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:** $\{p\}$ `Prog` $\{q\}$
– **Denotational semantics:** `Prog` *denotes* a *mathematical* function $[\![\texttt{Prog}]\!]$
– **Operational semantics:** state transitions of an abstract machine

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.

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

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

- ***Textbook:*** "overriding union" operator "$\overline{\mathrm{U}}$"
- ***Haskell:*** $\mathit{addListToMap} :: \mathit{Ord}\ \mathit{key} \Rightarrow \mathit{Map}\ k\ v \rightarrow [\,(k,\ v)\,] \rightarrow \mathit{Map}\ k\ v$
  $\mathit{addListToMap} = \mathit{foldr}\ (\mathit{uncurry}\ \mathit{Map.insert})$
- 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)

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: $\mathbb{B} = \{\mathsf{True}, \mathsf{False}\}$, $\mathbb{N}$, $\mathbb{Z}$, Char, $\mathsf{seq}\ \mathit{Char}$, Ident

Domains for Program States:

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

Operational Semantics

Two kinds of assertions:

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

— Execution of statement s starting in store σ_1 results in store σ_2 $\qquad \sigma_1(s) \Rightarrow \sigma_2$

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

Execution rules:

$$\frac{\mathit{premise}}{\mathit{conclusion}} \qquad \text{or} \qquad \frac{\mathit{premise}_1 \ \ldots\ \mathit{premise}_n}{\mathit{conclusion}}$$

Example rule — **addition:**

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

(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 $\qquad \sigma(e) \Rightarrow v$
— Execution of statement s starting in store σ_1 results in store σ_2 $\qquad \sigma_1(s) \Rightarrow \sigma_2$

This notation stands for two **ternary relations**, which are **partial functions** for deterministic programming languages:
— expression evaluation: $\quad eval : State_1 \times Expr \nrightarrow Val_1$
— statement execution: $\quad exec : State_1 \times Stmt \nrightarrow State_1$

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

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

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

data Value1 = ValInt Int | ValBool Bool
type State1 = Map Variable Value1 -- even simpler than State0
```


Interpreter: Expression Evaluation

```
evalExpr :: Expression → State1 → Maybe Value1

data Value1 = ValInt Int
        | 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)) → Just (ValInt (v1 + v2))
  _                                    → Nothing
```


Interpreter: Expression Evaluation (*Maybe* Monad)

```
evalExpr :: Expression → State1 → Maybe Value1

data Value1 = ValInt Int
        | 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 = do
  ValInt v1 ← evalExpr e1 s
  ValInt v2 ← 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:

$$\frac{\dfrac{\sigma_1(x) \Rightarrow 39 \qquad\qquad \sigma_1(3) \Rightarrow 3}{\sigma_1(x+3) \Rightarrow 42}}{\sigma_1(x := x+3) \Rightarrow \{x \mapsto 42, y \mapsto 7\}}$$

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

Interpreter: Assignment

evalExpr :: *Expression* → *State1* → *Maybe Value1*
interpStmt :: *Statement* → *State1* → *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* = **case** *evalExpr e s* **of**
  Just val → *Just* (*Map.insert var val s*)
  Nothing → *Nothing*

(Using the *Maybe* monad:)

interpStmt (*Assignment var e*) *s* = **do**
  val ← *evalExpr e s* **of**
  Just (*Map.insert var val s*)

Sequencing, Conditionals, Loops

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

$$\frac{\sigma(b) \Rightarrow \mathsf{True} \qquad \sigma(s_1) \Rightarrow \sigma_1}{\sigma(\textbf{if } b \textbf{ then } s_1 \textbf{ else } s_2 \textbf{ fi}) \Rightarrow \sigma_1}$$

$$\frac{\sigma(b) \Rightarrow \mathsf{False} \qquad \sigma(s_2) \Rightarrow \sigma_2}{\sigma(\textbf{if } b \textbf{ then } s_1 \textbf{ else } s_2 \textbf{ fi}) \Rightarrow \sigma_2}$$

$$\frac{\sigma(b) \Rightarrow \mathsf{True} \qquad \sigma(s) \Rightarrow \sigma_1 \qquad \sigma_1(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2}{\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2}$$

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

Loop Example

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

$$\dfrac{\{x \mapsto 7\}\,(x < 50) \Rightarrow \mathsf{True} \qquad \{x \mapsto 7\}\,(x := 2 * x) \Rightarrow \{x \mapsto 14\} \qquad \dfrac{\{x \mapsto 14\}\,(x < 50) \Rightarrow \mathsf{True} \qquad \{x \mapsto 14\}\,(x := 2 * x) \Rightarrow \{x \mapsto 28\} \qquad \dfrac{\{x \mapsto 28\}\,(x < 50) \Rightarrow \mathsf{True} \qquad \{x \mapsto 28\}\,(x := 2 * x) \Rightarrow \{x \mapsto 56\} \qquad \dfrac{\{x \mapsto 56\}\,(x < 50) \Rightarrow \mathsf{False}}{\{x \mapsto 56\}(P) \Rightarrow \{x \mapsto 56\}}}{\{x \mapsto 28\}(P) \Rightarrow \{x \mapsto 56\}}}{\{x \mapsto 14\}(P) \Rightarrow \{x \mapsto 56\}}}{\{x \mapsto 7\}(P) \Rightarrow \{x \mapsto 56\}}$$

$$\frac{\sigma(b) \Rightarrow \mathsf{True} \qquad \sigma(s) \Rightarrow \sigma_1 \qquad \sigma_1(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2}{\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2} \qquad \frac{\sigma(b) \Rightarrow \mathsf{False}}{\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma}$$

