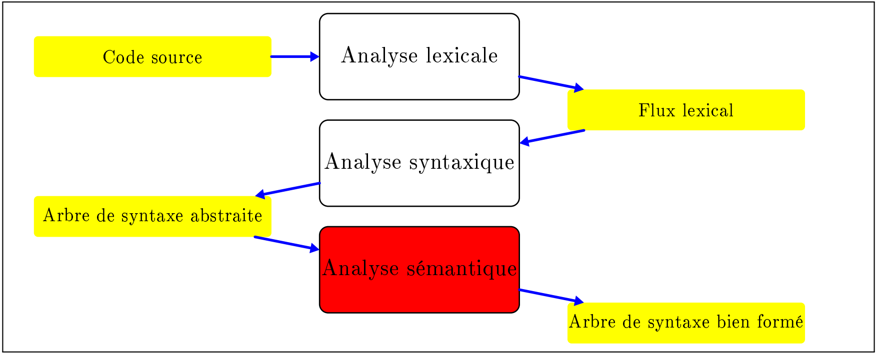
Introduction à la Compilation
Cours 7 : Syntaxe

Yann Régis-Gianas
yrg@pps.univ-paris-diderot.fr

PPS - Université Denis Diderot – Paris 7


Code source
Analyse lexicale
Flux lexical
Analyse syntaxique
Arbre de syntaxe abstraite
Analyse sémantique
Arbre de syntaxe bien formé

Qu'appelle-t-on "syntaxe" ?

Préliminaire : vocabulaire et notation

- À partir de maintenant, nous travaillerons uniquement sur des arbres.
- Plutôt que de dessiner ces arbres, nous continuerons à utiliser une notation textuelle mais qu'il faut interpréter implicitement comme un arbre.
- On définira des grammaires d'arbre, en gardant implicites les priorités et les associativités utilisées pour leur représentation textuelle. Par exemple :

$$\boxed{\begin{array}{rcl} e & ::= & n \\ & \mid & e + e \\ & \mid & e * e \end{array}}$$

  est une grammaire d'arbres, que nous appelerons aussi abusivement langage.
- Le fait que "1 + 2 * 3" représente l'arbre "1 + (2 * 3)" est implicite.
- Dans ces grammaires, les non-terminaux sont appelés "méta-variables".
- Les arbres obtenus seront appelés termes .

Exemple 1 : le langage des expressions arithmétiques

$$
\begin{array}{rcl}
e & ::= & n \\
& | & e + e \\
& | & e * e \\
& | & e / e \\
& | & e - e \\
& | & -e
\end{array}
$$

Exemple 2 : le langage des commandes MiniLogo

- Un programme est une liste d'instructions.
- Une instruction est de l'une des deux formes suivantes :
  - `Avance N` où `N` est un entier représentant un nombre de pixel.
  - `Tourne N` où `N` est un entier représentant un angle en degré.

devient, plus formellement :

$$
\begin{array}{lcll}
\mathit{prog} & ::= & \mathit{instr}\,;\,\mathit{prog} & \\
 & \mid & \mathbf{NeFaitRien} & \\
 & & & \\
\mathit{instr} & ::= & \mathbf{Avance}\ n & n \in \mathbb{N} \\
 & \mid & \mathbf{Tourne}\ d & d \in [0 \ldots 359]
\end{array}
$$

Le sens associé à un terme

- Les arbres ont une structure.
- Lors du premier cours, nous avons vu qu'il était intuitivement plus facile de donner du sens à un objet structuré plutôt qu'à un objet linéaire.
- Dans une structure hiérarchique, comme un arbre, on peut *donner un sens à la structure globale en s'appuyant sur le sens de ses sous-structures*.

$\Rightarrow$ Sujet du cours d'aujourd'hui :

La compréhension et la formulation de cette intuition

Comment définir le sens d'un terme ?

- Le sens d'un terme est une interprétation de ce terme.
- Avant toute chose, il faut donc se donner un domaine d'interprétation.
- Il y a une *infinité* de domaines intéressants.
- Par exemple, on peut interpréter une expression arithmétique comme :
  - un calcul dans $\mathbb{N}$ caractérisé par sa valeur finale $v \in \mathbb{N}$;
  - un calcul dans $\mathbb{N}$ caractérisé par sa valeur finale $v \in \mathbb{N}$ et la séquence des opérations atomiques qui a mené à cette valeur ;
  - un calcul dans $\mathbb{Z}/n\mathbb{Z}$, c'est-à-dire un calcul sur des entiers bornés ;
  - un entier représentant la hauteur de son arbre de syntaxe abstraite ;
  - une formule logique, un entier faisant référence à une variable propositionnelle.
  - ...

$\Rightarrow$ On peut observer une interprétation donnée à différents niveaux de détails.

- La relation associant termes et interprétations est appelée sémantique.

À l'aide du langage naturel ?

- Comment spécifier cette relation ?
- On ne veut pas énumérer tous les couples (terme, interprétation).
- La structure d'arbre suggère une définition récursive.

