Formal Methods and Functional Programming

Part II

Peter Müller

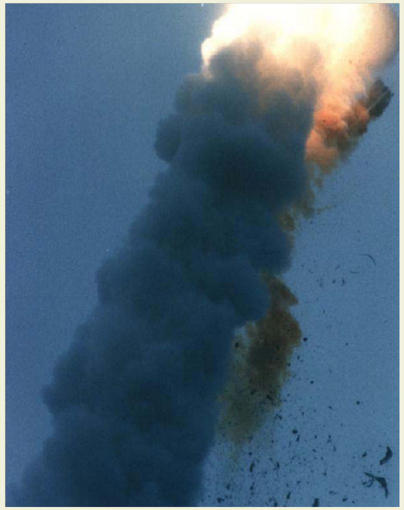
Chair of Programming Methodology
ETH Zurich

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)

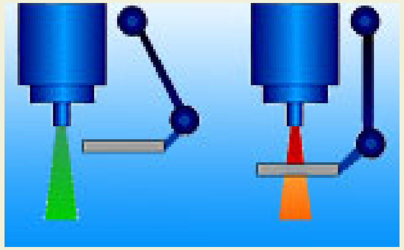
- Software bugs in baggage handling system of the airport of Denver lead to damage of around $1 million per day (for almost a year)

- 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

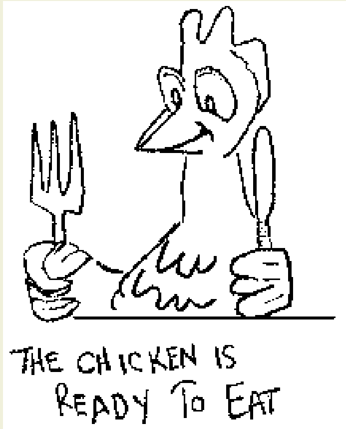


- Software error in Therac-25 medical linear accelerator lead to overdose, which killed six people

- 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 (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

    $$\forall x, y, z.(x + +y) + +z = x + +(y + +z)$$

  - Formalizing language semantics

    $$(\lambda x.M)N \hookrightarrow M[x \leftarrow N]$$

  - Proving language properties

    $$\text{If } e \hookrightarrow e' \text{ and } \vdash e :: \tau \text{ then } \vdash e' :: \tau$$

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({})` $\rightarrow$ `{}` ✓

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 $\forall i, j \in [0, n]. i < j \Rightarrow e'_i \le e'_j$.

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 $e_0 \dots e_n$, 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
- 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)

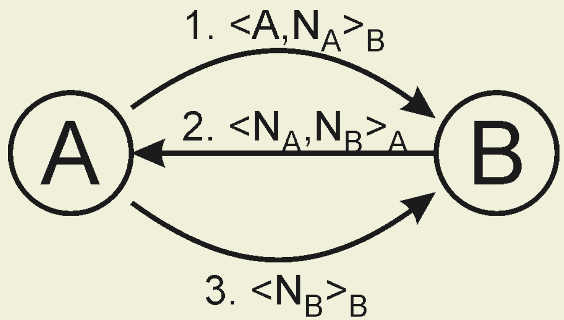
- Alternative approach: state space exploration
  - Enumerate all possible states of a system
  - Check properties on the states and their transitions
  - Absence of deadlock: check for each state that there is a way to reach the terminal state

