Design and Selection of Sequential Programming Languages

SFWR ENG 3E03, Fall 2003

WOLFRAM KAHL

Department of Computing and Software
Faculty of Engineering
McMaster University
Lecture 1

Semantics
Syntax and Semantics

Syntax — *Shape* of PL constructs

- What are the *tokens* of the language? — *Lexical* syntax, “word level”
- How are programs built from tokens? — Mostly use **Context-Free Grammars** (CFG) or **Backus-Naur-Form** (BNF) to describe syntax at the “sentence level”

“**Static semantics**: aspects of program structure that are checked at compile time, but cannot be captured by CFGs (→ context-sensitive syntax):

- Scopes of names
- Typing

Semantics — *Meaning* of PL constructs

Three major approaches:

- **Axiomatic semantics**: \( \{p\} \text{Prog} \{q\} \)
- **Denotational semantics**: \( \text{Prog} \) *denotes a mathematical function* \( [[\text{Prog}]] \)
- **Operational semantics**: state transitions of an abstract machine
Simple Semantic Domains

From the textbook:

A semantic domain is any set whose properties and operations are independently well-understood and upon which the functions that define the semantics of a language are ultimately based.

**Primitive domains:** \( \mathbb{B} = \{ \text{True, False} \}, \ \mathbb{N}, \ \mathbb{Z}, \ \text{Char}, \ \text{seq Char}, \ \text{Ident} \)

**Domains for Program States:**

- **Locations** are usually natural numbers: \( \text{Loc} = \mathbb{N} \)
- **Values** are, in a simple context, integers: \( \text{Val}_0 = \mathbb{Z} \)
- **Memory states** can be considered as partial functions: \( \text{Mem}_0 = \mathbb{N} \rightarrow \text{Val}_0 \)
- **Simple environments** are partial functions, too: \( \text{Env}_0 = \text{Ident} \rightarrow \text{Loc} \)
- A simple **state** is pair: \( \text{State}_0 = \text{Env}_0 \times \text{Mem}_0 \)
- A **simple store** directly maps identifiers to values: \( \text{Store}_0 = \text{Ident} \rightarrow \text{Val}_0 \)
Relation Overriding

Given $Q, R : A \leftrightarrow B$.

The relation $Q \oplus R$ relates everything in the domain of $R$ to the same objects as $R$ does, and everything else in the domain of $Q$ to the same objects as $Q$ does.

$$Q \oplus R = \{ (x, y) : Q \mid x \not\in \text{dom } R \} \cup R$$
Relation Overriding

Given $Q, R : A \leftrightarrow B$.

The relation $Q \oplus R$ relates everything in the domain of $R$ to the same objects as $R$ does, and everything else in the domain of $Q$ to the same objects as $Q$ does.

$$Q \oplus R = \{(x, y) : Q \mid x \notin \text{dom } R\} \cup R$$

- $\oplus$ is not commutative
Relation Overriding

Given $Q, R : A \leftrightarrow B$.

The relation $Q \oplus R$ relates everything in the domain of $R$ to the same objects as $R$ does, and everything else in the domain of $Q$ to the same objects as $Q$ does.

$$Q \oplus R = \{(x, y) : Q \mid x \not\in \text{dom } R\} \cup R$$

- $\oplus$ is not commutative

- **Textbook**: “overriding union” operator “$\overline{\cup}$”
Relation Overriding

Given $Q, R : A \leftrightarrow B$.

The relation $Q \oplus R$ relates everything in the domain of $R$ to the same objects as $R$ does, and everything else in the domain of $Q$ to the same objects as $Q$ does.

$$Q \oplus R = \{(x, y) : Q \mid x \notin \text{dom } R\} \cup R$$

- $\oplus$ is **not commutative**

- **Textbook:** “overriding union” operator “$\overline{U}$”

- **Haskell:**

  $$\text{addListToFM} :: \text{Ord } \text{key} \Rightarrow \text{FiniteMap } k \text{ v} \rightarrow [ (k, v)] \rightarrow \text{FiniteMap } k \text{ v}$$
Relation Overriding

Given $Q, R : A \leftrightarrow B$.

The relation $Q \oplus R$ relates everything in the domain of $R$ to the same objects as $R$ does, and everything else in the domain of $Q$ to the same objects as $Q$ does.

$$Q \oplus R = \{(x, y) : Q \mid x \notin \text{dom } R\} \cup R$$

- $\oplus$ is not commutative

- **Textbook:** “overriding union” operator “$\bar{U}$”

- **Haskell:**

  $$\text{addListToFM} :: \text{Ord } \text{key} \Rightarrow \text{ FiniteMap } k \text{ v} \rightarrow \left[ (k, v) \right] \rightarrow \text{ FiniteMap } k \text{ v}$$

- If $Q$ and $R$ are both partial functions, then $Q \oplus R$ is a partial function, too.
Relation Overriding

Given $Q, R : \mathcal{A} \leftrightarrow \mathcal{B}$.

The relation $Q \oplus R$ relates everything in the domain of $R$ to the same objects as $R$ does, and everything else in the domain of $Q$ to the same objects as $Q$ does.

$$Q \oplus R = \{(x, y) : Q \mid x \notin \text{dom } R\} \cup R$$

- $\oplus$ is not commutative

- **Textbook:** “overriding union” operator “$\overline{U}$”

- **Haskell:**
  
  ```haskell```
  `addListToFM :: Ord key \Rightarrow FiniteMap k v \rightarrow [(k, v)] \rightarrow FiniteMap k v`
  ```haskell```

- If $Q$ and $R$ are both partial functions, then $Q \oplus R$ is a partial function, too.

- $\oplus$ is used to model
  - writing into memory or store locations
  - insertion into environments (shadowing previous bindings)
Operational Semantics

Two kinds of assertions:

— Evaluating expression $e$ starting in store $\sigma$ produces value $v$ 

\[ \sigma(e) \Rightarrow v \]
Operational Semantics

Two kinds of assertions:

— Evaluating expression \( e \) starting in store \( \sigma \) produces value \( v \)
  \[ \sigma(e) \Rightarrow v \]

— Execution of statement \( s \) starting in store \( \sigma_1 \) results in store \( \sigma_2 \)
  \[ \sigma_1(s) \Rightarrow \sigma_2 \]
Operational Semantics

Two kinds of assertions:

— Evaluating expression $e$ starting in store $\sigma$ produces value $v$  \[ \sigma(e) \Rightarrow v \]

— Execution of statement $s$ starting in store $\sigma_1$ results in store $\sigma_2$  \[ \sigma_1(s) \Rightarrow \sigma_2 \]
Operational Semantics

Two kinds of assertions:

— Evaluating expression \( e \) starting in store \( \sigma \) produces value \( v \) \( \sigma(e) \Rightarrow v \)

— Execution of statement \( s \) starting in store \( \sigma_1 \) results in store \( \sigma_2 \) \( \sigma_1(s) \Rightarrow \sigma_2 \)

Execution axioms: \( \sigma(c) \Rightarrow c \) \( \sigma(v) \Rightarrow \sigma v \) if \( v \in \text{dom } \sigma \)
Operational Semantics

Two kinds of assertions:

— Evaluating expression $e$ starting in store $\sigma$ produces value $v$  
\[ \sigma(e) \Rightarrow v \]

— Execution of statement $s$ starting in store $\sigma_1$ results in store $\sigma_2$  
\[ \sigma_1(s) \Rightarrow \sigma_2 \]

Execution axioms:  
\[ \sigma(c) \Rightarrow c \]
\[ \sigma(v) \Rightarrow \sigma v \quad \text{if } v \in \text{dom } \sigma \]

Execution rules:  
\[ \frac{\text{premise}}{\text{conclusion}} \quad \text{or} \quad \frac{\text{premise}_1 \ldots \text{premise}_n}{\text{conclusion}} \]
Operational Semantics

Two kinds of assertions:

— Evaluating expression \( e \) starting in store \( \sigma \) produces value \( v \) \( \sigma(e) \Rightarrow v \)
— Execution of statement \( s \) starting in store \( \sigma_1 \) results in store \( \sigma_2 \) \( \sigma_1(s) \Rightarrow \sigma_2 \)

Execution axioms: \( \sigma(c) \Rightarrow c \quad \sigma(v) \Rightarrow \sigma v \quad \text{if } v \in \text{dom } \sigma \)

Execution rules:

\[
\frac{\text{premise}}{\text{conclusion}} \quad \text{or} \quad \frac{\text{premise}_1 \ldots \text{premise}_n}{\text{conclusion}}
\]

Example rule — addition:

\[
\sigma(e_1) \Rightarrow v_1 \quad \sigma(e_2) \Rightarrow v_2 \quad \sigma(e_1 + e_2) \Rightarrow v_1 + v_2
\]
Operational Semantics

Two kinds of assertions:
— Evaluating expression \( e \) starting in store \( \sigma \) produces value \( v \) \( \sigma(e) \Rightarrow v \)
— Execution of statement \( s \) starting in store \( \sigma_1 \) results in store \( \sigma_2 \) \( \sigma_1(s) \Rightarrow \sigma_2 \)

Execution axioms: \( \sigma(c) \Rightarrow c \quad \sigma(v) \Rightarrow \sigma v \) if \( v \in \text{dom} \sigma \)

Execution rules:

\[
\begin{align*}
\text{premise} & \quad \text{or} \quad \text{premise}_1 \ldots \text{premise}_n \\
\hline
\text{conclusion}
\end{align*}
\]

Example rule — addition:

\[
\sigma(e_1) \Rightarrow v_1 \quad \sigma(e_2) \Rightarrow v_2
\]

\[
\sigma(e_1 + e_2) \Rightarrow v_1 + v_2
\]

(The left “+” is syntax, the right “+” is a mathematical operation on numbers.)
Mechanized Operational Semantics: Interpreter

Two kinds of assertions:
— Evaluating expression $e$ starting in store $\sigma$ produces value $v$  \[ \sigma(e) \Rightarrow v \]
— Execution of statement $s$ starting in store $\sigma_1$ results in store $\sigma_2$  \[ \sigma_1(s) \Rightarrow \sigma_2 \]
Mechanized Operational Semantics: Interpreter

Two kinds of assertions:
— Evaluating expression $e$ starting in store $\sigma$ produces value $\nu$ $\sigma(e) \Rightarrow \nu$
— Execution of statement $s$ starting in store $\sigma_1$ results in store $\sigma_2$ $\sigma_1(s) \Rightarrow \sigma_2$

This notation stands for two \textit{ternary relations}, which are \textit{partial functions} for deterministic programming languages
Mechanized Operational Semantics: Interpreter

Two kinds of assertions:
— Evaluating expression $e$ starting in store $\sigma$ produces value $v$ 
  \[ \sigma(e) \Rightarrow v \]
— Execution of statement $s$ starting in store $\sigma_1$ results in store $\sigma_2$ 
  \[ \sigma_1(s) \Rightarrow \sigma_2 \]

This notation stands for two *ternary relations*, which are *partial functions* for deterministic programming languages:

<table>
<thead>
<tr>
<th>Expression evaluation:</th>
<th>Statement execution:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$eval : State_1 \times Expr \rightarrow Val_1$</td>
<td></td>
</tr>
<tr>
<td>$exec : State_1 \times Stmt \rightarrow State_1$</td>
<td></td>
</tr>
</tbody>
</table>
Mechanized Operational Semantics: Interpreter

Two kinds of assertions:

— Evaluating expression $e$ starting in store $\sigma$ produces value $v$ $\quad \sigma(e) \Rightarrow v$
— Execution of statement $s$ starting in store $\sigma_1$ results in store $\sigma_2$ $\quad \sigma_1(s) \Rightarrow \sigma_2$

This notation stands for two ternary relations, which are partial functions for deterministic programming languages:

— expression evaluation: $eval : State_1 \times Expr \rightarrow Val_1$
— statement execution: $exec : State_1 \times Stmt \rightarrow State_1$

Note: one syntactic and one semantic argument.
Mechanized Operational Semantics: Interpreter

Two kinds of assertions:

— Evaluating expression \( e \) starting in store \( \sigma \) produces value \( v \)

\[ \sigma(e) \Rightarrow v \]

— Execution of statement \( s \) starting in store \( \sigma_1 \) results in store \( \sigma_2 \)

\[ \sigma_1(s) \Rightarrow \sigma_2 \]

This notation stands for two **ternary relations**, which are **partial functions** for deterministic programming languages:

— expression evaluation: \( \text{eval} : \text{State}_1 \times \text{Expr} \rightarrow \text{Val}_1 \)

— statement execution: \( \text{exec} : \text{State}_1 \times \text{Stmt} \rightarrow \text{State}_1 \)

**Note:** one **syntactic** and one **semantic** argument.

Two **interpreter functions** (assuming **deterministic** semantics):

\( \text{evalExpr} :: \text{Expression} \rightarrow \text{State}_1 \rightarrow \text{Maybe Value}_1 \)

\( \text{interpStmt} :: \text{Statement} \rightarrow \text{State}_1 \rightarrow \text{Maybe State}_1 \)
Mechanized Operational Semantics: Interpreter

Two kinds of assertions:
— Evaluating expression $e$ starting in store $\sigma$ produces value $v$ \( \sigma(e) \Rightarrow v \)
— Execution of statement $s$ starting in store $\sigma_1$ results in store $\sigma_2$ \( \sigma_1(s) \Rightarrow \sigma_2 \)

This notation stands for two ternary relations, which are partial functions for deterministic programming languages:
— expression evaluation: \( \text{eval} : State_1 \times Expr \rightarrow Val_1 \)
— statement execution: \( \text{exec} : State_1 \times Stmt \rightarrow State_1 \)

*Note:* one syntactic and one semantic argument.

Two interpreter functions (assuming deterministic semantics):

\( \text{evalExpr} :: Expression \rightarrow State1 \rightarrow \text{Maybe Value1} \)
\( \text{interpStmt} :: Statement \rightarrow State1 \rightarrow \text{Maybe State1} \)

**data** \( Value1 = \text{VallInt Int} | \text{ValBool Bool} \)

**type** \( State1 = \text{FiniteMap Variable Value1} \) — even simpler than \( State_0 \)
Interpreter: Expression Evaluation

evalExpr :: Expression → State1 → Maybe Value1

data Value1 = VallInt Int
  | ValBool Bool

type State1 = FiniteMap Variable Value1 —— even simpler than State₀
Interpreter: Expression Evaluation

\[
\text{evalExpr} :: \text{Expression} \rightarrow \text{State1} \rightarrow \text{Maybe Value1}
\]

\[
\text{data Value1} = \text{VallInt Int} \\
\hspace{1cm} \mid \text{ValBool Bool}
\]

\[
\text{type State1} = \text{FiniteMap Variable Value1}
\]

\[
\text{evalExpr (Var v) s} =
\]
Interpreter: Expression Evaluation

evalExpr :: Expression → State1 → Maybe Value1

data Value1 = ValInt Int
  | ValBool Bool

type State1 = FiniteMap Variable Value1

evalExpr (Var v) s = lookupFM s v
Interpreter: Expression Evaluation

\[ \text{evalExpr :: Expression} \rightarrow \text{State1} \rightarrow \text{Maybe Value1} \]

data Value1 = ValInt Int
  | ValBool Bool

type State1 = FiniteMap Variable Value1

evalExpr (Var v) s = lookupFM s v
evalExpr (Value (LitInt i)) s = Just (ValInt i)
 evalExpr (Value (LitBool b)) s = Just (ValBool b)
Interpreter: Expression Evaluation

evalExpr :: Expression → State1 → Maybe Value1

data Value1 = ValInt Int
  | ValBool Bool

type State1 = FiniteMap Variable Value1

evalExpr (Var v) s = lookupFM s v

evalExpr (Value (LitInt i)) s = Just (ValInt i)  -- better: function litToVal

evalExpr (Value (LitBool b)) s = Just (ValBool b)
Interpreter: Expression Evaluation

\[ \text{evalExpr} :: \text{Expression} \rightarrow \text{State1} \rightarrow \text{Maybe Value1} \]

\textbf{data} \hspace{1em} \text{Value1} = \text{ValInt Int} \\
\hspace{1em} | \hspace{1em} \text{ValBool Bool} \\

\textbf{type} \hspace{1em} \text{State1} = \text{FiniteMap Variable Value1} \\

\text{evalExpr} \hspace{0.5em} (\text{Var v}) \hspace{0.5em} s = \text{lookupFM} \hspace{0.5em} s \hspace{0.5em} v \\
\text{evalExpr} \hspace{0.5em} (\text{Value} \hspace{0.5em} (\text{LitInt i})) \hspace{0.5em} s = \text{Just} \hspace{0.5em} (\text{ValInt i}) \hspace{2em} \text{better: function litToVal} \\
\text{evalExpr} \hspace{0.5em} (\text{Value} \hspace{0.5em} (\text{LitBool b})) \hspace{0.5em} s = \text{Just} \hspace{0.5em} (\text{ValBool b}) \\
\text{evalExpr} \hspace{0.5em} (\text{Binary} \hspace{0.5em} (\text{MkArithOp Plus}) \hspace{0.5em} e1 \hspace{0.5em} e2) \hspace{0.5em} s = \]
Interpreter: Expression Evaluation

\( \text{evalExpr} :: \text{Expression} \rightarrow \text{State1} \rightarrow \text{Maybe Value1} \)

data \( \text{Value1} = \) VallInt \( \text{Int} \)

| \( \text{ValBool \ Boolean} \)

type \( \text{State1} = \text{FiniteMap Variable Value1} \)

\( \text{evalExpr (Var v)} \) \( s = \) lookupFM \( s \ v \)

\( \text{evalExpr (Value (LitInt \ i))} \) \( s = \) Just \( (\text{VallInt \ i}) \) \( \quad \) better: function \( \text{litToVal} \)

\( \text{evalExpr (Value (LitBool \ b))} \) \( s = \) Just \( (\text{ValBool \ b}) \)

\( \text{evalExpr (Binary (MkArithOp \ Plus) \ e1 \ e2)} \) \( s = \)

\( \text{case} \ (\text{evalExpr \ e1 \ s}, \text{evalExpr \ e2 \ s}) \) \( \text{of} \)
Interpreter: Expression Evaluation

\( \text{evalExpr} :: \text{Expression} \rightarrow \text{State1} \rightarrow \text{Maybe Value1} \)

data Value1 = ValInt Int
  | ValBool Bool

type State1 = FiniteMap Variable Value1  -- even simpler than State_0

evalExpr (Var v) s = lookupFM s v

\( \text{evalExpr (Value (LitInt i)) s} = \text{Just (ValInt i)} \)  -- better: function litToVal

evalExpr (Value (LitBool b)) s = Just (ValBool b)

evalExpr (Binary (MkArithOp Plus) e1 e2) s =
   case (evalExpr e1 s, evalExpr e2 s) of
      (Just (ValInt v1), Just (ValInt v2)) →
**Interpreter: Expression Evaluation**

\[ \text{evalExpr} :: \text{Expression} \rightarrow \text{State1} \rightarrow \text{Maybe Value1} \]

**data** \( \text{Value1} = \text{ValInt} \text{ Int} \)

\( \mid \text{ValBool} \text{ Bool} \)

**type** \( \text{State1} = \text{FiniteMap Variable Value1} \)

\[ \text{evalExpr} (\text{Var} \ v) \ s = \text{lookupFM} \ s \ v \]
\[ \text{evalExpr} (\text{Value} (\text{LitInt} \ i)) \ s = \text{Just} (\text{ValInt} \ i) \quad \text{--- better: function \text{litToVal}} \]
\[ \text{evalExpr} (\text{Value} (\text{LitBool} \ b)) \ s = \text{Just} (\text{ValBool} \ b) \]
\[ \text{evalExpr} (\text{Binary} (\text{MkArithOp Plus}) \ e1 \ e2) \ s = \]
\[ \text{case} (\text{evalExpr} \ e1 \ s, \text{evalExpr} \ e2 \ s) \ \text{of} \]
\[ (\text{Just} (\text{ValInt} \ v1), \text{Just} (\text{ValInt} \ v2)) \rightarrow \text{Just} (\text{ValInt} (v1 + v2)) \]
**Interpreter: Expression Evaluation**

\[\text{evalExpr :: Expression} \rightarrow \text{State1} \rightarrow \text{Maybe Value1}\]

\[\text{data Value1 = ValInt Int} \quad | \quad \text{ValBool Bool}\]

\[\text{type State1 = FiniteMap Variable Value1}\]

\[\text{evalExpr (Var v)} s = \text{lookupFM s v}\]
\[\text{evalExpr (Value (LitInt i))} s = \text{Just (ValInt i)} \quad \text{--- better: function litToVal}\]
\[\text{evalExpr (Value (LitBool b))} s = \text{Just (ValBool b)}\]
\[\text{evalExpr (Binary (MkArithOp Plus) e1 e2)} s = \]
\[\text{case (evalExpr e1 s, evalExpr e2 s)} \text{ of}\]
\[\quad \text{(Just (ValInt v1), Just (ValInt v2))} \rightarrow \text{Just (ValInt (v1 + v2))}\]
\[\quad _\rightarrow \text{Nothing}\]
Interpreter: Expression Evaluation (*Maybe Monad*)

\[ \text{evalExpr :: Expression} \rightarrow \text{State1} \rightarrow \text{Maybe Value1} \]

**data** \( \text{Value1} = \text{ValInt Int} \)

\| \text{ValBool Bool} \)

**type** \( \text{State1} = \text{FiniteMap Variable Value1} \)

\[ \text{evalExpr ( Var v) s} = \text{lookupFM s v} \]
\[ \text{evalExpr ( Value ( LitInt i)) s} = \text{Just ( ValInt i)} \quad \text{— better: function litToVal} \]
\[ \text{evalExpr ( Value ( LitBool b)) s} = \text{Just ( ValBool b)} \]
\[ \text{evalExpr ( Binary ( MkArithOp Plus) e1 e2) s} = \text{do} \]
\[ \text{ValInt v1} \leftarrow \text{evalExpr e1 s} \]
\[ \text{ValInt v2} \leftarrow \text{evalExpr e2 s} \]
\[ \text{Just ( ValInt ( v1 + v2))} \]
Assignment

\[
\sigma(e) \Rightarrow v \\
\sigma(x := e) \Rightarrow \sigma \oplus \{x \mapsto v\}
\]

For example:

- Assume \(\sigma_1 = \{x \mapsto 39, y \mapsto 7\}\)

- Then:

\[
\sigma_1(x := x + 3) \Rightarrow
\]
Assignment

\[
\frac{\sigma(e) \Rightarrow \nu}{\sigma(x := e) \Rightarrow \sigma \oplus \{x \mapsto \nu\}}
\]

For example:

- Assume \( \sigma_1 = \{x \mapsto 39, y \mapsto 7\} \)

- Then:

\[
\frac{\sigma_1(x + 3) \Rightarrow \sigma_1(x := x + 3) \Rightarrow}
\]
Assignment

\[
\sigma(e) \Rightarrow v
\]
\[
\sigma(x := e) \Rightarrow \sigma \oplus \{x \mapsto v\}
\]

For example:

- Assume \(\sigma_1 = \{x \mapsto 39, y \mapsto 7\}\)

- Then:

\[
\sigma_1(x) \Rightarrow \quad \sigma_1(3) \Rightarrow
\]
\[
\sigma_1(x + 3) \Rightarrow
\]
\[
\sigma_1(x := x + 3) \Rightarrow
\]
Assignment

\[
\sigma(e) \Rightarrow v \\
\sigma(x := e) \Rightarrow \sigma \oplus \{x \mapsto v\}
\]

For example:

- Assume \(\sigma_1 = \{x \mapsto 39, y \mapsto 7\}\)
- Then:

\[
\sigma_1(x) \Rightarrow 39 \\
\sigma_1(3) \Rightarrow \\
\sigma_1(x + 3) \Rightarrow \\
\sigma_1(x := x + 3) \Rightarrow
\]
Assignment

\[
\begin{align*}
\sigma(e) & \Rightarrow v \\
\sigma(x := e) & \Rightarrow \sigma \oplus \{x \mapsto v\}
\end{align*}
\]

For example:

- Assume \(\sigma_1 = \{x \mapsto 39, y \mapsto 7\}\)

- Then:

\[
\begin{align*}
\sigma_1(x) & \Rightarrow 39 & \sigma_1(3) & \Rightarrow 3 \\
\sigma_1(x + 3) & \Rightarrow \\
\sigma_1(x := x + 3) & \Rightarrow
\end{align*}
\]
Assignment

\[
\sigma(e) \Rightarrow v \\
\sigma(x := e) \Rightarrow \sigma \oplus \{x \mapsto v\}
\]

For example:

- Assume \(\sigma_1 = \{x \mapsto 39, y \mapsto 7\}\)

- Then:

\[
\begin{align*}
\sigma_1(x) &\Rightarrow 39 \\
\sigma_1(3) &\Rightarrow 3 \\
\sigma_1(x + 3) &\Rightarrow 42 \\
\sigma_1(x := x + 3) &\Rightarrow
\end{align*}
\]
Assignment

\[ \sigma(e) \Rightarrow v \]
\[ \sigma(x := e) \Rightarrow \sigma \oplus \{ x \mapsto v \} \]

For example:

- Assume \( \sigma_1 = \{ x \mapsto 39, y \mapsto 7 \} \)

- Then:

\[
\begin{align*}
\sigma_1(x) & \Rightarrow 39 & \sigma_1(3) & \Rightarrow 3 \\
\sigma_1(x + 3) & \Rightarrow 42 \\
\sigma_1(x := x + 3) & \Rightarrow \{ x \mapsto 42, y \mapsto 7 \}
\end{align*}
\]
Assignment

$$\sigma(e) \Rightarrow v$$

$$\sigma(x := e) \Rightarrow \sigma \oplus \{x \mapsto v\}$$

For example:

- Assume $$\sigma_1 = \{x \mapsto 39, y \mapsto 7\}$$

- Then:

  $$\frac{\sigma_1(x) \Rightarrow 39 \quad \sigma_1(3) \Rightarrow 3}{\sigma_1(x + 3) \Rightarrow 42}$$

  $$\frac{\sigma_1(x := x + 3) \Rightarrow \{x \mapsto 42, y \mapsto 7\}}{\text{since } \sigma_1 \oplus \{x \mapsto 42\} = \{x \mapsto 42, y \mapsto 7\}}$$
Interpreter: Assignment

evalExpr :: Expression → State1 → Maybe Value1
interpStmt :: Statement → State1 → Maybe State1

data Value1 = ValInt Int
    | ValBool Bool

type State1 = FiniteMap Variable Value1

\[
\begin{align*}
\sigma(e) & \Rightarrow v \\
\sigma(x := e) & \Rightarrow \sigma \oplus \{x \mapsto v\}
\end{align*}
\]

interpStmt (Assignment var e) s =
Interpreter: Assignment

evalExpr :: Expression → State1 → Maybe Value1
interpStmt :: Statement → State1 → Maybe State1

data Value1 = ValInt Int  
               | ValBool Bool

type State1 = FiniteMap Variable Value1

\[
\begin{align*}
\sigma(e) & \Rightarrow v \\
\sigma(x := e) & \Rightarrow \sigma \oplus \{x \mapsto v\}
\end{align*}
\]

interpStmt (Assignment var e) s = case evalExpr e s of
  Just val → Just (addToFM s var val)
  Nothing → Nothing
Interpreter: Assignment

evalExpr :: Expression → State1 → Maybe Value1
interpStmt :: Statement → State1 → Maybe State1

data Value1 = ValInt Int
| ValBool Bool
type State1 = FiniteMap Variable Value1

\[
\begin{align*}
\sigma(e) \Rightarrow v \\
\sigma(x := e) \Rightarrow \sigma \oplus \{x \mapsto v\}
\end{align*}
\]

interpStmt (Assignment var e) s = case evalExpr e s of
  Just val → Just (addToFM s var val)
  Nothing → Nothing

(Using the Maybe monad:)

interpStmt (Assignment var e) s = do
  val ← evalExpr e s of
  Just (addToFM s var val)
Sequencing, Conditionals, Loops

\[
\begin{align*}
\sigma_1(s_1) \Rightarrow \sigma_2 & \quad \sigma_2(s_2) \Rightarrow \sigma_3 \\
\sigma_1(s_1; s_2) \Rightarrow \sigma_3
\end{align*}
\]

\[
\begin{align*}
\sigma(b) \Rightarrow \text{True} & \quad \sigma(s_1) \Rightarrow \sigma_1 \\
\sigma(\text{if } b \text{ then } s_1 \text{ else } s_2 \text{ fi}) \Rightarrow \sigma_1
\end{align*}
\]

\[
\begin{align*}
\sigma(b) \Rightarrow \text{False} & \quad \sigma(s_2) \Rightarrow \sigma_2 \\
\sigma(\text{if } b \text{ then } s_1 \text{ else } s_2 \text{ fi}) \Rightarrow \sigma_2
\end{align*}
\]

\[
\begin{align*}
\sigma(b) \Rightarrow \text{True} & \quad \sigma(s) \Rightarrow \sigma_1 & \quad \sigma_1(\text{while } b \text{ do } s \text{ od}) \Rightarrow \sigma_2 \\
\sigma(\text{while } b \text{ do } s \text{ od}) \Rightarrow \sigma_2
\end{align*}
\]

\[
\begin{align*}
\sigma(b) \Rightarrow \text{False} & \\
\sigma(\text{while } b \text{ do } s \text{ od}) \Rightarrow \sigma
\end{align*}
\]
Loop Example

\[ P \equiv \text{while } x < 50 \text{ do } x := 2 * x \text{ od} \]

\[ \{x \mapsto 7\}(P) \Rightarrow \]

\[
\begin{align*}
\sigma(b) &\Rightarrow \text{True} & \sigma(s) &\Rightarrow \sigma_1 & \sigma_1(\text{while } b \text{ do } s \text{ od}) &\Rightarrow \sigma_2 \\
\sigma(\text{while } b \text{ do } s \text{ od}) &\Rightarrow \sigma_2 & \sigma(b) &\Rightarrow \text{False} & \sigma(\text{while } b \text{ do } s \text{ od}) &\Rightarrow \sigma
\end{align*}
\]
Loop Example

\[ P \equiv \textbf{while} \ x < 50 \ \textbf{do} \ x := 2 \times x \ \textbf{od} \]
Loop Example

\[ P \equiv \text{while } x < 50 \text{ do } x := 2 \times x \text{ od} \]

\[
\{ x \mapsto 7 \} (x < 50) \Rightarrow \text{True}
\]

\[
\{ x \mapsto 7 \}(P) \Rightarrow
\]

\[
\sigma(b) \Rightarrow \text{True} \quad \sigma(s) \Rightarrow \sigma_1 \quad \sigma_1(\text{while } b \text{ do } s \text{ od}) \Rightarrow \sigma_2
\]

\[
\sigma(\text{while } b \text{ do } s \text{ od}) \Rightarrow \sigma_2
\]

\[
\sigma(b) \Rightarrow \text{False} \quad \sigma(\text{while } b \text{ do } s \text{ od}) \Rightarrow \sigma
\]
Loop Example

\[ P \equiv \textbf{while } x < 50 \textbf{ do } x := 2 \ast x \textbf{ od} \]

\[
\begin{array}{l}
\{ x \mapsto 7 \} (x < 50) \Rightarrow \text{True} \\
\{ x \mapsto 7 \} (x := 2 \ast x) \Rightarrow \\
\{ x \mapsto 7 \}(P) \Rightarrow \\
\end{array}
\]

\[
\begin{array}{l}
\sigma(b) \Rightarrow \text{True} \\
\sigma(s) \Rightarrow \sigma_1 \\
\sigma_1(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2 \\
\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2 \\
\sigma(b) \Rightarrow \text{False} \\
\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma
\end{array}
\]
Loop Example

\[ P \equiv \textbf{while} \ x < 50 \ \textbf{do} \ x := 2 * x \ \textbf{od} \]
Loop Example

\[ P \equiv \text{while } x < 50 \text{ do } x := 2 \times x \text{ od} \]

\[
\begin{align*}
\{x \mapsto 7\} (x < 50) &\Rightarrow \text{True} \\
\{x \mapsto 7\} (x := 2 \times x) &\Rightarrow \{x \mapsto 14\}
\end{align*}
\]

\[
\{x \mapsto 7\}(P) \Rightarrow
\]

\[
\begin{align*}
\sigma(b) &\Rightarrow \text{True} \quad \sigma(s) \Rightarrow \sigma_1 \quad \sigma_1(\text{while } b \text{ do } s \text{ od}) \Rightarrow \sigma_2 \\
\sigma(\text{while } b \text{ do } s \text{ od}) &\Rightarrow \sigma_2 \\
\sigma(b) &\Rightarrow \text{False} \\
\sigma(\text{while } b \text{ do } s \text{ od}) &\Rightarrow \sigma
\end{align*}
\]
Loop Example

\[ P \equiv \textbf{while} \ x < 50 \ \textbf{do} \ x := 2 \times x \ \textbf{od} \]

\[
\{ x \mapsto 14 \} (x < 50) \Rightarrow \\
\{ x \mapsto 7 \} (x < 50) \Rightarrow \text{True} \quad \{ x \mapsto 7 \} (x := 2 \times x) \Rightarrow \{ x \mapsto 14 \} \\
\{ x \mapsto 7 \} (P) \Rightarrow \\
\]

\[
\sigma(b) \Rightarrow \text{True} \quad \sigma(s) \Rightarrow \sigma_1 \quad \sigma_1(\textbf{while} \ b \ \textbf{do} \ s \ \textbf{od}) \Rightarrow \sigma_2 \\
\sigma(\textbf{while} \ b \ \textbf{do} \ s \ \textbf{od}) \Rightarrow \sigma_2 \\
\sigma(b) \Rightarrow \text{False} \quad \sigma(\textbf{while} \ b \ \textbf{do} \ s \ \textbf{od}) \Rightarrow \sigma
\]
Loop Example

\[
P \equiv \textbf{while } x < 50 \textbf{ do } x := 2 \times x \textbf{ od}
\]

\[
\{x \mapsto 14\} (x < 50) \Rightarrow \text{True}
\]

\[
\{x \mapsto 7\} (x < 50) \Rightarrow \text{True} \quad \{x \mapsto 7\} (x := 2 \times x) \Rightarrow \{x \mapsto 14\}
\]

\[
\{x \mapsto 7\}(P) \Rightarrow
\]

\[
\sigma(b) \Rightarrow \text{True} \quad \sigma(s) \Rightarrow \sigma_1 \quad \sigma_1(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2
\]

\[
\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2 \
\]

\[
\sigma(b) \Rightarrow \text{False} \quad \sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma
\]
Loop Example

\[ P \equiv \textbf{while } x < 50 \textbf{ do } x := 2 \times x \textbf{ od} \]

\[
\begin{align*}
\{x \mapsto 14\} (x < 50) & \Rightarrow \text{True} & \{x \mapsto 14\} (x := 2 \times x) & \Rightarrow \\
\{x \mapsto 7\} (x < 50) & \Rightarrow \text{True} & \{x \mapsto 7\} (x := 2 \times x) & \Rightarrow \{x \mapsto 14\} & \{x \mapsto 14\}(P) & \Rightarrow \\
\{x \mapsto 7\}(P) & \Rightarrow
\end{align*}
\]

\[
\begin{align*}
\sigma(b) & \Rightarrow \text{True} & \sigma(s) & \Rightarrow \sigma_1 & \sigma_1(\textbf{while } b \textbf{ do } s \textbf{ od}) & \Rightarrow \sigma_2 & \sigma(b) & \Rightarrow \text{False} \\
\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) & \Rightarrow \sigma_2 & \sigma(b) & \Rightarrow \text{False}
\end{align*}
\]
Loop Example

\[ P \equiv \textbf{while } x < 50 \textbf{ do } x := 2 \ast x \textbf{ od} \]

\[
\begin{align*}
\{x \mapsto 14\} (x < 50) &\Rightarrow \text{True} & \{x \mapsto 14\} (x := 2 \ast x) &\Rightarrow \{x \mapsto 28\} \\
\{x \mapsto 7\} (x < 50) &\Rightarrow \text{True} & \{x \mapsto 7\} (x := 2 \ast x) &\Rightarrow \{x \mapsto 14\} \\
\{x \mapsto 7\}(P) &\Rightarrow \text{False}
\end{align*}
\]

\[
\begin{align*}
\sigma(b) &\Rightarrow \text{True} & \sigma(s) &\Rightarrow \sigma_1 & \sigma_1(\textbf{while } b \textbf{ do } s \textbf{ od}) &\Rightarrow \sigma_2 \\
\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) &\Rightarrow \sigma_2 & \sigma(b) &\Rightarrow \text{False} & \sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) &\Rightarrow \sigma
\end{align*}
\]
Loop Example

\[ P \equiv \textbf{while} \ x < 50 \ \textbf{do} \ x := 2 \times x \ \textbf{od} \]

\[
\begin{align*}
\{ x \mapsto 14 \} \ (x < 50) & \Rightarrow \text{True} & \{ x \mapsto 14 \} \ (x := 2 \times x) & \Rightarrow \{ x \mapsto 28 \} & \{ x \mapsto 28 \} (P) & \Rightarrow \\
\{ x \mapsto 7 \} \ (x < 50) & \Rightarrow \text{True} & \{ x \mapsto 7 \} \ (x := 2 \times x) & \Rightarrow \{ x \mapsto 14 \} & \{ x \mapsto 14 \} (P) & \Rightarrow \\
& & \{ x \mapsto 7 \} (P) & \Rightarrow \\
\end{align*}
\]

\[
\begin{align*}
\sigma(b) & \Rightarrow \text{True} & \sigma(s) & \Rightarrow \sigma_1 & \sigma_1(\textbf{while} \ b \ \textbf{do} \ s \ \textbf{od}) & \Rightarrow \sigma_2 \\
\sigma(\textbf{while} \ b \ \textbf{do} \ s \ \textbf{od}) & \Rightarrow \sigma_2 & \sigma(b) & \Rightarrow \text{False} & \sigma(\textbf{while} \ b \ \textbf{do} \ s \ \textbf{od}) & \Rightarrow \sigma
\end{align*}
\]
Loop Example

\[ P \equiv \textbf{while } x < 50 \textbf{ do } x := 2 \ast x \textbf{ od} \]

\[ \{x \mapsto 28\} (x < 50) \Rightarrow \]

\[ \{x \mapsto 14\} (x < 50) \Rightarrow \text{True} \]
\[ \{x \mapsto 14\} (x := 2 \ast x) \Rightarrow \{x \mapsto 28\} \]
\[ \{x \mapsto 28\}(P) \Rightarrow \]

\[ \{x \mapsto 7\} (x < 50) \Rightarrow \text{True} \]
\[ \{x \mapsto 7\} (x := 2 \ast x) \Rightarrow \{x \mapsto 14\} \]
\[ \{x \mapsto 14\}(P) \Rightarrow \]

\[ \{x \mapsto 7\}(P) \Rightarrow \]

\[ \sigma(b) \Rightarrow \text{True} \]
\[ \sigma(s) \Rightarrow \sigma_1 \]
\[ \sigma_1(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2 \]
\[ \sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2 \]
\[ \sigma(b) \Rightarrow \text{False} \]
\[ \sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma \]
Loop Example

\[ P \equiv \textbf{while } x < 50 \textbf{ do } x := 2 \ast x \textbf{ od} \]

\[
\begin{align*}
\{x \mapsto 28\} (x < 50) & \Rightarrow \text{True} \\
\{x \mapsto 14\} (x < 50) & \Rightarrow \text{True} \\
\{x \mapsto 7\} (x < 50) & \Rightarrow \text{True}
\end{align*}
\]

\[
\begin{align*}
\{x \mapsto 28\}(P) & \Rightarrow \\
\{x \mapsto 14\}(P) & \Rightarrow \\
\{x \mapsto 7\}(P) & \Rightarrow
\end{align*}
\]

\[
\begin{align*}
\sigma(b) & \Rightarrow \text{True} \\
\sigma(s) & \Rightarrow \sigma_1 \\
\sigma_1(\textbf{while } b \textbf{ do } s \textbf{ od}) & \Rightarrow \sigma_2
\end{align*}
\]

\[
\begin{align*}
\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) & \Rightarrow \sigma_2 \\
\sigma(b) & \Rightarrow \text{False} \\
\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) & \Rightarrow \sigma
\end{align*}
\]
Loop Example

\[ P \equiv \text{while } x < 50 \text{ do } x := 2 \times x \text{ od} \]

\[
\begin{align*}
\{x \mapsto 28\} (x < 50) &\Rightarrow \text{True} & \{x \mapsto 28\} (x := 2 \times x) &\Rightarrow \\
\{x \mapsto 14\} (x < 50) &\Rightarrow \text{True} & \{x \mapsto 14\} (x := 2 \times x) &\Rightarrow \{x \mapsto 28\} \\
\{x \mapsto 7\} (x < 50) &\Rightarrow \text{True} & \{x \mapsto 7\} (x := 2 \times x) &\Rightarrow \{x \mapsto 14\} \\
\{x \mapsto 7\} (P) &\Rightarrow & \{x \mapsto 14\} (P) &\Rightarrow \\
\{x \mapsto 7\} (P) &\Rightarrow & \{x \mapsto 14\} (P) &\Rightarrow
\end{align*}
\]

\[
\begin{align*}
\sigma(b) &\Rightarrow \text{True} & \sigma(s) &\Rightarrow \sigma_1 & \sigma_1(\text{while } b \text{ do } s \text{ od}) &\Rightarrow \sigma_2 & \sigma(b) &\Rightarrow \text{False} \\
\sigma(\text{while } b \text{ do } s \text{ od}) &\Rightarrow \sigma_2
\end{align*}
\]
Loop Example

\[ P \equiv \text{while } x < 50 \text{ do } x := 2 \times x \text{ od} \]

\[
\begin{align*}
\{x \mapsto 28\} \ (x < 50) & \Rightarrow \text{True} & \{x \mapsto 28\} \ (x := 2 \times x) & \Rightarrow \{x \mapsto 56\} \\
\{x \mapsto 14\} \ (x < 50) & \Rightarrow \text{True} & \{x \mapsto 14\} \ (x := 2 \times x) & \Rightarrow \{x \mapsto 28\} \\
\{x \mapsto 7\} \ (x < 50) & \Rightarrow \text{True} & \{x \mapsto 7\} \ (x := 2 \times x) & \Rightarrow \{x \mapsto 14\} \\
\{x \mapsto 7\}(P) & \Rightarrow & \{x \mapsto 14\}(P) & \Rightarrow & \{x \mapsto 7\}(P) & \Rightarrow
\end{align*}
\]

\[
\begin{align*}
\sigma(b) & \Rightarrow \text{True} & \sigma(s) & \Rightarrow \sigma_1 & \sigma_1(\text{while } b \text{ do } s \text{ od}) & \Rightarrow \sigma_2 \\
\sigma(\text{while } b \text{ do } s \text{ od}) & \Rightarrow \sigma_2 & \sigma(b) & \Rightarrow \text{False} & \sigma(\text{while } b \text{ do } s \text{ od}) & \Rightarrow \sigma
\end{align*}
\]
Loop Example

\[ P \equiv \text{while } x < 50 \text{ do } x := 2 \times x \text{ od} \]

\[
\begin{align*}
\{x \mapsto 28\} (x < 50) & \Rightarrow \text{True} & \{x \mapsto 28\} (x := 2 \times x) & \Rightarrow \{x \mapsto 56\} & \{x \mapsto 56\}(P) & \Rightarrow \\
\{x \mapsto 14\} (x < 50) & \Rightarrow \text{True} & \{x \mapsto 14\} (x := 2 \times x) & \Rightarrow \{x \mapsto 28\} & \{x \mapsto 28\}(P) & \Rightarrow \\
\{x \mapsto 7\} (x < 50) & \Rightarrow \text{True} & \{x \mapsto 7\} (x := 2 \times x) & \Rightarrow \{x \mapsto 14\} & \{x \mapsto 14\}(P) & \Rightarrow \\
\{x \mapsto 7\}(P) & \Rightarrow \\
\end{align*}
\]

