra]
  - Exhaustive testing possible only for trivial programs
  - Some errors are hard to find / reproduce (data races, deadlocks)
  - Achieving good test coverage is difficult (rare cases)

Alternative: Formal Methods

> Formal methods are mathematical approaches to software and system development which support the rigorous specification, design, and verification of computer systems. [FME]

- Programs, programming languages, designs, etc. are mathematical objects and can be treated by mathematical methods
- Examples from Part I of the course:
  - Proving program properties

    $$\forall xs, ys, zs.\, (xs \ {+}{+} \ ys) \ {+}{+} \ zs = xs \ {+}{+} \ (ys \ {+}{+} \ zs)$$

  - Formalizing language semantics

    $$(\lambda x.t)t' \hookrightarrow t[x \leftarrow t']$$

  - Proving language properties

    If $t \hookrightarrow t'$ and $A \vdash t :: \tau$ then $A \vdash t' :: \tau$

Example 1: Sorting Function

`void sort(int[] input)`

- Informal specification:
  Method `sort` sorts the elements of `input` in ascending order

Example 1: Sorting Function

`void sort(int[] input)`

- Informal specification:
  Method `sort` sorts the elements of `input` in ascending order

- Testing
  - `sort({})` → `{}` ✓

Example 1: Sorting Function

`void sort(int[] input)`

- Informal specification:
  Method `sort` sorts the elements of `input` in ascending order

- Testing
  - `sort({})` → `{}` ✓
  - `sort({2})` → `{2}` ✓

Example 1: Sorting Function

`void sort(int[] input)`

- Informal specification:
  Method `sort` sorts the elements of `input` in ascending order

- Testing
  - `sort({})` → `{}` ✓
  - `sort({2})` → `{2}` ✓
  - `sort({2,3,1})` → `{1,2,3}` ✓

Example 1: Sorting Function

```
void sort(int[] input)
```

- Informal specification:
  Method `sort` sorts the elements of `input` in ascending order
- Testing
  - `sort({})` → `{}` ✓
  - `sort({2})` → `{2}` ✓
  - `sort({2,3,1})` → `{1,2,3}` ✓
  - `sort({2,2,1})` → `{1,2,1}` ✗

Example 1: Sorting Function

```
void sort(int[] input)
```

- Informal specification:
  Method `sort` sorts the elements of `input` in ascending order
- Testing
  - `sort({})` → `{}` ✓
  - `sort({2})` → `{2}` ✓
  - `sort({2,3,1})` → `{1,2,3}` ✓
  - `sort({2,2,1})` → `{1,2,1}` ✗
  - `sort(null)` → ↯ ✗

Example 1: Sorting Function—Formal Treatment

- Specification
  - Pre and postcondition in predicate logic (contract)
  - If a is a non-null array of integers and
    in the state before a call `sort(a)`, the elements of a are $e_0 \dots e_n$,
    then the call terminates and immediately after the call,
    the elements of a, $e'_0 \dots e'_n$, are a permutation of $e_0 \dots e_n$
    and $\forall i, j \in [0, n]. i < j \Rightarrow e'_i \leq e'_j$.

Example 1: Sorting Function—Formal Treatment

- Specification
  - Pre and postcondition in predicate logic (contract)
  - If a is a non-null array of integers and
    in the state before a call `sort(a)`, the elements of a are $e_0 \dots e_n$,
    then the call terminates and immediately after the call,
    the elements of a, $e'_0 \dots e'_n$, are a permutation of $e_0 \dots e_n$
    and $\forall i, j \in [0, n]. i < j \Rightarrow e'_i \leq e'_j$.
- Verification
  - Prove that `sort` satisfies its specification using a formal semantics of the programming language

Example 1: Sorting Function—Formal Treatment

- Specification
  - Pre and postcondition in predicate logic (contract)
  - If a is a non-null array of integers and in the state before a call `sort(a)`, the elements of a are $e_0 \ldots e_n$, then the call terminates and immediately after the call, the elements of a, $e'_0 \ldots e'_n$, are a permutation of $e_0 \ldots e_n$ and $\forall i, j \in [0, n].i < j \Rightarrow e'_i \leq e'_j$.
- Verification
  - Prove that `sort` satisfies its specification using a formal semantics of the programming language
- Observations
  - Specification permits duplicate elements in array: Test `sort({2,2,1})` reveals error in implementation
  - Specification excludes `null` from the valid arguments to `sort`: Test `sort(null)` is an invalid test case
  - Correctness proof covers all valid inputs, not just selected test cases

