# Fondements de la programmation

Partiel 1, 31/11/2022

**Consignes :**
- Matériel de cours autorisé.
- Examen d'une durée de 3h.
- Dans les inductions :
  - indiquez la liste des cas à traiter et indiquant les cas impossibles,
  - traitez au moins 3 cas possibles,
  - s'il a un cas qui fait intervenir un lieur, traitez en un (parmi les 3 ci-dessus),
  - s'il a un cas qui fait intervenir une union, traitez en un (parmi les 3 ci-dessus),
  - s'il a un cas qui fait intervenir la règle (sous-prog.), traitez le (parmi les 3 ci-dessus).
- Si vous ne savez pas faire quelque chose :
  - admettez le explicitement et continuez
  - prétendre avoir prouvé une partie sans aucun raisonnement (ou un raisonnement absurde) sera sanctionné, il faut que vous soyez vous même convaincu par la preuve.

## 1 Spécification d'un fragment (inspiré) de JavaScript

Il s'agit ici de travailler avec un JavaScript typé, simplifié et modifié. [1]
Nous utiliserons le langage :

### Type

| | | |
|---|---|---|
| $\tau ::=$ | `Num` | (tyNombres) |
| | `Str` | (tyStrings) |
| | $\tau_1 \to \tau_2$ | (tyFonctions) |
| | $\tau_1 \cup \tau_2$ | (Union) |
| | `Undefined` | (tyUndefined) |

### Programme

| | | |
|---|---|---|
| $r ::=$ | $\epsilon$ | (prog vide) |
| | $c\ r$ | (commande) |

### Commande

| | | |
|---|---|---|
| $c ::=$ | $e;$ | (expression) |
| | `let` $x = e;$ | (definition) |
| | `if` $(e)\ c_1$ `else` $c_2$ | (ifthenelse) |

### Expression

| | | |
|---|---|---|
| $e ::=$ | $x$ | (variable) |
| | $n$ | (nombre littéral) |
| | $s$ | (texte littéral) |
| | $e_1 + e_2$ | (addition) |
| | $\{r\}$ | (sous-prog.) |
| | `undefined` | (undefined) |
| | `toStr` | (tostr) |
| | $e_1(e_2)$ | (application) |
| | $x \Rightarrow e$ | (abstraction) |

### Valeur

| | | |
|---|---|---|
| $v ::=$ | $n$ | (valNombre) |
| | $s$ | (valString) |
| | $x \Rightarrow e$ | (valFonction) |
| | `undefined` | (valUndef) |
| | `toStr` | (valTostr) |

### Contexte

$\Gamma ::= x_1 : \tau_1, \ldots, x_n : \tau_n$

**Priorité** :

(commande) < (ifthenelse)
< (abstraction)
< (addition)
< (application)
< (TyFonctions)
< (Union)

**Associativité gauche** :
application, union, addition,

**Associativité droite** :
tyFonction, commande

**Lieurs** :
definition : $x$ dans les commandes suivantes
abstraction : $x$ dans $e$

1. Voir la section "À lire après l'examen" si vous êtes intéressé par les différences avec le vrais JS.

où les tokens $x \in [a-zA-Z\_][a-zA-Z0-9\_]^*$, $n \in [0-9]^+$, et $s \in ["][\hat{}"]^*["]$ sont respectivement des variables de termes, entiers, et chaînes de caractères de OCaml.

## 1.1 Types

### Règles de typage des programmes

$$\frac{}{\Gamma \vdash} \ (\vdash \epsilon) \qquad \frac{\Gamma \vdash c \dashv \Gamma' \qquad \Gamma' \vdash r}{\Gamma \vdash c\ r} \ (\vdash ;;)$$