\[
\begin{align*}
\sigma(b) & \Rightarrow \text{True} & \sigma(s) & \Rightarrow \sigma_1 & \sigma_1(\text{while } b \text{ do } s \text{ od}) & \Rightarrow \sigma_2 & \sigma(b) & \Rightarrow \text{False} & \sigma(\text{while } b \text{ do } s \text{ od}) & \Rightarrow \sigma \\
\sigma(\text{while } b \text{ do } s \text{ od}) & \Rightarrow \sigma_2
\end{align*}
\]
Loop Example

\[ P \equiv \textbf{while} \ x < 50 \ \textbf{do} \ x := 2 \times x \ \textbf{od} \]

\[
\begin{array}{c}
\{x \mapsto 28\} \ (x < 50) \Rightarrow \text{True} \\
\{x \mapsto 28\} \ (x := 2 \times x) \Rightarrow \{x \mapsto 56\} \\
\{x \mapsto 56\} \ (x < 50) \Rightarrow \text{False} \\
\{x \mapsto 28\} (P) \Rightarrow \\
\{x \mapsto 14\} \ (x < 50) \Rightarrow \text{True} \\
\{x \mapsto 14\} \ (x := 2 \times x) \Rightarrow \{x \mapsto 28\} \\
\{x \mapsto 28\} (P) \Rightarrow \\
\{x \mapsto 7\} \ (x < 50) \Rightarrow \text{True} \\
\{x \mapsto 7\} \ (x := 2 \times x) \Rightarrow \{x \mapsto 14\} \\
\{x \mapsto 14\} (P) \Rightarrow \\
\{x \mapsto 7\} (P) \Rightarrow \\
\end{array}
\]

\[
\begin{array}{c}
\sigma(b) \Rightarrow \text{True} \\
\sigma(s) \Rightarrow \sigma_1 \\
\sigma_1(\textbf{while} \ b \ \textbf{do} \ s \ \textbf{od}) \Rightarrow \sigma_2 \\
\sigma(\textbf{while} \ b \ \textbf{do} \ s \ \textbf{od}) \Rightarrow \sigma_2 \\
\sigma(b) \Rightarrow \text{False} \\
\sigma(\textbf{while} \ b \ \textbf{do} \ s \ \textbf{od}) \Rightarrow \sigma \\
\end{array}
\]
Loop Example

\[ P \equiv \textbf{while } x < 50 \textbf{ do } x := 2 \times x \textbf{ od } \]

\[
\begin{align*}
\{x \mapsto 28\} (x < 50) & \Rightarrow \text{True} & \{x \mapsto 28\} (x := 2 \times x) & \Rightarrow \{x \mapsto 56\} & \{x \mapsto 56\} (x < 50) & \Rightarrow \text{False} \\
\{x \mapsto 14\} (x < 50) & \Rightarrow \text{True} & \{x \mapsto 14\} (x := 2 \times x) & \Rightarrow \{x \mapsto 28\} & \{x \mapsto 28\}(P) & \Rightarrow \\
\{x \mapsto 7\} (x < 50) & \Rightarrow \text{True} & \{x \mapsto 7\} (x := 2 \times x) & \Rightarrow \{x \mapsto 14\} & \{x \mapsto 14\}(P) & \Rightarrow \\
\{x \mapsto 7\}(P) & \Rightarrow \\
\end{align*}
\]

\[
\begin{align*}
\sigma(b) & \Rightarrow \text{True} & \sigma(s) & \Rightarrow \sigma_1 & \sigma_1(\textbf{while } b \textbf{ do } s \textbf{ od}) & \Rightarrow \sigma_2 & \sigma(b) & \Rightarrow \text{False} & \sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) & \Rightarrow \sigma \\
\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) & \Rightarrow \sigma_2 \\
\end{align*}
\]
Loop Example

\[ P \equiv \text{while } x < 50 \text{ do } x := 2 \times x \text{ od} \]

\[
\begin{array}{ll}
\{x \mapsto 28\} (x < 50) \Rightarrow \text{True} & \{x \mapsto 28\} (x := 2 \times x) \Rightarrow \{x \mapsto 56\} \\
\{x \mapsto 14\} (x < 50) \Rightarrow \text{True} & \{x \mapsto 14\} (x := 2 \times x) \Rightarrow \{x \mapsto 28\} \\
\{x \mapsto 7\} (x < 50) \Rightarrow \text{True} & \{x \mapsto 7\} (x := 2 \times x) \Rightarrow \{x \mapsto 14\} \\
\{x \mapsto 7\} (P) \Rightarrow & \{x \mapsto 14\} (P) \Rightarrow \\
\{x \mapsto 56\} (x < 50) \Rightarrow \text{False} & \{x \mapsto 56\} (P) \Rightarrow \{x \mapsto 56\} \\
\end{array}
\]
Loop Example

\[ P \equiv \textbf{while } x < 50 \textbf{ do } x \leftarrow 2 \times x \textbf{ od} \]

\[
\begin{array}{ll}
\{x \leftarrow 28\} (x < 50) & \Rightarrow \text{True} \\
\{x \leftarrow 28\} (x := 2 \times x) & \Rightarrow \{x \leftarrow 56\} \\
\{x \leftarrow 28\}(P) & \Rightarrow \{x \leftarrow 56\} \\
\{x \leftarrow 14\} (x < 50) & \Rightarrow \text{True} \\
\{x \leftarrow 14\} (x := 2 \times x) & \Rightarrow \{x \leftarrow 28\} \\
\{x \leftarrow 14\}(P) & \Rightarrow \{x \leftarrow 56\} \\
\{x \leftarrow 7\} (x < 50) & \Rightarrow \text{True} \\
\{x \leftarrow 7\} (x := 2 \times x) & \Rightarrow \{x \leftarrow 14\} \\
\{x \leftarrow 7\}(P) & \Rightarrow \text{} \\
\end{array}
\]

\[
\sigma(b) \Rightarrow \text{True} \quad \sigma(s) \Rightarrow \sigma_1 \quad \sigma_1(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2 \quad \sigma(b) \Rightarrow \text{False} \quad \sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma
\]
Loop Example

\[ P \equiv \textbf{while } x < 50 \textbf{ do } x := 2 \times x \textbf{ od} \]

\[
\begin{array}{c}
\{x \mapsto 28\} (x < 50) \Rightarrow \text{True} \quad \{x \mapsto 28\} (x := 2 \times x) \Rightarrow \{x \mapsto 56\} \\
\{x \mapsto 28\}(P) \Rightarrow \{x \mapsto 56\}
\end{array}
\]

\[
\begin{array}{c}
\{x \mapsto 14\} (x < 50) \Rightarrow \text{True} \quad \{x \mapsto 14\} (x := 2 \times x) \Rightarrow \{x \mapsto 28\} \\
\{x \mapsto 14\}(P) \Rightarrow \{x \mapsto 56\}
\end{array}
\]

\[
\begin{array}{c}
\{x \mapsto 7\} (x < 50) \Rightarrow \text{True} \quad \{x \mapsto 7\} (x := 2 \times x) \Rightarrow \{x \mapsto 14\} \\
\{x \mapsto 7\}(P) \Rightarrow \{x \mapsto 56\}
\end{array}
\]

\[
\begin{array}{c}
\sigma(b) \Rightarrow \text{True} \quad \sigma(s) \Rightarrow \sigma_1 \quad \sigma_1(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2 \\
\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2 \\
\sigma(b) \Rightarrow \text{False} \quad \sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma
\end{array}
\]
Sequencing, Conditionals, Loops

\[
\begin{align*}
\sigma_1(s_1) & \Rightarrow \sigma_2 & \sigma_2(s_2) & \Rightarrow \sigma_3 \\
\sigma_1(s_1; s_2) & \Rightarrow \sigma_3 \\
\end{align*}
\]

\[
\begin{align*}
\sigma(b) & \Rightarrow \text{True} & \sigma(s_1) & \Rightarrow \sigma_1 \\
\sigma(\text{if } b \text{ then } s_1 \text{ else } s_2 \text{ fi}) & \Rightarrow \sigma_1 \\
\sigma(b) & \Rightarrow \text{False} & \sigma(s_2) & \Rightarrow \sigma_2 \\
\sigma(\text{if } b \text{ then } s_1 \text{ else } s_2 \text{ fi}) & \Rightarrow \sigma_2 \\
\end{align*}
\]

\[
\begin{align*}
\sigma(b) & \Rightarrow \text{True} & \sigma(s) & \Rightarrow \sigma_1 & \sigma_1(\text{while } b \text{ do } s \text{ od}) & \Rightarrow \sigma_2 \\
\sigma(\text{while } b \text{ do } s \text{ od}) & \Rightarrow \sigma_2 \\
\sigma(b) & \Rightarrow \text{False} & \sigma(\text{while } b \text{ do } s \text{ od}) & \Rightarrow \sigma \\
\end{align*}
\]
Additional Control Structures

- do { ... } while ( ... )
- repeat { ... } until ( ... )
- for (... , ... , ...) { ... }
- for $i = beg$ to $end$ do { ... }
Additional Control Structures

- do { ... } while ( ... )
- repeat { ... } until ( ... )
- for (... , ..., ...) { ... }
- for $i = \text{beg to end}$ do { ... }

Options:
- **Direct definition** using new operational semantics rules
Additional Control Structures

- `do { ... } while ( ... )`
- `repeat { ... } until ( ... )`
- `for (..., ..., ...) { ... }
- `for i = beg to end do { ... }

Options:
- **Direct definition** using new operational semantics rules

\[
\begin{align*}
\sigma_1(s) &\Rightarrow \sigma_2 \quad \sigma_2(b) \Rightarrow \text{False} \\
\sigma_1( \text{do } s \text{ while } (b)) &\Rightarrow \sigma_2
\end{align*}
\]

\[
\begin{align*}
\sigma_1(s) &\Rightarrow \sigma_2 \quad \sigma_2(b) \Rightarrow \text{True} \\
\sigma_2( \text{do } s \text{ while } (b)) &\Rightarrow \sigma_3
\end{align*}
\]
Additional Control Structures

- `do { ... } while ( ... )`
- `repeat { ... } until ( ... )`
- `for (..., ..., ...) { ... }`
- `for i = beg to end do { ... }

Options:

- **Direct definition** using new operational semantics rules

  \[
  \sigma_1(s) \Rightarrow \sigma_2 \quad \sigma_2(b) \Rightarrow \text{False} \quad \sigma_1(s) \Rightarrow \sigma_2 \quad \sigma_2(b) \Rightarrow \text{True} \quad \sigma_2(\text{do s while (b)}) \Rightarrow \sigma_3
  \]

  \[
  \sigma_1(\text{do s while (b)}) \Rightarrow \sigma_2 \\
  \sigma_1(\text{do s while (b)}) \Rightarrow \sigma_3
  \]

- **Translation into core language** — *derived* features
Additional Control Structures

- `do { ... } while ( ... )`
- `repeat { ... } until ( ... )`
- `for (..., ..., ...) { ... }`
- `for i = beg to end do { ... }`

Options:

- **Direct definition** using new operational semantics rules
  \[
  \begin{align*}
  \sigma_1(s) &\Rightarrow \sigma_2 \quad \sigma_2(b) \Rightarrow \text{False} \\
  \sigma_1(\ \text{do } s \ \text{while } (b)) &\Rightarrow \sigma_2 \\
  \sigma_1(s) &\Rightarrow \sigma_2 \quad \sigma_2(b) \Rightarrow \text{True} \\
  \sigma_2(\ \text{do } s \ \text{while } (b)) &\Rightarrow \sigma_3 \\
  \end{align*}
  \]

- **Translation into core language** — *derived* features
  \[
  \begin{align*}
  \sigma_1(s \ ; \ \text{while } b \ \text{do } s \ \text{od}) &\Rightarrow \sigma_2 \\
  \sigma_1(\ \text{do } s \ \text{while } (c)) &\Rightarrow \sigma_2 \\
  \end{align*}
  \]
Additional Language Features

- Output: `print (e)`
- Input: `read (e)`
- Nested Scopes (declarations in inner blocks)
- Function and procedure calls
- Side-effecting expressions
Additional Language Features

- Output: \texttt{print} \((e)\)
- Input: \texttt{read} \((e)\)
- Nested Scopes (declarations in inner blocks)
- Function and procedure calls
- Side-effecting expressions

Main tasks:
AdditionaL Language Features

- Output: `print (e)`
- Input: `read (e)`
- Nested Scopes (declarations in inner blocks)
- Function and procedure calls
- Side-effecting expressions

Main tasks:
- Define an appropriate state space
Additional Language Features

- Output: `print (e)`
- Input: `read (e)`
- Nested Scopes (declarations in inner blocks)
- Function and procedure calls
- Side-effecting expressions

Main tasks:
- Define an appropriate state space
- Adapt assertion schemas if necessary
Additional Language Features

- Output: `print (e)`
- Input: `read (e)`
- Nested Scopes (declarations in inner blocks)
- Function and procedure calls
- Side-effecting expressions

**Main tasks:**

- Define an appropriate state space
- Adapt assertion schemas if necessary
  
  e.g., expression evaluation with side-effects: \( \sigma(e) \Rightarrow (\sigma', \nu) \)
Additional Language Features

- Output: \texttt{print (e)}
- Input: \texttt{read (e)}
- Nested Scopes (declarations in inner blocks)
- Function and procedure calls
- Side-effecting expressions

Main tasks:

- Define an appropriate state space
- Adapt assertion schemas if necessary
e.g., expression evaluation with side-effects: $\sigma(e) \Rightarrow (\sigma', \nu)$
- “Port” all existing feature definitions to the new states
Additional Language Features

- Output: `print (e)`
- Input: `read (e)`
- Nested Scopes (declarations in inner blocks)
- Function and procedure calls
- Side-effecting expressions

**Main tasks:**

- Define an appropriate state space
- Adapt assertion schemas if necessary
  
  e.g., expression evaluation with side-effects: \( \sigma(e) \Rightarrow (\sigma', \nu) \)
- “Port” all existing feature definitions to the new states
- Appropriately define the new features
Additional Language Features

- Output: \texttt{print} (e)
- Input: \texttt{read} (e)
- Nested Scopes (declarations in inner blocks)
- Function and procedure calls
- Side-effecting expressions

**Main tasks:**

- Define an appropriate state space
- Adapt assertion schemas if necessary
  
  e.g., expression evaluation with side-effects: \( \sigma(e) \Rightarrow (\sigma', \nu) \)
- “Port” all existing feature definitions to the new states
- Appropriately define the new features
- Prove “conservative extension”
Additional Language Features

- Output: \texttt{print} \((e)\)
- Input: \texttt{read} \((e)\)
- Nested Scopes (declarations in inner blocks)
- Function and procedure calls
- Side-effecting expressions

Main tasks:

- Define an appropriate state space
- Adapt assertion schemas if necessary
  
e.g., expression evaluation with side-effects: \(\sigma(e) \Rightarrow (\sigma', \nu)\)
- “Port” all existing feature definitions to the new states
- Appropriately define the new features
- Prove “conservative extension”: mapping from old states to new is injective and preserves transitions.
New Language Feature Example: Output

Assume a new statement “\texttt{print (e)}”
New Language Feature Example: Output

Assume a new statement “print \((e)\)” that prints the \texttt{integer} expression \(e\) to the screen.
New Language Feature Example: Output

Assume a new statement “print \((e)\)” that prints the \texttt{integer} expression \(e\) to the screen.

- New typing rule: the argument of \texttt{PRINT} has to be of type integer.
New Language Feature Example: Output

Assume a new statement “\texttt{print (e)}” that prints the \texttt{integer} expression \(e\) to the screen.

- New typing rule: the argument of \texttt{PRINT} has to be of type integer.
- New abstract syntax constructor: \texttt{Print :: Expression \rightarrow Statement}
New Language Feature Example: Output

Assume a new statement “\texttt{print (e)}” that prints the \texttt{integer} expression \( e \) to the screen.

- New typing rule: the argument of \texttt{PRINT} has to be of type \texttt{integer}.
- New abstract syntax constructor: \( \texttt{Print :: Expression \rightarrow Statement} \)
- New state space: \( \texttt{State}_2 = \texttt{State}_1 \times [\mathbb{Z}] \)
New Language Feature Example: Output

Assume a new statement “\texttt{print}(e)” that prints the \texttt{integer} expression $e$ to the screen.

- New typing rule: the argument of $PRINT$ has to be of type integer.
- New abstract syntax constructor: \texttt{Print} :: \texttt{Expression} $\rightarrow$ \texttt{Statement}
- New state space: $State_2 = State_1 \times \mathbb{Z}$
- New statement assertion schema: $(\sigma_1, out_1)(s) \Rightarrow (\sigma_2, out_2)$
New Language Feature Example: Output

Assume a new statement “print (e)” that prints the integer expression e to the screen.

- New typing rule: the argument of PRINT has to be of type integer.
- New abstract syntax constructor: Print :: Expression → Statement
- New state space: $State_2 = State_1 \times [\mathbb{Z}]$
- New statement assertion schema: $(\sigma_1, out_1)(s) \Rightarrow (\sigma_2, out_2)$
- Adapted rules
New Language Feature Example: Output

Assume a new statement “\texttt{print}(e)\texttt{)” that prints the \texttt{integer} expression \(e\) to the screen.

- New typing rule: the argument of \texttt{PRINT} has to be of type integer.
- New abstract syntax constructor: \texttt{Print :: Expression → Statement}
- New state space: \(\text{State}_2 = \text{State}_1 \times [\mathbb{Z}]\)
- New statement assertion schema: \((\sigma_1, out_1)(s) \Rightarrow (\sigma_2, out_2)\)
- Adapted rules, e.g.:
  \[
  \frac{\sigma(e) \Rightarrow v}{(\sigma, out)(x := e) \Rightarrow (\sigma \oplus \{x \mapsto v\}_1, out)}
  \]
New Language Feature Example: Output

Assume a new statement “\texttt{print (e)}” that prints the \texttt{integer} expression \texttt{e} to the screen.

- New typing rule: the argument of \texttt{PRINT} has to be of type integer.
- New abstract syntax constructor: \texttt{Print :: Expression \rightarrow Statement}
- New state space: \( \texttt{State}_2 = \texttt{State}_1 \times [\mathbb{Z}] \)
- New statement assertion schema: \((\sigma_1, \texttt{out}_1)(s) \Rightarrow (\sigma_2, \texttt{out}_2)\)
- Adapted rules, e.g.:
  \[
  \frac{\sigma(e) \Rightarrow v}{(\sigma, \texttt{out})(x := e) \Rightarrow (\sigma \oplus \{x \mapsto v\}, \texttt{out})}
  \]
- Rules for new feature:
New Language Feature Example: Output

Assume a new statement “print \( e \)” that prints the \texttt{integer} expression \( e \) to the screen.

- New typing rule: the argument of \texttt{PRINT} has to be of type integer.
- New abstract syntax constructor: \texttt{Print :: Expression \rightarrow Statement}
- New state space: \( State_2 = State_1 \times [\mathbb{Z}] \)
- New statement assertion schema: \((\sigma_1, out_1)(s) \implies (\sigma_2, out_2)\)

\[
\begin{aligned}
\sigma(e) & \implies v \\
(\sigma, out)(x := e) & \implies (\sigma \oplus \{x \mapsto v\}, out)
\end{aligned}
\]

\[
\begin{aligned}
\sigma(e) & \implies i \\
(\sigma, out)(\texttt{print } (e)) & \implies (\sigma, out \downarrow [i])
\end{aligned}
\]
New Language Feature Example: Output

Assume a new statement “**print** (e)” that prints the **integer** expression e to the screen.

- New typing rule: the argument of **PRINT** has to be of type integer.
- New abstract syntax constructor: **Print :: Expression → Statement**
- New state space: \( State_2 = State_1 \times [\mathbb{Z}] \)
- New statement assertion schema: \( (\sigma_1, out_1)(s) \Rightarrow (\sigma_2, out_2) \)
- Adapted rules, e.g.:
  \[
  \sigma(e) \Rightarrow v \\
  (\sigma, out)(x := e) \Rightarrow (\sigma \oplus \{x \mapsto v\}, out)
  \]
- Rules for new feature:
  \[
  \sigma(e) \Rightarrow i \\
  (\sigma, out)(\textbf{print} (e)) \Rightarrow (\sigma, out + [i])
  \]
- Check determinism, add to interpreter.
Exceptions
Exceptions

• An exception is an event that needs to be handled in a special way
Exceptions

- An **exception** is an event that needs to be handled in a special way
- Examples: system errors, program errors, user errors, undefined operations
Exceptions

• An exception is an event that needs to be handled in a special way
• Examples: system errors, program errors, user errors, undefined operations
• Most importantly: events that cannot be conveniently handled where generated
Exceptions

- An exception is an event that needs to be handled in a special way
- Examples: system errors, program errors, user errors, undefined operations
- Most importantly: events that cannot be conveniently handled where generated

Exception Handling
Exceptions

- An exception is an event that needs to be handled in a special way
- Examples: system errors, program errors, user errors, undefined operations
- Most importantly: events that cannot be conveniently handled where generated

Exception Handling:

- A generated exception is thrown to a higher part of the code
Exceptions

- An exception is an event that needs to be handled in a special way
- Examples: system errors, program errors, user errors, undefined operations
- Most importantly: events that cannot be conveniently handled where generated

Exception Handling:

- A generated exception is thrown to a higher part of the code
- A thrown exception is caught by an appropriate exception handler that processes the exception
Exceptions

• An exception is an event that needs to be handled in a special way
• Examples: system errors, program errors, user errors, undefined operations
• Most importantly: events that cannot be conveniently handled where generated

Exception Handling:

• A generated exception is thrown to a higher part of the code
• A thrown exception is caught by an appropriate exception handler that processes the exception
• Benefits of an exception handling mechanism:
  – Exception-handling code can be separated from regular code
Exceptions

• An exception is an event that needs to be handled in a special way
• Examples: system errors, program errors, user errors, undefined operations
• Most importantly: events that cannot be conveniently handled where generated

Exception Handling:

• A generated exception is thrown to a higher part of the code
• A thrown exception is caught by an appropriate exception handler that processes the exception
• Benefits of an exception handling mechanism:
  – Exception-handling code can be separated from regular code
  – Exceptions can be handled at the most appropriate place in the code, not necessarily where they are generated
Exceptions in Java
Exceptions in Java

In Java, exceptions are represented by objects in the `java.lang.Throwable` class.
Exceptions in Java

In Java, exceptions are represented by objects in the `java.lang.Throwable` class.

`Throwable` has two subclasses:
Exceptions in Java

In Java, exceptions are represented by objects in the `java.lang.Throwable` class.

`Throwable` has two subclasses:

- `Exception`: Exceptions which can be thrown and caught
Exceptions in Java

In Java, exceptions are represented by objects in the `java.lang.Throwable` class.

`Throwable` has two subclasses:

- **Exception**: Exceptions which can be thrown and caught
- **Error**: **Nonrecoverable** errors thrown by the system
Exceptions in Java

In Java, exceptions are represented by objects in the `java.lang Throwable` class.

`Throwable` has two subclasses:

- **Exception**: Exceptions which can be thrown and caught
- **Error**: **Nonrecoverable** errors thrown by the system

Two kinds of Exceptions:
Exceptions in Java

In Java, exceptions are represented by objects in the java.lang.Throwable class.

Throwable has two subclasses:

- **Exception**: Exceptions which can be thrown and caught
- **Error**: Nonrecoverable errors thrown by the system

Two kinds of Exceptions:

- **Checked Exceptions**, which must be declared with a `throws` clause in a method declaration
Exceptions in Java

In Java, exceptions are represented by objects in the `java.lang.Throwable` class.

`Throwable` has two subclasses:

- **Exception**: Exceptions which can be thrown and caught
- **Error**: **Nonrecoverable** errors thrown by the system

Two kinds of Exceptions:

- **Checked Exceptions** which must be declared with a `throws` clause in a method declaration:

  All user-defined exceptions, `IOExc.`, `ClassNotFoundExc.`, …
Exceptions in Java

In Java, exceptions are represented by objects in the `java.lang.Throwable` class.

`Throwable` has two subclasses:

- **Exception**: Exceptions which can be thrown and caught
- **Error**: Nonrecoverable errors thrown by the system

Two kinds of Exceptions:

- **Checked Exceptions** which must be declared with a `throws` clause in a method declaration:
  
  All user-defined exceptions, `IOException`, `ClassNotFoundException`, …

- **RuntimeException**: Abnormal runtime events which need not be declared with a `throws` clause:
  
  `ArithmeticException`, `ClassCastException`, `IllegalArgumentException`, `IndexOutOfBoundsException`, `NullPointerException`, …
Exception Handling in Java
Exception Handling in Java

- Any Java code can construct an exception and then throw it
Exception Handling in Java

- Any Java code can construct an exception and then throw it
- Exceptions are caught and handled with a try–catch–finally statement
Exception Handling in Java

• Any Java code can construct an exception and then throw it

• Exceptions are caught and handled with a try–catch–finally statement
  – Raising an exception terminates the current block
Exception Handling in Java

• Any Java code can construct an exception and then throw it

• Exceptions are caught and handled with a try–catch–finally statement
  – Raising an exception terminates the current block
  – Exceptions propagate up through the code until they are caught by a catch substatement
Exception Handling in Java

- Any Java code can construct an exception and then throw it.
- Exceptions are caught and handled with a `try–catch–finally` statement:
  - Raising an exception terminates the current block.
  - Exceptions propagate up through the code until they are caught by a `catch` substatement.
- Every **checked exception** that can be thrown in a method must be either caught in the method or declared in the method with a `throws` clause.
Exception Handling in Java

- Any Java code can construct an exception and then throw it.
- Exceptions are caught and handled with a try–catch–finally statement.
  - Raising an exception terminates the current block.
  - Exceptions propagate up through the code until they are caught by a catch substatement.
- Every checked exception that can be thrown in a method must be either caught in the method or declared in the method with a throws clause.
- Benefits of Java’s exception handling mechanism:
Exception Handling in Java

• Any Java code can construct an exception and then throw it

• Exceptions are caught and handled with a try–catch–finally statement
  – Raising an exception terminates the current block
  – Exceptions propagate up through the code until they are caught by a catch substatement

• Every checked exception that can be thrown in a method must be either caught in the method or declared in the method with a throws clause

• Benefits of Java’s exception handling mechanism:
  – A class of exceptions can be subclassed
Exception Handling in Java

- Any Java code can construct an exception and then throw it

- Exceptions are caught and handled with a try–catch–finally statement
  - Raising an exception terminates the current block
  - Exceptions propagate up through the code until they are caught by a catch substatement

- Every *checked exception* that can be thrown in a method must be either caught in the method or declared in the method with a throws clause

- Benefits of Java’s exception handling mechanism:
  - A class of exceptions can be subclassed
  - There is an *enforced discipline* for checked exceptions
Try-Catch-Finally Statement

try {
    try body
}
catch (Exception_1 var_1) {
    catch_1 body
}
catch (Exception_2 var_2) {
    catch_2 body
}
...
catch (Exception_n var_n) {
    catch_n body
}
finally {
    finally body
}
Try-Catch-Finally Example

import java.io.*;
class Read1 {
    public static void main(String[] args) {
        BufferedReader in =
            new BufferedReader(new InputStreamReader(System.in));
        try {
            System.out.println("How old are you? ");
            String inputLine = in.readLine();
            int age = Integer.parseInt(inputLine);
            age++;
            System.out.println("Next year, you’ll be " + age);
        } catch (IOException exception) {
            System.out.println("Input/output error " + exception);
        } catch (NumberFormatException exception) {
            System.out.println("Input was not a number ");
        } finally {
            if (in != null) {
                try {
                    in.close();
                } catch (IOException exception) {
                }
            }
        }
    }
}
Ways of Handling Exceptions
Ways of Handling Exceptions

- Capture it and execute some code to deal with it
Ways of Handling Exceptions

• Capture it and execute some code to deal with it

• Capture it, execute some code, and then rethrow the exception
Ways of Handling Exceptions

- Capture it and execute some code to deal with it
- Capture it, execute some code, and then rethrow the exception
- Capture it, execute some code, and then throw a new exception
Ways of Handling Exceptions

- Capture it and execute some code to deal with it
- Capture it, execute some code, and then rethrow the exception
- Capture it, execute some code, and then throw a new exception
- Capture it and execute no code (ignore the exception)
Ways of Handling Exceptions

- Capture it and execute some code **to deal with it**
- Capture it, execute some code, and then **rethrow** the exception
- Capture it, execute some code, and then **throw a new exception**
- Capture it and execute no code (ignore the exception) — **bad idea!**
Ways of Handling Exceptions

- Capture it and execute some code to deal with it
- Capture it, execute some code, and then rethrow the exception
- Capture it, execute some code, and then throw a new exception
- Capture it and execute no code (ignore the exception) — bad idea!
- Do not capture it (let it propagate up)
Ways of Handling Exceptions

- Capture it and execute some code to deal with it
- Capture it, execute some code, and then rethrow the exception
- Capture it, execute some code, and then throw a new exception
- Capture it and execute no code (ignore the exception) — bad idea!
- Do not capture it (let it propagate up) — may need to declare!
Exceptions — Example

class Simulate4 {
    private static void println(String s)
        {System.out.println(s);}
    public static int _q = 0;
    public static void main(String[] a)
        { int s = g(2);
            println("* "+s+" "+_q);
        }
    public static int f(int k, int m) {
        println("f("+k+","+m+")");
        _q += m;
        int r = g(k) +_q;
        println("f("+k+","+m+")="+r);
        return r;
    }
}

public static int g(int n) {
    println("g(" + n + ")");
    int t = 3 * n;
    if ( t < 10 ) {
        try { t = (f (n + 1, _q));
            }
        catch (Exception e) {
            println("g: caught exception!");
            _q += n;
        }
        t = t / _q;
        println("g( " + n + " )= " + t);
        return t;
    }
}
Operational Semantics of Exceptions
Operational Semantics of Exceptions

Originally: Two kinds of assertions:
\[ \sigma(e) \Rightarrow v \] — evaluating expression \( e \) starting in state \( \sigma \) can produce value \( v \)
\[ \sigma_1(s) \Rightarrow \sigma_2 \] — execution of statement \( s \) starting in state \( \sigma_1 \) can successfully terminate in state \( \sigma_2 \)
Operational Semantics of Exceptions

**Originally:** Two kinds of assertions:

\[ \sigma(e) \Rightarrow v \] — evaluating expression \( e \) starting in state \( \sigma \) can produce value \( v \)

\[ \sigma_1(s) \Rightarrow \sigma_2 \] — execution of statement \( s \) starting in state \( \sigma_1 \) can successfully terminate in state \( \sigma_2 \)

**Now an additional possibility:**

\[ \sigma_1(s) \xrightarrow{1} (\sigma_2, x) \] — execution of statement \( s \) starting in state \( \sigma_1 \) can terminate in state \( \sigma_2 \) raising exception \( x \)
Operational Semantics of Exceptions

Originally: Two kinds of assertions:
- $\sigma(e) \Rightarrow v$ — evaluating expression $e$ starting in state $\sigma$ can produce value $v$
- $\sigma_1(s) \Rightarrow \sigma_2$ — execution of statement $s$ starting in state $\sigma_1$ can successfully terminate in state $\sigma_2$

Now an additional possibility:
- $\sigma_1(s) \not\Rightarrow (\sigma_2, x)$ — execution of statement $s$ starting in state $\sigma_1$ can terminate in state $\sigma_2$ raising exception $x$

Two additional sequencing rules:
- $\sigma_1(s_1) \not\Rightarrow (\sigma_2, z)$
- $\sigma_1(s_1; s_2) \not\Rightarrow (\sigma_2, z)$
Operational Semantics of Exceptions

Originally: Two kinds of assertions:

\( \sigma(e) \Rightarrow v \) — evaluating expression \( e \) starting in state \( \sigma \) can produce value \( v \)

\( \sigma_1(s) \Rightarrow \sigma_2 \) — execution of statement \( s \) starting in state \( \sigma_1 \) can successfully terminate in state \( \sigma_2 \)

Now an additional possibility:

\( \sigma_1(s) \Rightarrow (\sigma_2, x) \) — execution of statement \( s \) starting in state \( \sigma_1 \) can terminate in state \( \sigma_2 \) raising exception \( x \)

Two additional sequencing rules:

\[
\begin{align*}
\sigma_1(s_1) \Rightarrow (\sigma_2, z) \\
\sigma_1(s_1; s_2) \Rightarrow (\sigma_2, z)
\end{align*}
\]

\[
\begin{align*}
\sigma_1(s_1) \Rightarrow \sigma_2 \\
\sigma_2(s_2) \Rightarrow (\sigma_3, z) \\
\sigma_1(s_1; s_2) \Rightarrow (\sigma_3, z)
\end{align*}
\]
Operational Semantics of Exceptions

Originally: Two kinds of assertions:
\[ \sigma(e) \Rightarrow v \] — evaluating expression \( e \) starting in state \( \sigma \) can produce value \( v \)
\[ \sigma_1(s) \Rightarrow \sigma_2 \] — execution of statement \( s \) starting in state \( \sigma_1 \) can successfully terminate in state \( \sigma_2 \)

Now an additional possibility:
\[ \sigma_1(s) \overset{!}{\Rightarrow} (\sigma_2, x) \] — execution of statement \( s \) starting in state \( \sigma_1 \) can terminate in state \( \sigma_2 \) raising exception \( x \)

Two additional sequencing rules:
\[
\begin{align*}
\sigma_1(s_1) & \overset{!}{\Rightarrow} (\sigma_2, z) \\
\sigma_1(s_1; s_2) & \overset{!}{\Rightarrow} (\sigma_2, z)
\end{align*}
\]
\[
\begin{align*}
\sigma_1(s_1) & \Rightarrow \sigma_2 \\
\sigma_2(s_2) & \overset{!}{\Rightarrow} (\sigma_3, z)
\end{align*}
\]

Two additional if rules (no exceptions in expression evaluation yet):
\[
\begin{align*}
\sigma(b) & \Rightarrow \text{True} \\
\sigma(s_1) & \overset{!}{\Rightarrow} (\sigma_1, x) \\
\sigma(\text{if } b \text{ then } s_1 \text{ else } s_2 \text{ fi}) & \overset{!}{\Rightarrow} (\sigma_1, x)
\end{align*}
\]
\[
\begin{align*}
\sigma(b) & \Rightarrow \text{False} \\
\sigma(s_2) & \overset{!}{\Rightarrow} (\sigma_2, x) \\
\sigma(\text{if } b \text{ then } s_1 \text{ else } s_2 \text{ fi}) & \overset{!}{\Rightarrow} (\sigma_2, x)
\end{align*}
\]
Exceptions — Interpreter

Original statement interpretation:

\[\texttt{interpStmt} :: \textit{Statement} \rightarrow \textit{State1} \rightarrow \textit{Maybe State1}\]
Exceptions — Interpreter

Original statement interpretation:

\[ \text{interpStmt} :: \text{Statement} \rightarrow \text{State1} \rightarrow \text{Maybe State1} \]

meaning:

\[ \sigma_1(s) \Rightarrow \sigma_2 \quad \text{iff} \quad \text{interpStmt}\ s\ \sigma_1 = \text{Just } \sigma_2 \]

\[ \neg \exists \sigma_2 \bullet \sigma_1(s) \Rightarrow \sigma_2 \quad \text{iff} \quad \text{interpStmt}\ s\ \sigma_1 \in \{\bot, \text{Nothing}\} \]
Exceptions — Interpreter

Original statement interpretation:

\[ \text{interpStmt} :: \text{Statement} \rightarrow \text{State1} \rightarrow \text{Maybe State1} \]

meaning:

\[ \sigma_1(s) \Rightarrow \sigma_2 \quad \text{iff} \quad \text{interpStmt} s \sigma_1 = \text{Just } \sigma_2 \]

\[ \neg \exists \sigma_2 \cdot \sigma_1(s) \Rightarrow \sigma_2 \quad \text{iff} \quad \text{interpStmt} s \sigma_1 \in \{\bot, \text{Nothing}\} \]

Statement interpretation with exceptions:

\[ \text{interpStmtExc} :: \text{Statement} \rightarrow \text{State1} \rightarrow \text{Maybe (Either State1 (State1, Exc))} \]

meaning:

\[ \sigma_1(s) \Rightarrow \sigma_2 \quad \text{iff} \]

\[ \sigma_1(s) \Rightarrow (\sigma_2, x) \quad \text{iff} \]
Exceptions — Interpreter

Original statement interpretation:

\[ \text{interpStmt} ::\ Statement \rightarrow \text{State1} \rightarrow \text{Maybe State1} \]

meaning:

\[ \sigma_1(s) \Rightarrow \sigma_2 \quad \text{iff} \quad \text{interpStmt } s \sigma_1 = \text{Just } \sigma_2 \]
\[ \neg \exists \sigma_2 \cdot \sigma_1(s) \Rightarrow \sigma_2 \quad \text{iff} \quad \text{interpStmt } s \sigma_1 \in \{\bot, \text{Nothing}\} \]

Statement interpretation with exceptions:

\[ \text{interpStmtExc} ::\ Statement \rightarrow \text{State1} \rightarrow \text{Maybe (Either State1 (State1, Exc))} \]

meaning:

\[ \sigma_1(s) \Rightarrow \sigma_2 \quad \text{iff} \quad \text{interpStmt } s \sigma_1 = \text{Just (Left } \sigma_2) \]
\[ \sigma_1(s) \Rightarrow (\sigma_2, x) \quad \text{iff} \]
Exceptions — Interpreter

Original statement interpretation:

\[ \text{interpStmt} :: \text{Statement} \rightarrow \text{State1} \rightarrow \text{Maybe} \text{ State1} \]

meaning:

\[ \sigma_1(s) \Rightarrow \sigma_2 \iff \text{interpStmt} s \sigma_1 = \text{Just} \sigma_2 \]
\[ \neg \exists \sigma_2 \bullet \sigma_1(s) \Rightarrow \sigma_2 \iff \text{interpStmt} s \sigma_1 \in \{ \bot, \text{Nothing} \} \]

Statement interpretation with exceptions:

\[ \text{interpStmtExc} :: \text{Statement} \rightarrow \text{State1} \rightarrow \text{Maybe} (\text{Either} \text{ State1} (\text{State1}, \text{Exc})) \]

meaning:

\[ \sigma_1(s) \Rightarrow \sigma_2 \iff \text{interpStmt} s \sigma_1 = \text{Just} (\text{Left} \sigma_2) \]
\[ \sigma_1(s) \Rightarrow (\sigma_2, x) \iff \text{interpStmt} s \sigma_1 = \text{Just} (\text{Right} (\sigma_2, x)) \]
Exceptions — Interpreter

Original statement interpretation:

\[ interpStmt :: \text{Statement} \rightarrow \text{State1} \rightarrow \text{Maybe State1} \]

meaning:

\[
\sigma_1(s) \Rightarrow \sigma_2 \quad \text{iff} \quad \text{interpStmt } s \sigma_1 = \text{Just } \sigma_2 \\
\neg \exists \sigma_2 \cdot \sigma_1(s) \Rightarrow \sigma_2 \quad \text{iff} \quad \text{interpStmt } s \sigma_1 \in \{\bot, \text{Nothing}\}
\]

Statement interpretation with exceptions:

\[ interpStmtExc :: \text{Statement} \rightarrow \text{State1} \rightarrow \text{Maybe (Either State1 (State1, Exc))} \]

meaning:

\[
\sigma_1(s) \Rightarrow \sigma_2 \quad \text{iff} \quad \text{interpStmt } s \sigma_1 = \text{Just (Left } \sigma_2) \\
\sigma_1(s) \Rightarrow (\sigma_2, x) \quad \text{iff} \quad \text{interpStmt } s \sigma_1 = \text{Just (Right (} \sigma_2, x)) \\
\text{iff} \quad \text{interpStmt } s \sigma_1 \in \{\bot, \text{Nothing}\}
\]
Exceptions — Interpreter

Original statement interpretation:

\[ \text{interpStmt} :: \text{Statement} \rightarrow \text{State1} \rightarrow \text{Maybe State1} \]

meaning:

\[
\begin{align*}
\sigma_1(s) &\Rightarrow \sigma_2 \quad \text{iff} \quad \text{interpStmt} s \sigma_1 = \text{Just } \sigma_2 \\
\neg \exists \sigma_2 \bullet \sigma_1(s) &\Rightarrow \sigma_2 \quad \text{iff} \quad \text{interpStmt} s \sigma_1 \in \{ \bot, \text{Nothing} \}
\end{align*}
\]

Statement interpretation with exceptions:

\[ \text{interpStmtExc} :: \text{Statement} \rightarrow \text{State1} \rightarrow \text{Maybe} \left( \text{Either State1 (State1, Exc)} \right) \]

meaning:

\[
\begin{align*}
\sigma_1(s) &\Rightarrow \sigma_2 \quad \text{iff} \quad \text{interpStmt} s \sigma_1 = \text{Just } \left( \text{Left } \sigma_2 \right) \\
\sigma_1(s) &\Rightarrow \left( \sigma_2, x \right) \quad \text{iff} \quad \text{interpStmt} s \sigma_1 = \text{Just } \left( \text{Right } (\sigma_2, x) \right) \\
\neg \exists \sigma_2, x \bullet \sigma_1(s) &\Rightarrow \sigma_2 \quad \lor \\
\sigma_1(s) &\Rightarrow \left( \sigma_2, x \right) \quad \text{iff} \quad \text{interpStmt} s \sigma_1 \in \{ \bot, \text{Nothing} \}
\end{align*}
\]
Exceptions in Expression Evaluation

