Interpretation and Compilation of Languages

Master Programme in Computer Science

Mário Pereira mjp.pereira@fct.unl.pt

Nova School of Science and Technology, Portugal

March 15, 2025

Lecture 2

based on lectures by Jean-Christophe Filliâtre and Léon Gondelman
previous editions by João Costa Seco, Luís Caires, and Bernardo Toninho

Today: Abstract Syntax and Semantics

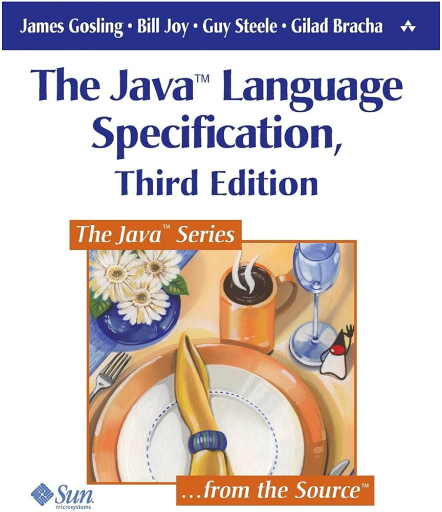
1. Abstract syntax
2. Semantics
   - Defining the formal semantics of a program
   - Focus on big step operational semantics
3. Interpreters

ITA Demo on how to write an interpreter in OCaml

How to define the meaning of programs?

Most of the time, we are satisfied with an informal description, in natural language (ISO norm, standard, reference book, etc.).

Yet it is imprecise, sometimes even ambiguous.



The Java programming language guarantees that the operands of operators appear to be evaluated in a specific evaluation order, namely, from left to right.

It is recommended that code not rely crucially on this specification.

Formal semantics gives a mathematical characterization of the computations defined by a program.

Useful to make tools (interpreters, compilers, etc.)

Necessary to reason about programs.

(Software Verification course, 1st semester)

Raises Another Question

What is a program?

As a syntactic object (sequence of characters),
it is to complex to apprehend.

That's why we switch to abstract syntax.

source

↓

lexical analysis

↓

stream of tokens

↓

parsing

↓

abstract syntax tree

↓

semantic analysis

↓

abstract syntax + symbol table

↓

code production

↓

assembly code

↓

assembler (`as`)

↓

machine language

↓

linking (`ld`)

↓

executable

Abstract Syntax (recap)

The texts

```
2*(x+1)
```

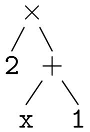
and

```
(2 * ((x) + 1))
```

and

```
2 * /* I double */ ( x + 1 )
```

all map to the same abstract syntax tree.



We define an abstract syntax using a grammar

$$
\begin{array}{rcll}
e & ::= & c & \textit{constant} \\
  & | & x & \textit{variable} \\
  & | & e+e & \textit{addition} \\
  & | & e\times e & \textit{multiplication} \\
  & | & \ldots &
\end{array}
$$

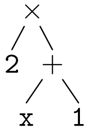
reads "an expression, noted e, is

- either a constant c,
- either a variable x,
- either the addition of two expressions,
- etc."

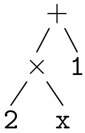
Notation $e_1 + e_2$ of the abstract syntax borrows the symbol of the concrete syntax.

But we could have picked something else, e.g.
$Add(e_1, e_2)$, $+(e_1, e_2)$, $e_1 \oplus e_2$, etc.

There is no constructor for parentheses in abstract syntax
in concrete syntax `2 * (x + 1)`,
parentheses are used to build this tree



rather than this one



(the lecture on parsing will explain how)

We call syntactic sugar a construct of concrete syntax that does not exist in abstract syntax.

It is thus translated in terms of other constructs of abstract syntax (typically during parsing).