### Règles de typage des commandes

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash e; \dashv \Gamma} \ (\vdash \text{commande}) \qquad \frac{\Gamma, x : \tau \vdash e : \tau}{\Gamma \vdash \texttt{let}\ x = e \dashv\ \Gamma, x : \tau} \ (\vdash \text{def.rec.})$$

$$\frac{\Gamma \vdash e : \texttt{Num} \qquad \Gamma \vdash c_1 \dashv \Gamma' \qquad \Gamma \vdash c_2 \dashv \Gamma'}{\Gamma \vdash \texttt{if}\ (e)\ c_1\ \texttt{else}\ c_2 \dashv \Gamma'} \ (\vdash \text{ifthenelse})$$

### Règles de typage des expressions

$$\frac{}{\Gamma, x : \tau \vdash x : \tau} \ (\vdash \text{id}) \qquad \frac{}{\Gamma \vdash n : \texttt{Num}} \ (\vdash \text{num}) \qquad \frac{}{\Gamma \vdash s : \texttt{Str}} \ (\text{str})$$

$$\frac{\Gamma \vdash e_1 : \texttt{Num} \qquad \Gamma \vdash e_2 : \texttt{Num}}{\Gamma \vdash e_1 + e_2 : \texttt{Num}} \ (\vdash \text{n+n}) \qquad \frac{\Gamma \vdash e_1 : \texttt{Num} \cup \texttt{Str} \qquad \Gamma \vdash e_2 : \texttt{Num} \cup \texttt{Str}}{\Gamma \vdash e_1 + e_2 : \texttt{Num} \cup \texttt{Str}} \ (\vdash \text{a+a})$$

$$\frac{\Gamma \vdash e_1 : \texttt{Str} \qquad \Gamma \vdash e_2 : \texttt{Num} \cup \texttt{Str}}{\Gamma \vdash e_1 + e_2 : \texttt{Str}} \ (\vdash \text{s+a}) \qquad \frac{\Gamma \vdash e_1 : \texttt{Num} \cup \texttt{Str} \qquad \Gamma \vdash e_2 : \texttt{Str}}{\Gamma \vdash e_1 + e_2 : \texttt{Str}} \ (\vdash \text{a+s})$$

$$\frac{\Gamma \vdash r}{\Gamma \vdash \{r\} : \texttt{Undefined}} \ (\vdash \text{s.p.}) \qquad \frac{}{\Gamma \vdash \texttt{undefined} : \texttt{Undefined}} \ (\vdash \text{undef.}) \qquad \frac{}{\Gamma \vdash \texttt{toStr} : \texttt{Num} \to \texttt{Str}} \ (\vdash \text{tostr})$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1(e_2) : \tau_2} \ (@) \qquad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash x \Rightarrow e : \tau_1 \to \tau_2} \ (\lambda)$$

$$\frac{\Gamma \vdash e : \tau \qquad \tau \subseteq \tau'}{\Gamma \vdash e : \tau'} \ (\vdash \cup)$$

### Règles de sous-typage

$$\frac{}{\tau \subseteq \tau} \ (\subseteq \text{id}) \qquad \frac{}{\tau \subseteq \texttt{Undefined}} \ (\subseteq \text{undef}) \qquad \frac{}{\tau \subseteq \tau \cup \tau'} \ (\subseteq \cup 1) \qquad \frac{}{\tau \subseteq \tau' \cup \tau} \ (\subseteq \cup 2)$$

$$\frac{\tau_1 \subseteq \tau_2 \qquad \tau_2 \subseteq \tau_3}{\tau_1 \subseteq \tau_3} \ (\subseteq tr) \qquad \frac{\tau_1 \subseteq \tau_3 \qquad \tau_2 \subseteq \tau_3}{\tau_1 \cup \tau_2 \subseteq \tau_3} \ (\subseteq \cup 3) \qquad \frac{\tau_1' \subseteq \tau_1 \qquad \tau_2 \subseteq \tau_2'}{\tau_1 \to \tau_2 \subseteq \tau_1' \to \tau_2'} \ (\subseteq \to)$$

