

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 (\rightarrow context-sensitive syntax):

- Scopes of names
- Typing

Semantics — *Meaning* of PL constructs

Three major approaches:

- **Axiomatic semantics**: $\{p\} \text{Prog} \{q\}$
- **Denotational semantics**: Prog denotes a mathematical function $\llbracket \text{Prog} \rrbracket$
- **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: $\mathbf{B} = \{\text{True}, \text{False}\}$, \mathbf{N} , \mathbf{Z} , Char , seq Char , Ident

Domains for Program States:

- **Locations** are usually natural numbers: $\text{Loc} = \mathbf{N}$
- **Values** are, in a simple context, integers: $\text{Val}_0 = \mathbf{Z}$
- **Memory states** can be considered as partial functions: $\text{Mem}_0 = \mathbf{N} \rightarrow \text{Val}_0$
- **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 $Q, R : \mathcal{A} \leftrightarrow \mathcal{B}$.

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 “ $\bar{\cup}$ ”
- **Haskell**: $\text{addListToMap} :: \text{Ord key} \Rightarrow \text{Map } k \ v \rightarrow [(k, v)] \rightarrow \text{Map } k \ v$
 $\text{addListToMap} = \text{foldr } (\text{uncurry } \text{Map.insert})$
- 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 σ produces value v $\sigma(e) \Rightarrow v$
- Execution of statement s starting in store σ_1 results in store σ_2 $\sigma_1(s) \Rightarrow \sigma_2$

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

Execution rules:

$$\frac{\text{premise}}{\text{conclusion}} \quad \text{or} \quad \frac{\text{premise}_1 \ \dots \ \text{premise}_n}{\text{conclusion}}$$

Example rule — **addition**:

$$\frac{\sigma(e_1) \Rightarrow v_1 \quad \sigma(e_2) \Rightarrow v_2}{\sigma(e_1 + e_2) \Rightarrow v_1 + v_2}$$

(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 σ produces value v $\sigma(e) \Rightarrow v$
- Execution of statement s starting in store σ_1 results in store σ_2 $\sigma_1(s) \Rightarrow \sigma_2$

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

- expression evaluation: $eval : State_1 \times Expr \mapsto Val_1$
- statement execution: $exec : State_1 \times Stmt \mapsto State_1$

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

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

$evalExpr :: Expression \rightarrow State1 \rightarrow Maybe Value1$
 $interpStmt :: Statement \rightarrow State1 \rightarrow Maybe State1$

data $Value1 = ValInt Int \mid ValBool Bool$

type $State1 = Map Variable Value1$ — even simpler than $State_0$

Interpreter: Expression Evaluation

$evalExpr :: Expression \rightarrow State1 \rightarrow Maybe Value1$

data $Value1 = ValInt Int$
 $\mid ValBool Bool$

type $State1 = Map Variable Value1$

$evalExpr (Var v) s = Map.lookup v s$
 $evalExpr (Value (LitInt i)) s = Just (ValInt i)$ — 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**
 ($Just (ValInt v1)$, $Just (ValInt v2)$) $\rightarrow Just (ValInt (v1 + v2))$
 — $\rightarrow Nothing$

Interpreter: Expression Evaluation (Maybe Monad)

$evalExpr :: Expression \rightarrow State1 \rightarrow Maybe Value1$

data $Value1 = ValInt Int$
 $\mid ValBool Bool$

type $State1 = Map Variable Value1$

$evalExpr (Var v) s = Map.lookup v s$
 $evalExpr (Value (LitInt i)) s = Just (ValInt i)$ — better: function litToVal
 $evalExpr (Value (LitBool b)) s = Just (ValBool b)$
 $evalExpr (Binary (MkArithOp Plus) e1 e2) s = \mathbf{do}$
 $ValInt v1 \leftarrow evalExpr e1 s$
 $ValInt v2 \leftarrow evalExpr e2 s$
 $Just (ValInt (v1 + v2))$

Assignment

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

For example:

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

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

$$\sigma_1(x := x + 3) \Rightarrow \{x \mapsto 42, y \mapsto 7\}$$

since $\sigma_1 \oplus \{x \mapsto 42\} = \{x \mapsto 42, y \mapsto 7\}$

Interpreter: Assignment

$evalExpr :: Expression \rightarrow State1 \rightarrow Maybe Value1$
 $interpStmt :: Statement \rightarrow State1 \rightarrow Maybe State1$

data $Value1 = ValInt Int$
 | $ValBool Bool$

type $State1 = Map Variable Value1$

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

$interpStmt$ (Assignment var e) $s = \mathbf{case}$ $evalExpr$ e s **of**
 $Just\ val \rightarrow Just$ (Map.insert var val s)
 $Nothing \rightarrow Nothing$