- 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

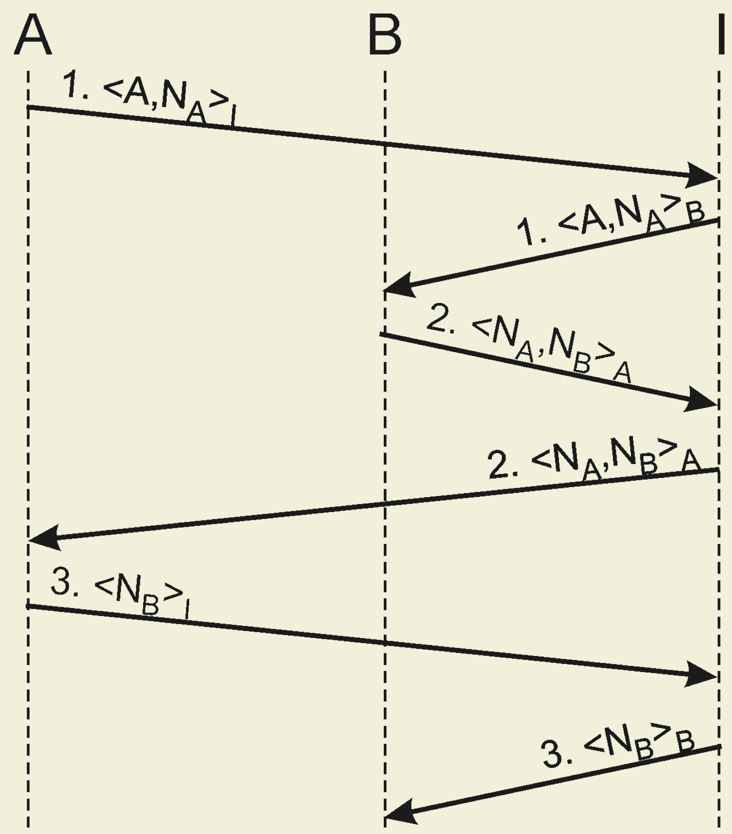
- Establish a common secret over an insecure channel
  1. Alice sends random number N_A to Bob, encrypted with Bob's public key: $\langle A, N_A \rangle_B$
  2. Bob sends random number N_B to Alive, encrypted with Alice's public key: $\langle N_A, N_B \rangle_A$
  3. Alice responds with $\langle N_B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



- 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`
    - Absence of certain faults (e.g., buffer overflow)
  - Liveness properties: Something good will happen eventually
    - Termination of `ConvertDays`
    - Deadlock freedom of `transfer`
  - 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 method
  - Deductive: proof system
    Example: prove termination in a program logic
  - Algorithmic: state space exploration (model checking)
    Example: enumerate and check protocol runs

Formal Methods: Ingredients

- Specification language
  - Modeling or programming language with precise semantics
  - Desired properties expressed as logical formulas or abstract system
  - Precise meaning of "system satisfies property"
- Proof method
  - Method to establish or refute that a system satisfies a property
- Tool support
  - For specification and verification
  - Proofs are often simple, but tedious (in contrast to mathematics)
  - Tools needed to check details
  - Main examples: 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
  - Programs (e.g., termination proof)
  - Software designs (e.g., protocol verification)
  - Programming languages (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
  - Programs (e.g., termination proof)
  - Software designs (e.g., protocol verification)
  - Programming languages (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, design specifications, etc.
  - Increases awareness of subte 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
  - Many tools quickly reach limits (scope, computing resources)
- Most formal methods require specialist users
  - Strong background in mathematics
  - Training in formal modelling
- 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 programming languages
   - Operational semantics
   - Hoare logic
2. State space explotarion
   - Temporal logic
   - Model checking

Organization

- Most aspects do not change (web page, homework)
- Different tutors
  - Tuesday 16-18, IFW C42 (Alex Summers, English)
  - Tuesday 16-18, IFW B42 (Yannis Kassios, English)
  - Tuesday 16-18, IFW A34 (Malte Schwerhoff, German)
  - Wednesday 15-17, IFW A32.1 (Alex Summers, English)
  - Wednesday 15-17, IFW B42 (Yannis Kassios, English)

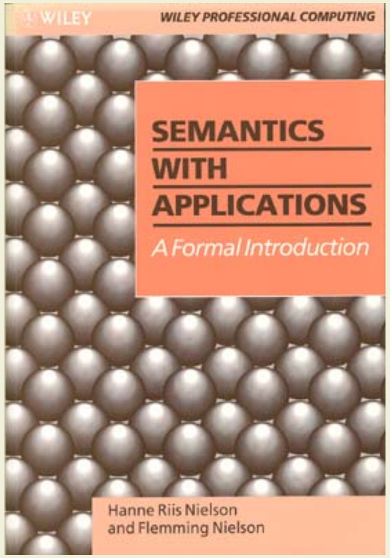
  Please attend the same session as in the first half of the course
- Homework can be submitted in one of two ways:
  - By email to the appropriate tutor
  - By hand in the appropriate box outside room RZ F1