## 1.2 Système de transition

### Règles de transition des programmes/commandes

$$\frac{e \mapsto e'}{e;\ r \mapsto e';\ r} \ (\text{exprG}) \qquad \frac{v \text{ valeur}}{v;\ r \mapsto r} \ (\text{exprD}) \qquad \frac{e \mapsto e'}{\texttt{let}\ x{=}e;\ r \mapsto \texttt{let}\ x{=}e';\ r} \ (\text{letG})$$

$$\frac{v \text{ valeur}}{\texttt{let}\ x{=}v;\ r \mapsto r[v/x]} \ (\text{letD}) \qquad \frac{e \mapsto e'}{\texttt{if}\ (e)\ c_1\ \texttt{else}\ c_2\ r \mapsto \texttt{if}\ (e')\ c_1\ \texttt{else}\ c_2\ r} \ (\text{ifG})$$

$$\frac{}{\texttt{if}\ (0)\ c_1\ \texttt{else}\ c_2\ r \mapsto c_2\ r} \ (\text{ifD1}) \qquad \frac{n \neq 0}{\texttt{if}\ (n)\ c_1\ \texttt{else}\ c_2\ r \mapsto c_1\ r} \ (\text{ifD2})$$

**Règles de transition des expressions**

$$\frac{e_1 \mapsto e_1'}{e_1 + e_2 \mapsto e_1' + e_2}\ (+\text{G}) \qquad \frac{e_2 \mapsto e_2'}{v_1 + e_2 \mapsto v_1 + e_2'}\ (+\text{D}) \qquad \frac{n_1 + n_2 = n}{n_1 + n_2 \mapsto n}\ (+\text{num})$$

$$\frac{}{s_1 + s_2 \mapsto s_1 s_2}\ (+\text{str}) \qquad \frac{}{n_1 + s_2 \mapsto \texttt{tostr}(n_1) + s_2}\ (+\text{str}) \qquad \frac{}{s_1 + n_2 \mapsto s_1 + \texttt{tostr}(n_2)}\ (+\text{str})$$

$$\frac{e_1 \mapsto e_1'}{e_1(e_2) \mapsto e_1'(e_2)}\ (\text{appG}) \qquad \frac{e_2 \mapsto e_2'}{e_1(e_2) \mapsto e_1(e_2')}\ (\text{appD}) \qquad \frac{}{(x \Rightarrow e_1)(v_2) \mapsto e_1[v_2/x]}\ (\text{beta})$$

$$\frac{s \text{ est l'écriture décimale de } n}{\texttt{toStr}(n) \mapsto s}\ (\mapsto\text{tostr}) \qquad \frac{r \mapsto r'}{\{r\} \mapsto \{r'\}}\ (\mapsto\text{rG}) \qquad \frac{}{\{\} \mapsto \texttt{undefined}}\ (\mapsto\text{rD})$$

où $s_1 s_2$ représente la concaténation des chaînes de caractères $s_1$ et $s_2$ et où la représentation décimal de 42 est la chaîne "42".

## 2 À lire après l'examen

En plus de la syntaxe réduite, on considère les variables immutables, c'est à dire que l'on s'interdit de pouvoir écrire `x = 3;` hors d'un `let`. Cela nous permet de parler de substitution, car la sémantique opérationnel définie par substitution n'est pas valable pour les structures mutable (il faut conserver le lien entre les différentes instances). Pour ça on préférera une sémantique par machine abstraite/virtuelle, telle celle que sera vu en compilation au prochain semestre.

L'expression composé d'un programme n'existe pas non plus : il s'agit d'une structure que certaines implémentations de JS (au moins celles de Mozilla et Chrome) acceptent sur des cas simples mais le comportement est indéfini. Par contre, on a bien des commandes formées de programme, ainsi en JS réel, `{let x = 3; let y = 25;}` est une seule commande car il y a les accolades.

La commande (definition) est en fait une expression et se lie après avoir été hissée (voire le hissage/hoisting dans le cours de compile au prochain semestre) au niveau de bloque courant, ce qui est un peu étrange à manipuler, et interdit, notamment, d'avoir plusieurs `let` dans le même bloque, comme en question A.3.