Example 2: Zune Bug

- Zune 30 did not work on Dec. 31, 2008
- Official fix: drain battery and recharge after midday on Jan. 01, 2009

```
//-------------------------
// Split total days since
// Jan. 01, ORIGINYEAR
// into year, month and day
//-------------------------
BOOL ConvertDays(UINT32 days, ...) {
  int year = ORIGINYEAR; /* =1980 */

  while (days > 365) {
    if (IsLeapYear(year)) {
      if (days > 366) {
        days -= 366; year += 1;
      }
    } else {
      days -= 365; year += 1;
    }
  }
  ... }
```


Example 2: Zune Bug—Formal Treatment

- Prove termination formally
- Repetition: Sufficient condition for termination of recursive functions: Arguments are smaller along a well-founded order
- Similar technique for loops
- Zune example:
  - Termination measure: variable days
  - Well-founded order: $<$ with lower bound 365 (loop condition)
  - Error: measure not decreased if `IsLeapYear(year)` and `days==366`

```
while (days > 365) {
  if (IsLeapYear(year)) {
    if (days > 366) {
      days -= 366; year += 1;
    }
  } else {
    days -= 365; year += 1;
  }
}
```


Example 3: Deadlock

- Threads are synchronized via locks
- Interleaved execution of `a.transfer(b,n)` and `b.transfer(a,m)` might deadlock
- Multi-threaded programs are extremely hard to test

```
class Account {
  int balance;

  void transfer(Account to, int amount) {
    acquire this;
    acquire to;
    this.balance -= amount;
    to.balance += amount;
    release this;
    release to;
  }
}
```

Example 3: Deadlock—Formal Treatment (1)

- Prevent deadlocks by acquiring locks in ascending order
- Prove absence of deadlocks by:
  - Defining an order on locks
  - Proving for each `acquire o` that o is above all other locks held by the current thread

```
class Account {
  int balance;
  int number; // unique account number

  void transfer(Account to, int amount) {
    if (this.number < to.number) {
      acquire this;
      acquire to;
    } else {
      acquire to;
      acquire this;
    }
    this.balance -= amount;
    to.balance += amount;
    release this;
    release to;
  }
}
```


Example 3: Deadlock—Formal Treatment (2)

- Alternative approach: state space exploration
  - Enumerate all possible states of a system
  - Check properties on the states and their transitions
  - Absence of deadlock: check for each state that there is a way to reach the terminal state

Example 3: Deadlock—Formal Treatment (2)

- Alternative approach: state space exploration
  - Enumerate all possible states of a system
  - Check properties on the states and their transitions
  - Absence of deadlock: check for each state that there is a way to reach the terminal state

- Main problem: size of state space
- Explore abstractions of real program (here, `balance` does not matter)
- Explore state space for limited executions
  - Small number of threads (here, two are sufficient)
  - Small number of objects (here, two are sufficient)
  - Small number of context switches (here, one is sufficient)

Example 3: Deadlock—Formal Treatment (2)

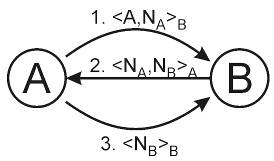
- Alternative approach: state space exploration
  - Enumerate all possible states of a system
  - Check properties on the states and their transitions
  - Absence of deadlock: check for each state that there is a way to reach the terminal state

- Main problem: size of state space
- Explore abstractions of real program (here, `balance` does not matter)
- Explore state space for limited executions
  - Small number of threads (here, two are sufficient)
  - Small number of objects (here, two are sufficient)
  - Small number of context switches (here, one is sufficient)

- State space exploration typically gives no correctness guarantee
  - Similar to testing
  - Very effective in practice

Example 4: Needham-Schroeder Protocol

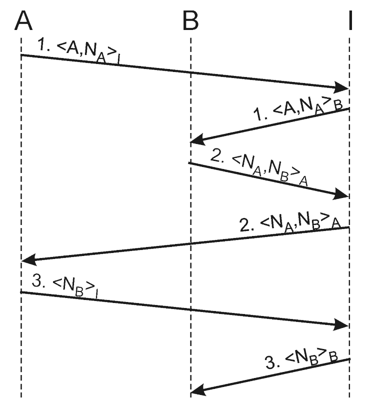
- Establish a common secret over an insecure channel
  1. Alice sends random number N_A to Bob, encrypted with Bob's public key: $\langle A, N_A \rangle_B$
  2. Bob sends random number N_B to Alive, encrypted with Alice's public key: $\langle N_A, N_B \rangle_A$
  3. Alice responds with $\langle N_B \rangle_B$