Exemple 1 : le langage des expressions arithmétiques

- Soit le langage :

$$\begin{array}{rcl} e & ::= & n \\ & | & e+e \\ & | & e*e \\ & | & e/e \\ & | & e-e \\ & | & -e \end{array}$$

- Une façon naturelle de donner du sens à toute expression e est :
  - <u>Si</u> $e \equiv n$ <u>alors</u> $sens(e) = n \in \mathbb{N}$
  - <u>Si</u> $e \equiv e_1 + e_2$ <u>et</u> $sens(e_1) = n_1$ <u>et</u> $sens(e_2) = n_2$ <u>alors</u> $sens(e) = n_1 +_{\mathbb{N}} n_2$
  - <u>Si</u> $e \equiv e_1 * e_2$ <u>et</u> $sens(e_1) = n_1$ <u>et</u> $sens(e_2) = n_2$ <u>alors</u> $sens(e) = n_1 *_{\mathbb{N}} n_2$
  - <u>Si</u> $e \equiv e_1 - e_2$ <u>et</u> $sens(e_1) = n_1$ <u>et</u> $sens(e_2) = n_2$ <u>alors</u> $sens(e) = n_1 -_{\mathbb{N}} n_2$
  - <u>Si</u> $e \equiv e_1/e_2$ <u>et</u> $sens(e_1) = n_1$ <u>et</u> $sens(e_2) = n_2$ <u>alors</u> $sens(e) = n_1/_{\mathbb{N}}n_2$
  - <u>Si</u> $e \equiv -e_1$ <u>et</u> $sens(e_1) = n_1$ <u>alors</u> $sens(e) = -_{\mathbb{N}}n_1$

Exemple 2 : le langage des commandes MiniLogo

$$
\begin{array}{lcll}
\textit{prog} & ::= & \textit{instr} ; \textit{prog} & \\
 & \mid & \textbf{NeFaitRien} & \\
 & & & \\
\textit{instr} & ::= & \textbf{Avance}\ n & n \in \mathbb{N} \\
 & \mid & \textbf{Tourne}\ d & d \in [0 \ldots 359]
\end{array}
$$

- Soit un stylo S, caractérisé par une position $(x, y) \in \mathbb{R}^2$ et un vecteur direction (dx, dy).
- Si *prog* $\equiv$ *instr*; *prog*′ alors *sens*(*prog*) est la transformation de S définie par "*sens*(*prog*) $\circ$ *sens*(*instr*)".
- Si *prog* $\equiv$ **NeFaitRien** alors *sens*(*prog*) est l'identité.
- Si *instr* $\equiv$ **Avance** n

  alors *sens*(*instr*) est $\left\{ \begin{array}{l} (x, y) \mapsto (x + n * dx, y + n * dy) \\ (dx, dy) \mapsto (dx, dy) \end{array} \right.$
- Si *instr* $\equiv$ **Tourne** d

  alors *sens*(*instr*) est $\left\{ \begin{array}{l} (x, y) \mapsto (x, y) \\ (dx, dy) \mapsto (dx * cos(d), dy * sin(d)) \end{array} \right.$

Exemple 3 : appels de procédure

$$\begin{array}{rcl}
\mathit{prog} & ::= & \mathit{definition}^{\star}\,;\,\mathit{expression} \\
\\
\mathit{definition} & ::= & \textbf{def}\ f\ (x_1, \ldots, x_n) = \mathit{expression} \\
\\
\mathit{expression} & ::= & n \\
& | & x \\
& | & \mathit{expression}\ \delta\ \mathit{expression} \\
& | & f\,(\mathit{expression}_1, \ldots, \mathit{expression}_n) \\
\\
\delta & ::= & + \mid - \mid / \mid *
\end{array}$$

Exercice

Sauriez-vous donner, en français, une sémantique “intéressante” à ce langage ?

À l'aide d'un programme O'CAML ?

```
type binop = Add | Mul | Sub | Div

type exp = Int of int | Binop of binop × exp × exp

let rec eval = function
  | Int x → x
  | Binop (Add, e1, e2) → eval e1 + eval e2
  | Binop (Mul, e1, e2) → eval e1 × eval e2
  | Binop (Sub, e1, e2) → eval e1 - eval e2
  | Binop (Div, e1, e2) → eval e1 / eval e2
```

Comment définir mathématiquement une relation sur les termes ?

Propriétés de la fonction O'CAML

```
type binop = Add | Mul | Sub | Div

type exp = Int of int | Binop of binop × exp × exp

let rec eval = function
  | Int x → x
  | Binop (Add, e1, e2) → eval e1 + eval e2
  | Binop (Mul, e1, e2) → eval e1 × eval e2
  | Binop (Sub, e1, e2) → eval e1 - eval e2
  | Binop (Div, e1, e2) → eval e1 / eval e2
```

- Cette fonction établit une relation (fonctionnelle) entre un terme et une valeur de type **int**.
- Elle est définie par récurrence.
- Que peut-on observer lors de l'évaluation d'une expression ?