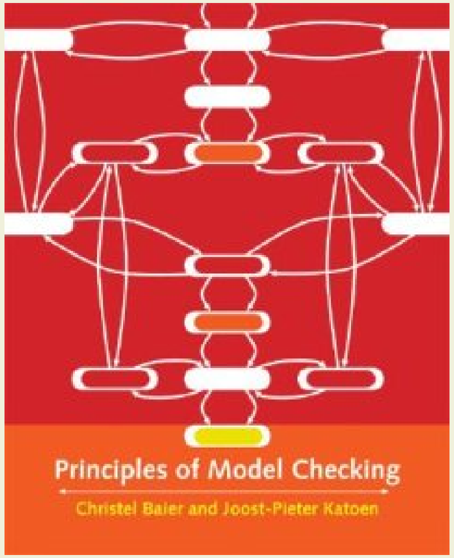
  Solutions must be received by 9:15 on the Monday after the exercise is published, in order to receive feedback.

Exam

- The final exam will take place on Thursday, June 10, 2010, 9:00–11:00
  - See web page for details
- The grade in the course will be determined based on the average points received in the midterm and the final exam.
- We offer a Q&A session in the very last lecture
  - Thursday, June 03, 10:00 - 12:00 in HG F 7

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

1. Introduction to Language Semantics

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;
  public  void inc1( )
    { this.inc2( ); }
  private void inc2( )
    { x++; }
}
```

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

```
CS1 cs = new CS1(5);
cs.inc2( );
System.out.println(cs.x);
```

Java: Dynamic Method Binding

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

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

```
CS1 cs = new CS1(5);
cs.inc2( );
System.out.println(cs.x);
```

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

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

```
CS2 cs = new CS2(5);
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
  - Compilers
  - Interpreters
  - Portability
- Reasoning about programs
  - Formal verification of program properties
  - Extended static checking

Language Properties

- Type safety:
  In each execution state, a variable of type T holds a value of 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[]`?

Language Properties

- Type safety:
  In each execution state, a variable of type T holds a value of 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

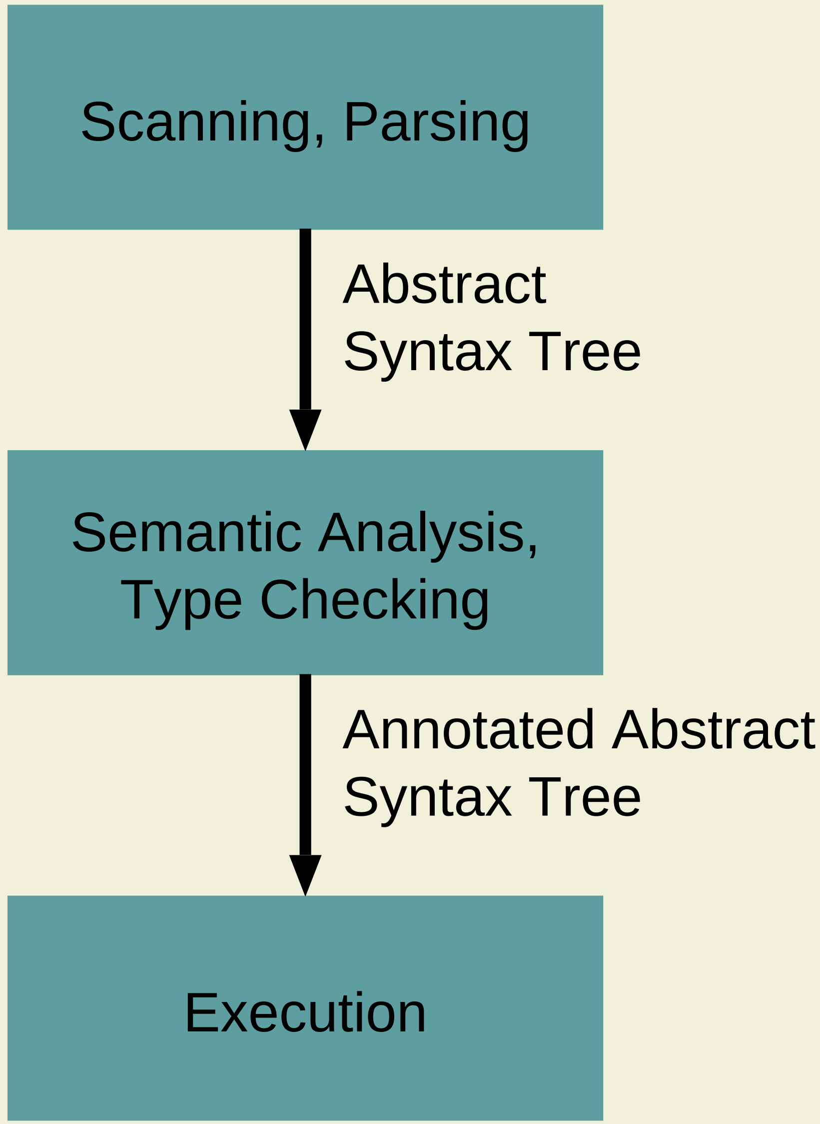
- Verification could run by induction
- Induction hypothesis: $n \geq 0 \Rightarrow fac(n) = n!$
- Induction base is trivial
- Induction step requires to prove $n \times (n-1)! = n!$ which is not the case in computer arithmetic

