Conception de langage
Notes de cours 2006

(Brouillon)

1 Lambda-calcul

Définition les termes
- un ensemble V de variables ;
- l'application $(t\ u)$;
- l'abstraction $\lambda x.t$ avec $x \in V$.

Abréviations : $\lambda x_1 \dots x_n.t$ pour $\lambda x_1.\dots.\lambda x_n.t$; $(t\ u_1 \dots u_n)$ pour $(\dots(t\ u_1)\dots u_n)$.

Définition variables libres
- $VL(x) = \{x\}$;
- $VL(\lambda x.t) = VL(t) - \{x\}$;
- $VL(t\ u) = VL(t) \cup VL(u)$;
- $VL((t)) = VL(t)$.

x est *libre* dans t si $x \in VL(t)$.

Définition terme clos : $VL(t) = \emptyset$.

Définition substitution
- $x[v/x] = u$;
- $y[v/x] = y$ si $y \neq x$;
- $(\lambda x.t)[v/x] = \lambda x.t$;
- $(\lambda y.t)[v/x] = \lambda z.(t[z/y][v/x]$ si $x \neq y$ avec $z \notin VL(t) \cup VL(v)$;
- $(t\ u)[v/x] = (t[v/x]\ u[v/x])$.

Définition β-réduction : calcul par substitution
- $(\lambda x.t\ u) \Rightarrow_\beta t[u/x]$ (*redex*) ;
- $\lambda x.t \Rightarrow_\beta \lambda x.t'$ si $t \Rightarrow_\beta t'$;
- $(t\ u) \Rightarrow_\beta (t'\ u)$ si $t \Rightarrow_\beta t'$;
- $(t\ u) \Rightarrow_\beta (t\ u')$ si $u \Rightarrow_\beta u'$.

Clôture transitive : $t \Rightarrow_\beta^* t_n$ pour $t \Rightarrow_\beta t_1 \Rightarrow_\beta \dots \Rightarrow_\beta t_n$.

Définition
- forme normale : ne contient plus de redex.
  - x normal ;
  - $(x\ u_1 \dots u_n)$ si $u_1 \dots u_n$ normaux ;
  - $\lambda x.t$ si t normal.
- forme normale *faible* : x ou $\lambda x.t$ pour tout t ou $(x\ u_1 \dots u_n)$ avec $u_1, \dots, u_n$ en forme normale faible ; on ne «réduit pas sous les λ».

Si $t \twoheadrightarrow_\beta^* v$ et v normal, on note $t \twoheadrightarrow v$. Si v est normal, on note $v \twoheadrightarrow v$.

Il existe des termes dont la réduction est infinie :

$$(\lambda x.(x\ x)\ \lambda x.(x\ x)) \twoheadrightarrow_\beta \lambda x.(x\ x)[\lambda x.(x\ x)/x] \twoheadrightarrow_\beta (\lambda x.(x\ x)\ \lambda x.(x\ x))$$

1.1 Stratégies de réduction

Choix de l'ordre de traitement des redex.

Appel par valeur les arguments avant la fonction, réduction faible.

- $x \twoheadrightarrow x$
- $\lambda x.t \twoheadrightarrow \lambda x.t$
- $(t\ u) \twoheadrightarrow (t\ u')$ si $u \twoheadrightarrow u'$
- $(t\ v) \twoheadrightarrow (t'\ v)$ si $t \twoheadrightarrow t'$ et $v \twoheadrightarrow v$
- $(\lambda x.t\ v) \twoheadrightarrow t[v/x]$ si $v \twoheadrightarrow v$
- et c'est tout...

Bien : on n'évalue qu'une fois ; pas bien : on évalue l'inutile.

Appel par nom la fonction avant les arguments, réduction faible de tête.

- $x \twoheadrightarrow x$
- $\lambda x.t \twoheadrightarrow \lambda x.t$
- $(t\ u) \twoheadrightarrow (t'\ u)$ si $t \twoheadrightarrow t'$
- $(\lambda x.t\ u) \twoheadrightarrow t[u/x]$
- et c'est tout...

Bien : on n'évalue que l'utile ; pas bien : on évalue plusieurs fois.

Discussion faut-il choisir une stratégie, et laquelle ?

Le if-then-else : codable en lambda-calcul.

Codage des booléens : $\lambda x\lambda y.x$ pour vrai et $\lambda x\lambda y.y$ pour faux. On pose : $if = \lambda b\lambda x\lambda y.(b\ x\ y)$. Si b est l'un des booléens, le terme $(if\ b\ t_1\ t_2)$ se réduit bien comme «if b then t_1 else t_2». Mais, en appel par valeur, t_1 et t_2 sont évalués. On a égalité *dénotationnelle* (ie. même valeur) mais différence *opérationnelle* (ie. comportement calculatoire différents). En revanche, égalité opérationnelle en appel par nom.

Le point fixe (pour les fonctions récursives) : codable en lambda-calcul.