Examples:

- in C, expression `a[i]` is syntactic sugar for `*(a+i)`
- in JAVA, expression `x -> {...}` is sugar for the construction of an object in some anonymous class that implements `Function`
- in OCAML, expression `[`e_1`; `e_2`; `...`; `e_n`]` is sugar for e_1 `::` e_2 `::` ... `::` e_n `:: []`

From Abstract Syntax to Formal Semantics

Formal semantics is defined over abstract syntax.

There are many approaches

- axiomatic semantics
- denotational semantics
- semantics by translation
- operational semantics

Axiomatic Semantics (Software Verification course)

Also called Floyd-Hoare logic
(Robert Floyd, *Assigning meanings to programs*, 1967
Tony Hoare, *An axiomatic basis for computer programming*, 1969)

Defines programs by means of their properties; we introduce a triple

$$\{P\}\ i\ \{Q\}$$

meaning "if formula P holds before the execution of statement i, then formula Q holds after the execution"

Example:

$$\{x \geq 0\}\ x := x + 1\ \{x > 0\}$$

Example of rule:

$$\{P[x \leftarrow E]\}\ x := E\ \{P(x)\}$$

Denotational semantics maps each program expression e to its denotation $[\![e]\!]$, a mathematical object that represents the computation denoted by e.

Example: arithmetic expressions with a single variable x

$$e ::= x \mid n \mid e \ \texttt{+} \ e \mid e \ \texttt{*} \ e \mid \ldots$$

The denotation is a function that maps the value of x to the value of the expression

$$\begin{aligned}
[\![x]\!] &= x \mapsto x \\
[\![n]\!] &= x \mapsto n \\
[\![e_1 \ \texttt{+} \ e_2]\!] &= x \mapsto [\![e_1]\!](x) + [\![e_2]\!](x) \\
[\![e_1 \ \texttt{*} \ e_2]\!] &= x \mapsto [\![e_1]\!](x) \times [\![e_2]\!](x)
\end{aligned}$$

Semantics by Translation

(also called Strachey semantics)

We can define the semantics of a language by means of its translation to another language for which the semantics is already defined

Example

An esoteric language whose syntax consists of 8 characters and whose semantics is defined by translation to the C language.

command	translation to C
(prelude)	`char array[30000] = 0;` `char *ptr = array;`
`>`	`++ptr;`
`<`	`--ptr;`
`+`	`++*ptr;`
`-`	`--*ptr;`
`.`	`putchar(*ptr);`
`,`	`*ptr = getchar();`
`[`	`while (*ptr) {`
`]`	`}`

Operational semantics describes the sequence of elementary computations from the expression to its outcome (its value).

It operates directly over abstract syntax.

Two kinds of operational semantics

- "natural semantics" or "big steps"

$$e \Downarrow v$$

- "reduction semantics" or "small steps"

$$e \to e_1 \to e_2 \to \cdots \to v$$

Let us illustrate big-step operational semantics on a minimal language

e	::=		**expression**
	|	v	integer or Boolean constant
	|	x	variable
	|	$e \oplus e$	binary operator ($+$, $<$, ...)
v	::=		VALUES
	|	n	integer value
	|	b	Boolean value
s	::=		**statement**
	|	x `:=` e	assignment
	|	`if` (e) s `else` s	conditional
	|	`while` (e) s	loop
	|	s `;` s	block

Example

```
a = 0;
b = 1;
while (b < 100)
  b = a+b;
  a = b-a
```

Big Steps Operational Semantics of While

We seek to define a relation between some expression e and a value v

$$e \Downarrow v$$

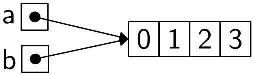
Here, values are limited to integers and Boolean constants

$$\begin{array}{rcll} v & ::= & & \textsc{Values} \\ & | & n & \text{integer value} \\ & | & b & \text{Boolean value} \end{array}$$

Caveat: with most languages, values do not coincide with constants.

