Semantics of Programming Languages

Introduction

Prof. Peter Müller

Software Component Technology

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Agenda for Today

1. Introduction
   - 1.1 Motivation
   - 1.2 Overview
   - 1.3 Course Outline
   - 1.4 The Language IMP
   - 1.5 Semantics of Expressions
   - 1.6 Properties of the Semantics

Objectives

- Motivation for formal language semantics
- Foundations for the rest of the course

1. Introduction

C: Expression Evaluation

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

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

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

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 ( div 2 0 )
```

SML

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

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

Haskell and SML: Evaluation

Haskell

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

```
const ( div 2 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 C2 {
  int x;
  public    void inc1( )
    { this.inc2( ); }
  protected void inc2( )
    { x++; }
}
```

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

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

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

```
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

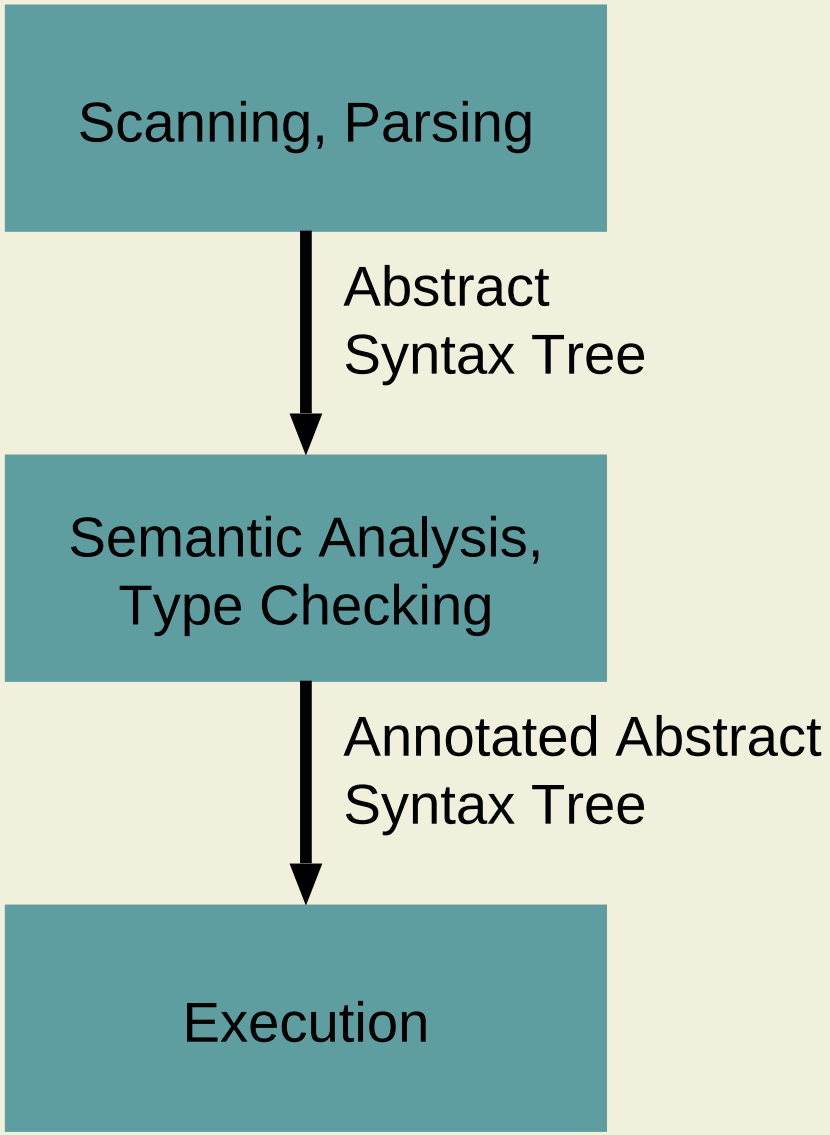
- Verification could run by induction
- Induction hypothesis: $n \geq 0 \Rightarrow fac(n) = n!$
- Induction base is trivial
- Induction step requires to prove $n \times (n-1)! = n!$ which is not the case in computer arithmetic