On pose $Y = \lambda f.(\lambda r.(f\ (r\ r))\lambda r.(f\ (r\ r)))$. Soit F un terme supposé en forme normale (faible) : on a, en réduction par valeur, $(Y\ F) \twoheadrightarrow (\lambda r.(F\ (r\ r))\lambda r.(F\ (r\ r))) \twoheadrightarrow (F\ (\lambda r.(F\ (r\ r))\ \lambda r.(F\ (r\ r))))$. Si on s'obstine en réduction par valeur, on obtient une suite du genre $(Y\ F), (F\ (YF)), (F\ (F\ (Y\ F))), etc.$ Qui ne permettra jamais d'évaluer F.

Moralité : il faut *mélanger les stratégies.*

1.2 Évaluateur

Pour une réduction faible en appel par valeur.

On retarde les substitutions : environnement (liste d'associations) ; clôture (ou fermeture) : couple terme/environnement.

On se souvient des applications : pile.

Un triplet (terme, environnement, pile) correspond au terme auquel on applique la substitution représentée par l'environnement et les arguments contenus dans la pile. La fonction *evalc* restitue le terme dénoté par un triplet :

$$\begin{array}{lcl} evalc(t, [], []) & = & t \\ evalc(t, [], (u_1, e_1) : s) & = & evalc((t\ evalc(u_1, e_1, [])), [], s) \\ evalc(t, (x, u, e') : e, s) & = & evalc(t[x/evalc(u, e', [])], e, s) \end{array}$$

L'évaluateur prend trois arguments : un terme, un environnement, une pile. Il calcule l'état (terme, environnement, pile) correspondant à la forme normale faible des valeurs d'entrée.

Si e est un environnement, on note $e(x)$ la valeur pour x dans e. Soit x variable, t,u termes, e environnement et s pile.

$$\begin{array}{lcll} eval(x,e,s) & = & eval(t,e',s) & \text{si } e(x) = (t,e') \\ eval(x,e,s) & = & (x,e,s) & \text{si } e(x) \text{ non définie} \\ eval((t\ u),e,s) & = & eval(t,e,((u',e):s) & \text{avec } eval(u,e,[]) = (u',e',s') \\ eval(\lambda x.t,e,c:s) & = & eval(t,((x,c):e),s & \\ eval(\lambda x.t,e,[]) & = & (\lambda x.t,e,[]) & \end{array}$$

1.3 Implantation

Réalisation de `(print(eval(read)))`.

1.3.1 `read` et `print` : syntaxe

Lexique et grammaire. Définition/reconnaissance. Expressions régulières, BNF.

Mots Un *alphabet* A, une opération (produit) de *concaténation*. Intuitivement si $a_1, a_2, \ldots, a_n \in A$ alors la juxtaposition $a_1 a_2 \ldots a_n$ est un mot. Chaque mot est fini, leur ensemble est infini.

Formellement $w : [1..n] \to A$, fonction d'un *segment initial* de $\mathbb{N}$ dans A. Si $w : [1..n] \to A$, n est la *longueur* de w, notée $|w|$. La concaténation de $w_1 : [1..n] \to A$ et $w_2 : [1..m] \to A$ est le mot (fonction) $w : [1..n+m] \to A$ telle que pour tout $i \in [1..n+m]$, $w(i) = w_1(i)$ si $i \leq n$ et $w(i) = w_2(i)$, sinon. On note $w = w_1 \cdot w_2$.

Élément neutre : mot vide (ε) ; associativité $w_1 \cdot (w_2 \cdot w_3) = (w_1 \cdot w_2) \cdot w_3$. *Monoïde.*

Exponentiation, étoile : $A^0 = \emptyset$, $A^1 = \{[1] \to A\}$, $A^{n+1} = \{a \cdot w; a \in A \text{ et } w \in A^n\}$, $A^* = \{w; \exists n \in \mathbb{N}.w \in A^n\}$. Propriété : $A^{n+m} = \{w_1 \cdot w_2; w_1 \in A^n \text{ et } w_2 \in A^m\}$.

Les opérations de concaténation et d'étoile s'étendent aux ensembles de mots : $W_1 \cdot W_2 = \{w_1 \cdot w_2; w_1 \in W_1 \text{ et } w_2 \in W_2\}$; $W^0 = \emptyset$, $W^{n+1} = W \cdot W^n$, $W^* = \{w; \exists n.w \in W^n\} = \bigcup_{n \in \mathbb{N}} W^n$.

Expressions régulières mots formés sur un alphabet par opérations de clôture : produit (concaténation), union, étoile.

Formellement : soit un alphabet A, l'ensemble $R(A)$ des langages rationnels (ou réguliers) sur A est défini par

- si $a \in A$ alors $\{a\} \in R(A)$
- si $W_1, W_2 \in R(A)$ alors $W_1 \cup W_2 \in R(A)$
- si $W_1, W_2 \in R(A)$ alors $W_1 \cdot W_2 \in R(A)$ (avec $W_1 \cdot W_2 = \{w_1 \cdot w_2; w_1 \in W_1 \text{ et } w_2 \in W_2\}$)
- si $W \in R(A)$ alors $W^* \in A$.