In JAVA (or Python, OCAML, etc.), a value may be an address, even if we do not have addresses among the literal constants of the language

```
int[] a = new int[4];
...
int[] b = a;
b[2] = 42;
...
```

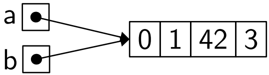


(more about this in lecture on evaluation strategies)

Example

In JAVA (or Python, OCAML, etc.), a value may be an address, even if we do not have addresses among the literal constants of the language

```
int[] a = new int[4];
...
int[] b = a;
b[2] = 42;
...
```



(more about this in lecture on evaluation strategies)

The value of a variable is given by an environment σ
(a function from variables to values).

We are going to define a relation

$$\sigma, e \Downarrow v$$

that reads "in environment σ, expression e has value v"

In environment

$$\sigma = \{a \mapsto 34,\ b \mapsto 55\}$$

the expression

$$a + b$$

has value

$$89$$

which we write

$$\sigma, a + b \Downarrow 89$$

A relation may be defined as the smallest relation satisfying a set of rules with no premises (axioms) written

$$\frac{}{P}$$

and a set of rules with premises written

$$\frac{P_1 \quad P_2 \quad \dots \quad P_n}{P}$$

This is called inference rules.

We can define the relation Even(n) with two rules

$$\frac{}{\text{Even}(0)} \qquad \text{et} \qquad \frac{\text{Even}(n)}{\text{Even}(n+2)}$$

that reads as follows

$$\begin{array}{rl} \text{on the one hand} & \text{Even}(0) \\ \text{on the other hand} & \forall n.\ \text{Even}(n) \Rightarrow \text{Even}(n+2) \end{array}$$

The smallest relation satisfying these two properties coincide with the property "n is an even natural number":

- even natural numbers are included, by induction
- if odd numbers were included, we could remove the smallest

A derivation is a tree whose internal nodes are rules with premises and whose leaves are axioms.

Example:

$$\dfrac{\dfrac{\overline{\mathsf{Even}(0)}}{\mathsf{Even}(2)}}{\mathsf{Even}(4)}$$

The set of derivations characterizes the smallest relation satisfying the inference rules.

- A constant n has value n

$$\frac{}{\sigma, n \Downarrow n}$$

- A variable x has a value if $E(x)$ is defined

$$\frac{x \text{ in } \sigma}{\sigma, x \Downarrow \sigma(x)}$$

- An addition $e_1 + e_2$ has a value if e_1 has a value n_1, if e_2 has a value n_2 and if $n_1 + n_2$ does not overflow

$$\frac{\sigma, e_1 \Downarrow n_1 \quad \sigma, e_2 \Downarrow n_2 \quad n \stackrel{\text{def}}{=} n_1 + n_2}{\sigma, e_1 + e_2 \Downarrow n}$$

- etc.

With $\sigma = \{a \mapsto 34,\ b \mapsto 39\}$, we have

$$\dfrac{\dfrac{a \in \mathrm{dom}(\sigma)}{\sigma, a \Downarrow 34} \quad \dfrac{b \in \mathrm{dom}(\sigma)}{\sigma, b \Downarrow 39} \quad 73 = 34 + 39}{\sigma, a + b \Downarrow 73}$$

Note: one can see such a tree as a proof.

There are expressions e for which there is no value v such that $\sigma, e \Downarrow v$

Examples:

- `x + 1` with a variable `x` not defined in E
- `42 + false`

These are two different situations.

- the case of an undefined variable is detected during type checking and the program is rejected.
- the case of a type mismatch, also detected during type checking.

A statement may modify the value of some variables (through assignments).

