École Polytechnique

CSC_52064 – **Compilation**

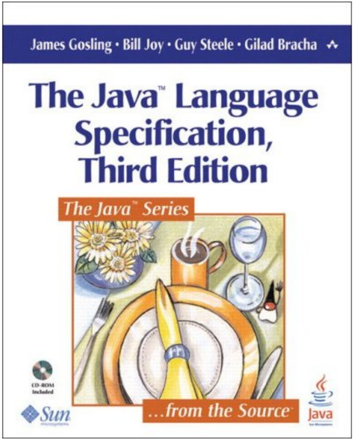
Jean-Christophe Filliâtre

abstract syntax, semantics

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.

1. abstract syntax
2. formal semantics
   - big-step operational semantics
   - interpreter
   - small-step operational semantics
3. application
   - correctness of a compiler

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

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

the texts

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

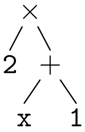
and

```
(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**

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

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. $Add(e_1, e_2)$, $+(e_1, e_2)$, etc.

we use classes to build abstract syntax trees, as follows:

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

abstract class Expr {}
class Cte extends Expr { int n; }
class Var extends Expr { String x; }
class Bin extends Expr { Binop op; Expr e1, e2; }
...
```

(constructors are omitted)

expression `2 * (x + 1)` is then represented as

```
new Bin(Mul, new Cte(2), new Bin(Add, new Var("x"), new Cte(1)))
```

we use algebraic data types to build abstract syntax trees, as follows:

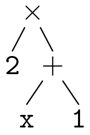
```
type binop = Add | Mul | ...

type expr =
  | Cte of int
  | Var of string
  | Bin of binop * expr * expr
  | ...
```

expression `2 * (x + 1)` is then represented as

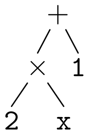
```
Bin (Mul, Cte 2, Bin (Add, Var "x", Cte 1))
```

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 $[e_1;\ e_2;\ \dots;\ e_n]$ is sugar for e_1 :: e_2 :: ... :: e_n :: []

formal semantics is defined over abstract syntax

there are many approaches

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

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

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

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

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

(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"

$$e \twoheadrightarrow v$$

- "reduction semantics" or "small steps"

$$e \rightarrow e_1 \rightarrow e_2 \rightarrow \cdots \rightarrow v$$

let us illustrate big-step operational semantics on a tiny fragment of C

e	$::=$		**expression**
	$\mid$	n	constant (signed 32-bit integer)
	$\mid$	x	variable
	$\mid$	$e\ op\ e$	binary operator $(+, <\ldots)$

s	$::=$		**statement**
	$\mid$	`x=e;`	assignment
	$\mid$	`if (e) s else s`	conditional
	$\mid$	`while (e) s`	loop
	$\mid$	`{ s ... s }`	block

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

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

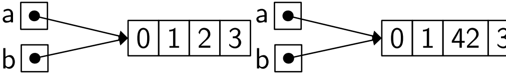
$$e \twoheadrightarrow v$$

here, values are limited to integers

v	$::=$		**value**
	$\mid$	n	integer value (signed 32-bit integer)

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 5)

the value of a variable is given by an **environment** E
(a function from variables to values)

we are going to define a relation

$$E, e \twoheadrightarrow v$$

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

in environment

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

the expression

$$a + b$$

has value

$$89$$

which we write

$$E, a + b \twoheadrightarrow 89$$

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 $\text{Even}(n)$ 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

$$\frac{}{E, n \twoheadrightarrow n}$$

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

$$\frac{x \text{ in } E}{E, x \twoheadrightarrow E(x)}$$

- 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

$$\frac{E, e_1 \twoheadrightarrow n_1 \quad E, e_2 \twoheadrightarrow n_2 \quad n \stackrel{\text{def}}{=} n_1 + n_2 \quad -2^{31} \leq n < 2^{31}}{E, e_1 + e_2 \twoheadrightarrow n}$$

- etc.