1. Introduction

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
  - **Specifies 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 )
```

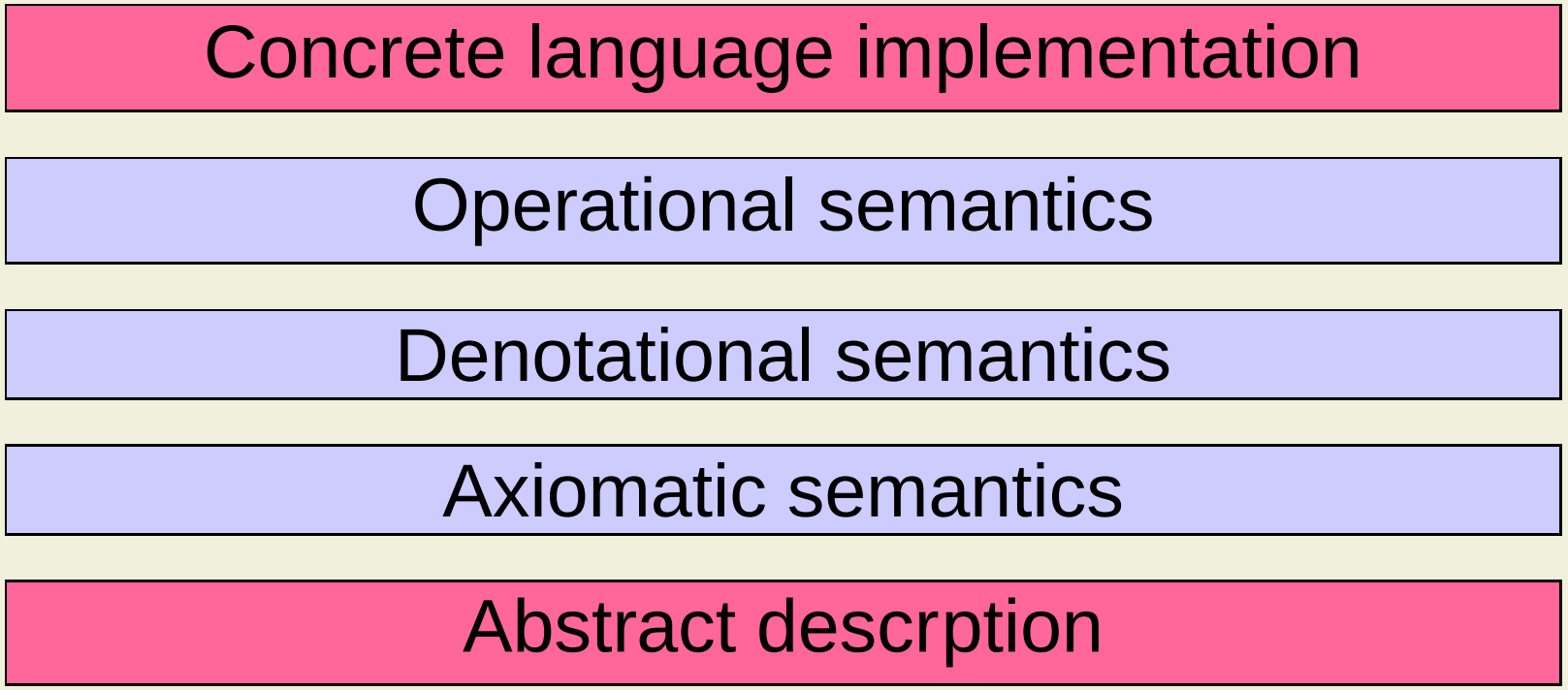
- "The program computes a partial function from states to states: the final state will be equal to the initial state except that the value of x will be 1 and the value of y will be equal to the factorial of the value of x in the initial state"
- Two kinds of denotational semantics
  - Direct Style Semantics
  - 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



Selection Criteria

- Constructs of the programming language
  - Imperative
  - Functional
  - Concurrent
  - Object-oriented
  - Non-deterministic
  - Etc.
- Application of the semantics
  - Understanding the language
  - Program verification
  - Prototyping
  - Compiler construction
  - Program analysis
  - Etc.

1. Introduction

After this Course, you should

- Understand the fundamental ideas behind the three major approaches to semantics
- Be able to compare the approaches and understand their relationship
- Be able to apply formal semantics as a tool
- Have a better understanding of programming languages in general

Approach

- We discuss the major approaches to semantics for a small imperative language IMP
  - Similarities and differences
  - Applications
  - Important theoretical results
- We discuss the semantics of a simple object-oriented language COOL
  - Advanced language features

Course Outline

1. Introduction
2. Operational Semantics
   - Natural and structural operational semantics of IMP
   - Equivalence
3. Denotational Semantics of IMP
   - Direct style denotational semantics of IMP
   - Equivalence of denotational and operational semantics