To define the semantics of a statement s, we thus introduce the relation

$$\sigma, s \Downarrow \sigma'$$

that reads "in environment σ, the evaluation of statement s terminates and leads to environment σ'"

If e has a value, then the assignment evaluates and adds/replaces variable `x`

$$\frac{\sigma, e \Downarrow v}{\sigma, x\texttt{:=}e \Downarrow \sigma\{x \mapsto v\}}$$

If the test e has a value, and if the corresponding branch evaluates, then `if` evaluates

$$\frac{\sigma, e \Downarrow \texttt{true} \quad \sigma, s_1 \Downarrow \sigma_1}{\sigma, \texttt{if}\,(e)\ s_1\ \texttt{else}\ s_2 \Downarrow \sigma_1} \qquad \frac{\sigma, e \Downarrow \texttt{false} \quad \sigma, s_2 \Downarrow \sigma_2}{\sigma, \texttt{if}\,(e)\ s_1\ \texttt{else}\ s_2 \Downarrow \sigma_2}$$

With $\sigma = \{a \mapsto 21\}$, we have

$$\dfrac{\dfrac{\dfrac{a \in \mathtt{dom}(\sigma)}{\sigma, a \Downarrow 21} \quad \sigma, 0 \Downarrow 0}{\sigma, a > 0 \Downarrow \mathtt{true}} \qquad \dfrac{\sigma, 2 \Downarrow 2 \quad \dfrac{a \in \mathtt{dom}(\sigma)}{\sigma, 2 \times a \Downarrow 21}}{\sigma, a \mathtt{:=} 2 \times a \Downarrow \{a \mapsto 42\}}}{\sigma, \mathtt{if}\ \mathtt{(}a > 0\mathtt{)}\ a \mathtt{:=} 2 \times a\ \mathtt{else}\ \mathtt{skip} \Downarrow \{a \mapsto 42\}}$$

A sequence evaluates if its statements evaluate in order

$$\frac{}{\sigma, \texttt{skip} \Downarrow \sigma} \qquad \frac{\sigma, s_1 \Downarrow \sigma_1 \quad \sigma_1, s_2 \Downarrow \sigma_2}{\sigma, s_1\,;\, s_2 \Downarrow \sigma_2}$$

A loop evaluates if it terminates

$$\frac{\sigma, e \Downarrow 0}{\sigma, \texttt{while}\,(e)\ s \Downarrow \sigma}$$

$$\frac{\sigma, e \Downarrow n \neq 0 \quad \sigma, s \Downarrow \sigma_1 \quad \sigma_1, \texttt{while}\,(e)\ s \Downarrow \sigma_2}{\sigma, \texttt{while}\,(e)\ s \Downarrow \sigma_2}$$

There are statements s that do not evaluate.
In other words, there are no σ and σ' such that $\sigma, s \Downarrow \sigma'$.

Examples:
```
while (1) skip
if (42) skip else skip
```

1. an infinite computation, for which we cannot build a finite derivation tree. In general, this is an undecidable question (the halting problem).
2. type mismatch (detected by the type-checking phase)

To establish a property of a relation defined by a set of inference rules, on can reason by structural induction on the derivation, *i.e.* one can use the induction hypothesis on any sub-derivation.

Equivalently, one can say that we perform an induction over the height of the derivation.

In practice, we proceed by induction on the derivation and by case on the last rule of the derivation.

Proposition (evaluation of expressions is deterministic)

If $\sigma, e \Downarrow v$ and $\sigma, e \Downarrow v'$ then $v = v'$.

By induction over the derivations of $\sigma, e \Downarrow v$ and $\sigma, e \Downarrow v'$.

Case of an addition $e = e_1 + e_2$

$$\frac{\begin{array}{c}(D_1)\\ \vdots\\ \sigma, e_1 \Downarrow n_1\end{array} \qquad \begin{array}{c}(D_2)\\ \vdots\\ \sigma, e_2 \Downarrow n_2\end{array}}{\sigma, e_1 + e_2 \Downarrow v} \qquad \frac{\begin{array}{c}(D_1')\\ \vdots\\ \sigma, e_1 \Downarrow n_1'\end{array} \qquad \begin{array}{c}(D_2')\\ \vdots\\ \sigma, e_2 \Downarrow n_2'\end{array}}{\sigma, e_1 + e_2 \Downarrow v'}$$

with $v = n_1 + n_2$ et $v' = n_1' + n_2'$.
By IH we have $n_1 = n_1'$ and $n_2 = n_2'$ and thus $v = v'$.