(Using the *Maybe* monad:)

$interpStmt$ (Assignment var e) $s = \mathbf{do}$
 $val \leftarrow evalExpr\ e\ s$
 $Just$ (Map.insert var val s)

Sequencing, Conditionals, Loops

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

$$\frac{\sigma(b) \Rightarrow \text{True} \quad \sigma(s_1) \Rightarrow \sigma_1}{\sigma(\mathbf{if}\ b\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2\ \mathbf{fi}) \Rightarrow \sigma_1}$$

$$\frac{\sigma(b) \Rightarrow \text{False} \quad \sigma(s_2) \Rightarrow \sigma_2}{\sigma(\mathbf{if}\ b\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2\ \mathbf{fi}) \Rightarrow \sigma_2}$$

$$\frac{\sigma(b) \Rightarrow \text{True} \quad \sigma(s) \Rightarrow \sigma_1 \quad \sigma_1(\mathbf{while}\ b\ \mathbf{do}\ s\ \mathbf{od}) \Rightarrow \sigma_2}{\sigma(\mathbf{while}\ b\ \mathbf{do}\ s\ \mathbf{od}) \Rightarrow \sigma_2}$$

$$\frac{\sigma(b) \Rightarrow \text{False}}{\sigma(\mathbf{while}\ b\ \mathbf{do}\ s\ \mathbf{od}) \Rightarrow \sigma}$$

Loop Example

$P \equiv \mathbf{while}\ x < 50\ \mathbf{do}\ x := 2 * x\ \mathbf{od}$

$$\frac{\frac{\frac{\frac{\sigma(x \mapsto 28) (x < 50) \Rightarrow \text{True} \quad \sigma(x \mapsto 28) (x := 2 * x) \Rightarrow \{x \mapsto 56\}}{\sigma(x \mapsto 28)(P) \Rightarrow \{x \mapsto 56\}} \quad \frac{\sigma(x \mapsto 56) (x < 50) \Rightarrow \text{False}}{\sigma(x \mapsto 56)(P) \Rightarrow \{x \mapsto 56\}}}{\sigma(x \mapsto 14) (x < 50) \Rightarrow \text{True} \quad \sigma(x \mapsto 14) (x := 2 * x) \Rightarrow \{x \mapsto 28\}} \quad \sigma(x \mapsto 28)(P) \Rightarrow \{x \mapsto 56\}}{\sigma(x \mapsto 7) (x < 50) \Rightarrow \text{True} \quad \sigma(x \mapsto 7) (x := 2 * x) \Rightarrow \{x \mapsto 14\}} \quad \sigma(x \mapsto 14)(P) \Rightarrow \{x \mapsto 56\}}{\sigma(x \mapsto 7)(P) \Rightarrow \{x \mapsto 56\}}$$

Interpreter: Sequencing

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

This corresponds to a special case of our Jay ASTs:

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

General case:

$interpStmt$ (MkBlock $stmts$) = $\lambda s \rightarrow interpBlock\ stmts\ s$

$interpBlock :: [Statement] \rightarrow (State1 \rightarrow Maybe State1)$

$interpBlock [] = Just$

$interpBlock$ ($stmt : stmts$) = $\lambda s \rightarrow \mathbf{case}\ interpStmt\ stmt\ s$ **of**

$Just\ s1 \rightarrow interpBlock\ stmts\ s1$

$Nothing \rightarrow Nothing$

Interpreter: Loops

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

$$\frac{\sigma(b) \Rightarrow \text{False}}{\sigma(\text{while } b \text{ do } s \text{ od}) \Rightarrow \sigma}$$

$\text{interpStmt} (\text{Loop cond body}) = \lambda s \rightarrow \text{case} (\text{evalExpr cond}) s \text{ of}$
 $\text{Just} (\text{ValBool False}) \rightarrow \text{Just } s$
 $\text{Just} (\text{ValBool True}) \rightarrow \text{case} (\text{interpStmt body}) s \text{ of}$
 $\text{Just } s1 \rightarrow (\text{interpStmt} (\text{Loop cond body})) s1$
 $\text{Nothing} \rightarrow \text{Nothing}$
 $\text{Just} (\text{ValInt } i) \rightarrow \text{Nothing}$
 $\text{Nothing} \rightarrow \text{Nothing}$

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

“ $\text{interpStmt} (\text{Loop cond body})$ ” occurs also on the right-hand side.