with $E = \{a \mapsto 34,\ b \mapsto 55\}$, we have

$$\dfrac{\dfrac{a \in \mathrm{dom}(E)}{E, a \twoheadrightarrow 34} \quad \dfrac{b \in \mathrm{dom}(E)}{E, b \twoheadrightarrow 55} \quad 89 = 34 + 55}{E, a + b \twoheadrightarrow 89}$$

note: one can see such a tree as a proof there are expressions e for which there is no value v such that $E, e \twoheadrightarrow v$

examples:

- `x + 1` with a variable `x` not defined in E
- `2000000000 + 1000000000` because of an overflow

these are two different situations

- the case of an undefined variable is detected during type checking (see lecture 4) and the program is rejected

- the case of a (signed) arithmetic overflow is an **undefined behavior** in the C language

  the program is accepted, compiled, and executed,
  but the compiler is free to do **whatever it wants** when an UB occurs

on the code

```
bool f(int x) {
  return x+1 < 10;
}
```

the compiler gcc produces

```
xorl    %eax, %eax
cmpl    $8, %edi
setle   %al
ret
```

which means it computes `x <= 8`

when `x` is $2^{31}-1$, the function returns `false` even if `x+1` would be -2^{31} (if it was computed) and thus `x+1 < 10` would be `true` a statement may modify the value of some variables (through assignments)

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

$$E, s \twoheadrightarrow E'$$

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

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

$$\frac{E, e \twoheadrightarrow v}{E, x\texttt{=}e\texttt{;} \twoheadrightarrow E\{x \mapsto v\}}$$

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

$$\frac{E, e \twoheadrightarrow n \neq 0 \quad E, s_1 \twoheadrightarrow E_1}{E, \texttt{if (}e\texttt{)}\ s_1\ \texttt{else}\ s_2 \twoheadrightarrow E_1} \qquad \frac{E, e \twoheadrightarrow 0 \quad E, s_2 \twoheadrightarrow E_2}{E, \texttt{if (}e\texttt{)}\ s_1\ \texttt{else}\ s_2 \twoheadrightarrow E_2}$$

with $E = \{a \mapsto 21\}$, we have

$$\dfrac{\dfrac{E, a \twoheadrightarrow 21 \quad E, 0 \twoheadrightarrow 0}{E, a > 0 \twoheadrightarrow \texttt{true}} \qquad \dfrac{\dfrac{E, 2 \twoheadrightarrow 2 \quad E, a \twoheadrightarrow 21}{E, 2 \times a \twoheadrightarrow 42}}{E, a\texttt{=}2 \times a\texttt{;} \twoheadrightarrow \{a \mapsto 42\}}}{E, \texttt{if (}a > 0\texttt{)}\ a\texttt{=}2 \times a\texttt{; else \{ \}} \twoheadrightarrow \{a \mapsto 42\}}$$

- a block evaluates if its statements evaluate in order

$$\frac{}{E, \{\ \} \twoheadrightarrow E} \qquad \frac{E, s_1 \twoheadrightarrow E_1 \quad E_1, \{\ s_2 \dots \} \twoheadrightarrow E_2}{E, \{\ s_1\ s_2 \dots \} \twoheadrightarrow E_2}$$

- a loop evaluates if it terminates

$$\frac{E, e \twoheadrightarrow 0}{E, \texttt{while}\,(e)\ s \twoheadrightarrow E}$$

$$\frac{E, e \twoheadrightarrow n \neq 0 \quad E, s \twoheadrightarrow E_1 \quad E_1, \texttt{while}\,(e)\ s \twoheadrightarrow E_2}{E, \texttt{while}\,(e)\ s \twoheadrightarrow E_2}$$

there are statements s that do not evaluate

example: `while (1) { }`

(and many other examples of statements involving expressions without a value) 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 is deterministic)

If $E, e \twoheadrightarrow v$ and $E, e \twoheadrightarrow v'$ then $v = v'$.

