École Polytechnique

INF564 – 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.)

unsatisfactory, since it is imprecise, sometimes even ambiguous
informal semantics

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.
raises another question

what is a program?

as a syntactic object (sequence of characters),
it is too 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
the texts

\[ 2 \times (x+1) \]

and

\[ (2 \times ((x) + 1)) \]

and

\[ 2 \times /* I \text{ double } */ (x + 1) \]

all map to the same abstract syntax tree

\[
\begin{array}{c}
\times \\
/ \ \ \ \ \ \ / \\
2 \ \ \ + \\
/ \ \ \ \ / \\
x \ \ \ 1
\end{array}
\]
we define an abstract syntax using a grammar

\[
\begin{align*}
e & ::= c \quad \text{constant} \\
   & | x \quad \text{variable} \\
   & | e + e \quad \text{addition} \\
   & | e \times e \quad \text{multiplication} \\
   & | ...
\end{align*}
\]

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. $\text{Add}(e_1, e_2), + (e_1, e_2)$, etc.
we use classes to build abstract syntax trees, as follows:

```java
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

```java
new Bin(Mul, new Cte(2), new Bin(Add, new Var("x"), new Cte(1)))
```
we use algebraic data types to abstract syntax trees, as follows:

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

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

expression \(2 \ast (x + 1)\) 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 $2 \times (x + 1)$, parentheses are used to build this tree

$$
\times
\bigg/ \bigg/ \\
2 \quad + \\
\quad \bigg/ \bigg/ \\
x \quad 1
$$

rather than this one

$$
+ \\
\bigg/ \bigg/ \\
\times \quad 1 \\
\quad \bigg/ \bigg/ \\
2 \quad x
$$

(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 `\([e_1; e_2; \ldots; e_n]\)` is sugar for `\(e_1 :: e_2 :: \ldots :: e_n :: []\)`
formal semantics is defined over abstract syntax

there are many approaches

- axiomatic semantics
- denotational semantics
- semantics by translation
- operational semantics
also called **Hoare logic**

*(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 $\llbracket e \rrbracket$, a mathematical object that represents the computation denoted by $e$

example: arithmetic expressions with a single variable $x$

$$e ::= x \mid n \mid e + e \mid e \ast e \mid \ldots$$

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

$$\begin{align*}
\llbracket x \rrbracket &= x \mapsto x \\
\llbracket n \rrbracket &= x \mapsto n \\
\llbracket e_1 + e_2 \rrbracket &= x \mapsto \llbracket e_1 \rrbracket(x) + \llbracket e_2 \rrbracket(x) \\
\llbracket e_1 \ast e_2 \rrbracket &= x \mapsto \llbracket e_1 \rrbracket(x) \times \llbracket e_2 \rrbracket(x)
\end{align*}$$
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
an esoteric language whose syntax consists of 8 characters and whose semantics is defined by translation to the C language

<table>
<thead>
<tr>
<th>command</th>
<th>translation to C</th>
</tr>
</thead>
</table>
| (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”

\[ e \rightarrow v \]

• “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 minimal language

\[ e ::= \text{expression} \]

\[ \mid c \quad \text{constant} \]

\[ \mid x \quad \text{variable} \]

\[ \mid e \; \text{op} \; e \quad \text{binary operator} \; (+, <\ldots) \]

\[ c ::= \text{constant} \]

\[ \mid n \quad \text{integer constant} \; (-17, 42\ldots) \]

\[ \mid b \quad \text{Boolean constant} \; (\text{true, false}) \]
\[ s ::= \text{statement} \]

\[ \quad \mid x \leftarrow e \quad \text{assignment} \]

\[ \quad \mid \text{if } e \text{ then } s \text{ else } s \quad \text{conditional} \]

\[ \quad \mid \text{while } e \text{ do } s \quad \text{loop} \]

\[ \quad \mid s; s \quad \text{sequence} \]

\[ \quad \mid \text{skip} \quad \text{do nothing} \]
\[
a \leftarrow 0;
b \leftarrow 1;
\text{while } b < 1000 \text{ do }
b \leftarrow a + b;
a \leftarrow b - a
\]
we seek to define a relation between some expression $e$ and a value $v$

\[ e \rightarrow v \]

due to the fact that values are limited to integer and Boolean constants

\[
\begin{align*}
v & ::= \text{value} \\
   & | \ n \quad \text{integer value} \\
   & | \ b \quad \text{Boolean value}
\end{align*}
\]

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

