Examen du cours "INTRODUCTION À LA COMPILATION"

Licence 3 – Université Paris Diderot Paris 7

Durée: 3 heures

Tout document non manuscrit autorisé.

Le soin apporté à la rédaction et à la présentation ainsi que la rigueur des réponses seront pris en compte dans la notation. Ce sujet se décompose en 2 parties indépendantes. La première est une application directe du cours d'analyse syntaxique. La seconde porte sur une extension du langage de programmation étudié cette année. Pour ce faire, on rappelle la syntaxe abstraite :

e	::=		***Expression***
		$x, f, \ldots$	*Variable*
		$\mathtt{c}$	*Constante* $\mathtt{c} \in \mathbb{C}$
		$e\ e$	*Application*
		$\mathtt{fun}\ x \Rightarrow e$	*Fonction*
a, v	::=		***Valeurs***
		x, f	*Variable*
		$\mathtt{c}$	*Constante* $\mathtt{c} \in \mathbb{C}$
		$\mathtt{fun}\ x \Rightarrow e$	*Fonction*

t $x = e_1$ in e_2" est autorisée. On écrira aussi "let $f\ x_1 \ldots x_n = e_1$ in e_2" pour l'expression "let $f = \mathtt{fun}\ x_1 \Rightarrow \ldots \mathtt{fun}\ x_n \Rightarrow e_1$ in e_2". L'ensemble des constantes $\mathbb{C}$ est formé des booléens, du branchement conditionnel et de l'opérateur de point fixe.

L'utilisation du sucre syntaxique "le e_2 in e_2" pour l'expression "let $f =$ fu des entiers, des opérations arithmétiques

1 Analyse syntaxique

Soit la grammaire suivante :

$$
\begin{array}{lcl}
S & \rightarrow & P\,\# \\
P & \rightarrow & \epsilon \mid P\,I \\
I & \rightarrow & l\,a\,O \\
O & \rightarrow & \epsilon \mid l
\end{array}
$$

où :

- S, P, I et O sont des symboles non-
- l et a sont des symboles terminaux.

terminaux ;

Exercice 1 7/5

1. *Est-ce que cette grammaire est ada*
2. *Transformez cette grammaire en un*
3. *Calculez la fonction FIRST.*
4. *Calculez la fonction FOLLOW.*
5. *Pourquoi cette grammaire n'est pas*
6. *Proposez un algorithme d'analyse s*

ptée à l'analyse syntaxique LL ? Pourquoi ?

e grammaire équivalente de façon à l'adapter à l'analyse LL.

LL(1) ?

ntaxique pour cette grammaire.

□

2 Erreurs et exceptions

On étend la grammaire des termes du langage étudié en cours en rajoutant deux nouvelles constructions :

$$
\begin{array}{rcll}
e & ::= & \ldots & \\
 & | & \texttt{error} & \textit{Déclenchement d'une erreur} \\
 & | & \texttt{try}\ e\ \texttt{catch}\ e & \textit{Gestionnaire d'erreur}
\end{array}
$$

La première construction "`error`" représente le passage dans une situation anormale du calcul (par exemple, lors d'une division par zéro ou lorsqu'une précondition d'une fonction n'est pas respectée). La seconde construction "`try` e_1 `catch` e_2" évalue une expression e_1 et rattrape une éventuelle erreur issue de cette évaluation à l'aide d'un gestionnaire d'erreur e_2. Une fois l'erreur traitée, le calcul reprend son cours normal.