Implanation : lex (ou flex) ou ocamllex. Générateurs d'analyseur lexicaux à partir d'une description d'expressions régulières. Voir les diverses docs.

Exemple pour le lambda calcul :

- Symboles et mots réservés : `( ) LAMBDA` espace, tabulation ou retour-chariot comme *séparateurs*
- Identificateurs (variables) : `ident = [a-zA-Z0-9]*` ensemble de mots.

BNF Régles d'assemblages pour les mots, production de suites d'*unités lexicales* valides. Récursivité.

Terminaux : les unités lexicales

Non terminaux : ensembles, sous ensembles de phrases (assemblage) définis.

LTERM ::= ident
| (LTERMS)
| (LAMBDA (IDENTS) LTERM)

LTERMS ::=
| LTERM LTERMS

IDENTS ::= ident
| ident IDENTS

Implantations : yacc ou ocamlyacc. générateurs d'analyseur syntaxiques à partir d'une description de grammaire BNF. Grammaires attribuées, action *sémantique* (ie. code C ou OCAML) associé à chaque règle. Couplé avec lex ou ocammlex. Voir diverses docs.

1.3.2 Syntaxe abstraite

Structure arborescente reflet de la structure de production des termes.

À la ML type somme :

```
type lterm =
    LBD of string list * lterm
  | APP of lterm list
  | VAR of string
```

À la C tag, struct et union

```
enum lterm_cases { VAR, LBD, APP }

struct lterm_struct {
  lterm_cases case;
  union {
    char *var;
    struct {
      char **vars;
      struct lterm_struct *term;
    } lambda;
    struct {
      struct lterm_struct **terms;
    } apply;
  }
}
```

Le `read` : analyse syntaxique, traduction vers la syntaxe abstraite.
Le `print` : (re)présentation de la syntaxe abstraite en syntaxe concrète (affiche, chaîne de caractères).

On veut `(print(read` X`))` $= X$

Exercice: implanter : un type pour les lambda-termes, l'analyse lexicale et syntaxe, la fonction d'évaluation, la fonction de sortie du résultat (application de la substitution/environnement, renommage) ; utiliser les abréviations $\lambda x_1 \dots x_n.t$ et $(t\ u_1 \dots u_n)$.

1.4 Des extensions

1.4.1 Types *concrets*

Booléens, entiers, chaînes etc.
Nouveaux symboles pour les constantes (formes normales). Si c est une constante pour un type concret :

$$eval(c, e, s) \quad = \quad (c, [], [])$$

Valeur immédiate on note $[t]$ les clôtures dont le terme est t et l'environnement $[]$. On a une *valeur immédiate* si on a $[c]$ avec c constante pour un type concret.

Opérateurs symboles ou identificateurs pour les opérateurs. Calcul de la valeur confié à un évaluateur externe pour les valeur concrètes : $evalop(op, v_1, \ldots, v_n)$

$$eval((+\ u_1\ u_2), e, s) \quad = \quad evalop(+, eval(u_1, e, []), eval(u_2, e, []))$$

avec

$$evalop(+, [n_1], [n_2]) \quad = \quad [n_1 + n_2] \quad \text{si } n_1 \text{ et } n_2 \text{ valeurs numériques immédiates}$$

1.4.2 Contrôler l'évaluation

L'évaluateur implante une stratégie en appel par valeur. Il faut aussi de l'appel par nom (if-then-else, point fixe). Retarder l'évaluation : quote.

$$eval('t, e, s) \quad = \quad (t, [], [])$$

Vérifier sur : $(\lambda x.\lambda y.y\ '\Omega\ v)$.

1.4.3 Paires

Structure de données de base, codable en lambda-calcul.

$$\begin{array}{lcl} pair & = & \lambda x.\lambda y.\lambda p.(p\ x\ y) \\ fst & = & \lambda c.(c\ \lambda x.\lambda y.x) \\ snd & = & \lambda c.(c\ \lambda x.\lambda y.y) \end{array}$$

Pour *pair*, construction primitive (appel par nom) : $\langle t_1, t_2 \rangle$.
Pour les projections, deux primitives qui ne s'appliquent qu'aux paires, d'où, en optimisant :

$$\begin{array}{lcl} eval(fst, e, (\langle t_1, \_ \rangle, e') : s) & = & eval(t_1, e', s) \\ eval(snd, e, (\langle \_, t_2 \rangle, e') : s) & = & eval(t_2, e', s) \end{array}$$

Reconnaisseur : *pair*?

$$\begin{array}{lcll} eval(pair?, e, (\langle \_, \_ \rangle, e') : s) & = & [true] & \\ eval(pair?, e, s) & = & [false] & \text{sinon} \end{array}$$

