École Polytechnique

CSC_52064 – Compilation

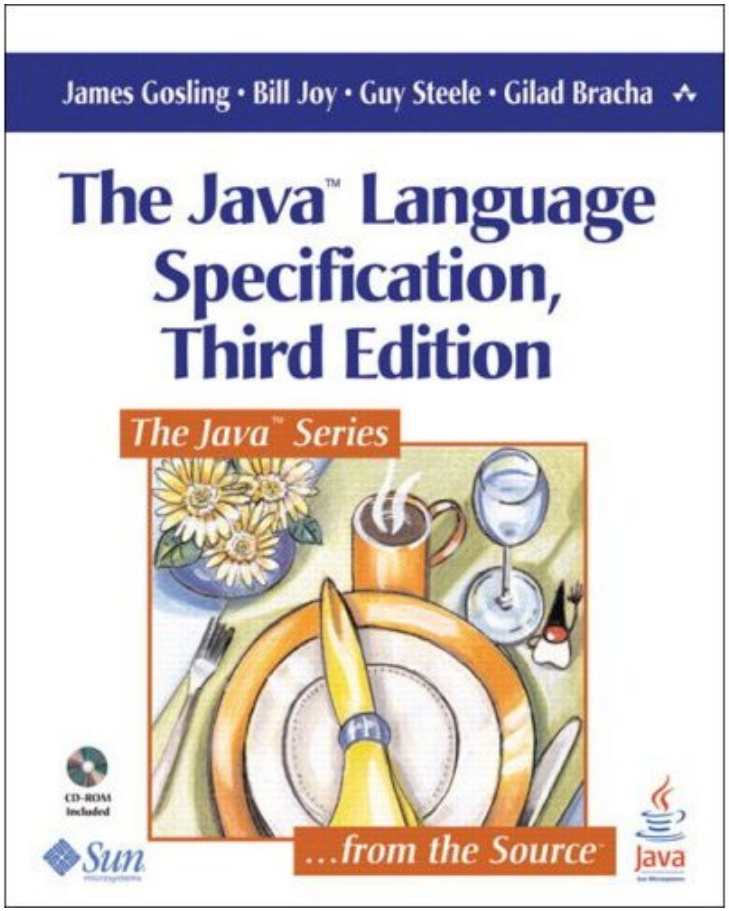
Jean-Christophe Filliâtre

syntaxe abstraite, sémantique

comment définir la signification des programmes écrits dans un langage ?

la plupart du temps, on se contente d'une description informelle, en langue naturelle (norme ISO, standard, ouvrage de référence, etc.)

s'avère peu satisfaisant, car souvent imprécis, voire ambigu



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.

la **sémantique formelle** caractérise mathématiquement les calculs décrits par un programme