1. Introduction to Language Semantics

Language Definition

Dynamic Semantics	• State of a program execution • Transformation of states
Static Semantics	• Type rules • Name resolution
Syntax	• Syntax rules, defined by grammar

Compilation and Execution



Three Kinds of Semantics

- Operational semantics
  - Describes execution on an abstract machine
  - Describes how the effect is achieved
- Denotational semantics
  - 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
  - Structural Operational Semantics

Denotational Semantics

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

- "For input values of x greater than 0, the program computes a partial function from states to states: the final state will be equal to the initial state except that the value of x will be 1 and the value of y will be equal to the factorial of the value of x in the initial state"
- Two kinds of denotational semantics
  - Direct Style Semantics
  - Continuation Style Semantics

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
  - Total correctness

Abstraction

Concrete language implementation
Operational semantics
Denotational semantics
Axiomatic semantics
Abstract description

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 IMP
  - Similarities and differences
  - Important theoretical results
- Operational Semantics
  - Natural and structural operational semantics of IMP
  - Equivalence
- Axiomatic Semantics
  - Axiomatic semantics of IMP
  - Soundness and completeness

1. Introduction to Language Semantics

The Language IMP

- Expressions
  - Boolean and arithmetic expressions
  - No side-effects in expressions
- Variables
  - All variables range over integers
  - All variables are initialized
- IMP does not include
  - Heap allocation and pointers
  - Variable declarations
  - Procedures
  - Concurrency

Syntax of IMP: Characters and Tokens

Characters

$$\begin{array}{lll} \text{Letter} & = & \texttt{'A'} \mid \ldots \mid \texttt{'Z'} \mid \texttt{'a'} \mid \ldots \mid \texttt{'z'} \\ \text{Digit} & = & \texttt{'0'} \mid \texttt{'1'} \mid \texttt{'2'} \mid \texttt{'3'} \mid \texttt{'4'} \mid \texttt{'5'} \mid \texttt{'6'} \mid \texttt{'7'} \mid \texttt{'8'} \mid \texttt{'9'} \end{array}$$

Tokens

$$\begin{array}{lll} \text{Ident} & = & \text{Letter } \{ \text{ Letter} \mid \text{Digit } \} \\ \text{Numeral} & = & \text{Digit} \mid \text{Numeral Digit} \\ \text{Var} & = & \text{Ident} \end{array}$$

Syntax of IMP: Expressions

Arithmetic expressions

$$\begin{array}{lcl} \text{Aexp} & = & \text{'('}\ \text{Aexp Op Aexp}\ \text{')'} \mid \text{Var} \mid \text{Numeral} \\ \text{Op} & = & \text{'+'} \mid \text{'-'} \mid \text{'*'} \end{array}$$

Boolean expressions

$$\begin{array}{lcl} \text{Bexp} & = & \text{'('}\ \text{Bexp}\ \text{'}\texttt{or}\text{'}\ \text{Bexp}\ \text{')'} \mid \text{'('}\ \text{Bexp}\ \text{'}\texttt{and}\text{'}\ \text{Bexp}\ \text{')'} \\ & & \mid \text{'}\texttt{not}\text{'}\ \text{Bexp} \mid \text{Aexp RelOp Aexp} \\ \text{RelOp} & = & \text{'}\texttt{=}\text{'} \mid \text{'}\texttt{\#}\text{'} \mid \text{'}\texttt{<}\text{'} \mid \text{'}\texttt{<=}\text{'} \mid \text{'}\texttt{>}\text{'} \mid \text{'}\texttt{>=}\text{'} \end{array}$$