by induction over the derivations of $E, e \twoheadrightarrow v$ and $E, e \twoheadrightarrow v'$
case of an addition $e = e_1 + e_2$

$$
\frac{\begin{array}{c}(D_1)\\ \vdots\\ E, e_1 \twoheadrightarrow n_1\end{array} \qquad \begin{array}{c}(D_2)\\ \vdots\\ E, e_2 \twoheadrightarrow n_2\end{array}}{E, e_1 + e_2 \twoheadrightarrow v}
\qquad
\frac{\begin{array}{c}(D'_1)\\ \vdots\\ E, e_1 \twoheadrightarrow n'_1\end{array} \qquad \begin{array}{c}(D'_2)\\ \vdots\\ E, e_2 \twoheadrightarrow n'_2\end{array}}{E, e_1 + e_2 \twoheadrightarrow v'}
$$

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 is deterministic)

If $E, s \twoheadrightarrow E'$ and $E, s \twoheadrightarrow E''$ then $E' = E''$.

exercise: do this proof

remark: in the case of rule

$$\frac{E, e \twoheadrightarrow n \neq 0 \quad E, s \twoheadrightarrow E_1 \quad E_1, \texttt{while}\,(e)\ s \twoheadrightarrow E_2}{E, \texttt{while}\,(e)\ s \twoheadrightarrow E_2}$$

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) an evaluation relation is not necessarily deterministic

example: we add a primitive *random* to draw an integer 0 or 1 at random, with the rule

$$\frac{0 \leq n < 2}{E, \texttt{random()} \twoheadrightarrow n}$$

then we have $E, \texttt{random()} \twoheadrightarrow 0$ as well as $E, \texttt{random()} \twoheadrightarrow 1$

we can code 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; }
...
```

(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

$$E, e \twoheadrightarrow v$$

the environment E 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

$$\frac{}{E, n \twoheadrightarrow n}$$

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

$$\frac{x \text{ in } E}{E, x \twoheadrightarrow E(x)}$$

```
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;
  }
}
```

$$\frac{E, e_1 \twoheadrightarrow n_1 \quad E, e_2 \twoheadrightarrow n_2 \quad n = n_1 + n_2 \quad -2^{31} \leq n < 2^{31}}{E, e_1 + e_2 \twoheadrightarrow n} \qquad \text{etc.}$$

```
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());
    ...
    }
  }
}
```

(we could check the absence of arithmetic overflow) the method `eval` **dynamically** fails on an expression involving an undefined variable

we could have detected this error **statically** with typing (see lecture 4)

$$\begin{array}{rcl} \text{statically} & = & \text{at compile time} \\ \text{dynamically} & = & \text{during execution} \end{array}$$

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

$$\frac{}{E, \{\ \} \twoheadrightarrow E} \qquad \frac{E, s_1 \twoheadrightarrow E_1 \quad E_1, \{\ s_2 \dots \} \twoheadrightarrow E_2}{E, \{\ s_1\ s_2 \dots \} \twoheadrightarrow E_2}$$

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

$$\frac{E, e \twoheadrightarrow v}{E, x\texttt{=}e\texttt{;} \twoheadrightarrow E\{x \mapsto v\}}$$

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

(the environment is a mutable data structure)

$$\frac{E, e \twoheadrightarrow n \neq 0 \quad E, s_1 \twoheadrightarrow E_1}{E, \texttt{if (}e\texttt{)}\ s_1\ \texttt{else}\ s_2 \twoheadrightarrow E_1} \qquad \frac{E, e \twoheadrightarrow 0 \quad E, s_2 \twoheadrightarrow E_2}{E, \texttt{if (}e\texttt{)}\ s_1\ \texttt{else}\ s_2 \twoheadrightarrow E_2}$$

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

$$\frac{E, e \twoheadrightarrow n \neq 0 \quad E, s \twoheadrightarrow E_1 \quad E_1, \texttt{while}\,(e)\ s \twoheadrightarrow E_2}{E, \texttt{while}\,(e)\ s \twoheadrightarrow E_2}$$

$$\frac{E, e \twoheadrightarrow 0}{E, \texttt{while}\,(e)\ s \twoheadrightarrow E}$$