(other cases are similar or simpler)

Proposition (evaluation of statements is deterministic)

If $\sigma, s \Downarrow \sigma'$ and $\sigma, s \Downarrow \sigma''$ then $\sigma' = \sigma''$.

Exercise: do this proof.

Remark: in the case of rule

$$\frac{\sigma, e \Downarrow n \neq 0 \quad \sigma, s \Downarrow \sigma_1 \quad \sigma_1, \texttt{while}\,(e)\ s \Downarrow \sigma_2}{\sigma, \texttt{while}\,(e)\ s \Downarrow \sigma_2}$$

it is clear that induction is performed on the size of the derivation and not on the size of the statement (which does not decrease).

From Formal Semantics to Interpreters

We can implement an interpreter following the rules of the natural semantics.

Let's do it in JAVA.

As explained earlier

```
enum Binop { Add, ... }
```

```
abstract class Expr {}
class Ecte extends Expr { int n; }
class Evar extends Expr { String x; }
class Ebin extends Expr { Binop op; Expr e1, e2; }
```

```
abstract class Value {}
class Vint extends Value { int n; }
class Vbool extends Value { boolean b; }
...
```

(constructors are omitted)

Similarly for statements

```
abstract class Stmt {}
class Sassign extends Stmt { String x; Expr e; }
class Sif    extends Stmt { Expr e; Stmt s1, s2; }
class Swhile extends Stmt { Expr e; Stmt s; }
class Sblock extends Stmt { List<Stmt> l; }
```

Let's start with relation

$$\sigma, e \Downarrow v$$

The environment σ is represented by a class

```
class Environment {
  HashMap<String, Value> vars = new HashMap<>();
}
```

One solution is to declare a method

```
abstract class Expr {}
  abstract Value eval(Environment env);
}
```

and then to define it within any sub-class.

$$\frac{}{\sigma, n \Downarrow n}$$

```
class Ecte extends Expr {
  Value eval(Environment env) { return new Vint(n); }
}
```

$$\frac{x \text{ in } \mathrm{dom}(\sigma)}{\sigma, x \Downarrow \sigma(x)}$$

```
class Evar extends Expr {
  Value eval(Environment env) {
    Value v = env.vars.get(x);
    if (v == null)
      throw new Error("unbound variable " + x);
    return v;
  }
}
```

$$\frac{\sigma, e_1 \Downarrow n_1 \quad \sigma, e_2 \Downarrow n_2 \quad n = n_1 + n_2}{\sigma, e_1 + e_2 \Downarrow n} \qquad \text{etc.}$$

```
class Ebin extends Expr {
  Value eval(Environment env) {
    Value v1 = e1.eval(env), v2 = e2.eval(env);
    switch (op) {
    case Add:
      return new Vint(v1.asInt() + v2.asInt());
    ...
    }
  }
}
```

The method `eval` dynamically fails on an expression involving an undefined variable.

Use of `asInt` in class `Ebin`

Dynamic cast

We could have detected this error statically with a type-checker.

statically	=	at compile time
dynamically	=	during execution

We proceed similarly for statements by adding a method in class `Stmt`

```
abstract class Stmt {
  abstract void eval(Environment env);
}
```

That we define within any sub-class.

`eval` returns nothing, but it mutates the environment.

$$\frac{}{\sigma, \texttt{skip} \Downarrow \sigma} \qquad \frac{\sigma, s_1 \Downarrow \sigma_1 \quad \sigma_1, s_2 \Downarrow \sigma_2}{\sigma, s_1\,;\, s_2 \Downarrow \sigma_2}$$

```
class Sblock extends Stmt {
  void eval(Environment env) {
    for (Stmt s: l)
      s.eval(env);
  }
}
```

$$\frac{\sigma, e \Downarrow v}{\sigma, x := e \Downarrow \sigma\{x \mapsto v\}}$$

```
class Sassign extends Stmt {
  void eval(Environment env) {
    env.vars.put(x, e.eval(env));
  }
}
```