La «paire vide» constante $\langle\rangle$

1.4.4 Oublier l'inutile

L'évaluation de $(\lambda x.\lambda y.x\ v_1\ v_2)$ créera deux entrées dans l'environnement : une pour lier x et v_1, une seconde pour lier y et v_2 alors que seule celle pour x et v_1 sera utile.
On se donne une «abstraction d'oubli», notée $\lambda\_$, telle que

$$eval(\lambda\_.t, e, c : s) \quad = \quad eval(t, e, s)$$

1.4.5 Alternative, *forme if*

Une constante primitive ? telle que (? *true*) se comporte comme la première projection et (? *false*) comme la seconde. Ce qui donne :

$$\begin{array}{lcl} eval(?, e, [true] : (\langle t_1, \_ \rangle, e') : s) & = & eval(t_1, e', s) \\ eval(?, e, [false] : (\langle \_, t_2 \rangle, e') : s) & = & eval(t_2, e_2, s) \end{array}$$

En ayant recours à de l'appel par nom, on obtient «if b then t_1 else t_2» avec (? b $\langle {}'t_1, {}'t_2 \rangle$). On a donc, non pas une fonction, mais une *forme if* : l'application (if b t_1 t_2) est *macro expansée* en (? b $\langle {}'t_1, {}'t_2 \rangle$).

1.4.6 Combinateur de point fixe

Dans la pratique, pour écrire une fonction récursive avec un combinateur de point fixe, on écrit d'abord un terme de la forme $\lambda f.t$, où t contient f, puis on prend (Y $\lambda f.t$) comme terme pour la fonction récursive. On obtient (grossièrement) la réduction : (Y $\lambda f.t$) $\twoheadrightarrow$ ($\lambda f.t(Y\ \lambda f.t)$) $\twoheadrightarrow$ $t[(Y\ \lambda f.t)/f]$.

On se donne un nouveau *lieur* (!) qui opère la manipulation de l'environnement suivante

$$eval(!f.t, e, s) \quad = \quad eval(t, (f, (!f.t, e)) : e, s)$$

Quelques liens utiles

Pour en savoir plus

- Sur le lambda-calcul :
  `http://www.lsv.ens-cachan.fr/~goubault/Lambda/lambda.pdf`
- Sur les ensembles et notations ensemblistes :
  `http://www.liafa.jussieu.fr/~ig/l2/handoutens.pdf`
- Sur l'analyse lexicale et les expressions régulières :
  `http://www.pps.jussieu.fr/~rifflet/enseignements/AF3/index.html`
- Sur l'analyse syntaxique et les grammaires BNF :
  `http://www.enseignement.polytechnique.fr/profs/informatique/Jean-Jacques.Levy/poly/main6/node2.`
- sur Lex/Yacc :
  `http://dinosaur.compilertools.net/`
- Sur Ocamllex/Ocamlyacc :
  `http://caml.inria.fr/pub/docs/manual-ocaml/manual026.html`

2 S-expressions

On définit un noyau du langage LISP : S-expressions, définitions. Avec son interprétation (traduction) en lambda-calcul. Ce qui nous donne un évaluateur.

2.1 Le langage

Lexique Symboles réservés : `' ()`

Unités lexicales autres :

```
num   =  -?[0-9]+\.?[0-9]*
str   =  "([^"]|\\")*"
atom  =  (alpha|sym)(alpha|sym|num)*
```

avec

```
alpha =  [a-zA-Z]
sym   =  [+=@_-#~&:\?%\*$%!/.]
```

Grammaire

SEXP	::=	num
	|	str
	|	atom
	|	(SEXPS)
SEXPS	::=	
	|	SEXP SEXPS

2.2 Traduction

On note $\|E\|$ le traduit en lambda-calcul de E. On note Xs une suite d'atomes et Es une suite de S-expressions. On note $\|Xs\|$ et $\|Es\|$ leurs traduits.

$S-expression(s)$	->	$lambda-terme$	
$\|$num$\|$		num	
$\|$str$\|$		str	
$\|$atom$\|$		x-ident	
$\|$`(LAMBDA (`Xs`)` E`)`$\|$		$\lambda\|Xs\|.\|E\|$	
$\|$`(IF` E_1 E_2 E_3`)`$\|$		$(?\ \|E_1\|\ \langle\|E_2\|.\|E_3\|\rangle)$	
$\|$`'`$E$$|$		$'\|E\|$	
$\|$`NIL`$\|$		$\langle\rangle$	
$\|$`(CONS` E_1 E_2`)`$\|$		$(\lambda x_1\ x_2.\langle x_1.x_2\rangle\ \|E_1\|\ \|E_2\|)$	
$\|$`(CAR` E`)`$\|$		$(fst\ \|E\|)$	
$\|$`(CDR` E`)`$\|$		$(snd\ \|E\|)$	
$\|$`(PAIR ?` E`)`$\|$		$(pair?\ \|E\|)$	
$\|$`(NIL ?` E`)`$\|$		$(nil?\ \|E\|)$	
$\|$`(`Xs`)`$\|$		$(\|Xs\|)$	
$\|$`(LET (`X E_1`)` E_2`)`$\|$		$(\lambda\|X\|.\|E_2\|\ \|E_1\|)$	
$\|$`(DEF (`X Xs`)` E`)` $Es$$|$		$(\lambda\|X\|.\|Es\|\ \lambda\|Xs\|.\|E\|)$	si X n'apparaît pas dans E
		$(\lambda\|X\|.\|Es\|\ !\|X\|.\lambda\|Xs\|.\|E\|)$	sinon.
Cas dégénéré : Xs est vide			
$\|$`(DEF (`X`)` E`)` $Es$$|$		$(\lambda\|X\|.\|Es\|\ \|E\|)$	