Additional Control Structures

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

Options:

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

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

- **Translation into core language** — *derived* features

$$\frac{\sigma_1(s ; \text{while } b \text{ do } s \text{ od}) \Rightarrow \sigma_2}{\sigma_1(\text{do } s \text{ while } (c)) \Rightarrow \sigma_2}$$

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: $\sigma(e) \Rightarrow (\sigma', v)$
- “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 “**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: $\text{Print} :: \text{Expression} \rightarrow \text{Statement}$
- New state space: $\text{State}_2 = \text{State}_1 \times [\mathbb{Z}]$

- New statement assertion schema: $(\sigma_1, \text{out}_1)(s) \Rightarrow (\sigma_2, \text{out}_2)$

- Adapted rules, e.g.:

$$\frac{\sigma(e) \Rightarrow v}{(\sigma, \text{out})(x := e) \Rightarrow (\sigma \oplus \{x \mapsto v\}, \text{out})}$$

- **Rules for new feature:**

$$\frac{\sigma(e) \Rightarrow i}{(\sigma, \text{out})(\text{print } (e)) \Rightarrow (\sigma, \text{out} ++ [i])}$$

- Check determinism, add to interpreter.

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

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.*, ...

- **RuntimeException**: Abnormal runtime events which need not be declared with a `throws` clause:

ArithmeticExc., *ClassCastExc.*, *IllegalArgumentExc.*, *IndexOutOfBoundsExc.*, *NullPointerException.*, ...

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 ( Exception1 var1 ) {
    catch1 body
}
catch ( Exception2 var2 ) {
    catch2 body
}
...
catch ( Exceptionn varn ) {
    catchn 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

- 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

Originally: Two kinds of assertions:

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

$\sigma_1(s) \Rightarrow \sigma_2$ — execution of statement s starting in state σ_1 can successfully terminate in state σ_2

Now an additional possibility:

$\sigma_1(s) \overset{\dagger}{\Rightarrow} (\sigma_2, x)$ — execution of statement s starting in state σ_1 can terminate in state σ_2 **raising exception** x

Two additional sequencing rules:

$$\frac{\sigma_1(s_1) \overset{\dagger}{\Rightarrow} (\sigma_2, z)}{\sigma_1(s_1; s_2) \overset{\dagger}{\Rightarrow} (\sigma_2, z)} \qquad \frac{\sigma_1(s_1) \Rightarrow \sigma_2 \qquad \sigma_2(s_2) \overset{\dagger}{\Rightarrow} (\sigma_3, z)}{\sigma_1(s_1; s_2) \overset{\dagger}{\Rightarrow} (\sigma_3, z)}$$

Two additional if rules (no exceptions in expression evaluation yet):

$$\frac{\sigma(b) \Rightarrow \text{True} \qquad \sigma(s_1) \overset{\dagger}{\Rightarrow} (\sigma_1, x)}{\sigma(\text{if } b \text{ then } s_1 \text{ else } s_2 \text{ fi}) \overset{\dagger}{\Rightarrow} (\sigma_1, x)} \qquad \frac{\sigma(b) \Rightarrow \text{False} \qquad \sigma(s_2) \overset{\dagger}{\Rightarrow} (\sigma_2, x)}{\sigma(\text{if } b \text{ then } s_1 \text{ else } s_2 \text{ fi}) \overset{\dagger}{\Rightarrow} (\sigma_2, x)}$$

Exceptions — Interpreter

Original statement interpretation:

$interpStmt :: Statement \rightarrow State1 \rightarrow Maybe\ State1$

meaning:

$$\begin{aligned} \sigma_1(s) \Rightarrow \sigma_2 & \quad \mathbf{iff} \quad interpStmt\ s\ \sigma_1 = Just\ \sigma_2 \\ \neg \exists \sigma_2 \bullet \sigma_1(s) \Rightarrow \sigma_2 & \quad \mathbf{iff} \quad interpStmt\ s\ \sigma_1 \in \{\perp, Nothing\} \end{aligned}$$

Statement interpretation **with exceptions**:

$interpStmtExc :: Statement \rightarrow State1 \rightarrow Maybe\ (Either\ State1\ (State1,\ Exc))$

meaning:

$$\begin{aligned} \sigma_1(s) \Rightarrow \sigma_2 & \quad \mathbf{iff} \quad interpStmt\ s\ \sigma_1 = Just\ (Left\ \sigma_2) \\ \sigma_1(s) \overset{!}{\Rightarrow} (\sigma_2, x) & \quad \mathbf{iff} \quad interpStmt\ s\ \sigma_1 = Just\ (Right\ (\sigma_2, x)) \\ \neg \exists \sigma_2, x \bullet \sigma_1(s) \Rightarrow \sigma_2 \vee & \\ \sigma_1(s) \overset{!}{\Rightarrow} (\sigma_2, x) & \quad \mathbf{iff} \quad interpStmt\ s\ \sigma_1 \in \{\perp, Nothing\} \end{aligned}$$