```java
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

\[ E, e \rightarrow v \]

that reads “in environment \( E \), expression \( e \) has value \( v \)”
in environment

\[ E = \{ a \mapsto 34, \ b \mapsto 55 \} \]

expression

\[ a + b \]

has value

\[ 89 \]

which we write

\[ E, a + b \rightarrow 89 \]
a relation may be defined as the **smallest relation** satisfying a set of rules with no premises (axioms) written

\[
\begin{array}{c}
\hline
P \\
\end{array}
\]

and a set of rules with premises written

\[
\begin{array}{c}
P_1 & P_2 & \ldots & P_n \\
\hline
P \\
\end{array}
\]

this is called **inference rules**
we can define the relation $\text{Even}(n)$ with two rules

\[
\begin{align*}
\text{Even}(0) & \quad \text{et} \quad \text{Even}(n) \\
\text{Even}(n) & \Rightarrow \text{Even}(n + 2)
\end{align*}
\]

that reads as follows

- on the one hand $\text{Even}(0)$
- on the other hand $\forall n. \text{Even}(n) \Rightarrow \text{Even}(n + 2)$

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 nodes are rules with premises and leaves are axioms

example:

```
          Even(0)
         /    |
       Even(2)
      /   |
    Even(4)
```

the set of derivations characterizes the smallest relation satisfying the inference rules
the relation $E, e \rightarrow v$ is defined by the following inference rules:

\[
\begin{align*}
E, n &\rightarrow n & E, b &\rightarrow b \\
E, x &\rightarrow E(x) \\
E, e_1 &\rightarrow n_1 & E, e_2 &\rightarrow n_2 & n = n_1 + n_2 \\
E, e_1 + e_2 &\rightarrow n
\end{align*}
\]

etc.
with \( E = \{ a \mapsto 34, \ b \mapsto 55 \} \), we have

\[
\begin{align*}
E, a & \rightarrow 34 \\
E, b & \rightarrow 55 \\
E, a + b & \rightarrow 89
\end{align*}
\]

\[89 = 34 + 55\]

note: one can see such a tree as a proof
a statement may modify the value of some variables (through assignments)

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

$$E, s \rightarrow E'$$

that reads “in environment $E$, the evaluation of statement $s$ terminates and leads to environment $E'$”
semantics of statements

\[
\begin{align*}
E, \text{skip} & \rightarrow E & E, s_1 & \rightarrow E_1 & E_1, s_2 & \rightarrow E_2 \\
E, s_1; s_2 & \rightarrow E_2 \\
E, e & \rightarrow v \\
E, x \leftarrow e & \rightarrow E\{x \leftarrow v\} \\
E, e & \rightarrow \text{true} & E, s_1 & \rightarrow E_1 \\
E, \text{if } e \text{ then } s_1 \text{ else } s_2 & \rightarrow E_1 \\
E, e & \rightarrow \text{false} & E, s_2 & \rightarrow E_2 \\
E, \text{while } e \text{ do } s & \rightarrow E_2 \\
E, e & \rightarrow \text{false} \\
E, \text{while } e \text{ do } s & \rightarrow E
\end{align*}
\]
with $E = \{a \mapsto 21\}$, we have

\[
\begin{array}{c}
E, a \mapsto 21 \quad E, 0 \mapsto 0 \\
\hline
E, a > 0 \mapsto \text{true} \\
\hline
E, \text{if } a > 0 \text{ then } a \leftarrow 2 \times a \text{ else skip} \mapsto \{a \mapsto 42\}
\end{array}
\quad \begin{array}{c}
E, 2 \mapsto 2 \quad E, a \mapsto 21 \\
\hline
E, 2 \times a \mapsto 42 \\
\hline
E, a \leftarrow 2 \times a \mapsto \{a \mapsto 42\}
\end{array}
\]
expressions without value

there are expressions $e$ for which there is no value $v$ such that $E, e \rightarrow v$

example: $1 + \text{true}$

similarly, there are statements $s$ for which there is no evaluation $E, s \rightarrow E'$

example: while true do skip
Induction on the derivation

to establish a property of a relation defined by a set of inference rules, one 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
**Proposition (evaluation is deterministic)**