Exercice 2 (Sémantique opérationnelle des erreurs) *On donne la sémantique à petits pas suivante du λ-calcul en appel par valeur avec gestionnaires d'erreurs :*

$$\beta\text{-REDUCTION} \qquad (\texttt{fun}\ x \Rightarrow e)\ v \to e\{x \mapsto v\}$$

$$\textsc{App-G} \quad \frac{t_1 \to t_2}{v\ t_1 \to v\ t_2} \qquad\qquad \textsc{App-D} \quad \frac{t_1 \to t_2}{t_1\ t \to t_2\ t}$$

$$\textsc{Erreur-G} \quad \texttt{error}\ t \to \texttt{error} \qquad\qquad \textsc{Erreur-D} \quad t\ \texttt{error} \to \texttt{error}$$

$$\textsc{Try-OK} \quad \texttt{try}\ v\ \texttt{catch}\ t \to v \qquad \textsc{Try-Echec} \quad \texttt{try error catch}\ t \to t \qquad \textsc{Try-Sous-Terme} \quad \frac{t_1 \to t_2}{\texttt{try}\ t_1\ \texttt{catch}\ t \to t_2}$$

1. *Parmi ces règles, quelles sont les règles de passage au contexte et les règles de réduction ?*
2. *En quoi ces règles correspondent-elles à une stratégie d'appel par valeur ?*
3. *À quoi servent les règles* Erreur-G *et* Erreur-D *?*
4. *Donnez les règles de réduction, de passage au contexte et de propagation des erreurs pour les expressions de la forme "*`if` e_1 `then` e_2 `else` e_3*".*
5. *Donnez la dérivation d'évaluation du programme suivant, en précisant pour chaque étape, la règle de sémantique utilisée :*

```
let div x =
  try
    if x = 0 then error else 10/x
  catch 0
in
  div 0
```

6. *Quels sont les termes bloqués de ce langage de programmation ?*

□

Exercice 3 (Sémantique opérationnelle des exceptions simples) *On veut maintenant associer une valeur à une exception de façon à transporter de l'information sur le contexte qui a déclenché l'exception jusqu'au gestionnaire d'exception. La construction* `error` *est donc remplacée par une construction plus générale :*

e	::=	...	
	|	`raise` e	*Déclenchement d'une exception*
	|	`try` e `with` e	*Gestionnaire d'exception*

La construction "`raise` e*" évalue e en une valeur v et lance une erreur à laquelle est associée cette valeur v. Le gestionnaire d'exception "*`try` e_1 `with` e_2*" évalue e_1 et si cette dernière se réécrit en* `raise` v*, elle évalue "$e_2\ v$".*

1. *En vous inspirant de la sémantique à petits pas vue en cours, spécifiez une sémantique à petits pas pour ce nouveau langage.*
2. *Donnez la dérivation d'évaluation du programme suivant, en précisant pour chaque étape, la règle de sémantique que vous avez utilisée :*

```
let div = fun x →
  try
    if x = 0 then raise (x + 1) else 10/x
  with exn → exn
in
div 0
```

3. *Même question pour le programme suivant :*

```
let div = fun x →
  if x = 0 then raise (x + 1) else 10/x
in
raise (div 0)
```

4. *Quels sont les termes bloqués de ce nouveau langage de programmation ?*

□

Exercice 4 (Sémantique à petits pas des exceptions structurées) *Il n'y a pas toujours un seul type d'erreur dans un programme. On souhaite donc autoriser plusieurs types d'erreur à être déclenchables et un gestionnaire d'erreur à ne traiter qu'un sous-ensemble de ces erreurs. Pour ces raisons, un ensemble $\mathbb{E}$ est fixé et ses éléments $E \in \mathbb{E}$ sont les noms d'exceptions utilisables. La nouvelle syntaxe des constructions du langage dédiées aux exceptions est donc :*

e	::=	...	
	|	`raise` $(E\ e)$	*Déclenchement d'une exception*
	|	`try` e `with` $E_1 \to e_1 \mid \ldots \mid E_n \to e_n$	*Gestionnaire d'exceptions*

La construction "`raise` $(E\ e)$*" déclenche une exception nommée E à laquelle est associée la valeur issue de l'évaluation de e. La construction "*`try` e `with` $E_1 \to e_1 \mid \ldots \mid E_n \to e_n$*" évalue e et si de cette évaluation résulte une exception non rattrapée $E\ (v)$ alors :*

- *s'il existe un $i \in [\![1..n]\!]$ tel que $E = E_i$ alors $e_i\ v$ est calculée,*
- *sinon, si pour tout $i \in [\![1..n]\!]$, $E \neq E_i$ alors l'évaluation se poursuit en propageant l'exception $E\ (v)$.*