2.2.1 Commentaires

Presque tout est trivial : on avait ce qu'il fallait dans notre lambda calcul étendu. La fonction *eval* définie pour le lambda-calcul donne une fonction d'évaluation pour notre noyau LISP :

$$lisp.eval(E) = lambda.eval(\|E\|)$$

Deux cas appellent des remarques.

Le `CONS` On n'a pas fait $\|$`(CONS` E_1 E_2`)`$\| = \langle\|E_1\|.\|E_2\|\rangle$ pour forcer l'évaluation des membres de la paire.

Autre remarque : ça marche, mais ça n'est pas trés économique : `(CONS 1 (CONS 2 (CONS 3 NIL)))` est expensé en $(\lambda x_1\ x_2.\langle x_1.x_2\rangle\ 1\ (\lambda x_1\ x_2.\langle x_1.x_2\rangle\ 2\ (\lambda x_1\ x_2.\langle x_1.x_2\rangle\ 3\ \langle\rangle)))$. On gagnera à considérer `CONS` comme un symbole réservé défini dans un *environnement initial*. On pourra faire de même pour `CAR`, `CDR`, `PAIR ?` et `NIL ?`.

Les `DEF` Noter l'utilisation du point fixe.

Soit le petit programme `(DEF (X) 1) (ADD X 1)`. Il est traduite en $(\lambda X.(+\ X\ 1)\ 1)$. Son évaluation passe par les étapes ;

terme	env.	pile
$(\lambda X.(+\ X\ 1)\ 1)$	$[]$	$[]$
$\lambda X.(+\ X\ 1)$	$[]$	$[(1,[])]$
$(+\ X\ 1)$	$[(X,1,[])]$	$[]$

que l'on peut raccourcir en

$(\lambda X.(+\ X\ 1)\ 1)$	$[]$	$[]$
$(+\ X\ 1)$	$[(X,1,[])]$	$[]$

Il en sera toujours ainsi, pour toute suite de définitions (`DEF`).

En paramétrant la traduction par un environnement (σ), on pose :

$$\|\texttt{(DEF (}X\texttt{) }E\texttt{) }Es\|,\sigma \qquad \|Es\|,((X,\|E\|,\sigma):\sigma)$$

On donne ainsi la *sémantique* de `DEF` qui est bien de *modifier* l'environnement d'évaluation.

Conclusion en paramétrant la traduction par un environnement, on exprime aussi bien la sémantique des programmes LISP : quelles valeurs les S-expressions prennent-elle.

2.3 Sémantique dénotationnelle du noyau LISP

Notons Env l'ensemble (ou type) des environnements et $Sexp$ celui des S-expressions. On définit la fonction $\mathbf{L} : Sexp \to Env \to Val$. Laissons pour l'heure dans le flou le contenu de l'ensemble des *valeurs* Val. On note $\mathbf{L}[[E]],\sigma$ la dénotation (ie. la valeur) de l'expression E dans l'environnement σ.

Soit σ un environnement. On note $\sigma(x)$ la valeur associée à x par σ ; on note $(x,v):\sigma$ l'ajout de la liaison entre la variable x et la valeur v à σ : $((x,v):\sigma)(x) = v$ et $((x,v):\sigma)(y) = \sigma(y)$ pour $y \neq x$. On note id l'environnement identité ; $id(x) = x$.