- Intruders may:
  - Intercept, store, and replay messages
  - Initiate or participate in runs of the protocol
  - Decrypt messages only if encrypted with intruder's public key
- Error: intruder can pretend to be another party

Example 4: Needham-Schroeder Protocol—Formal Treatment

- State space exploration: enumerate protocol runs
  - Develop formal model of intruder as non-deterministic program
  - Simplifications: two agents, one intruder with limited memory
  - Check whether there is a protocol run such that agent believes to talk to other agent, but in fact talks to intruder
- Error was found this way 17 years after protocol was published



Observations: Formal Specification

- Use mathematical notations to describe:
  - Assumptions about the environment (e.g., intruder model)
  - Requirements for the system (desired properties, e.g., deadlock freedom)
  - System design to accomplish these requirements (e.g., program code)

Observations: Formal Specification

- Use mathematical notations to describe:
  - Assumptions about the environment (e.g., intruder model)
  - Requirements for the system (desired properties, e.g., deadlock freedom)
  - System design to accomplish these requirements (e.g., program code)

- Requirements
  - Safety properties: Something bad will never happen
    - Functional behavior of `sort` (no "incorrect" return-values)
    - Absence of certain faults (e.g., null-pointer exception, buffer overflow)
  - Liveness properties: Something good will happen eventually
    - Termination of `ConvertDays`
    - Each request gets served eventually
  - Non-functional requirements
    - Resource consumption, e.g., memory usage
    - Runtime, e.g., realtime guarantees

Observations: Formal Verification

- Use formal logic to:
  - Validate specifications by checking consistency
    Example: termination measure uses well-founded order
  - Prove that design satisfies requirements under given assumptions
    Example: code does not deadlock
  - Prove that a more detailed design implements a more abstract one (refinement)
    Example: protocol implementation refines protocol specification

Observations: Formal Verification

- Use formal logic to:
  - Validate specifications by checking consistency
    Example: termination measure uses well-founded order
  - Prove that design satisfies requirements under given assumptions
    Example: code does not deadlock
  - Prove that a more detailed design implements a more abstract one (refinement)
    Example: protocol implementation refines protocol specification

- Proof methods
  - Deductive: proof system
    Example: prove termination in a program logic
  - Algorithmic: state space exploration (model checking)
    Example: enumerate and check protocol runs

ETH zürich Peter Müller—Formal Methods and Functional Programming, SS14 p. 19

Formal Methods: Ingredients

- Underlying programming/modeling system
  - Programming language with precise (formal) semantics or
  - Modeling language for constructing formal models of software
- Specification language
  - Desired properties expressed as logical formulas in a formal logic
  - Precise meaning for "the system satisfies a property"
- Proof method
  - Method to establish or refute that a system satisfies a property
  - When not satisfied, may also provide a counterexample
- Tool support
  - For specification and verification
  - Proofs are often simple, but long and tedious (unlike in mathematics)
  - Tools needed to check details, e.g., theorem provers and model checkers

Benefits of Formal Methods

- Strong guarantees
  - Detect faults with greater certainty than testing
  - Guarantee absence of specific faults
  - Unambiguous communication and documentation

Benefits of Formal Methods

- Strong guarantees
  - Detect faults with greater certainty than testing
  - Guarantee absence of specific faults
  - Unambiguous communication and documentation
- Universality
  - Properties of concrete programs (e.g., termination proof)
  - Software designs (e.g., protocol verification)
  - Programming languages / new features (e.g., type safety proof)
  - Hardware (e.g., refinement proof between gate and transistor design)

Benefits of Formal Methods

- Strong guarantees
  - Detect faults with greater certainty than testing
  - Guarantee absence of specific faults
  - Unambiguous communication and documentation
- Universality
  - Properties of concrete programs (e.g., termination proof)
  - Software designs (e.g., protocol verification)
  - Programming languages / new features (e.g., type safety proof)
  - Hardware (e.g., refinement proof between gate and transistor design)
- Didactic value: Studying formal methods:
  - Leads to deep understanding of semantics of programs and specifications
  - Increases awareness of subtle issues of programs, languages, etc.
  - Makes you a better engineer!