Formalisation à l'aide de règles

- On peut s'inspirer de cette fonction pour formaliser l'évaluation à l'aide d'un jugement que l'on écrit :

$$e \Downarrow n$$

  que l'on *choisit de lire* :

  « *L'évaluation de e mène à l'entier n.* »

- Le jugement est valide si il est déduit de l'application d'une de ces règles :

(Const) « $n \Downarrow n$ »
(Add) « $e_1 + e_2 \Downarrow n_1 +_{\mathbb{N}} n_2$ » si « $e_1 \Downarrow n_1$ » et « $e_2 \Downarrow n_2$ »
(Sub) « $e_1 * e_2 \Downarrow n_1 *_{\mathbb{N}} n_2$ » si « $e_1 \Downarrow n_1$ » et « $e_2 \Downarrow n_2$ »
(Mul) « $e_1 - e_2 \Downarrow n_1 -_{\mathbb{N}} n_2$ » si « $e_1 \Downarrow n_1$ » et « $e_2 \Downarrow n_2$ »
(Div) « $e_1 / e_2 \Downarrow n_1 /_{\mathbb{N}} n_2$ » si « $e_1 \Downarrow n_1$ » et « $e_2 \Downarrow n_2$ »

- Par exemple, « $1 + 2 + 3 \Downarrow 6$ » car « $1 + 2 \Downarrow 3$ » et « $3 \Downarrow 3$ » ; et « $1 + 2 \Downarrow 3$ » car « $1 \Downarrow 1$ » et « $2 \Downarrow 2$ ».

Arbre de preuve

$$
\dfrac{
  \dfrac{1 \Downarrow 1 \qquad 2 \Downarrow 2}{1 + 2 \Downarrow 3}\,(\textsc{Add})
  \qquad
  \dfrac{}{3 \Downarrow 3}\,(\textsc{Const})
}{1 + 2 + 3 \Downarrow 6}\,(\textsc{Add})
$$

- La dérivation précédente se représente sous la forme d'un arbre.

Règle d'inférence

$$\frac{H_1 \dots H_n}{C}$$

- Une règle d'inférence est applicable si il existe un arbre de preuve pour chaque H_i, les hypothèses de la règle. On obtient alors un arbre de preuve pour la validité du jugement C, la conclusion de la règle.
- Une règle sans hypothèse est un axiome.

Les règles de l'évaluation des expressions arithmétiques

$$\text{(CONST)}\ \frac{}{n \Downarrow n} \qquad \text{(ADD)}\ \frac{e_1 \Downarrow n_1 \qquad e_2 \Downarrow n_2}{e_1 + e_2 \Downarrow n_1 +_{\mathbb{N}} n_2} \qquad \text{(MUL)}\ \frac{e_1 \Downarrow n_1 \qquad e_2 \Downarrow n_2}{e_1 * e_2 \Downarrow n_1 *_{\mathbb{N}} n_2}$$

$$\text{(SUB)}\ \frac{e_1 \Downarrow n_1 \qquad e_2 \Downarrow n_2}{e_1 - e_2 \Downarrow n_1 -_{\mathbb{N}} n_2} \qquad \text{(DIV)}\ \frac{e_1 \Downarrow n_1 \qquad e_2 \Downarrow n_2}{e_1 / e_2 \Downarrow n_1 /_{\mathbb{N}} n_2}$$

- Il faut **instancier** les méta-variables de ces règles pour les utiliser.
- Quel objet mathématique représente-t-on avec cet ensemble de règles ?

Relation inductive

- Soit un ensemble $\mathcal{U}$.
- Soit un ensemble de règles $(R_i)_{i \in I}$.
- On définit la fonctionnelle :

$$\left\{ \begin{array}{lcl} \mathfrak{P}(\mathcal{U}) & \rightarrow & \mathfrak{P}(\mathcal{U}) \\ A & \mapsto & \bigcup_{i \in I} \{ \, R_i(a_1, \ldots, a_n) \mid a_1, \ldots, a_n \in A, n \equiv arite(R_i) \, \} \end{array} \right.$$

- Elle possède un plus petit point fixe (théorème de Tarski).

Exemple 1 : « être pair »

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

Exemple 2 : « être un arbre binaire de recherche »

$$\frac{}{ADR(\bullet)} \qquad \frac{ADR(t_1) \qquad ADR(t_2) \qquad PP(x, t_1) \qquad PG(x, t_2)}{ADR(\text{nœud}(t_1, x, t_2))}$$

$$\frac{PP(x, t_1) \qquad PP(x, t_2) \qquad x > y}{PP(x, \text{nœud}(t_1, y, t_2))} \qquad \frac{}{PP(x, \bullet)}$$

$$\frac{PG(x, t_1) \qquad PG(x, t_2) \qquad x < y}{PG(x, \text{nœud}(t_1, y, t_2))} \qquad \frac{}{PG(x, \bullet)}$$

- $\bullet \equiv$ “l’arbre vide”.
- $ADR(t) \equiv$ “t est un arbre binaire de recherche”
- $PP(x, t) \equiv$ “tous les entiers de t sont plus petits que x”
- $PG(x, t) \equiv$ “tous les entiers de t sont plus grands que x”

Quelques définitions inductives . . .

Définition inductive des termes

- On peut définir les termes de façon inductive.
- Soit Σ, un ensemble et une fonction *arite* : $\Sigma \rightarrow \mathbb{N}$
- Soit $\mathcal{U}$, l'ensemble des arbres étiquetés par des éléments de Σ.

$$(\textsc{Construct})\ \frac{t_1 \ldots t_n}{f(t_1, \ldots, t_n)} \qquad \text{si n} = \text{arite (f)}$$

- Cette règle a une condition de bord.
- Elle contraint l'application de la règle.
- Les seuls arbres "bien formés" sont ceux qui respectent l'arité de f.
- L'ensemble obtenu s'appelle une $\Sigma-$algèbre.