1. *Modifiez vos règles de sémantique pour prendre en compte ces nouvelles constructions.*
2. *La construction "*`try` e `with` $E_1 \to e_1 \mid \ldots \mid E_n \to e_n$ `finally` e*" (présente dans le langage* Java*) permet de rajouter à toute gestionnaire d'exceptions une partie finale à exécuter quelque soit l'issue de la confrontation de l'exception E à l'ensemble des $(E_i)_{i \in [\![1..n]\!]}$. Peut-on l'intégrer à notre langage sous la forme d'un sucre syntaxique ? Justifiez votre réponse. Est-ce que cette construction est utile dans un langage fonctionnel pur ?*

□

Exercice 5 (Sémantique statique)

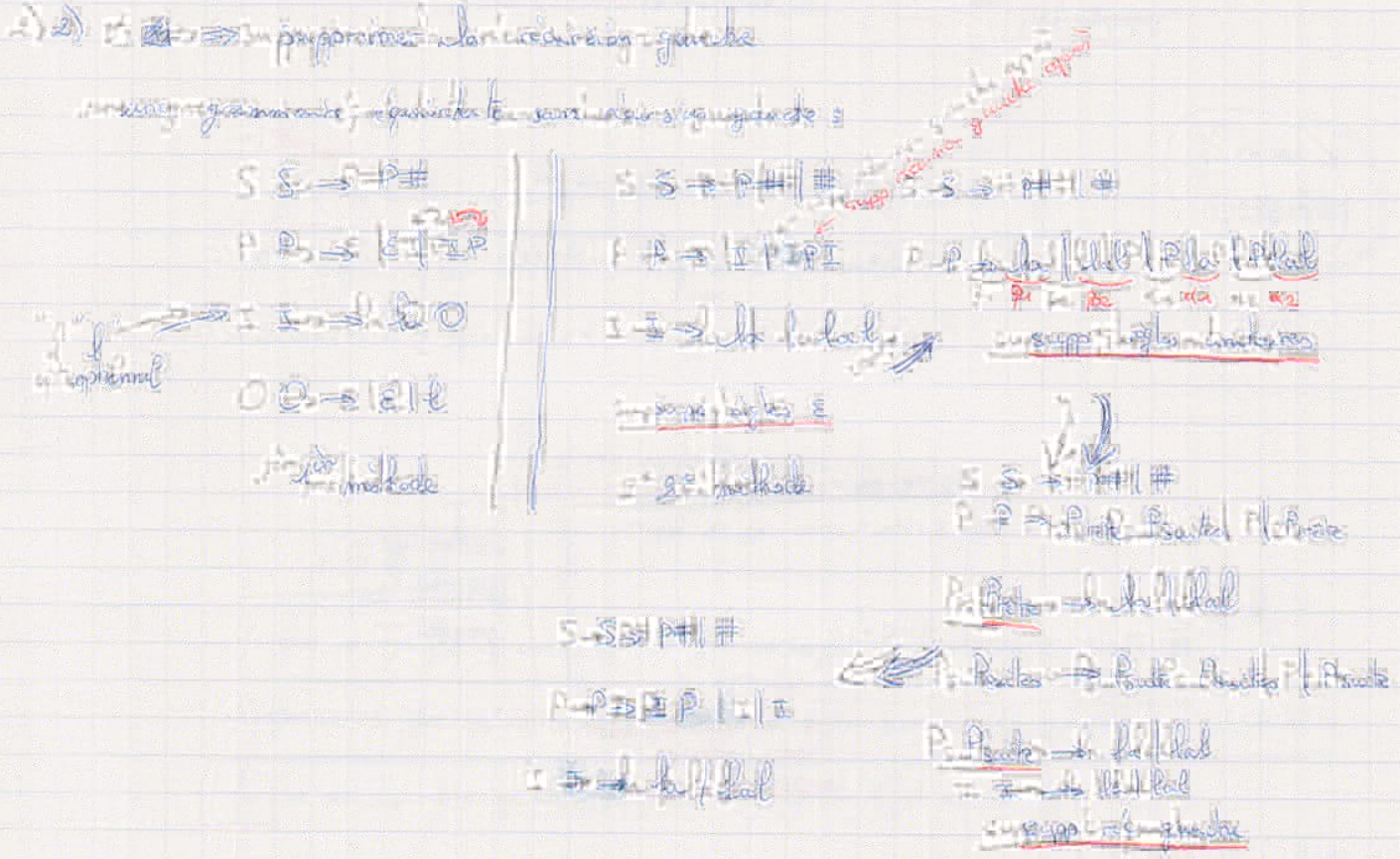
1. *Proposez des règles de typage pour le λ-calcul avec erreur.*
2. *Proposez des règles de typage pour le λ-calcul avec exception simple.*
3. *(bonus) Proposez une analyse statique qui détermine si une expression peut lever une exception non rattrapée par un gestionnaire d'exception.*