Exercise
Another Loop Example

\[ P \equiv \textbf{while } x \neq 0 \textbf{ do } s := s + x ; x := x - 1 \textbf{ od} \]

\[ \{ s \mapsto 0, x \mapsto -4 \}(P) \Rightarrow \]

\[ \sigma(b) \Rightarrow \text{True} \quad \sigma(s) \Rightarrow \sigma_1 \quad \sigma_1(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2 \]

\[ \sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2 \quad \sigma(b) \Rightarrow \text{False} \quad \sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma \]
Another Loop Example

\[ P \equiv \textbf{while } x \neq 0 \textbf{ do } s := s + x ; x := x - 1 \textbf{ od} \]

\[
\{ s \mapsto 0, \\
\quad x \mapsto -4 \} \Rightarrow \{ s \mapsto 0, x \mapsto -4 \}(P) \Rightarrow \\
\]

\[
\sigma(b) \Rightarrow \text{True} \quad \sigma(s) \Rightarrow \sigma_1 \quad \sigma_1(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2 \\
\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2 \\
\sigma(b) \Rightarrow \text{False} \quad \sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma
\]
Another Loop Example

\[ P \equiv \textbf{while } x \neq 0 \textbf{ do } s := s + x ; x := x - 1 \textbf{ od} \]

\[
\{s \mapsto 0, \\
x \mapsto -4\} \ (x \neq 0) \Rightarrow \text{True}
\]

\[
\{s \mapsto 0, x \mapsto -4\}(P) \Rightarrow
\]

\[
\sigma(b) \Rightarrow \text{True} \quad \sigma(s) \Rightarrow \sigma_1 \quad \sigma_1(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2 \\
\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2 \\
\sigma(b) \Rightarrow \text{False} \quad \sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma
\]
Another Loop Example

\[ P \equiv \textbf{while} \ x \neq 0 \ \textbf{do} \ s := s + x \ ; \ x := x - 1 \ \textbf{od} \]

\[
\begin{array}{ll}
\{s \mapsto 0, \ x \mapsto -4\} \ (x \neq 0) \Rightarrow \text{True} & \{s \mapsto 0, \ x \mapsto -4\} \ (s := s + x ; x := x - 1) \Rightarrow \\
\{s \mapsto 0, x \mapsto -4\}(P) \Rightarrow \\
\end{array}
\]

\[
\begin{array}{llllll}
\sigma(b) \Rightarrow \text{True} & \sigma(s) \Rightarrow \sigma_1 & \sigma_1(\textbf{while} \ b \ \textbf{do} \ s \ \textbf{od}) \Rightarrow \sigma_2 & \sigma(b) \Rightarrow \text{False} \\
\sigma(\textbf{while} \ b \ \textbf{do} \ s \ \textbf{od}) \Rightarrow \sigma_2 & \sigma(\textbf{while} \ b \ \textbf{do} \ s \ \textbf{od}) \Rightarrow \sigma
\end{array}
\]
Another Loop Example

\[
P \equiv \textbf{while } x \neq 0 \textbf{ do } s := s + x ; x := x - 1 \textbf{ od}
\]

\[
\{s \mapsto 0, \\
x \mapsto -4\} (x \neq 0) \Rightarrow \text{ True} \\
\{s \mapsto 0, \\
x \mapsto -4\} (s := s + x ; x := x - 1) \Rightarrow \{s \mapsto -4, \\
x \mapsto -5\}
\]

\[
\{s \mapsto 0, x \mapsto -4\}(P) \Rightarrow
\]

\[
\sigma(b) \Rightarrow \text{ True} \\
\sigma(s) \Rightarrow \sigma_1 \\
\sigma_1(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2
\]

\[
\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2
\]

\[
\sigma(b) \Rightarrow \text{ False} \\
\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma
\]
Another Loop Example

\[ P \equiv \textbf{while } x \neq 0 \textbf{ do } s := s + x \ ; x := x - 1 \textbf{ od} \]

\[
\begin{align*}
\{s \mapsto 0, \\
x \mapsto -4\} & \Rightarrow \text{True} \\
\{s \mapsto 0, \\
x \mapsto -4\} & \Rightarrow \{s \mapsto -4, \\
x \mapsto -5\}
\end{align*}
\]

\[
\{s \mapsto 0, x \mapsto -4\} (P) \Rightarrow
\]

\[
\begin{align*}
\sigma(b) & \Rightarrow \text{True} \\
\sigma(s) & \Rightarrow \sigma_1 \\
\sigma_1(\textbf{while } b \textbf{ do } s \textbf{ od}) & \Rightarrow \sigma_2 \\
\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) & \Rightarrow \sigma_2 \\
\sigma(b) & \Rightarrow \text{False} \\
\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) & \Rightarrow \sigma
\end{align*}
\]
Another Loop Example

\[ P \equiv \textbf{while} \ x \neq 0 \ \textbf{do} \ s := s + x ; \ x := x - 1 \ \textbf{od} \]

\[
\begin{align*}
\{s \mapsto -4, \ x \mapsto -5\} \ (x \neq 0) & \Rightarrow \\
\{s \mapsto 0, \ x \mapsto -4\} \ (x \neq 0) & \Rightarrow \text{True} \\
\{s \mapsto 0, \ x \mapsto -4\} \ (s := s + x ; \ x := x - 1) & \Rightarrow \{s \mapsto -4, \ x \mapsto -5\} \\
\{s \mapsto 0, x \mapsto -4\} (P) & \Rightarrow \\
\sigma(b) & \Rightarrow \text{True} \quad \sigma(s) \Rightarrow \sigma_1 \quad \sigma_1(\textbf{while} \ b \ \textbf{do} \ s \ \textbf{od}) \Rightarrow \sigma_2 \\
\sigma(\textbf{while} \ b \ \textbf{do} \ s \ \textbf{od}) & \Rightarrow \sigma_2 \\
\sigma(b) & \Rightarrow \text{False} \quad \sigma(\textbf{while} \ b \ \textbf{do} \ s \ \textbf{od}) \Rightarrow \sigma
\end{align*}
\]
Another Loop Example

\[ P \equiv \textbf{while } x \neq 0 \textbf{ do } s := s + x \ ; \ x := x - 1 \textbf{ od} \]

\[
\{s \mapsto -4, \ x \mapsto -5\} \Rightarrow \text{True}
\]

\[
\{s \mapsto 0, \ x \mapsto -4\} \ (x \neq 0) \Rightarrow \text{True} \quad \{s \mapsto 0, \ x \mapsto -4\} \ (s := s + x \ ; \ x := x - 1) \Rightarrow \{s \mapsto -4, \ x \mapsto -5\}
\]

\[
\{s \mapsto 0, \ x \mapsto -4\} \ (P) \Rightarrow 
\]

\[
\sigma(b) \Rightarrow \text{True} \quad \sigma(s) \Rightarrow \sigma_1 \quad \sigma_1(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2 \quad \sigma(b) \Rightarrow \text{False} \quad \sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma
\]

\[
\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2
\]
Another Loop Example

\[ P \equiv \textbf{while } x \neq 0 \textbf{ do } s := s + x ; x := x - 1 \textbf{ od} \]

\[
\begin{align*}
\{ s \mapsto -4, \quad (x \neq 0) \Rightarrow \text{True} \\
\quad \quad x \mapsto -5 \} & \quad \{ s \mapsto -4, \quad (s := s + x ; x := x - 1) \Rightarrow \\
\quad \quad \quad \quad \quad x \mapsto -5 \}
\end{align*}
\]

\[
\begin{align*}
\{ s \mapsto 0, \quad (x \neq 0) \Rightarrow \text{True} \\
\quad \quad x \mapsto -4 \} & \quad \{ s \mapsto 0, \quad (s := s + x ; x := x - 1) \Rightarrow \\
\quad \quad \quad \quad \quad x \mapsto -4 \}
\end{align*}
\]

\[
\{ s \mapsto 0, x \mapsto -4 \} (P) \Rightarrow \\
\{ s \mapsto 0, x \mapsto -4 \} (P) \Rightarrow \\
\{ s \mapsto -4, x \mapsto -5 \}
\]

\[
\begin{align*}
\sigma(b) \Rightarrow \text{True} & \quad \sigma(s) \Rightarrow \sigma_1 & \quad \sigma_1(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2 \\
\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2 & \quad \sigma(b) \Rightarrow \text{False} \quad \sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma
\end{align*}
\]
Another Loop Example

\[ P \equiv \textbf{while } x \neq 0 \textbf{ do } s := s + x \; ; \; x := x - 1 \textbf{ od} \]

\[
\begin{align*}
\{s \mapsto -4, x \mapsto -5\} & \Rightarrow \text{True} \\
\{s \mapsto -4, (s := s + x ; x := x - 1) \Rightarrow \{s \mapsto -9, x \mapsto -6\}\}
\end{align*}
\]

\[
\begin{align*}
\{s \mapsto 0, x \mapsto -4\} & \Rightarrow \text{True} \\
\{s \mapsto 0, (s := s + x ; x := x - 1) \Rightarrow \{s \mapsto -4, x \mapsto -5\}\}
\end{align*}
\]

\[
\{s \mapsto 0, x \mapsto -4\}(P) \Rightarrow
\]

\[
\begin{align*}
\sigma(b) & \Rightarrow \text{True} \\
\sigma(s) & \Rightarrow \sigma_1 \\
\sigma_1(\textbf{while } b \textbf{ do } s \textbf{ od}) & \Rightarrow \sigma_2
\end{align*}
\]

\[
\begin{align*}
\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) & \Rightarrow \sigma_2 \\
\sigma(b) & \Rightarrow \text{False}
\end{align*}
\]

\[
\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma
\]
Another Loop Example

\[ P \equiv \textbf{while } x \neq 0 \textbf{ do } s := s + x \textbf{ ; } x := x - 1 \textbf{ od} \]

\[
\begin{array}{ll}
\{s \mapsto -4, \ x \mapsto -5\} (x \neq 0) \Rightarrow \text{True} & \{s \mapsto -4, \ s := s + x \textbf{ ; } x := x - 1\} \Rightarrow \{s \mapsto -9, \ x \mapsto -6\} \\
\{s \mapsto 0, \ x \mapsto -4\} (x \neq 0) \Rightarrow \text{True} & \{s \mapsto 0, \ s := s + x \textbf{ ; } x := x - 1\} \Rightarrow \{s \mapsto -4, \ x \mapsto -5\} \Rightarrow \{s \mapsto 0, x \mapsto -4\} (P) \Rightarrow \\
\end{array}
\]

\[
\sigma(b) \Rightarrow \text{True} \quad \sigma(s) \Rightarrow \sigma_1 \quad \sigma_1(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2 \\
\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2 \\
\sigma(b) \Rightarrow \text{False} \quad \sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma
\]
Another Loop Example

\[ P \equiv \text{while } x \neq 0 \text{ do } s := s + x \ ; x := x - 1 \text{ od} \]

\[
\begin{align*}
\{ s \mapsto -9, \quad (x \neq 0) \Rightarrow x \mapsto -6 \} \\
\{ s \mapsto -4, \quad (x \neq 0) \Rightarrow \text{True} \quad (s := s + x \ ; x := x - 1) \Rightarrow \{ s \mapsto -9, \quad x \mapsto -6 \} \\
\{ s \mapsto 0, \quad (x \neq 0) \Rightarrow \text{True} \quad (s := s + x \ ; x := x - 1) \Rightarrow \{ s \mapsto -4, \quad x \mapsto -5 \} \\
\{ s \mapsto 0, x \mapsto -4 \} (P) \Rightarrow \{ s \mapsto 0, x \mapsto -4 \} (P) \Rightarrow \{ s \mapsto 0, x \mapsto -4 \} (P) \Rightarrow \{ s \mapsto 0, x \mapsto -4 \} (P) \Rightarrow
\end{align*}
\]

\[
\sigma(b) \Rightarrow \text{True} \quad \sigma(s) \Rightarrow \sigma_1 \quad \sigma_1(\text{while } b \text{ do } s \text{ od}) \Rightarrow \sigma_2 \quad \sigma(b) \Rightarrow \text{False} \quad \sigma(\text{while } b \text{ do } s \text{ od}) \Rightarrow \sigma
\]
Another Loop Example

\[ P \equiv \textbf{while} \ x \neq 0 \ \textbf{do} \ s := s + x ; \ x := x - 1 \ \textbf{od} \]

\[
\{s \mapsto -9, \ x \mapsto -6\} \ (x \neq 0) \Rightarrow \text{True} \\
\{s \mapsto -4, \ x \mapsto -5\} \ (x \neq 0) \Rightarrow \text{True} \\
\{s \mapsto 0, \ x \mapsto -4\} \ (x \neq 0) \Rightarrow \text{True} \\
\{s \mapsto 0, \ x \mapsto -4\} \ (s := s + x ; x := x - 1) \Rightarrow \{s \mapsto -9, \ x \mapsto -6\} \\
\{s \mapsto -4, \ x \mapsto -5\} \ (s := s + x ; x := x - 1) \Rightarrow \{s \mapsto -4, \ x \mapsto -5\} \\
\{s \mapsto 0, x \mapsto -4\} \ (P) \Rightarrow \\
\{s \mapsto -9, (P) \Rightarrow \\
\{s \mapsto -4, (P) \Rightarrow \\
\{s \mapsto 0, (P) \Rightarrow \\
\{s \mapsto 0, x \mapsto -4\} \ (P) \Rightarrow \\
\{s \mapsto -9, x \mapsto -6\} \ (P) \Rightarrow \\
\{s \mapsto -4, x \mapsto -5\} \ (P) \Rightarrow \\
\{s \mapsto 0, x \mapsto -4\} \ (P) \Rightarrow \\
\sigma(b) \Rightarrow \text{True} \quad \sigma(s) \Rightarrow \sigma_1 \quad \sigma_1(\textbf{while} \ b \ \textbf{do} \ s \ \textbf{od}) \Rightarrow \sigma_2 \\
\sigma(\textbf{while} \ b \ \textbf{do} \ s \ \textbf{od}) \Rightarrow \sigma_2 \\
\sigma(b) \Rightarrow \text{False} \quad \sigma(\textbf{while} \ b \ \textbf{do} \ s \ \textbf{od}) \Rightarrow \sigma
Another Loop Example

\[ P \equiv \textbf{while} \ x \neq 0 \ \textbf{do} \ s := s + x \ ; \ x := x - 1 \ \textbf{od} \]

<table>
<thead>
<tr>
<th>{s \mapsto 9, \ (x \neq 0) \Rightarrow \text{True} }</th>
<th>{s \mapsto 9, \ (s := s + x ; x := x - 1) \Rightarrow }</th>
</tr>
</thead>
<tbody>
<tr>
<td>{s \mapsto 4, \ (x \neq 0) \Rightarrow \text{True} }</td>
<td>{s \mapsto 4, \ (s := s + x ; x := x - 1) \Rightarrow {s \mapsto 9, \ x \mapsto 6}}</td>
</tr>
<tr>
<td>{s \mapsto 0, \ (x \neq 0) \Rightarrow \text{True} }</td>
<td>{s \mapsto 0, \ (s := s + x ; x := x - 1) \Rightarrow {s \mapsto 4, \ x \mapsto 5}}</td>
</tr>
</tbody>
</table>
| \{s \mapsto 0, \ x \mapsto 4\} \Rightarrow \{s \mapsto 0, \ x \mapsto 4\} \Rightarrow \}

\[ \sigma(b) \Rightarrow \text{True} \quad \sigma(s) \Rightarrow \sigma_1 \quad \sigma_1(\textbf{while} \ b \ \textbf{do} \ s \ \textbf{od}) \Rightarrow \sigma_2 \]

\[ \sigma(\textbf{while} \ b \ \textbf{do} \ s \ \textbf{od}) \Rightarrow \sigma_2 \quad \sigma(b) \Rightarrow \text{False} \quad \sigma(\textbf{while} \ b \ \textbf{do} \ s \ \textbf{od}) \Rightarrow \sigma \]
Another Loop Example

\[ P \equiv \textbf{while} \ x \neq 0 \ \textbf{do} \ s := s + x \ ; \ x := x - 1 \ \textbf{od} \]

<table>
<thead>
<tr>
<th>Initial State</th>
<th>Predicate</th>
<th>Final State</th>
</tr>
</thead>
<tbody>
<tr>
<td>{s \leftarrow -9, \ x \leftarrow -6} (x \neq 0) \Rightarrow \text{True}</td>
<td>{s \leftarrow -9, \ (s := s + x ; x := x - 1) \Rightarrow {s \leftarrow -15, \ x \leftarrow -7}</td>
<td>{s \leftarrow -9, \ x \leftarrow -6} (P) \Rightarrow</td>
</tr>
<tr>
<td>{s \leftarrow -4, \ x \leftarrow -5} (x \neq 0) \Rightarrow \text{True}</td>
<td>{s \leftarrow -4, \ (s := s + x ; x := x - 1) \Rightarrow {s \leftarrow -9, \ x \leftarrow -6}</td>
<td>{s \leftarrow -4, \ x \leftarrow -5} (P) \Rightarrow</td>
</tr>
<tr>
<td>{s \leftarrow 0, \ x \leftarrow -4} (x \neq 0) \Rightarrow \text{True}</td>
<td>{s \leftarrow 0, \ (s := s + x ; x := x - 1) \Rightarrow {s \leftarrow -4, \ x \leftarrow -5}</td>
<td>{s \leftarrow 0, x \leftarrow -4} (P) \Rightarrow</td>
</tr>
</tbody>
</table>

\sigma(b) \Rightarrow \text{True} \quad \sigma(s) \Rightarrow \sigma_1 \quad \sigma_1(\textbf{while} \ b \ \textbf{do} \ s \ \textbf{od}) \Rightarrow \sigma_2 \quad \sigma(b) \Rightarrow \text{False} \quad \sigma(\textbf{while} \ b \ \textbf{do} \ s \ \textbf{od}) \Rightarrow \sigma

\sigma(\textbf{while} \ b \ \textbf{do} \ s \ \textbf{od}) \Rightarrow \sigma_2
Another Loop Example

\[ P \equiv \textbf{while } x \neq 0 \textbf{ do } s := s + x ; x := x - 1 \textbf{ od} \]

\[
\begin{align*}
\{s &\rightarrow -9, \quad (x \neq 0) \Rightarrow \text{True} \\
&\quad x \rightarrow -6 \} \\
\{s &\rightarrow -4, \quad (x \neq 0) \Rightarrow \text{True} \\
&\quad x \rightarrow -5 \} \\
\{s &\rightarrow 0, \quad (x \neq 0) \Rightarrow \text{True} \\
&\quad x \rightarrow -4 \}
\end{align*}
\]

\[
\begin{align*}
\{s &\rightarrow -9, \quad (s := s + x ; x := x - 1) \Rightarrow \{s \rightarrow -15, \quad x \rightarrow -7\} \\
&\quad (P) \Rightarrow \} \\
\{s &\rightarrow -4, \quad (s := s + x ; x := x - 1) \Rightarrow \{s \rightarrow -9, \quad x \rightarrow -6\} \\
&\quad (P) \Rightarrow \} \\
\{s &\rightarrow -4, \quad (s := s + x ; x := x - 1) \Rightarrow \{s \rightarrow -4, \quad x \rightarrow -5\} \\
&\quad (P) \Rightarrow \}
\end{align*}
\]

\[\{s \rightarrow 0, x \rightarrow -4\} (P) \Rightarrow\]

\[
\begin{align*}
\sigma(b) &\Rightarrow \text{True} \\
\sigma(s) &\Rightarrow \sigma_1 \\
\sigma_1(\textbf{while } b \textbf{ do } s \textbf{ od}) &\Rightarrow \sigma_2 \\
\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) &\Rightarrow \sigma_2 \\
\end{align*}
\]

\[
\begin{align*}
\sigma(b) &\Rightarrow \text{False} \\
\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) &\Rightarrow \sigma
\end{align*}
\]
Another Loop Example

\[ P \equiv \textbf{while } x \neq 0 \textbf{ do } s := s + x ; x := x - 1 \textbf{ od} \]

\[
\begin{align*}
\{s \mapsto -9, x \mapsto -6\} \quad &\text{(}x \neq 0\text{)} \Rightarrow \text{True} \\
\{s \mapsto -9, x \mapsto -6\} \quad &\text{(}s := s + x ; x := x - 1\text{)} \Rightarrow \{s \mapsto -15, x \mapsto -7\} \\
\{s \mapsto -9, x \mapsto -6\} \quad &\text{(}P\text{)} \Rightarrow \\
\end{align*}
\]

\[
\begin{align*}
\{s \mapsto -4, x \mapsto -5\} \quad &\text{(}x \neq 0\text{)} \Rightarrow \text{True} \\
\{s \mapsto -4, x \mapsto -5\} \quad &\text{(}s := s + x ; x := x - 1\text{)} \Rightarrow \{s \mapsto -9, x \mapsto -6\} \\
\{s \mapsto -4, x \mapsto -5\} \quad &\text{(}P\text{)} \Rightarrow \\
\end{align*}
\]

\[
\begin{align*}
\{s \mapsto 0, x \mapsto -4\} \quad &\text{(}x \neq 0\text{)} \Rightarrow \text{True} \\
\{s \mapsto 0, x \mapsto -4\} \quad &\text{(}s := s + x ; x := x - 1\text{)} \Rightarrow \{s \mapsto -4, x \mapsto -5\} \\
\{s \mapsto 0, x \mapsto -4\} \quad &\text{(}P\text{)} \Rightarrow \\
\end{align*}
\]

\[\{s \mapsto 0, x \mapsto -4\}(P) \Rightarrow\]

\[\sigma(b) \Rightarrow \text{True} \quad \sigma(s) \Rightarrow \sigma_1 \quad \sigma_1(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2 \]

\[\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2\]

\[\sigma(b) \Rightarrow \text{False} \quad \sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma\]
Another Loop Example

\[ P \equiv \textbf{while } x \neq 0 \textbf{ do } s := s + x ; x := x - 1 \textbf{ od} \]

\[
\begin{align*}
\{s \mapsto -9, \ x \mapsto -6\} & \quad (x \neq 0) \Rightarrow \text{True} \\
\{s \mapsto -9, \ x \mapsto -6\} & \quad (s := s + x ; x := x - 1) \Rightarrow \{s \mapsto -15, \ x \mapsto -7\}
\end{align*}
\]

\[
\begin{align*}
\{s \mapsto -4, \ x \mapsto -5\} & \quad (x \neq 0) \Rightarrow \text{True} \\
\{s \mapsto -4, \ x \mapsto -5\} & \quad (s := s + x ; x := x - 1) \Rightarrow \{s \mapsto -9, \ x \mapsto -6\} \\
\{s \mapsto -4, \ x \mapsto -5\} & \quad (P) \Rightarrow \\
\{s \mapsto -9, \ x \mapsto -6\} & \quad (P) \Rightarrow \\
\{s \mapsto -4, \ x \mapsto -5\} & \quad (P) \Rightarrow \\
\{s \mapsto 0, \ x \mapsto -4\} & \quad (x \neq 0) \Rightarrow \text{True} \\
\{s \mapsto 0, \ x \mapsto -4\} & \quad (s := s + x ; x := x - 1) \Rightarrow \{s \mapsto -4, \ x \mapsto -5\} \\
\{s \mapsto 0, \ x \mapsto -4\} & \quad (P) \Rightarrow
\end{align*}
\]

\[
\{s \mapsto 0, \ x \mapsto -4\}(P) \Rightarrow
\]

\[
\sigma(b) \Rightarrow \text{True} \quad \sigma(s) \Rightarrow \sigma_1 \quad \sigma_1(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2 \\
\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2 \\
\sigma(b) \Rightarrow \text{False} \quad \sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma
\]
Another Loop Example

\[ P \equiv \textbf{while } x \neq 0 \textbf{ do } s := s + x \textbf{ ; } x := x - 1 \textbf{ od} \]

\[
\begin{align*}
\{s \leftarrow -9, x \leftarrow -6\} & \Rightarrow \text{True} & \{s \leftarrow -9, x \leftarrow -6\} & \Rightarrow \{s \leftarrow 15, x \leftarrow 7\} & \Rightarrow \{s \leftarrow -15, x \leftarrow 7\} \\
\{s \leftarrow -4, x \leftarrow -5\} & \Rightarrow \text{True} & \{s \leftarrow -4, x \leftarrow -5\} & \Rightarrow \{s \leftarrow -9, x \leftarrow -6\} & \Rightarrow \{s \leftarrow 15, x \leftarrow 7\} \\
\{s \leftarrow 0, x \leftarrow -4\} & \Rightarrow \text{True} & \{s \leftarrow 0, x \leftarrow -4\} & \Rightarrow \{s \leftarrow -4, x \leftarrow -5\} & \Rightarrow \{s \leftarrow 15, x \leftarrow 7\}
\end{align*}
\]

This is not a direct proof of non-termination!

\[
\begin{align*}
\sigma(b) & \Rightarrow \text{True} & \sigma(s) & \Rightarrow \sigma_1 & \sigma_1(\textbf{while } b \textbf{ do } s \textbf{ od}) & \Rightarrow \sigma_2 \\
\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) & \Rightarrow \sigma_2 & \sigma(b) & \Rightarrow \text{False} & \sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) & \Rightarrow \sigma
\end{align*}
\]
From Operational to Relational Denotational Semantics

The derivable assertions of shape \( \sigma(e) \Rightarrow \nu \)
The derivable assertions of shape \( \sigma(e) \Rightarrow v \), meaning:

```
“Evaluating expression \( e \) starting in state \( \sigma \) can produce value \( v \)”
```
From Operational to Relational Denotational Semantics

The derivable assertions of shape “σ(e) ⇒ v”, meaning:

“Evaluating expression e starting in state σ can produce value v”

give rise to a ternary relation evalExpr_0 : P (State × Expr × Value)
From Operational to Relational Denotational Semantics

The derivable assertions of shape \( \sigma(e) \Rightarrow v \), meaning:

\[
\text{“Evaluating expression } e \text{ starting in state } \sigma \text{ can produce value } v \”
\]
give rise to a ternary relation \( \text{evalExpr}_0 : \mathcal{P} (State_1 \times Expr \times Value) \)
which is equivalent to a total relation-valued function:

\[
M_{Expr} : Expr \to \mathcal{P} (State_1 \times Value)
\]
From Operational to Relational Denotational Semantics

The derivable assertions of shape “$\sigma(e) \Rightarrow v$”, meaning:

“Evaluating expression $e$ starting in state $\sigma$ can produce value $v$”
give rise to a ternary relation $\text{evalExpr}_0 : \mathbb{P} (State_1 \times Expr \times Value)$
which is equivalent to a total relation-valued function:

$M_{\text{Expr}} : Expr \rightarrow \mathbb{P} (State_1 \times Value)$

$M_{\text{Expr}} : Expr \rightarrow (State_1 \leftrightarrow Value)$
From Operational to Relational Denotational Semantics

The derivable assertions of shape “$\sigma(e) \Rightarrow v$”, meaning:

“Evaluating expression $e$ starting in state $\sigma$ can produce value $v$”

give rise to a ternary relation $\text{evalExpr}_0 : \mathcal{P}(\text{State}_1 \times \text{Expr} \times \text{Value})$

which is equivalent to a total relation-valued function:

$M_{\text{Expr}} : \text{Expr} \rightarrow \mathcal{P}(\text{State}_1 \times \text{Value})$

$M_{\text{Expr}} : \text{Expr} \rightarrow (\text{State}_1 \leftrightarrow \text{Value})$

If expression evaluation is deterministic, the result relations are all partial functions
From Operational to Relational Denotational Semantics

The derivable assertions of shape “\( \sigma(e) \Rightarrow v \)”, meaning:

“Evaluating expression \( e \) starting in state \( \sigma \) can produce value \( v \)”

give rise to a ternary relation \( \text{evalExpr}_0 : \mathcal{P} (State_1 \times Expr \times Value) \)

which is equivalent to a total relation-valued function:

\[
M_{Expr} : Expr \rightarrow \mathcal{P} (State_1 \times Value)
\]

\[
M_{Expr} : Expr \rightarrow (State_1 \leftrightarrow Value)
\]

If expression evaluation is deterministic, the result relations are all partial functions:

\[
M_{Expr} : Expr \rightarrow (State_1 \leftrightarrow Value)
\]
From Operational to Relational Denotational Semantics

The derivable assertions of shape “$\sigma(e) \Rightarrow v$”, meaning:

“Evaluating expression $e$ starting in state $\sigma$ can produce value $v$”

give rise to a ternary relation $\text{evalExpr}_0 : \mathcal{P} (State_1 \times Expr \times Value)$

which is equivalent to a total relation-valued function:

$$M_{\text{Expr}} : \text{Expr} \rightarrow \mathcal{P} (State_1 \times Value)$$

$$M_{\text{Expr}} : \text{Expr} \rightarrow (State_1 \leftrightarrow Value)$$

If expression evaluation is deterministic, the result relations are all partial functions:

$$M_{\text{Expr}} : \text{Expr} \rightarrow (State_1 \mapsto Value)$$

This corresponds to the Haskell type we have chosen:

$\text{evalExpr} :: \text{Expression} \rightarrow (\text{State1} \rightarrow \text{Maybe Value1})$
From Operational to Relational Denotational Semantics

The derivable assertions of shape “$\sigma(e) \Rightarrow \nu$”, meaning:

“Evaluating expression $e$ starting in state $\sigma$ can produce value $\nu$”
give rise to a ternary relation $\text{evalExpr}_0 : \mathbb{P} (\text{State}_1 \times \text{Expr} \times \text{Value})$

which is equivalent to a total relation-valued function:

$$M_{\text{Expr}} : \text{Expr} \rightarrow \mathbb{P} (\text{State}_1 \times \text{Value})$$

$$M_{\text{Expr}} : \text{Expr} \rightarrow (\text{State}_1 \leftrightarrow \text{Value})$$

If expression evaluation is deterministic, the result relations are all partial functions:

$$M_{\text{Expr}} : \text{Expr} \rightarrow (\text{State}_1 \mapsto \text{Value})$$

This corresponds to the Haskell type we have chosen:

$\text{evalExpr} :: \text{Expression} \rightarrow (\text{State}1 \rightarrow \text{Maybe Value}1)$

**Note:** For an expression $e$, we write “$\llbracket e \rrbracket_E$” instead of “$M_{\text{Expr}}(e)$”.
Relational Denotational Statement Semantics

The derivable assertions of shape \( \sigma(s) \Rightarrow \sigma' \), meaning:
Relational Denotational Statement Semantics

The derivable assertions of shape "σ(s) ⇒ σ′", meaning:

“Executing statement s starting in state σ can terminate in state σ′”
Relational Denotational Statement Semantics

The derivable assertions of shape “\( \sigma(s) \Rightarrow \sigma' \)”, meaning:

“Executing statement \( s \) starting in state \( \sigma \) can terminate in state \( \sigma' \)”
give rise to a ternary relation \( \text{execStmt}_0 : \mathbb{P} (State_1 \times Stmt \times State_1) \)
Relational Denotational Statement Semantics

The derivable assertions of shape \( \sigma(s) \Rightarrow \sigma' \), meaning:

"Executing statement \( s \) starting in state \( \sigma \) can terminate in state \( \sigma' \)"

give rise to a ternary relation \( \text{execStmt}_0 : \mathcal{P} (\text{State}_1 \times \text{Stmt} \times \text{State}_1) \)

which is equivalent to a total relation-valued function:

\[
M_{\text{Stmt}} : \text{Stmt} \rightarrow (\text{State}_1 \leftrightarrow \text{State}_1)
\]
Relational Denotational Statement Semantics

The derivable assertions of shape \( \sigma(s) \Rightarrow \sigma' \)”, meaning:

“Executing statement \( s \) starting in state \( \sigma \) can terminate in state \( \sigma' \)”

give rise to a ternary relation \( \text{execStmt}_0 : \mathcal{P}(State_1 \times Stmt \times State_1) \)

which is equivalent to a total relation-valued function:

\[
M_{\text{Stmt}} : Stmt \rightarrow (State_1 \leftrightarrow State_1)
\]

If statement execution is deterministic, we have partial functions again:

\[
M_{\text{Stmt}} : Stmt \rightarrow (State_1 \rightarrow State_1)
\]
Relational Denotational Statement Semantics

The derivable assertions of shape “$\sigma(s) \Rightarrow \sigma'$”, meaning:

“Executing statement $s$ starting in state $\sigma$ can terminate in state $\sigma'$”
give rise to a ternary relation $\text{execStmt}_0 : \mathcal{P} (\text{State}_1 \times \text{Stmt} \times \text{State}_1)$
which is equivalent to a total relation-valued function:

$$M_{\text{Stmt}} : \text{Stmt} \rightarrow (\text{State}_1 \leftrightarrow \text{State}_1)$$

If statement execution is deterministic, we have partial functions again:

$$M_{\text{Stmt}} : \text{Stmt} \rightarrow (\text{State}_1 \Rightarrow \text{State}_1)$$

Note: For a statement $s$, we write “$[[s]]_S$” instead of “$M_{\text{Stmt}}(s)$”.
Relational Denotational Statement Semantics

The derivable assertions of shape “\( \sigma(s) \Rightarrow \sigma' \)”, meaning:

“Executing statement \( s \) starting in state \( \sigma \) can terminate in state \( \sigma' \)”

give rise to a ternary relation \( \text{execStmt}_0 : \mathbb{P}(\text{State}_1 \times \text{Stmt} \times \text{State}_1) \)

which is equivalent to a total relation-valued function:

\[
M_{\text{Stmt}} : \text{Stmt} \rightarrow (\text{State}_1 \leftrightarrow \text{State}_1)
\]

If statement execution is deterministic, we have partial functions again:

\[
M_{\text{Stmt}} : \text{Stmt} \rightarrow (\text{State}_1 \rightarrow \text{State}_1)
\]

Note: For a statement \( s \), we write “\( \llbracket s \rrbracket_s \)” instead of “\( M_{\text{Stmt}}(s) \)”.

This can be used for proving undefinedness.
Relational Denotational Statement Semantics

The derivable assertions of shape “σ(s) ⇒ σ′”, meaning:

“Executing statement s starting in state σ can terminate in state σ′”
give rise to a ternary relation \( \text{execStmt}_0 : \mathcal{P}(\text{State}_1 \times \text{Stmt} \times \text{State}_1) \)
which is equivalent to a total relation-valued function:

\[
M_{\text{Stmt}} : \text{Stmt} \rightarrow (\text{State}_1 \leftrightarrow \text{State}_1)
\]

If statement execution is deterministic, we have partial functions again:

\[
M_{\text{Stmt}} : \text{Stmt} \rightarrow (\text{State}_1 \rightarrow \text{State}_1)
\]

Note: For a statement s, we write “[\text{[s]}_S]” instead of “\(M_{\text{Stmt}}(s)\)”.

This can be used for proving undefinedness:

\[
[[\text{while } x \neq 0 \text{ do } s := s + x ; x := x - 1 \text{ od}]]_S
\]
Relational Denotational Statement Semantics

The derivable assertions of shape “$\sigma(s) \Rightarrow \sigma$”, meaning:

“Executing statement $s$ starting in state $\sigma$ can terminate in state $\sigma$”

give rise to a ternary relation $\text{execStmt}_0 : \mathcal{P} (State_1 \times Stmt \times State_1)$

which is equivalent to a total relation-valued function:

$$M_{Stmt} : Stmt \rightarrow (State_1 \leftrightarrow State_1)$$

If statement execution is deterministic, we have partial functions again:

$$M_{Stmt} : Stmt \rightarrow (State_1 \rightarrow State_1)$$

**Note:** For a statement $s$, we write “$[[s]]_S$” instead of “$M_{Stmt}(s)$”.

This can be used for proving **undefinedness**:

$$\text{dom} \ [[\text{while } x \neq 0 \ do \ s := s + x ; x := x - 1 \ od]]_S$$
Relational Denotational Statement Semantics

The derivable assertions of shape “$\sigma(s) \Rightarrow \sigma'$”, meaning:

“Executing statement $s$ starting in state $\sigma$ can terminate in state $\sigma'$”
give rise to a ternary relation $\text{execStmt}_0 : \mathcal{P}(\text{State}_1 \times \text{Stmt} \times \text{State}_1)$
which is equivalent to a total relation-valued function:

$$M_{\text{Stmt}} : \text{Stmt} \rightarrow (\text{State}_1 \leftrightarrow \text{State}_1)$$

If statement execution is deterministic, we have partial functions again:

$$M_{\text{Stmt}} : \text{Stmt} \rightarrow (\text{State}_1 \rightarrow \text{State}_1)$$

Note: For a statement $s$, we write “$\llbracket s \rrbracket_S$” instead of “$M_{\text{Stmt}}(s)$”.

This can be used for proving undefinedness:

$$\{s \mapsto 0, x \mapsto -4\} \notin \text{dom } \llbracket \text{while } x \neq 0 \text{ do } s := s + x ; x := x - 1 \text{ od} \rrbracket_S$$
Relational Denotational Statement Semantics

The derivable assertions of shape \("\sigma(s) \Rightarrow \sigma'\)\), meaning:

\("\text{Executing statement } s \text{ starting in state } \sigma \text{ can terminate in state } \sigma'\)\)

give rise to a ternary relation \(\text{execStmt}_0 : \mathbb{P} (State_1 \times Stmt \times State_1)\)

which is equivalent to a total relation-valued function:

\[ M_{Stmt} : Stmt \rightarrow (State_1 \leftrightarrow State_1) \]

If statement execution is deterministic, we have partial functions again:

\[ M_{Stmt} : Stmt \rightarrow (State_1 \rightarrow State_1) \]

Note: For a statement \(s\), we write \("\llbracket s \rrbracket_S\)" instead of \("M_{Stmt}(s)\)".

This can be used for proving undefinedness:

\{s \mapsto 0, x \mapsto -4\} \notin \text{dom} \llbracket \text{while } x \neq 0 \text{ do } s := s + x \text{ ; } x := x - 1 \text{ od} \rrbracket_S \]

This uses properties of mathematical object found as denotational semantics of a statement.
Denotational Semantics is Compositional

\[
\text{evalExpr :: Expression} \rightarrow (\text{State1} \rightarrow \text{Maybe Value1})
\]
Denotational Semantics is Compositional

\[ evalExpr :: Expression \rightarrow (State1 \rightarrow Maybe\ Value1) \]

We can reflect these parentheses in the definition:

\[ evalExpr(\text{Var } v) = \lambda s \rightarrow \text{lookupFM } s \ v \]
\[ evalExpr(\text{Value } \text{lit}) = \text{const} (\text{Just} (\text{litToVal } \text{lit})) \]
Denotational Semantics is Compositional

\[ \text{evalExpr} :: \text{Expression} \rightarrow (\text{State1} \rightarrow \text{Maybe Value1}) \]

We can reflect these parentheses in the definition:

\[ \text{evalExpr} (\text{Var}\ v) = \lambda s \rightarrow \text{lookupFM}\ s\ v \]
\[ \text{evalExpr} (\text{Value}\ \text{lit}) = \text{const} (\text{Just} (\text{litToVal}\ \text{lit})) \]
\[ \text{evalExpr} (\text{Binary} (\text{MkArithOp Plus})\ e1\ e2) = \lambda s \rightarrow \text{case} ((\text{evalExpr}\ e1)\ s, (\text{evalExpr}\ e2)\ s)\ of \]
\[ \text{Just} (\text{VallInt}\ v1), \text{Just} (\text{VallInt}\ v2)) \rightarrow \text{Just} (\text{VallInt} (v1 + v2)) \]
\[ _- \rightarrow \text{Nothing} \]
Denotational Semantics is Compositional

evalExpr :: Expression → (State1 → Maybe Value1)

We can reflect these parentheses in the definition:

\[
evalExpr (\text{Var } v) = \lambda s \rightarrow \text{lookupFM } s \ v
\]
\[
evalExpr (\text{Value } \text{lit}) = \text{const} (\text{Just } (\text{litToVal } \text{lit}))
\]
\[
evalExpr (\text{Binary } (\text{MkArithOp Plus}) e1 e2) = \lambda s \rightarrow
\]
\[
\textbf{case} ((\text{evalExpr } e1) s, (\text{evalExpr } e2) s) \text{ of}
\]
\[
(\text{Just } (\text{VallInt } v1), \text{Just } (\text{VallInt } v2)) \rightarrow \text{Just } (\text{VallInt } (v1 + v2))
\]
\[
\_ \quad \_ \rightarrow \text{Nothing}
\]

Translating the last case back into mathematical notation:
Denotational Semantics is Compositional

\[ \text{evalExpr} :: \text{Expression} \rightarrow (\text{State} \rightarrow \text{Maybe Value}) \]

We can reflect these parentheses in the definition:

\[
\begin{align*}
\text{evalExpr} (\text{Var \ v}) &= \lambda s \rightarrow \text{lookupFM} s \ v \\
\text{evalExpr} (\text{Value \ lit}) &= \text{const} (\text{Just} (\text{litToVal} \ \text{lit})) \\
\text{evalExpr} (\text{Binary} (\text{MkArithOp \ Plus}) \ e1 \ e2) &= \lambda s \rightarrow \\
\text{case} ((\text{evalExpr} \ e1) \ s, (\text{evalExpr} \ e2) \ s) \ 	ext{of} \\
(\text{Just} (\text{Vallnt} \ v1), \text{Just} (\text{Vallnt} \ v2)) &\rightarrow \text{Just} (\text{Vallnt} (v1 + v2)) \\
_ &\rightarrow \text{Nothing}
\end{align*}
\]

Translating the last case back into mathematical notation:

\[
\begin{align*}
[ e_1 + e_2 ]_E := \lambda s &\rightarrow [ e_1 ]_E s + [ e_2 ]_E s
\end{align*}
\]
Denotational Semantics is Compositional

\[ \text{evalExpr :: Expression} \rightarrow (\text{State1} \rightarrow \text{Maybe Value1}) \]

We can reflect these parentheses in the definition:

\[
\begin{align*}
\text{evalExpr (Var v)} & = \lambda s \rightarrow \text{lookupFM} s v \\
\text{evalExpr (Value lit)} & = \text{const (Just (litToVal lit))} \\
\text{evalExpr (Binary (MkArithOp Plus) e1 e2)} & = \lambda s \rightarrow \\
& \text{case } ( (\text{evalExpr e1}) s, (\text{evalExpr e2}) s ) \text{ of} \\
& (\text{Just (VallInt v1), Just (VallInt v2)}) \rightarrow \text{Just (VallInt (v1 + v2))} \\
& _ \rightarrow \text{Nothing}
\end{align*}
\]

Translating the last case back into mathematical notation:

\[
\llbracket e_1 + e_2 \rrbracket_E := \lambda s \rightarrow \llbracket e_1 \rrbracket_E s + \llbracket e_2 \rrbracket_E s \quad \text{(using partial operations)}
\]
Denotational Semantics is Compositional

evalExpr :: Expression → (State1 → Maybe Value1)

We can reflect these parentheses in the definition:

evalExpr (Var v) = λ s → lookupFM s v
evalExpr (Value lit) = const (Just (litToVal lit))
evalExpr (Binary (MkArithOp Plus) e1 e2) = λ s →
  case ((evalExpr e1) s, (evalExpr e2) s) of
    (Just (Vallnt v1), Just (Vallnt v2)) → Just (Vallnt (v1 + v2))
    _ → Nothing

Translating the last case back into mathematical notation:

\[
\llbracket e_1 + e_2 \rrbracket_E := \lambda s \rightarrow \llbracket e_1 \rrbracket_E s + \llbracket e_2 \rrbracket_E s
\]

(Using partial operations)

Compositional Semantics

The semantics of each syntactic construct is defined in terms of the semantics of its constituents.
Sequencing, Conditionals, Loops in Operational Semantics

\[ \sigma_1(s_1) \Rightarrow \sigma_2 \quad \sigma_2(s_2) \Rightarrow \sigma_3 \]

\[ \sigma_1(s_1; s_2) \Rightarrow \sigma_3 \]

\[ \sigma(b) \Rightarrow \text{True} \quad \sigma(s_1) \Rightarrow \sigma_1 \quad \sigma(b) \Rightarrow \text{False} \quad \sigma(s_2) \Rightarrow \sigma_2 \]

\[ \sigma(\text{if } b \text{ then } s_1 \text{ else } s_2 \text{ fi}) \Rightarrow \sigma_1 \quad \sigma(\text{if } b \text{ then } s_1 \text{ else } s_2 \text{ fi}) \Rightarrow \sigma_2 \]

\[ \sigma(b) \Rightarrow \text{False} \]

\[ \sigma(\text{while } b \text{ do } s \text{ od}) \Rightarrow \sigma \]

\[ \sigma(b) \Rightarrow \text{True} \quad \sigma(s) \Rightarrow \sigma_1 \quad \sigma_1(\text{while } b \text{ do } s \text{ od}) \Rightarrow \sigma_2 \]

\[ \sigma(\text{while } b \text{ do } s \text{ od}) \Rightarrow \sigma_2 \]
Sequencing, Conditionals, Loops in Operational Semantics

\[
\begin{align*}
\sigma_1(s_1) \Rightarrow \sigma_2 & \quad \quad \sigma_2(s_2) \Rightarrow \sigma_3 \\
\hline
\sigma_1(s_1; s_2) \Rightarrow \sigma_3
\end{align*}
\]

\[
\begin{align*}
\sigma(b) \Rightarrow \text{True} & \quad \quad \sigma(s_1) \Rightarrow \sigma_1 \\
\sigma(\text{if } b \text{ then } s_1 \text{ else } s_2 \text{ fi}) \Rightarrow \sigma_1
\end{align*}
\]

\[
\begin{align*}
\sigma(b) \Rightarrow \text{False} & \quad \quad \sigma(s_2) \Rightarrow \sigma_2 \\
\sigma(\text{if } b \text{ then } s_1 \text{ else } s_2 \text{ fi}) \Rightarrow \sigma_2
\end{align*}
\]

\[
\begin{align*}
\sigma(b) \Rightarrow \text{False} \\
\sigma(\text{while } b \text{ do } s \text{ od}) \Rightarrow \sigma
\end{align*}
\]

\[
\begin{align*}
\sigma(b) \Rightarrow \text{True} & \quad \quad \sigma(s) \Rightarrow \sigma_1 \\
\sigma_1(\text{while } b \text{ do } s \text{ od}) \Rightarrow \sigma_2 \\
\sigma(\text{while } b \text{ do } s \text{ od}) \Rightarrow \sigma_2
\end{align*}
\]