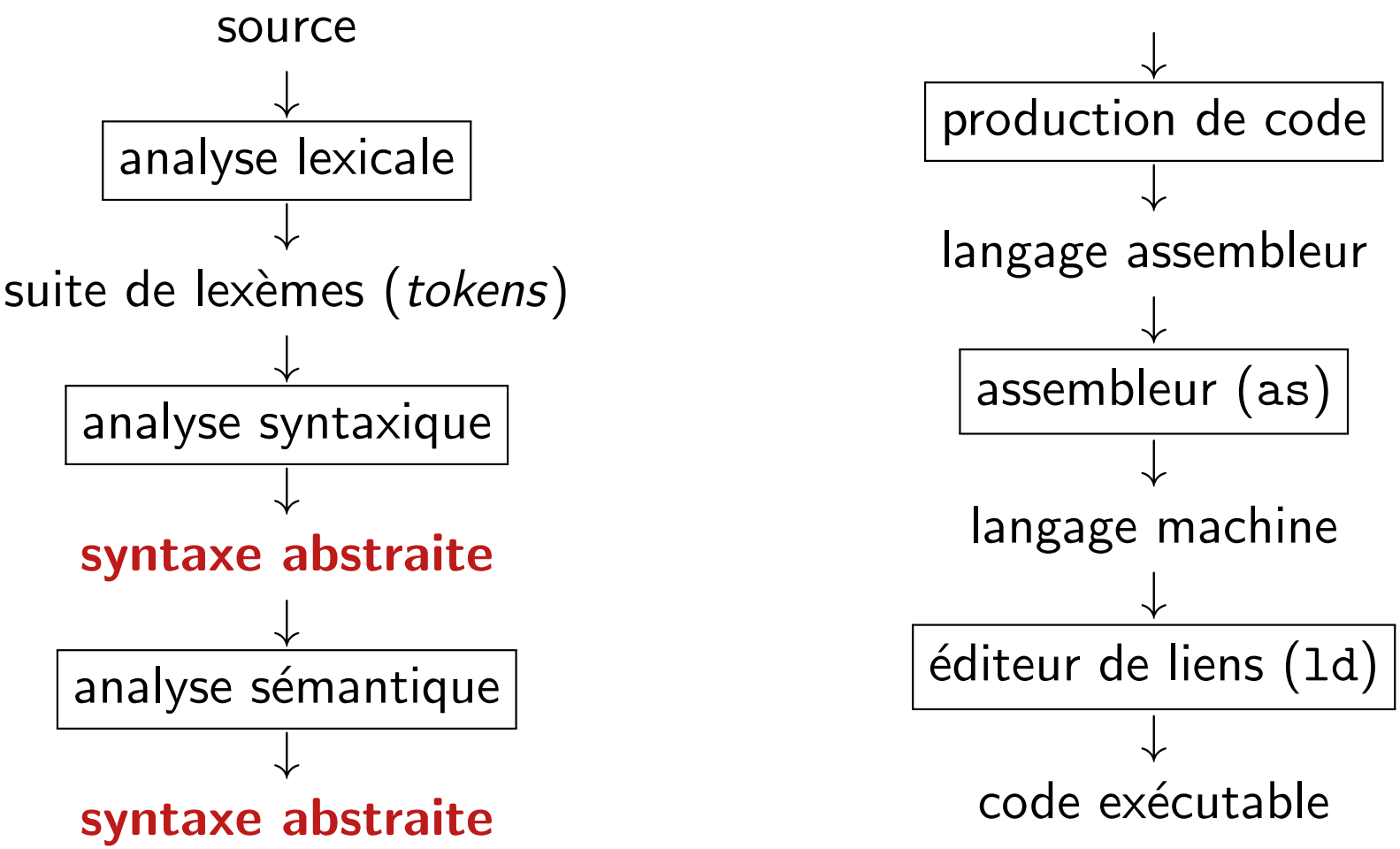
utile pour la réalisation d'outils (interprètes, compilateurs, etc.)

indispensable pour raisonner sur les programmes

mais qu'est-ce qu'un programme ?

en tant qu'objet syntaxique (suite de caractères),
il est trop difficile à manipuler

on préfère utiliser la **syntaxe abstraite**


source
analyse lexicale
suite de lexèmes (*tokens*)
analyse syntaxique
syntaxe abstraite
analyse sémantique
syntaxe abstraite
production de code
langage assembleur
assembleur (as)
langage machine
éditeur de liens (ld)
code exécutable

les textes

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

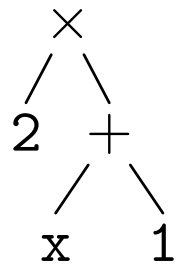
et

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

et

```
2 * /* je multiplie par deux */ ( x + 1 )
```

représentent tous le même **arbre de syntaxe abstraite**



on définit une syntaxe abstraite par une **grammaire**

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

se lit « une expression, notée e, est

- soit une constante c,
- soit une variable x,
- soit l'addition de deux expressions,
- etc. »

la notation $e_1 + e_2$ de la syntaxe abstraite emprunte le symbole de la syntaxe concrète

mais on aurait pu tout aussi bien choisir $Add(e_1, e_2)$, $+(e_1, e_2)$, etc.

représentation de la syntaxe abstraite en Java

on réalise les arbres de syntaxe abstraite par des classes :

```
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; }
...
```