```
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")
```

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** 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());
      ...
  }
  ...
}
```

the interface `Interpreter` is specific to our needs

we could instead provide a general-purpose visitor

```
interface Visitor {
  void visit(Ecte e);
  void visit(Evar e);
  void visit(Ebin e);
}
```

so that we can use it for other purposes, e.g. printing

methods `visit` return nothing, but this is not an issue
(see this week's lab)

natural semantics makes no distinction between programs that crash, such as

$$x + 1$$

with an undefined variable x and programs whose evaluation does not terminate, such as

```
while (1) { }
```

small-step operational semantics remedies this by introducing a notion of elementary computation $E_1, s_1 \rightarrow E_2, s_2$, which we iterate

then we can distinguish

1. successful termination

$$E, s \rightarrow E_1, s_1 \rightarrow E_2, s_2 \rightarrow \cdots \rightarrow E', \{\ \}$$

2. evaluation stuck on E_n, s_n with $s_n \neq \{\ \}$

$$E, s \rightarrow E_1, s_1 \rightarrow E_2, s_2 \rightarrow \cdots \rightarrow E_n, s_n$$

3. non-terminating evaluation

$$E, s \rightarrow E_1, s_1 \rightarrow E_2, s_2 \rightarrow \cdots$$

we can keep our big-step semantics for expressions,
since expressions always terminate

but for a more complex language (say, with function calls in expressions),
we would use small-step semantics for expressions as well

small-step operational semantics for WHILE

$$\frac{E, e \twoheadrightarrow v}{E, x\texttt{=}e\texttt{;} \rightarrow E\{x \mapsto v\}, \{\ \}}$$

$$\frac{}{E, \{\ \{\ \}\ s \ldots\ \} \rightarrow E, \{\ s \ldots\ \}} \qquad \frac{E, s_1 \rightarrow E_1, s_1'}{E, \{\ s_1\ s_2 \ldots\ \} \rightarrow E_1, \{\ s_1'\ s_2 \ldots\ \}}$$

$$\frac{E, e \twoheadrightarrow n \neq 0}{E, \texttt{if}\ \texttt{(}e\texttt{)}\ s_1\ \texttt{else}\ s_2 \rightarrow E, s_1} \qquad \frac{E, e \twoheadrightarrow 0}{E, \texttt{if}\ \texttt{(}e\texttt{)}\ s_1\ \texttt{else}\ s_2 \rightarrow E, s_2}$$

$$\frac{E, e \twoheadrightarrow n \neq 0}{E, \texttt{while}\ \texttt{(}e\texttt{)}\ s \rightarrow E, \{\ s\ \texttt{while}\ \texttt{(}e\texttt{)}\ s\ \}}$$

$$\frac{E, e \twoheadrightarrow 0}{E, \texttt{while}\ \texttt{(}e\texttt{)}\ s \rightarrow E, \{\ \}}$$

$$
\begin{array}{rl}
& \{\}, \{\, x\texttt{=}40\texttt{;}\ \texttt{while}\,(x < 42)\ x\texttt{=}x+1\texttt{;}\,\} \\
\to & \{x \mapsto 40\}, \{\,\{\,\}\ \texttt{while}\,(x < 42)\ x\texttt{=}x+1\texttt{;}\,\} \\
\to & \{x \mapsto 40\}, \texttt{while}\,(x < 42)\ x\texttt{=}x+1\texttt{;} \\
\to & \{x \mapsto 40\}, \{\, x\texttt{=}x+1\texttt{;}\ \texttt{while}\,(x < 42)\ x\texttt{=}x+1\texttt{;}\,\} \\
\to & \{x \mapsto 41\}, \{\,\{\,\}\ \texttt{while}\,(x < 42)\ x\texttt{=}x+1\texttt{;}\,\} \\
\to & \{x \mapsto 41\}, \texttt{while}\,(x < 42)\ x\texttt{=}x+1\texttt{;} \\
\to & \{x \mapsto 41\}, \{\, x\texttt{=}x+1\texttt{;}\ \texttt{while}\,(x < 42)\ x\texttt{=}x+1\texttt{;}\,\} \\
\to & \{x \mapsto 42\}, \{\,\{\,\}\ \texttt{while}\,(x < 42)\ x\texttt{=}x+1\texttt{;}\,\} \\
\to & \{x \mapsto 42\}, \texttt{while}\,(x < 42)\ x\texttt{=}x+1\texttt{;} \\
\to & \{x \mapsto 42\}, \{\,\}
\end{array}
$$

we could replace the two rules for `while` with the following rule

$$\frac{}{E, \texttt{while}\,(e)\ s \rightarrow E, \texttt{if}\,(e)\ \{\, s\ \texttt{while}\,(e)\ s\,\}\ \texttt{else}\ \{\ \}}$$