*The last operational semantics rule here is not compositional!*
Interpreter: Sequencing

\[ \sigma_1(s_1) \Rightarrow \sigma_2 \quad \sigma_2(s_2) \Rightarrow \sigma_3 \]

\[ \sigma_1(s_1 ; s_2) \Rightarrow \sigma_3 \]
Interpreter: Sequencing

\[
\sigma_1(s_1) \Rightarrow \sigma_2 \quad \quad \sigma_2(s_2) \Rightarrow \sigma_3 \\
\sigma_1(s_1; s_2) \Rightarrow \sigma_3
\]

This corresponds to a special case of our Jay ASTs:

\[\text{interpStmt}\ (\text{MkBloc}k\ [\text{stmt1}, \text{stmt2}]) = \]
Interpreter: Sequencing

\[
\sigma_1(s_1) \Rightarrow \sigma_2 \quad \sigma_2(s_2) \Rightarrow \sigma_3
\]

\[
\sigma_1(s_1 ; s_2) \Rightarrow \sigma_3
\]

This corresponds to a special case of our Jay ASTs:

\[
interpStmt \ (MkBloc k \ [stmt1, stmt2]) = \lambda \ s \rightarrow \text{case} \ (interpStmt \ stmt1) \ s \ of
\]
Interpreting: Sequencing

$$\sigma_1(s_1) \Rightarrow \sigma_2 \quad \frac{\sigma_2(s_2) \Rightarrow \sigma_3}{\sigma_1(s_1 ; s_2) \Rightarrow \sigma_3}$$

This corresponds to a special case of our Jay ASTs:

$\text{interpStmt} (\text{MkBloc}\left[\text{stmt1}, \text{stmt2}\right]) = \lambda \ s \rightarrow \text{case} \ (\text{interpStmt} \ \text{stmt1}) \ s \ \text{of} \ Just \ s_1 \rightarrow$
Interpreter: Sequencing

\[ \sigma_1(s_1) \Rightarrow \sigma_2 \quad \sigma_2(s_2) \Rightarrow \sigma_3 \]

\[ \sigma_1(s_1 \;;
\sigma_2(s_2) \Rightarrow \sigma_3 \]

This corresponds to a special case of our Jay ASTs:

\[
interpStmt (MkBloc k \ [ stmt1, stmt2 ]) = \lambda \ s \rightarrow \text{case} \ (interpStmt \ stmt1) \ s \ \text{of} \\
Just \ s1 \rightarrow (interpStmt \ stmt2) \ s1
\]
**Interpreter: Sequencing**

$$\sigma_1(s_1) \Rightarrow \sigma_2 \quad \quad \quad \sigma_2(s_2) \Rightarrow \sigma_3$$

$$\sigma_1(s_1; s_2) \Rightarrow \sigma_3$$

This corresponds to a special case of our Jay ASTs:

$$interpStmt \ (MkBloc k \ [\ stmt1, \ stmt2]) = \lambda \ s \rightarrow \text{case} \ (interpStmt \ stmt1) \ s \ \text{of}$$

$$\text{Just} \ s1 \rightarrow (interpStmt \ stmt2) \ s1$$

$$\text{Nothing} \rightarrow \text{Nothing}$$
Interpreter: Sequencing

\[
\sigma_1(s_1) \Rightarrow \sigma_2 \quad \sigma_2(s_2) \Rightarrow \sigma_3
\]

\[
\sigma_1(s_1; s_2) \Rightarrow \sigma_3
\]

This corresponds to a special case of our Jay ASTs:

\[
\text{interpStmt} \ (\text{MkBloc k} \ [\text{stmt1, stmt2}]) = \lambda \ s \rightarrow \textbf{case} \ (\text{interpStmt} \ \text{stmt1}) \ s \ \textbf{of}
\]

\[
\text{Just } s_1 \rightarrow (\text{interpStmt} \ \text{stmt2}) \ s_1
\]

\[
\text{Nothing} \rightarrow \text{Nothing}
\]

General case:

\[
\text{interpStmt} \ (\text{MkBloc k} \ \text{stmts}) = \lambda \ s \rightarrow \text{interpBlock} \ \text{stmts} \ s
\]
Interpreter: Sequencing

\[
\begin{align*}
\sigma_1(s_1) \Rightarrow \sigma_2 & \quad \sigma_2(s_2) \Rightarrow \sigma_3 \\
\therefore \quad \sigma_1(s_1 \; ; \; s_2) \Rightarrow \sigma_3
\end{align*}
\]

This corresponds to a special case of our Jay ASTs:

\[
\text{interpStmt} \; (\text{MkBloc}k \; [\; \text{stmt}1, \; \text{stmt}2 \; ]) = \lambda \; s \rightarrow \text{case} \; (\; \text{interpStmt} \; \text{stmt}1 \; ) \; s \; \text{of}
\]

\[
\text{Just} \; s_1 \rightarrow (\; \text{interpStmt} \; \text{stmt}2 \; ) \; s_1
\]

\[
\text{Nothing} \rightarrow \text{Nothing}
\]

General case:

\[
\text{interpStmt} \; (\text{MkBloc}k \; \text{stmts}) = \lambda \; s \rightarrow \text{interpBlock} \; \text{stmts} \; s
\]

\[
\text{interpBlock} :: [\; \text{Statement} \; ] \rightarrow (\; \text{State}1 \rightarrow \text{Maybe} \; \text{State}1 \; )
**Interpreter: Sequencing**

\[
\begin{align*}
\sigma_1(s_1) \Rightarrow \sigma_2 & \quad \sigma_2(s_2) \Rightarrow \sigma_3 \\
\sigma_1(s_1 \cdot s_2) & \Rightarrow \sigma_3
\end{align*}
\]

This corresponds to a special case of our Jay ASTs:

\[
\text{interpStmt} \ (\text{MkBloc}k \ [\ stmt1, \ stmt2]) = \lambda \ s \rightarrow \text{case} \ (\text{interpStmt} \ stmt1) \ s \ of \\
\text{Just} \ s1 \rightarrow (\text{interpStmt} \ stmt2) \ s1 \\
\text{Nothing} \rightarrow \text{Nothing}
\]

General case:

\[
\text{interpStmt} \ (\text{MkBloc}k \ \text{stmts}) = \lambda \ s \rightarrow \text{interpBlock} \ \text{stmts} \ s
\]

\[
\text{interpBlock} :: [\ \text{Statement}] \rightarrow (\text{State1} \rightarrow \text{Maybe State1})
\]

\[
\text{interpBlock} \ [\ ] = \]

Interpretation: Sequencing

\[
\begin{align*}
\sigma_1(s_1) & \Rightarrow \sigma_2 & \sigma_2(s_2) & \Rightarrow \sigma_3 \\
\hline
\sigma_1(s_1 ; s_2) & \Rightarrow \sigma_3
\end{align*}
\]

This corresponds to a special case of our Jay ASTs:

\[
interpStmt \ (MkBloc k \ [ \ stmt1, \ stmt2]) = \lambda \ s \rightarrow \text{case} \ (interpStmt \ stmt1) \ s \ \text{of}
\]
\[
\text{Just} \ s1 \rightarrow (interpStmt \ stmt2) \ s1 \\
\text{Nothing} \rightarrow \text{Nothing}
\]

General case:

\[
interpStmt \ (MkBloc k \ stmts) = \lambda \ s \rightarrow interpBlock \ stmts \ s
\]

\[
interpBlock :: [ \ Statement ] \rightarrow (\ State1 \rightarrow \text{Maybe} \ State1)
\]

\[
interpBlock \ [] = \text{Just}
\]
Interpreter: Sequencing

\[ \sigma_1(s_1) \Rightarrow \sigma_2 \quad \sigma_2(s_2) \Rightarrow \sigma_3 \]
\[ \sigma_1(s_1 ; s_2) \Rightarrow \sigma_3 \]

This corresponds to a special case of our Jay ASTs:

\[ \text{interpStmt} (\text{MkBloc k} [\text{stmt1, stmt2}]) = \lambda \ s \to \text{case} (\text{interpStmt} \ \text{stmt1}) \ s \ \text{of} \]
\[ \text{Just} \ s1 \to (\text{interpStmt} \ \text{stmt2}) \ s1 \]
\[ \text{Nothing} \to \text{Nothing} \]

General case:

\[ \text{interpStmt} (\text{MkBloc k} \ \text{stmts}) = \lambda \ s \to \text{interpBlock} \ \text{stmts} \ s \]

\[ \text{interpBlock} :: [\text{Statement}] \to (\text{State1} \to \text{Maybe State1}) \]
\[ \text{interpBlock} [\ ] = \text{Just} \]
\[ \text{interpBlock} (\text{stmt : stmts}) = \]
Interpreter: Sequencing

\[
\sigma_1(s_1) \Rightarrow \sigma_2 \quad \sigma_2(s_2) \Rightarrow \sigma_3 \\
\hline
\sigma_1(s_1 ; s_2) \Rightarrow \sigma_3
\]

This corresponds to a special case of our Jay ASTs:

\[
interpStmt (MkBloc k\ [\ stmt1,\ stmt2]) = \lambda\ s \rightarrow case\ (interpStmt\ stmt1)\ s\ of
Just\ s1 \rightarrow (interpStmt\ stmt2)\ s1
Nothing \rightarrow Nothing
\]

General case:

\[
interpStmt (MkBloc k\ stmts) = \lambda\ s \rightarrow interpBlock\ stmts\ s
\]

\[
interpBlock :: [\ Statement] \rightarrow (State1 \rightarrow Maybe\ State1)
interpBlock\ [\ ] = Just
interpBlock\ (stmt :\ stmts) = \lambda\ s \rightarrow
\]
Interpreter: Sequencing

\[
\sigma_1(s_1) \Rightarrow \sigma_2 \quad \sigma_2(s_2) \Rightarrow \sigma_3
\]

\[
\sigma_1(s_1 ; s_2) \Rightarrow \sigma_3
\]

This corresponds to a special case of our Jay ASTs:

\[
\text{interpStmt} ( \text{MkBloc}k [ \text{stmt1}, \text{stmt2} ]) = \lambda \ s \rightarrow \text{case} ( \text{interpStmt} \ \text{stmt1}) \ s \ \text{of}
\]

Just \( s1 \rightarrow ( \text{interpStmt} \ \text{stmt2}) \ s1 \\
Nothing \rightarrow Nothing

General case:

\[
\text{interpStmt} ( \text{MkBloc}k \ \text{stmts}) = \lambda \ s \rightarrow \text{interpBlock} \ \text{stmts} \ s
\]

\[
\text{interpBlock} :: [ \text{Statement} ] \rightarrow ( \text{State1} \rightarrow \text{Maybe State1})
\]

\[
\text{interpBlock} [ ] = \text{Just}
\]

\[
\text{interpBlock} ( \text{stmt} : \text{stmts}) = \lambda \ s \rightarrow \text{case} \ \text{interpStmt} \ \text{stmt} \ s \ \text{of}
\]

Interpreter: Sequencing

\[
\sigma_1(s_1) \Rightarrow \sigma_2 \quad \sigma_2(s_2) \Rightarrow \sigma_3 \\
\sigma_1(s_1; s_2) \Rightarrow \sigma_3
\]

This corresponds to a special case of our Jay ASTs:

\[
\text{interpStmt } (\text{MkBlock } [\text{stmt1}, \text{stmt2}]) = \lambda \ s \rightarrow \text{case } (\text{interpStmt } \text{stmt1}) \ s \text{ of} \\
\quad \text{Just } s1 \rightarrow (\text{interpStmt } \text{stmt2}) \ s1 \\
\quad \text{Nothing } \rightarrow \text{Nothing}
\]

General case:

\[
\text{interpStmt } (\text{MkBlock } \text{stmts}) = \lambda \ s \rightarrow \text{interpBlock } \text{stmts} \ s
\]

\[
\text{interpBlock } :: [\text{Statement}] \rightarrow (\text{State1} \rightarrow \text{Maybe State1}) \\
\text{interpBlock } [] = \text{Just} \\
\text{interpBlock } (\text{stmt} : \text{stmts}) = \lambda \ s \rightarrow \text{case } \text{interpStmt } \text{stmt} \ s \text{ of} \\
\quad \text{Just } s1 \rightarrow
\]
Interpreters: Sequencing

\[
\sigma_1(s_1) \Rightarrow \sigma_2 \quad \quad \quad \sigma_2(s_2) \Rightarrow \sigma_3
\]

\[
\sigma_1(s_1 ; s_2) \Rightarrow \sigma_3
\]

This corresponds to a special case of our Jay ASTs:

\[
interpStmt(MkBloc k[stmt1 , stmt2]) = \lambda \ s \rightarrow \text{case} ( interpStmt stmt1) \ s \ of
\]

\[
\text{Just } s1 \rightarrow (interpStmt stmt2) \ s1
\]

\[
\text{Nothing } \rightarrow \text{Nothing}
\]

General case:

\[
interpStmt(MkBloc k stmts) = \lambda \ s \rightarrow interpBlock stmts \ s
\]

\[
interpBlock : : [ \ Statement ] \rightarrow ( State1 \rightarrow \text{Maybe State1})
\]

\[
interpBlock [ ] = \text{Just}
\]

\[
interpBlock ( stmt : stmts) = \lambda \ s \rightarrow \text{case} interpStmt stmt \ s \ of
\]

\[
\text{Just } s1 \rightarrow interpBlock stmts \ s1
\]
Interpreter: Sequencing

\[ \sigma_1(s_1) \Rightarrow \sigma_2 \quad \sigma_2(s_2) \Rightarrow \sigma_3 \]

\[ \sigma_1(s_1 ; s_2) \Rightarrow \sigma_3 \]

This corresponds to a special case of our Jay ASTs:

\[ interpStmt (\text{MkBloc k} [\text{stmt1}, \text{stmt2}]) = \lambda \ s \rightarrow \text{case} (interpStmt\ \text{stmt1}) \ \text{of} \]

\[ \text{Just } s1 \rightarrow (interpStmt\ \text{stmt2})\ s1 \]

\[ \text{Nothing} \rightarrow \text{Nothing} \]

General case:

\[ interpStmt (\text{MkBloc k} \ \text{stmts}) = \lambda \ s \rightarrow interpBlock\ \text{stmts}\ s \]

\[ interpBlock :: [\text{Statement}] \rightarrow (\text{State1} \rightarrow \text{Maybe State1}) \]

\[ interpBlock\ [\ ] = \text{Just} \]

\[ interpBlock\ (\text{stmt} : \text{stmts}) = \lambda \ s \rightarrow \text{case} interpStmt\ \text{stmt}\ s \ \text{of} \]

\[ \text{Just } s1 \rightarrow interpBlock\ \text{stmts}\ s1 \]

\[ \text{Nothing} \rightarrow \text{Nothing} \]
**Interpreter: Loops**

\[
\begin{align*}
\sigma(b) & \Rightarrow \text{True} & \sigma(s) & \Rightarrow \sigma_1 & \sigma_1(\text{while } b \text{ do } s \text{ od}) & \Rightarrow \sigma_2 \\
\hline
\end{align*}
\]

\[
\sigma(\text{while } b \text{ do } s \text{ od}) \Rightarrow \sigma_2
\]

\[
\begin{align*}
\sigma(b) & \Rightarrow \text{False} \\
\hline
\sigma(\text{while } b \text{ do } s \text{ od}) & \Rightarrow \sigma
\end{align*}
\]
Interpreter: Loops

\[
\begin{align*}
\sigma(b) &\Rightarrow \text{True} & \sigma(s) &\Rightarrow \sigma_1 & \sigma_1(\text{while } b \text{ do } s \text{ od}) &\Rightarrow \sigma_2 \\
\sigma(\text{while } b \text{ do } s \text{ od}) &\Rightarrow \sigma_2 \\
\sigma(b) &\Rightarrow \text{False} & \sigma(\text{while } b \text{ do } s \text{ od}) &\Rightarrow \sigma
\end{align*}
\]

\text{interpStmt (Loop cond body) =}
Interpreter: Loops

\[
\sigma(b) \Rightarrow True \quad \sigma(s) \Rightarrow \sigma_1 \quad \sigma_1(\textbf{while} \ b \ \textbf{do} \ s \ \textbf{od}) \Rightarrow \sigma_2
\]

\[
\sigma(\textbf{while} \ b \ \textbf{do} \ s \ \textbf{od}) \Rightarrow \sigma_2
\]

\[
\sigma(b) \Rightarrow False
\]

\[
\sigma(\textbf{while} \ b \ \textbf{do} \ s \ \textbf{od}) \Rightarrow \sigma
\]

\[
\text{interpStmt} \ (\textit{Loop} \ \textit{cond} \ \textit{body}) = \lambda \ s \rightarrow \textbf{case} \ (\text{evalExpr} \ \textit{cond}) \ s \ \textbf{of}
\]
Interpreter: Loops

\[
\begin{align*}
\sigma(b) & \Rightarrow \text{True} & \sigma(s) & \Rightarrow \sigma_1 & \sigma_1(\text{while } b \text{ do } s \text{ od}) & \Rightarrow \sigma_2 \\
\sigma(\text{while } b \text{ do } s \text{ od}) & \Rightarrow \sigma_2 \\
\sigma(b) & \Rightarrow \text{False} & \sigma(\text{while } b \text{ do } s \text{ od}) & \Rightarrow \sigma
\end{align*}
\]

\[
\text{interpStmt} \ (\text{Loop cond body}) = \lambda \ s \rightarrow \text{case} \ (\text{evalExpr cond}) \ s \ of
\]
\[
\text{Just} \ (\text{ValBool False}) \rightarrow \\
\text{Just} \ (\text{ValBool True}) \rightarrow
\]

\[
\text{interpStmt} \ (\text{Loop cond body}) = \lambda \ s \rightarrow \text{case} \ (\text{evalExpr cond}) \ s \ of
\]
\[
\text{Just} \ (\text{ValBool False}) \rightarrow \\
\text{Just} \ (\text{ValBool True}) \rightarrow
\]
**Interpreter: Loops**

\[
\sigma(b) \Rightarrow \text{True} \quad \sigma(s) \Rightarrow \sigma_1 \quad \sigma_1(\text{while } b \text{ do } s \text{ od}) \Rightarrow \sigma_2 \\
\sigma(\text{while } b \text{ do } s \text{ od}) \Rightarrow \sigma_2
\]

\[
\sigma(b) \Rightarrow \text{False} \\
\sigma(\text{while } b \text{ do } s \text{ od}) \Rightarrow \sigma
\]

\[
\text{interpStmt} (\text{Loop cond body}) = \lambda s \rightarrow \text{case} (\text{evalExpr } \text{cond}) \text{ s of} \\
\text{Just (ValBool False)} \rightarrow \text{Just } s \\
\text{Just (ValBool True)} \rightarrow
\]


**Interpreter: Loops**

\[
\begin{align*}
\sigma(b) & \Rightarrow \text{True} & \sigma(s) & \Rightarrow \sigma_1 & \sigma_1(\text{while } b \text{ do } s \text{ od}) & \Rightarrow \sigma_2 \\
\sigma(\text{while } b \text{ do } s \text{ od}) & \Rightarrow \sigma_2 \\
\sigma(b) & \Rightarrow \text{False} & \sigma(\text{while } b \text{ do } s \text{ od}) & \Rightarrow \sigma
\end{align*}
\]

\[
\text{interpStmt (Loop cond body)} = \lambda \ s \rightarrow \text{case} (\text{evalExpr cond}) \ s \ of
\]

\[
\begin{align*}
\text{Just (ValBool False)} & \rightarrow \text{Just } s \\
\text{Just (ValBool True)} & \rightarrow \text{case} (\text{interpStmt body}) \ s \ of
\]

\[
\begin{align*}
\text{Just } s_1 & \rightarrow
\end{align*}
\]
**Interpreter: Loops**

\[
\sigma(b) \Rightarrow \text{True} \quad \quad \quad \quad \quad \sigma(s) \Rightarrow \sigma_1 \quad \quad \quad \quad \quad \sigma_1(\text{while } b \text{ do } s \text{ od}) \Rightarrow \sigma_2 \\
\sigma(\text{while } b \text{ do } s \text{ od}) \Rightarrow \sigma_2
\]

\[
\sigma(b) \Rightarrow \text{False} \\
\sigma(\text{while } b \text{ do } s \text{ od}) \Rightarrow \sigma
\]

\[
\text{interpStmt } (\text{Loop cond body}) = \lambda s \rightarrow \text{case } (\text{evalExpr cond}) \text{ of} \\
\text{Just (ValBool False)} \rightarrow \text{Just } s \\
\text{Just (ValBool True)} \rightarrow \text{case } (\text{interpStmt body}) \text{ of} \\
\text{Just } s_1 \rightarrow (\text{interpStmt } (\text{Loop cond body})) \text{ s1}
\]
Interpreter: Loops

\[
\begin{align*}
\sigma(b) \Rightarrow \text{True} & \quad \sigma(s) \Rightarrow \sigma_1 & \quad \sigma_1(\text{while } b \text{ do } s \text{ od}) \Rightarrow \sigma_2 \\
\hline
\sigma(\text{while } b \text{ do } s \text{ od}) \Rightarrow \sigma_2 & \quad \sigma(b) \Rightarrow \text{False} \\
& \quad \sigma_1(\text{while } b \text{ do } s \text{ od}) \Rightarrow \sigma \\
\end{align*}
\]

\[\text{interpStmt (Loop cond body)} = \lambda s \to \text{case (evalExpr cond) s of} \]
\[\text{Just (ValBool False)} \to \text{Just s} \]
\[\text{Just (ValBool True)} \to \text{case (interpStmt body) s of} \]
\[\text{Just } s_1 \to (\text{interpStmt (Loop cond body)})) s_1 \]
\[\text{Nothing } \to \text{Nothing} \]
Interpreter: Loops

\[
\begin{align*}
\sigma(b) & \Rightarrow \text{True} & \sigma(s) & \Rightarrow \sigma_1 & \sigma_1(\text{while } b \text{ do } s \text{ od}) & \Rightarrow \sigma_2 \\
\sigma(\text{while } b \text{ do } s \text{ od}) & \Rightarrow \sigma_2 \\
\sigma(b) & \Rightarrow \text{False} & \sigma(\text{while } b \text{ do } s \text{ od}) & \Rightarrow \sigma
\end{align*}
\]

\[
\text{interpStmt } (\text{Loop } \text{cond body}) = \lambda \ s \rightarrow \text{case } (\text{evalExpr } \text{cond}) \ s \ of
\]
\[
\begin{align*}
\text{Just } (\text{ValBool False}) & \rightarrow \text{Just } s \\
\text{Just } (\text{ValBool True}) & \rightarrow \text{case } (\text{interpStmt } \text{body}) \ s \ of \\
\text{Just } s1 & \rightarrow (\text{interpStmt } (\text{Loop } \text{cond body})) \ s1 \\
\text{Nothing} & \rightarrow \text{Nothing}
\end{align*}
\]
\[
\text{Just } (\text{ValInt } i) \rightarrow \text{Nothing} \\
\text{Nothing} \rightarrow \text{Nothing}
\]
**Interpreter: Loops**

\[
\begin{align*}
\sigma(b) \Rightarrow \text{True} & \quad \sigma(s) \Rightarrow \sigma_1 & \quad \sigma_1(\text{while } b \text{ do } s \text{ od}) \Rightarrow \sigma_2 \\
\hline
\sigma(\text{while } b \text{ do } s \text{ od}) \Rightarrow \sigma_2 \\
\sigma(b) \Rightarrow \text{False} & \quad \sigma(\text{while } b \text{ do } s \text{ od}) \Rightarrow \sigma
\end{align*}
\]

\[\text{interpStmt (Loop cond body)} = \lambda \, s \to \text{case (evalExpr cond) s of}
\]

\[\text{Just (ValBool False)} \to \text{Just s}
\]

\[\text{Just (ValBool True)} \to \text{case (interpStmt body) s of}
\]

\[\text{Just s1} \to \text{case (interpStmt (Loop cond body)) s1}
\]

\[\text{Nothing} \to \text{Nothing}
\]

\[\text{Just (ValInt i)} \to \text{Nothing}
\]

\[\text{Nothing} \to \text{Nothing}
\]

This is **not compositional**
Interpreter: Loops

\[
\begin{align*}
\sigma(b) & \Rightarrow \text{True} & \sigma(s) & \Rightarrow \sigma_1 & \sigma_1(\text{while } b \text{ do } s \text{ od}) & \Rightarrow \sigma_2 \\
\sigma(\text{while } b \text{ do } s \text{ od}) & \Rightarrow \sigma_2 \\
\sigma(b) & \Rightarrow \text{False} & \sigma(\text{while } b \text{ do } s \text{ od}) & \Rightarrow \sigma
\end{align*}
\]

\[
\text{interpStmt}(\text{Loop cond body}) = \lambda \; s \rightarrow \text{case}(\text{evalExpr cond}) \; s \; \text{of}
\]

\[
\text{Just}(\text{ValBool False}) \rightarrow \text{Just} \; s
\]

\[
\text{Just}(\text{ValBool True}) \rightarrow \text{case}(\text{interpStmt body}) \; s \; \text{of}
\]

\[
\text{Just} \; s1 \rightarrow (\text{interpStmt}(\text{Loop cond body})) \; s1
\]

\[
\text{Nothing} \rightarrow \text{Nothing}
\]

\[
\text{Just}(\text{ValInt i}) \rightarrow \text{Nothing}
\]

\[
\text{Nothing} \rightarrow \text{Nothing}
\]

This is \textbf{not compositional}, but \textbf{recursive}
**Interpreters: Loops**

\[ \sigma(b) \Rightarrow \text{True} \quad \sigma(s) \Rightarrow \sigma_1 \quad \sigma_1(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2 \]

\[ \sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2 \]

\[ \sigma(b) \Rightarrow \text{False} \]

\[ \sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma \]

\[
\text{interpStmt (Loop cond body)} = \lambda s \rightarrow \text{case (evalExpr cond) of}
\]

\[
\text{Just (ValBool False)} \rightarrow \text{Just } s
\]

\[
\text{Just (ValBool True)} \rightarrow \text{case (interpStmt body) of}
\]

\[
\text{Just } s1 \rightarrow (\text{interpStmt (Loop cond body)}) \, s1
\]

\[
\text{Nothing} \rightarrow \text{Nothing}
\]

\[
\text{Just (ValInt i)} \rightarrow \text{Nothing}
\]

\[
\text{Nothing} \rightarrow \text{Nothing}
\]

This is **not compositional**, but **recursive:**

“\text{interpStmt (Loop cond body)}”
Interpreter: Loops

\[ \sigma(b) \Rightarrow \text{True} \quad \sigma(s) \Rightarrow \sigma_1 \quad \sigma_1(\text{while } b \text{ do } s \text{ od}) \Rightarrow \sigma_2 \]

\[ \sigma(\text{while } b \text{ do } s \text{ od}) \Rightarrow \sigma_2 \]

\[ \sigma(b) \Rightarrow \text{False} \]

\[ \sigma(\text{while } b \text{ do } s \text{ od}) \Rightarrow \sigma \]

\[ \text{interpStmt}(\text{Loop cond body}) = \lambda \ s \rightarrow \text{case}(\text{evalExpr cond}) \ s \text{ of} \]

\[ \text{Just}(\text{ValBool False}) \rightarrow \text{Just } s \]

\[ \text{Just}(\text{ValBool True}) \rightarrow \text{case}(\text{interpStmt body}) \ s \text{ of} \]

\[ \text{Just } s1 \rightarrow (\text{interpStmt}(\text{Loop cond body})) \ s1 \]

\[ \text{Nothing} \rightarrow \text{Nothing} \]

\[ \text{Just}(\text{ValInt } i) \rightarrow \text{Nothing} \]

\[ \text{Nothing} \rightarrow \text{Nothing} \]

This is \textbf{not compositional}, but \textbf{recursive}:

"\text{interpStmt}(\text{Loop cond body})" occurs also on the right-hand side."
Recursive *Definitions*?
Recursive Definitions?

Translating the Haskell definition back into the mathematical notation we obtain something of the following shape:

\[
\left[ \texttt{while } b \texttt{ do } s \texttt{ od} \right]_S = F( \left[ \texttt{while } b \texttt{ do } s \texttt{ od} \right]_S )
\]
Recursive Definitions?

Translating the Haskell definition back into the mathematical notation we obtain something of the following shape:

\[
\left[\text{while } b \text{ do } s \text{ od}\right]_S = F\left(\left[\text{while } b \text{ do } s \text{ od}\right]_S\right)
\]

In mathematics:
Recursive Definitions?

Translating the Haskell definition back into the mathematical notation we obtain something of the following shape:

$$[[\text{while } b \text{ do } s \text{ od}]]_S = F( [[\text{while } b \text{ do } s \text{ od}]]_S )$$

In mathematics:

- “$$x = \langle \langle \text{term not containing } x \rangle \rangle$$”
Recursive Definitions?

Translating the Haskell definition back into the mathematical notation we obtain something of the following shape:

\[
\llbracket \text{while } b \text{ do } s \text{ od} \rrbracket_S = F( \llbracket \text{while } b \text{ do } s \text{ od} \rrbracket_S )
\]

In mathematics:

- “\( x = \langle \langle \text{term not containing } x \rangle \rangle \)” is an explicit definition of \( x \)
Recursive Definitions?

Translating the Haskell definition back into the mathematical notation we obtain something of the following shape:

\[
\llbracket \text{while } b \text{ do } s \text{ od} \rrbracket_S = F( \llbracket \text{while } b \text{ do } s \text{ od} \rrbracket_S )
\]

In mathematics:

- “\(x = \langle \langle \text{term not containing } x \rangle \rangle\)” is an explicit definition of \(x\)
- “\(x = F(x)\)”
Recursive Definitions?

Translating the Haskell definition back into the mathematical notation we obtain something of the following shape:

$$\llbracket \text{while } b \text{ do } s \text{ od} \rrbracket_S = F( \llbracket \text{while } b \text{ do } s \text{ od} \rrbracket_S )$$

In mathematics:

- “\( x = \langle \langle \text{term not containing } x \rangle \rangle \)” is an explicit definition of \( x \)
- “\( x = F(x) \)” is an equation
Recursive Definitions?

Translating the Haskell definition back into the mathematical notation we obtain something of the following shape:

\[
\llbracket \text{while } b \text{ do } s \text{ od} \rrbracket_S = F( \llbracket \text{while } b \text{ do } s \text{ od} \rrbracket_S )
\]

In mathematics:

- “\(x = \llangle \text{term not containing } x \rrangle\)” is an explicit definition of \(x\)
- “\(x = F(x)\)” is an equation that may have many solutions!
Recursive Definitions?

Translating the Haskell definition back into the mathematical notation we obtain something of the following shape:

\[ \text{while } b \text{ do } s \text{ od} \] 

\[ F(\text{while } b \text{ do } s \text{ od}) \]

In mathematics:

- “\( x = \langle \langle \text{term not containing } x \rangle \rangle \)’’ is an **explicit definition** of \( x \)
- “\( x = F(x) \)’’ is an **equation** that may have many solutions!
- “\( x = F(x) \)’’ can be used as **implicit definition** of \( x \)
Recursive Definitions?

Translating the Haskell definition back into the mathematical notation we obtain something of the following shape:

\[
[[\text{while } b \text{ do } s \text{ od}]_S] = F( [[\text{while } b \text{ do } s \text{ od}]_S])
\]

In mathematics:

- “\(x = \langle\langle\text{term not containing } x\rangle\rangle\)” is an explicit definition of \(x\)
- “\(x = F(x)\)” is an equation that may have many solutions!
- “\(x = F(x)\)” can be used as implicit definition of \(x\) only if the equation is known to have exactly one solution!
Recursive Definitions?

Translating the Haskell definition back into the mathematical notation we obtain something of the following shape:

\[
\left[\text{while } b \text{ do } s \text{ od}\right]_S = F\left( \left[\text{while } b \text{ do } s \text{ od}\right]_S \right)
\]

In mathematics:

• “\(x = \langle \langle \text{term not containing } x \rangle \rangle\)” is an explicit definition of \(x\)
• “\(x = F(x)\)” is an equation that may have many solutions!
• “\(x = F(x)\)” can be used as implicit definition of \(x\) only if the equation is known to have exactly one solution! (I.e., writing such a definition produces a proof obligation of well-definedness)
Recursive Definitions?

Translating the Haskell definition back into the mathematical notation we obtain something of the following shape:

\[
\text{while } b \text{ do } s \text{ od} \]_S = F( \text{while } b \text{ do } s \text{ od} )_S

In mathematics:

- “\(x = \langle \text{term not containing } x \rangle \)’’ is an explicit definition of \(x\)
- “\(x = F(x)\)’’ is an equation that may have many solutions!
- “\(x = F(x)\)’’ can be used as implicit definition of \(x\) only if the equation is known to have exactly one solution! (I.e., writing such a definition produces a proof obligation of well-definedness)

In denotational semantics:

“\(\text{while } b \text{ do } s \text{ od} \]_S = F( \text{while } b \text{ do } s \text{ od} )_S ”
Recursive Definitions?

Translating the Haskell definition back into the mathematical notation we obtain something of the following shape:

\[
\llbracket \text{while } b \text{ do } s \text{ od} \rrbracket_S = F( \llbracket \text{while } b \text{ do } s \text{ od} \rrbracket_S )
\]

In mathematics:

- “\(x = \langle \langle \text{term not containing } x \rangle \rangle\)” is an explicit definition of \(x\).
- “\(x = F(x)\)” is an equation that may have many solutions!
- “\(x = F(x)\)” can be used as implicit definition of \(x\) only if the equation is known to have exactly one solution! (I.e., writing such a definition produces a proof obligation of well-definedness)

In denotational semantics:

“\(\llbracket \text{while } b \text{ do } s \text{ od} \rrbracket_S = F( \llbracket \text{while } b \text{ do } s \text{ od} \rrbracket_S )\)” considered as equation in “\(\llbracket \text{while } b \text{ do } s \text{ od} \rrbracket_S\)”
Recursive Definitions?

Translating the Haskell definition back into the mathematical notation we obtain something of the following shape:

\[
\llbracket \text{while } b \text{ do } s \text{ od} \rrbracket_S = F( \llbracket \text{while } b \text{ do } s \text{ od} \rrbracket_S )
\]

In mathematics:

- “\(x = \langle\langle \text{term not containing } x \rangle\rangle\)” is an explicit definition of \(x\)
- “\(x = F(x)\)” is an equation that may have many solutions!
- “\(x = F(x)\)” can be used as implicit definition of \(x\) only if the equation is known to have exactly one solution! (I.e., writing such a definition produces a proof obligation of well-definedness)

In denotational semantics:

- “\(\llbracket \text{while } b \text{ do } s \text{ od} \rrbracket_S = F( \llbracket \text{while } b \text{ do } s \text{ od} \rrbracket_S )\)” considered as equation in
- “\(\llbracket \text{while } b \text{ do } s \text{ od} \rrbracket_S\)” usually has many solutions!
Recursive Function Definitions

How to convert a recursive function definition into an explicit definition?
Recursive Function Definitions

How to convert a recursive function definition into an explicit definition?

Start (Haskell):

\[
\text{fact } n = \text{ if } n \equiv 0 \text{ then 1 else } n * \text{ fact } (n-1)
\]
Recursive Function Definitions

How to convert a recursive function definition into an explicit definition?

Start (Haskell):

\[ \text{fact } n = \text{if } n \equiv 0 \text{ then 1 else } n \times \text{fact } (n-1) \]

Principle of \textbf{extensionality}: two functions are equal iff all their resp. applications to the same argument are equal:
Recursive Function Definitions

How to convert a recursive function definition into an explicit definition?

Start (Haskell):

$$fact \ n = \ \text{if} \ n \equiv 0 \ \text{then} \ 1 \ \text{else} \ n \times fact \ (n-1)$$

Principle of extensionality: two functions are equal iff all their resp. applications to the same argument are equal:

$$fact = \lambda \ n \rightarrow \text{if} \ n \equiv 0 \ \text{then} \ 1 \ \text{else} \ n \times fact \ (n-1)$$
Recursive Function Definitions

How to convert a recursive function definition into an explicit definition?

Start (Haskell):

\[
fact \ n = \ \text{if} \ n \equiv 0 \ \text{then} \ 1 \ \text{else} \ n \ast fact \ (n-1)
\]

Principle of extensionality: two functions are equal iff all their resp. applications to the same argument are equal:

\[
fact = \lambda \ n \rightarrow \ \text{if} \ n \equiv 0 \ \text{then} \ 1 \ \text{else} \ n \ast fact \ (n-1)
\]

Reverse \(\beta\)-reduction to isolate the RHS occurrence of \(fact\):

\[
fact = (\lambda \ f \rightarrow \lambda \ n \rightarrow \ \text{if} \ n \equiv 0 \ \text{then} \ 1 \ \text{else} \ n \ast f \ (n-1)) \ fact
\]
Recursive Function Definitions

How to convert a recursive function definition into an explicit definition?

Start (Haskell):

\[
\text{fact } n = \text{if } n \equiv 0 \text{ then 1 else } n \ast \text{fact } (n-1)
\]

Principle of extensionality: two functions are equal iff all their resp. applications to the same argument are equal:

\[
\text{fact} = \lambda n \rightarrow \text{if } n \equiv 0 \text{ then 1 else } n \ast \text{fact } (n-1)
\]

Reverse \(\beta\)-reduction to isolate the RHS occurrence of \(\text{fact}\):

\[
\text{fact} = (\lambda f \rightarrow \lambda n \rightarrow \text{if } n \equiv 0 \text{ then 1 else } n \ast f (n-1)) \text{ fact}
\]

Defining (via an explicit definition)

\[
\tau = \lambda f \rightarrow \lambda n \rightarrow \text{if } n \equiv 0 \text{ then 1 else } n \ast f (n-1)
\]
Recursive Function Definitions

How to convert a recursive function definition into an explicit definition?

Start (Haskell):

\[
\text{fact } n = \text{if } n \equiv 0 \text{ then } 1 \text{ else } n \ast \text{fact } (n-1)
\]

Principle of \textit{extensionality}: two functions are equal iff all their resp. applications to the same argument are equal:

\[
\text{fact} = \lambda \ n \rightarrow \text{if } n \equiv 0 \text{ then } 1 \text{ else } n \ast \text{fact } (n-1)
\]

Reverse \(\beta\)-reduction to isolate the RHS occurrence of \textit{fact}:

\[
\text{fact} = (\lambda \ f \rightarrow \lambda \ n \rightarrow \text{if } n \equiv 0 \text{ then } 1 \text{ else } n \ast f (n-1)) \text{fact}
\]

Defining (via an explicit definition)

\[
\tau = \lambda \ f \rightarrow \lambda \ n \rightarrow \text{if } n \equiv 0 \text{ then } 1 \text{ else } n \ast f (n-1)
\]

we recognise a \textit{fixedpoint equation} (stating that \textit{fact} is a fixedpoint of \(\tau\)):

\[
\text{fact} = \tau \text{ fact}
\]
Fixedpoint Approximation

For the functional $\tau$ associated with the definition of the factorial function $\text{fact}$, we observe:

$$\tau^0 \perp = \perp = \{\}$$
Fixedpoint Approximation

For the functional $\tau$ associated with the definition of the factorial function $\text{fact}$, we observe:

$$\tau^0 \perp = \perp = \emptyset$$
$$\tau^1 \perp = \tau \perp = \{0 \mapsto 1\}$$
Fixedpoint Approximation

For the functional $\tau$ associated with the definition of the factorial function $\text{fact}$, we observe:

$$
\begin{align*}
\tau^0 \perp &= \perp &= \emptyset \\
\tau^1 \perp &= \tau \perp &= \{0 \mapsto 1\} \\
\tau^2 \perp &= \tau (\tau \perp) &= \{0 \mapsto 1, 1 \mapsto 1\}
\end{align*}
$$
Fixedpoint Approximation

For the functional $\tau$ associated with the definition of the factorial function $\text{fact}$, we observe:

\[
\begin{align*}
\tau^0 \perp &= \perp &= \emptyset \\
\tau^1 \perp &= \tau \perp &= \{0\mapsto 1\} \\
\tau^2 \perp &= \tau (\tau \perp) &= \{0\mapsto 1, 1\mapsto 1\} \\
\tau^3 \perp &= \tau (\tau (\tau \perp)) &= \{0\mapsto 1, 1\mapsto 1, 2\mapsto 2\}
\end{align*}
\]
Fixedpoint Approximation

For the functional $\tau$ associated with the definition of the factorial function $\text{fact}$, we observe:

\[
\begin{align*}
\tau^0 & \perp = \perp = \emptyset \\
\tau^1 & \perp = \tau \perp = \{0\mapsto 1\} \\
\tau^2 & \perp = \tau (\tau \perp ) = \{0\mapsto 1, 1\mapsto 1\} \\
\tau^3 & \perp = \tau (\tau (\tau \perp )) = \{0\mapsto 1, 1\mapsto 1, 2\mapsto 2\} \\
\tau^4 & \perp = \tau (\tau (\tau (\tau \perp ))) = \{0\mapsto 1, 1\mapsto 1, 2\mapsto 2, 3\mapsto 6\}
\end{align*}
\]
Fixedpoint Approximation

For the functional $\tau$ associated with the definition of the factorial function $\text{fact}$, we observe:

\[
\begin{array}{ll}
\tau^0 \perp = \perp = \emptyset \\
\tau^1 \perp = \tau \perp = \{0\mapsto 1\} \\
\tau^2 \perp = \tau(\tau \perp) = \{0\mapsto 1, 1\mapsto 1\} \\
\tau^3 \perp = \tau(\tau(\tau \perp)) = \{0\mapsto 1, 1\mapsto 1, 2\mapsto 2\} \\
\tau^4 \perp = \tau(\tau(\tau(\tau \perp))) = \{0\mapsto 1, 1\mapsto 1, 2\mapsto 2, 3\mapsto 6\} \\
\tau^5 \perp = \tau(\tau(\tau(\tau(\tau \perp)))) = \{0\mapsto 1, 1\mapsto 1, 2\mapsto 2, 3\mapsto 6, 4\mapsto 24\}
\end{array}
\]
Fixedpoint Approximation

For the functional $\tau$ associated with the definition of the factorial function $\text{fact}$, we observe:

\[
\begin{align*}
\tau^0 \perp &= \perp &= \{\} \\
\tau^1 \perp &= \tau \perp &= \{0\mapsto1\} \\
\tau^2 \perp &= \tau (\tau \perp) &= \{0\mapsto1, 1\mapsto1\} \\
\tau^3 \perp &= \tau (\tau (\tau \perp)) &= \{0\mapsto1, 1\mapsto1, 2\mapsto2\} \\
\tau^4 \perp &= \tau (\tau (\tau (\tau \perp))) &= \{0\mapsto1, 1\mapsto1, 2\mapsto2, 3\mapsto6\} \\
\tau^5 \perp &= \tau (\tau (\tau (\tau (\tau \perp)))) &= \{0\mapsto1, 1\mapsto1, 2\mapsto2, 3\mapsto6, 4\mapsto24\}
\end{align*}
\]

In addition:

\[
\perp \subseteq \tau \subseteq \tau^2 \subseteq \tau^3 \subseteq \tau^4 \subseteq \tau^5 \subseteq \ldots \subseteq \tau^{1,000,000} \subseteq \ldots
\]
Fixedpoint Approximation

For the functional \( \tau \) associated with the definition of the factorial function \( \text{fact} \), we observe:

\[
\begin{align*}
\tau^0 \bot &= \bot &= \{\} \\
\tau^1 \bot &= \tau \bot &= \{0\mapsto1\} \\
\tau^2 \bot &= \tau (\tau \bot ) &= \{0\mapsto1, 1\mapsto1\} \\
\tau^3 \bot &= \tau (\tau (\tau \bot )) &= \{0\mapsto1, 1\mapsto1, 2\mapsto2\} \\
\tau^4 \bot &= \tau (\tau (\tau (\tau \bot )))))) &= \{0\mapsto1, 1\mapsto1, 2\mapsto2, 3\mapsto6\} \\
\tau^5 \bot &= \tau (\tau (\tau (\tau (\tau \bot )))))) &= \{0\mapsto1, 1\mapsto1, 2\mapsto2, 3\mapsto6, 4\mapsto24\}
\end{align*}
\]

In addition:

\[
\bot \subseteq \tau \subseteq \tau^2 \subseteq \tau^3 \subseteq \tau^4 \subseteq \tau^5 \subseteq \ldots \subseteq \tau^{1,000,000} \subseteq \ldots
\]

Iterated application of \( \tau \) yields better and better \textbf{finite approximations}!
Fixedpoint Approximation

For the functional $\tau$ associated with the definition of the factorial function $\text{fact}$, we observe:

\[
\begin{align*}
\tau^0 & \downarrow = \downarrow = \{\} \\
\tau^1 & \downarrow = \tau \downarrow = \{0 \mapsto 1\} \\
\tau^2 & \downarrow = \tau (\tau \downarrow) = \{0 \mapsto 1, 1 \mapsto 1\} \\
\tau^3 & \downarrow = \tau (\tau (\tau \downarrow)) = \{0 \mapsto 1, 1 \mapsto 1, 2 \mapsto 2\} \\
\tau^4 & \downarrow = \tau (\tau (\tau (\tau \downarrow))) = \{0 \mapsto 1, 1 \mapsto 1, 2 \mapsto 2, 3 \mapsto 6\} \\
\tau^5 & \downarrow = \tau (\tau (\tau (\tau (\tau \downarrow)))) = \{0 \mapsto 1, 1 \mapsto 1, 2 \mapsto 2, 3 \mapsto 6, 4 \mapsto 24\}
\end{align*}
\]

In addition:

\[
\downarrow \subseteq \tau \downarrow \subseteq \tau^2 \downarrow \subseteq \tau^3 \downarrow \subseteq \tau^4 \downarrow \subseteq \tau^5 \downarrow \subseteq \ldots \subseteq \tau^{1,000,000} \downarrow \subseteq \ldots
\]

Iterated application of $\tau$ yields better and better finite approximations!