(constructeurs omis)

l'expression `2 * (x + 1)` est représentée par

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

représentation de la syntaxe abstraite en OCaml

on réalise les arbres de syntaxe abstraite par des types algébriques

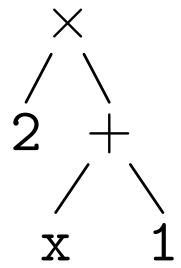
```
type binop = Add | Mul | ...

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

l'expression `2 * (x + 1)` est représentée par

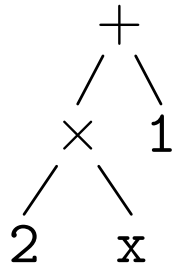
```
Bin (Mul, Cte 2, Bin (Add, Var "x", Cte 1))
```

il n'y a pas de constructeur dans la syntaxe abstraite pour les parenthèses

dans la syntaxe **concrète** `2 * (x + 1)`,
les parenthèses servent à reconnaître cet arbre



plutôt que



(on expliquera comment dans un autre cours)

on appelle **sucre syntaxique** une construction de la syntaxe concrète qui n'existe pas dans la syntaxe abstraite

elle est donc traduite à l'aide d'autres constructions de la syntaxe abstraite (généralement à l'analyse syntaxique)

exemples :

- en C, l'expression `a[i]` est du sucre pour `*(a+i)`
- en Java, l'expression `x -> {...}` est du sucre pour la construction d'un objet dans une classe anonyme qui implémente `Function`
- en OCaml, l'expression `[`e_1`;` e_2`; ...;` e_n`]` est du sucre pour e_1 `::` e_2 `::` ... `::` e_n `::` `[]`

c'est sur la syntaxe abstraite que l'on va définir la sémantique

il existe de nombreuses approches

- sémantique axiomatique
- sémantique dénotationnelle
- sémantique par traduction
- sémantique opérationnelle

encore appelée **logique de Hoare**
(*An axiomatic basis for computer programming*, 1969)

caractérise les programmes par l'intermédiaire des propriétés satisfaites par les variables ; on introduit le triplet

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

signifiant « si la formule P est vraie avant l'exécution de l'instruction i, alors la formule Q sera vraie après »

exemple :

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

exemple de règle :

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

la **sémantique dénotationnelle** associe à chaque expression e sa dénotation $[\![e]\!]$, qui est un objet mathématique représentant le calcul désigné par e

exemple : expressions arithmétiques avec une seule variable x

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

la dénotation peut être une fonction qui associe à la valeur de x la valeur de l'expression

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

(encore appelée sémantique dénotationnelle à la Strachey)

on peut définir la sémantique d'un langage en le traduisant vers un langage dont la sémantique est déjà connue

un langage ésotérique dont la syntaxe est composée de huit caractères et dont la sémantique peut être définie par traduction vers le langage C

commande	traduction en C
(prélude)	`char array[30000] = {0};` `char *ptr = array;`
`>`	`++ptr;`
`<`	`--ptr;`
`+`	`++*ptr;`
`-`	`--*ptr;`
`.`	`putchar(*ptr);`
`,`	`*ptr = getchar();`
`[`	`while (*ptr) {`
`]`	`}`

la **sémantique opérationnelle** décrit l'enchaînement des calculs élémentaires qui mènent de l'expression à son résultat (sa valeur)

elle opère directement sur des objets syntaxiques (la syntaxe abstraite)

deux formes de sémantique opérationnelle

- « sémantique naturelle » ou « à grands pas » (*big-steps semantics*)

$$e \twoheadrightarrow v$$

- « sémantique à réductions » ou « à petits pas » (*small-steps semantics*)

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

illustrons la sémantique naturelle sur un petit fragment du langage C

e	::=		**expression**
	|	n	constante (entier 32 bits signé)
	|	x	variable
	|	e *op* e	opérateur binaire ($+$, $<$, etc.)

s	::=		**instruction**
	|	x`=`e`;`	affectation
	|	`if (`e`)` s `else` s	conditionnelle
	|	`while (`e`)` s	boucle
	|	`{` s `...` s `}`	bloc

exemple

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

sémantique opérationnelle à grands pas de WHILE

on cherche à définir une relation entre une expression e et une **valeur** v

$$e \twoheadrightarrow v$$

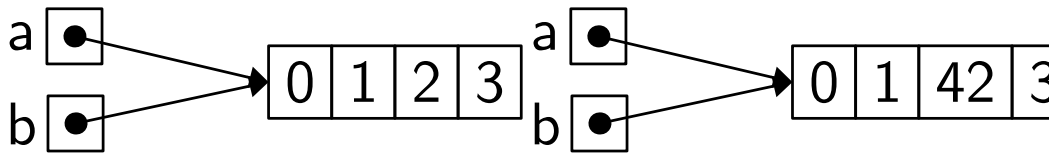
les valeurs sont ici réduites aux entiers

$$\begin{array}{rcll} v & ::= & & \textbf{valeur} \\ & | & n & \text{valeur entière (entier 32 bits signé)} \end{array}$$

attention : de manière générale, les valeurs ne coïncident pas nécessairement avec les constantes

avec un langage comme Java (ou Python, OCaml, etc.), une valeur peut être une adresse, sans pour autant qu'on ait d'adresse dans les constantes littérales du langage

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



on en reparlera plus loin (et dans le cours 5)

la valeur d'une variable est donnée par un **environnement** E
(une fonction des variables vers les valeurs)

on va donc définir une relation de la forme

$$E, e \twoheadrightarrow v$$

qui se lit comme « dans l'environnement E, l'expression e a la valeur v »

dans l'environnement

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

l'expression

$$a + b$$

a la valeur

$$89$$

ce que l'on note

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

une relation peut être définie comme la **plus petite relation** satisfaisant un ensemble de règles sans prémisses (axiomes) de la forme

$$\frac{}{P}$$

et un ensemble de règles avec prémisses de la forme

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

on appelle cela des **règles d'inférence**

exemple : on peut définir la relation Pair(n) par les deux règles