$$
\begin{array}{lcll}
\mathbf{L}[[n]],\sigma & = & n & \text{avec } n \in \mathsf{num} \\
\mathbf{L}[[s]],\sigma & = & s & \text{avec } s \in \mathsf{str} \\
\\
\mathbf{L}[[X]],\sigma & = & \sigma(X) & \text{avec } X \in \mathsf{atom} \\
\\
\mathbf{L}[[(\texttt{LAMBDA}(X_1 \ldots) E)]],\sigma & = & \lambda v_1 \ldots (\mathbf{L}[[E]], (X_1, v_1) : \ldots : \sigma) & \\
\\
\mathbf{L}[[(\texttt{IF } E_1\ E_2\ E_3)]],\sigma & = & (?\ (\mathbf{L}[[E_1]],\sigma)\ \langle(\mathbf{L}[[E_2]],\sigma).(\mathbf{L}[[E_3]],\sigma)\rangle) & \\
\\
\mathbf{L}[[\texttt{'}E]],\sigma & = & \mathbf{L}[[E]], id & \\
\\
\mathbf{L}[[(\texttt{LET } (X\ E_1)\ E_2)]],\sigma & = & \mathbf{L}[[E_2]], (X, (\mathbf{L}[[E_1]],\sigma)) : \sigma & \\
\\
\mathbf{L}[[(E\ E_1 \ldots E_n)]],\sigma & = & ((\mathbf{L}[[E]],\sigma)\ (\mathbf{L}[[E_1]],\sigma) \ldots (\mathbf{L}[[E_n]],\sigma)) & \\
\\
\mathbf{L}[[(\texttt{DEF } X\ E)\ Es]],\sigma & = & \mathbf{L}[[Es]], (X, (\mathbf{L}[[E]],\sigma)) : \sigma & \\
\mathbf{L}[[(\texttt{DEF } (X\ X_1 \ldots)\ E)\ Es]],\sigma & = & \mathbf{L}[[Es]], (X, (\lambda v_1 \ldots (\mathbf{L}[[E]], (X_1, v_1) : \ldots : \sigma))) : \sigma & \\
 & & & \text{si } X \text{ non libre dans } E \\
\mathbf{L}[[(\texttt{DEF } (X\ X_1 \ldots)\ E)\ Es]],\sigma & = & \mathbf{L}[[Es]], (X, (!f.\lambda v_1 \ldots (\mathbf{L}[[E]], (X, f) : (X_1, v_1) : \ldots : \sigma))) : \sigma & \\
 & & & \text{sinon}
\end{array}
$$

La fonction **L** donne la dénotation des programmes LISP, mais ce n'est pas encore le résultat de leur évaluation. Exemple : soit σ_0 l'environnement initial (il contiendra dans notre exemple la définition de l'opérateur `+` de LISP – ie. l'opérateur + du lambda-calcul étendu :)

$$
\begin{array}{lcl}
\mathbf{L}[[\texttt{(DEF (SUCC X) (+ X 1)) (SUCC 1)}]],\sigma_0 & = & \mathbf{L}[[\texttt{(SUCC 1)}]], (\texttt{SUCC}, (\lambda v.(\mathbf{L}[[\texttt{(+ X 1)}]], (\texttt{X}, v) : \sigma_0))) : \sigma_0 \\
 & = & \mathbf{L}[[\texttt{(SUCC 1)}]], (\texttt{SUCC}, \lambda v.(+\ v\ 1)) : \sigma_0 \\
 & = & ((\mathbf{L}[[\texttt{SUCC}]], (\texttt{SUCC}, \lambda v.(+\ v\ 1)) : \sigma_0)\ (\mathbf{L}[[\texttt{1}]], \ldots)) \\
 & = & (\lambda v.(+\ v\ 1)\ 1)
\end{array}
$$

La valeur du programme `(DEF (SUCC X) (+ X 1)) (SUCC 1)` est bien celle du lambda-terme $(\lambda v.(+\ v\ 1)\ 1)$ qui est 2 *après* évaluation.

Liaison statique la valeur d'une variable (ou d'une fonction) capture son environnement de définition.

$$
\begin{array}{lcl}
\mathbf{L}[[\texttt{(DEF X 1)(DEF Y (+ X 1))(DEF X 0) Y}]],\sigma_0 & = & \mathbf{L}[[\texttt{(DEF Y (+ X 1))(DEF X 0) Y}]], (\texttt{X}, 1) : \sigma_0 \\
 & = & \mathbf{L}[[\texttt{(DEF X 0) Y}]], (\texttt{Y}, \mathbf{L}[[\texttt{(+ X 1)}]], (\texttt{X}, 1) : \sigma_0) : (\texttt{X}, 1) : \sigma_0 \\
 & = & \mathbf{L}[[\texttt{Y}]], (\texttt{X}, 0) : (\texttt{Y}, \mathbf{L}[[\texttt{(+ X 1)}]], (\texttt{X}, 1) : \sigma_0) : (\texttt{X}, 1) : \sigma_0 \\
 & = & \mathbf{L}[[\texttt{(+ X 1)}]], (\texttt{X}, 1) : \sigma_0) \\
 & = & (+\ (\mathbf{L}[[\texttt{X}]], (\texttt{X}, 1) : \sigma_0)\ 1) \\
 & = & (+\ 1\ 1)
\end{array}
$$

Un ouvrage utile sur les évaluateurs et la sémantiques lispienne
– C. Queinnec, *Les langages LISP*, InterEditions, 1994

3 Un noyau impératif

3.1 Extensions du noyau fonctionnel

On introduit deux traits impératifs dans les S-expressions : l'affectation et la séquence.