Success Stories

- Paris driverless metro (Meteor)
  - Safety-critical system
  - Pilot software developed through stepwise refinement in B
  - Most detailed design translated automatically to 30,000 lines of Ada
  - 28,000 proofs

Success Stories

- Paris driverless metro (Meteor)
  - Safety-critical system
  - Pilot software developed through stepwise refinement in B
  - Most detailed design translated automatically to 30,000 lines of Ada
  - 28,000 proofs
- Static Driver Verifier/SLAM at Microsoft
  - Windows device drivers running in kernel mode should respect API
  - Third-party device drivers not respecting APIs responsible for 90% of Windows crashes
  - SLAM inspects C code using a combination of model checking and theorem proving

Success Stories

- Paris driverless metro (Meteor)
  - Safety-critical system
  - Pilot software developed through stepwise refinement in B
  - Most detailed design translated automatically to 30,000 lines of Ada
  - 28,000 proofs
- Static Driver Verifier/SLAM at Microsoft
  - Windows device drivers running in kernel mode should respect API
  - Third-party device drivers not respecting APIs responsible for 90% of Windows crashes
  - SLAM inspects C code using a combination of model checking and theorem proving
- Airbus 380 flight controller
  - Safety-critical system
  - Static analysis of 500,000 lines of C code
  - Proved absence of runtime errors (e.g., buffer overflows)

Limitations

- Incorrect specifications
  - Formal methods per se do not guarantee correctness
  - Verifying the wrong specification is useless
  - It is difficult to get specifications right
- Technical limitations
  - Almost all interesting properties are undecidable, in general
  - Many tools quickly reach limits (scope, computing resources)
- Many applications of formal methods require specialist users
  - Strong background in mathematics / training in formal modeling
  - Some tools try to hide this complexity from users (research topic)
- Application of formal methods is expensive
  - But testing is expensive, too

Formal Methods and Testing

- Formal methods and testing complement each other

Formal Methods and Testing

- Formal methods and testing complement each other
- Testing still necessary
  - Validate specifications
  - Test properties not formally proven (e.g., performance)
  - Detect errors in environment (e.g., compiler)

Formal Methods and Testing

- Formal methods and testing complement each other
- Testing still necessary
  - Validate specifications
  - Test properties not formally proven (e.g., performance)
  - Detect errors in environment (e.g., compiler)
- Formal methods aid testing
  - Derive test cases, test data, and test oracles from specifications
  - Increase test coverage
  - Replace (infinitely) many tests

Course Outline—Part II

- Focus: formal methods for (stateful) software
  - Imperative programs and languages
  - Software designs

1. Formal semantics of imperative programming languages
   - Operational semantics
   - Axiomatic semantics (Hoare logic)
2. Modeling and state space exploration techniques
   - Constructing models of software designs
   - Temporal logic and model checking

C: Expression Evaluation

```
int print(char* text) {
  printf("%s\n", text);
  return 5;
}
```

```
print("One")+print("Two");
```


C: Expression Evaluation

```
int print(char* text) {
  printf("%s\n", text);
  return 5;
}
```

```
print("One")+print("Two");
```

One Two	Two One

In C and C++, evaluation order of expressions is undefined

- Precedence and associativity define rules for structuring expressions
- But do not define operand evaluation order

Haskell and SML: Evaluation

Haskell

```
const :: Int -> Int
const x = 1
```

```
const ( 2 'div' 0 )
```

SML

```
fun const (x: int):int = 1;
```

```
const ( 2 div 0 );
```


Haskell and SML: Evaluation

Haskell

```
const :: Int -> Int
const x = 1
```

```
const ( 2 'div' 0 )
```

```
1
```

SML

```
fun const (x: int):int = 1;
```

```
const ( 2 div 0 );
```

```
uncaught exception divide by zero
```

- Haskell uses lazy evaluation:
  Arguments are evaluated when they are needed
- SML uses eager evaluation:
  Arguments are evaluated when function is applied

Java: Dynamic Method Binding

```
class C1 {
  int x = 5;
  public  void inc1( )
    { this.inc2( ); }
  private void inc2( )
    { x++; }
}
```

```
class CS1 extends C1 {
  public void inc2( )
    { inc1( ); }
}
```

```
CS1 cs = new CS1();
cs.inc2( );
System.out.println(cs.x);
```


Java: Dynamic Method Binding

```
class C1 {
  int x = 5;
  public  void inc1( )
    { this.inc2( ); }
  private void inc2( )
    { x++; }
}
```

```
class CS1 extends C1 {
  public void inc2( )
    { inc1( ); }
}
```

```
CS1 cs = new CS1();
cs.inc2( );
System.out.println(cs.x);
```

```
class C2 {
  int x = 5;
  public    void inc1( )
    { this.inc2( ); }
  protected void inc2( )
    { x++; }
}
```

```
class CS2 extends C2 {
  public void inc2( )
    { inc1( ); }
}
```

```
CS2 cs = new CS2();
cs.inc2( );
System.out.println(cs.x);
```

Java: Class Initialization

```
class C {
  public static int x;
}
```

```
class D {
  public static char y;
  ...

}
```

```
C.x = 0;
D.y = '?';
System.out.println(C.x);
```


Java: Class Initialization

```
class C {
  public static int x;
}
```

```
class D {
  public static char y;

  static { C.x = C.x + 1; }
}
```

```
C.x = 0;
D.y = '?';
System.out.println(C.x);
```

```
1
```


Why Formal Semantics?