Exemple 1 : Les multiplications

- On choisit $\Sigma = \{0, 1, \ldots, *_2\}$.
- L'indice 2 signifie que l'opérateur $*$ est d'arité 2.

Exemple 2 : Les expressions arithmétiques

- On choisit $\Sigma = \{0, 1, \ldots, *_2, +_2, -_2, /_2\}$.

Exemple 3 : . . . avec branchement conditionnel

- On choisit $\Sigma = \{\mathbf{true}, \mathbf{false}, 0, 1, \ldots, \mathbf{if}_3, <_2, >_2, =_2, *_2, +_2, -_2, /_2\}$.

$\Rightarrow$ Les grammaires de termes sont tout de même plus lisibles.

Définition d'un prédicat inductif sur les termes

- Pour raisonner sur les termes nous allons utiliser des prédicats inductifs.
- Ses prédicats porteront souvent sur les termes.
- De plus, on aura très souvent une règle par constructeur de la syntaxe.
- On dit alors que le système de règles obtenu est dirigé par la syntaxe.

Exemple : « une multiplication qui contient un 0 »

$$\frac{}{Z(0)} \qquad \frac{Z(e_1)}{Z(e_1 * e_2)} \qquad \frac{Z(e_2)}{Z(e_1 * e_2)}$$

- $Z(e)$ est lu "l'expression e contient un zéro."
- Que dire du jugement $Z(1)$?

Exemple : « une multiplication qui contient un 0 »

$$\frac{}{Z(0)} \qquad \frac{Z(e_1)}{Z(e_1 * e_2)} \qquad \frac{Z(e_2)}{Z(e_1 * e_2)}$$

- $Z(e)$ est lu "l'expression e contient un zéro."
- Que dire du jugement $Z(1)$?

$\Rightarrow$ Il n'est pas **dérivable** à partir de cet ensemble de règles.

Exemple : « une multiplication qui contient un 0 »

$$\frac{}{Z(0)} \qquad \frac{Z(e_1)}{Z(e_1 * e_2)} \qquad \frac{Z(e_2)}{Z(e_1 * e_2)}$$

- $Z(e)$ est lu "l'expression e contient un zéro."
- Que dire du jugement $Z(1)$?

$\Rightarrow$ Il n'est pas **dérivable** à partir de cet ensemble de règles.

- Ce système est-il dirigé par la syntaxe ?

Exemple : « une multiplication qui contient un 0 »

$$\frac{}{Z(0)} \qquad \frac{Z(e_1)}{Z(e_1 * e_2)} \qquad \frac{Z(e_2)}{Z(e_1 * e_2)}$$

- $Z(e)$ est lu "l'expression e contient un zéro."
- Que dire du jugement $Z(1)$?

$\Rightarrow$ Il n'est pas **dérivable** à partir de cet ensemble de règles.

- Ce système est-il dirigé par la syntaxe ?

$\Rightarrow$ Non ! Il y a deux règles pour le cas "$e_1 * e_2$"

Définition d'une relation inductive sur les termes

- Les règles que nous avons écrit pour définir l'évaluation d'une expression arithmétique ne définissent pas un prédicat inductif mais une relation inductive entre un terme et un entier.
- Elles sont dirigées par la syntaxe du terme.
- De plus, on peut prouver :

$$\forall e\ v_1\ v_2,\ e \Downarrow v_1 \ \wedge\ e \Downarrow v_2 \Rightarrow v_1 = v_2$$

$\Rightarrow$ Comment ?

Preuve par induction sur les termes

- Par induction sur les termes e :
  - Les cas de base sont les constructeurs d'arité 0. Ici, les entiers. On doit prouver :
    $$\forall n, n \Downarrow n \;\wedge\; n \Downarrow n \Rightarrow n = n$$
    C'est vrai par réflexivité de l'égalité.
  - Soit $e \equiv e_1 + e_2$. L'hypothèse d'induction nous apprend que
    $$\begin{array}{c}\forall n_1\ n_1', e_1 \Downarrow n_1 \;\wedge\; e_1 \Downarrow n_1' \Rightarrow n_1 = n_1' \\ \forall n_2\ n_2', e_2 \Downarrow n_2 \;\wedge\; e_2 \Downarrow n_2' \Rightarrow n_2 = n_2'\end{array}$$
    Soient n et n' tels que $e_1 + e_2 \Downarrow n$ et $e_1 + e_2 \Downarrow n'$.
    Seule règle (Add) peut avoir dérivé chacun de ces jugements.
    Pour être dérivé, le premier (resp. le second) jugement a nécessairement utilisé un certain k_1 (resp. k_1') et une dérivation de $e_1 \Downarrow k_1$ (resp. $e_1 \Downarrow k_1$) ainsi qu'un certain k_2 (resp. k_2') et une dérivation de $e_2 \Downarrow k_2$ (resp. $e_2 \Downarrow k_2'$), tels que $n = k_1 + k_2$ et $n' = k_1' + k_2'$. Par application de l'hypothèse d'induction, $k_1 = k_1'$ et $k_2 = k_2'$ et donc $n = n'$.
  - On pourrait traiter les autres cas de façon similaire.