Interpreter: Sequencing

$$\frac{\sigma_1(s_1) \Rightarrow \sigma_2 \qquad \sigma_2(s_2) \Rightarrow \sigma_3}{\sigma_1(s_1 \mathbin{\textbf{;}} s_2) \Rightarrow \sigma_3}$$

This corresponds to a special case of our Jay ASTs:

interpStmt (*MkBlock* [*stmt1*, *stmt2*]) = λ *s* → **case** (*interpStmt stmt1*) *s* **of**
  Just s1 → (*interpStmt stmt2*) *s1*
  Nothing → *Nothing*

General case:

interpStmt (*MkBlock stmts*) = λ *s* → *interpBlock stmts s*

interpBlock :: [*Statement*] → (*State1* → *Maybe State1*)
interpBlock [ ] = *Just*
interpBlock (*stmt* : *stmts*) = λ *s* → **case** *interpStmt stmt s* **of**
  Just s1 → *interpBlock stmts s1*
  Nothing → *Nothing*

Interpreter: Loops

$$\frac{\sigma(b) \Rightarrow \mathsf{True} \qquad \sigma(s) \Rightarrow \sigma_1 \qquad \sigma_1(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2}{\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2}$$

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

```
interpStmt ( Loop cond body ) = λ s → case ( evalExpr cond ) s of
  Just ( ValBool False ) → Just s
  Just ( ValBool True ) → case ( interpStmt body ) s of
    Just s1 → ( interpStmt ( Loop cond body ) ) s1
    Nothing → Nothing
  Just ( ValInt i ) → Nothing
  Nothing → Nothing
```

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

"*interpStmt (Loop cond body)*" occurs also on the right-hand side.

Additional Control Structures

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

Options:

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

$$\frac{\sigma_1(s) \Rightarrow \sigma_2 \qquad \sigma_2(b) \Rightarrow \mathsf{False}}{\sigma_1(\textbf{ do } s \textbf{ while } (b)) \Rightarrow \sigma_2} \qquad \frac{\sigma_1(s) \Rightarrow \sigma_2 \qquad \sigma_2(b) \Rightarrow \mathsf{True} \qquad \sigma_2(\textbf{ do } s \textbf{ while } (b)) \Rightarrow \sigma_3}{\sigma_1(\textbf{ do } s \textbf{ while } (b)) \Rightarrow \sigma_3}$$

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

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

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: *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.: $$\frac{\sigma(e) \Rightarrow v}{(\sigma, out)(x := e) \Rightarrow (\sigma \oplus \{x \mapsto v\}, out)}$$
- **Rules for new feature:** $$\frac{\sigma(e) \Rightarrow i}{(\sigma, out)(\textbf{ print } (e)) \Rightarrow (\sigma, out \mathbin{+\!\!+} [i])}$$
- 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.*, *NullPointerExc.*, …

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) {}  } }
}}
```


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;
  }
}
```


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!*

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:

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

Two additional sequencing rules:

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

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

$$\frac{\sigma(b) \Rightarrow \mathsf{True} \qquad \sigma(s_1) \stackrel{!}{\Rightarrow} (\sigma_1, x)}{\sigma(\textbf{if } b \textbf{ then } s_1 \textbf{ else } s_2 \textbf{ fi}) \stackrel{!}{\Rightarrow} (\sigma_1, x)} \qquad \frac{\sigma(b) \Rightarrow \mathsf{False} \qquad \sigma(s_2) \stackrel{!}{\Rightarrow} (\sigma_2, x)}{\sigma(\textbf{if } b \textbf{ then } s_1 \textbf{ else } s_2 \textbf{ fi}) \stackrel{!}{\Rightarrow} (\sigma_2, x)}$$

Exceptions — Interpreter

Original statement interpretation:

$interpStmt :: Statement \to State1 \to Maybe\ State1$

meaning:

$$\sigma_1(s) \Rightarrow \sigma_2 \quad \textbf{iff} \quad interpStmt\ s\ \sigma_1 = Just\ \sigma_2$$
$$\neg\exists\sigma_2 \bullet \sigma_1(s) \Rightarrow \sigma_2 \quad \textbf{iff} \quad interpStmt\ s\ \sigma_1 \in \{\bot,\ Nothing\}$$

Statement interpretation **with exceptions**:

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

meaning:

$$\sigma_1(s) \Rightarrow \sigma_2 \quad \textbf{iff} \quad interpStmt\ s\ \sigma_1 = Just\ (Left\ \sigma_2)$$
$$\sigma_1(s) \overset{!}{\Rightarrow} (\sigma_2, x) \quad \textbf{iff} \quad interpStmt\ s\ \sigma_1 = Just\ (Right\ (\sigma_2,\ x))$$
$$\neg\exists\sigma_2, x \bullet\ \sigma_1(s) \Rightarrow \sigma_2\ \vee$$
$$\sigma_1(s) \overset{!}{\Rightarrow} (\sigma_2, x) \quad \textbf{iff} \quad interpStmt\ s\ \sigma_1 \in \{\bot,\ Nothing\}$$

Exceptions in Expression Evaluation

Exercise