À noter que le typage de JS n'est pas statique et que les versions typés de JS (il en existes plusieurs) vont soit utiliser des typages plus stricte (à la OCaml, sans types union), soit des typages plus lourd (types unions+intersection+complément+gradualité). Il y a énormément de recherche sur ces sujets, car l'absence de type statique pour les gros programmes JS (notamment en backend) est un énorme frein technologique (moins efficace et plus de bugs).

# 3 Questions

## 3.1 Grammaire

**Question A. Écrire les ABTs des types et programmes suivants :.**

1. $\texttt{Num} \to \texttt{Str} \cup \texttt{Num} \cup (\texttt{Num} \cup \texttt{Str} \to \texttt{Str}) \to \texttt{Str} \cup \texttt{Num}$
2. `let toto=42; let foo = b=>c=>c+b; if (toto) {foo(toto)("toto");foo(3);}; else {};`
3. `let x = x => x+x; let x = x(3)+2; let y = (x => x+x)(x)+x+y;`

**Question B. alpha-équivalence.** Écrire une forme alpha-équivalente du programme 3. ci-dessus la plus général possible (chaque lieur lie une variable différente).

**Question C. (dificile) Le ';' en JS.** Le point-virgule de JS n'est pas toujours traité ainsi, dans certaines implémentations de JS, les commandes (expression) et (def.rec.) n'ont pas de ';' à la fin, à la place il y a une commande (pointvirg) :

$$\begin{array}{rll} c ::= & e & \text{(expression)} \\ & \texttt{let}\ x = e & \text{(definition)} \\ & c\ ; & \text{(pointvirg)} \\ & \dots & \end{array}$$

Un tel changement dans la grammaire ajouterait-il des programmes reconnus ? en retirerait-il ? Pour chaque question argumentez en une ou deux lignes.

## 3.2 Types

Vous pouvez admettre le lemme d'affaiblissement dans les preuves et dérivations suivantes.

**Question D. Montrer sans faire d'induction que pour tout types $\tau_1$, $\tau_2$, on a.**

$$\frac{}{\tau_1 \cup \tau_2 \subseteq \tau_2 \cup \tau_1}$$

**Question E. Trouver une dérivation de type pour chacun des séquents suivants :.**

- $f : \texttt{Num} \cup \texttt{Str} \to \texttt{Num} \vdash \texttt{f(42)} + \texttt{"test"} : \texttt{Str}$
- $f : \texttt{Num} \cup \texttt{Str} \to \texttt{Num} \vdash f : \texttt{Num} \to \texttt{Num} \cup \texttt{Str}$
- $x : \texttt{Num} \vdash \texttt{if (x) let y = x; else let y = "test";} \dashv x : \texttt{Num}, y : \texttt{Num} \cup \texttt{Str}$

**Question F. (difficile) Prouvez le lemme d'extensionalité, qui prend deux formes :.**

$$\frac{x \notin \texttt{dom}(\Gamma) \qquad \Gamma \vdash x \Rightarrow e(x) : \tau_1 \to \tau_2}{\Gamma \vdash e : \tau_1 \to \tau_2}\ (\eta 1) \qquad \frac{x \notin \texttt{VL}(e) \qquad \Gamma, x : \tau_1 \vdash e(x) : \tau_2}{\Gamma \vdash e : \tau_1 \to \tau_2}\ (\eta 2)$$

**Question G. Type de base.** Prouvez qu'une expression composée d'entier, de sting et de sommes est toujours du type $\texttt{Num} \cup \texttt{Str}$

## 3.3 Transitions

**Question H. Réduisez les termes suivants jusqu'à obtenir une valeur ou un programme vide.**

- `21+21+"test"+21+21`
- `let x = x => x+x; (y => x(y))(3);`

**Question I. Sûreté (dificile).** Prouvez :

- qu'un programme bien typé a toujours une transition à moins d'être un programme vide,
- qu'une expression bien typée a toujours une transition à moins d'être une valeur.