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

by induction over the derivations of $E, e \rightarrow v$ and $E, e \rightarrow v'$

case of an addition $e = e_1 + e_2$

$\begin{align*}
(D_1) & \quad (D_2) \\
E, e_1 \rightarrow n_1 & \quad E, e_2 \rightarrow n_2 \\
\vdots & \quad \vdots \\
E, e_1 + e_2 \rightarrow v & \quad \quad \\
\end{align*}$

$\begin{align*}
(D_1') & \quad (D_2') \\
E, e_1 \rightarrow n'_1 & \quad E, e_2 \rightarrow n'_2 \\
\vdots & \quad \vdots \\
E, e_1 + e_2 \rightarrow v' & \quad \quad \\
\end{align*}$

with $v = v_1 + n_2$ 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 $E, s \rightarrow E'$ and $E, s \rightarrow E''$ then $E' = E''$.

exercise: do this proof

remark: in the case of rule

\[
E, e \rightarrow \text{true} \quad E, s \rightarrow E_1 \quad E_1, \text{while } e \text{ do } s \rightarrow E_2
\]

\[
E, \text{while } e \text{ do } s \rightarrow E_2
\]

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 \textit{random} to draw an integer 0 or 1 at random, with the rule

\[
\frac{0 \leq n < 2}{E, \text{random} \rightarrow n}
\]

then we have \(E, \text{random} \rightarrow 0\) as well as \(E, \text{random} \rightarrow 1\)
we can code an interpreter following the rules of the natural semantics

let’s do it in Java
as explained earlier

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

abstract class Expr {}
class Ecte extends Expr { Value v; }
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

```java
abstract class Stmt {}

class Sskip extends 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 Sseq extends Stmt { Stmt s1, s2; }
```
let's start with relation

\[ E, e \rightarrow v \]

the environment \( E \) is represented by a class

```java
class Environment {
    HashMap<String, Value> vars = new HashMap<>();
}
```
evaluation of an expression

one solution is to declare a method

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

and then to define it within any sub-class
evaluation of an expression

\[
E, n \rightarrow n \quad E, b \rightarrow b
\]

class Ecte extends Expr {
    Value eval(Environment env) {
        return v;
    }
}

\[
E, x \rightarrow E(x)
\]

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;
    }
}
evaluation of an expression

\[
\begin{align*}
E, e_1 & \rightarrow n_1 \\
E, e_2 & \rightarrow n_2 \\
n & = n_1 + n_2 \\
E, e_1 + e_2 & \rightarrow n
\end{align*}
\]

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());
            ...
        }
    }
}
method `asInt` (in class `Value`) checks that the values in an integer and returns it otherwise, `asInt` raises an exception consequently, our interpreter fails *dynamically* on an expression such as `1+true`

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

\[
\begin{align*}
\text{statically} & \quad = \quad \text{at compile time} \\
\text{dynamically} & \quad = \quad \text{during execution}
\end{align*}
\]
we proceed similarly for statements by adding a method in class Stmt

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

that we define within any sub-class

eval returns nothing, the environment being mutated
evaluation of a statement

\[ E, \text{skip} \rightarrow E \]

```java
class Sskip extends Stmt {
    void eval(Environment env) {}
}
```

\[ E, s_1 \rightarrow E_1 \quad E_1, s_2 \rightarrow E_2 \]

\[ E, s_1; s_2 \rightarrow E_2 \]

```java
class Sseq extends Stmt {
    void eval(Environment env) {
        s1.eval(env);
        s2.eval(env);
    }
}
```
evaluation of a statement

\[
E, e \rightarrow v \\
\frac{}{E, x \leftarrow e \rightarrow E\{x \mapsto v\}}
\]