Proposition (equivalence of the two semantics)

The two operational semantics are equivalent on programs whose evaluation terminate, i.e.

$$E, s \twoheadrightarrow E' \quad \textit{if and only if} \quad E, s \rightarrow^\star E', \{\ \}$$

(where $\rightarrow^\star$ is the reflexive transitive closure of $\rightarrow$).

Proposition (big steps imply small steps)

If $E, s \twoheadrightarrow E'$, then $E, s \rightarrow^\star E', \{\ \}$.

by induction on the derivation $E, s \twoheadrightarrow E'$ and by case on the last rule

- case of $\{\ s_1\ \ s_2 \ldots\ \}$

$$\frac{E, s_1 \twoheadrightarrow E_1 \quad E_1, \{\ s_2 \ldots\ \} \twoheadrightarrow E_2}{E, \{\ s_1\ \ s_2 \ldots\ \} \twoheadrightarrow E_2}$$

then $E, s_1 \rightarrow^\star E_1, \{\ \}$ by IH
consequently,

$$\begin{array}{rcll} E, \{\ s_1\ \ s_2 \ldots\ \} & \rightarrow^\star & E_1, \{\ \{\ \}\ s_2 \ldots\ \} & \text{(small steps)} \\ & \rightarrow & E_1, \{\ s_2 \ldots\ \} & \\ & \rightarrow^\star & E_2, \{\ \} & \text{(IH)} \end{array}$$

- case of $\texttt{while}\,(e)\ s$

if

$$\frac{E, e \twoheadrightarrow n \neq 0 \quad E, s \twoheadrightarrow E_1 \quad E_1, \texttt{while}\,(e)\ s \twoheadrightarrow E_2}{E, \texttt{while}\,(e)\ s \twoheadrightarrow E_2}$$

then

$$\begin{array}{rcll}
E, \texttt{while}\,(e)\ s & \rightarrow & E, \{\, s\ \texttt{while}\,(e)\ s \,\} & \\
& \rightarrow^\star & E_1, \{\, \{\,\}\ \texttt{while}\,(e)\ s \,\} & \text{(IH + block rule)} \\
& \rightarrow & E_1, \{\, \texttt{while}\,(e)\ s \,\} & \\
& \rightarrow^\star & E_2, \{\,\} & \text{(IH)}
\end{array}$$

exercise: do the other cases

Lemma

If $E_1, s_1 \to E_2, s_2 \twoheadrightarrow E'$, then $E_1, s_1 \twoheadrightarrow E'$.

by induction over $\twoheadrightarrow$

- case $s_1 = \{\, u_1\ \ v_1 \ldots \,\}$
  - case $u_1 = \{\,\}$
    we have $E_1, \{\, \{\,\}\ v_1 \ldots \,\} \to E_1, \{\, v_1 \ldots \,\} \twoheadrightarrow E'$ and thus
    $$\frac{E_1, \{\,\} \twoheadrightarrow E_1 \quad E_1, \{\, v_1 \ldots \,\} \twoheadrightarrow E'}{E_1, \{\, \{\,\}; v_1 \ldots \,\} \twoheadrightarrow E'}$$
  - case $u_1 \neq \{\,\}$
    $E_1, \{\, u_1\ \ v_1 \ldots \,\} \to E_2, \{\, u_2\ \ v_1 \ldots \,\} \twoheadrightarrow E'$ *i.e.* $E_1, u_1 \to E_2, u_2$ and
    $$\frac{E_2, u_2 \twoheadrightarrow E_2' \quad E_2', \{\, v_1 \ldots \,\} \twoheadrightarrow E'}{E_2, \{\, u_2\ \ v_1 \ldots \,\} \twoheadrightarrow E'}$$
    by IH we deduce
    $$\frac{E_1, u_1 \twoheadrightarrow E_2' \quad E_2', \{\, v_1 \ldots \,\} \twoheadrightarrow E'}{E_1, \{\, u_1\ \ v_1 \ldots \,\} \twoheadrightarrow E'}$$

