

## Programming Languages

- **What Is a Programming Language?**  
Syntax, Semantics, Pragmatics, Implementations, Support
- **Programming Language Paradigms**  
Imperative, OO, Functional, Logic
- **Historical Development**  
From *making it easy for the machine*  
to *making it easier for the programmer*
- **Criteria for Programming Language Evaluation**  
Readability, Conceptual Integrity, Abstraction, Verification, Cost
- **Design of New Programming Languages**  
Domain-Specific Languages (DSLs) proliferate ...

## 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  $\llbracket \text{Prog} \rrbracket$
- **Operational semantics**: state transitions of an abstract machine

## Syntax, Grammars, Lexing, Parsing

- **Formal languages**: generated by grammars, accepted by automata
- Chomsky-3: Regular languages, regular grammars, regular expressions, (D/N)FSA (railroad diagrams)
  - **Lexical level**: keywords, identifiers, literals (numbers, strings)  
Lexer generators: `lex`, `flex`, `Alex`, `ocamllex`
- Chomsky-2: Context-free languages, CFG ((E)BNF), (D)PDA
  - **Syntax level**: expressions, statements, declarations, modules, ...  
LL Grammars: Recursive descent parsing  
Parser generators: `yacc`, `bison`, `javacc`, `ANTLR`, `happy`, `ocamlyacc`

## Parsing and Abstract Syntax

- Parsers are based on the **concrete syntax**
- Parsers **accept** token strings as defined by the **concrete grammar**
- Parsers **implicitly** construct a derivation tree for the **concrete grammar**
- Parsers are usually directed to construct **abstract syntax trees**
- Abstract syntax trees **represent the program for compilation**
- Abstract syntax trees represent the program for **all semantical purposes**
- When comparing programming languages:

*Which constructs are available in the abstract syntax?*

## Abstract Syntax

48 / y / x

(48 / y) / x

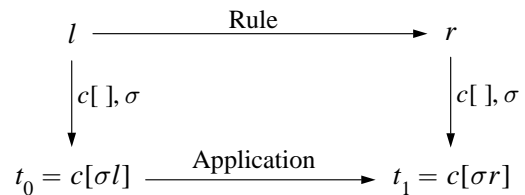
- **different** derivations in expression grammar
- **different** parse trees
- “**essentially the same**” expression
- after parsing, the **same datastructure** in the compiler:
- identical **abstract syntax trees**

### Abstract syntax trees

- are, in a certain sense, “shortcuts” of the concrete syntax trees
- contain only the semantically relevant part of the program structure
- correspond to (abstract datatype of) **terms** in logic
- serve as bridge between syntax and semantics
  - in language definitions
  - in compilers

## Term Rewriting and $\lambda$ -Calculus

- **Term Rewriting Systems** (TRSs) define named functions
- Application of term rewriting rule  $l \rightarrow r$ :



- $\lambda$ -abstraction defines **anonymous functions** using **variable binding**
- $\beta$ -Reduction:  $c[\ (\lambda x \bullet B) A \ ] \rightarrow_{\beta} c[\ B[x \setminus A] \ ]$
- Term rewriting and  $\lambda$ -calculus can serve as **operational semantics** for (side-effect free) expression evaluation and functional programming
  - Leftmost-outermost strategy: call by name — terminates if possible
  - Leftmost-innermost strategy: call by value — easier to implement

## Type Systems

- **Static typing** allows a range of **compile-time** consistency checks.
- Automatic conversions (casts) undermine the safety of static typing systems
- Polymorphism allows abstraction over types:
  - **Parametric polymorphism**: type parameters can be instantiated arbitrarily
  - **Ad-hoc polymorphism (overloading)**: type parameters can be instantiated be members of a given set of types.
  - **Subtyping**: involves a **subtype ordering** on the set of types, and every type defines the *cone* of its subtypes that are also acceptable in its place.
    - Subtyping in PVS is quite different form subtyping in Java!
- Type languages can have very different complexity and expressivity (FORTRAN — Haskell)
- **Type inference**: easy for the programmer, hard for the machine
- **Type annotations**: part of documentation, easy for the machine

## 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:  $eval : State_1 \times Expr \mapsto Val_1$
- 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  $State_0$

## 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 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  $\sigma_2 \models Q$

## Denotational Semantics (*not on the Final*)

- Mappings from **abstract syntax** to **semantic domains**
- Simple deterministic imperative programs:  $\llbracket \_ \rrbracket_S : Stmt \rightarrow (State \rightarrow State)$
- Expressions with side-effects:  $\llbracket \_ \rrbracket_E : Expr \rightarrow (State \rightarrow (State \times Val))$
- Non-deterministic imperative programs:  $\llbracket \_ \rrbracket_S : Stmt \rightarrow (State \leftrightarrow State)$
- Modelling complex programming language features requires complex mathematical constructions
- Semantics of `while` and recursion is defined as a **fixedpoint** that can be **approximated** starting from the undefined function
- Functional programming languages are “closer” to their denotational semantics

## Semantics and Language Design

- We **use the rules** of axiomatic semantics to prove properties of programs
- We **use the rules** 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

## 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

## Data Types

- **Abstract data types:** sorts (types), operations on these sorts, and laws describing how the operations interact
- **Concrete data structures:** mathematical models of carrier sets and operations
- Elementary Data Types: Scalars, Characters, elements of small sets, Strings
- Basic mathematical data type constructions:
  - **Aggregation:** cartesian product  $A \times B$  (tuples, records)
  - **Alternative:** disjoint union (direct sum)  $A + B$  (*Either*, variants, tagged unions)
  - Optional values  $\mathbb{1} + A$ , (*Maybe*)
- Frequently available **predefined data type constructions:** arrays, lists, sequences, stacks, queues, finite maps (dictionaries)
- **Data type implementation:** mapping mathematical constructions to constructions offered by programming language of libraries

## Modular Program Organisation

- **Modules:** grouping and encapsulating sets of program components, defining an **access interface**
- **Classes:** encapsulating the internal state of kinds of *objects*, defining **access interfaces**, fine-grained export control
- **Java Interfaces:** defining an **access interface** without restricting the kinds of objects that may provide it (similar to Haskell type classes)
- **Java Packages:** a kind of modules that can contain only class and interface definitions, coarse export control
- **Module parameterization:** Modules that can have **different instantiations** based on different implementations of **parameter interfaces**  
Ada: generic packages, \*ML: functors

## Outlook

- **Concurrent Programming** — 3B04
- **Relational Programming in Databases** — 3H03
- **Parallel and Distributed Programming** — 4F03
- **Computability** (why can't the compiler catch nontermination?) — 4I03
- **Formal Languages and Automata** — 4I03
- **Compilers** — CS 4TB3

...