Principe de preuve par induction (Rappel)

- Soit E, un ensemble.
- Soit $<$, un ordre strict bien fondé.
- Montrer une propriété P sur E par induction sur $<$, c'est :
  1. Montrer que P est vraie pour tous les éléments minimaux, c'est-à-dire :
  $$\forall x, x \text{ minimal } \Rightarrow P(x)$$
  2. Montrer que P est vraie pour tout élément x de E si on suppose que P est vraie pour les prédécesseurs de x, c'est-à-dire :
  $$\forall x, (\forall y, y < x \Rightarrow P(y)) \Rightarrow P(x)$$

Preuve par induction sur une relation ou un prédicat inductif

- Dans le cas des arbres de preuve, la relation d'ordre « est une hypothèse de » est un ordre strict. On peut donc montrer des propriétés vraies pour toute dérivation d'une certaine relation ou d'un prédicat.

Exemple de preuve par induction sur une relation inductive

- Montrons que :
$$\forall e, Z(e) \Rightarrow \forall n, e \Downarrow n \Rightarrow n = 0$$
par induction sur les arbres de dérivation de la forme $Z(e)$, pour tout e.
- Cas de base. Ici, il s'agit de l'axiome :
$$\frac{}{Z(0)}$$
Cela signifie que $e \equiv 0$. Or, si on a aussi une dérivation de $0 \Downarrow n$, cela implique nécessairement que $n = 0$.
- Cas d'induction. Ici, il y a deux sous-cas :
$$\frac{Z(e_1)}{Z(e_1 * e_2)} \qquad \frac{Z(e_2)}{Z(e_1 * e_2)}$$
Supposons être dans le premier cas. On a par induction :
$$Z(e_1) \Rightarrow \forall n, e_1 \Downarrow n \Rightarrow n = 0$$
De plus, la dérivation $e_1 * e_2 \Downarrow n$ résulte nécessairement de l'application de la règle $(\textsc{Mul})$ donc il existe n_1 tel que $n = n_1 *_{\mathbb{N}} n_2$ et $e_1 \Downarrow n_1$. Or, l'induction nous apprend que $n_1 = 0$ donc $n = 0$.

Comment définir une sémantique ?

Sémantique opérationnelle à grands pas

- La sémantique que nous avons définie *via* le jugement « $e \Downarrow n$ » est une relation entre un terme e et l'entier n qui résulte de son évaluation.
- Une telle sémantique est dite à grands pas car elle s'attache à l'observation du **résultat** d'un calcul et **non aux étapes précises** qui y mènent.
- On dit aussi que c'est une sémantique naturelle.

Sémantique opérationnelle à petits pas

- Si on est intéressé par les étapes du calcul, une sémantique à grands pas est plus difficile à utiliser.
- On utilise couramment un système de réécriture pour définir formellement **un pas** d'évaluation.
- Ce système de réécriture définit un jugement de la forme « $e \rightarrow e'$ » qui se lit : « Une étape d'évaluation réécrit le terme e en le terme e'. »
- Une sémantique spécifiée ainsi est dite à petits pas ou encore structurelle.

Exemple

$$\frac{e_1 \to e_1'}{e_1 \oplus e_2 \to e_1' \oplus e_2} \qquad \frac{e_2 \to e_2'}{n_1 \oplus e_2 \to n_1 \oplus e_2'}$$

$$\frac{}{n_1 \oplus n_2 \to m} \quad \text{avec } m = n_1 \oplus_{\mathbb{N}} n_2$$

- Il faudrait écrire ces trois règles pour chaque opérateur $+, -, /, *$.
  (Ici, on les a factorisé : $\oplus$ peut être choisi parmi $+, -, /, *$.)
- On remarque que les jugements « $1 + 2 * 3 \to 1 + 6$ » et « $1 + 6 \to 7$ » sont dérivables. On écrit cela généralement en les composant ainsi :

$$1 + 2 * 3 \to 1 + 6 \to 7$$

Classification des règles

- On note deux types de règles.
- Les règles de passage au contexte :

$$\frac{e_1 \to e_1'}{e_1 \oplus e_2 \to e_1' \oplus e_2} \qquad \frac{e_2 \to e_2'}{n_1 \oplus e_2 \to n_1 \oplus e_2'}$$

  qui permettent la réécriture d'un sous-terme.
- et les règles de réduction :

$$\frac{}{n_1 \oplus n_2 \to m} \quad \text{avec } m = n_1 \oplus_{\mathbb{N}} n_2$$

  qui transforme le terme en un terme "plus proche" du résultat final.

Relation entre sémantique à grands pas et à petits pas

- Soit la fermeture transitive et réflexive "$\rightarrow^\star$" de la relation "$\rightarrow$".
- Montrons que :

$$\forall e\ n\ , (e \rightarrow^\star n) \Leftrightarrow (e \Downarrow n)$$

Sémantique dénotationnelle

- Il existe des sémantiques dénotationnelles qui interprètent les termes par des objets mathématiques.
- Typiquemement, pour les expressions arithmétiques, on peut interpréter les termes dans $\mathbb{N}$. Ainsi, l'interprétation $(\!|\ e\ |\!)$ se définit ainsi :

$$
\begin{array}{rcl}
(\!|\ n\ |\!) & = & n \\
(\!|\ e_1 \oplus e_2\ |\!) & = & (\!|\ e_1\ |\!) \oplus_{\mathbb{N}} (\!|\ e_2\ |\!)
\end{array}
$$

- Dans ce cadre, les propriétés sur les termes découlent alors immédiatement des propriétés mathématiques du domaine d'interprétation.
- Cependant, même si le cas des expressions arithmétiques est très simple, quels objets mathématiques seraient de "bonnes" interprétations des programmes Caml ou Java ?