L'affectation quel sens lui donner ? Celui d'une (re)définition !

$$\mathbf{L}[[\texttt{(SET! } X\ E\texttt{)}]],\sigma \quad = \quad (\texttt{X}, \mathbf{L}[[E]],\sigma) : \sigma$$

Problème : on a ici $\mathbf{L} : Sexp \to Env \to Env$ alors qu'on avait $\mathbf{L} : Sexp \to Env \to Val$.

La séquence ça paraît plus simple : évaluer l'un puis l'autre. Ce qui se simule en utilisant liaison à une variable impossible

$$\begin{array}{lcl} \mathbf{L}[[\texttt{(PROGN } E\ Es\texttt{)}]],\sigma & = & (\lambda \bullet .(\mathbf{L}[[\texttt{(PROGN } Es\texttt{)}]],\sigma)\ (\mathbf{L}[[E]],\sigma)) \\ \mathbf{L}[[\texttt{(PROGN } E\texttt{)}]],\sigma & = & \mathbf{L}[[E]],\sigma \end{array}$$

L'ordre est donné par la stratégie d'évaluation par valeur. La valeur d'une séquence est la valeur du dernier élément.

Effets de bord comment les retrouver ?

On voudrait

$$\begin{array}{lcl} \mathbf{L}[[\texttt{(PROGN (SET! X 0) X)}]],\sigma & = & \mathbf{L}[[\texttt{X}]],(\texttt{X},0) : \sigma \\ & = & 0 \end{array}$$

Notons que

$$(\texttt{X},0) : \sigma = (\texttt{X}, (\mathbf{L}[[\texttt{0}]],\sigma)) : \sigma = \mathbf{L}[[\texttt{(SET! X 0)}]],\sigma$$

On aurait donc

$$\mathbf{L}[[\texttt{(PROGN (SET! X 0) X)}]],\sigma \quad = \quad \mathbf{L}[[\texttt{X}]], (\mathbf{L}[[\texttt{(SET! X 0)}]],\sigma)$$

Mais *quid* de $\mathbf{L}[[\texttt{(PROGN 1 2)}]],\sigma$? Ce ne peut être $\mathbf{L}[[2]], (\mathbf{L}[[1]],\sigma)$, car $\mathbf{L}[[1]],\sigma$ n'est pas un environnement.

En revanche, si on a $\mathbf{L}[[\texttt{(DEF } X\ E\texttt{)}]],\sigma = (\texttt{X}, (\mathbf{L}[[E]],\sigma)) : \sigma$, on retrouve

$$\mathbf{L}[[\texttt{(DEF } X\ E\texttt{) } Es]],\sigma \quad = \quad \mathbf{L}[[Es]], (\mathbf{L}[[\texttt{(DEF } X\ E\texttt{)}]],\sigma)$$

Moralite scinder la sémantique selon que l'on veut une valeur ou un environnement.

Deux fonctions sémantiques : $\mathbf{V} : Sexp \to Env \to Val$ et $\mathbf{E} : Sexp \to Env \to Env$

$$\begin{array}{lcll} \mathbf{V}[[n]],\sigma & = & n & \text{avec } n \in \mathsf{num} \\ \mathbf{V}[[s]],\sigma & = & s & \text{avec } s \in \mathsf{str} \\ \\ \mathbf{V}[[X]],\sigma & = & \sigma(X) & \text{avec } X \in \mathsf{atom} \\ \\ \mathbf{V}[[\texttt{(LAMBDA(}X_1 \ldots \texttt{)}E\texttt{)}]],\sigma & = & \lambda v_1 \ldots (\mathbf{V}[[E]], (X_1, v_1) : \ldots : \sigma) & \\ \\ \mathbf{V}[[\texttt{(IF } E_1\ E_2\ E_3\texttt{)}]],\sigma & = & (?\ (\mathbf{V}[[E_1]],\sigma)\ \langle(\mathbf{V}[[E_2]],\sigma).(\mathbf{V}[[E_3]],\sigma)\rangle) & \\ \\ \mathbf{V}[[\texttt{'}E]],\sigma & = & \mathbf{V}[[E]], id & \\ \\ \mathbf{V}[[\texttt{(LET (}X\ E_1\texttt{) } E_2\texttt{)}]],\sigma & = & \mathbf{V}[[E_2]], (X, (\mathbf{V}[[E_1]],\sigma)) : \sigma & \\ \\ \mathbf{V}[[\texttt{(PROGN } E\texttt{)}]],\sigma & = & \mathbf{V}[[E]],\sigma & \\ \mathbf{V}[[\texttt{(PROGN } E\ Es\texttt{)}]],\sigma & = & \mathbf{V}[[\texttt{(PROGN } Es\texttt{)}]], (\mathbf{E}[[E]],\sigma) & \\ \\ \mathbf{V}[[\texttt{(}E\ E_1 \ldots E_n\texttt{)}]],\sigma & = & ((\mathbf{V}[[E]],\sigma)\ (\mathbf{V}[[E_1]],\sigma) \ldots (\mathbf{V}[[E_n]],\sigma)) & \end{array}$$

