

Introduction to Programming Language Semantics

Zoe Paraskevopoulou

School of Electrical and Computer Engineering
National Technical University of Athens

2025



What is a Programming Language?

A **programming language** is defined by:

- Its **syntax**
 - A context-free grammar that recognizes a formal language
- Its **static semantics**
 - Checked **statically** by the language implementation
 - E.g. scoping, typing
- Its **dynamic semantics**
 - What a program does when it **executes**
 - E.g. what does `(print "hello "; 6)*(print "world "; 7)` do?

What is a Programming Language?

A **programming language** is defined by:

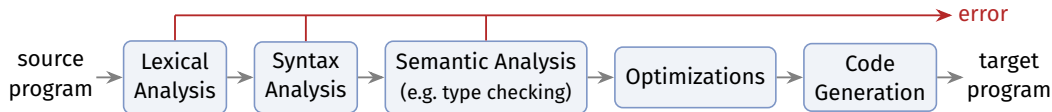
- Its **syntax**
 - A context-free grammar that recognizes a formal language
- Its **static semantics**
 - Checked **statically** by the language implementation
 - E.g. scoping, typing
- Its **dynamic semantics**
 - What a program does when it **executes**
 - E.g. what does `(print "hello "; 6)*(print "world "; 7)` do?

```
# (print_string "hello "; 6) * (print_string "world"; 7);;  
worldhello - : int = 42
```

How is a Programming Language Implemented?

- **Compiler**

- Translation to machine language
- E.g. C, C++, Rust, Haskell



- **Interpreter**

- A program that executes another program
- Usually input is transformed to intermediate representation (e.g. Python, JavaScript)

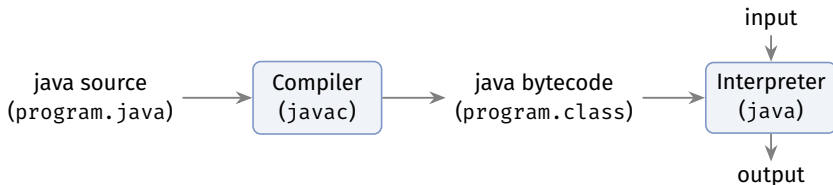
- **Transpiler**

- Source-to-source transformation
- E.g. Rocq to OCaml

How is a Programming Language Implemented?

- **Compiler**

- Translation to machine language
- E.g. C, C++, Rust, Haskell



- **Interpreter**

- A program that executes another program
- Usually input is transformed to intermediate representation (e.g. Python, JavaScript)

- **Transpiler**

- Source-to-source transformation
- E.g. Rocq to OCaml

Static Semantics

- Syntactic checking of *semantic* properties of the program
 - Correct use of names (scoping), correct use of types (type checking)
- Rejection of programs that would lead to errors:
 - `1 + "a"`
 - `if "this is a string" then 42 else 43`
- Conservative approach (may reject programs that do not lead to errors):
 - `1 + if true then 1 else "a string"`
 - `1 + if complex_computation then 1 else "a string"`
- Type safety: *Well-typed programs cannot go wrong* (Robin Milner)

A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute.

Types and Programming Languages

Benjamin C. Pierce

Dynamic Semantics

- The behavior of an executing program
- Does this program:
 - Terminate?
 - Throw a runtime error?
 - Compute the GCD?
- **Rice's theorem:** all non-trivial semantic properties of programs are undecidable
- Using the formal semantics of a language we can prove a semantic property for a program
- This process can *to some extent* be automated

Dynamic Semantics

- The behavior of an executing program
- Does this program:
 - Terminate?
 - Throw a runtime error?
 - Compute the GCD?
- **Rice's theorem:** all non-trivial semantic properties of programs are undecidable
- Using the formal semantics of a language we can prove a semantic property for a program
- This process can *to some extent* be automated

There is no **sound** and **complete** static method that can determine whether a program has a semantic property.

How is the Semantics of a Programming Language Defined?

- Syntax is **formally** defined by a **context-free grammar**. The spec is implemented by the parser.
- **What about semantics?**
 - The **language implementation** gives **semantics** to programs
 - But there can be **multiple implementations** that may **differ!**
 - Hard to **understand**
 - Cannot be used for **analyzing program behavior**
 - Most languages have a **specification** (language specification/standard) that defines semantics in natural language (e.g. ANSI C)
 - Often **imprecise** and **ambiguous**
 - A **formal semantics** gives **precise** and **clear** semantics to a programming language

Languages with Formal Semantics

- **ML**
 - The language of one of the first automated theorem provers (LCF, Robin Milner)
 - The first language with parametric polymorphism
 - Formal proof that a well-typed ML program does not lead to runtime errors (type safety, Milner 1978)
 - $\mathcal{W}$: type inference algorithm for ML

A Semantic Soundness Theorem (based on a formal semantics for the language) states that well-typed programs cannot “go wrong” and a Syntactic Soundness Theorem states that if $\mathcal{W}$ accepts a program then it is well typed.

— A Theory of Type Polymorphism in Programming, Robin Milner

Languages with Formal Semantics

- **WebAssembly**

- Low-level language (bytecode) implemented by all major web browsers (and beyond)
- Formal semantics (static and dynamic) and type safety proof
(*Bringing the Web up to Speed with WebAssembly*, Haas et al., 2017)



Soundness

The [type system](#) of WebAssembly is *sound*, implying both *type safety* and *memory safety* with respect to the WebAssembly semantics.