We omit parentheses if permitted by the usual operator precedence

Syntax of IMP: Statemens

$$
\begin{aligned}
\text{Stm} \quad &= \text{'}\texttt{skip}\text{'} \\
&\mid \text{Var '}\texttt{:=}\text{' Aexp} \\
&\mid \text{Stm '}\texttt{;}\text{' Stm} \\
&\mid \text{'}\texttt{if}\text{' Bexp '}\texttt{then}\text{' Stm '}\texttt{else}\text{' Stm '}\texttt{end}\text{'} \\
&\mid \text{'}\texttt{while}\text{' Bexp '}\texttt{do}\text{' Stm '}\texttt{end}\text{'}
\end{aligned}
$$

Notation

Meta-variables (written in *italic* font)

x, y, z	for variables (Var)
e, e', e_1, e_2	for arithmetic expressions (Aexp)
b, b_1, b_2	for boolean expressions (Bexp)
s, s', s_1, s_2	for statements (Stm)

Keywords are written in `typewriter` font

Syntax of IMP: Example

```
res := 1;
while n > 1 do
  res := res * n;
  n := n - 1
end
```

1. Introduction to Language Semantics

Semantic Categories

Syntactic category: Numeral   Semantic category: Val = $\mathbb{Z}$

$$\boxed{\texttt{101}} \longrightarrow \boxed{5}$$

$$\boxed{\texttt{101}} \longrightarrow \boxed{101}$$

- Semantic functions map elements of syntactic categories to elements of semantic categories
- To define the semantics of IMP, we need semantic functions for
  - Numerals (syntactic category Numeral)
  - Arithmetic expressions (syntactic category Aexp)
  - Boolean expressions (syntactic category Bexp)
  - Statements (syntactic category Stm)

Semantics of Numerals

The semantic function

$$\boxed{\mathcal{N} : \text{Numeral} \to \text{Val}}$$

maps a numeral n to an integer value $\mathcal{N}[\![n]\!]$

$$\begin{array}{llll}
\mathcal{N}[\![0]\!] & = 0 & \mathcal{N}[\![1]\!] & = 1 \\
\ldots & & & \\
\mathcal{N}[\![8]\!] & = 8 & \mathcal{N}[\![9]\!] & = 9 \\
\mathcal{N}[\![n\ 0]\!] & = \mathcal{N}[\![n]\!] \times 10 + 0 & \mathcal{N}[\![n\ 1]\!] & = \mathcal{N}[\![n]\!] \times 10 + 1 \\
\ldots & & & \\
\mathcal{N}[\![n\ 8]\!] & = \mathcal{N}[\![n]\!] \times 10 + 8 & \mathcal{N}[\![n\ 9]\!] & = \mathcal{N}[\![n]\!] \times 10 + 9
\end{array}$$

States

$$\boxed{\texttt{x+1}} \longrightarrow \boxed{??}$$

- The meaning of an expression depends on the values bound to the variables that occur in it
- A state associates a value to each variable

$$\boxed{\text{State} : \text{Var} \to \text{Val}}$$

- We represent a state σ as a finite function

$$\boxed{\sigma = \{x_1 \mapsto v_1, x_2 \mapsto v_2, \ldots, x_n \mapsto v_n\}}$$

where $x_1, x_2, \ldots, x_n$ are different elements of Var and $v_1, v_2, \ldots, v_n$ are elements of Val.