4. Axiomatic Semantics
   - Axiomatic semantics of IMP
   - Soundness and completeness
5. Semantics of COOL

Organization

- Exam: There will be a 90 minute **written exam** in the exam period
  - Contents of exercises are crucial for the exam

- Web site: sct.inf.ethz.ch/teaching/ss2004/sps/index.html
  - Check regularly for announcements
  - Slides will be available three days before the lecture. Please bring a hardcopy to the lecture

Using this Course

- Master Program:
  This course is a **Specialized Course** (Vertiefungsfach) for the Major in Software Engineering

- Diploma Program:
  This course is a **Specialized Course** (Vertiefungsfach)

Related Courses

- Abstract State Machines (Stärk, SS)
- Informatik III (Basin, Stärk, WS)
- Konzepte objektorientierter Programmierung (Müller, WS)
- Trusted Components (Meyer, WS)
- Compiler Design I and II (Th. Gross)
- Formal Verification (Biere, WS)

Related Seminars

- Specification and Verification of Object-Oriented Software (Biere, Müller, WS)
- References and Aliasing in Object-Oriented Programs (Biere, Müller, SS)
- FATS Formal Approaches to Software (Biere, Meyer, Müller, Stärk)

Literature

- Hanne Riis Nielson, Flemming Nielson. Semantics with Applications: A Formal Introduction. Wiley, 1992.

  `www.daimi.au.dk/~bra8130/Wiley_book/wiley.html`

- Glynn Winskel. The Formal Semantics of Programming Languages: an Introduction. The MIT Press, 1993.

- Betrand Meyer. Introduction to the Theory of Programming Languages. Prentice Hall, 1990.

- **See course web page for a comprehensive list**

1. Introduction

The Language IMP

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

Syntax of IMP: Characters and Tokens

Characters