The union of all these approximations is the factorial function.
For partial functions, the least upper bound of an ascending chain is given by set-theoretic union over all elements of the chain.
Fixedpoint Semantics for Recursion

For partial functions, the least upper bound of an ascending chain is given by set-theoretic union over all elements of the chain.
Fixedpoint Semantics for Recursion

For partial functions, the least upper bound of an ascending chain is given by set-theoretic union over all elements of the chain.

Semantics for a recursive function $f$ is arrived at as follows:
Fixedpoint Semantics for Recursion

For partial functions, the least upper bound of an ascending chain is given by set-theoretic union over all elements of the chain.

**Semantics for a recursive function** $f$ is arrived at as follows:

- The recursive definition is transformed into a **fixedpoint equation** $f = \tau f$. 
Fixedpoint Semantics for Recursion

For partial functions, the least upper bound of an ascending chain is given by set-theoretic union over all elements of the chain.

Semantics for a recursive function $f$ is arrived at as follows:

- The recursive definition is transformed into a fixedpoint equation $f = \tau f$.
- The “functional” $\tau$ is extracted from that equation.
Fixedpoint Semantics for Recursion

For partial functions, the least upper bound of an ascending chain is given by set-theoretic union over all elements of the chain.

Semantics for a recursive function $f$ is arrived at as follows:

- The recursive definition is transformed into a fixedpoint equation $f = \tau f$.
- The “functional” $\tau$ is extracted from that equation.
- Use $\tau$ for fixedpoint iteration
  \[
  \bot \sqsubseteq \tau \sqsubseteq \tau^2 \sqsubseteq \tau^3 \sqsubseteq \ldots
  \]
Fixedpoint Semantics for Recursion

For partial functions, the least upper bound of an ascending chain is given by set-theoretic union over all elements of the chain.

Semantics for a recursive function $f$ is arrived at as follows:

- The recursive definition is transformed into a fixedpoint equation $f = \tau f$.

- The “functional” $\tau$ is extracted from that equation.

- Use $\tau$ for fixedpoint iteration

  $\bot \sqsubseteq \tau \bot \sqsubseteq \tau^2 \bot \sqsubseteq \tau^3 \bot \sqsubseteq \ldots$

- The semantics of $f$ is the least upper bound of this chain:

  $$[[f]] = \bigcup\{k : \mathbb{N} \bullet \tau^k \bot \}$$
Fixedpoint Semantics for Recursion

For partial functions, the least upper bound of an ascending chain is given by set-theoretic union over all elements of the chain.

**Semantics for a recursive function** $f$ is arrived at as follows:

- The recursive definition is transformed into a **fixedpoint equation** $f = \tau f$.
- The “functional” $\tau$ is extracted from that equation.
- Use $\tau$ for **fixedpoint iteration**
  \begin{align*}
  \bot & \subseteq \tau \ \\
  \bot & \subseteq \tau^2 \ \\
  \bot & \subseteq \tau^3 \ \\
  \vdots &
  \end{align*}
- The semantics of $f$ is the least upper bound of this chain:
  \[
  \llbracket f \rrbracket = \bigcup \{ k : \mathbb{N} \bullet \tau^k \bot \}
  \]
- This least upper bound is the **least fixedpoint** of $\tau$
**Fixedpoint Semantics for Recursion**

For partial functions, the least upper bound of an ascending chain is given by set-theoretic union over all elements of the chain.

**Semantics for a recursive function** $f$ is arrived at as follows:

- The recursive definition is transformed into a **fixedpoint equation** $f = \tau f$.
- The “functional” $\tau$ is extracted from that equation.
- Use $\tau$ for **fixedpoint iteration**
  
  \[
  \bot \subseteq \tau \subseteq \tau^2 \subseteq \tau^3 \subseteq \ldots
  \]
- The semantics of $f$ is the least upper bound of this chain:
  
  \[
  [[f]] = \bigcup\{k : \mathbb{N} \cdot \tau^k \subseteq \bot\}
  \]
- This least upper bound is the **least fixedpoint** of $\tau$

- We write “$Y \tau$” for the least fixedpoint of $\tau$. 
while-Loop Semantics

\[
[[\_]]_S : Stmt \rightarrow (State \rightarrow State)
\]

For \(p : Stmt\) and \(e : Expr\):

\[
[[\text{while } e \text{ do } p]]_S
\]

\[
\begin{align*}
= & \text{Y} \left( \lambda f : State \rightarrow State \cdot \lambda s : State \cdot \begin{cases} 
  f([[p]]_S(s)) & \text{if } [[e]]_E(s) = \text{True} \\
  s & \text{if } [[e]]_E(s) = \text{False} \\
  \bot & \text{otherwise}
\end{cases} \right)
\end{align*}
\]
Example Statement Semantics

\[[\text{while True do skip}]_S =\]
Example Statement Semantics

\[
[[\text{while True do skip}]]_S = Y(\lambda f : State \rightarrow State \bullet \lambda s : State \bullet f([[\text{skip}]]_S(s)))
\]
Example Statement Semantics

$$[[\textbf{while True do skip}]_S = Y(\lambda f : State \leftrightarrow State \bullet \lambda s : State \bullet f([\textbf{skip}]_S (s)))$$

$$= Y(\lambda f : State \leftrightarrow State \bullet \lambda s : State \bullet f(s))$$
Example Statement Semantics

$$[[\text{while True do skip}]_S] = Y(\lambda f : State \rightarrow State \cdot \lambda s : State \cdot f([[\text{skip}]_S(s))$$

$$= Y(\lambda f : State \rightarrow State \cdot \lambda s : State \cdot f(s))$$

$$= Y(\lambda f : State \rightarrow State \cdot f)$$
Example Statement Semantics

\[ [[\textbf{while} \ True \ \textbf{do} \ \textbf{skip}]_S] = Y(\lambda f : \text{State} \rightarrow \text{State} \bullet \lambda s : \text{State} \bullet f([[[\textbf{skip}]_S(s)])]
\]
\[ = Y(\lambda f : \text{State} \rightarrow \text{State} \bullet \lambda s : \text{State} \bullet f(s))
\]
\[ = Y(\lambda f : \text{State} \rightarrow \text{State} \bullet f)
\]
\[ = \bot \]
Example Statement Semantics

\[
[[\textbf{while} \ True \ \textbf{do} \ \textbf{skip}]]_S = Y(\lambda \ f : \ State \ \rightarrow \ State \ \bullet \ \lambda \ s : \ State \ \bullet \ f([\textbf{skip}]_S(s)))
\]

\[
= Y(\lambda \ f : \ State \ \rightarrow \ State \ \bullet \ \lambda \ s : \ State \ \bullet \ f(s))
\]

\[
= Y(\lambda \ f : \ State \ \rightarrow \ State \ \bullet \ f)
\]

\[
= \bot
\]

For \( k : \mathbb{N} \), we have:

\[
[[\textbf{while} \ n > 0 \ \textbf{do} \ (r := n \ast r ; \ n := n - 1)]]_S(\{n\mapsto k, \ r\mapsto 1\}) = \{n\mapsto 0, \ r\mapsto k!\}.
\]
Summary
Summary

- The syntax of programming languages is best described mathematically.
Summary

• The **syntax** of programming languages is best described mathematically.
  – Different formalisation paradigms: grammars, regular expressions, automata.
Summary

• The syntax of programming languages is best described mathematically.
  – Different formalisation paradigms: grammars, regular expressions, automata.
  – Formalising context-free syntax is easy.
Summary

• The **syntax** of programming languages is best described mathematically.
  – Different formalisation paradigms: grammars, regular expressions, automata.
  – Formalising context-free syntax is easy.
  – Formalisations can be used by lexer / parser generators etc.
Summary

- The **syntax** of programming languages is best described mathematically.
  - Different formalisation paradigms: grammars, regular expressions, automata.
  - Formalising context-free syntax is easy.
  - Formalisations can be used by lexer / parser generators etc.
  - Formalising scope and typing rules can be more involved.
Summary

• The **syntax** of programming languages is best described mathematically.
  – Different formalisation paradigms: grammars, regular expressions, automata.
  – Formalising context-free syntax is easy.
  – Formalisations can be used by lexer / parser generators etc.
  – Formalising scope and typing rules can be more involved.

• The **semantics** of programming languages is best described mathematically.
Summary

• The **syntax** of programming languages is best described mathematically.
  – Different formalisation paradigms: grammars, regular expressions, automata.
  – Formalising context-free syntax is easy.
  – Formalisations can be used by lexer / parser generators etc.
  – Formalising scope and typing rules can be more involved.

• The **semantics** of programming languages is best described mathematically.
  – Different formalisation paradigms: operational, denotational, axiomatic.
Summary

• The **syntax** of programming languages is best described mathematically.
  – Different formalisation paradigms: grammars, regular expressions, automata.
  – Formalising context-free syntax is easy.
  – Formalisations can be used by lexer / parser generators etc.
  – Formalising scope and typing rules can be more involved.

• The **semantics** of programming languages is best described mathematically.
  – Different formalisation paradigms: operational, denotational, axiomatic.
  – *Correctness proofs for programs* only make sense if the semantics is formalised!
Summary

• The **syntax** of programming languages is best described mathematically.
  – Different formalisation paradigms: grammars, regular expressions, automata.
  – Formalising context-free syntax is easy.
  – Formalisations can be used by lexer / parser generators etc.
  – Formalising scope and typing rules can be more involved.

• The **semantics** of programming languages is best described mathematically.
  – Different formalisation paradigms: operational, denotational, axiomatic.
  – **Correctness proofs for programs** only make sense if the semantics is formalised!
  – Formalising semantics can employ a large mathematical toolbox.
Summary

• The syntax of programming languages is best described mathematically.
  – Different formalisation paradigms: grammars, regular expressions, automata.
  – Formalising context-free syntax is easy.
  – Formalisations can be used by lexer / parser generators etc.
  – Formalising scope and typing rules can be more involved.

• The semantics of programming languages is best described mathematically.
  – Different formalisation paradigms: operational, denotational, axiomatic.
  – Correctness proofs for programs only make sense if the semantics is formalised!
  – Formalising semantics can employ a large mathematical toolbox.
  – Much of the complexity can be hidden in simple calculi like Hoare logic.
Summary

- The **syntax** of programming languages is best described mathematically.
  - Different formalisation paradigms: grammars, regular expressions, automata.
  - Formalising context-free syntax is easy.
  - Formalisations can be used by lexer / parser generators etc.
  - Formalising scope and typing rules can be more involved.

- The **semantics** of programming languages is best described mathematically.
  - Different formalisation paradigms: operational, denotational, axiomatic.
  - **Correctness proofs for programs** only make sense if the semantics is formalised!
  - Formalising semantics can employ a large mathematical toolbox.
  - Much of the complexity can be hidden in simple calculi like Hoare logic.

- Some programming languages have a “**more mathematical semantics**”
Summary

• The **syntax** of programming languages is best described mathematically.
  – Different formalisation paradigms: grammars, regular expressions, automata.
  – Formalising context-free syntax is easy.
  – Formalisations can be used by lexer / parser generators etc.
  – Formalising scope and typing rules can be more involved.

• The **semantics** of programming languages is best described mathematically.
  – Different formalisation paradigms: operational, denotational, axiomatic.
  – **Correctness proofs for programs** only make sense if the semantics is formalised!
  – Formalising semantics can employ a large mathematical toolbox.
  – Much of the complexity can be hidden in simple calculi like Hoare logic.

• Some programming languages have a “**more mathematical semantics**”: mathematical rules of reasoning can be applied directly to program constructs
Summary

• The **syntax** of programming languages is best described mathematically.
  – Different formalisation paradigms: grammars, regular expressions, automata.
  – Formalising context-free syntax is easy.
  – Formalisations can be used by lexer / parser generators etc.
  – Formalising scope and typing rules can be more involved.

• The **semantics** of programming languages is best described mathematically.
  – Different formalisation paradigms: operational, denotational, axiomatic.
  – **Correctness proofs for programs** only make sense if the semantics is formalised!
  – Formalising semantics can employ a large mathematical toolbox.
  – Much of the complexity can be hidden in simple calculi like Hoare logic.

• Some programming languages have a “**more mathematical semantics**”: mathematical rules of reasoning can be applied directly to program constructs

• **Programming is a mathematical activity**
Semantics with Exceptions — Simple Statements

\[
[\_ \_ ]_S : \ Stmt \rightarrow (\ Store \ \text{div} \ (\ Store \ + \ (\ Store \ \times \ \text{Num}\ )))
\]
Semantics with Exceptions — Simple Statements

\[ \llbracket \_ \rrbracket_S : Stmt \rightarrow (\text{Store} \mapsto (\text{Store} + (\text{Store} \times \text{Num}))) \]

\[ \llbracket \text{skip} \rrbracket_S = \text{Left} \]
Semantics with Exceptions — Simple Statements

\[ \llbracket \_ \rrbracket_S : Stmt \rightarrow (Store \leftrightarrow (Store + (Store \times Num))) \]

\[ \llbracket \text{skip} \rrbracket_S = \text{Left} \]

\[ \llbracket s_1 ; s_2 \rrbracket_S (s) = \begin{cases} \llbracket s_2 \rrbracket_S (t) & \text{if } \llbracket s_1 \rrbracket_S (s) = \text{Left } t \\ \text{Right}(t, e) & \text{if } \llbracket s_1 \rrbracket_S (s) = \text{Right } (t, e) \end{cases} \]
Semantics with Exceptions — Simple Statements

\[ [\_ ]_S : Stmt \rightarrow (\text{Store} \oplus (\text{Store} \times \text{Num})) \]

\[ [\text{skip}]_S = \text{Left} \]

\[ [s_1 ; s_2]_S (s) = \begin{cases} [s_2]_S (t) & \text{if } [s_1]_S (s) = \text{Left } t \\ \text{Right}(t, e) & \text{if } [s_1]_S (s) = \text{Right } (t, e) \end{cases} \]

\[ [\text{try } s_1 \text{ catch}(i) s_2]_S (s) = \begin{cases} t & \text{if } [s_1]_S (s) = \text{Left } t \\ [s_2]_S (t \oplus \{ i \mapsto e \}) & \text{if } [s_1]_S (s) = \text{Right } (t, e) \\ \bot & \text{if } s \notin \text{dom } [s_1]_S \end{cases} \]
Semantics with Exceptions — Expressions

Expr → (Store → (Val + Num))
Semantics with Exceptions — Expressions

\[
\begin{align*}
Expr & \rightarrow (Store \rightarrow (Val + Num)) \\
\begin{array}{ll}
\llbracket \text{throw } e \rrbracket_S(s) &= \\
&= \begin{cases} \\
Right (s, \text{val}) & \text{if } \llbracket e \rrbracket_E(s) = \text{Left val} \\
Right (s, \text{exc}) & \text{if } \llbracket e \rrbracket_E(s) = \text{Right exc}
\end{cases}
\end{array}
\end{align*}
\]
Semantics with Exceptions — Expressions

$Expr \rightarrow (Store \rightarrow (Val + Num))$

$\llbracket throw\ e\rrbracket_{S}(s) = \begin{cases} 
Right\ (s,\ val) & \text{if}\ \llbracket e\rrbracket_{E}(s) = Left\ val \\
Right\ (s,\ exc) & \text{if}\ \llbracket e\rrbracket_{E}(s) = Right\ exc
\end{cases}$

$\llbracket v := e\rrbracket_{S}(s) = \begin{cases} 
Left\ (s \oplus \{v \mapsto val\}) & \text{if}\ \llbracket e\rrbracket_{E}(s) = Left\ val \\
Right\ (s,\ exc) & \text{if}\ \llbracket e\rrbracket_{E}(s) = Right\ exc
\end{cases}$
Semantics with Exceptions — Expressions

\[ \text{Expr} \rightarrow (\text{Store} \rightarrow (\text{Val} + \text{Num})) \]

\[
\lfloor \text{throw } e \rfloor_S(s) = \begin{cases} 
\text{Right } (s, \text{val}) & \text{if } \lfloor e \rfloor_E(s) = \text{Left val} \\
\text{Right } (s, \text{exc}) & \text{if } \lfloor e \rfloor_E(s) = \text{Right exc}
\end{cases}
\]

\[
\lfloor v := e \rfloor_S(s) = \begin{cases} 
\text{Left } (s \oplus \{v \mapsto \text{val}\}) & \text{if } \lfloor e \rfloor_E(s) = \text{Left val} \\
\text{Right } (s, \text{exc}) & \text{if } \lfloor e \rfloor_E(s) = \text{Right exc}
\end{cases}
\]

\[
\lfloor \text{if } b \text{ then } s_1 \text{ else } s_2 \rfloor_S(s) = \begin{cases} 
\lfloor s_1 \rfloor_S(s) & \text{if } \lfloor b \rfloor_E(s) = \text{Left True} \\
\lfloor s_2 \rfloor_S(s) & \text{if } \lfloor b \rfloor_E(s) = \text{Left False} \\
\text{Right } (s, \text{exc}_B) & \text{if } \lfloor b \rfloor_E(s) = \text{Left } n \land n \in \text{Num} \\
\text{Right } (s, \text{exc}) & \text{if } \lfloor b \rfloor_E(s) = \text{Right exc}
\end{cases}
\]
while-Semantics with Exceptions
while-Semantics with Exceptions

Exercise!
Output

Semantic Domains for simple imperative programs with print statements:
Output

**Semantic Domains** for simple imperative programs with **print** statements:

\[
SVal = \text{Bool} + \text{Num} \quad \text{storable values}
\]

\[
Store = \text{Id} \rightarrow SVal \quad \text{(simple) stores}
\]
Output

**Semantic Domains** for simple imperative programs with **print** statements:

\[
\begin{align*}
  SVal & = \text{Bool} + \text{Num} & \text{storable values} \\
  Store & = \text{Id} \rightarrow SVal & \text{(simple) stores} \\
  State & = \text{Store}^{-1} \times [\text{Num}] & \text{states including output}
\end{align*}
\]
**Output**

**Semantic Domains** for simple imperative programs with `print` statements:

- $SVal = \text{Bool} + \text{Num}$ storable values
- $Store = \text{Id} \mapsto SVal$ (simple) stores
- $State = Store^\perp \times \text{[Num]}$ states including output
- $Val = SVal$ values
- $Store \mapsto Val$ (expression semantics)
- $State \rightarrow State$ (new statement semantics)
Output

**Semantic Domains** for simple imperative programs with **print** statements:

\[
\begin{align*}
SVal &= \text{Bool} + \text{Num} & \text{storable values} \\
Store &= \text{Id} \rightarrow SVal & \text{(simple) stores} \\
State &= Store^\bot \times [\text{Num}] & \text{states including output} \\
Val &= SVal & \text{values} \\
Store &\rightarrow Val & \text{(expression semantics)} \\
State &\rightarrow State & \text{(new statement semantics)}
\end{align*}
\]

In case of program errors or nontermination, **previous output is not lost**!

\[
[[\text{print } e]]_S = \lambda (s, ns) : State \begin{cases} 
(s, n:ns) & \text{if } n = [[e]]_E(s) \in \text{Num} \\
(\bot, ns) & \text{otherwise}
\end{cases}
\]
Output

Semantic Domains for simple imperative programs with print statements:

- \( SVal = Bool + Num \) storable values
- \( Store = Id \to SVal \) (simple) stores
- \( State = Store^{⊥} \times [Num] \) states including output
- \( Val = SVal \) values
- \( Store \to Val \) (expression semantics)
- \( State \to State \) (new statement semantics)

In case of program errors or nontermination, previous output is not lost!

\[
\llbracket \text{print } e \rrbracket_S = \lambda (s, ns) : State \cdot \begin{cases} 
(s, n:ns) & \text{if } n = \llbracket e \rrbracket_E(s) \in Num \\
(\bot, ns) & \text{otherwise}
\end{cases}
\]

Note: statement semantics here is oversimplified — fixpoint construction in \( State \to State \) does not work, except with Haskell-like list domains.
Input
Input

- **Output** is reflected by the introduction of a state component representing past output:

\[
State = Store^\perp \times [Num]
\]
Input

• **Output** is reflected by the introduction of a state component representing past output:

\[
State = Store^\perp \times [\text{Num}]
\]

• (Additional) **Input** is reflected by the introduction of a state component representing future input:

\[
State = Store^\perp \times [\text{Num}] \times [\text{Num}]
\]
Input

- **Output** is reflected by the introduction of a state component representing *past output*:

  \[
  \text{State} = \text{Store}^\perp \times \text{[Num]} \]

- *(Additional) Input* is reflected by the introduction of a state component representing *future input*:

  \[
  \text{State} = \text{Store}^\perp \times \text{[Num]} \times \text{[Num]} \]

\[
[[\text{read } v]]_S = \\
\lambda (s, outs, ins) : \text{State} \mapsto \begin{cases} 
(s \oplus \{v\mapsto\text{in}\}, outs, ins') & \text{if } ins = \text{in}:\text{ins'} \\
(\bot, outs, ins) & \text{if } ins = [] 
\end{cases}
\]
Scope

Nested scopes with shadowing of identifiers are modelled as stacks (lists) of environments:

\[ Env = Id \rightarrow SVal^\perp \]

environments (with \( \perp \) for uninit. var.)

\[ Store = [Env] \]

stores

\[ State = Store^\perp \times [Num] \times [Num] \]

states including I/O
Records

Semantic Domains: Only storable values change:

\[ SVal = \{Bool + Num + (Id \mapsto SVal)\} \]

\[ State = \{Id \mapsto SVal\} \]

\[ State \mapsto SVal \]

\[ State \mapsto State \]

New record field expressions:

\[ \llbracket e.f \rrbracket_E = \lambda s : State \cdot (\llbracket e \rrbracket_E(s)) f \]

New record construction expressions (not in C or Oberon, but e.g. in Ada):

\[ \llbracket \text{record}(f_1 = e_1, \ldots, f_n = e_n) \rrbracket_E = \lambda s : State \cdot \{f_1 \mapsto \llbracket e_1 \rrbracket_E(s), \ldots, f_n \mapsto \llbracket e_n \rrbracket_E(s)\} \]

New record field assignment statements:

\[ \llbracket r.f := e \rrbracket_S = \lambda s : State \cdot s \oplus \{r \mapsto ((s \cdot r) \oplus \{f \mapsto \llbracket e \rrbracket_E(s)\})\} \]
Semantics  Semantics  Semantics  Semantics  Semantics  Semantics  Semantics  Semantics
Semantics  Semantics  Semantics  Semantics  Semantics  Semantics  Semantics  Semantics
Semantics  Semantics  Semantics  Semantics  Semantics  Semantics  Semantics  Semantics
Semantics  Semantics  Semantics  Semantics  Semantics  Semantics  Semantics  Semantics
Semantics  Semantics
Lecture 2

Axiomatic Semantics
Axiomatic Semantics
Axiomatic Semantics

Derivation of judgements written as “Hoare triples”

\[ \{P\}S\{Q\} \]
Axiomatic Semantics

Derivation of judgements written as “Hoare triples”

\[ \{P\} S \{Q\} \]

where \( P \) and \( Q \) are formulae denoting conditions on execution states
Axiomatic Semantics

Derivation of judgements written as “Hoare triples”

\[
\{P\}S\{Q\}
\]

where \(P\) and \(Q\) are formulae denoting conditions on execution states:

- \(P\) is the **precondition**
- \(S\) is a program fragment (statement)
- \(Q\) is the **postcondition**
Axiomatic Semantics

Derivation of judgements written as “Hoare triples”

\( \{P\}S\{Q\} \)

where \( P \) and \( Q \) are formulae denoting conditions on execution states:

- \( P \) is the **precondition**
- \( S \) is a program fragment (statement)
- \( Q \) is the **postcondition**

A Hoare triple \( \{P\}S\{Q\} \) has two readings
Axiomatic Semantics

Derivation of judgements written as “Hoare triples”

\[ \{ P \} S \{ Q \} \]

where \( P \) and \( Q \) are formulae denoting conditions on execution states:

- \( P \) is the **precondition**
- \( S \) is a program fragment (statement)
- \( Q \) is the **postcondition**

A Hoare triple \( \{ P \} S \{ Q \} \) has two readings:

**Total correctness:** If \( S \) starts in a state satisfying \( P \), then it terminates and its terminating state satisfies \( Q \)
Axiomatic Semantics

Derivation of judgements written as “Hoare triples”

\(\{P\}S\{Q\}\)

where \(P\) and \(Q\) are formulae denoting conditions on execution states:

- \(P\) is the **precondition**
- \(S\) is a program fragment (statement)
- \(Q\) is the **postcondition**

A Hoare triple \(\{P\}S\{Q\}\) has two readings:

**Total correctness:** If \(S\) starts in a state satisfying \(P\), 
then it terminates and its terminating state satisfies \(Q\)

**Partial correctness:** If \(S\) starts in a state satisfying \(P\) and terminates, 
then its terminating state satisfies \(Q\)
Axiomatic Semantics

Derivation of judgements written as “Hoare triples”

\[ \{P\} S \{Q\} \]

where \( P \) and \( Q \) are formulae denoting conditions on execution states:

- \( P \) is the **precondition**
- \( S \) is a program fragment (statement)
- \( Q \) is the **postcondition**

A Hoare triple \( \{P\} S \{Q\} \) has two readings:

**Total correctness:** \( \text{If } S \text{ starts in a state satisfying } P, \text{ then it terminates} \) and its terminating state satisfies \( Q \)

**Partial correctness:** \( \text{If } S \text{ starts in a state satisfying } P \text{ and terminates, then} \) its terminating state satisfies \( Q \)

(“terminates” means “terminates without run-time error”)

Axiomatic Semantics

Derivation of judgements written as “Hoare triples”

\[ \{P\}S\{Q\} \]

where \( P \) and \( Q \) are formulae denoting conditions on execution states:

- \( P \) is the **precondition**
- \( S \) is a program fragment (statement)
- \( Q \) is the **postcondition**

A Hoare triple \( \{P\}S\{Q\} \) has two readings:

**Total correctness:**  
*If* \( S \) starts in a state satisfying \( P \),  
*then it terminates* and its terminating state satisfies \( Q \)

**Partial correctness:**  
*If* \( S \) starts in a state satisfying \( P \) and *terminates*,  
*then* its terminating state satisfies \( Q \)

(“terminates” means “terminates without run-time error”)

Axiomatic Semantics

Derivation of judgements written as “Hoare triples”

\[ \{P\}S\{Q\} \]

where \( P \) and \( Q \) are formulae denoting conditions on execution states:

- \( P \) is the **precondition**
- \( S \) is a program fragment (statement)
- \( Q \) is the **postcondition**

A Hoare triple \( \{P\}S\{Q\} \) has two readings:

**Total correctness:**  
*If* \( S \) starts in a state satisfying \( P \),  
*then it terminates* and its terminating state satisfies \( Q \)  
— “\( S \) is **totally correct with respect to** \( P \) and \( Q \)”

**Partial correctness:**  
*If* \( S \) starts in a state satisfying \( P \) and **terminates**,  
*then* its terminating state satisfies \( Q \)  
— “\( S \) is **partially correct with respect to** \( P \) and \( Q \)”

(“**terminates**” means “**terminates without run-time error**”)
Axiomatic Semantics vs. Operational Semantics

- Operational semantics relates *states* via statements
Axiomatic Semantics vs. Operational Semantics

- Operational semantics relates \textit{states} via statements
- Axiomatic semantics relates \textit{conditions on states} via statements
Axiomatic Semantics vs. Operational Semantics

- Operational semantics relates states via statements
- Axiomatic semantics relates conditions on states via statements

Therefore:
- Operational semantics facilitates investigation of examples
Axiomatic Semantics vs. Operational Semantics

- Operational semantics relates states via statements
- Axiomatic semantics relates conditions on states via statements

Therefore:
- Operational semantics facilitates investigation of examples ("testing")
Axiomatic Semantics vs. Operational Semantics

- Operational semantics relates **states** via statements
- Axiomatic semantics relates **conditions on states** via statements

*Therefore:*
- Operational semantics facilitates investigation of examples ("testing")
- Axiomatic semantics facilitates relating a program with its specification
Axiomatic Semantics vs. Operational Semantics

- Operational semantics relates states via statements
- Axiomatic semantics relates conditions on states via statements

Therefore:
- Operational semantics facilitates investigation of examples ("testing")
- Axiomatic semantics facilitates relating a program with its specification — verification
Relating Axiomatic and Operational Semantics

- Operational semantics relates **states** via statements
- Axiomatic semantics relates **conditions on states** via statements
Relating Axiomatic and Operational Semantics

- Operational semantics relates states via statements
- Axiomatic semantics relates conditions on states via statements

Relating states with conditions on states:
Relating Axiomatic and Operational Semantics

- Operational semantics relates **states** via statements
- Axiomatic semantics relates **conditions on states** via statements

Relating states with conditions on states:

- “$s \models P$” means “condition $P$ holds, or is valid, in state $s$”
Relating Axiomatic and Operational Semantics

- Operational semantics relates states via statements
- Axiomatic semantics relates conditions on states via statements

Relating states with conditions on states:
- “$s \models P$” means “condition $P$ holds, or is valid, in state $s$”

For example:
- $\{x \mapsto 5, y \mapsto 7\} \models x > 0$
Relating Axiomatic and Operational Semantics

- Operational semantics relates **states** via statements
- Axiomatic semantics relates **conditions on states** via statements

Relating states with conditions on states:

- \( s \models P \) means “condition \( P \) holds, or is valid, in state \( s \)”

For example:

- \( \{x \mapsto 5, y \mapsto 7\} \models x > 0 \)
- \( \{x \mapsto 5, y \mapsto 7\} \models \sum_{i=0}^{10} = 55 \)
Relating Axiomatic and Operational Semantics

- Operational semantics relates **states** via statements
- Axiomatic semantics relates **conditions on states** via statements

Relating states with conditions on states:
- “$s \models P$” means “condition $P$ holds, or is valid, in state $s$”

For example:
- $\{x \mapsto 5, y \mapsto 7\} \models x > 0$
- $\{x \mapsto 5, y \mapsto 7\} \models \sum_{i=0}^{10} = 55$
- $\{x \mapsto 5, y \mapsto 7\} \not\models x > y$
Relating Axiomatic and Operational Semantics

- Operational semantics relates \textit{states} via statements
- Axiomatic semantics relates \textit{conditions on states} via statements

Relating states with conditions on states:
- “\( s \models P \)” means “condition \( P \) holds, or is valid, in state \( s \)’’
Relating Axiomatic and Operational Semantics

- Operational semantics relates states via statements
- Axiomatic semantics relates conditions on states via statements

Relating states with conditions on states:
- “$s \models P$” means “condition $P$ holds, or is valid, in state $s$”

The two readings of a Hoare triple $\{P\}S\{Q\}$:

**Partial correctness:** If $S$ starts in a state satisfying $P$ and terminates, then its terminating state satisfies $Q$
Relating Axiomatic and Operational Semantics

- Operational semantics relates **states** via statements
- Axiomatic semantics relates **conditions on states** via statements

Relating states with conditions on states:
- “$s \models P$” means “condition $P$ holds, or is valid, in state $s$”

The two readings of a Hoare triple $\{P\}S\{Q\}$:

**Partial correctness:** If $S$ starts in a state satisfying $P$ and **terminates**, then its terminating state satisfies $Q$

*I.e.: For all* states $\sigma_1$ and $\sigma_2$
Relating Axiomatic and Operational Semantics

- Operational semantics relates states via statements
- Axiomatic semantics relates conditions on states via statements

Relating states with conditions on states:
- "s ⊨ P" means "condition P holds, or is valid, in state s"

The two readings of a Hoare triple \{P\}S\{Q\}:

**Partial correctness:**  If S starts in a state satisfying P and terminates, then its terminating state satisfies Q

I.e.: For all states \(\sigma_1\) and \(\sigma_2\), if \(\sigma_1 \models P\)
Relating Axiomatic and Operational Semantics

- Operational semantics relates states via statements
- Axiomatic semantics relates conditions on states via statements

Relating states with conditions on states:
- “$s \models P$” means “condition $P$ holds, or is valid, in state $s$”

The two readings of a Hoare triple $\{P\}S\{Q\}$:

**Partial correctness:** If $S$ starts in a state satisfying $P$ and terminates, then its terminating state satisfies $Q$

*I.e.*: For all states $\sigma_1$ and $\sigma_2$,
if $\sigma_1 \models P$ and $\sigma_1(S) \Rightarrow \sigma_2$
Relating Axiomatic and Operational Semantics

- Operational semantics relates states via statements
- Axiomatic semantics relates conditions on states via statements

Relating states with conditions on states:
- “$s \models P$” means “condition $P$ holds, or is valid, in state $s$”

The two readings of a Hoare triple $\{P\}S\{Q\}$:

**Partial correctness:** If $S$ starts in a state satisfying $P$ and terminates, then its terminating state satisfies $Q$

I.e.: For all states $\sigma_1$ and $\sigma_2$,
if $\sigma_1 \models P$ and $\sigma_1(S) \Rightarrow \sigma_2$, then $\sigma_2 \models Q$
Relating Axiomatic and Operational Semantics

- Operational semantics relates states via statements
- Axiomatic semantics relates conditions on states via statements

Relating states with conditions on states:
- “$s \models P$” means “condition $P$ holds, or is valid, in state $s$”

The two readings of a Hoare triple $\{P\}S\{Q\}$:

**Partial correctness:** If $S$ starts in a state satisfying $P$ and terminates, then its terminating state satisfies $Q$  
**I.e.:** For all states $\sigma_1$ and $\sigma_2$, if $\sigma_1 \models P$ and $\sigma_1(S) \Rightarrow \sigma_2$, then $\sigma_2 \models Q$

**Total correctness:** If $S$ starts in a state satisfying $P$, then it terminates and its terminating state satisfies $Q$
Relating Axiomatic and Operational Semantics

- Operational semantics relates states via statements
- Axiomatic semantics relates conditions on states via statements

Relating states with conditions on states:
- “$s \models P$” means “condition $P$ holds, or is valid, in state $s$”

The two readings of a Hoare triple \{P\}S\{Q\}:

**Partial correctness:** *If* $S$ starts in a state satisfying $P$ and *terminates*, *then* its terminating state satisfies $Q$

*I.e.: For all* states $\sigma_1$ and $\sigma_2$,
*if* $\sigma_1 \models P$ and $\sigma_1(S) \Rightarrow \sigma_2$, *then* $\sigma_2 \models Q$

**Total correctness:** *If* $S$ starts in a state satisfying $P$,* then it terminates* and its terminating state satisfies $Q$

*I.e.: For all* states $\sigma_1$
Relating Axiomatic and Operational Semantics

- Operational semantics relates states via statements
- Axiomatic semantics relates conditions on states via statements

Relating states with conditions on states:
- “$s \models P$” means “condition $P$ holds, or is valid, in state $s$”

The two readings of a Hoare triple $\{P\}S\{Q\}$:

**Partial correctness:** If $S$ starts in a state satisfying $P$ and terminates, then its terminating state satisfies $Q$

*I.e.: For all* states $\sigma_1$ and $\sigma_2$,

*if* $\sigma_1 \models P$ *and* $\sigma_1(S) \Rightarrow \sigma_2$, *then* $\sigma_2 \models Q$

**Total correctness:** If $S$ starts in a state satisfying $P$, then it terminates and its terminating state satisfies $Q$

*I.e.: For all* states $\sigma_1$, *if* $\sigma_1 \models P$
Relating Axiomatic and Operational Semantics

- Operational semantics relates **states** via statements
- Axiomatic semantics relates **conditions on states** via statements

Relating states with conditions on states:
- “$s \models P$” means “condition $P$ holds, or is valid, in state $s$”

The two readings of a Hoare triple $\{P\}S\{Q\}$:

**Partial correctness:** If $S$ starts in a state satisfying $P$ and **terminates**, then its terminating state satisfies $Q$

I.e.: For all states $\sigma_1$ and $\sigma_2$,
if $\sigma_1 \models P$ and $\sigma_1(S) \Rightarrow \sigma_2$, then $\sigma_2 \models Q$

**Total correctness:** If $S$ starts in a state satisfying $P$,
then it terminates and its terminating state satisfies $Q$

I.e.: For all states $\sigma_1$, if $\sigma_1 \models P$,
then there is a state $\sigma_2$
Relating Axiomatic and Operational Semantics

- Operational semantics relates states via statements
- Axiomatic semantics relates conditions on states via statements

Relating states with conditions on states:
• “s \models P” means “condition P holds, or is valid, in state s”

The two readings of a Hoare triple \{P\}S\{Q\}:

**Partial correctness:** If S starts in a state satisfying P and terminates, then its terminating state satisfies Q

*I.e.: For all* states \(\sigma_1\) and \(\sigma_2\),
if \(\sigma_1 \models P\) and \(\sigma_1(S) \Rightarrow \sigma_2\), then \(\sigma_2 \models Q\)

**Total correctness:** If S starts in a state satisfying P, then it terminates and its terminating state satisfies Q

*I.e.: For all* states \(\sigma_1\), if \(\sigma_1 \models P\), then there is a state \(\sigma_2\)
such that \(\sigma_1(S) \Rightarrow \sigma_2\)
Relating Axiomatic and Operational Semantics

- Operational semantics relates states via statements
- Axiomatic semantics relates conditions on states via statements

Relating states with conditions on states:
- “$s \models P$” means “condition $P$ holds, or is valid, in state $s$”

The two readings of a Hoare triple $\{P\}S\{Q\}$:

**Partial correctness:** If $S$ starts in a state satisfying $P$ and terminates, then its terminating state satisfies $Q$

*I.e.: For all* states $\sigma_1$ and $\sigma_2$,
if $\sigma_1 \models P$ and $\sigma_1(S) \Rightarrow \sigma_2$, *then* $\sigma_2 \models Q$

**Total correctness:** If $S$ starts in a state satisfying $P$, then it terminates and its terminating state satisfies $Q$

*I.e.: For all* states $\sigma_1$, *if* $\sigma_1 \models P$,
*then there is* a state $\sigma_2$
such that $\sigma_1(S) \Rightarrow \sigma_2$, *and* $\sigma_2 \models Q
Proving Partial and Total Correctness

Total correctness of $\{P\} S \{Q\}$

is equivalent to
Proving Partial and Total Correctness

Total correctness of \( \{P\} \ S \ \{Q\} \)

is equivalent to

partial correctness of \( \{P\} \ S \ \{Q\} \) together with the fact that \( S \) terminates when started in a state satisfying \( P \)
Proving Partial and Total Correctness

**Total correctness** of \( \{P\} S \{Q\} \)

*is equivalent to*

**partial correctness** of \( \{P\} S \{Q\} \text{ together with} \) the fact that \( S \) terminates when started in a state satisfying \( P \)

\[ \Rightarrow \] usually, separate **termination proof**!
Proving Partial and Total Correctness

Total correctness of $\{P\} S \{Q\}$

is equivalent to

partial correctness of $\{P\} S \{Q\}$ together with the fact that $S$ terminates when started in a state satisfying $P$

$\Rightarrow$ usually, separate termination proof!

• For partial correctness, it is relatively easy to give a direct proof calculus
Proving Partial and Total Correctness

Total correctness of \( \{P\} S \{Q\} \)

is equivalent to

partial correctness of \( \{P\} S \{Q\} \) together with the fact that \( S \) terminates when started in a state satisfying \( P \)

\[ \Rightarrow \] usually, separate termination proof!

- For partial correctness, it is relatively easy to give a direct proof calculus
- Proving partial correctness therefore does not need operational semantics
Proving Partial and Total Correctness

Total correctness of \{P\} S \{Q\} is equivalent to partial correctness of \{P\} S \{Q\} together with the fact that S terminates when started in a state satisfying P

\[\Rightarrow\] usually, separate termination proof!

- For partial correctness, it is relatively easy to give a direct proof calculus
- Proving partial correctness therefore does not need operational semantics
- In the following, we will study and use this calculus
Proving Partial and Total Correctness

Total correctness of \( \{P\} S \{Q\} \) is equivalent to partial correctness of \( \{P\} S \{Q\} \) together with the fact that \( S \) terminates when started in a state satisfying \( P \)

\[ \Rightarrow \] usually, separate termination proof!

- For partial correctness, it is relatively easy to give a direct proof calculus
- Proving partial correctness therefore does not need operational semantics
- In the following, we will study and use this calculus
- (Termination proofs use different methods — well-ordered sets)
Proving Partial and Total Correctness

Total correctness of \( \{P\} S \{Q\} \)

is equivalent to

partial correctness of \( \{P\} S \{Q\} \) together with the fact that \( S \) terminates when started in a state satisfying \( P \)

\[ \Rightarrow \] usually, separate termination proof!