Semantics of Arithmetic Expressions

The semantic function

$$\mathcal{A} : \mathrm{Aexp} \to \mathrm{State} \to \mathrm{Val}$$

maps an arithmetic expression e and a state σ to a value $\mathcal{A}[\![e]\!]\sigma$

$$\begin{array}{ll} \mathcal{A}[\![x]\!]\sigma & = \sigma(x) \\ \mathcal{A}[\![n]\!]\sigma & = \mathcal{N}[\![n]\!] \\ \mathcal{A}[\![e_1 \ op \ e_2]\!]\sigma & = \mathcal{A}[\![e_1]\!]\sigma \ \overline{op} \ \mathcal{A}[\![e_2]\!]\sigma \quad \text{for } op \in \ \mathrm{Op} \end{array}$$

$\overline{op}$ is the operation $\mathrm{Val} \times \mathrm{Val} \to \mathrm{Val}$ corresponding to op

Semantics of Boolean Expressions

The semantic function

$$\boxed{\mathcal{B} : \text{Bexp} \rightarrow \text{State} \rightarrow \text{Bool}}$$

maps a boolean expression b and a state σ to a truth value $\mathcal{B}[\![b]\!]\sigma$

$$\mathcal{B}[\![e_1 \; op \; e_2]\!]\sigma \quad = \begin{cases} tt & \text{if } \mathcal{A}[\![e_1]\!]\sigma \; \overline{op} \; \mathcal{A}[\![e_2]\!]\sigma \\ ff & \text{otherwise} \end{cases}$$

$op \in$ RelOp and $\overline{op}$ is the relation Val $\times$ Val corresponding to op

Boolean Expressions (cont'd)

$$\mathcal{B}[\![b_1 \ \texttt{or} \ b_2]\!]\sigma = \begin{cases} tt & \text{if } \mathcal{B}[\![b_1]\!]\sigma = tt \text{ or } \mathcal{B}[\![b_2]\!]\sigma = tt \\ f\!f & \text{otherwise} \end{cases}$$

$$\mathcal{B}[\![b_1 \ \texttt{and} \ b_2]\!]\sigma = \begin{cases} tt & \text{if } \mathcal{B}[\![b_1]\!]\sigma = tt \text{ and } \mathcal{B}[\![b_2]\!]\sigma = tt \\ f\!f & \text{otherwise} \end{cases}$$

$$\mathcal{B}[\![\texttt{not} \ b]\!]\sigma = \begin{cases} tt & \text{if } \mathcal{B}[\![b]\!]\sigma = f\!f \\ f\!f & \text{otherwise} \end{cases}$$

1. Introduction to Language Semantics

Well-Founded Relations

- Definition

  > A binary relation $\prec$ on a set A is *well-founded* iff there are no infinite descending chains
  > $$\ldots \prec a_i \prec \ldots \prec a_1 \prec a_0$$

- Examples

  $<$ is a well-founded relation on $\mathbb{N}$
  $<$ is not well-founded on $\mathbb{Z}$
  $\leq$ is not well-founded on $\mathbb{N}$

- Well-founded relations are also called Noetherian orders

Well-Founded Induction

- Principle of well-founded induction

> Let $\prec$ be a well-founded relation on a set A. Let P be a property. Then the following equivalence holds.
>
> $$(\forall a \in A : ((\forall b \in A : b \prec a \Rightarrow P(b)) \Rightarrow P(a)))$$
> $$\Leftrightarrow \forall a \in A : P(a)$$

- Mathematical induction is a special case of well-founded induction
  - Set: $\mathbb{N}$
  - Relation: $n \prec m$ iff $m = n + 1$

- Structural induction is a special case of well-founded induction
  - Set: the set of terms of an algebraic data type
  - Relation: $n \prec m$ iff n is a (proper) sub-term of m

Structural Induction: Example

- Syntax defined as algebraic data type

$$\text{Aexp} \quad = \text{'('}\ \text{Aexp Op Aexp}\ \text{')'}\ \mid\ \text{Var}\ \mid\ \text{Numeral}$$

  - Constructors are left implicit

- Structural induction for arithmetic expressions

$$\begin{array}{l}
(\forall n \in \text{Numeral} : P(n)) \land \\
(\forall x \in \text{Var} : P(x)) \land \\
(\forall e_1, e_2 \in \text{Aexp} : P(e_1) \land P(e_2) \Rightarrow P(e_1\ op\ e_2)) \\
\Leftrightarrow \\
\forall e \in \text{Aexp} : P(e)
\end{array}$$