(do the other cases)

we deduce

Proposition (small steps imply big steps)

Si $E, s \rightarrow^\star E', \{\ \}$, alors $E, s \twoheadrightarrow E'$.

proof: we have

$$E, s \rightarrow E_1, s_1 \rightarrow E_2, s_2 \rightarrow \cdots \rightarrow E_n, s_n \rightarrow E', \{\ \}$$

but $E', \{\ \} \twoheadrightarrow E'$ so $E_n, s_n \twoheadrightarrow E'$ by the lemma above,
then $E_{n-1}, s_{n-1} \twoheadrightarrow E'$ by the same lemma, etc.,
until we get $E_, s \twoheadrightarrow E'$
(induction on n, the number of steps)

let us add pointers to our fragment of C this means extending the notion of value, of expressions, and of statements

$$
\begin{array}{rcll}
v & ::= & & \textbf{value} \\
 & | & n & \text{integer value (signed 32-bit integer)} \\
 & | & \ell & \text{memory address} \\
 \\
e & ::= & \dots & \textbf{expression} \\
 & | & \texttt{malloc(4)} & \text{allocate memory} \\
 & | & \texttt{*}e & \text{read from memory} \\
 \\
s & ::= & \dots & \textbf{statement} \\
 & | & \texttt{*}e\texttt{=}e\texttt{;} & \text{write to memory mémoire}
\end{array}
$$

to include the memory in our semantics, we extend the semantics relation as follows,

$$M, E, e \twoheadrightarrow v$$

where M is a function from addresses (ℓ) to integers (n)

we add a rule to allocate memory

$$\frac{\ell \text{ an address that is not in } M}{M, E, \texttt{malloc(4)} \twoheadrightarrow \ell}$$

and another to read from memory

$$\frac{M, E, e \twoheadrightarrow \ell \quad \ell \text{ in } M}{M, E, \texttt{*}e \twoheadrightarrow M(\ell)}$$

we add a rule to write to memory

$$\frac{M, E, e_1 \twoheadrightarrow \ell \quad \ell \text{ in } M \quad M, E, e_2 \twoheadrightarrow n}{M, E, \texttt{*}e_1\texttt{=}e_2\texttt{;} \twoheadrightarrow M\{\ell \mapsto n\}, E}$$

as it is defined, our semantics makes a clear distinction between integers and addresses

in particular, an expression such as `**e` has no value

we could accommodate such expressions, but at the price of a much more complex semantics let us add **functions** to our fragment of C

to make things simpler, functions do not return any value;
but they are useful anyway:

```
void f(int x, int *p) {
  while (x) {
    *p = *p + x;
    x = x - 1;
  }
}
```

beside, we only have local variables

$$
\begin{array}{rcll}
s & ::= & \dots & \textbf{statement} \\
 & | & f\,(e, \dots, e)\,; & \text{function call} \\
 & & & \\
p & ::= & d \dots d & \textbf{program} \\
 & & & \\
d & ::= & \texttt{void}\ f\,(x, \dots, x)\ s & \textbf{function definition}
\end{array}
$$

for a function call to evaluate, we require that

- the function exists, with the right number of parameters
- all arguments evaluate to values
- the body of the function evaluates

$$\frac{\text{there exists a function } \texttt{void}\ f\,(x_1,\ldots,x_n)\ s \qquad M, E, e_i \twoheadrightarrow v_i \quad M, \{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\}, s \twoheadrightarrow M', E'}{M, E, f\,(e_1,\ldots,e_n)\,; \twoheadrightarrow M', E}$$

$$\frac{\text{there exists a function } \texttt{void } f\,\texttt{(}x_1,\ldots,x_n\texttt{)}\ s \qquad M, E, e_i \twoheadrightarrow v_i \quad M, \{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\}, s \twoheadrightarrow M', E'}{M, E, f\,\texttt{(}e_1, \ldots, e_n\texttt{)}\,\texttt{;} \twoheadrightarrow M', E}$$