- Programming language design
  - Formal verification of language properties
  - Reveal ambiguities
  - Support for standardization
- Implementation of programming languages
  - Specification for developing compilers
  - Generation of interpreters
  - Portability (abstract description of language semantics)
  - Evaluation of new programming language features
- Reasoning about programs
  - Formal verification of program properties

Programming Language Properties

- Type safety:
  In each execution state, a variable of type T holds a value of type T (or a subtype of T)
- Very important question for language designers
- Example:
  If `String` is a subtype of `Object`, should `String[]` be a subtype of `Object[]`?

Programming Language Properties

- Type safety:
  In each execution state, a variable of type T holds a value of type T (or a subtype of T)
- Very important question for language designers
- Example:
  If `String` is a subtype of `Object`, should `String[]` be a subtype of `Object[]`?

```
void m(Object[] oa) {
  oa[0]=new Integer(5);
}
```

```
String[] sa=new String[10];
m(sa);
String s = sa[0];
```


Compiler Optimization

- Common subexpression elimination

```
d = a * Math.sqrt(c);
e = b * Math.sqrt(c);
```

```
double tmp=Math.sqrt(c);
d = a * tmp;
e = b * tmp;
```


Compiler Optimization

- Common subexpression elimination

```
d = a * Math.sqrt(c);
e = b * Math.sqrt(c);
```

```
double tmp=Math.sqrt(c);
d = a * tmp;
e = b * tmp;
```

- Optimization works only for side-effect free expressions

```
d = a * c++;
e = b * c++;
```

```
double tmp = c++;
d = a * tmp;
e = b * tmp;
```


Formal Verification

```
/* returns the
   factorial of n */
int fac(int n) {
  if (n>1)
    return n*fac(n-1);
  else
    return 1;
}
```

Formal Verification

```
/* returns the
   factorial of n */
int fac(int n) {
  if (n>1)
    return n*fac(n-1);
  else
    return 1;
}
```

`fac(17);` -288522240

Formal Verification

```
/* returns the
   factorial of n */
int fac(int n) {
  if (n>1)
    return n*fac(n-1);
  else
    return 1;
}
```

`fac(17);` -288522240

- Verification could run by induction
- Induction hypothesis: $n \geq 0 \Rightarrow \mathit{fac}(n) = n!$
- Induction base is trivial
- Induction step requires to prove $n \times (n-1)! = n!$ which is not the case in computer arithmetic (for ints)

Three Kinds of Programming Language Semantics

- Operational semantics
  - Describes execution on an abstract machine
  - Describes how the effect is achieved; abstractly, how the program runs
- Denotational semantics (not in this course)
  - Programs are regarded as functions in a mathematical domain
  - Describes only the effect, not how it is obtained
- Axiomatic semantics
  - Specific properties of the effect of executing a program are expressed
  - Some aspects of the computation may be ignored

Operational Semantics

```
y := 1;
while not(x=1) do ( y := x*y; x := x-1 )
```

- "First we assign 1 to `y`, then we test whether `x` is 1 or not. If it is then we stop and otherwise we update `y` to be the product of `x` and the previous value of `y` and then we decrement `x` by 1. Now we test whether the new value of `x` is 1 or not..."
- Two kinds of operational semantics
  - Natural Semantics (coarse-grained view of execution)
  - Structural Operational Semantics (fine-grained view of execution)

Axiomatic Semantics

```
y := 1;
while not(x=1) do ( y := x*y; x := x-1 )
```

- "If x= n holds before the program is executed then y= $n!$ will hold when the execution terminates (if it terminates)"
- Two kinds of axiomatic semantics
  - Partial correctness (properties modulo program termination)
  - Total correctness (prove termination as additional property)

Which Semantics to Use—Selection Criteria

Constructs of the language

- Imperative
- Functional
- Concurrent
- Object-oriented
- Non-deterministic
- etc.

Application of the semantics

- Understanding the language
- Program verification
- Prototyping
- Compiler construction
- Program analysis
- etc.

Focus of this Course

- We discuss the major approaches to semantics for a small imperative language (called IMP)
  - Similarities and differences between semantics
  - Important theoretical results
- Operational Semantics
  - Natural and structural operational semantics of IMP
- Axiomatic Semantics
  - Hoare logic for IMP