Inductive Definitions

The semantics is given by recursive definitions of functions

- The values for the basis elements are defined directly
- The values for composite elements are defined inductively in terms of the immediate constituents

$$
\begin{array}{ll}
\mathcal{A}[\![x]\!]\sigma & = \sigma(x) \\
\mathcal{A}[\![n]\!]\sigma & = \mathcal{N}[\![n]\!] \\
\mathcal{A}[\![e_1 \ op \ e_2]\!]\sigma & = \mathcal{A}[\![e_1]\!]\sigma \ \overline{op} \ \mathcal{A}[\![e_2]\!]\sigma \quad \text{for } op \in \text{ Op}
\end{array}
$$

- Since the decomposition of the elements is unique this means that the semantics is well-defined
- Inductive definitions enable proofs by structural induction

Using Structural Induction

- Lemma: The equations for $\mathcal{N}$ define a total function $\mathcal{N} : \text{Numeral} \to \text{Val}$
- To prove the lemma, we show that for each $n \in \text{Numeral}$ there is exactly one $v \in \text{Val}$ such that $\mathcal{N}[\![n]\!] = v$
- $\mathcal{N}$ is defined inductively:

$$
\begin{array}{llll}
\mathcal{N}[\![0]\!] & = 0 & \mathcal{N}[\![1]\!] & = 1 \\
\dots & & & \\
\mathcal{N}[\![8]\!] & = 8 & \mathcal{N}[\![9]\!] & = 9 \\
\mathcal{N}[\![n\ 0]\!] & = \mathcal{N}[\![n]\!] \times 10 + 0 & \mathcal{N}[\![n\ 1]\!] & = \mathcal{N}[\![n]\!] \times 10 + 1 \\
\dots & & & \\
\mathcal{N}[\![n\ 8]\!] & = \mathcal{N}[\![n]\!] \times 10 + 8 & \mathcal{N}[\![n\ 9]\!] & = \mathcal{N}[\![n]\!] \times 10 + 9
\end{array}
$$

- Therefore, we can prove the lemma by structural induction on n

Proof: $\mathcal{N}$ is a Total Function

1. Induction base: all terms of height 1 (digits)
   There are ten cases for the induction base for the ten different digits.
   $\mathcal{N}$ maps each digit to exactly one value in Val.

2. Induction step: $n \equiv n_1\ d$ for some digit d
   There are ten cases for the induction step. Here, we show the case for a digit d
   - n_1 is a sub-term of n
   - By applying the induction hypothesis to n_1, we get:
     (a) there is exactly one $v_1 \in \mathrm{Val}$ such that $\mathcal{N}[\![n_1]\!] = v_1$
   - The equations for $\mathcal{N}$ define $\mathcal{N}[\![n_1\ d]\!] = \mathcal{N}[\![n_1]\!] \times 10 + v_d$ where v_d is the integer value for digit d. (b) There is exactly one such v_d
   - By using (a) and (b), we get that there is exactly one pair of values $v_1, v_d \in \mathrm{Val}$ such that $\mathcal{N}[\![n_1\ d]\!] = v_1 \times 10 + v_d$
   - Since multiplication and addition are total functions, we can conclude that there is exactly one value for $v_1 \times 10 + v_d$ and, thus, for $\mathcal{N}[\![n_1\ d]\!]$

$\mathcal{A}$ is a Total Function

- Lemma: The equations for $\mathcal{A}$ define a total function $\mathcal{A} : \text{Aexp} \to \text{State} \to \text{Val}$

- To prove the lemma, we show that for each $e \in \text{Aexp}$ and $\sigma \in \text{State}$ there is exactly one $v \in \text{Val}$ such that $\mathcal{A}[\![e]\!]\sigma = v$

- $\mathcal{N}$ is defined inductively:

$$
\boxed{
\begin{array}{ll}
\mathcal{A}[\![x]\!]\sigma & = \sigma(x) \\
\mathcal{A}[\![n]\!]\sigma & = \mathcal{N}[\![n]\!] \\
\mathcal{A}[\![e_1 \; op \; e_2]\!]\sigma & = \mathcal{A}[\![e_1]\!]\sigma \; \overline{op} \; \mathcal{A}[\![e_2]\!]\sigma \quad \text{for } op \in \text{Op}
\end{array}
}
$$