note how

- the function body s is evaluated in a **new environment**, with only variables x_i and whose final value E' is discarded
- we return to the environment E from which we started, unchanged
- the memory, on the contrary, is possibly modified

here, the **evaluation order** of the function arguments is not significant, since the evaluation of an expression has no side effects

in the (real) C language, however, arguments may have effects

```
f(i++, j++)
```

and it is **not specified** in which order they are evaluated
(this is an *implementation-defined* behavior)

we have evaluated function arguments **before** the call,
and we passed their **values**

we call this **call by value**

this is what C does, but this is not the only option (see lecture 5) the lecture notes also contain the operational semantics for Mini-ML

$$
\begin{array}{rcll}
e & ::= & x & \text{identifier} \\
 & | & c & \text{constant } (1,\ 2,\ \ldots,\ \mathit{true},\ \ldots) \\
 & | & op & \text{primitive } (+,\ \times,\ \mathit{fst},\ \ldots) \\
 & | & \mathtt{fun}\ x \rightarrow e & \text{anonymous function} \\
 & | & e\ e & \text{application} \\
 & | & (e,\ e) & \text{pair} \\
 & | & \mathtt{let}\ x = e\ \mathtt{in}\ e & \text{local binding}
\end{array}
$$

(section 2.2, page 20)

application

---

correctness of a compiler

a compiler must respect the semantics

if the input language is equipped with a semantics $\rightarrow_s$ and the target language with a semantics $\rightarrow_m$, and if some expression e is compiled to $C(e)$ the we must have "a commutative diagram":

$$\begin{array}{ccc} e & \xrightarrow{\star}_s & v \\ \downarrow & & \approx \\ C(e) & \xrightarrow{\star}_m & v' \end{array}$$

where $v \approx v'$ states that values v and v' coincide let us consider arithmetic expressions with no variables

$$e ::= n \mid e + e$$

and let us show the correctness of a very simple compiler to x86-64 that uses the stack to store intermediate computations we set a small-step semantics for the input language

$$\frac{n = n_1 + n_2}{n_1 + n_2 \rightarrow n} \qquad \frac{e_1 \rightarrow e_1'}{e_1 + e_2 \rightarrow e_1' + e_2} \qquad \frac{e_2 \rightarrow e_2'}{n_1 + e_2 \rightarrow n_1 + e_2'}$$

similarly, we set a small-step semantics for the target language

$$\begin{array}{rcl} m & ::= & \texttt{movq}\ \$n, r \\ & | & \texttt{addq}\ \$n, r \mid \texttt{addq}\ r, r \\ & | & \texttt{movq}\ (r), r \mid \texttt{movq}\ r, (r) \mid \\ r & ::= & \texttt{\%rdi} \mid \texttt{\%rsi} \mid \texttt{\%rsp} \end{array}$$

a state gathers the values of registers, R,
and the contents of the memory, M

$$\begin{array}{rcl} R & ::= & \{\texttt{\%rdi} \mapsto n; \texttt{\%rsi} \mapsto n; \texttt{\%rsp} \mapsto n\} \\ M & ::= & \mathbb{N} \to \mathbb{Z} \end{array}$$

we then define the semantics of an instruction m using a relation

$$R, M, m \xrightarrow{m} R', M'$$

the relation $R, M, m \xrightarrow{m} R', M'$ is defined as follows:

$$\begin{array}{rcl}
R, M, \texttt{movq}\ \$n, r & \xrightarrow{m} & R\{r \mapsto n\}, M \\
R, M, \texttt{addq}\ \$n, r & \xrightarrow{m} & R\{r \mapsto R(r) + n\}, M \\
R, M, \texttt{addq}\ r_1, r_2 & \xrightarrow{m} & R\{r_2 \mapsto R(r_1) + R(r_2)\}, M \\
R, M, \texttt{movq}\ (r_1), r_2 & \xrightarrow{m} & R\{r_2 \mapsto M(R(r_1))\}, M \\
R, M, \texttt{movq}\ r_1, (r_2) & \xrightarrow{m} & R, M\{R(r_2) \mapsto R(r_1)\}
\end{array}$$