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

qui doivent se lire comme

$$\begin{array}{rl} \text{d'une part} & \text{Pair}(0) \\ \text{et d'autre part} & \forall n.\ \text{Pair}(n) \Rightarrow \text{Pair}(n+2) \end{array}$$

la plus petite relation satisfaisant ces deux propriétés coïncide avec la propriété « n est un entier pair » :

- les entiers pairs sont clairement dedans, par récurrence
- s'il y avait un entier impair, on pourrait enlever le plus petit

une **dérivation** est un arbre dont les nœuds correspondent à des règles avec prémisses et les feuilles à des axiomes ; exemple

$$\dfrac{\dfrac{\dfrac{}{\mathsf{Pair}(0)}}{\mathsf{Pair}(2)}}{\mathsf{Pair}(4)}$$

l'ensemble des dérivations possibles caractérise exactement la plus petite relation satisfaisant les règles d'inférence

- une constante n a pour valeur n

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

- une variable x a une valeur si $E(x)$ est défini

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

- une addition $e_1 + e_2$ a une valeur si e_1 a une valeur n_1, si e_2 a une valeur n_2 et si $n_1 + n_2$ ne fait pas de débordement arithmétique

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

- etc.

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

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

(on peut voir cet arbre comme une preuve)

il existe des expressions e pour lesquelles il n'y a pas de valeur v telle que $E, e \twoheadrightarrow v$

exemples :

- `x + 1` avec une variable `x` non définie dans E
- `2000000000 + 1000000000` car débordement arithmétique

ce sont là deux situations très différentes

- le cas d'une variable non définie est détecté au typage (cf cours 4) et le programme est rejeté

- le cas d'un débordement arithmétique (signé) est un **comportement non défini** du langage C (*undefined behavior*)

  le programme est accepté, compilé et exécuté,
  mais le compilateur est libre de faire **ce qu'il veut**

sur le code

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

le compilateur gcc produit

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

c'est-à-dire qu'il calcule `x <= 8`

avec `x` valant $2^{31} - 1$, cette fonction renvoie donc faux (0) alors que `x+1` vaudrait -2^{31} si on l'avait calculé et donc `x+1 < 10` serait vrai

une instruction peut modifier la valeur de certaines variables (affectations)

pour donner la sémantique d'une instruction s, on introduit donc la relation

$$E, s \twoheadrightarrow E'$$

qui se lit « dans l'environnement E, l'évaluation de l'instruction s termine et mène à l'environnement E' »

- si e a une valeur, alors l'affectation s'évalue et ajoute/écrase la variable `x`

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

- si le test e a une valeur, et si la branche correspondante s'évalue, alors `if` s'évalue

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

en posant $E = \{a \mapsto 21\}$

$$\dfrac{\dfrac{E, a \twoheadrightarrow 21 \quad E, 0 \twoheadrightarrow 0}{E, a > 0 \twoheadrightarrow 1} \qquad \dfrac{\dfrac{E, 2 \twoheadrightarrow 2 \quad E, a \twoheadrightarrow 21}{E, 2 \times a \twoheadrightarrow 42}}{E, \texttt{a=2} \times a\texttt{;} \twoheadrightarrow \{a \mapsto 42\}}}{E, \texttt{if (}a > 0\texttt{) a=2} \times a\texttt{; else \{ \}} \twoheadrightarrow \{a \mapsto 42\}}$$

- un bloc s'évalue si ses différentes instructions s'évaluent en séquence

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

- une boucle s'évalue si elle termine

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

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

il existe des instructions s qui ne s'évaluent pas

exemple : `while (1) { }`

(et plein d'autres exemples d'instructions impliquant des expressions qui ne s'évaluent pas)

pour établir une propriété d'une relation définie par un ensemble de règles d'inférence, on peut raisonner par **récurrence structurelle** sur la dérivation *i.e.* on peut appliquer l'hypothèse de récurrence à toute sous-dérivation

de manière équivalente, on peut dire que l'on raisonne par récurrence sur la hauteur de la dérivation

Proposition (déterminisme de l'évaluation)

Si $E, e \twoheadrightarrow v$ et $E, e \twoheadrightarrow v'$ alors $v = v'$.

par récurrence sur les dérivations de $E, e \twoheadrightarrow v$ et de $E, e \twoheadrightarrow v'$
cas d'une addition $e = e_1 + e_2$

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

avec $v = v_1 + n_2$ et $v' = n_1 + n_2$
or par HR on a $n_1 = n'_1$ et $n_2 = n'_2$ donc $v = v'$

(les autres cas sont identiques ou encore plus simples)

Proposition (déterminisme de l'évaluation)

Si $E, s \twoheadrightarrow E'$ et $E, s \twoheadrightarrow E''$ alors $E' = E''$.

exercice : faire cette preuve

remarque : dans le cas de la règle

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

on voit bien que la récurrence est faite sur la taille de la dérivation et non pas sur la taille de l'instruction (qui ne diminue pas)

remarque : une relation d'évaluation n'est pas nécessairement déterministe

exemple : on ajoute une primitive `random` pour tirer au hasard un entier 0 ou 1 et donc la règle

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

on a alors $E, \texttt{random()} \twoheadrightarrow 0$ aussi bien que $E, \texttt{random()} \twoheadrightarrow 1$

on peut programmer un **interprète** en suivant les règles de la sémantique naturelle

faisons-le en Java

comme expliqué plus haut