$\Rightarrow$ Nous n'utiliserons pas cette forme de sémantique dans ce cours car ils nécessitent des outils mathématiques hors de notre portée.

Application : sémantique pour un langage avec variables

Exemple : expressions arithmétiques avec un **let**

$$
\begin{array}{rcll}
e & ::= & n & \\
  & \mid & x & (1) \\
  & \mid & e + e & \\
  & \mid & e * e & \\
  & \mid & \textbf{let } x := e \textbf{ in } e & (2)
\end{array}
$$

- Le langage des expressions arithmétiques avec deux nouvelles constructions :
  (1) Une variable x qui **fait référence à un résultat intermédiaire du calcul**.
  (2) Une introduction de variable qui **nomme un résultat intermédiaire**.

Variables libres d'une expression

- Quel sens donné à l'expression « **let** $x := 21$ **in** $x + x$ » ?

⇒ Les références à x peuvent être substituées par 21 dans l'expression « $x + x$ ».

- Quel sens donné à l'expression « x » ?

⇒ Cette occurence de x est libre : le contexte sous lequel elle est apparaît ne permet pas de déterminer sa valeur.

⇒ En d'autres termes, la variable x n'est pas liée dans cette seconde expression alors que dans la première expression, elle était liée par le **let**.

- On dit aussi qu'une variable liée est muette. En effet, on peut la renommer sans changer le sens de l'expression.

Exercice

Sauriez-vous définir le jugement :

« Au moins une occurrence de x est libre dans e »

?

Variables libres d'une expression

$$\frac{}{x \in \mathrm{FV}(x)} \qquad \frac{x \in \mathrm{FV}(e_1)}{x \in \mathrm{FV}(e_1 \oplus e_2)} \qquad \frac{x \in \mathrm{FV}(e_2)}{x \in \mathrm{FV}(e_1 \oplus e_2)}$$

$$\frac{x \in \mathrm{FV}(e_1)}{x \in \mathrm{FV}(\textbf{let } y := e_1 \textbf{ in } e_2)} \qquad \frac{x \in \mathrm{FV}(e_2) \qquad x \neq y}{x \in \mathrm{FV}(\textbf{let } y := e_1 \textbf{ in } e_2)}$$

Calcul des variables libres d'une expression

```
type binop = Add | Mul | Div | Sub
type variable = string
type e =
  | Int of int
  | Binop of binop × e × e
  | Var of variable
  | Let of variable × e × e

module SSet = Set.Make (String)

let rec fv :  e → SSet.t = function
  | Var x → SSet.singleton x
  | Binop (_, e1, e2) → SSet.union (fv e1) (fv e2)
  | Let (x, e1, e2) → SSet.union (fv e1) (SSet.remove x (fv e2))
  | _ → SSet.empty
```

Expression close

- Une expression e est close si il n'existe pas de variable libre dans e.
- Intuitivement : si une expression e est close, on peut l'évaluer en un entier.

Substitution

- On veut définir le **mécanisme calculatoire/de réduction** correspondant à la substitution d'une variable x par sa valeur calculée par le membre gauche du **let** qui l'introduit.
- Intuitivement : si l'expression à réduire est de la forme « **let** $x := n$ **in** e», on doit remplacer toutes les occurences de x par n dans e.

Exercice

Sauriez-vous définir le jugement « e' est e dans lequel x est substituée par n » ?

Substitution

$$\begin{array}{rcll}
(n)\{x/n'\} & = & n & \\
(x)\{x/n\} & = & n & \\
(x)\{y/n\} & = & y & \text{si } x \neq y \\
(e_1 \oplus e_2)\{x/n\} & = & (e_1)\{x/n\} \oplus (e_2)\{x/n\} & \\
(\mathbf{let}\ y := e_1\ \mathbf{in}\ e_2)\{x/n\} & = & \mathbf{let}\ y := (e_1)\{x/n\}\ \mathbf{in}\ (e_2)\{x/n\} &
\end{array}$$

$\Rightarrow$ Est-ce correct ?

- Non ! Contre-exemple :

$$(\mathbf{let}\ x := x + 1\ \mathbf{in}\ x * 2)\{x/1\} \;=\; \mathbf{let}\ x := 1 + 1\ \mathbf{in}\ 1 * 2$$

- Nous aimerions plutôt obtenir :

$$(\mathbf{let}\ x := x + 1\ \mathbf{in}\ x * 2)\{x/1\} \;=\; \mathbf{let}\ x := 1 + 1\ \mathbf{in}\ x * 2$$

Substitution, définition corrigée

$$
\begin{array}{rcll}
(n)\{x/n'\} & = & n & \\
(x)\{x/n\} & = & n & \\
(x)\{y/n\} & = & y & \text{si } x \neq y \\
(e_1 \oplus e_2)\{x/n\} & = & (e_1)\{x/n\} \oplus (e_2)\{x/n\} & \\
(\textbf{let } y := e_1 \textbf{ in } e_2)\{x/n\} & = & \textbf{let } y := (e_1)\{x/n\} \textbf{ in } (e_2)\{x/n\} & \text{si } x \neq y \\
(\textbf{let } x := e_1 \textbf{ in } e_2)\{x/n\} & = & \textbf{let } x := (e_1)\{x/n\} \textbf{ in } e_2 &
\end{array}
$$

Exercice

Montrer que : si dans un terme e, tous les **let** introduisent des noms de variables distincts alors, dans ce cas, la première définition de la substitution est équivalente à cette version corrigée.