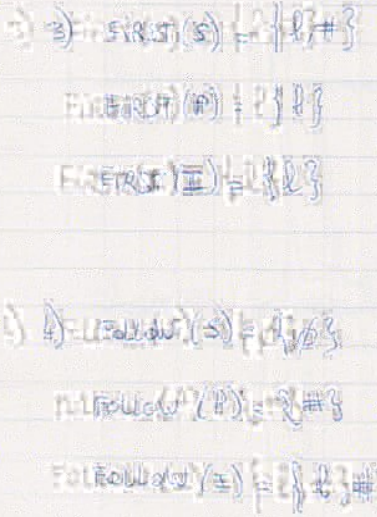
□

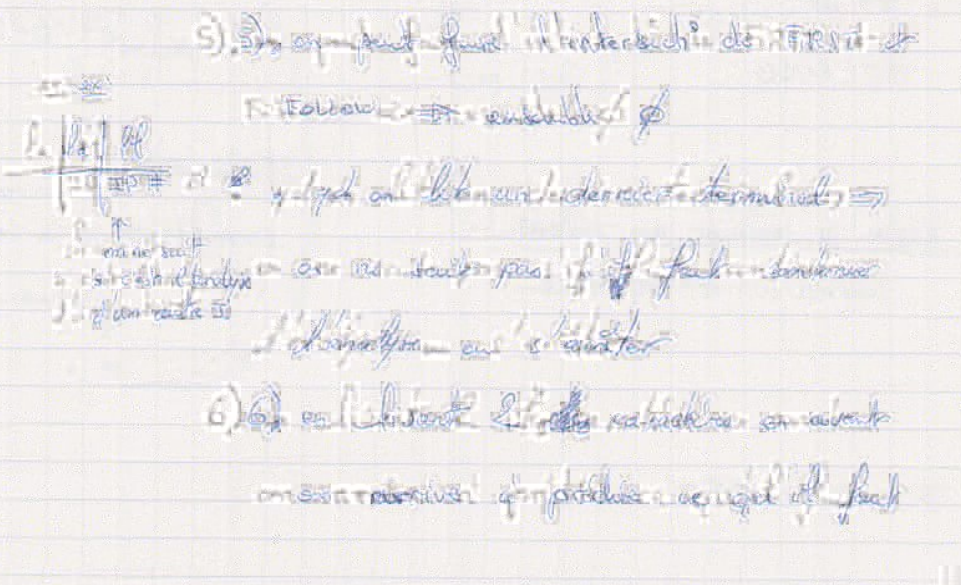
Analyse

→ revoir les algorithmes vus en cours / TP

1] Analyse syntaxique







2] Erreurs et exceptions

Sémantique ⇒ façon de spécifier la façon dont on

~~[illegible]~~

$e \Downarrow v$ grands pas : signifie une expression s'évalue en une valeur

$$\frac{2 \Downarrow 2 \qquad 3 \Downarrow 3}{\dfrac{1 \Downarrow 1 \qquad 2*3 \Downarrow 6}{1+2*3 \Downarrow 7}}$$