```
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; }
...
```

(constructeurs omis)

de même pour les instructions

```
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; }
```

commençons par le jugement

$$E, e \twoheadrightarrow v$$

l'environnement E est représenté par une classe

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

une solution consiste à ajouter une méthode

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

que l'on définit ensuite dans chaque sous-classe

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

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

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

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

(on pourrait vérifier l'absence de débordement arithmétique)

la méthode `eval` échoue **dynamiquement** sur une expression qui contient une variable non définie

dans ce cas précis, on aurait pu détecter cette erreur **statiquement** en faisant du typage (cf cours 4)

$$\begin{aligned} \text{statiquement} &= \text{pendant la compilation} \\ \text{dynamiquement} &= \text{pendant l'exécution} \end{aligned}$$

on procède de même pour les instructions en ajoutant une méthode dans la classe `Stmt`

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

que l'on définit dans chaque sous-classe

`eval` ne renvoie rien, car l'environnement est modifié en place

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

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

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

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

(on manipule ici un environnement mutable)

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

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

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

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

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

on peut faire la même chose en OCaml

le filtrage joue le rôle des méthodes dynamiques

```
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")
```

qu'est-ce qui distingue

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

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

en OCaml, le code de `eval` est à un seul endroit et traite tous les cas

en Java, il est éclaté dans l'ensemble des classes

brève comparaison fonctionnel / objet

	extension horizontale = ajout d'un cas	extension verticale = ajout d'une fonction
Java	**facile** (un seul fichier)	pénible (plusieurs fichiers)
OCaml	pénible (plusieurs fichiers)	**facile** (un seul fichier)

organisation différente du code Java

on peut souhaiter écrire le code Java autrement, avec

- des classes pour la syntaxe abstraite d'une part
- une classe pour l'interprète d'autre part

pour y parvenir, on peut utiliser le patron de conception du **visiteur**
(en anglais *visitor pattern*)

on commence par définir une interface pour un interprète

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

on utilise ici la **surcharge** de Java pour donner le même nom à toutes ces méthodes

dans la classe `Expr`, on fournit une méthode `accept` pour appliquer un interprète