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

(the environment is a mutable data structure)
evaluation of a statement

\[
\frac{E, e \rightarrow true \quad E, s_1 \rightarrow E_1}{E, \text{if } e \text{ then } s_1 \text { else } s_2 \rightarrow E_1}
\]

\[
\frac{E, e \rightarrow false \quad E, s_2 \rightarrow E_2}{E, \text{if } e \text{ then } s_1 \text { else } s_2 \rightarrow E_2}
\]

class Sif extends Stmt {
    void eval(Environment env) {
        if (e.eval(env).asBoolean())
            s1.eval(env);
        else
            s2.eval(env);
    }
}
evaluation of a statement

\[
\frac{E, e \rightarrow \text{true} \quad E, s \rightarrow E_1 \quad E_1, \text{while } e \text{ do } s \rightarrow E_2}{E, \text{while } e \text{ do } s \rightarrow E_2}
\]

\[
\frac{E, e \rightarrow \text{false}}{E, \text{while } e \text{ do } s \rightarrow E}
\]

class Swhile extends Stmt {
    void eval(Environment env) {
        while (e.eval(env).asBoolean())
            s.eval(env);
    }
}
we can do the same in OCaml

pattern matching plays the role of dynamic methods

```ocaml
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

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

```java
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

<table>
<thead>
<tr>
<th></th>
<th>horizontal extension</th>
<th>vertical extension</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>= adding a case</td>
<td>= adding a function</td>
</tr>
<tr>
<td>Java</td>
<td><strong>easy</strong></td>
<td>painful</td>
</tr>
<tr>
<td></td>
<td>(one file)</td>
<td>(several files)</td>
</tr>
<tr>
<td>OCaml</td>
<td>painful</td>
<td><strong>easy</strong></td>
</tr>
<tr>
<td></td>
<td>(several files)</td>
<td>(one file)</td>
</tr>
</tbody>
</table>
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

```java
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

```java
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

```java
class Interp implements Interpreter {
    Environment env = new Environment();
    Value interp(Ecte e) {
        return e.v;
    }
    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());
            ...
        }
    }
    ...
}
```
natural semantics makes no distinction between programs that crash, such as

\[ 1 + \text{true} \]

and programs whose evaluation does not terminate, such as

```
while true do skip
```
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

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

2. evaluation stuck on $E_n, s_n$

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

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
small-step operational semantics for **WHILE**

\[
\begin{align*}
E, e & \rightarrow v \\
E, x & \leftarrow e \rightarrow E \{ x \leftarrow v \}, \text{skip} \\
E, \text{skip}; s & \rightarrow E, s \\
E, s & \rightarrow E_1, s'_1 \\
E_1, s'_1; s_2 & \rightarrow E_1, s'_1; s_2 \\
E, \text{if } e \text{ then } s_1 \text{ else } s_2 & \rightarrow E, s_1 \\
E, \text{if } e \text{ then } s_1 \text{ else } s_2 & \rightarrow E, s_2 \\
E, \text{true} & \rightarrow \text{true} \\
E, \text{while } e \text{ do } s & \rightarrow E, s; \text{while } e \text{ do } s \\
E, \text{false} & \rightarrow \text{false} \\
E, \text{while } e \text{ do } s & \rightarrow E, \text{skip}
\end{align*}
\]
Proposition (equivalence of the two semantics)

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

\[ E, s \rightarrow E' \quad \text{if and only if} \quad E, s \rightarrow^* E', \text{skip} \]

(where \( \rightarrow^* \) is the reflexive transitive closure of \( \rightarrow \)).
Proposition (big steps imply small steps)

If \( E, s \rightarrow E' \), then \( E, s \rightarrow^* E', \text{skip} \).

by induction on the derivation \( E, s \rightarrow E' \) and by case on the last rule

- case of \( s_1; s_2 \)

\[
\begin{align*}
E, s_1 & \rightarrow E_1 \\
E_1, s_2 & \rightarrow E_2
\end{align*}
\]

\[
E, s_1; s_2 \rightarrow E_2
\]

then \( E, s_1 \rightarrow^* E_1, \text{skip} \) by IH

consequently,

\[
\begin{align*}
E, s_1; s_2 & \rightarrow^* E_1, \text{skip}; s_2 \quad \text{(small steps for ;)} \\
& \rightarrow E_1, s_2 \\
& \rightarrow^* E_2, \text{skip} \quad \text{(IH)}
\end{align*}
\]
• case of while e do s

if \[ E, e \rightarrow \text{true} \quad E, s \rightarrow E_1 \quad E_1, \text{while } e \text{ do } s \rightarrow E_2 \]