Sémantique à grands pas ⇒ on combine

$e \to e'$ petits pas

- $(2*3)+1 \longrightarrow 6+1 \longrightarrow 7$ (réduc)
- $(2*3)+(2*2) \xrightarrow{1} 6+(2*2) \xrightarrow{1} 6+4 \to$

ou

$(2*3)+(2*2) \xrightarrow{2} (2*3)+4 \xrightarrow{2} 6+4 \to$

$$\frac{e_1 \to e_1'}{e_1 + e_2 \to e_1' + e_2}$$

$$\frac{e \to e'}{v + e \to v + e'}$$

on évalue à gauche en premier

on attend d'abord une valeur avant d'évaluer

1) Règles de réduction :

β-Reduction

TRY-OK

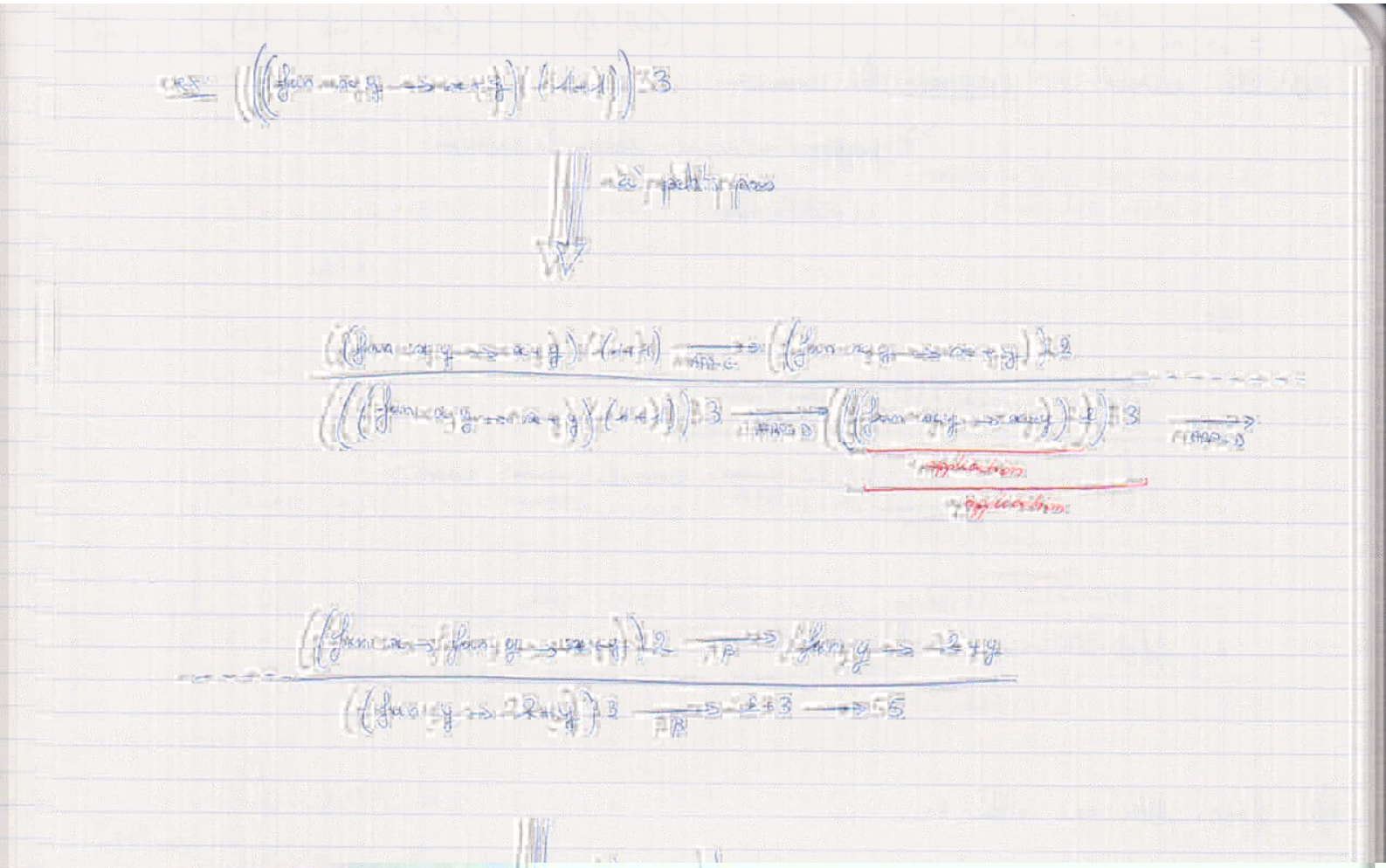
TRY-ECHEC

ex : $(\text{fun } x \to x+x)\ 21$ $\xrightarrow{\beta}$ $21+21$ $\to$ 42

$(\text{fun } x \to x+x)\ (10+11)$ $\xrightarrow{\text{APP-G}}$

Règles de passage au contexte : (réécrire une sous-expression)

TRY-SOUS-TERME



- Passage d'arguments : par valeur / référence
- Stratégie d'évaluation : par valeur / par nom

3) Elles servent à propager des erreurs