the final value of an expression is stored in `%rdi`

$$
\begin{array}{rcl}
\mathit{code}(n) & = & \texttt{movq}\ \$n, \texttt{\%rdi} \\[1em]
\mathit{code}(e_1 + e_2) & = & \mathit{code}(e_1) \\
& & \texttt{addq}\ \ \$\texttt{-8}, \texttt{\%rsp} \\
& & \texttt{movq}\ \ \texttt{\%rdi}, (\texttt{\%rsp}) \\
& & \mathit{code}(e_2) \\
& & \texttt{movq}\ \ (\texttt{\%rsp}), \texttt{\%rsi} \\
& & \texttt{addq}\ \ \$8, \texttt{\%rsp} \\
& & \texttt{addq}\ \ \texttt{\%rsi}, \texttt{\%rdi}
\end{array}
$$

we seek to prove that if

$$e \xrightarrow{\star} n$$

and if

$$R, M, code(e) \xrightarrow{m}{}^{\star} R', M'$$

then $R'(\texttt{\%rdi}) = n$

we proceed by structural induction on e

we show a stronger property (an **invariant**), namely

if $e \xrightarrow{\star} n$ and $R, M, code(e) \xrightarrow{m}^{\star} R', M'$ then

$$\begin{cases} R'(\texttt{\%rdi}) = n \\ R'(\texttt{\%rsp}) = R(\texttt{\%rsp}) \\ \forall a \geq R(\texttt{\%rsp}),\ M'(a) = M(a) \end{cases}$$

- case $e = n$

  we have $e \xrightarrow{\star} n$ and $code(e) = \texttt{movq } \$n, \%\texttt{rdi}$ and the result is immediate

- case $e = e_1 + e_2$

  we have $e \xrightarrow{\star} n_1 + e_2 \xrightarrow{\star} n_1 + n_2$ with $e_1 \xrightarrow{\star} n_1$ and $e_2 \xrightarrow{\star} n_2$
  thus we can invoke the induction hypothesis on e_1 and e_2

	R, M	
$code(e_1)$	R_1, M_1	by induction hypothesis $R_1(\%\mathtt{rdi}) = n_1$ and $R_1(\%\mathtt{rsp}) = R(\%\mathtt{rsp})$ $\forall a \geq R(\%\mathtt{rsp}),\ \ M_1(a) = M(a)$
`addq $-8, %rsp` `movq %rdi, (%rsp)`	R'_1, M'_1	$R'_1 = R_1\{\%\mathtt{rsp} \mapsto R(\%\mathtt{rsp}) - 8\}$ $M'_1 = M_1\{R(\%\mathtt{rsp}) - 8 \mapsto n_1\}$
$code(e_2)$	R_2, M_2	by induction hypothesis $R_2(\%\mathtt{rdi}) = n_2$ and $R_2(\%\mathtt{rsp}) = R(\%\mathtt{rsp}) - 8$ $\forall a \geq R(\%\mathtt{rsp}) - 8,\ \ M_2(a) = M'_1(a)$

`movq (%rsp), %rsi`
`addq $8, %rsp`
`addq %rsi, %rdi`
	R', M_2	$R'(\%\mathtt{rdi}) = n_1 + n_2$
		$R'(\%\mathtt{rsp}) = R(\%\mathtt{rsp}) - 8 + 8 = R(\%\mathtt{rsp})$
		$\forall a \geq R(\%\mathtt{rsp}),$
		$\quad M_2(a) = M'_1(a) = M_1(a) = M(a)$

such a proof can be done for a realistic compiler

example: CompCert, an optimizing compiler from C to PowerPC, ARM, RISC-V, and x86, has been formally verified using the Coq proof assistant

see `http://compcert.inria.fr/`

- lab 2 (rooms 31 and 32)
    - a mini-Python interpreter
    - in Java or OCaml (your choice)
    - take your time to **read and understand** the code that is provided
- lecture 3
    - parsing

```
> ./mini-python tests/good/pascal.py
*
**
***
****
*****
******
*******
*000000*
**00000**
***0000***
****000****
*****00*****
******0******
*************
*000000*000000*
**00000**00000**
***0000***0000***
****000****000****
*****00*****00*****
******0******0******
********************
```