then

\[ E, \text{while } e \text{ do } s \rightarrow E, s; \text{while } e \text{ do } s \]
\[ \rightarrow^* E_1, \text{skip}; \text{while } e \text{ do } s \quad (\text{IH} + \text{rule };) \]
\[ \rightarrow E_1, \text{while } e \text{ do } s \]
\[ \rightarrow^* E_2, \text{skip} \quad (\text{IH}) \]

exercise: do the other cases
Lemma

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

by induction over $\rightarrow$

- case $s_1 = u_1; v_1$
  - case $u_1 = \text{skip}$
    
      we have $E_1, \text{skip}; v_1 \rightarrow E_1, v_1 \rightarrow E'$ and thus
      \[
      \frac{E_1, \text{skip} \rightarrow E_1 \quad E_1, v_1 \rightarrow E'}{E_1, \text{skip}; v_1 \rightarrow E'}
      \]

- case $u_1 \neq \text{skip}$
  
    we have $E_1, u_1; v_1 \rightarrow E_2, u_2; v_1 \rightarrow E'$ that is $E_1, u_1 \rightarrow E_2, u_2$ and
    \[
    \frac{E_2, u_2 \rightarrow E'_2 \quad E'_2, v_1 \rightarrow E'}{E_2, u_2; v_1 \rightarrow E'}
    \]
    by IH we deduce
    \[
    \frac{E_1, u_1 \rightarrow E'_2 \quad E'_2, v_1 \rightarrow E'}{E_1, u_1; v_1 \rightarrow E'}
    \]

(do the other cases)
we deduce

**Proposition (small steps imply big steps)**

\[
\text{Si } E, s \rightarrow^* E', \text{skip, alors } E, s \rightarrow E'.
\]

proof: we have

\[
E, s \rightarrow E_1, s_1 \rightarrow E_2, s_2 \rightarrow \cdots \rightarrow E_n, s_n \rightarrow E', \text{skip}
\]

but \( E', \text{skip} \rightarrow E' \) so \( E_n, s_n \rightarrow E' \) by the lemma above, then \( E_{n-1}, s_{n-1} \rightarrow E' \) by the same lemma, etc., until we get \( E, s \rightarrow E' \)

(induction of the number \( n \) of steps)
the lecture notes also contain the operational semantics for Mini-ML

\[
e \ ::= \ x \quad \text{identifier} \\
| \quad c \quad \text{constant } (1, 2, \ldots, \text{true}, \ldots) \\
| \quad op \quad \text{primitive } (+, \times, \text{fst}, \ldots) \\
| \quad \text{fun } x \rightarrow e \quad \text{anonymous function} \\
| \quad e \ e \quad \text{application} \\
| \quad (e, e) \quad \text{pair} \\
| \quad \text{let } x = e \text{ in } e \quad \text{local binding}
\]

(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”:

$$
e \xrightarrow{\ast} \downarrow \approx \xrightarrow{\ast} C(e)
$$

where $v \approx v'$ states that values $v$ and $v'$ coincide
let us consider arithmetic expression 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

\[
\begin{align*}
n &= n_1 + n_2 & e_1 &\rightarrow e_1' \\
n_1 + n_2 \rightarrow n & e_1 + e_2 &\rightarrow e_1' + e_2 \\
e_2 &\rightarrow e_2' & n_1 + e_2 &\rightarrow n_1 + e_2'
\end{align*}
\]
similarly, we set a small-step semantics for the target language

\[
m ::= \text{movq} \ $n, r \\
    \quad \mid \text{addq} \ $n, r \mid \text{addq} \ r, r \\
    \quad \mid \text{movq} \ (r), r \mid \text{movq} \ r, (r) \\
\]

\[
r ::= \%rdi \mid \%rsi \mid \%rsp
\]

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

\[
R ::= \{ \%rdi \mapsto n; \%rsi \mapsto n; \%rsp \mapsto n \}
\]

\[
M ::= \mathbb{N} \rightarrow \mathbb{Z}
\]