- For partial correctness, it is relatively easy to give a direct proof calculus
- Proving partial correctness therefore does not need operational semantics
- In the following, we will study and use this calculus
- (Termination proofs use different methods — well-ordered sets)

Unless explicitly mentioned, we read “\( \{P\} S \{Q\} \)” as meaning partial correctness.
Derivation Rules for Sequencing, Conditionals, Loops

Logical consequence: 

\[ P \Rightarrow P' \quad \{P'\} S\{Q'\} \quad Q' \Rightarrow Q \]

\[ \{P\} S\{Q\} \]
Derivation Rules for Sequencing, Conditionals, Loops

Logical consequence:

\[
\begin{align*}
P & \Rightarrow P' \\
\{P'\}S\{Q'\} & \\
Q' & \Rightarrow Q \\
\{P\}S\{Q\}
\end{align*}
\]
Derivation Rules for Sequencing, Conditionals, Loops

Logical consequence: \[ P \Rightarrow P' \quad \{P'\}S\{Q'\} \quad Q' \Rightarrow Q \]
\[ \{P\}S\{Q\} \]

Sequence: \[ \{P\}S_1\{R\} \quad \{R\}S_2\{Q\} \]
\[ \{P\}S_1; \ S_2\{Q\} \]
Derivation Rules for Sequencing, Conditionals, Loops

**Logical consequence:**

\[
\frac{P \Rightarrow P'}{\{P\}'S\{Q\}'} \quad \frac{Q' \Rightarrow Q}{\{P\}'S\{Q\}}
\]

**Sequence:**

\[
\frac{\{P\}S_1\{R\} \quad \{R\}S_2\{Q\}}{\{P\}S_1; \ S_2\{Q\}}
\]

**Conditional:**

\[
\frac{\{P \land b\}S_1\{Q\} \quad \{P \land \neg b\}S_2\{Q\}}{\{P\}\text{if } b \text{ then } S_1 \text{ else } S_2 \text{ fi}\{Q\}}
\]
Derivation Rules for Sequencing, Conditionals, Loops

Logical consequence: \[ P \Rightarrow P' \quad \{P'\}S\{Q'\} \quad Q' \Rightarrow Q \]
\[ \{P\}S\{Q\} \]

Sequence: \[ \{P\}S_1\{R\} \quad \{R\}S_2\{Q\} \]
\[ \{P\}S_1; \ S_2\{Q\} \]

Conditional: \[ \{P \land b\}S_1\{Q\} \quad \{P \land \neg b\}S_2\{Q\} \]
\[ \{P\}\text{if }b\text{ then }S_1\text{ else }S_2\text{ fi}\{Q\} \]

while-Loop: \[ \{\text{INV} \land b\}S\{\text{INV}\} \]
\[ \{\text{INV}\}\text{while }b\text{ do }S\text{ od}\{\text{INV} \land \neg b\} \]
Axiom Schema for Assignments

\[ \{P[x \setminus e]\}x := e\{P\} \]
Axiom Schema for Assignments

\[ \{P[x \setminus e]\}_x := e\{P\} \]

Examples:

- \( \{2 = 2\}_x := 2\{x = 2\} \)
Axiom Schema for Assignments

\[ \{P[x \setminus e]\}x := e\{P\} \]

Examples:

- \( \{2 = 2\}x := 2\{x = 2\} \)

- \( \{x + 1 = 2\}x := x + 1\{x = 2\} \)
Axiom Schema for Assignments

\[ \{P[x \setminus e]\}x := e\{P\} \]

Examples:

- \(\{2 = 2\}x := 2\{x = 2\}\)
- \(\{x + 1 = 2\}x := x + 1\{x = 2\}\)
- \(\{n + 1 = 2\}x := n + 1\{x = 2\}\)
Axiom Schema for Assignments

\[ \{P[x \setminus e]\}x := e\{P\} \]

Examples:

- \( \{2 = 2\}x := 2\{x = 2\} \)
- \( \{x + 1 = 2\}x := x + 1\{x = 2\} \)
- \( \{n + 1 = 2\}x := n + 1\{x = 2\} \)

Typically, Hoare triples are derived starting from the postcondition
Axiom Schema for Assignments

\[ \{P[x \setminus e]\}x := e\{P\} \]

Examples:

- \(\{2 = 2\}x := 2\{x = 2\}\)
- \(\{x + 1 = 2\}x := x + 1\{x = 2\}\)
- \(\{n + 1 = 2\}x := n + 1\{x = 2\}\)

Typically, Hoare triples are derived starting from the postcondition — backward reasoning.
Axiom Schema for Assignments

\[ \{P[x \setminus e]\}x := e\{P\} \]

Examples:

- \( \{2 = 2\}x := 2\{x = 2\} \)
- \( \{x + 1 = 2\}x := x + 1\{x = 2\} \)
- \( \{n + 1 = 2\}x := n + 1\{x = 2\} \)

Typically, Hoare triples are derived starting from the postcondition — backward reasoning.

Considering this axiom schema as a way to calculate a precondition from assignment and postcondition, it calculates the weakest precondition that completes a valid Hoare triple.
Example Verification

\{\text{True}\} k := 0; s := 0; \textbf{while} k \neq n \textbf{do} k := k + 1; s := s + k \textbf{od}\{s = \sum_{i=1}^{n} i\}
Example Annotated Program

\{ \text{True} \} \Rightarrow \{ 0 = \sum_{i=1}^{0} i \} \quad k := 0; \quad \{ 0 = \sum_{i=1}^{k} i \} \quad s := 0; \quad \{ s = \sum_{i=1}^{k} i \} \quad \text{while } k \neq n \quad \begin{array}{l}
\text{do} \quad \{ s = \sum_{i=1}^{k} i \land k \neq n \} \Rightarrow \{ s + k + 1 = \sum_{i=1}^{k+1} i \} \\
\quad k := k + 1; \quad \{ s + k = \sum_{i=1}^{k} i \} \\
\quad s := s + k \quad \{ s = \sum_{i=1}^{k} i \} \\
\text{od} \quad \{ s = \sum_{i=1}^{k} i \land k = n \} \Rightarrow \{ s = \sum_{i=1}^{n} i \}
Example Verification

\{\text{True}\} k := 0; s := 0; \textbf{while} k \neq n \textbf{ do } k := k + 1; s := s + k \textbf{ od}\{ s = \sum_{i=1}^{n} i \}
Example Verification

\{\text{True}\} k := 0; s := 0; \textbf{while} k \neq n \textbf{ do } k := k + 1; s := s + k \textbf{ od}\{s = \sum_{i=1}^{n} i\}

\Leftrightarrow \{\text{True}\} k := 0; s := 0; \textbf{while} k \neq n \textbf{ do } k := k + 1; s := s + k \textbf{ od}\{s = \sum_{i=1}^{k} i \land k = n\}
Example Verification

\{\text{True}\} k := 0; \ s := 0; \textbf{while} \ k \neq n \textbf{ do} \ k := k + 1; \ s := s + k \textbf{ od} \{s = \sum_{i=1}^{n} i\}

\Leftrightarrow \{\text{True}\} k := 0; \ s := 0; \textbf{while} \ k \neq n \textbf{ do} \ k := k + 1; \ s := s + k \textbf{ od} \{s = \sum_{i=1}^{k} i \land k = n\}

\Leftrightarrow \{\text{True}\} k := 0; \ s := 0 \{s = \sum_{i=1}^{k} i\}

\land
Example Verification

\{ \text{True} \} k := 0; s := 0; \textbf{while} k \neq n \textbf{ do } k := k + 1; s := s + k \textbf{ od}\{ s = \sum_{i=1}^{n} i \}\]

\Leftrightarrow \{ \text{True} \} k := 0; s := 0; \textbf{while} k \neq n \textbf{ do } k := k + 1; s := s + k \textbf{ od}\{ s = \sum_{i=1}^{k} i \land k = n \}\]

\Leftrightarrow \{ \text{True} \} k := 0; s := 0\{ s = \sum_{i=1}^{k} i \}\]

\land \{ s = \sum_{i=1}^{k} i \} \textbf{while} k \neq n \textbf{ do } k := k + 1; s := s + k \textbf{ od}\{ s = \sum_{i=1}^{k} i \land k = n \}
Example Verification

\{\text{True}\} k := 0; s := 0; \textbf{while} k \neq n \textbf{ do } k := k + 1; s := s + k \textbf{ od}\{s = \sum_{i=1}^{n} i\}

\iffalse \{\text{True}\} k := 0; s := 0; \textbf{while} k \neq n \textbf{ do } k := k + 1; s := s + k \textbf{ od}\{s = \sum_{i=1}^{k} i \land k = n\}
\fi

\iffalse \{\text{True}\} k := 0; s := 0\{s = \sum_{i=1}^{k} i\}
\land \{s = \sum_{i=1}^{k} i\} \textbf{ while } k \neq n \textbf{ do } k := k + 1; s := s + k \textbf{ od}\{s = \sum_{i=1}^{k} i \land k = n\}
\fi

\iffalse \{\text{True}\} k := 0\{0 = \sum_{i=1}^{k} i\}
\land
\land
\fi
Example Verification

\{\text{True}\} \quad k := 0; \quad s := 0; \quad \textbf{while} \ k \neq n \ \textbf{do} \ \ k := k + 1; \quad s := s + k \ \textbf{od}\{s = \sum_{i=1}^{n} i\}

\Leftrightarrow \{\text{True}\} \quad k := 0; \quad s := 0; \quad \textbf{while} \ k \neq n \ \textbf{do} \ \ k := k + 1; \quad s := s + k \ \textbf{od}\{s = \sum_{i=1}^{n} i \land k = n\}

\Leftrightarrow \{\text{True}\} \quad k := 0; \quad s := 0\{s = \sum_{i=1}^{k} i\}

\land \{s = \sum_{i=1}^{k} i\} \quad \textbf{while} \ k \neq n \ \textbf{do} \ \ k := k + 1; \quad s := s + k \ \textbf{od}\{s = \sum_{i=1}^{k} i \land k = n\}

\Leftrightarrow \{\text{True}\} \quad k := 0\{0 = \sum_{i=1}^{k} i\}

\land \{0 = \sum_{i=1}^{k} i\} \quad s := 0\{s = \sum_{i=1}^{k} i\}

\land
Example Verification

\{\text{True}\} k := 0; s := 0; \textbf{while} k \neq n \textbf{ do } k := k + 1; s := s + k \textbf{ od}\{s = \sum_{i=1}^{n} i\}

\Leftrightarrow \{\text{True}\} k := 0; s := 0; \textbf{while} k \neq n \textbf{ do } k := k + 1; s := s + k \textbf{ od}\{s = \sum_{i=1}^{k} i \land k = n\}

\Leftrightarrow \{\text{True}\} k := 0; s := 0\{s = \sum_{i=1}^{k} i\}

\land \{s = \sum_{i=1}^{k} i\} \textbf{while} k \neq n \textbf{ do } k := k + 1; s := s + k \textbf{ od}\{s = \sum_{i=1}^{k} i \land k = n\}

\Leftrightarrow \{\text{True}\} k := 0\{0 = \sum_{i=1}^{k} i\}

\land \{0 = \sum_{i=1}^{k} i\} s := 0\{s = \sum_{i=1}^{k} i\}

\land \{s = \sum_{i=1}^{k} i \land k \neq n\} k := k + 1; s := s + k\{s = \sum_{i=1}^{k} i\}
Example Verification (ctd.)

\[ \iff (\text{True} \implies 0 = \sum_{i=1}^{0} i) \land \{0 = \sum_{i=1}^{0} i\} k := 0 \{0 = \sum_{i=1}^{k} i\} \]

\[ \land \text{True} \]

\[ \land \{s = \sum_{i=1}^{k} i \land k \neq n\} k := k + 1 \{s + k = \sum_{i=1}^{k} i\} \]

\[ \land \{s + k = \sum_{i=1}^{k} i\} s := s + k \{s = \sum_{i=1}^{k} i\} \]
Example Verification (ctd.)

\[
\iff \ (\text{True} \implies 0 = \sum_{i=1}^{0} i) \quad \land \quad \{0 = \sum_{i=1}^{0} i\} k := 0 \{0 = \sum_{i=1}^{k} i\}
\land \text{True}
\land \{s = \sum_{i=1}^{k} i \land k \neq n\} k := k + 1 \{s + k = \sum_{i=1}^{k} i\}
\land \{s + k = \sum_{i=1}^{k} i\} s := s + k \{s = \sum_{i=1}^{k} i\}
\iff \text{True}
\land
\land
\land
\land
\]
Example Verification (ctd.)

\[
\iffalse (\text{True} \implies 0 = \sum_{i=1}^{0} i) \land \left\{ 0 = \sum_{i=1}^{0} i \right\} k := 0 \left\{ 0 = \sum_{i=1}^{k} i \right\} \\
\land \text{True} \\
\land \left\{ s = \sum_{i=1}^{k} i \land k \neq n \right\} k := k + 1 \left\{ s + k = \sum_{i=1}^{k} i \right\} \\
\land \left\{ s + k = \sum_{i=1}^{k} i \right\} s := s + k \left\{ s = \sum_{i=1}^{k} i \right\} \\
\iffalse \text{True} \\
\land s = \sum_{i=1}^{k} i \land k \neq n \implies s + k + 1 = \sum_{i=1}^{k+1} i \\
\land \\
\land
\]
Example Verification (ctd.)

\[ \iff (\text{True} \implies 0 = \sum_{i=1}^{0} i) \land \{0 = \sum_{i=1}^{0} i\}k := 0\{0 = \sum_{i=1}^{k} i\} \]

\land \text{True}

\land \{s = \sum_{i=1}^{k} i \land k \neq n\}k := k + 1\{s + k = \sum_{i=1}^{k} i\}

\land \{s + k = \sum_{i=1}^{k} i\}s := s + k\{s = \sum_{i=1}^{k} i\}

\iff \text{True}

\land s = \sum_{i=1}^{k} i \land k \neq n \implies s + k + 1 = \sum_{i=1}^{k+1} i

\land \{s + k + 1 = \sum_{i=1}^{k+1} i\}k := k + 1\{s + k = \sum_{i=1}^{k} i\}

\land \]
Example Verification (ctd.)

\[ \Leftrightarrow (\text{True } \Rightarrow 0 = \sum_{i=1}^{0} i ) \land \{ 0 = \sum_{i=1}^{0} i \} k := 0 \{ 0 = \sum_{i=1}^{k} i \} \]

\[ \land \text{ True} \]

\[ \land \{ s = \sum_{i=1}^{k} i \land k \neq n \} k := k + 1 \{ s + k = \sum_{i=1}^{k} i \} \]

\[ \land \{ s + k = \sum_{i=1}^{k} i \} s := s + k \{ s = \sum_{i=1}^{k} i \} \]

\[ \Leftrightarrow \text{ True} \]

\[ \land s = \sum_{i=1}^{k} i \land k \neq n \Rightarrow s + k + 1 = \sum_{i=1}^{k+1} i \]

\[ \land \{ s + k + 1 = \sum_{i=1}^{k+1} i \} k := k + 1 \{ s + k = \sum_{i=1}^{k} i \} \]

\[ \land \text{ True} \]
Example Verification (ctd.)

\[ \iff (\text{True} \implies 0 = \sum_{i=1}^{0} i) \land \{0 = \sum_{i=1}^{0} i\} k := 0 \{0 = \sum_{i=1}^{k} i\} \]
\[ \land \text{True} \]
\[ \land \{s = \sum_{i=1}^{k} i \land k \neq n\} k := k + 1 \{s + k = \sum_{i=1}^{k} i\} \]
\[ \land \{s + k = \sum_{i=1}^{k} i\} s := s + k \{s = \sum_{i=1}^{k} i\} \]

\[ \iff \text{True} \]
\[ \land s = \sum_{i=1}^{k} i \land k \neq n \implies s + k + 1 = \sum_{i=1}^{k+1} i \]
\[ \land \{s + k + 1 = \sum_{i=1}^{k+1} i\} k := k + 1 \{s + k = \sum_{i=1}^{k} i\} \]
\[ \land \text{True} \]
\[ \iff \text{True} \]
Finding Proofs of Partial Correctness

- Normally, **Backward reasoning** drives the proof
Finding Proofs of Partial Correctness

• Normally, **Backward reasoning** drives the proof:
  
  Start to consider the postcondition and how the last statement achieves it
Finding Proofs of Partial Correctness

- Normally, **Backward reasoning** drives the proof:
  Start to consider the postcondition and how the last statement achieves it

- **Forward reasoning** from the precondition can be useful for simple assignment sequences and for exploration
Finding Proofs of Partial Correctness

- Normally, **Backward reasoning** drives the proof:
  Start to consider the postcondition and how the last statement achieves it

- **Forward reasoning** from the precondition can be useful for simple assignment sequences and for exploration

- For **while** loops, the postcondition needs to consist of
Finding Proofs of Partial Correctness

- Normally, **Backward reasoning** drives the proof:
  Start to consider the postcondition and how the last statement achieves it

- **Forward reasoning** from the precondition can be useful for simple assignment sequences and for exploration

- For **while** loops, the postcondition needs to consist of
  - the **invariant** of this loop, and
  - the negation of the **loop** condition
Finding Proofs of Partial Correctness

- Normally, **Backward reasoning** drives the proof:
  Start to consider the postcondition and how the last statement achieves it

- **Forward reasoning** from the precondition can be useful for simple assignment sequences and for exploration

- For **while** loops, the postcondition needs to consist of
  - the **invariant** of this loop, and
  - the negation of the **loop** condition

  *Auxiliary variables used in a loop are usually involved in the invariant!*
Finding Proofs of Partial Correctness

• Normally, **Backward reasoning** drives the proof:
  Start to consider the postcondition and how the last statement achieves it

• **Forward reasoning** from the precondition can be useful for simple assignment sequences and for exploration

• For **while** loops, the postcondition needs to consist of
  – the **invariant** of this loop, and
  – the negation of the **loop** condition

  *Auxiliary variables used in a loop are usually involved in the invariant!*

Given a loop “**while b do S od**” and a postcondition **Q**
Finding Proofs of Partial Correctness

- Normally, **Backward reasoning** drives the proof:
  Start to consider the postcondition and how the last statement achieves it

- **Forward reasoning** from the precondition can be useful for simple assignment sequences and for exploration

- For **while** loops, the postcondition needs to consist of
  - the **invariant** of this loop, and
  - the negation of the **loop condition**

**Auxiliary variables used in a loop are usually involved in the invariant!**

Given a loop “**while** b **do** S **od**” and a postcondition \( Q \), use the consequence rule to strengthen \( Q \) to \( Q' \), such that
Finding Proofs of Partial Correctness

- Normally, **Backward reasoning** drives the proof:
  Start to consider the postcondition and how the last statement achieves it

- **Forward reasoning** from the precondition can be useful for simple assignment sequences and for exploration

- For **while** loops, the postcondition needs to consist of
  - the **invariant** of this loop, and
  - the negation of the **loop** condition

*Auxiliary variables used in a loop are usually involved in the invariant!*

Given a loop “**while b do S od**” and a postcondition \( Q \), use the consequence rule to strengthen \( Q \) to \( Q' \), such that

- \( Q' \Rightarrow Q \)
Finding Proofs of Partial Correctness

- Normally, **Backward reasoning** drives the proof:
  Start to consider the postcondition and how the last statement achieves it

- **Forward reasoning** from the precondition can be useful for simple assignment sequences and for exploration

- For **while** loops, the postcondition needs to consist of
  - the **invariant** of this loop, and
  - the negation of the **loop** condition

*Auxiliary variables used in a loop are usually involved in the invariant!*  
Given a loop “**while** b **do** S **od**” and a postcondition \( Q \), use the consequence rule to strengthen \( Q \) to \( Q' \), such that
- \( Q' \Rightarrow Q \) (strengthening)
Finding Proofs of Partial Correctness

• Normally, **Backward reasoning** drives the proof:
  Start to consider the postcondition and how the last statement achieves it

• **Forward reasoning** from the precondition can be useful for simple assignment sequences and for exploration

• For **while** loops, the postcondition needs to consist of
  – the **invariant** of this loop, and
  – the negation of the **loop** condition

  **Auxiliary variables used in a loop are usually involved in the invariant!**

  Given a loop “**while b do S od**” and a postcondition \( Q \), use the consequence rule to strengthen \( Q \) to \( Q' \), such that
  – \( Q' \Rightarrow Q \) (strengthening)
  – \( Q' \) involves all auxiliary variables
Finding Proofs of Partial Correctness

- Normally, **Backward reasoning** drives the proof:
  Start to consider the postcondition and how the last statement achieves it

- **Forward reasoning** from the precondition can be useful for simple assignment sequences and for exploration

- For **while** loops, the postcondition needs to consist of
  - the **invariant** of this loop, and
  - the negation of the **loop** condition

*Auxiliary variables used in a loop are usually involved in the invariant!*

Given a loop “**while** *b* **do** *S* **od**” and a postcondition *Q*, use the consequence rule to strengthen *Q* to *Q’*, such that

- *Q’* ⇒ *Q* (strengthening)
- *Q’* involves all auxiliary variables — **generalisation!**
Finding Proofs of Partial Correctness

- Normally, **Backward reasoning** drives the proof:
  
  Start to consider the postcondition and how the last statement achieves it.

- **Forward reasoning** from the precondition can be useful for simple assignment sequences and for exploration.

- For **while** loops, the postcondition needs to consist of:
  - the **invariant** of this loop, and
  - the negation of the **loop** condition.

  **Auxiliary variables used in a loop are usually involved in the invariant!**

Given a loop “**while b do S od**” and a postcondition $Q$, use the consequence rule to strengthen $Q$ to $Q'$, such that:

- $Q' \Rightarrow Q$ (strengthening)
- $Q'$ involves all auxiliary variables — **generalisation!**
- $Q'$ is of shape $INV \land \neg b$
Simultaneous Assignments

\[
\{P[x_1 \ \backslash \ e_1, \ldots, x_n \ \backslash \ e_n]\}(x_1, \ldots, x_n) := (e_1, \ldots e_n)\{P\}
\]
Simultaneous Assignments

\[ \{P[x_1 \setminus e_1, \ldots, x_n \setminus e_n]\}(x_1, \ldots, x_n) := (e_1, \ldots e_n)\{P\} \]

Examples:

- \( \{1 = 2^0\}(k, n) := (0, 1)\{n = 2^k\} \)
Simultaneous Assignments

\[ \{P[x_1 \setminus e_1, \ldots, x_n \setminus e_n]\}(x_1, \ldots, x_n) := (e_1, \ldots e_n)\{P\} \]

Examples:

- \( \{1 = 2^0\}(k, n) := (0, 1)\{n = 2^k\} \)
- \( \{y \geq x + 2\}(x, y) := (y, x)\{x \geq y + 2\} \)
Simultaneous Assignments

\[ \{P[x_1 \setminus e_1, \ldots, x_n \setminus e_n]\}(x_1, \ldots, x_n) := (e_1, \ldots e_n)\{P\} \]

Examples:

- \( \{1 = 2^0\}(k, n) := (0, 1)\{n = 2^k\} \)
- \( \{y \geq x + 2\}(x, y) := (y, x)\{x \geq y + 2\} \)

Simultaneous assignments

- shorten code
- save auxiliary variables (for example for swapping)
Simultaneous Assignments

\[ \{P[x_1 \setminus e_1, \ldots, x_n \setminus e_n]\}(x_1, \ldots, x_n) := (e_1, \ldots e_n)\{P\} \]

Examples:

- \( \{1 = 2^0\}(k, n) := (0, 1)\{n = 2^k\} \)
- \( \{y \geq x + 2\}(x, y) := (y, x)\{x \geq y + 2\} \)

Simultaneous assignments

- shorten code
- save auxiliary variables (for example for swapping)
- make proofs easier
Simultaneous Assignments

\[ \{P[x_1 \setminus e_1, \ldots, x_n \setminus e_n]\}(x_1, \ldots, x_n) := (e_1, \ldots e_n)\{P\} \]

Examples:

- \(\{1 = 2^0\}(k, n) := (0, 1)\{n = 2^k\}\)

- \(\{y \geq x + 2\}(x, y) := (y, x)\{x \geq y + 2\}\)

Simultaneous assignments

- shorten code

- save auxiliary variables (for example for swapping)

- make proofs easier

- **require simultaneous substitution**
Example Problems (with Simultaneous Assignments)

\[ \{ n \geq 0 \} \quad (y, a, b) := (0, 1, 1) ; \]
\[ \textbf{while} \ y \neq n \ \textbf{do} \ (y, a, b) := (y + 1, b, a + b) \ \textbf{od} \quad \{ a = \text{fib}_n \} \]

Given an \( n \)-element C-like array \( s \), prove partial correctness:

\[ \{ \text{True} \} \]
\[ (i, a) := (0, 0) ; \]
\[ \textbf{while} \ i \neq n \]
\[ \textbf{do} \quad \text{if} \ x = s[i] \]
\[ \quad \text{then} \ (i, a) := (i + 1, a + 1) \]
\[ \textbf{fi} \ \textbf{od} \]
\[ \{ a = \# \{ j : \mathbb{N} \mid s[j] = x \land 0 \leq j < n \} \} \]

What does this program do?
Semantics and Language Design
Semantics and Language Design

- We use the rules of axiomatic semantics to prove properties of programs
Semantics and Language Design

- We use the rules of axiomatic semantics to prove properties of programs
- We use the rules of operational semantics to demonstrate particular execution paths
Semantics and Language Design

- We use the rules of axiomatic semantics to prove properties of programs
- We use the rules of operational semantics to demonstrate particular execution paths

The rules are not sacrosanct!
Semantics and Language Design

- We use the rules of axiomatic semantics to prove properties of programs
- We use the rules of operational semantics to demonstrate particular execution paths

The rules are not sacrosanct!

- Different languages have different rules
Semantics and Language Design

- We use the rules of axiomatic semantics to prove properties of programs
- We use the rules of operational semantics to demonstrate particular execution paths

The rules are not sacrosanct!

- Different languages have different rules
- Such rule sets are specifications of language implementations
Semantics and Language Design

- We use the rules of axiomatic semantics to prove properties of programs
- We use the rules of operational semantics to demonstrate particular execution paths

The rules are not sacrosanct!

- Different languages have different rules
- Such rule sets are specifications of language implementations
- We define the rules for language features and extensions
Semantics and Language Design

- We use the rules of axiomatic semantics to prove properties of programs
- We use the rules of operational semantics to demonstrate particular execution paths

The rules are not sacrosanct!

- Different languages have different rules
- Such rule sets are specifications of language implementations
- We define the rules for language features and extensions
- We justify the rules against different presentations of the defined features
Semantics and Language Design

- We use the rules of axiomatic semantics to prove properties of programs
- We use the rules of operational semantics to demonstrate particular execution paths

The rules are not sacrosanct!

- Different languages have different rules
- Such rule sets are specifications of language implementations
- We define the rules for language features and extensions
- We justify the rules against different presentations of the defined features
- We derive the rules e.g. from source-to-source translations
Fibonacci

\[
\{ n \geq 0 \} \quad (y, a, b) := (0, 1, 1) ; \\
\textbf{while} \ y \neq n \ \textbf{do} \ (y, a, b) := (y + 1, b, a + b) \ \textbf{od} \quad \{ a = \text{fib}_n \}
\]
Fibonacci

\[ \{ n \geq 0 \} \quad (y, a, b) := (0, 1, 1) ; \]

\[ \textbf{while} \ y \neq n \ \textbf{do} \ (y, a, b) := (y + 1, b, a + b) \ \textbf{od} \quad \{ a = \text{fib}_n \} \]

\( \iff \langle \ ( \text{right consequence} ) \ \rangle \)

\[ \{ n \geq 0 \} \ \mathcal{P} \ \{ a = \text{fib}_y \wedge b = \text{fib}_{y+1} \wedge y = n \} \]

\[ \wedge (a = \text{fib}_y \wedge b = \text{fib}_{y+1} \wedge y = n \Rightarrow a = \text{fib}_n) \]
Fibonacci

\{n \geq 0\} \quad (y, a, b) := (0, 1, 1) ;

\textbf{while} y \neq n \textbf{ do } (y, a, b) := (y + 1, b, a + b) \textbf{ od} \quad \{a = \text{fib}_n\}

\Leftrightarrow \langle \text{ ( right consequence ) } \rangle

\{n \geq 0\} \quad P \quad \{a = \text{fib}_y \land b = \text{fib}_{y+1} \land y = n\}

\land (a = \text{fib}_y \land b = \text{fib}_{y+1} \land y = n \Rightarrow a = \text{fib}_n)

\Leftrightarrow \langle \text{ ( sequence , logic ) } \rangle

\{n \geq 0\} \quad (y, a, b) := (0, 1, 1) \quad \{a = \text{fib}_y \land b = \text{fib}_{y+1}\} \land

\{a = \text{fib}_y \land b = \text{fib}_{y+1}\} \textbf{while} y \neq n \textbf{ do } A \textbf{ od} \{a = \text{fib}_y \land b = \text{fib}_{y+1} \land y = n\}

\land \text{ True}
Fibonacci (ctd.)

\[\langle (\text{left consequence}, \ \textbf{while}) \rangle \]

\[(n \geq 0 \implies 1 = \text{fib}_0 \land 1 = \text{fib}_{0+1}) \]
\[\land \{1 = \text{fib}_0 \land 1 = \text{fib}_{0+1}\} \ (y, a, b) := (0, 1, 1) \{a = \text{fib}_y \land b = \text{fib}_{y+1}\} \]
\[\land \{a = \text{fib}_y \land b = \text{fib}_{y+1} \land y \neq n\} \ (y, a, b) := (y + 1, b, a + b) \]
\[\{a = \text{fib}_y \land b = \text{fib}_{y+1}\}\]
Fibonacci (ctd.)

\[
\Leftrightarrow \langle \mathrm{(left\ consequence, \ while) } \rangle \\
(n \geq 0 \Rightarrow 1 = \text{fib}_0 \land 1 = \text{fib}_{0+1}) \\
\land \{1 = \text{fib}_0 \land 1 = \text{fib}_{0+1}\} \ (y, a, b) := (0, 1, 1) \ \{a = \text{fib}_y \land b = \text{fib}_{y+1}\} \\
\land \{a = \text{fib}_y \land b = \text{fib}_{y+1} \land y \neq n\} \ (y, a, b) := (y + 1, b, a + b) \\
\quad \{a = \text{fib}_y \land b = \text{fib}_{y+1}\} \\
\Leftrightarrow \langle \mathrm{(arithmetic, \ assignment, \ left\ consequence) } \rangle \\
\text{True} \land \text{True} \\
\land (a = \text{fib}_y \land b = \text{fib}_{y+1} \land y \neq n \Rightarrow b = \text{fib}_{y+1} \land a + b = \text{fib}_{(y+1)+1}) \\
\land \{b = \text{fib}_{y+1} \land a + b = \text{fib}_{(y+1)+1}\} \ (y, a, b) := (y + 1, b, a + b) \\
\quad \{a = \text{fib}_y \land b = \text{fib}_{y+1}\} \\
\]
Fibonacci (ctd.)

\[ \langle \text{(left consequence , while )} \rangle \]
\[ (n \geq 0 \Rightarrow 1 = \text{fib}_0 \land 1 = \text{fib}_{0+1}) \]
\[ \land \{1 = \text{fib}_0 \land 1 = \text{fib}_{0+1}\} (y, a, b) := (0, 1, 1) \{a = \text{fib}_y \land b = \text{fib}_{y+1}\} \]
\[ \land \{a = \text{fib}_y \land b = \text{fib}_{y+1} \land y \neq n\} (y, a, b) := (y + 1, b, a + b) \]
\[ \{a = \text{fib}_y \land b = \text{fib}_{y+1}\} \]

\[ \langle \text{(arithmetic , assignment , left consequence )} \rangle \]
\[ \text{True} \land \text{True} \]
\[ \land (a = \text{fib}_y \land b = \text{fib}_{y+1} \land y \neq n \Rightarrow b = \text{fib}_{y+1} \land a + b = \text{fib}_{(y+1)+1}) \]
\[ \land \{b = \text{fib}_{y+1} \land a + b = \text{fib}_{(y+1)+1}\} (y, a, b) := (y + 1, b, a + b) \]
\[ \{a = \text{fib}_y \land b = \text{fib}_{y+1}\} \]

\[ \langle \text{(arithmetic , assignment )} \rangle \]
\[ \text{True} \land \text{True} \]
Array Traversal

Given an $n$-element C-like array $s$, prove partial correctness:

\{True\}
$(i, a) := (0, 0)$;
while $i \neq n$
do if $x = s[i]$
then $(i, a) := (i + 1, a + 1)$
fi od
\{ $a = \#\{j : \mathbb{N} \mid s[j] = x \land 0 \leq j < n\}$ \}

\{True\} $P$ \{ $a = \#\{j : \mathbb{N} \mid s[j] = x \land 0 \leq j < n\}$ \}
\iff \langle ( right consequence ) \rangle
\{True\} Init ; W \{ $a = \#\{j : \mathbb{N} \mid s[j] = x \land 0 \leq j < i\} \land i = n$ \}
\land ((a = \#\{j : \mathbb{N} \mid s[j] = x \land 0 \leq j < i\} \land i = n) \Rightarrow Post)\\
\iff \langle ( sequence , logic ) \rangle
\[
\{ \text{True} \} \quad (i, a) := (0, 0) \quad \{ a = \# \{ j : \mathbb{N} \mid s[j] = x \land 0 \leq j < i \} \} \\
\wedge \{ a = \# \{ j : \mathbb{N} \mid s[j] = x \land 0 \leq j < i \} \} W_\text{True} \\
\{ a = \# \{ j : \mathbb{N} \mid s[j] = x \land 0 \leq j < i \} \} \wedge i = n \\
\wedge \text{True} \\
\leftarrow \langle ( \text{left consequence, while}) \rangle \\
(\text{True} \Rightarrow 0 = \# \{ j : \mathbb{N} \mid s[j] = x \land 0 \leq j < 0 \}) \\
\wedge \{ 0 = \# \{ j : \mathbb{N} \mid s[j] = x \land 0 \leq j < 0 \} \} (i, a) := (0, 0) \\
\{ a = \# \{ j : \mathbb{N} \mid s[j] = x \land 0 \leq j < i \} \} \\
\wedge \{ a = \# \{ j : \mathbb{N} \mid s[j] = x \land 0 \leq j < i \} \} \wedge i \neq n \} B \\
\{ a = \# \{ j : \mathbb{N} \mid s[j] = x \land 0 \leq j < i \} \} \\
\leftarrow \langle ( \text{logic and arithmetic, assignment, conditional}) \rangle \\
\text{True} \wedge \text{True} \\
\wedge \{ a = \# \{ j : \mathbb{N} \mid s[j] = x \land 0 \leq j < i \} \wedge i \neq n \wedge x = s[i] \} \\
(i, a) := (i + 1, a + 1) \{ a = \# \{ j : \mathbb{N} \mid s[j] = x \land 0 \leq j < i \} \} \\
\wedge ((a = \# \{ j : \mathbb{N} \mid s[j] = x \land 0 \leq j < i \} \wedge i \neq n \wedge x \neq s[i]) \Rightarrow a = \# \{ j : \mathbb{N} \mid s[j] = x \land 0 \leq j < i \})}
\[ \left( \text{left consequence, logic} \right) \]
\[
((a = \# \{ j : \mathbb{N} \mid s[j] = x \land 0 \leq j < i \} \land i \neq n \land x = s[i])
\Rightarrow a + 1 = \# \{ j : \mathbb{N} \mid s[j] = x \land 0 \leq j < i + 1 \})
\land \{ a + 1 = \# \{ j : \mathbb{N} \mid s[j] = x \land 0 \leq j < i + 1 \} \} (i, a) := (i + 1, a + 1)
\land \text{True}
\]
\[ \left( \text{logic, assignment} \right) \]
\[
\text{True} \land \text{True}
\]
Array Traversal

Given an $n$-element C-like array $s$, prove partial correctness:

$\{ \text{True} \}
\{ (i, a) := (0, 0) ; \}
\text{while } i \neq n
\text{do } \text{if } x = s[i]
\quad \text{then } (i, a) := (i + 1, a + 1)
\text{fi od}
\{ a = \#\{ j : \mathbb{N} \mid s[j] = x \land 0 \leq j < n \} \}

What does this program do?
Integer Square Root

\{n \geq 0\}

(a, b) := (0, n + 1) ;

while \ a + 1 \neq b

\ do \ d := (a + b)/2 ;

\ if \ d \cdot d \leq n

\ then \ a := \ d

\ else \ b := \ d

\ fi

\ od

\{a^2 \leq n < (a + 1)^2\}

All variables and expressions are of type integer.
Exercise 11.1(a, b)

True

\[
x \geq -5 \Rightarrow x \geq -6 \quad \text{(arith.)}
\]

True

\[
x \geq -5 \Rightarrow 5 - x \leq 11 \quad \text{(arith.)}
\]

\[
\begin{align*}
\{5 - x \leq 11\} z := 5 - x \quad \{z \leq 11\} \\
\{x \geq -5\} z := 5 - x \quad \{z \leq 11\}
\end{align*}
\]

(assign .) (conseq .)
Exercise 11.1(a, b)

True

\[ x \geq -5 \Rightarrow x \geq -6 \] (arith.)

\[ x \geq -5 \Rightarrow 5 - x \leq 11 \] (arith.)

\[ \{5 - x \leq 11\} z := 5 - x \{z \leq 11\} \] (assign.)

\[ \{x \geq -5\} z := 5 - x \{z \leq 11\} \] (conseq.)

\[ \{x \geq -5\} z := 5 - x \{z \leq 11 \land x \geq -7\} \]

\[ \Leftrightarrow \langle (\text{left consequence}) \rangle \]

\[ (x \geq -5 \Rightarrow 5 - x \leq 11 \land x \geq -7) \]

\[ \land \{5 - x \leq 11 \land x \geq -7\} z := 5 - x \{z \leq 11 \land x \geq -7\} \]

\[ \Leftrightarrow \langle (\text{arithmetic, assignment}) \rangle \]

\[ (x \geq -5 \Rightarrow x \geq -6 \land x \geq -7) \land \text{True} \]

\[ \Leftrightarrow \langle (\text{arithmetic}) \rangle \]

True
Exercise 11.1(c)

\[ \{x \geq -5\} \ z := 5 - x \ \{z \leq 11 \land x \geq -3\} \]
Exercise 11.1(c)

\[ \{ x \geq -5 \} \ z := 5 - x \ \{ z \leq 11 \land x \geq -3 \} \]

Using operational semantics, we can prove a counterexample:

\[ \{ x \mapsto -5 \}(z := 5 - x) \Rightarrow \]
Exercise 11.1(c)

\[ \{x \geq -5\} z := 5 - x \{z \leq 11 \land x \geq -3\} \]

Using operational semantics, we can prove a counterexample:

\[
\begin{align*}
\{x \mapsto -5\}(5 - x) & \Rightarrow \\
\{x \mapsto -5\}(z := 5 - x) & \Rightarrow
\end{align*}
\]
Exercise 11.1(c)

\[ \{x \geq -5\} \ z := 5 - x \ \{z \leq 11 \land x \geq -3\} \]

Using operational semantics, we can prove a counterexample:

\[
\begin{align*}
\{x \mapsto -5\}(5) & \Rightarrow \\
\{x \mapsto -5\}(x) & \Rightarrow \\
\{x \mapsto -5\}(5 - x) & \Rightarrow \\
\{x \mapsto -5\}(z := 5 - x) & \Rightarrow
\end{align*}
\]
Exercise 11.1(c)

\[ \{ x \geq -5 \} \ z := 5 - x \ \{ z \leq 11 \land x \geq -3 \} \]

Using operational semantics, we can prove a counterexample:

\[
\begin{array}{c}
\{ x \mapsto -5 \}(5) \Rightarrow 5 \\
\hline
\{ x \mapsto -5 \}(x) \Rightarrow \\
\hline
\{ x \mapsto -5 \}(5 - x) \Rightarrow \\
\hline
\{ x \mapsto -5 \}(z := 5 - x) \Rightarrow \\
\end{array}
\]
Exercise 11.1(c)

\[ \{x \geq -5\} \ z := 5 - x \ \{z \leq 11 \land x \geq -3\} \]

Using operational semantics, we can prove a counterexample:

\[
\begin{align*}
\{x \mapsto -5\}(5) \Rightarrow 5 & \quad \quad \{x \mapsto -5\}(x) \Rightarrow -5 \\
\{x \mapsto -5\}(5 - x) \Rightarrow \\
\{x \mapsto -5\}(z := 5 - x) \Rightarrow 
\end{align*}
\]
Exercise 11.1(c)

\[ \{ x \geq -5 \} \ z := 5 - x \ \{ z \leq 11 \land x \geq -3 \} \]

Using operational semantics, we can prove a counterexample:

\[ \{ x \mapsto -5 \}(5) \Rightarrow 5 \]
\[ \{ x \mapsto -5 \}(x) \Rightarrow -5 \quad \]
\[ \frac{\{ x \mapsto -5 \}(5 - x) \Rightarrow 10}{\{ x \mapsto -5 \}(z := 5 - x) \Rightarrow} \]
Exercise 11.1(c)

\[ \{x \geq -5\} z := 5 - x \ {z \leq 11 \land x \geq -3}\]

Using operational semantics, we can prove a counterexample:

\[
\begin{align*}
\{x \mapsto -5\}(5) &\Rightarrow 5 & \{x \mapsto -5\}(x) &\Rightarrow -5 \\
\{x \mapsto -5\}(5 - x) &\Rightarrow 10 \\
\{x \mapsto -5\}(z := 5 - x) &\Rightarrow \{x \mapsto -5, z \mapsto 10\}
\end{align*}
\]
Exercise 11.1(c)

\[ \{x \geq -5\} \ z := 5 - x \ \{z \leq 11 \land x \geq -3\} \]

Using operational semantics, we can prove a counterexample:

\[
\begin{align*}
\{x \mapsto -5\}(5) & \Rightarrow 5 & \{x \mapsto -5\}(x) & \Rightarrow -5 \\
\{x \mapsto -5\}(5 - x) & \Rightarrow 10 \\
\{x \mapsto -5\}(z := 5 - x) & \Rightarrow \{x \mapsto -5, z \mapsto 10\}
\end{align*}
\]

This last state clearly does not satisfy \(\{z \leq 11 \land x \geq -3\}\)
Exercise 11.1(d)

\[ \{x \geq -5\} z := 5 - x ; x := x + 2 \{z \leq 11 \land x \geq -3\} \]
Exercise 11.1(d)

\[
\begin{align*}
\{x \geq -5\} & \quad z := 5 - x ; \quad x := x + 2 \quad \{z \leq 11 \land x \geq -3\} \\
\iff \quad \langle \text{sequence rule} \rangle \\
\{x \geq -5\} & \quad z := 5 - x \quad \{z \leq 11 \land x + 2 \geq -3\} \\
\land \quad \{z \leq 11 \land x + 2 \geq -3\} & \quad x := x + 2 \quad \{z \leq 11 \land x \geq -3\}
\end{align*}
\]
Exercise 11.1(d)

\[
\{x \geq -5\} z := 5 - x ; x := x + 2 \{z \leq 11 \land x \geq -3\}
\]
\[
\Leftrightarrow \langle \text{sequence rule} \rangle \\
\{x \geq -5\} z := 5 - x \{z \leq 11 \land x + 2 \geq -3\}
\]
\[
\land \{z \leq 11 \land x + 2 \geq -3\} x := x + 2 \{z \leq 11 \land x \geq -3\}
\]
\[
\Leftrightarrow \langle \text{left consequence, assignment} \rangle \\
(x \geq -5 \Rightarrow (5 - x \leq 11 \land x + 2 \geq -3))
\]
\[
\land \{5 - x \leq 11 \land x + 2 \geq -3\} z := 5 - x \{z \leq 11 \land x + 2 \geq -3\}
\]
\[
\land \text{True}
\]
Exercise 11.1(d)

\[
\{ x \geq -5 \} \quad z := 5 - x ; \quad x := x + 2 \quad \{ z \leq 11 \land x \geq -3 \}
\]
\[
\iff \quad \langle \text{sequence rule} \rangle
\]
\[
\{ x \geq -5 \} \quad z := 5 - x \quad \{ z \leq 11 \land x + 2 \geq -3 \} \quad x := x + 2 \quad \{ z \leq 11 \land x \geq -3 \}
\]
\[
\iff \quad \langle \text{left consequence, assignment} \rangle
\]
\[
(x \geq -5 \Rightarrow (5 - x \leq 11 \land x + 2 \geq -3))
\]
\[
\land \quad \{ 5 - x \leq 11 \land x + 2 \geq -3 \} \quad z := 5 - x \quad \{ z \leq 11 \land x + 2 \geq -3 \}
\]
\[
\land \quad \text{True}
\]
\[
\iff \quad \langle \text{logic and arithmetic, assignment} \rangle
\]
\[
(x \geq -5 \Rightarrow x \geq -6) \land (x \geq -5 \Rightarrow x \geq -5) \land \text{True}
\]
Exercise 11.1(d)

\{x \geq -5\} z := 5 - x ; x := x + 2 \{z \leq 11 \land x \geq -3\}

\iff \langle \text{sequence rule} \rangle

\{x \geq -5\} z := 5 - x \{z \leq 11 \land x + 2 \geq -3\}
\land \{z \leq 11 \land x + 2 \geq -3\} x := x + 2 \{z \leq 11 \land x \geq -3\}

\iff \langle \text{left consequence, assignment} \rangle

(x \geq -5 \Rightarrow (5 - x \leq 11 \land x + 2 \geq -3))
\land \{5 - x \leq 11 \land x + 2 \geq -3\} z := 5 - x \{z \leq 11 \land x + 2 \geq -3\}
\land \text{True}