(the environment is a mutable data structure)

$$\frac{\sigma, e \Downarrow n \neq 0 \quad \sigma, s_1 \Downarrow \sigma_1}{\sigma, \texttt{if}\,(e)\ s_1\ \texttt{else}\ s_2 \Downarrow \sigma_1} \qquad \frac{\sigma, e \Downarrow 0 \quad \sigma, s_2 \Downarrow \sigma_2}{\sigma, \texttt{if}\,(e)\ s_1\ \texttt{else}\ s_2 \Downarrow \sigma_2}$$

```
class Sif extends Stmt {
  void eval(Environment env) {
    if (e.eval(env).asInt() != 0)
      s1.eval(env);
    else
      s2.eval(env);
  }
}
```

$$\frac{\sigma, e \Downarrow \texttt{true} \quad \sigma, s \Downarrow \sigma_1 \quad \sigma_1, \texttt{while}\,(e)\ s \Downarrow \sigma_2}{\sigma, \texttt{while}\,(e)\ s \Downarrow \sigma_2}$$

$$\frac{\sigma, e \Downarrow \texttt{false}}{\sigma, \texttt{while}\,(e)\ s \Downarrow \sigma}$$

```
class Swhile extends Stmt {
  void eval(Environment env) {
    while (e.eval(env).asInt() != 0)
      s.eval(env);
  }
}
```

We can do the same in OCaml.

Pattern matching plays the role of dynamic methods

```
let rec eval env = function
  | Ecte v ->
      v
  | Evar x ->
      (try  Hashtbl.find env x
       with Not_found -> failwith ("unbound variable" ^ x))
  | Ebin (op, e1, e2) ->
      (match op, eval env e1, eval env e2 with
      | Add, Vint n1, Vint n2 -> Vint (n1 + n2)
      | ...
      | _ -> failwith "illegal operands")
```

brief comparison functional/object programming

What distinguishes

```
type expr = Cte of value | Evar of string | ...
```

```
abstract class Expr {...} class Ecte extends Expr {...}
```

In OCaml, the code of `eval` is a single function and it covers all cases.

In Java, it is scattered in all classes.

Brief Comparison Functional/Object Programming

	horizontal extension = adding a case	vertical extension = adding a function
Java	easy (one file)	painful (several files)
OCaml	painful (several files)	easy (one file)

Another Way of Writing the JAVA Code

The JAVA code may be organized differently, with

- classes for the abstract syntax on one side,
- a class for the interpreter on the other side

To do that, we can use the visitor pattern.

Visitor (1/3)

We start by introducing an interface for the interpreter

```
interface Interpreter {
  Value interp(Ecte e);
  Value interp(Evar e);
  Value interp(Ebin e);
}
```

Note: we use Java's overloading to give all these methods the same name.

In class Expr, we provide a method accept to apply the interpreter

```
abstract class Expr {
  abstract Value accept(Interpreter i);
}
class Ecte extends Expr {
  Value accept(Interpreter i) { return i.interp(this); }
}
class Evar extends Expr {
  Value accept(Interpreter i) { return i.interp(this); }
}
class Ebin extends Expr {
  Value accept(Interpreter i) { return i.interp(this); }
}
```

This is the only intrusion in the classes of abstract syntax.

Finally, we can code the interpreter in a separate class, that implements interface `Interpreter`

```
class Interp implements Interpreter {
  Environment env = new Environment();
  Value interp(Ecte e) {
    return new Vint(e.n);
  }
  Value interp(Ebin e) {
    Value v1 = e.e1.accept(this), v2 = e.e2.accept(this);
    switch (e.op) {
    case Add:
      return new Vint(v1.asInt() + v2.asInt());
      ...
  }
  ...
}
```

It's Demo Time!

Bibliography

- Semantics with Applications: an Appetizer, Hanne Riis Nielson & Flemming Nielson, Springer-Verlag.
- Types and Programming Languages, Benjamin Pierce, The MIT Press.