↳ propager l'erreur dans le contexte d'évaluation

ex :

- $$\frac{(\text{fun } x \to \text{error})\ 1 \xrightarrow[\beta]{} \text{error}}{\left(\left(\text{fun } x \to \text{error}\right) 1\right) 2 \xrightarrow[\text{APP-D}]{} \text{error}\ 2 \xrightarrow[\text{ERREUR-G}]{} \text{error}}$$

- $(\text{fun } x \to x)\ (\text{error}) \xrightarrow[\text{ERREUR-D}]{} \text{error}$

4) if e_1 then e_2 else e_3

if true then e_1 else $e_2 \longrightarrow e_1$

if false then e_1 else $e_2 \longrightarrow e_2$

Règles de réduction

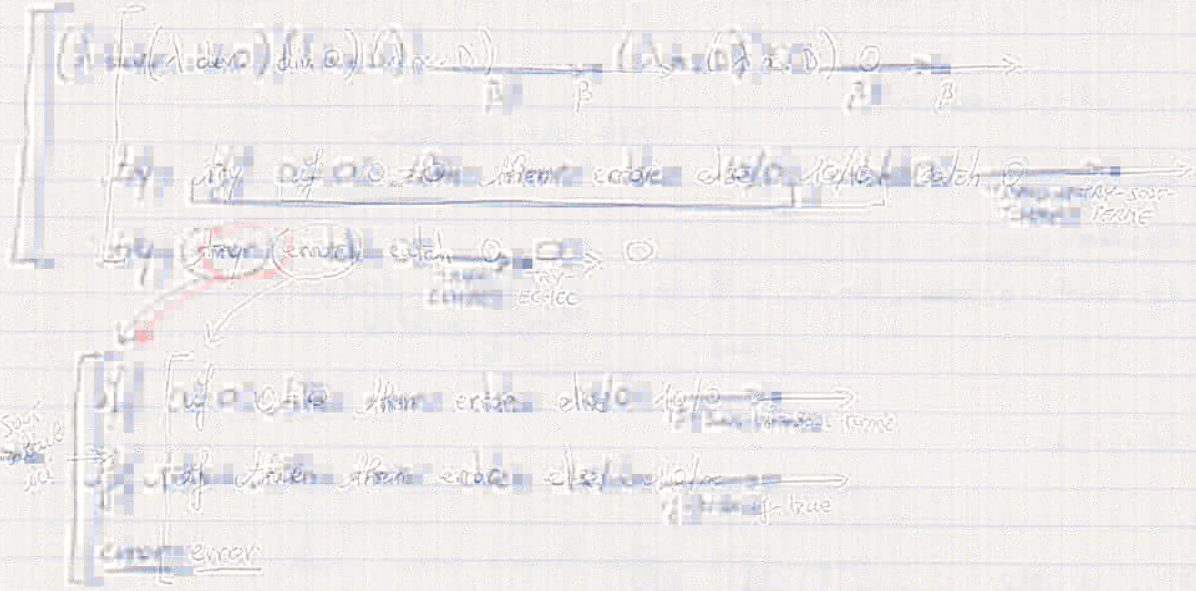
$e_1 \longrightarrow e_1'$

if e_1 then e_2 else $e_3 \longrightarrow$ if e_1' then e_2 else e_3

Règle de passage au contexte

if error then e_1 else $e_2 \longrightarrow$ error

Règle de propagation des erreurs



sucre syntaxique ← (let div = $\lambda x.$)   (λ = fun)