```
Letter  = 'A' ... 'Z' | 'a' ... 'z'
Digit   = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
```

Tokens

```
Ident   = Letter { Letter | Digit }
Integer = Digit { Digit }
Var     = Ident
```

Syntax of IMP: Expressions

Arithmetic expressions

```
Aexp  = Aexp Op Aexp | Var | Integer
Op    = '+' | '-' | '*' | '/' | 'mod'
```

Boolean expressions

```
Bexp    = Bexp 'or' Bexp | Bexp 'and' Bexp
        | 'not' Bexp | Aexp RelOp Aexp
RelOp   = '=' | '#' | '<' | '<=' | '>' | '>='
```

Syntax of IMP: Statemens

```
Stm  = 'skip'
     | Var ':=' Aexp
     | Stm ';' Stm
     | 'if' Bexp 'then' Stm 'else' Stm 'end'
     | 'while' Bexp 'do' Stm 'end'
```

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

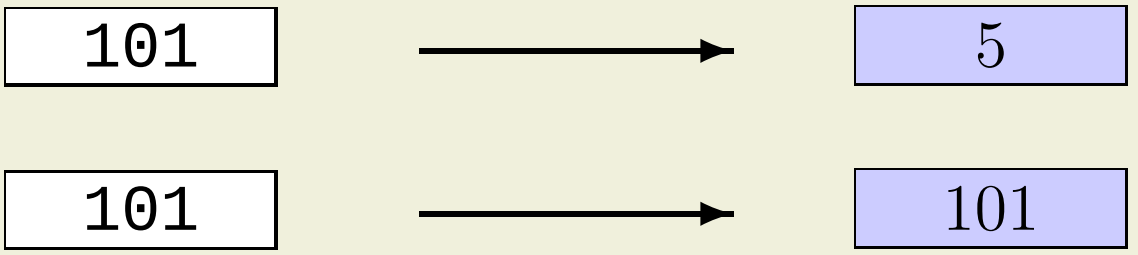
Syntax of IMP: Example

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

1. Introduction

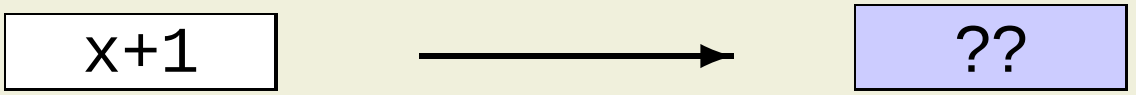
Semantic Categories

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



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

States



- 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

$$\text{State} : \text{Var} \longrightarrow \text{Val}$$

- We represent a state σ as a finite function

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

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

$$\mathcal{A} : \text{Aexp} \longrightarrow \text{State} \longrightarrow \text{Val}$$

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

$$\begin{array}{lll} \mathcal{A}[\![x]\!]\sigma & = \sigma(x) & \\ \mathcal{A}[\![i]\!]\sigma & = i & \text{for } i \in \mathbb{Z} \\ \mathcal{A}[\![e_1\ op\ e_2]\!]\sigma & = \mathcal{A}[\![e_1]\!]\sigma\ \overline{op}\ \mathcal{A}[\![e_2]\!]\sigma & \text{for } op \in \text{Op} \end{array}$$

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

Semantics of Boolean Expressions

The semantic function

$$\mathcal{B} : \text{Bexp} \longrightarrow \text{State} \longrightarrow \text{Bool}$$

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

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

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

Boolean Expressions (cont'd)

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

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

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

1. Introduction

Mathematical Induction

Principle of mathematical induction

$$
\begin{array}{l}
(P(0) \wedge \\
(\forall m \in \mathbb{N} : P(m) \Rightarrow P(m+1))) \\
\Leftrightarrow \forall n \in \mathbb{N} : P(n)
\end{array}
$$

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

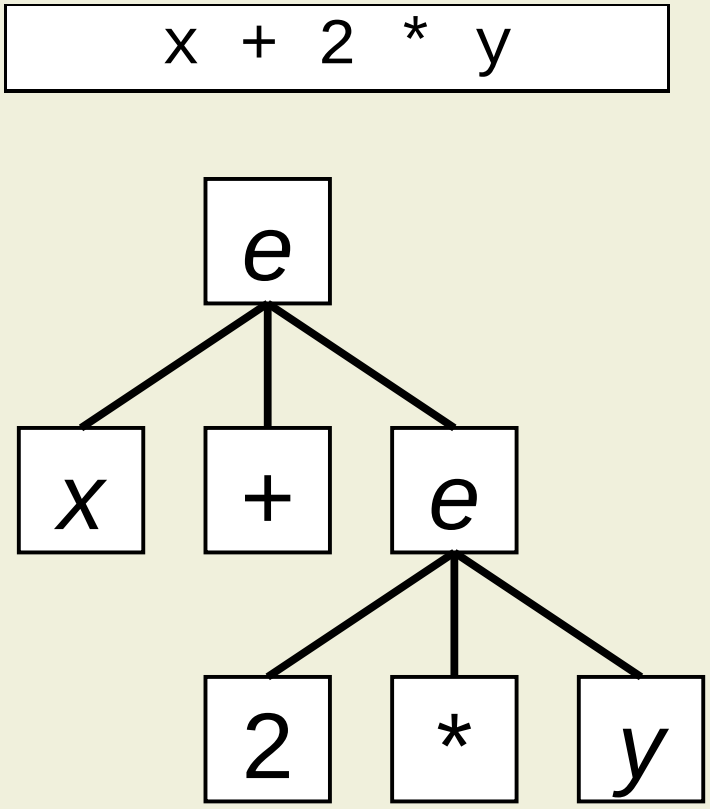
- Principle of well-founded induction

> 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 $m = n + 1$

Well-Founded Relations on Trees

- The syntactic categories are specified by an abstract syntax giving a **unique decomposition** of each element into its constituents
- For each syntactic category we can define a well-founded relation, for instance, $<$ on the height of the abstract syntax tree



Structural Induction

Structural induction on syntactic categories

1. **Induction base**: Prove that the property holds for all the **basis** elements of the syntactic category