- Therefore, we can prove the lemma by structural induction on e

Proof: $\mathcal{A}$ is a Total Function

1. Induction base: all terms of height 1
   - Case 1: $e \equiv n$
     The equations define $\mathcal{A}[\![n]\!]\sigma = \mathcal{N}[\![n]\!]$. According to the previous lemma, $\mathcal{N}$ is a total function and, thus, $\mathcal{N}[\![n]\!]$ yields exactly one value in Val
   - Case 2: $e \equiv x$
     The equations define $\mathcal{A}[\![x]\!]\sigma = \sigma(x)$. σ is a total function, $\sigma(x) \in \mathrm{Val}$

2. Induction step: $e \equiv e_1 \ op \ e_2$
   - e_1 and e_2 are sub-terms of e
   - By applying the induction hypothesis to e_1 and e_2, we get:
     (a) there is exactly one $v_1 \in \mathrm{Val}$ such that $\mathcal{A}[\![e_1]\!]\sigma = v_1$ and
     (b) there is exactly one $v_2 \in \mathrm{Val}$ such that $\mathcal{A}[\![e_2]\!]\sigma = v_2$ and
   - The equations for $\mathcal{A}$ define $\mathcal{A}[\![e_1 \ op \ e_2]\!]\sigma = \mathcal{A}[\![e_1]\!]\sigma \ \overline{op} \ \mathcal{A}[\![e_2]\!]\sigma$
   - By using (a) and (b), we get that there is exactly one pair of values $v_1, v_2 \in \mathrm{Val}$ such that $\mathcal{A}[\![e_1 \ op \ e_2]\!]\sigma = v_1 \ \overline{op} \ v_2$
   - Since $\overline{op}$ is a total function (addition, subtraction, or multiplication), we can conclude that there is exactly one value for $v_1 \ \overline{op} \ v_2$ and, thus, for $\mathcal{A}[\![e_1 \ op \ e_2]\!]\sigma$

Inductive Definitions: Example

New arithmetic expression: $-e$

- Inductive definition of $\mathcal{A}[\![-e]\!]\sigma$

$$\boxed{\mathcal{A}[\![-e]\!]\sigma = 0 - \mathcal{A}[\![e]\!]\sigma}$$

- e is a subterm of $-e$
- For the induction step we may assume the induction hypothesis for e

- Non-inductive definition of $\mathcal{A}[\![-e]\!]\sigma$

$$\boxed{\mathcal{A}[\![-e]\!]\sigma = \mathcal{A}[\![0-e]\!]\sigma}$$

- $0-e$ is not a subterm of $-e$
- For the induction step we may not assume the induction hypothesis for $0-e$

Free Variables

Arithmetic expressions

$$\begin{aligned} FV(e_1\ op\ e_2) &= FV(e_1) \cup FV(e_2) \\ FV(n) &= \emptyset \\ FV(x) &= \{x\} \end{aligned}$$

Boolean expressions

$$\begin{aligned} FV(b_1\ op\ b_2) &= FV(b_1) \cup FV(b_2), op \in \mathrm{RelOp} \\ FV(\texttt{not}\ b) &= FV(b) \\ FV(b_1\ \texttt{or}\ b_2) &= FV(b_1) \cup FV(b_2) \\ FV(b_1\ \texttt{and}\ b_2) &= FV(b_1) \cup FV(b_2) \end{aligned}$$

Statements

$$\begin{aligned} FV(\texttt{skip}) &= \emptyset \\ FV(x\texttt{:=}e) &= \{x\} \cup FV(e) \\ FV(s_1\texttt{;}s_2) &= FV(s_1) \cup FV(s_2) \\ FV(\texttt{if}\ b\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{end}) &= FV(b) \cup FV(s_1) \cup FV(s_2) \\ FV(\texttt{while}\ b\ \texttt{do}\ s\ \texttt{end}) &= FV(b) \cup FV(s) \end{aligned}$$

Syntactic Abbreviations

`if` b `then` s `end`	`if` b `then` s `else skip end`
`repeat` s `until` b	s`; while not` b `do` s `end`

`for` x `:=` e_1 `to` e_2 `do` s `end`
$x \notin FV(e_2), y \notin FV(s)$
	x `:=` e_1`;`
	`var` y `:=` e_2 `in`
	   `while` x `<=` y `do`
	      s`;` x `:=` x `+ 1`
	   `end`
	`end`

`true`	`1=1`