5) let div x =

D [try
  if x = 0 then error else 10/x
catch 0]

in
div 0

ex:
[let x = e_1 in e_2 ≡
(λx e_2) e_1 SUCRE SYNTAXIQUE]

⇒ [illegible] fun, [illegible] 0
([illegible] (div catch [illegible] , 0))

(λ div.(div 0)) ($\lambda x.$ D) $\xrightarrow{\beta}$ ($\lambda x.$ D) 0 $\xrightarrow{\beta}$

try if 0 = 0 then error else 10/0 catch 0 → TRY-SOUS-TERME

try error catch 0 → TRY-ERROR 0

if 0 = 0 then error else 10/0 → if-sous-terme

if true then error else 10/0 → if-true

error

?) [illegible] une exception qui n'est pas une valeur [illegible] qu'on peut [illegible] particulier

Erreur [illegible] typage [illegible] true 2
if [illegible] then ...

La valeur "Error" est une valeur

La valeur "Error" n'est pas une valeur [illegible] bloquée [illegible] en cours

Exo 3 :

1) β-réduction, APP-G et APP-D ne changent pas.

- $(\text{raise } v)\ t \longrightarrow \text{raise } v$
- $t\ (\text{raise } v) \longrightarrow \text{raise } v$
- $\text{if raise } v \text{ then } e_1 \text{ else } e_2 \longrightarrow \text{raise } v$

} règles de propagation d'erreur

- $\text{try } v \text{ with } e \longrightarrow v$
- $\text{try raise } v \text{ with } e \longrightarrow e\ v$

} règles de réduction

$$\frac{e_1 \longrightarrow e_1'}{\text{try } e_1 \text{ with } e_2 \longrightarrow \text{try } e_1' \text{ with } e_2}$$

$$\frac{e \longrightarrow e'}{\text{raise } e \longrightarrow \text{raise } e'}$$

} règles de passage au contexte

2) $\text{let div} = \lambda x.$

$$D \left[\begin{array}{l} \text{try} \\ \quad \text{if } x = 0 \text{ then raise } (x+1) \text{ else } 10/x \\ \text{with } \lambda y.\ y \end{array} \right.$$

in

$\text{div } 0$

$$(\lambda\, \text{div}.\ \text{div } 0)(\lambda x.\ D) \xrightarrow{\beta} (\lambda x.\ D)\ 0 \xrightarrow{\beta}$$

$\text{try if } 0 = 0 \text{ then raise } (0+1) \text{ else } 10/0 \text{ with } \lambda y.\ y \longrightarrow$ try sous-terme