- Rocq, C (subset, via CompCert), PROLOG, Scheme, ...

Approaches to Defining Dynamic Semantics

- **Operational Semantics** ✓

Definition of an “abstract machine” describing program execution

- **Axiomatic Semantics** ✓

Definition of a set of rules (axioms) with properties that must hold for program state before and after execution of a statement

- **Denotational Semantics**

The meaning of a program is defined as a mathematical object (e.g., number, function)

✓ will be covered in this course

Abstract Syntax Tree

- A tree representation of the structure of a program or expression of a formal language
- Does not describe all grammar details (e.g., parentheses, comments, ...), only the structure
- Widely used in language implementation, program analysis, defining formal semantics, ...

Abstract Syntax Tree: Example

Grammar: $e := n \mid e + e \mid e * e \mid (e), \quad n \in \mathbb{Z}$

AST representation in functional languages:

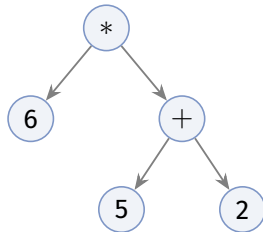
OCaml:

```
type expr =  
  | Plus of expr * expr  
  | Mul of expr * expr  
  | Num of int
```

Rocq:

```
Inductive expr : Type :=  
  | Plus : expr → expr → expr  
  | Mul  : expr → expr → expr  
  | Num  :  $\mathbb{Z}$  → expr.
```

The expression $(6 * (5 + 2))$ is represented as an abstract syntax tree:



Dynamic Semantics: Arithmetic Expressions

AST of simple arithmetic expressions:

$$e := n \mid e + e \mid e * e, \quad n \in \mathbb{Z}$$

What does $(6 * (5 + 2))$ mean?

We will define the **dynamic semantics** of the language.

Operational Semantics

We can formally define a mathematical relation describing the execution of an expression e . Two main approaches:

- **Big-step** or **natural** semantics: $e \Downarrow v$
Expression e evaluates to value v . Describes the final result. Resembles an interpreter.
- **Small-step** semantics: $e \rightarrow e'$
Expression e takes one execution step producing e' . Final result produced by a sequence of such steps describing intermediate stages.

Big-Step (Natural) Semantics

$$\frac{}{n \Downarrow n} \text{ NUM}$$

$$\frac{e_1 \Downarrow n_1 \quad e_2 \Downarrow n_2}{e_1 + e_2 \Downarrow n_1 +_{\mathbb{Z}} n_2} \text{ ADD}$$

$$\frac{e_1 \Downarrow n_1 \quad e_2 \Downarrow n_2}{e_1 * e_2 \Downarrow n_1 *_{\mathbb{Z}} n_2} \text{ MUL}$$

Big-Step (Natural) Semantics

$$\frac{}{n \Downarrow n} \text{ NUM}$$

$$\frac{e_1 \Downarrow n_1 \quad e_2 \Downarrow n_2}{e_1 + e_2 \Downarrow n_1 +_{\mathbb{Z}} n_2} \text{ ADD}$$

$$\frac{e_1 \Downarrow n_1 \quad e_2 \Downarrow n_2}{e_1 * e_2 \Downarrow n_1 *_{\mathbb{Z}} n_2} \text{ MUL}$$

Note: $+$ and $+_{\mathbb{Z}}$ are **different**! The first is syntax, the second is the *mathematical operation*. Similarly for $*$ and $*_{\mathbb{Z}}$.



Big-Step Semantics of an Expression

Derivation tree for $(6 * (5 + 2)) \Downarrow 42$:

$$\frac{\frac{\frac{}{6 \Downarrow 6} \text{ NUM} \quad \frac{\frac{\frac{}{5 \Downarrow 5} \text{ NUM} \quad \frac{\frac{}{2 \Downarrow 2} \text{ NUM}}{5 + 2 \Downarrow 7} \text{ ADD}}{6 * (5 + 2) \Downarrow 42} \text{ MUL}}{}}{}}$$

Small-Step Semantics

$$\frac{}{n_1 + n_2 \rightarrow n_1 +_{\mathbb{Z}} n_2} \text{ ADD}$$

$$\frac{e_1 \rightarrow e'_1}{e_1 + e_2 \rightarrow e'_1 + e_2} \text{ LADD}$$

$$\frac{e_2 \rightarrow e'_2}{n_1 + e_2 \rightarrow n_1 + e'_2} \text{ RADD}$$

$$\frac{}{n_1 * n_2 \rightarrow n_1 *_{\mathbb{Z}} n_2} \text{ MUL}$$

$$\frac{e_1 \rightarrow e'_1}{e_1 * e_2 \rightarrow e'_1 * e_2} \text{ LMUL}$$

$$\frac{e_2 \rightarrow e'_2}{n_1 * e_2 \rightarrow n_1 * e'_2} \text{ RMUL}$$

Small-Step Semantics of an Expression

Step 1:

$$\frac{\frac{}{5 + 2 \rightarrow 7} \text{ ADD}}{6 * (5 + 2) \rightarrow 6 * 7} \text{ RMUL}$$

Step 2:

$$\frac{}{6 * 7 \rightarrow 42} \text{ MUL}$$

Finally:

$$6 * (5 + 2) \rightarrow 6 * 7 \rightarrow 42$$