\iff \langle \text{logic and arithmetic, assignment} \rangle

(x \geq -5 \Rightarrow x \geq -6) \land (x \geq -5 \Rightarrow x \geq -5) \land \text{True}

\iff \langle \text{arithmetic} \rangle

\text{True}
Exercise 11.1(e)

\[ \{ x \geq - 5 \} \ z := 5 - x ; \ x := x + z \ \{ z \leq 11 \land x = 2 \} \]
Exercise 11.1(e)

\[ \{x \geq -5\} \overset{z := 5 - x; x := x + z}{\Rightarrow} \{z \leq 11 \land x = 2\} \]

We again use operational semantics (expression evaluation not shown) to prove a counterexample:

\[
\begin{align*}
\sigma_1 &= \{x \mapsto 0\} \\
\sigma_2 &= \{x \mapsto 0, z \mapsto 5\} \\
\sigma_3 &= \{x \mapsto 5, z \mapsto 5\}
\end{align*}
\]

\[
\sigma_1(5 - x) \Rightarrow 5 \quad \quad \sigma_2(x + z) \Rightarrow 5
\]

\[
\sigma_1(z := 5 - x) \Rightarrow \sigma_2 \quad \quad \sigma_2(x := x + z) \Rightarrow \sigma_3
\]

\[
\sigma_1(z := 5 - x; x := x + z) \Rightarrow \sigma_3
\]

Although \(\sigma_1 = \{x \mapsto 0\}\) satisfies the precondition \(\{x \geq -5\}\), the final state \(\sigma_3 = \{x \mapsto 5, z \mapsto 5\}\) does not satisfy the postcondition \(\{z \leq 11 \land x = 2\}\).
Exercise 11.1(f)

\[
\{ z = \text{abs}(x) \} \text{ if } x \geq 0 \text{ then } z := -z \text{ fi } \{xz = -x^2\}
\]
Exercise 11.1(f)

one-sided conditional:

\[
\{ z = \text{abs}(x) \} \text{ if } x \geq 0 \text{ then } z := -z \text{ fi } \{xz = -x^2\}
\]
Exercise 11.1(f)

Rule for one-sided conditional:

\[
\frac{\{P \land b\} \ S_1 \ \{Q\} \quad P \land \neg b \Rightarrow Q}{\{P\} \ \text{if } b \ \text{then } S_1 \ \text{fi } \{Q\}}
\]

\[\{z = \text{abs}(x)\} \ \text{if } x \geq 0 \ \text{then } z := -z \ \text{fi } \{xz = -x^2\}\]
Exercise 11.1(f)

Rule for one-sided conditional:

\[
\frac{\{P \land b\} \ S_1 \ \{Q\}}{P \land \neg b \Rightarrow Q}
\]

\[
\{P\} \text{ if } b \text{ then } S_1 \text{ fi } \{Q\}
\]

\[
\begin{align*}
\{z = abs(x)\} \text{ if } x \geq 0 \text{ then } z := -z \text{ fi } \{xz = -x^2\} \\
\iff \langle \text{ one-sided conditional } \rangle \\
\{z = abs(x) \land x \geq 0\} \ z := -z \ \{xz = -x^2\} \\
\land (z = abs(x) \land x < 0 \Rightarrow xz = -x^2)
\end{align*}
\]
Exercise 11.1(f)

Rule for one-sided conditional:

\[
\frac{\{ P \land b \} \ S_1 \ \{ Q \} \quad P \land \neg b \Rightarrow Q}{\{ P \} \ \textbf{if} \ b \ \textbf{then} \ S_1 \ \textbf{fi} \ \{ Q \}}
\]

\[
\{ z = \text{abs}(x) \} \ \textbf{if} \ x \geq 0 \ \textbf{then} \ z := -z \ \textbf{fi} \ \{ xz = -x^2 \}
\]

\[
\iff \ \langle \text{one-sided conditional} \rangle
\]

\[
\{ z = \text{abs}(x) \land x \geq 0 \} \ z := -z \ \{ xz = -x^2 \}
\]

\[
\land (z = \text{abs}(x) \land x < 0 \Rightarrow xz = -x^2)
\]

\[
\iff \ \langle \text{left consequence, arithmetic} \rangle
\]

\[
(z = \text{abs}(x) \land x \geq 0 \Rightarrow x \cdot (-z) = -x^2)
\]

\[
\land \{ x \cdot (-z) = -x^2 \} \ z := -z \ \{ xz = -x^2 \}
\]

\[
\land (z = -x \Rightarrow xz = -x^2)
\]
Exercise 11.1(f)

Rule for one-sided conditional:

\[
\begin{align*}
&\frac{\{P \land b\} \quad S_1 \quad \{Q\}}{P \land \neg b \Rightarrow Q} \\
&\quad \{P\} \text{ if } b \text{ then } S_1 \text{ fi } \{Q\}
\end{align*}
\]

\[
\{z = \text{abs}(x)\} \text{ if } x \geq 0 \text{ then } z := -z \text{ fi } \{xz = -x^2\}
\]

\[
\Leftrightarrow \langle \text{one-sided conditional} \rangle
\]

\[
\{z = \text{abs}(x) \land x \geq 0\} \quad z := -z \quad \{xz = -x^2\}
\]

\[
\quad \land \ (z = \text{abs}(x) \land x < 0 \Rightarrow xz = -x^2)
\]

\[
\Leftrightarrow \langle \text{left consequence, arithmetic} \rangle
\]

\[
(z = \text{abs}(x) \land x \geq 0 \Rightarrow x \cdot ( -z ) = -x^2)
\]

\[
\quad \land \ {x \cdot ( -z ) = -x^2} \quad z := -z \quad \{xz = -x^2\}
\]

\[
\quad \land \ (z = -x \Rightarrow xz = -x^2)
\]

\[
\Leftrightarrow \langle \text{arithmetic, assignment, arithmetic} \rangle
\]

\[
(z = x \Rightarrow x \cdot ( -z ) = -x^2) \land \text{True} \land \text{True}
\]
Exercise 11.1(f)

**Rule for one-sided conditional:**

\[
\begin{align*}
\{ P \land b \} & \quad S_1 \quad \{ Q \} \\
\Rightarrow & \\
\{ P \} & \quad \text{if } b \text{ then } S_1 \quad \text{fi} \quad \{ Q \}
\end{align*}
\]

\[
\{ z = \text{abs}(x) \} \quad \text{if } x \geq 0 \text{ then } z := -z \land \{ xz = -x^2 \}
\]

\[
\Leftarrow \quad \langle \text{one-sided conditional} \rangle \\
\{ z = \text{abs}(x) \land x \geq 0 \} \quad z := -z \land \{ xz = -x^2 \}
\]

\[
\land (z = \text{abs}(x) \land x < 0 \Rightarrow xz = -x^2)
\]

\[
\Leftarrow \quad \langle \text{left consequence , arithmetic} \rangle \\
(z = \text{abs}(x) \land x \geq 0 \Rightarrow x \cdot (-z) = -x^2)
\]

\[
\land \{ x \cdot (-z) = -x^2 \} \quad z := -z \land \{ xz = -x^2 \}
\]

\[
\land (z = -x \Rightarrow xz = -x^2)
\]

\[
\Leftarrow \quad \langle \text{arithmetic , assignment , arithmetic} \rangle \\
(z = x \Rightarrow x \cdot (-z) = -x^2) \land \text{True} \land \text{True}
\]

\[
\Leftarrow \quad \langle \text{arithmetic} \rangle \\
\text{True}
\]
Exercise 11.1(g)

\[
\{ z = 0 \} \quad \textbf{if} \; x = 0 \; \textbf{then} \; w := \text{True} \; \textbf{else} \; z := 1/x \; \textbf{fi} \; \{ \neg w \rightarrow xz = 1 \}\]
Exercise 11.1(g)

\[
\{ z = 0 \} \textbf{ if } x = 0 \textbf{ then } w := \text{True} \textbf{ else } z := 1/x \textbf{ fi } \{ \neg w \rightarrow xz = 1 \}
\]

\[\iff \langle \text{ conditional } \rangle \]
\[
\{ z = 0 \land x = 0 \} \quad w := \text{True} \quad \{ \neg w \rightarrow xz = 1 \}
\]
\[\land \quad \{ z = 0 \land x \neq 0 \} \quad z := 1/x \quad \{ \neg w \rightarrow xz = 1 \}\]
Exercise 11.1(g)

\{z = 0\} \textbf{if } x = 0 \textbf{ then } w := \text{True} \textbf{ else } z := \frac{1}{x} \textbf{ fi } \{\neg w \rightarrow xz = 1\}

\Leftarrow \langle \text{conditional} \rangle

\{z = 0 \land x = 0\} w := \text{True} \{\neg w \rightarrow xz = 1\}
\land \{z = 0 \land x \neq 0\} z := \frac{1}{x} \{\neg w \rightarrow xz = 1\}

\Leftarrow \langle \text{left consequence, left consequence} \rangle

(z = 0 \land x = 0 \Rightarrow (\neg \text{True} \rightarrow xz = 1))
\land \{\neg \text{True} \rightarrow xz = 1\} w := \text{True} \{\neg w \rightarrow xz = 1\}
\land (z = 0 \land x \neq 0 \Rightarrow (\neg w \rightarrow x \cdot (1/x) = 1))
\land \{\neg w \rightarrow x \cdot (1/x) = 1\} z := \frac{1}{x} \{\neg w \rightarrow xz = 1\}
Exercise 11.1(g)

\[
\{ z = 0 \} \textbf{if} \ x = 0 \textbf{then} \ w := \text{True} \textbf{else} \ z := 1/x \ \{ \neg w \rightarrow xz = 1 \} \\
\iff \langle \text{conditional} \rangle \\
\{ z = 0 \land x = 0 \} \ w := \text{True} \ \{ \neg w \rightarrow xz = 1 \} \\
\land \{ z = 0 \land x \neq 0 \} \ z := 1/x \ \{ \neg w \rightarrow xz = 1 \} \\
\iff \langle \text{left consequence , left consequence} \rangle \\
(z = 0 \land x = 0 \Rightarrow (\neg \text{True} \rightarrow xz = 1)) \\
\land \{ \neg \text{True} \rightarrow xz = 1 \} \ w := \text{True} \ \{ \neg w \rightarrow xz = 1 \} \\
\land (z = 0 \land x \neq 0 \Rightarrow (\neg w \rightarrow x \cdot (1/x) = 1)) \\
\land \{ \neg w \rightarrow x \cdot (1/x) = 1 \} \ z := 1/x \ \{ \neg w \rightarrow xz = 1 \} \\
\iff \langle \text{logic , assignment , logic , assignment} \rangle \\
(z = 0 \land x = 0 \Rightarrow (\text{False} \rightarrow xz = 1)) \land \text{True} \land (x \neq 0 \Rightarrow x \cdot (1/x) = 1)
Exercise 11.1(g)

\{z = 0\} \textbf{if} x = 0 \textbf{then} w := \text{True} \textbf{else} z := 1/x \textbf{fi} \ \{\neg w \rightarrow xz = 1\}

\iffalse \text{conditional}\fffalse
\{z = 0 \land x = 0\} \ w := \text{True} \ \{\neg w \rightarrow xz = 1\}
\land \ \{z = 0 \land x \neq 0\} \ z := 1/x \ \{\neg w \rightarrow xz = 1\}
\fi

\iffalse \text{left consequence, left consequence}\fffalse
(z = 0 \land x = 0 \Rightarrow (\neg \text{True} \rightarrow xz = 1))
\land \ \{\neg \text{True} \rightarrow xz = 1\} \ w := \text{True} \ \{\neg w \rightarrow xz = 1\}
\land \ (z = 0 \land x \neq 0 \Rightarrow (\neg w \rightarrow x \cdot (1/x) = 1))
\land \ \{\neg w \rightarrow x \cdot (1/x) = 1\} \ z := 1/x \ \{\neg w \rightarrow xz = 1\}
\fi

\iffalse \text{logic, assignment, logic, assignment}\fffalse
(z = 0 \land x = 0 \Rightarrow (\text{False} \rightarrow xz = 1)) \land \text{True} \land (x \neq 0 \Rightarrow x \cdot (1/x) = 1)
\fi

\iffalse \text{ex falso quodlibet, arithmetic}\fffalse
(z = 0 \land x = 0 \Rightarrow \text{True}) \land \text{True}
\fi
Exercise 11.1(g)

\{z = 0\} \textbf{if } x = 0 \textbf{ then } w := \text{True}\textbf{ else } z := 1/x \text{ fi} \{\neg w \rightarrow xz = 1\}

\Leftarrow \langle \text{conditional} \rangle

\{z = 0 \land x = 0\} w := \text{True} \{\neg w \rightarrow xz = 1\}
\land \{z = 0 \land x \neq 0\} z := 1/x \{\neg w \rightarrow xz = 1\}

\Leftarrow \langle \text{left consequence, left consequence} \rangle

(z = 0 \land x = 0 \Rightarrow (\neg \text{True} \rightarrow xz = 1))
\land \{\neg \text{True} \rightarrow xz = 1\} w := \text{True} \{\neg w \rightarrow xz = 1\}
\land (z = 0 \land x \neq 0 \Rightarrow (\neg w \rightarrow x \cdot (1/x) = 1))
\land \{\neg w \rightarrow x \cdot (1/x) = 1\} z := 1/x \{\neg w \rightarrow xz = 1\}

\Leftarrow \langle \text{logic, assignment, logic, assignment} \rangle

(z = 0 \land x = 0 \Rightarrow (\text{False} \rightarrow xz = 1)) \land \text{True} \land (x \neq 0 \Rightarrow x \cdot (1/x) = 1)

\Leftarrow \langle \text{ex falso quodlibet, arithmetic} \rangle

(z = 0 \land x = 0 \Rightarrow \text{True}) \land \text{True}

\Leftarrow \langle \text{logic} \rangle

\text{True}
repeat ... until ...

Operational Semantics:
repeat \ldots \text{ until} \ldots

\textit{Operational Semantics:}

\[
\sigma(s) \Rightarrow \sigma_1 \quad \sigma_1(b) \Rightarrow \text{True}
\]

\[
\sigma(\text{repeat } s \text{ until } b) \Rightarrow \sigma_1
\]
repeat ... until ...

Operational Semantics:

$$\sigma(s) \Rightarrow \sigma_1 \quad \sigma_1(b) \Rightarrow \text{True}$$

$$\sigma(\text{repeat } s \text{ until } b) \Rightarrow \sigma_1$$

$$\sigma(s) \Rightarrow \sigma_1 \quad \sigma_1(b) \Rightarrow \text{False} \quad \sigma_1(\text{repeat } s \text{ until } b) \Rightarrow \sigma_2$$

$$\sigma(\text{repeat } s \text{ until } b) \Rightarrow \sigma_2$$
repeat \ldots until \ldots

**Operational Semantics:**

\[
\begin{align*}
\sigma(s) & \Rightarrow \sigma_1 \\
\sigma_1(b) & \Rightarrow \text{True} \\
\sigma(\text{repeat } s \text{ until } b) & \Rightarrow \sigma_1
\end{align*}
\]

\[
\begin{align*}
\sigma(s) & \Rightarrow \sigma_1 \\
\sigma_1(b) & \Rightarrow \text{False} \\
\sigma_1(\text{repeat } s \text{ until } b) & \Rightarrow \sigma_2
\end{align*}
\]

\[
\sigma(\text{repeat } s \text{ until } b) \Rightarrow \sigma_2
\]

**Axiomatic Semantics:**

\[
\{\text{INV}\} \ S \ \{\text{INV}\}
\]

\[
\{\text{INV}\} \ \text{repeat } s \ \text{until } b \ \{\text{INV} \ \land \ b\}
\]
Proper for-Loops — Operational Semantics

*Proper for*-loop: Number of iterations determined at loop entrance:

- the upper limit cannot be changed
- the loop variable cannot be advanced or reset
Proper for-Loops — Operational Semantics

Proper for-loop: Number of iterations determined at loop entrance:

• the upper limit cannot be changed
• the loop variable cannot be advanced or reset

Operational Semantics:

\[
\forall i : \mathbb{N} \mid b \leq i \leq e \bullet
\begin{align*}
\sigma(beg) &\Rightarrow b \\
\sigma(end) &\Rightarrow e \\
\sigma_b &= \sigma \\
(\sigma_i \oplus \{v \mapsto i\})(s) &\Rightarrow \sigma_{i+1}
\end{align*}
\]

\[
\sigma(\textbf{for } v := \textit{beg to end do } s \textit{ od}) \Rightarrow \sigma_{\text{max}(b,e+1)}
\]
Proper for-Loops — Operational Semantics

**Proper for**-loop: Number of iterations determined at loop entrance:

- the upper limit cannot be changed
- the loop variable cannot be advanced or reset

**Operational Semantics:**

\[
\forall i : \mathbb{N} \mid b \leq i \leq e \bullet
\]

\[
\begin{align*}
\sigma(beg) & \Rightarrow b \\
\sigma(\text{end}) & \Rightarrow e \\
\sigma_b & = \sigma \\
(\sigma_i \oplus \{v \mapsto i\})(s) & \Rightarrow \sigma_{i+1}
\end{align*}
\]

\[
\sigma(\text{for } v := \text{beg to end do } s \text{ od}) \Rightarrow \sigma_{\text{max}(b, e+1)}
\]

- This resets the loop variable at the beginning of each iteration
- A static test can prevent assignments to the loop variable
Exercise 11.2

\{ \text{True} \}
\( (i, j, s) := (0, 0, 0) \);

\textbf{while } i \neq n \textbf{ do }

\textbf{if } i = j \\
\textbf{then } (i, j, s) := (i + 1, 0, s + 1) \\
\textbf{else } (j, s) := (j + 1, s + 2) \\
\textbf{fi}

\textbf{od}
\\{ s = n^2 + 2j \}
Exercise 11.2 — Proof

\{\text{True}\} (i, j, s) := (0, 0, 0); \textbf{while } i \neq n \textbf{ do } B \textbf{ od } \{s = n^2 + 2j\}
Exercise 11.2 — Proof

\{\text{True}\} (i, j, s) := (0, 0, 0) \ ; \textbf{while} \ i \neq n \ \textbf{do} \ B \ \textbf{od} \ \{s = n^2 + 2j\}

\Leftarrow \langle \text{right consequence} \rangle

\{\text{True}\} (i, j, s) := (0, 0, 0) \ ; \textbf{while} \ i \neq n \ \textbf{do} \ B \ \textbf{od} \ \{s = i^2 + 2j \land i = n\}

\land (s = i^2 + 2j \land i = n \Rightarrow s = n^2 + 2j)
Exercise 11.2 — Proof

\{True\} (i, j, s) := (0, 0, 0); while i ≠ n do B od \{s = n^2 + 2j\}
\Leftarrow \langle \text{right consequence} \rangle
\{True\} (i, j, s) := (0, 0, 0); while i ≠ n do B od \{s = i^2 + 2j \land i = n\}
\land (s = i^2 + 2j \land i = n \Rightarrow s = n^2 + 2j)
\Leftarrow \langle \text{sequence, logic} \rangle
\{True\} (i, j, s) := (0, 0, 0) \{s = i^2 + 2j\}
\land \{s = i^2 + 2j\} while i ≠ n do B od \{s = i^2 + 2j \land i = n\}
\land True
Exercise 11.2 — Proof

\{\text{True} \} \ (i, j, s) := (0, 0, 0) ; \textbf{while} \ i \neq n \ \textbf{do} \ B \ \textbf{od} \ \{s = n^2 + 2j\}

\iff \langle \text{right consequence} \rangle

\{\text{True} \} \ (i, j, s) := (0, 0, 0) ; \textbf{while} \ i \neq n \ \textbf{do} \ B \ \textbf{od} \ \{s = i^2 + 2j \land i = n\}
\land (s = i^2 + 2j \land i = n \Rightarrow s = n^2 + 2j)

\iff \langle \text{sequence, logic} \rangle

\{\text{True} \} \ (i, j, s) := (0, 0, 0) \ \{s = i^2 + 2j\}
\land \ {s = i^2 + 2j} \ \textbf{while} \ i \neq n \ \textbf{do} \ B \ \textbf{od} \ \{s = i^2 + 2j \land i = n\}
\land \ \text{True}

\iff \langle \text{left consequence, while-rule} \rangle

(\text{True} \Rightarrow 0 = 0^2 + 2 \cdot 0)
\land \ \{0 = 0^2 + 2 \cdot 0\} \ (i, j, s) := (0, 0, 0) \ \{s = i^2 + 2j\}
\land \ \{s = i^2 + 2j \land i \neq n\} \ \textbf{if} \ i = j \ \textbf{then} \ (i, j, s) := (i + 1, 0, s + 1)
\ \ \ \textbf{else} \ (j, s) := (j + 1, s + 2) \ \textbf{fi} \ \{s = i^2 + 2j\}
Exercise 11.2 — Proof (ctd.)

\[ \iff \langle \text{arithmetic, assignment, conditional} \rangle \]

\[ \text{True} \land \text{True} \]

\[ \land \{ s = i^2 + 2j \land i \neq n \land i = j \} \ (i, j, s) := (i + 1, 0, s + 1) \ \{ s = i^2 + 2j \} \]

\[ \land \{ s = i^2 + 2j \land i \neq n \land i \neq j \} \ (j, s) := (j + 1, s + 2) \ \{ s = i^2 + 2j \} \]
Exercise 11.2 — Proof (ctd.)

\[ \iff \langle \text{arithmetic, assignment, conditional} \rangle \]
\[ \text{True} \land \text{True} \land \{ s = i^2 + 2j \land i \neq n \land i = j \} \]
\[ (i, j, s) := (i + 1, 0, s + 1) \{ s = i^2 + 2j \} \]
\[ \land \{ s = i^2 + 2j \land i \neq n \land i \neq j \} \]
\[ (j, s) := (j + 1, s + 2) \{ s = i^2 + 2j \} \]
Exercise 11.2 — Proof (cont.)

\[ \iff \langle \text{arithmetic, assignment, conditional} \rangle \]

\[ \text{True} \land \text{True} \]

\[ \land \{ s = i^2 + 2j \land i \neq n \land i = j \} \]

\[ (i, j, s) := (i + 1, 0, s + 1) \{ s = i^2 + 2j \} \]

\[ \land \{ s = i^2 + 2j \land i \neq n \land i \neq j \} \]

\[ (j, s) := (j + 1, s + 2) \{ s = i^2 + 2j \} \]

\[ \iff \langle \text{left consequence, left consequence} \rangle \]

\[ (s = i^2 + 2j \land i \neq n \land i = j \Rightarrow s + 1 = (i + 1)^2 + 2 \cdot 0) \]

\[ \land \{ s + 1 = (i + 1)^2 + 2 \cdot 0 \} \]

\[ (i, j, s) := (i + 1, 0, s + 1) \{ s = i^2 + 2j \} \]

\[ \land (s = i^2 + 2j \land i \neq n \land i \neq j \Rightarrow s + 2 = i^2 + 2 \cdot (j + 1)) \]

\[ \land \{ s + 2 = i^2 + 2 \cdot (j + 1) \} \]

\[ (j, s) := (j + 1, s + 2) \{ s = i^2 + 2j \} \]
Exercise 11.2 — Proof (ctd.)

\[ \langle \text{arithmetic, assignment, conditional} \rangle \]

\[ \text{True} \land \text{True} \]

\[ \land \{ s = i^2 + 2j \land i \neq n \land i = j \} \quad (i, j, s) := (i + 1, 0, s + 1) \quad \{ s = i^2 + 2j \} \]

\[ \land \{ s = i^2 + 2j \land i \neq n \land i \neq j \} \quad (j, s) := (j + 1, s + 2) \quad \{ s = i^2 + 2j \} \]

\[ \langle \text{left consequence, left consequence} \rangle \]

\[ (s = i^2 + 2j \land i \neq n \land i = j \Rightarrow s + 1 = (i + 1)^2 + 2 \cdot 0) \]

\[ \land \{ s + 1 = (i + 1)^2 + 2 \cdot 0 \} \quad (i, j, s) := (i + 1, 0, s + 1) \quad \{ s = i^2 + 2j \} \]

\[ \land (s = i^2 + 2j \land i \neq n \land i \neq j \Rightarrow s + 2 = i^2 + 2 \cdot (j + 1)) \]

\[ \land \{ s + 2 = i^2 + 2 \cdot (j + 1) \} \quad (j, s) := (j + 1, s + 2) \quad \{ s = i^2 + 2j \} \]

\[ \langle \text{arithmetic, assignment, arithmetic, assignment} \rangle \]

\[ \text{True} \land \text{True} \land \text{True} \land \text{True} \]
Exercise 11.2 — Stronger Postcondition

\{
\text{True}
\}

\((i, j, s) := (0, 0, 0)\);

\textbf{while } i \neq n \textbf{ do}

\textbf{if } i = j

\textbf{then} \ ((i, j, s) := (i + 1, 0, s + 1))

\textbf{else} \ ((j, s) := (j + 1, s + 2))

\textbf{fi}

\textbf{od}

\{
\(s = n^2\)\}
Exercise 11.2 — Stronger Postcondition

\{ \text{True} \} \\
(i, j, s) := (0, 0, 0) ; \\
\textbf{while } i \neq n \textbf{ do} \\
\quad \textbf{if } i = j \\
\quad \quad \textbf{then} (i, j, s) := (i + 1, 0, s + 1) \\
\quad \quad \textbf{else} (j, s) := (j + 1, s + 2) \\
\quad \textbf{fi} \\
\textbf{od} \\
\{ s = n^2 \} \\

\textbf{Challenge:} How can you prove this?
Exercise 11.2 — Stronger Postcondition

\{True\}
(i, j, s) := (0, 0, 0) ;

while \(i \neq n\) do
  if \(i = j\)
    then (i, j, s) := (i + 1, 0, s + 1)
    else (j, s) := (j + 1, s + 2)
  fi
od
\{s = n^2\}

**Challenge:** How can you prove this?
Can you reformulate the program to make this easier?
Operational and Axiomatic Semantics

• **Operational Semantics** proves statements of the following shape:

\[ \text{State}_1 (\text{Statement} ) \Rightarrow \text{State}_2 \]
Operational and Axiomatic Semantics

- **Operational Semantics** proves statements of the following shape:

\[ State_1 (\text{Statement}) \Rightarrow State_2 \]

For some combinations of \text{Statement} and \text{State}_1, no such \text{State}_2 exists …
Operational and Axiomatic Semantics

- **Operational Semantics** proves statements of the following shape:

  \[ State_1 (\text{Statement}) \Rightarrow State_2 \]

  For some combinations of \text{Statement} and \text{State}_1, no such \text{State}_2 exists …

  Proofs about **a single execution path**: (counter-)examples rather than verification of specifications
Operational and Axiomatic Semantics

- **Operational Semantics** proves statements of the following shape:

  \[ \text{State}_1 (\text{Statement}) \implies \text{State}_2 \]

  For some combinations of \text{Statement} and \text{State}_1, no such \text{State}_2 exists …

  Proofs about **a single execution path**: (counter-)examples rather than verification of specifications

- **Axiomatic Semantics** proves statements of the following shape:

  \[ \{\text{Precondition}\} \text{ Statement } \{\text{Postcondition}\} \]
Operational and Axiomatic Semantics

- **Operational Semantics** proves statements of the following shape:
  
  \[
  \text{State}_1 (\text{Statement} ) \Rightarrow \text{State}_2
  \]

  For some combinations of \text{Statement} and \text{State}_1, no such \text{State}_2 exists …

  Proofs about a **single execution path**: (counter-)examples rather than verification of specifications

- **Axiomatic Semantics** proves statements of the following shape:
  
  \[
  \{ \text{Precondition} \} \text{ Statement} \{ \text{Postcondition} \}
  \]

  Proofs operate on **conditions on states** instead of states: **abstracting** away from states
Operational and Axiomatic Semantics

- **Operational Semantics** proves statements of the following shape:
  \[ \text{State}_1 (\text{Statement}) \Rightarrow \text{State}_2 \]
  For some combinations of \text{Statement} and \text{State}_1, no such \text{State}_2 exists …
  Proofs about **a single execution path**: (counter-)examples rather than verification of specifications

- **Axiomatic Semantics** proves statements of the following shape:
  \[ \{\text{Precondition}\} \text{Statement} \{\text{Postcondition}\} \]
  Proofs operate on **conditions on states** instead of states: abstracting away from states
  For different properties, different proofs **along the same program structure**.
Denotational Semantics

- Denotational Semantics proves statements of the following shape:

\[
[[\text{Statement}]] = F
\]

for some function or relation \( F \).
Denotational Semantics

- **Denotational Semantics** proves statements of the following shape:

\[
[[\text{Statement}]] = F
\]

for some function or relation \(F\).

\(F\) typically is a restricted kind of function between **semantic domains**.
Denotational Semantics

• Denotational Semantics proves statements of the following shape:

\[
\llbracket \text{Statement} \rrbracket = F
\]

for some function or relation \( F \).

\( F \) typically is a restricted kind of function between semantic domains.

Since \( F \) is a single mathematical object, it may be used as starting point for showing any kind of (functional) program properties.
Denotational Semantics

- **Denotational Semantics** proves statements of the following shape:
  \[
  \llbracket Statement \rrbracket = F
  \]
  for some function or relation \( F \).

\( F \) typically is a restricted kind of function between **semantic domains**.

Since \( F \) is a **single mathematical object**, it may be used as starting point for showing **any kind of (functional) program properties**.

In the textbook, denotational semantics appears mostly as a reorganisation of operational semantics.

**In general**, the denotational semantics is **far more abstract** than operational semantics, and employs advanced concepts from discrete mathematics.
Semantic Domains for Denotational Semantics

Usually, all *semantics domains* have
Semantic Domains for Denotational Semantics

Usually, all *semantics domains* have

– a definedness ordering \( \sqsubseteq \)
Semantic Domains for Denotational Semantics

Usually, all *semantics domains* have

– a *definedness ordering* $\sqsubseteq$, and

– a *least element* $\perp$ (read: “bottom”) wrt. $\sqsubseteq$
Semantic Domains for Denotational Semantics

Usually, all *semantics domains* have

- a *definedness ordering* $\sqsubseteq$, and

- a *least element* $\bot$ (read: "bottom") wrt. $\sqsubseteq$:

\[
\forall x : D \bullet \bot \sqsubseteq x
\]
Semantic Domains for Denotational Semantics

Usually, all semantics domains have

– a definedness ordering \( \sqsubseteq \), and

– a least element \( \bot \) (read: “bottom”) wrt. \( \sqsubseteq : \)

\[ \forall x : D \bullet \bot \sqsubseteq x \]

– (least upper bounds of chains \( x_0 \sqsubseteq x_1 \sqsubseteq x_2 \sqsubseteq \ldots \))
Semantic Domains for Denotational Semantics

Usually, all *semantics domains* have

- a **definedness ordering** $\sqsubseteq$, and

- a **least element** $\bot$ (read: “bottom”) wrt. $\sqsubseteq$:

  $$\forall x : D \bullet \bot \sqsubseteq x$$

- (least upper bounds of chains $x_0 \sqsubseteq x_1 \sqsubseteq x_2 \sqsubseteq \ldots$)

For **simple** semantics of imperative programs, **sets of partial functions** $A \rightharpoonup B$ can be used as domains:

- the subset ordering $\subseteq$ serves as definedness ordering:

  $$\forall f, g : A \rightharpoonup B \bullet f \sqsubseteq g :\iff f \subseteq g$$

- the empty function $\emptyset : A \rightharpoonup B$ is the a least element of $A \rightharpoonup B$. 
Semantic Domains for Simple Imperative Programs

\[
\begin{align*}
    \text{Bool} & = \{ \text{True, False} \} & \text{booleans} \\
    \text{Num}    & = \mathbb{Z} & \text{numbers}
\end{align*}
\]
Semantic Domains for Simple Imperative Programs

\[
\begin{align*}
Bool &\quad = \{\text{True, False}\} \quad \text{booleans} \\
Num &\quad = \mathbb{Z} \quad \text{numbers} \\
SVal &\quad = Bool + Num \quad \text{storable values}
\end{align*}
\]
Semantic Domains for Simple Imperative Programs

\[
\begin{align*}
    Bool & = \{\text{True}, \text{False}\} & \text{booleans} \\
    Num & = \mathbb{Z} & \text{numbers} \\
    SVal & = Bool + Num & \text{storable values} \\
    Id & & \text{identifiers}
\end{align*}
\]
Semantic Domains for Simple Imperative Programs

\[
\begin{align*}
  \text{Bool} &= \{\text{True, False}\} & \text{booleans} \\
  \text{Num} &= \mathbb{Z} & \text{numbers} \\
  \text{SVal} &= \text{Bool} + \text{Num} & \text{storable values} \\
  \text{Id} &= & \text{identifiers} \\
  \text{State} &= \text{Id} \mapsto \text{SVal} & \text{(simple) stores}
\end{align*}
\]
Semantic Domains for Simple Imperative Programs

\[
\begin{align*}
    \text{Bool} &\quad = \{\text{True, False}\} \quad \text{booleans} \\
    \text{Num} &\quad = \mathbb{Z} \quad \text{numbers} \\
    \text{SVal} &\quad = \text{Bool} + \text{Num} \quad \text{storable values} \\
    \text{Id} &\quad \text{identifiers} \\
    \text{State} &\quad = \text{Id} \rightarrow \text{SVal} \quad \text{(simple) stores} \\
    \text{Val} &\quad = \text{SVal} \quad \text{values}
\end{align*}
\]
## Semantic Domains for Simple Imperative Programs

<table>
<thead>
<tr>
<th>Domain</th>
<th>Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{Bool})</td>
<td>({\text{True}, \text{False}})</td>
<td>booleans</td>
</tr>
<tr>
<td>(\text{Num})</td>
<td>(\mathbb{Z})</td>
<td>numbers</td>
</tr>
<tr>
<td>(\text{SVal})</td>
<td>(\text{Bool} + \text{Num})</td>
<td>storable values</td>
</tr>
<tr>
<td>(\text{Id})</td>
<td></td>
<td>identifiers</td>
</tr>
<tr>
<td>(\text{State})</td>
<td>(\text{Id} \rightarrow \text{SVal})</td>
<td>(simple) stores</td>
</tr>
<tr>
<td>(\text{Val})</td>
<td>(\text{SVal})</td>
<td>values</td>
</tr>
<tr>
<td>(\text{State} \rightarrow \text{Val})</td>
<td></td>
<td>(expression semantics)</td>
</tr>
</tbody>
</table>
Semantic Domains for Simple Imperative Programs

\[
\begin{align*}
\text{Bool} & = \{\text{True, False}\} & \text{booleans} \\
\text{Num} & = \mathbb{Z} & \text{numbers} \\
\text{SVal} & = \text{Bool} + \text{Num} & \text{storable values} \\
\text{Id} & & \text{identifiers} \\
\text{State} & = \text{Id} \rightarrow \text{SVal} & (\text{simple})\text{ stores} \\
\text{Val} & = \text{SVal} & \text{values} \\
\text{State} \rightarrow \text{Val} & & (\text{expression semantics}) \\
\text{State} \rightarrow \text{State} & & (\text{statement semantics})
\end{align*}
\]
Direct Sums, or Disjoint Unions

“$A + B$” is the **direct sum** of the two sets $A$ and $B$. 
Direct Sums, or Disjoint Unions

“$A + B$” is the **direct sum** of the two sets $A$ and $B$.

You may have seen the definition of the equivalent **disjoint union**:

\[
A \uplus B = \{a : A \cdot (0, a)\} \cup \{b : B \cdot (1, b)\}
\]
Direct Sums, or Disjoint Unions

“A + B” is the direct sum of the two sets A and B.

You may have seen the definition of the equivalent disjoint union:

\[ A \sqcup B = \{a : A \cdot (0, a)\} \cup \{b : B \cdot (1, b)\} \]

In Haskell, there is the following prelude type constructor:

data Either a b = Left a | Right b
Direct Sums, or Disjoint Unions

“$A + B$” is the **direct sum** of the two sets $A$ and $B$.

You may have seen the definition of the equivalent **disjoint union**:

$$A \uplus B = \{a : A \cdot (0, a)\} \cup \{b : B \cdot (1, b)\}$$

In **Haskell**, there is the following prelude type constructor:

```haskell
data Either a b = Left a | Right b
```

This produces the two **constructors** for `Either` (which are **injections**):

- `Left :: a \rightarrow Either a b`
- `Right :: b \rightarrow Either a b`
Direct Sums, or Disjoint Unions

“A + B” is the **direct sum** of the two sets A and B.

You may have seen the definition of the equivalent **disjoint union**:

\[ A \uplus B = \{ a : A \cdot (0, a) \} \cup \{ b : B \cdot (1, b) \} \]

In *Haskell*, there is the following prelude type constructor:

```haskell
data Either a b = Left a | Right b
```

This produces the two **constructors** for *Either* (which are **injections**):

- **Left** :: \( a \rightarrow Either \ a \ \ b \)
- **Right** :: \( b \rightarrow Either \ a \ \ b \)

and allows pattern matching:

```haskell
valShow :: Either Integer Bool \rightarrow String
valShow (Left i) = "int:" ++ show i
valShow (Right b) = "bool:" ++ show b
```
Direct Sums, or Disjoint Unions

“$A + B$” is the **direct sum** of the two sets $A$ and $B$.

You may have seen the definition of the equivalent **disjoint union**:

$$A \uplus B = \{a : A \bullet (0, a)\} \cup \{b : B \bullet (1, b)\}$$

In **Haskell**, there is the following prelude type constructor:

```haskell
data Either a b = Left a | Right b
```

This produces the two **constructors** for `Either` (which are **injections**):

- `Left :: a \rightarrow Either a b`
- `Right :: b \rightarrow Either a b`

and allows pattern matching:

```haskell
valShow :: Either Integer Bool \rightarrow String
valShow (Left i) = "int:" ++ show i
valShow (Right b) = "bool:" ++ show b
```

In mathematical use, `Left` and `Right` are frequently not mentioned.
Semantic Functions

\[ [ \_ ]_E : \text{Expr} \rightarrow (\text{State} \rightarrow \text{Val}) \quad \text{expression semantics} \]

\[ [ \_ ]_S : \text{Stmt} \rightarrow (\text{State} \rightarrow \text{State}) \quad \text{statement semantics} \]
Semantic Functions

\[ \llbracket - \rrbracket_E : \text{Expr} \rightarrow (\text{State} \rightarrow \text{Val}) \quad \text{expression semantics} \]

\[ \llbracket - \rrbracket_S : \text{Stmt} \rightarrow (\text{State} \rightarrow \text{State}) \quad \text{statement semantics} \]

\[ \llbracket - \rrbracket_O : \text{Op} \rightarrow ((\text{Val} \times \text{Val}) \rightarrow \text{Val}) \quad \text{operator semantics (given)} \]
Semantic Functions

\[
\begin{align*}
\llbracket - \rrbracket_E &: \text{Expr} \rightarrow (\text{State} \mapsto \text{Val}) & \text{expression semantics} \\
\llbracket - \rrbracket_S &: \text{Stmt} \rightarrow (\text{State} \mapsto \text{State}) & \text{statement semantics} \\
\llbracket - \rrbracket_O &: \text{Op} \rightarrow ((\text{Val} \times \text{Val}) \mapsto \text{Val}) & \text{operator semantics (given)}
\end{align*}
\]

Textbook:

\[
\begin{align*}
M &: (\text{Expr} \times \text{State}) \mapsto \text{Val} & \text{expression semantics} \\
M &: (\text{Stmt} \times \text{State}) \mapsto \text{State} & \text{statement semantics} \\
\text{ApplyBinary} &: (\text{Op} \times \text{Val} \times \text{Val}) \mapsto \text{Val} & \text{operator semantics}
\end{align*}
\]
Semantic Functions

\[\llbracket - \rrbracket_E : \text{Expr} \rightarrow (\text{State} \rightarrow \text{Val})\] expression semantics

\[\llbracket - \rrbracket_S : \text{Stmt} \rightarrow (\text{State} \rightarrow \text{State})\] statement semantics

\[\llbracket - \rrbracket_O : \text{Op} \rightarrow ((\text{Val} \times \text{Val}) \rightarrow \text{Val})\] operator semantics (given)

**Textbook:**

\[M : (\text{Expr} \times \text{State}) \rightarrow \text{Val}\] expression semantics

\[M : (\text{Stmt} \times \text{State}) \rightarrow \text{State}\] statement semantics

\[\text{ApplyBinary} : (\text{Op} \times \text{Val} \times \text{Val}) \rightarrow \text{Val}\] operator semantics

- No clean separation between syntax and semantics
- Undefinedness ordering less obvious
Expression Semantics

\[ \text{Expr ::= Id | Num | Bool | Expr Op Expr} \]

\[ \llbracket \_ \rrbracket_E : \text{Expr} \rightarrow (\text{State} \rightarrow \text{Val}) \]

Assuming \( s : \text{State} \), i.e., \( s : \text{Id} \mapsto \text{SVal} \), we define:

for \( v : \text{Id} \):
\[ \llbracket v \rrbracket_E (s) = s(v) \]

— undefined if \( s(v) \) is undefined!

for \( n : \text{Num} \):
\[ \llbracket n \rrbracket_E (s) = n \]

for \( b : \text{Bool} \):
\[ \llbracket b \rrbracket_E (s) = b \]

for \( e_1, e_2 : \text{Expr}; \text{op : Op} \):
\[ \llbracket e_1 \text{ op } e_2 \rrbracket_E (s) = \llbracket \text{op} \rrbracket_O (\llbracket e_1 \rrbracket_E (s), \llbracket e_2 \rrbracket_E (s)) \]

— undefined if \( \llbracket e_1 \rrbracket_E (s) \) or \( \llbracket e_2 \rrbracket_E (s) \) undefined, or from \( \llbracket \text{op} \rrbracket_O \)!
### Expression Semantics

\[ \text{Expr ::= Id | Num | Bool | Expr Op Expr} \]

\[ \llbracket - \rrbracket_E : \text{Expr} \rightarrow (\text{State} \rightarrow \text{Val}) \]

Assuming \( s : \text{State} \), i.e., \( s : \text{Id} \rightarrow \text{SVal} \), we define:

- For \( v : \text{Id} \):
  \[ \llbracket v \rrbracket_E (s) = s(v) \]
  — **undefined** if \( s(v) \) is undefined!

- For \( n : \text{Num} \):
  \[ \llbracket n \rrbracket_E (s) = n \]

- For \( b : \text{Bool} \):
  \[ \llbracket b \rrbracket_E (s) = b \]

- For \( e_1, e_2 : \text{Expr}; \, op : \text{Op} \):
  \[ \llbracket e_1 \, op \, e_2 \rrbracket_E (s) = \llbracket op \rrbracket_O (\llbracket e_1 \rrbracket_E (s), \llbracket e_2 \rrbracket_E (s)) \]
  — **undefined** if \( \llbracket e_1 \rrbracket_E (s) \) or \( \llbracket e_2 \rrbracket_E (s) \) undefined, or from \( \llbracket op \rrbracket_O \)!

Where clear from the context, we write \( \llbracket e \rrbracket \) instead of \( \llbracket e \rrbracket_E \).
Expression Semantics — Examples

Examples: Let $s_1 = \{x \mapsto 5, y \mapsto 42, z \mapsto 0\}$:
Expression Semantics — Examples

Examples: Let $s_1 = \{x\mapsto 5, y\mapsto 42, z\mapsto 0\}$:

$$[[x + y]](s_1) = [[x]](s_1) + [[y]](s_1) = s_1(x) + s_1(y) = 5 + 42 = 47$$
Expression Semantics — Examples

Examples: Let $s_1 = \{x \mapsto 5, y \mapsto 42, z \mapsto 0\}$:

$[[x + y]](s_1) = [[x]](s_1) + [[y]](s_1) = s_1(x) + s_1(y) = 5 + 42 = 47$

$[[7 - q]](s_1) = [[7]](s_1) - [[q]](s_1) = 7 - s_1(q) = 7 - \bot = \bot$

uninit. var.!
Expression Semantics — Examples

Examples: Let $s_1 = \{ x \mapsto 5, y \mapsto 42, z \mapsto 0 \}$:

\[
\llbracket x + y \rrbracket(s_1) = \llbracket x \rrbracket(s_1) + \llbracket y \rrbracket(s_1) = s_1(x) + s_1(y) = 5 + 42 = 47
\]

\[
\llbracket 7 - q \rrbracket(s_1) = \llbracket 7 \rrbracket(s_1) - \llbracket q \rrbracket(s_1) = 7 - s_1(q) = 7 - \bot = \bot
\]

uninit. var.!