```
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); }
}
```

c'est là notre seule intrusion dans les classes de la syntaxe abstraite

enfin, on peut écrire l'interprète dans une classe à part, qui implémente l'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());
      ...
  }
  ...
}
```

la sémantique naturelle ne permet pas de distinguer les programmes dont le calcul ≪ plante ≫, comme

$$x + 1$$

avec une variable x qui n'existe pas, des programmes dont l'évaluation ne termine pas, comme

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

la sémantique opérationnelle **à petits pas** y remédie en introduisant une notion d'étape élémentaire de calcul $E_1, s_1 \to E_2, s_2$, que l'on va itérer

on peut alors distinguer trois situations

1. l'itération termine

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

2. l'itération bloque sur E_n, s_n

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

3. l'itération ne termine pas

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

on peut conserver la sémantique à grands pas pour les expressions, car les expressions terminent toujours

mais pour un langage plus complexe (avec par exemple des appels de fonctions dans les expressions), on utiliserait aussi une sémantique à petits pas pour les expressions

sémantique opérationnelle à petits pas pour WHILE

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

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

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

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

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

on pourrait remplacer les deux règles pour `while` par la règle suivante

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

Proposition (équivalence des deux sémantiques)

Les deux sémantiques opérationnelles sont équivalentes pour les programmes dont l'évaluation termine, i.e.

$$E, s \twoheadrightarrow E' \quad \textit{si et seulement si} \quad E, s \to^\star E', \{\ \}$$

(où $\to^\star$ est la clôture réflexive transitive de $\to$).

Proposition (grands pas impliquent petits pas)

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

par récurrence sur la dérivation $E, s \twoheadrightarrow E'$ et par cas sur la dernière règle utilisée

- cas de $\{\ s_1\ s_2 \dots\ \}$

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

alors $E, s_1 \to^\star E_1, \{\ \}$ par HR
par conséquent,

$$\begin{array}{lll} E, \{\ s_1\ s_2 \dots\ \} & \to^\star & E_1, \{\ \{\ \}\ s_2 \dots\ \} \quad \text{(petits pas)} \\ & \to & E_1, \{\ s_2 \dots\ \} \\ & \to^\star & E_2, \{\ \} \qquad \text{(HR)} \end{array}$$

- cas de `while (e) s`
  si

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

  alors

$$\begin{array}{lll} E, \texttt{while (e) s} & \rightarrow & E, \texttt{\{ s while (e) s \}} \\ & \rightarrow^\star & E_1, \texttt{\{ \{ \} while (e) s \}} \quad \text{(HR + règle bloc)} \\ & \rightarrow & E_1, \texttt{\{ while (e) s \}} \\ & \rightarrow^\star & E_2, \texttt{\{ \}} \quad \text{(HR)} \end{array}$$

exercice : traiter les autres cas

Lemme

Si $E_1, s_1 \to E_2, s_2 \twoheadrightarrow E'$, *alors* $E_1, s_1 \twoheadrightarrow E'$.

par récurrence sur la dérivation $\twoheadrightarrow$

- cas $s_1 = \{\ u_1\ v_1 \ldots\ \}$
  - cas $u_1 = \{\ \}$
    on a $E_1, \{\{\ \}\ v_1 \ldots\ \} \to E_1, \{\ v_1 \ldots\ \} \twoheadrightarrow E'$ et donc
    $$\frac{E_1, \{\ \} \twoheadrightarrow E_1 \quad E_1, \{\ v_1 \ldots\ \} \twoheadrightarrow E'}{E_1, \{\{\ \}; v_1 \ldots\ \} \twoheadrightarrow E'}$$
  - cas $u_1 \neq \{\ \}$
    on a $E_1, \{\ u_1\ v_1 \ldots\ \} \to E_2, \{\ u_2\ v_1 \ldots\ \} \twoheadrightarrow E'$ *i.e.* $E_1, u_1 \to E_2, u_2$ et
    $$\frac{E_2, u_2 \twoheadrightarrow E_2' \quad E_2', \{\ v_1 \ldots\ \} \twoheadrightarrow E'}{E_2, \{\ u_2\ v_1 \ldots\ \} \twoheadrightarrow E'}$$
    par HR on en déduit
    $$\frac{E_1, u_1 \twoheadrightarrow E_2' \quad E_2', \{\ v_1 \ldots\ \} \twoheadrightarrow E'}{E_1, \{\ u_1\ v_1 \ldots\ \} \twoheadrightarrow E'}$$