Calcul de la substitution

```
let rec subst x n = function
  | Var y when x = y → Int n
  | Binop (op, e1, e2) → Binop (op, subst x n e1, subst x n e2)
  | Let (y, e1, e2) when x = y → Let (y, subst x n e1, e2)
  | Let (y, e1, e2) → Let (y, subst x n e1, subst x n e2)
  | x → x
```

Sémantique à petits pas

$$\frac{e_1 \to e_1'}{e_1 \oplus e_2 \to e_1' \oplus e_2} \qquad \frac{e_2 \to e_2'}{n_1 \oplus e_2 \to n_1 \oplus e_2'}$$

$$\frac{}{n_1 \oplus n_2 \to m} \quad \text{avec } m = n_1 \oplus_{\mathbb{N}} n_2$$

$$\frac{e_1 \to e_1'}{\textbf{let } x := e_1 \textbf{ in } e_2 \to \textbf{let } x := e_1' \textbf{ in } e_2}$$

$$\frac{}{\textbf{let } x := n \textbf{ in } e_2 \to (e_2)\{x/n\}}$$

- Quelles sont les règles de passage au contexte ? Les règles de réduction ?

Évaluation des expressions closes

- On montre que :

$$\forall e, e \text{ close } \Rightarrow \exists n, e \rightarrow^\star n$$

(Par induction sur les termes.)

Écriture d'un interprète de la sémantique à petits pas

```
let rec eval_step : e → e = function
  (** Impossible cases. *)
  | (Int n as e) → assert (not (is_value e)); exit 1 (* By precondition. *)
  | (Var x as e) → assert (is_closed e); exit 1 (* By precondition. *)

  (** Cases. *)
  | Binop (op, Int n1, Int n2) → Int (eval_binop op n1 n2)
  | Binop (op, Int n, e) → Binop (op, Int n, eval_step e)
  | Binop (op, e1, e2) → Binop (op, eval_step e1, e2)
  | Let (x, Int n, e) → subst x n e
  | Let (x, e1, e2) → Let (x, eval_step e1, e2)
```

Critique de cet interprète

- La substitution effectue un parcours en profondeur du terme pour trouver toutes les occurrences de la variable à substituer.
- Une substitution a lieu en chaque nœud **let** de l'arbre de syntaxe.
- La complexité en pire cas de cet interprète est donc **quadratique**.

$\Rightarrow$ Peut-on faire mieux ?

Tentative pour une sémantique à grands pas

$$\frac{e_1 \Downarrow n_1 \qquad e_2 \Downarrow n_2}{e_1 \oplus e_2 \Downarrow n_1 \oplus_{\mathbb{N}} n_2} \qquad \frac{}{n \Downarrow n} \qquad \frac{e_1 \Downarrow n_1 \qquad (e_2)\{x/n_1\} \Downarrow n_2}{\textbf{let } x := e_1 \textbf{ in } e_2 \Downarrow n_2}$$

$$\frac{}{x \Downarrow \ ?}$$

- Si le terme e est supposé clos alors la règle des variables n'est pas nécessaire.

Interprète de la sémantique à grands pas (pour les expressions closes)

```
let rec eval : e → int = function
  (** Impossible cases. *)
  | (Var x as e) → assert (is_closed e); exit 1 (* By precondition. *)

  (** Cases. *)
  | Int n → n
  | Binop (op, e1, e2) → eval_binop op (eval e1) (eval e2)
  | Let (x, e1, e2) → eval (subst x (eval e1) e2)
```

- Ce programme est plus concis et sensiblement plus efficace que l'interprète à petits pas mais la substitution subsiste . . . donc la complexité reste $O(n^2)$.

Nouvelle tentative de sémantique à grands pas

$$\frac{e_1 \Downarrow n_1 \qquad e_2 \Downarrow n_2}{e_1 \oplus e_2 \Downarrow n_1 \oplus_{\mathbb{N}} n_2} \qquad \frac{}{n \Downarrow n} \qquad \frac{e_1 \Downarrow n_1 \qquad (e_2)\{x/n_1\} \Downarrow n_2}{\mathbf{let}\ x := e_1\ \mathbf{in}\ e_2 \Downarrow n_2}$$

$$\frac{}{x \Downarrow \color{red}{?}}$$

- On peut relâcher la contrainte « e clos » à condition de **se souvenir** de la valeur des variables libres.
- Ainsi, plutôt que de calculer la substitution $(e_2)\{x/n_1\}$, on continue l'évaluation avec de e_2 en notant “quelque part” que $x = n$ 1.
- Si on rencontre, au cours de l'évaluation de e_2, une occurrence de x, on renvoie le résultat n_1.

Environnement (lexical)

- La structure permettant de se souvenir de l'association entre variable et valeur entière est appelée un environnement.
- C'est moralement une liste d'associations. On peut se donner la syntaxe :

$$\begin{array}{rcl} \Gamma & ::= & \bullet \\ & \mid & \Gamma ; (x \mapsto n) \end{array}$$

- On se donne un jugement « $\Gamma(x) = n$ » défini ainsi :

$$\frac{}{(\Gamma ; (x \mapsto n))(x) = n} \qquad \frac{\Gamma(x) = n \qquad x \neq y}{(\Gamma ; (y \mapsto n'))(x) = n}$$

Exercice

Peut-on dériver le jugement « $(\bullet ; (x \mapsto 21) ; (x \mapsto 42))(x) = 21$ » ?