Writing “$\bot$” here is short-hand for indicating undefined terms.
Expression Semantics — Examples

Examples: Let $s_1 = \{ x \mapsto 5, y \mapsto 42, z \mapsto 0 \}$:

$$[[x + y]](s_1) = [[x]](s_1) + [[y]](s_1) = s_1(x) + s_1(y) = 5 + 42 = 47$$

$$[[7 - q]](s_1) = [[7]](s_1) - [[q]](s_1) = 7 - s_1(q) = 7 - \bot = \bot \text{ uninit. var.}!$$

$$[[12 / z]](s_1) = [[12]](s_1) / [[z]](s_1) = 12 / s_1(z) = 12 / 0 = \bot$$

Writing “$\bot$” here is short-hand for indicating undefined terms.
Expression Semantics — Examples

Examples: Let $s_1 = \{x \mapsto 5, y \mapsto 42, z \mapsto 0\}$:

\[
\llbracket x + y \rrbracket(s_1) = \llbracket x \rrbracket(s_1) + \llbracket y \rrbracket(s_1) = s_1(x) + s_1(y) = 5 + 42 = 47
\]

\[
\llbracket 7 - q \rrbracket(s_1) = \llbracket 7 \rrbracket(s_1) - \llbracket q \rrbracket(s_1) = 7 - s_1(q) = 7 - \bot = \bot
\]

uninit. var.!

\[
\llbracket 12 / z \rrbracket(s_1) = \llbracket 12 \rrbracket(s_1) / \llbracket z \rrbracket(s_1) = 12 / s_1(z) = 12 / 0 = \bot
\]

\[
\llbracket x \& \& y \rrbracket(s_1) = \llbracket x \rrbracket(s_1) \land \llbracket y \rrbracket(s_1) = s_1(x) \land s_1(y) = 5 \land 42 = \bot
\]

wrong type!

Writing “\(\bot\)” here is short-hand for indicating undefined terms.
Statement Semantics

\[ \text{S : Stmt} \rightarrow (\text{State} \rightarrow \text{State}) \]

For \( s : \text{State} \), i.e., \( s : \text{Id} \rightarrow \text{SVal} \), and \( p, p_1, p_2 : \text{Stmt} \) and \( e : \text{Expr} \) and \( v : \text{Id} \):

\[
\begin{align*}
\llbracket \text{skip} \rrbracket_S (s) &\quad = s \\
\llbracket v := e \rrbracket_S (s) &\quad = s \oplus \{ v \mapsto \llbracket e \rrbracket_E (s) \} \\
\quad \text{— undefined if } \llbracket e \rrbracket_E (s) \text{ is undefined!} \\
\llbracket p_1 ; p_2 \rrbracket_S &\quad = \llbracket p_2 \rrbracket_S \circ \llbracket p_1 \rrbracket_S \\
\llbracket \text{if } e \text{ then } p_1 \text{ else } p_2 \rrbracket_S (s) &\quad = \\
&\quad \begin{cases} \\
\llbracket p_1 \rrbracket_S (s) &\quad \text{if } \llbracket e \rrbracket_E (s) = \text{True} \\
\llbracket p_2 \rrbracket_S (s) &\quad \text{if } \llbracket e \rrbracket_E (s) = \text{False} \\
\text{undefined} &\quad \text{otherwise}
\end{cases}
\end{align*}
\]
Relating Simple Denotational and Operational Semantics

**Simple Denotational**

\[ \llbracket e \rrbracket_E (\sigma_1) = \nu \]

\[ \llbracket s \rrbracket_S (\sigma_1) = \sigma_2 \]

**Simple Operational**

\[ \sigma_1(e) \Rightarrow \nu \]

\[ \sigma_1(s) \Rightarrow \sigma_2 \]
Relating Simple Denotational and Operational Semantics

Simple Denotational

\[
\llbracket e \rrbracket_E (\sigma_1) = v
\]
\[
\llbracket s \rrbracket_S (\sigma_1) = \sigma_2
\]

Simple Operational

\[
\sigma_1(e) \Rightarrow v
\]
\[
\sigma_1(s) \Rightarrow \sigma_2
\]

- \([e]_E\) and \([s]_S\) are explicit functions
Relating Simple Denotational and Operational Semantics

**Simple Denotational**

\[
\llbracket e \rrbracket_E (\sigma_1) = v \\
\llbracket s \rrbracket_S (\sigma_1) = \sigma_2
\]

**Simple Operational**

\[
\sigma_1(e) \Rightarrow v \\
\sigma_1(s) \Rightarrow \sigma_2
\]

- \(\llbracket e \rrbracket_E\) and \(\llbracket s \rrbracket_S\) are explicit functions

- These can be considered as results of function abstraction from the operational semantics of \(e\) and \(s\). 
\textbf{\lambda-Calculus (Textbook 8.1) — Motivation for the \lambda-Notation}

The usual way to define functions:

\[ f(x) = 2 \times x - 3 \]
\[ f(x) = 2 \times x - 3 \]

This is not an explicit definition!
\( \lambda \)-Calculus (Textbook 8.1) — Motivation for the \( \lambda \)-Notation

The usual way to define functions:

\[
f(x) = 2 \times x - 3
\]

This is not an explicit definition!

For an explicit definition, the defined item needs to stand alone on the left-hand side of “=”. 
\textbf{\lambda-Calculus (Textbook 8.1) — Motivation for the \lambda-Notation}

The usual way to define functions:

\[ f(x) = 2 \times x - 3 \]

This is not an explicit definition!

For an explicit definition, the defined item needs to stand alone on the left-hand side of "\(=\)". Therefore, we need a way to denote a \textbf{function} on the right-hand side.
\textbf{\textit{\lambda}}-\textit{Calculus (Textbook 8.1) — Motivation for the \textit{\lambda}}-\textit{Notation}

The usual way to define functions:

\[ f(x) = 2 * x - 3 \]

This is not an explicit definition!

For an explicit definition, the defined item needs to stand alone on the left-hand side of "=". Therefore, we need a way to denote a \textbf{function} on the right-hand side. \textit{\lambda}-abstraction is such a notation:

\[ f = \lambda x \cdot 2 * x - 3 \]
The usual way to define functions:

\[ f(x) = 2 \times x - 3 \]

This is not an explicit definition!

For an explicit definition, the defined item needs to stand alone on the left-hand side of “=”. Therefore, we need a way to denote a function on the right-hand side. \( \lambda \)-abstraction is such a notation:

\[ f = \lambda x \cdot 2 \times x - 3 \]

This is equivalent to the above.
λ-Calculus (Textbook 8.1) — Motivation for the λ-Notation

The usual way to define functions:

\[ f(x) = 2 \cdot x - 3 \]

This is not an explicit definition!

For an explicit definition, the defined item needs to stand alone on the left-hand side of "=". Therefore, we need a way to denote a function on the right-hand side. λ-abstraction is such a notation:

\[ f = \lambda x \cdot 2 \cdot x - 3 \]

This is equivalent to the above. Therefore:

\[ f \ 5 = (\lambda x \cdot 2 \cdot x - 3) \ 5 = 2 \cdot 5 - 3 \]
\textbf{\lambda-Calculus (Textbook 8.1) — Motivation for the \lambda-Notation}

The usual way to define functions:

\[ f(x) = 2 \ast x - 3 \]

This is not an explicit definition!

For an explicit definition, the defined item needs to stand alone on the left-hand side of “\( = \)”. Therefore, we need a way to denote a \textbf{function} on the right-hand side. \lambda-abstraction is such a notation:

\[ f = \lambda x \bullet 2 \ast x - 3 \]

This is equivalent to the above. Therefore:

\[ f \, 5 \, = \, (\lambda x \bullet 2 \ast x - 3) \, 5 \, = \, 2 \ast 5 - 3 \]

\lambda-abstraction \textbf{binds} a variable (here: \( x \)).
**λ-Calculus (Textbook 8.1) — Motivation for the λ-Notation**

The usual way to define functions:

\[ f(x) = 2 \times x - 3 \]

This is not an explicit definition!

For an explicit definition, the defined item needs to stand alone on the left-hand side of “=”. Therefore, we need a way to denote a function on the right-hand side. λ-abstraction is such a notation:

\[ f = \lambda x \cdot 2 \times x - 3 \]

This is equivalent to the above. Therefore:

\[ f \ 5 \ = \ (\lambda x \cdot 2 \times x - 3) \ 5 \ = \ 2 \times 5 - 3 \]

λ-abstraction binds a variable (here: \( x \)). Application of a λ-abstraction to an argument is reduced to the body of the abstraction with the bound variable replaced by the argument.
λ-Terms

Now the formal definition of untyped λ-terms: An untyped λ-term is either
• a variable \(x, y, z, \ldots\), or
• a function application \((M)N\) of one untyped λ-term \(F\) (the function) to another \(A\) (the argument), or
• a function abstraction \(\lambda x \cdot B\) of an untyped λ-term \(B\) (the body) over a variable \(x\).
$\lambda$-Terms

Now the formal definition of **untyped $\lambda$-terms**: An untyped $\lambda$-term is either

- a **variable** $x, y, z, \ldots$, or
- a **function application** $(M)N$ of one untyped $\lambda$-term $F$ (the function) to another $A$ (the argument), or
- a **function abstraction** $\lambda x \bullet B$ of an untyped $\lambda$-term $B$ (the body) over a variable $x$.

**Note**: Every untyped $\lambda$-term can be used as function in function applications!
λ-Terms

Now the formal definition of **untyped λ-terms**: An untyped λ-term is either

- a **variable** $x, y, z, \ldots$, or
- a **function application** $(M) N$ of one untyped λ-term $F$ (the function) to another $A$ (the argument), or
- a **function abstraction** $\lambda x \cdot B$ of an untyped λ-term $B$ (the body) over a variable $x$.

**Note:** Every untyped λ-term can be used as function in function applications!

**Note:** We add and omit parentheses using the rules that are used in Haskell:
\textbf{λ-Terms}

Now the formal definition of \textit{untyped} \textit{λ}-\textit{terms}: An untyped \( \lambda \)-term is either

- a \textbf{variable} \( x, y, z, \ldots \), or
- a \textbf{function application} \((M) N\) of one untyped \( \lambda \)-term \( F \) (the function) to another \( A \) (the argument), or
- a \textbf{function abstraction} \( \lambda x \cdot B \) of an untyped \( \lambda \)-term \( B \) (the body) over a variable \( x \).

\textbf{Note:} Every untyped \( \lambda \)-term can be used as function in function applications!

\textbf{Note:} We add and omit parentheses using the rules that are used in Haskell:
- \( \lambda \)-abstraction extends as far right as possible, usually until an unmatched closing parenthesis or the end of the term.
**λ-Terms**

Now the formal definition of untyped λ-terms: An untyped λ-term is either

- a **variable** \( x, y, z, \ldots \), or
- a **function application** \((M)N\) of one untyped λ-term \(F\) (the function) to another \(A\) (the argument), or
- a **function abstraction** \(\lambda x \cdot B\) of an untyped λ-term \(B\) (the body) over a variable \(x\).

**Note:** Every untyped λ-term can be used as function in function applications!

**Note:** We add and omit parentheses using the rules that are used in Haskell:

- λ-abstraction extends as far right as possible, usually until an unmatched closing parenthesis or the end of the term.
- Application associates to the left, i.e., \(f \ x \ y\) is understood to mean \((f \ x) \ y\).

According to the definition above, this would actually have to be \(((f) \ x) \ y\).
**λ-Terms**

Now the formal definition of **untyped λ-terms**: An untyped λ-term is either

- a **variable** \(x, y, z, \ldots\), or
- a **function application** \((M) N\) of one untyped λ-term \(F\) (the function) to another \(A\) (the argument), or
- a **function abstraction** \(\lambda x \cdot B\) of an untyped λ-term \(B\) (the body) over a variable \(x\).

**Note:** Every untyped λ-term can be used as function in function applications!

**Note:** We add and omit parentheses using the rules that are used in Haskell:

- λ-abstraction extends as far right as possible, usually until an unmatched closing parenthesis or the end of the term.
- Application associates to the left, i.e., \(f x y\) is understood to mean \(f (x y)\).
  According to the definition above, this would actually have to be \(((f) x) y\).

The λ-calculus was intended by its inventor, **Alonzo Church** (1903–1995), as a foundation of mathematics based on functions instead of on sets.
Free Variables

The set $FV(M)$ of the variables occurring free in the $\lambda$-term $M$ is defined inductively over the construction of $\lambda$-terms (this is called: *structural induction*):

- $FV(x) = \{x\}$

- $FV(\lambda x \cdot M) = FV(M) \setminus \{x\}$

- $FV(M \, N) = FV(M) \cup FV(N)$
Variable Replacement (auxiliary concept)

$M[x \setminus y]$ denotes the term resulting from $M$ by replacing all free occurrences of variable $x$ with variable $y$:

- $v[x \setminus y] = \begin{cases} y & \text{if } v = x \\ v & \text{if } v \neq x \end{cases}$

- $(M \ N)[x \setminus y] = M[x \setminus y] \ N[x \setminus y]$

- $(\lambda \ v \cdot M)[x \setminus y] = \begin{cases} \lambda \ v \cdot M & \text{if } v = x \\ \lambda \ v \cdot (M[x \setminus y]) & \text{if } v \neq x \end{cases}$

— Variable replacement is only used in the definition of $\alpha$-conversion.
**α-Conversion**

If $y \notin FV(M)$, and if there is no $\lambda$-binding for $y$ in $M$, then the following **renaming of a bound variable** is defined:

$$\lambda x \cdot M \equiv_\alpha \lambda y \cdot M[x \setminus y]$$

This can also be applied in any context $C[\ ]$ (a context is a term with exactly one occurrence of the “hole” “[ ]”):

$$C[ \lambda x \cdot M ] \equiv_\alpha C[ \lambda y \cdot M[x \setminus y] ]$$

| α-Conversion | = | renaming of bound variables |
Substitution

Substitution is replacement of free variables by terms:

- \( v[x \setminus t] = \begin{cases} 
  t & \text{if } v = x \\
  v & \text{if } v \neq x 
\end{cases} \)

- \( (M \ N)[x \setminus t] = M[x \setminus t] \ N[x \setminus t] \)

- \( (\lambda \ v \bullet M)[x \setminus t] = \begin{cases} 
  \lambda \ v \bullet M & \text{if } v = x \lor x \notin \text{FV}(M) \\
  \lambda \ v \bullet (M[x \setminus t]) & \text{if } v \neq x \land x \in \text{FV}(M) \land v \notin \text{FV}(t) \\
  \textit{not permitted!} & \text{if } v \neq x \land x \in \text{FV}(M) \land v \in \text{FV}(t) 
\end{cases} \)
Substitution

**Substitution** is replacement of free variables by terms:

- \( v[x \backslash t] = \begin{cases} t & \text{if } v = x \\ v & \text{if } v \neq x \end{cases} \)

- \( (M N)[x \backslash t] = M[x \backslash t] \cdot N[x \backslash t] \)

- \( (\lambda v \cdot M)[x \backslash t] = \begin{cases} \lambda v \cdot M & \text{if } v = x \lor x \notin \text{FV}(M) \\ \lambda v \cdot (M[x \backslash t]) & \text{if } v \neq x \land x \in \text{FV}(M) \land v \notin \text{FV}(t) \\ \text{not permitted!} & \text{if } v \neq x \land x \in \text{FV}(M) \land v \in \text{FV}(t) \end{cases} \)

Where a substitution \([x \backslash t]\) is not permitted for a term \(M\), an \(\alpha\)-conversion \(M \equiv_{\alpha} M'\) is always possible such that the substitution is permitted for \(M'\).
**Substitution**

Substitution is replacement of free variables by terms:

- \( \nu[x \setminus t] = \begin{cases} \quad t & \text{if } \nu = x \\ \nu & \text{if } \nu \neq x \end{cases} \)

- \((M \cdot N)[x \setminus t] = M[x \setminus t] \cdot N[x \setminus t]\)

- \((\lambda \nu \bullet M)[x \setminus t] = \begin{cases} \lambda \nu \bullet M & \text{if } \nu = x \land x \notin FV(M) \\ \lambda \nu \bullet (M[x \setminus t]) & \text{if } \nu \neq x \land x \in FV(M) \land \nu \notin FV(t) \\ \text{not permitted!} & \text{if } \nu \neq x \land x \in FV(M) \land \nu \in FV(t) \end{cases} \)

Where a substitution \([x \setminus t]\) is not permitted for a term \(M\), an \(\alpha\)-conversion \(M \equiv_\alpha M'\) is always possible such that the substitution is permitted for \(M'\).

**Example:**

\((\lambda z \bullet f(z \cdot x))[x \setminus f \cdot z]\)
Substitution

Substitution is replacement of free variables by terms:

- \( v[x \setminus t] = \begin{cases} t & \text{if } v = x \\ v & \text{if } v \neq x \end{cases} \)

- \((M N)[x \setminus t] = M[x \setminus t] N[x \setminus t]\)

- \((\lambda v \cdot M)[x \setminus t] = \begin{cases} \lambda v \cdot M & \text{if } v = x \lor x \notin FV(M) \\ \lambda v \cdot (M[x \setminus t]) & \text{if } v \neq x \land x \in FV(M) \land v \notin FV(t) \\ \text{not permitted!} & \text{if } v \neq x \land x \in FV(M) \land v \in FV(t) \end{cases} \)

Where a substitution \([x \setminus t]\) is not permitted for a term \(M\), an \(\alpha\)-conversion \(M \equiv_\alpha M'\) is always possible such that the substitution is permitted for \(M'\).

Example:

\((\lambda z \cdot f (z x))[x \setminus f z] \equiv_\alpha (\lambda y \cdot f (y x))[x \setminus f z] \)
Substitution

**Substitution** is replacement of free variables by terms:

- \( v[x \backslash t] = \begin{cases} 
  t & \text{if } v = x \\
  v & \text{if } v \neq x
\end{cases} \)

- \((M N)[x \backslash t] = M[x \backslash t] N[x \backslash t]\)

- \((\lambda v \bullet M)[x \backslash t] = \begin{cases} 
  \lambda v \bullet M & \text{if } v = x \land x \notin FV(M) \\
  \lambda v \bullet (M[x \backslash t]) & \text{if } v \neq x \land x \in FV(M) \land v \notin FV(t) \\
  \text{not permitted!} & \text{if } v \neq x \land x \in FV(M) \land v \in FV(t)
\end{cases} \)

Where a substitution \([x \backslash t]\) is not permitted for a term \(M\), an \(\alpha\)-conversion \(M \equiv_\alpha M'\) is always possible such that the substitution is permitted for \(M'\).

**Example:**

\[
(\lambda z \bullet f(z x))[x \backslash f z] \equiv_\alpha (\lambda y \bullet f(y x))[x \backslash f z] = \lambda y \bullet f(y (f z))
\]
β-Reduction

The central reduction rule of λ-calculus:

$$(\lambda x \cdot B) A \rightarrow_{\beta} B[x \backslash A]$$
\( \beta \)-Reduction

The central reduction rule of \( \lambda \)-calculus:

\[
(\lambda x \cdot B) A \rightarrow_\beta B[x \backslash A]
\]

This can also be applied in any context \( C[ \ ] \):

\[
C[ (\lambda x \cdot B) A ] \rightarrow_\beta C[ B[x \backslash A ]]
\]
\( \beta \)-Reduction

The central reduction rule of \( \lambda \)-calculus:

\[
(\lambda x \cdot B) A \rightarrow_\beta B[x \backslash A]
\]

This can also be applied in any context \( C[ \ ] \):

\[
C[ (\lambda x \cdot B) A ] \rightarrow_\beta C[ B[x \backslash A ]] 
\]

Example:

\[
(\lambda x \cdot \lambda z \cdot x (z \ x)) (\lambda y \cdot z \ y) \rightarrow_\beta
\]
\[\beta\text{-Reduction}\]

The central reduction rule of \(\lambda\)-calculus:

\[\ (\lambda \, x \, \bullet \, B) \, A \quad \rightarrow_{\beta} \quad B[x \, \backslash \, A]\]

This can also be applied in any context \(C[\ ]\):

\[\ C[\ (\lambda \, x \, \bullet \, B) \, A ] \quad \rightarrow_{\beta} \quad C[\ B[x \, \backslash \, A ] ]\]

Example:

\[\ (\lambda \, x \, \bullet \, \lambda \, z \, \bullet \, x \, (z \, x)) \, (\lambda \, y \, \bullet \, z \, y) \quad \rightarrow_{\beta} \quad (\lambda \, z \, \bullet \, x \, (z \, x))[x \, \backslash \, (\lambda \, y \, \bullet \, z \, y)]\]
\[\beta\text{-Reduction}\]

The central reduction rule of \(\lambda\)-calculus:

\[
(\lambda x \bullet B) A \rightarrow_{\beta} B[x \backslash A]
\]

This can also be applied in any context \(C[\ ]\):

\[
C[ (\lambda x \bullet B) A ] \rightarrow_{\beta} C[ B[x \backslash A ]]
\]

Example:

\[
(\lambda x \bullet \lambda z \bullet x (z x)) (\lambda y \bullet z y) \rightarrow_{\beta} (\lambda z \bullet x (z x))[x \backslash (\lambda y \bullet z y)]
\]

\[
\equiv_{\alpha} (\lambda u \bullet x (u x))[x \backslash (\lambda y \bullet z y)]
\]
**β-Reduction**

The central reduction rule of λ-calculus:

\[(\lambda x \cdot B) A \rightarrow_\beta B[x \backslash A]\]

This can also be applied in any context \(C[\ ]\):

\[C[ (\lambda x \cdot B) A ] \rightarrow_\beta C[ B[x \backslash A ] ]\]

**Example:**

\[(\lambda x \cdot \lambda z \cdot x (z x)) (\lambda y \cdot z y) \rightarrow_\beta (\lambda z \cdot x (z x))[x \backslash (\lambda y \cdot z y)]\]

\[\equiv_{\alpha} (\lambda u \cdot x (u x))[x \backslash (\lambda y \cdot z y)]\]

\[= \lambda u \cdot ((\lambda y \cdot z y) (u (\lambda y \cdot z y)))\]
**β-Reduction**

The central reduction rule of λ-calculus:

\[(\lambda x \bullet B) A \rightarrow_\beta B[x \setminus A]\]

This can also be applied in any context \(C[\ ]\):

\[C[ (\lambda x \bullet B) A ] \rightarrow_\beta C[ B[x \setminus A ]]]\]

**Example:**

\[(\lambda x \bullet \lambda z \bullet x (z x)) (\lambda y \bullet z y) \rightarrow_\beta (\lambda z \bullet x (z x))[x \setminus (\lambda y \bullet z y)]\]

\[\equiv_\alpha (\lambda u \bullet x (u x))[x \setminus (\lambda y \bullet z y)]\]

\[= \lambda u \bullet ((\lambda y \bullet z y) (u (\lambda y \bullet z y)))\]

\[\rightarrow_\beta \lambda u \bullet ((z y)[y \setminus (u (\lambda y \bullet z y))])\]
\[ (\lambda x \cdot B) A \rightarrow_{\beta} B[x \setminus A] \]

This can also be applied in any context \( C[ \ ] \):

\[ C[ (\lambda x \cdot B) A ] \rightarrow_{\beta} C[ B[x \setminus A ] ] \]

**Example:**

\[
(\lambda x \cdot \lambda z \cdot x (z x)) (\lambda y \cdot z y) \\
\rightarrow_{\beta} \quad (\lambda z \cdot x (z x))[x \setminus (\lambda y \cdot z y)] \\
\equiv_{\alpha} \quad (\lambda u \cdot x (u x))[x \setminus (\lambda y \cdot z y)] \\
= \quad \lambda u \cdot ((\lambda y \cdot z y) (u (\lambda y \cdot z y))) \\
\rightarrow_{\beta} \quad \lambda u \cdot ((z y)[y \setminus (u (\lambda y \cdot z y))]) \\
= \quad \lambda u \cdot (z (u (\lambda y \cdot z y)))
\]
Reduction Strategies

• **Leftmost-outermost strategy**: among all outermost redexes the one starting farthest to the left.
Reduction Strategies

- **Leftmost-outermost strategy**: among all outermost redexes the one starting farthest to the left.

- **Leftmost-innermost strategy**: among all innermost redexes the one starting farthest to the left.
Reduction Strategies

- **Leftmost-outermost strategy**: among all outermost redexes the one starting farthest to the left.

- **Leftmost-innermost strategy**: among all innermost redexes the one starting farthest to the left.

“inner” and “outer” are determined by the abstract syntax tree.
Reduction Strategies

- **Leftmost-outermost strategy**: among all outermost redexes the one starting farthest to the left.

- **Leftmost-innermost strategy**: among all innermost redexes the one starting farthest to the left.

“inner” and “outer” are determined by the abstract syntax tree.

**Important properties:**

- Leftmost-outermost strategy (**Haskell**, **Miranda**, **Clean**):
  - call by name, lazy evaluation
  - terminates if possible
  - non-strict
Reduction Strategies

- **Leftmost-outermost strategy**: among all outermost redexes the one starting farthest to the left.

- **Leftmost-innermost strategy**: among all innermost redexes the one starting farthest to the left.

“inner” and “outer” are determined by the abstract syntax tree.

**Important properties:**

- Leftmost-outermost strategy (*Haskell*, *Miranda*, *Clean*):
  - call by name, lazy evaluation
  - terminates if possible
  - non-strict

- Leftmost-innermost strategy (*OCaml*, *SML*, *LISP*, *Scheme*):
  - call by value, eager evaluation
  - easier to implement
  - **strict**: for all $f$ we have $f(\bot) = \bot$
The Fixedpoint Combinator $Y$

Church’s fixed point combinator “$Y$”:

\[ Y = \lambda f \bullet (\lambda x \bullet f(x))(\lambda x \bullet f(x)) \]
The Fixedpoint Combinator $Y$

Church’s fixedpoint combinator “$Y$”:

$$Y = \lambda f \bullet (\lambda x \bullet f (x \ x))(\lambda x \bullet f (x \ x))$$

Proof of fixedpoint combinator property — for every $f$, the following holds:

$$Y \ f =$$
The Fixedpoint Combinator $Y$

Church’s fixedpoint combinator “$Y$”:

$$Y = \lambda f \quad (\lambda x \quad f \,(x\,x))(\lambda x \quad f \,(x\,x))$$

Proof of fixedpoint combinator property — for every $f$, the following holds:

$$Y\,f = (\lambda f \quad (\lambda x \quad f \,(x\,x))(\lambda x \quad f \,(x\,x)))\,f$$
The Fixedpoint Combinator \( Y \)

Church’s fixedpoint combinator “\( Y \)”: 
\[
Y = \lambda f \cdot (\lambda x \cdot f (x \ x))(\lambda x \cdot f (x \ x))
\]

Proof of fixedpoint combinator property — for every \( f \), the following holds: 
\[
Y f = (\lambda f \cdot (\lambda x \cdot f (x \ x))(\lambda x \cdot f (x \ x))) f \\
\rightarrow_\beta (\lambda x \cdot f (x \ x))(\lambda x \cdot f (x \ x))
\]
The Fixedpoint Combinator $Y$

Church’s fixedpoint combinator “$Y$”:

$$Y = \lambda f \bullet (\lambda x \bullet f(x x))(\lambda x \bullet f(x x))$$

Proof of fixedpoint combinator property — for every $f$, the following holds:

$$Y f = (\lambda f \bullet (\lambda x \bullet f(x x))(\lambda x \bullet f(x x))) f$$

$$\rightarrow_{\beta} (\lambda x \bullet f(x x))(\lambda x \bullet f(x x))$$

$$= (\lambda z \bullet f(z z))(\lambda x \bullet f(x x))$$
The Fixedpoint Combinator $Y$

Church’s fixedpoint combinator “$Y$”:

$$Y = \lambda f \cdot (\lambda x \cdot f(xx))(\lambda x \cdot f(xx))$$

Proof of fixedpoint combinator property — for every $f$, the following holds:

$$Yf = (\lambda f \cdot (\lambda x \cdot f(xx))(\lambda x \cdot f(xx)))f$$

$$\rightarrow _\beta (\lambda z \cdot f(zz))(\lambda x \cdot f(xx))$$

$$= (\lambda z \cdot f(zz))(\lambda x \cdot f(xx))$$

$$\rightarrow _\beta f(((\lambda x \cdot f(xx))(\lambda x \cdot f(xx))))$$
The Fixedpoint Combinator $Y$

Church’s fixedpoint combinator “$Y$”:

$$Y = \lambda f \bullet (\lambda x \bullet f (x x))(\lambda x \bullet f (x x))$$

Proof of fixedpoint combinator property — for every $f$, the following holds:

$$Y f = (\lambda f \bullet (\lambda x \bullet f (x x))(\lambda x \bullet f (x x))) f$$

$$\rightarrow_{\beta} (\lambda x \bullet f (x x))(\lambda x \bullet f (x x))$$

$$= (\lambda z \bullet f (z z))(\lambda x \bullet f (x x))$$

$$\rightarrow_{\beta} f ((\lambda x \bullet f (x x)) (\lambda x \bullet f (x x)))$$

$$= f (Y f)$$
The Fixedpoint Combinator $Y$

Church’s fixedpointcombinator “$Y$”:

$$ Y = \lambda f \bullet (\lambda x \bullet f (x \ x))(\lambda x \bullet f (x \ x)) $$

Proof of fixedpointcombinator property — for every $f$, the following holds:

$$ Y \ f = (\lambda f \bullet (\lambda x \bullet f (x \ x))(\lambda x \bullet f (x \ x))) \ f $$

$$ \rightarrow_{\beta} (\lambda x \bullet f (x \ x))(\lambda x \bullet f (x \ x)) $$

$$ = (\lambda z \bullet f (z \ z))(\lambda x \bullet f (x \ x)) $$

$$ \rightarrow_{\beta} f ((\lambda x \bullet f (x \ x)) (\lambda x \bullet f (x \ x))) $$

$$ = f (Y \ f) $$

In the theory of $\lambda$-calculus this yields the fixedpointequation $Y \ f = f (Y \ f)$.
The Fixedpoint Combinator $Y$

Church’s fix edpointcombinator “$Y$”:

$$Y = \lambda f \bullet (\lambda x \bullet f (x \ x))(\lambda x \bullet f (x \ x))$$

Proof of fix edpointcombinator property — for every $f$, the following holds:

$$Y f = (\lambda f \bullet (\lambda x \bullet f (x \ x))(\lambda x \bullet f (x \ x))) f$$

$$\rightarrow_\beta (\lambda x \bullet f (x \ x))(\lambda x \bullet f (x \ x))$$

$$= (\lambda z \bullet f (z \ z))(\lambda x \bullet f (x \ x))$$

$$\rightarrow_\beta f ((\lambda x \bullet f (x \ x)) \ (\lambda x \bullet f (x \ x)))$$

$$= f (Y f)$$

In the theory of $\lambda$-calculus this yields the fix edpointequation $Y f = f (Y f)$. Therefore, for every $f$, a fix edpoint $Y f$ can be obtained via application of the fix edpoint combinator $Y$. 
The Fixedpoint Combinator $Y$

Church’s fixedpoint combinator “$Y$”:

$$Y = \lambda f \bullet (\lambda x \bullet f (x \ x)) (\lambda x \bullet f (x \ x))$$

Proof of fixedpoint combinator property — for every $f$, the following holds:

$$Y f = (\lambda f \bullet (\lambda x \bullet f (x \ x)) (\lambda x \bullet f (x \ x))) f$$

$$\rightarrow_{\beta} (\lambda x \bullet f (x \ x)) (\lambda x \bullet f (x \ x))$$

$$= (\lambda z \bullet f (z \ z)) (\lambda x \bullet f (x \ x))$$

$$\rightarrow_{\beta} f ((\lambda x \bullet f (x \ x)) (\lambda x \bullet f (x \ x)))$$

$$= f (Y f)$$

In the theory of $\lambda$-calculus this yields the fixedpoint equation $Y f = f (Y f)$. Therefore, for every $f$, a fixedpoint $Y f$ can be obtained via application of the fixedpoint combinator $Y$.

All general fixedpoint combinators involve self-application like “$x \ x$”
The Fixedpoint Combinator $Y$

Church’s fixedpoint combinator “$Y$”:

$$Y = \lambda f \bullet (\lambda x \bullet f (x x))(\lambda x \bullet f (x x))$$

Proof of fixedpoint combinator property — for every $f$, the following holds:

$$Y f = (\lambda f \bullet (\lambda x \bullet f (x x))(\lambda x \bullet f (x x))) f$$

$$\rightarrow_{\beta} (\lambda x \bullet f (x x))(\lambda x \bullet f (x x))$$

$$= (\lambda z \bullet f (z z))(\lambda x \bullet f (x x))$$

$$\rightarrow_{\beta} f ((\lambda x \bullet f (x x)) (\lambda x \bullet f (x x)))$$

$$= f (Y f)$$

In the theory of $\lambda$-calculus this yields the fixedpoint equation $Y f = f (Y f)$. Therefore, for every $f$, a fixedpoint $Y f$ can be obtained via application of the fixedpoint combinator $Y$.

All general fixedpoint combinators involve self-application like “$x x$” — this possible in the untyped $\lambda$-calculus, but not in most typed systems.
General Fixedpoint Combinators

- In typed $\lambda$-calculi, no pure $\lambda$-term is a fixedpointcombinator
General Fixedpoint Combinators

- In typed $\lambda$-calculi, no pure $\lambda$-term is a fixedpoint combinator
- One can always extend the calculus:
General Fixedpoint Combinators

• In typed $\lambda$-calculi, no pure $\lambda$-term is a fixedpoint combinator

• One can always extend the calculus:
  – For at least some types $t$, the fixedpoint combinator $Y_t : (t \to t) \to t$ is added to the terms.
General Fixedpoint Combinators

• In typed $\lambda$-calculi, no pure $\lambda$-term is a fixedpoint combinator

• One can always extend the calculus:
  – For at least some types $t$, the fixedpoint combinator $Y_t : (t \to t) \to t$ is added to the terms.
  – The fixedpoint rules $Y_t \ f \rightarrow_Y f \ (Y_t \ f)$ are added to the rules.
General Fixedpoint Combinators

- In typed $\lambda$-calculi, no pure $\lambda$-term is a fix edpointcombinator

- One can always extend the calculus:
  - For at least some types $t$, the fix edpointcombinator $Y_t : (t \rightarrow t) \rightarrow t$ is added to the terms.
  - The fix edpointrules $Y_t f \rightarrow_Y f (Y_t f)$ are added to the rules.

- Note: This rule can give rise to non-termination with the left-most innermost strategy:

$$Y_{IN \rightarrow IN} \tau \ 3 \ \rightarrow_Y \ \tau (Y_{IN \rightarrow IN} \tau) \ 3 \ \rightarrow_Y \ \tau (\tau (Y_{IN \rightarrow IN} \tau)) \ 3 \ \rightarrow_Y \ ...$$
General Fixedpoint Combinators

- In typed $\lambda$-calculi, no pure $\lambda$-term is a fixedpoint combinator

- One can always extend the calculus:
  - For at least some types $t$, the fixedpoint combinator $Y_t : (t \to t) \to t$ is added to the terms.
  - The fixedpoint rules $Y_t f \to Y f (Y_t f)$ are added to the rules.

- **Note:** This rule can give rise to non-termination with the left-most innermost strategy:

$$Y_{\mathbb{IN} \to \mathbb{IN}} \tau \ 3 \to Y \tau (Y_{\mathbb{IN} \to \mathbb{IN}} \tau) \ 3 \to Y \tau (\tau (Y_{\mathbb{IN} \to \mathbb{IN}} \tau)) \ 3 \to Y \ldots$$

- We write “$Y$” also as fixedpoint combinator in a mathematical context
General Fixedpoint Combinators

- In typed $\lambda$-calculi, no pure $\lambda$-term is a fix edpoint combinator
- One can always extend the calculus:
  - For at least some types $t$, the fix edpoint combinator $Y_t : (t \rightarrow t) \rightarrow t$ is added to the terms.
  - The fix edpoint rules $Y_t f \rightarrow_Y f (Y_t f)$ are added to the rules.
- Note: This rule can give rise to non-termination with the left-most innermost strategy:
  
  \[
  Y_{IN \rightarrow IN} \tau \ 3 \rightarrow_Y \tau (Y_{IN \rightarrow IN} \tau) \ 3 \rightarrow_Y \tau (\tau (Y_{IN \rightarrow IN} \tau)) \ 3 \rightarrow_Y \ldots
  \]
- We write “$Y$” also as fix edpoint combinator in a mathematical context
- More precisely, we let “$Y F$” denote the least fixedpoint of $F$
General Fixedpoint Combinators

- In typed $\lambda$-calculi, no pure $\lambda$-term is a fixedpoint combinator.
- One can always extend the calculus:
  - For at least some types $t$, the fixedpoint combinator $Y_t : (t \rightarrow t) \rightarrow t$ is added to the terms.
  - The fixedpoint rules $Y_t \ f \rightarrow_y f \ (Y_t \ f)$ are added to the rules.
- **Note:** This rule can give rise to non-termination with the left-most innermost strategy:

  $Y_{IN \rightarrow IN} \tau \ 3 \rightarrow_y \tau \ (Y_{IN \rightarrow IN} \tau) \ 3 \rightarrow_y \tau \ (\tau \ (Y_{IN \rightarrow IN} \tau)) \ 3 \rightarrow_y \ldots$

- We write “$Y$” also as fixedpoint combinator in a mathematical context.
- More precisely, we let “$Y \ F$” denote the least fixedpoint of $F$.
- Other notations: “$\mu \ F$”, or “fix $F$”
Parameter Passing

Parameter passing has two sides — assume a parameterized subprogram $P$:

- **Formal Parameters:** The names used to refer to the parameters in the definition of $P$.
- **Actual Parameters:** The expressions supplied to $P$ as instances of its parameters for the purpose of creating an *incarnation* of $P$.

If several actual parameters are supplied to a subprogram defined with several formal parameters, how is a *correspondence* established?

- **Positional correspondence:** $n$-th actual parameter instantiates $n$-th formal parameter
- **Explicit parameter labels:** allow arbitrary order as far as labels are supplied (often positional fallback).

    Ada: Sub ( Y => B, X => 27 );
    OCaml: sub ~y:b ~x:27
Labelled Arguments in OCaml

```ocaml
data = x:int -> y:int -> int = <fun>                    — x and y are labels
#let x = 3 and y = 2 in f ~x ~y;; — x and y are labelled arguments — : int = 1
#let x = 3 and y = 2 in f ~y ~x;; — labelled arguments may be commuted — : int = 1
#let f ~x:x1 ~y:y1 = x1 - y1;;
val f : x:int -> y:int -> int = <fun>
  — x1 and y1 are formal parameters
  — x and y are labels
#f ~x:3 ~y:2;; — : int = 1  #f ~y:2 ~x:3;; — : int = 1
#f 3 2;; — labels can be omitted if all arguments are supplied! — : int = 1
```
Typical Application of Labelled Arguments in OCaml

```ocaml
ListLabels.fold_right : f:('a -> 'b -> 'b) -> 'a list -> init:'b -> 'b
val list_predsplit : f:('a -> bool) -> 'a list -> 'a list * 'a list
let list_predsplit ~f = ListLabels.fold_right ~init:([],[])          ~f:(fun x (xs,ys) -> if f x then (x :: xs,ys) else (xs,x::ys));;
val of_psp : ((f * 'a) list * f) -> 'a t
let of_psp (ps,tl) =   let d = singleton tl in
                        List.fold_right ~init:d ps    ~f:(fun p d -> let _ = fe_onto_start p d in d);;
...  (List.fold_right ns ~init:[] ~f:(fun n res ->
                      (try (match snd (Nodemap.find attrmap ~key:n) with
                      None -> res                  | Some (Left h) -> (n,h) :: res
                      | Some (Right q) -> res        )
                      with Not_found -> res       ));;
```
Optional Arguments in OCaml

```ocaml
#let bump ?(step = 1) x = x + step;;
val bump : ?step:int -> int -> int = <fun>
  — optional arguments have a label and a default value.

#bump 2;;
- : int = 3

#bump ~step:3 2;;
- : int = 5
```

A function taking some optional arguments must also take at least one non-labeled argument. (Important for partial applications)
OCamlBrowser
Evaluation Aspects of Parameter Passing

- **Call by value:** Actual parameter expression is evaluated to a value before instantiation of formal parameter.
  - leftmost-innermost strategy in $\lambda$-calculus
  - OCaml, C, Java, Oberon, Ada, …
  - strict function call semantics: undefined arguments always produce undefined results

- **Call by name:** Formal parameter $f$ is instantiated with unevaluated actual parameter expression $e$ — several occurrences of $f$ result in several copies of $e$.
  - leftmost-outermost strategy in $\lambda$-calculus

- **Lazy evaluation:** call by name with sharing instead of copying: $e$ is never duplicated; all occurrences of $f$ are instantiated with references to single $e$, which is evaluated at most once.

```ocaml
# const 4 (3 / 0);;
Exception: Division_by_zero.
```
- graph reduction implementation of leftmost-outermost
- most Haskell implementations
- **non-strict** function call semantics:
  undefined arguments **need not** produce undefined results

```haskell
Prelude> const 4 (3 / 0)
4
```
Storage Aspects of Parameter Passing

- **Call by constant value:** value available as local constant
- **Call by copy** (call by value): value copied into local variable
- **Call by reference:** actual parameter needs to be a reference, or define a reference; this is used as value of formal parameter.
- **Call by value-result:** actual parameter needs to be a reference, or define a reference \( r \); local variable \( l \) is initialized from \( r \)-value of \( r \), and on subprogram exit, \( r \) is overwritten with contents of \( l \).
- **Call by result:** Similar, but \( l \) starts out uninitialized.
Call by value, Call by reference, and Scoping

MODULE Scope1;
IMPORT Out;
VAR n : INTEGER;
PROCEDURE B(VAR x : INTEGER;
            z : INTEGER);
  VAR hv : INTEGER;
  BEGIN
    IF z = 0
    THEN x := 0
    ELSE hv := x;
         B(x, z-1);
         x := x+hv
    END;
  END B;
PROCEDURE A(y,x: INTEGER;
            VAR result: INTEGER);
  BEGIN
    IF x = 0
    THEN result:=1;
    ELSE A(y, x-1, result);
         B(result, y)
    END;
  END A;
BEGIN
  n := 0;
  A(2,1,n);
  Out.Int(n,0); Out.Ln
END Scope1.
Call by value, Call by reference, and Scoping

MODULE Scope1;
IMPORT Out;
VAR n : INTEGER;
PROCEDURE B(VAR x : INTEGER;
            z : INTEGER);
    VAR hv : INTEGER;
    BEGIN
        IF z = 0
        THEN x := 0
        ELSE hv := x;
            B(x, z-1);
            x := x+hv
        END;
    END B;

PROCEDURE A(y,x: INTEGER;
            VAR result: INTEGER);
    BEGIN
        IF x = 0
        THEN result:=1;
        ELSE A(y, x-1, result);
            B(result, y)
        END;
    END A;
BEGIN
    n := 0;
    A(2,1,n);
    Out.Int(n,0); Out.Ln
END Scope1.

\{x = a\} \quad B(x, z) \quad \{x = a \times z\}

\{\text{True}\} \quad A(y,x,\text{result}) \quad \{\text{result} = y^x\}
Parameter Passing

- **Formal parameters** — actual parameters

- **Correspondence aspects:** by position, by label, optional arguments

- **Evaluation aspects:** call by value, call by name, lazy evaluation

- **Storage aspects:** call by constant value, call by copy, call by reference, call by value-result, call by result
X