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:

\[
\begin{align*}
R, M, \text{movq} \ $n, r & \xrightarrow{m} R\{r \mapsto n\}, M \\
R, M, \text{addq} \ $n, r & \xrightarrow{m} R\{r \mapsto R(r) + n\}, M \\
R, M, \text{addq} \ r_1, r_2 & \xrightarrow{m} R\{r_2 \mapsto R(r_1) + R(r_2)\}, M \\
R, M, \text{movq} \ (r_1), r_2 & \xrightarrow{m} R\{r_2 \mapsto M(R(r_1))\}, M \\
R, M, \text{movq} \ r_1, (r_2) & \xrightarrow{m} R, M\{R(r_2) \mapsto R(r_1)\}
\end{align*}
\]
the final value of an expression is stored in \%rdi

\[
\begin{align*}
code(n) &= \text{movq } $n, \%rdi} \\
code(e_1 + e_2) &= \text{code}(e_1) \\
&\quad \text{addq } $-8, \%rsp} \\
&\quad \text{movq } \%rdi,(\%rsp} \\
&\quad \text{code}(e_2) \\
&\quad \text{movq } (\%rsp),\%rsi} \\
&\quad \text{addq } $8,\%rsp} \\
&\quad \text{addq } \%rsi,\%rdi
\end{align*}
\]
we seek to prove that if
\[ e \xrightarrow{\ast} n \]
and if
\[ R, M, \text{code}(e) \xrightarrow{m}^\ast R', M' \]
then \[ R'(%rdi) = n \]
we proceed by structural induction on \( e \)
we show a stronger property (an **invariant**), namely

if $e \stackrel{*}{\longrightarrow} n$ and $R, M, \text{code}(e) \stackrel{m}{\longrightarrow}^* R', M'$ then

\[
\begin{align*}
R'(\%rdi) &= n \\
R'(\%rsp) &= R(\%rsp) \\
\forall a \geq R(\%rsp), \ M'(a) &= M(a)
\end{align*}
\]
• case $e = n$

we have $e \rightarrow^* n$ and $\text{code}(e) = \text{movq} \; $n, %rdi$ and the result is immediate

• case $e = e_1 + e_2$

we have $e \rightarrow^* n_1 + e_2 \rightarrow^* n_1 + n_2$ with $e_1 \rightarrow^* n_1$ and $e_2 \rightarrow^* n_2$

thus we can invoke the induction hypothesis on $e_1$ and $e_2$
correctness of the compiler

<table>
<thead>
<tr>
<th>Function, Code</th>
<th>Result, Memory (R, M)</th>
<th>Induction Hypothesis</th>
</tr>
</thead>
</table>
| **code(e₁)**  | $R₁, M₁$              | $R₁(\%rdi) = n₁$ and $R₁(\%rsp) = R(\%rsp)$  
∀$a \geq R(\%rsp)$, $M₁(a) = M(a)$ |
| Addq $-8, \%rsp$ | $R₁', M₁'$           | $R₁' = R₁\{\%rsp \mapsto R(\%rsp) - 8\}$  
$M₁' = M₁\{R(\%rsp) - 8 \mapsto n₁\}$ |
| Movq $\%rdi, (\%rsp)$ | $R₂, M₂$              | $R₂(\%rdi) = n₂$ and $R₂(\%rsp) = R(\%rsp) - 8$  
∀$a \geq R(\%rsp) - 8$, $M₂(a) = M₁'(a)$ |
| **code(e₂)**  | $R₂, M₂$              | $R₂(\%rdi) = n₂$ and $R₂(\%rsp) = R(\%rsp) - 8$  
∀$a \geq R(\%rsp) - 8$, $M₂(a) = M₁'(a)$ |
| Movq $(\%rsp), \%rsi$ | $R', M₂$              | $R'(\%rdi) = n₁ + n₂$  
$R'(\%rsp) = R(\%rsp) - 8 + 8 = R(\%rsp)$  
∀$a \geq R(\%rsp)$,  
$M₂(a) = M₁'(a) = M₁(a) = M(a)$ |
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
  • a mini-Python interpreter
  • in Java or OCaml (your choice)

• lecture 3
  • parsing

> ./mini-python tests/good/pascal.py
*  
** 
*** 
**** 
***** 
****** 
******* 
********* 
*0000000*  
**000000**  
***000000***  
****000000****  
*****000000*****  
******000000******  
********000000*******