(traiter les autres cas)

on en déduit

Proposition (petits pas impliquent grands pas)

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

preuve : on a

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

or $E', \{\ \} \twoheadrightarrow E'$ donc $E_n, s_n \twoheadrightarrow E'$ par le lemme précédent,
du coup $E_{n-1}, s_{n-1} \twoheadrightarrow E'$ par le lemme précédent, etc.,
jusqu'à finalement $E, s \twoheadrightarrow E'$ (récurrence sur le nombre d'étapes)

ajoutons des pointeurs à notre fragment de C

cela veut dire étendre la notion de valeur, d'expression et d'instruction

v	::=		**valeur**
	|	n	valeur entière (entier 32 bits signé)
	|	ℓ	adresse mémoire
e	::=	...	**expression**
	|	`malloc(4)`	allouer de la mémoire
	|	`*`e	lire en mémoire
s	::=	...	**instruction**
	|	`*`e`=`e`;`	écrire en mémoire

pour parler de la mémoire dans notre sémantique, on étend la relation sous la forme

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

où M est une fonction des adresses (ℓ) vers les entiers (n)

on ajoute une règle pour allouer de la mémoire

$$\frac{\ell \text{ une adresse qui n'est pas dans } M}{M, E, \texttt{malloc(4)} \twoheadrightarrow \ell}$$

et une autre pour lire dans la mémoire

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

on ajoute une règle pour écrire dans la mémoire

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

telle qu'elle est définie, notre sémantique ne permet pas de « confondre » entiers et pointeurs

en particulier, une expression de la forme `**e` n'aura jamais de valeur

on pourrait l'autoriser, mais au prix d'une sémantique bien plus complexe

ajoutons des **fonctions** à notre fragment de C

pour simplifier les choses, les fonctions ne renvoient rien,
mais n'en restent pas moins utiles :

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

et par ailleurs il n'y a que des variables locales

s	$::=$	...	**instruction**
	$\mid$	$f(e, \ldots, e);$	appel de fonction
p	$::=$	$d \ldots d$	**programme**
d	$::=$	`void` $f(x, \ldots, x)$ s	**définition de fonction**

sémantique d'un appel de fonction

pour qu'un appel de fonction s'évalue, il faut

- que la fonction existe, avec le bon nombre de paramètres
- que les arguments s'évaluent en des valeurs
- que le corps de la fonction s'évalue

$$\frac{\text{il existe une fonction } \texttt{void}\ f\,(x_1, \ldots, x_n)\ s \qquad M, E, e_i \twoheadrightarrow v_i \quad M, \{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\}, s \twoheadrightarrow M', E'}{M, E, f\,(e_1, \ldots, e_n)\,; \twoheadrightarrow M', E}$$

$$\frac{\text{il existe une fonction } \texttt{void}\ f\,(x_1, \ldots, x_n)\ s \qquad M, E, e_i \twoheadrightarrow v_i \quad M, \{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\}, s \twoheadrightarrow M', E'}{M, E, f\,(e_1, \ldots, e_n)\,; \twoheadrightarrow M', E}$$

noter comment

- le corps s est évalué dans un **nouvel environnement**, ne contenant que les x_i et dont la valeur finale E' est jetée
- on retrouve l'environnement E dont on était parti, inchangé
- la mémoire, en revanche, est possiblement modifiée

ici, l'**ordre d'évaluation** des arguments d'une fonction n'est pas significatif, car l'évaluation d'une expression n'a pas d'effet

dans le (vrai) langage C, en revanche, les arguments peuvent avoir des effets

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

et il n'est **pas spécifié** dans quel ordre ils sont évalués
(on parle de comportement *implementation-defined*)

on a évalué les arguments de l'appel **avant** de faire l'appel,
et on a passé leurs **valeurs**