Implémentation du type abstrait des environnements

```
module Env : sig
  type t
  val bind : t → variable → int → t
  val empty : t
  val lookup : t → variable → int
end = struct
  type t = (variable × int) list
  let empty = []
  let bind e x n = (x, n) :: e
  let lookup e x = List.assoc x e
end
```

Sémantique à grands pas et à environnement

$$\frac{\Gamma \vdash e_1 \Downarrow n_1 \qquad \Gamma \vdash e_2 \Downarrow n_2}{\Gamma \vdash e_1 \oplus e_2 \Downarrow n_1 \oplus_{\mathbb{N}} n_2} \qquad \frac{}{\Gamma \vdash n \Downarrow n}$$

$$\frac{\Gamma \vdash e_1 \Downarrow n_1 \qquad \Gamma; (x \mapsto n_1) \vdash e_2 \Downarrow n_2}{\Gamma \vdash \textbf{let } x := e_1 \textbf{ in } e_2 \Downarrow n_2} \qquad \frac{\Gamma(x) = n}{\Gamma \vdash x \Downarrow n}$$

Évaluation

- À quelles conditions sur l'environnement Γ et le terme e existe-t-il un entier n ainsi qu'une dérivation de « $\Gamma \vdash e \Downarrow n$ » ?

$\Rightarrow$ Réponse lors du prochain cours.

Interprète de la sémantique à grands pas

```
let rec eval : Env.t → e → int =
  fun env → function
    | Var x → Env.lookup env x
    | Int n → n
    | Binop (op, e1, e2) → eval_binop op (eval env e1) (eval env e2)
    | Let (x, e1, e2) → eval (Env.bind env x (eval env e1)) e2

let eval : e → int = eval Env.empty
```

- Grâce à l'environnement, le coût de l'évaluation d'un **let** est constant.
- Par contre, l'évaluation d'une variable nécessite un parcours de l'environnement, dont la taille est bornée par la hauteur de l'arbre de syntaxe.

⇒ C'est mieux mais peut-on aller plus loin ?

Une implémentation rusée des variables

- On modifie la syntaxe des termes :

$$\begin{array}{rcll} e & ::= & n & \\ & | & \hat{\iota} & (1) \\ & | & e + e & \\ & | & e * e & \\ & | & \textbf{let } \iota := e \textbf{ in } e & (2) \end{array}$$

(1) On utilise des indices (des entiers) pour représenter les variables.

(2) Les **let**s sont numérotés : leur indice correspond **au nombre de lets rencontrés depuis la racine** du terme.

⇒ L'indice d'une variable est l'indice du **let** qui l'a introduite.

- Ce sont des indices de De Bruijn.

Exemple

- Le terme de la grammaire initiale :

$$\textbf{let } x := (\textbf{let } y := 21 \textbf{ in } y) \textbf{ in let } z := 20 \textbf{ in } z + x + 1$$

  s'écrit dans ce nouveau langage :

$$\textbf{let } 0 := (\textbf{let } 1 := 21 \textbf{ in } \hat{1}) \textbf{ in let } 1 := 20 \textbf{ in } \hat{1} + \hat{0} + 1$$

  (L'indice qui suit le **let** n'est pas essentiel puisqu'on peut le calculer facilement.)

Exercice

Sauriez-vous écrire une fonction qui traduit un terme du langage initial vers ce nouveau langage ?

Implémentation de l'environnement

- On peut implémenter l'environnement à l'aide d'un tableau (borné ou extensible) qui suit une discipline de pile :

```
module Env : sig
  type t
  val bind : t → variable → int → t
  val empty : t
  val lookup : t → variable → int
end = struct
  type t = int × int array
  let max_env_size = 42
  let empty = (0, Array.create max_env_size 0)
  let bind (top, stack) x n =
    assert (top = x);
    stack.(top) ← n;
    (top + 1, stack)
  let lookup (_, stack) x = stack.(x)
end
```

Interprète de la sémantique à grands pas

```
let rec eval : int → Env.t → e → int =
  fun depth env → function
    | Var x →
        Env.lookup env x
    | Int n →
        n
    | Binop (op, e1, e2) →
        eval_binop op (eval depth env e1) (eval depth env e2)
    | Let (e1, e2) →
        eval (depth + 1) (Env.bind env depth (eval depth env e1)) e2

let eval : e → int = eval 0 Env.empty
```

- L'accès aux valeurs des variables se fait en temps constant.
- Cet interprète est plutôt efficace !

  (On peut encore l'améliorer : la variable “depth” ne sert à rien. Pourquoi ?)

Synthèse

Synthèse

- Définition de relations et prédicats inductifs sur les termes d'un langage.
- Explicitation des sémantiques à grands pas et à petits pas.
- Preuve de propriétés sur ces sémantiques.
- Un premier mécanisme : l'accès à une variable.

$\Rightarrow$ Comment garantir que tous les accès aux variables seront valides ?

$\Rightarrow$ Comment étendre ce mécanisme aux appels de fonctions ?