2. **Induction step**: Prove that the property holds for all the **composite** elements of the syntactic category
   - **Induction hypothesis**: Assume that the property holds for all the immediate constituents of the elements
   - Prove that it also holds for the element itself

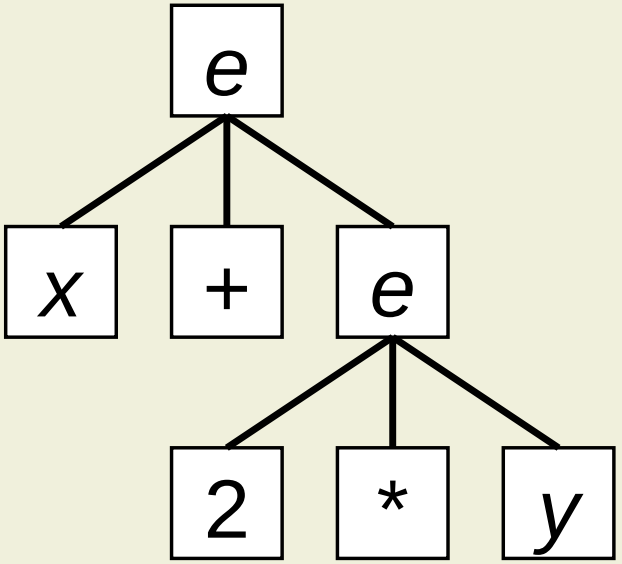
Structural induction is a special case of **well-founded induction**

Structural Induction: Example

Structural induction for arithmetic expressions

x + 2 * y

$$(\forall i \in \text{Integer} : P(i)) \wedge$$
$$(\forall x \in \text{Var} : P(x)) \wedge$$
$$(\forall e_1, e_2 \in \text{Aexp} : P(e_1) \wedge P(e_2) \Rightarrow P(e_1\ op\ e_2))$$
$$\Leftrightarrow$$
$$\forall e \in \text{Aexp} : P(e)$$



Compositional Definitions

- The semantics is defined by **compositional definitions** of functions
  - The values for the basis elements are defined directly
  - The values for composite elements are defined in terms of the immediate constituents

$$
\begin{array}{lll}
\mathcal{A}[\![x]\!]\sigma & = \sigma(x) & \\
\mathcal{A}[\![i]\!]\sigma & = i & \text{for } i \in \mathbb{Z} \\
\mathcal{A}[\![e_1\ op\ e_2]\!]\sigma & = \mathcal{A}[\![e_1]\!]\sigma\ \overline{op}\ \mathcal{A}[\![e_2]\!]\sigma & \text{for } op \in \text{Op}
\end{array}
$$

  - Since the decomposition of the elements is unique this means that the semantics is well-defined

- Compositional definitions are also called **inductive definitions** or **well-founded recursion**

Using Structural Induction

- Compositional definitions enable proofs by structural induction
- Lemma: The equations for $\mathcal{A}$ define a total function $\mathcal{A} : \text{Aexp} \longrightarrow \text{State} \longrightarrow \text{Val}$
- 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 $\mathcal{A}[\![e]\!]\sigma = v$

Proof

1. Induction base
   - Case 1: $e \equiv i$
     The equations define $\mathcal{A}[\![i]\!]\sigma = i$, $i \in$ Val
   - Case 2: $e \equiv x$
     The equations define $\mathcal{A}[\![x]\!]\sigma = \sigma(x)$
     σ is a total function, $\sigma(x) \in$ Val
2. Induction step: $e \equiv e_1\ op\ e_2$
   - The equations define $\mathcal{A}[\![e_1\ op\ e_2]\!]\sigma = \mathcal{A}[\![e_1]\!]\sigma\ \overline{op}\ \mathcal{A}[\![e_2]\!]\sigma$
   - There is exactly one value for $\mathcal{A}[\![e_1]\!]\sigma$ and $\mathcal{A}[\![e_2]\!]\sigma$, respectively (induction hypothesis)

Compositional Definitions: Example

New arithmetic expression: $\texttt{-}e$

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

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

- e is **an immediate constituent** of $\texttt{-}e$
- For the induction step we **may assume the induction hypothesis** for e

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

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

- $\texttt{0-}e$ is **no immediate constituent** of $\texttt{-}e$
- For the induction step we **may not assume the induction hypothesis** for $\texttt{0-}e$

Free Variables in Expressions

Arithmetic expressions

$$
\begin{aligned}
FV(e_1\ op\ e_2) &= FV(e_1) \cup FV(e_2) \\
FV(i) &= \emptyset, \quad i \text{ is an integer} \\
FV(x) &= \{x\}
\end{aligned}
$$

Boolean expressions

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

Free Variables in 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`