on parle d'**appel par valeur**

c'est le choix de C, mais ce n'est pas la seule option (voir cours 5)

le poly contient également la sémantique opérationnelle d'un autre langage, Mini-ML

e	$::=$	x	identificateur
	$\mid$	c	constante (1, 2, ..., *true*, ...)
	$\mid$	op	primitive ($+$, $\times$, *fst*, ...)
	$\mid$	$\texttt{fun}\ x \to e$	fonction anonyme
	$\mid$	$e\ e$	application
	$\mid$	(e, e)	paire
	$\mid$	$\texttt{let}\ x = e\ \texttt{in}\ e$	liaison locale

(section 2.2, page 22)

application

---

correction d'un compilateur

un compilateur doit respecter la sémantique du langage

si le langage source est muni d'une sémantique $\rightarrow_s$ et le langage machine d'une sémantique $\rightarrow_m$, et si l'expression e est compilée en $C(e)$ alors on doit avoir un $\ll$ diagramme qui commute $\gg$:

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

où $v \approx v'$ exprime que les valeurs v et v' coïncident

considérons uniquement des expressions arithmétiques sans variable

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

et montrons la correction d'une compilation simple vers x86-64
en utilisant la pile pour les calculs

on se donne une sémantique à réductions pour le langage source

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

on se donne de même une sémantique à réductions pour le langage cible

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

un état est la donnée de valeurs pour les registres, R,
et d'un état de la mémoire, M

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

on définit la sémantique d'une instruction m par une réduction de la forme

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

la réduction $R, M, m \xrightarrow{m} R', M'$ est définie par

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

$$
\begin{aligned}
code(n) &= \texttt{movq \$}n\texttt{,\%rdi} \\
code(e_1 + e_2) &= code(e_1) \\
&\quad \texttt{addq \$-8,\%rsp} \\
&\quad \texttt{movq \%rdi,(\%rsp)} \\
&\quad code(e_2) \\
&\quad \texttt{movq (\%rsp),\%rsi} \\
&\quad \texttt{addq \$8,\%rsp} \\
&\quad \texttt{addq \%rsi,\%rdi}
\end{aligned}
$$

on souhaite montrer que si

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

et si

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

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

on procède par récurrence structurelle sur e

on établit un résultat plus fort (**invariant**), à savoir :

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

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

- cas $e = n$

  on a $e \xrightarrow{\star} n$ et $code(e) = \texttt{movq \$}n\texttt{,\%rdi}$ et le résultat est immédiat

- cas $e = e_1 + e_2$

  on a $e \xrightarrow{\star} n_1 + e_2 \xrightarrow{\star} n_1 + n_2$ avec $e_1 \xrightarrow{\star} n_1$ et $e_2 \xrightarrow{\star} n_2$
  ce qui nous permet d'invoquer l'hypothèse de récurrence sur e_1 et e_2

	R, M	
$code(e_1)$	R_1, M_1	par hypothèse de récurrence $R_1(\texttt{\%rdi}) = n_1$ et $R_1(\texttt{\%rsp}) = R(\texttt{\%rsp})$ $\forall a \geq R(\texttt{\%rsp}),\ M_1(a) = M(a)$
`addq $-8,%rsp` `movq %rdi,(%rsp)`	R'_1, M'_1	$R'_1 = R_1\{\texttt{\%rsp} \mapsto R(\texttt{\%rsp}) - 8\}$ $M'_1 = M_1\{R(\texttt{\%rsp}) - 8 \mapsto n_1\}$
$code(e_2)$	R_2, M_2	par hypothèse de récurrence $R_2(\texttt{\%rdi}) = n_2$ et $R_2(\texttt{\%rsp}) = R(\texttt{\%rsp}) - 8$ $\forall a \geq R(\texttt{\%rsp}) - 8,\ M_2(a) = M'_1(a)$

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

une telle preuve peut être effectuée sur un vrai compilateur

exemple : CompCert, un compilateur C produisant du code optimisé pour PowerPC, ARM, RISC-V et x86, a été formellement vérifié avec l'assistant de preuve Coq

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

la suite

- TD 2
  - écriture d'un interprète pour mini-Python
  - au choix en Java ou en OCaml
- prochain cours
  - analyse syntaxique

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