$\text{try raise } 1 \text{ with } \lambda y.\ y \longrightarrow$

$(\lambda y.\ y)\ 1 \longrightarrow$ 1

$\text{if } 0 = 0 \text{ then raise } (0+1) \text{ else } 10/0 \longrightarrow$

$\text{if true then raise } (0+1) \text{ else } 10/0 \longrightarrow$

$\text{raise } (0+1) \longrightarrow$

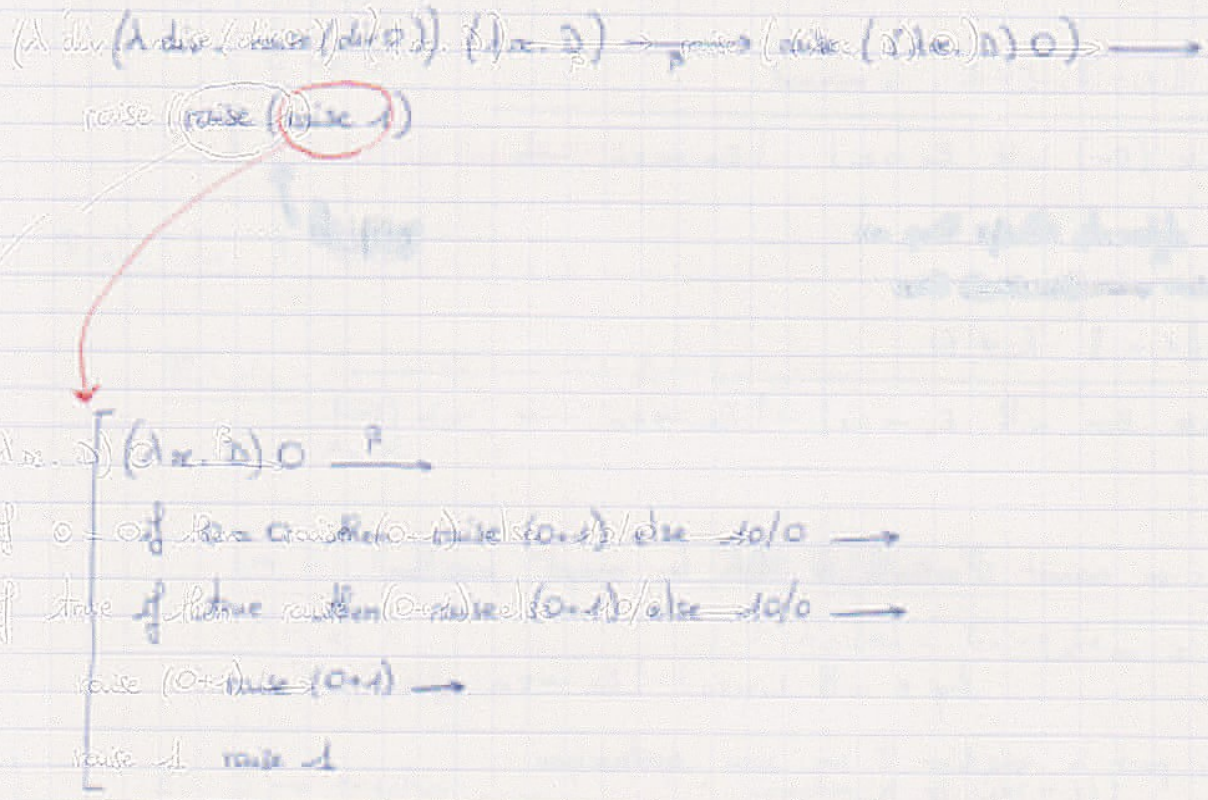
$\text{raise } 1$

3) let div = λ x

D[if x = 0 then raise (x+1) else 10/x

sin

(div (div) 0)



Exo 5 :

type exception

1) fun x -> raise Not_Found

Valeur ← type constructeur

type : $\alpha \to \beta$

a - Règles de typage :

$\tau ::= int \mid bool \mid \tau \to \tau$

$\mid \tau \to \tau$

$\mid \bot$ (le type bas, le type vide)

b - Règles de typage :

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash fun\ x \to e : \tau_1 \to \tau_2}$$

$$\frac{(x : \tau) \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma, (x : \tau_1) \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2}$$

2) $\tau \to exception$ — exception E of (int * int)

$\Gamma \vdash raise\ (E\ 3) : raise\ (E\ e) : \bot$ — E : int * int → exception

$$\frac{\Gamma \vdash e : \bot}{\Gamma \vdash e : \tau} \qquad \frac{\Gamma \vdash e : \bot}{\Gamma \vdash e : \tau}$$

3)

ex — ex — ex

$\Gamma \vdash true : bool$ — $\Gamma \vdash true \oplus bool$ — $\Gamma \vdash raise\ Not\_Found : raise\ Not\_Found : \bot$

$\Gamma \vdash if\ true\ then\ raise\ Not\_Found\ else\ raise\ Not\_Found : \bot$

raise) dans un programme ... programme ... type

algo - ... type ...

... Règles ...

... Sémantique ...