$$
\begin{array}{lcl}
\mathbf{E}[[\texttt{(SET!}\ X\ E\texttt{)}]],\sigma & = & (\texttt{X}, (\mathbf{V}[[E]],\sigma)) : \sigma \\
 & & \\
\mathbf{E}[[\texttt{(DEF}\ X\ E\texttt{)}]],\sigma & = & (X, (\mathbf{V}[[E]],\sigma)) : \sigma \\
\mathbf{E}[[\texttt{(DEF (}X\ X_1 \ldots\texttt{)}\ E\texttt{)}]],\sigma & = & (X, (\lambda v_1 \ldots (\mathbf{V}[[E]], (X_1, v_1) : \ldots : \sigma))) : \sigma \\
 & & \qquad \text{si } X \text{ non libre dans } E \\
\mathbf{E}[[\texttt{(DEF (}X\ X_1 \ldots\texttt{)}\ E\texttt{)}]],\sigma & = & (X, (!f.\lambda v_1 \ldots (\mathbf{V}[[E]], (X, f) : (X_1, v_1) : \ldots : \sigma))) : \sigma \\
 & & \qquad \text{sinon} \\
\mathbf{E}[[E]],\sigma & = & \mathbf{V}[[E]],\sigma \\
\mathbf{E}[[E\ Es]],\sigma & = & \mathbf{E}[[Es]], (\mathbf{E}[[E]],\sigma)
\end{array}
$$

La sémantique ne distingue pas affectation et définition ; la différence entre $\mathbf{V}[[\texttt{(PROGN}\ Es\texttt{)}]],\sigma$ et $\mathbf{E}[[Es]],\sigma$ ne saute pas aux yeux.

Nous allons préciser cela.

3.2 Syntaxe et sémantique

Si l'on regarde ce que sait traiter $\mathbf{E}[[Es]]$, on s'aperçoit que ceux sont des suites de la forme `(DEF...)` `(DEF...)` ... E où E est traitable par $\mathbf{V}[[]]$.

Si l'on regarde ce que sait traiter $\mathbf{V}[[\texttt{(PROGN}\ Es\texttt{)}]]$, on s'aperçoit que Es ne peut être que de la forme `(SET! ...)` `(SET! ...)` E) avec E traitable par $\mathbf{V}[[]]$.

On retrouve là deux catégories syntaxiques des langages de programmation : les *programmes* et les *instructions*. Elles n'étaient pas explicitées par la grammaire ; faisons le.

PROG ::= STAT
| DECS STAT

DEC ::= (DEF atom EXP)
| (DEF (atom ATOMS)EXP)

DECS ::= DEC
| DEC DECS

STAT ::= (SET! atom EXP)
| (PROGN STATS)

STATS ::= STAT
| STAT STATS

EXP ::= num
| str
| atom
| (LAMBDA (ATOMS)EXP)
| (IF EXP EXP EXP)
| (EXP EXPS)

EXPS ::= EXP
| EXP EXPS

ATOMS ::= atom
| atom ATOMS

La donnée affinée de la syntaxe permet de poser plus clairement : seules les expressions (EXP) auront une valeur (fonction sémantique $\mathbf{V}[[]]$) ; le reste (instructions –STAT– et programmes), non. C'est un choix, il

peut se discuter !

À chaque règle syntaxique, on associe une fonction sémantique :

D : DEC $\rightarrow Env \rightarrow Env$
Ds : DECS $\rightarrow Env \rightarrow Env$
S : STAT $\rightarrow Env \rightarrow Env$
Ss : STATS $\rightarrow Env \rightarrow Env$
E : EXP $\rightarrow Env \rightarrow Val$

Notons que l'on a

P : PROG $\rightarrow Env \rightarrow Env$

Soient X un atome, S une instruction, E une expression, D une définition et Ss, Es, Ds des suites d'instructions, expressions, définitions. On pose :

$$\begin{array}{lcl}
\mathbf{P}[[S]],\sigma & = & \mathbf{S}[[S]],\sigma \\
\mathbf{P}[[Ds\ S]],\sigma & = & \mathbf{S}[[S]],(\mathbf{Ds}[[Ds]],\sigma) \\
 & & \\
\mathbf{D}[[\texttt{(DEF}\ X\ E\texttt{)}]],\sigma & = & (X,\mathbf{E}[[E]],\sigma) : \sigma \\
\mathbf{Ds}[[D\ Ds]],\sigma & = & \mathbf{Ds}[[Ds]],(\mathbf{D}[[D]],\sigma) \\
 & & \\
\mathbf{S}[[\texttt{(SET!}\ \ X\ E\texttt{)}]],\sigma & = & \sigma[X := (\mathbf{E}[[E]],\sigma)] \\
\mathbf{S}[[\texttt{(PROGN}\ Ss\texttt{)}]],\sigma & = & \mathbf{Ss}[[Ss]],\sigma \\
 & & \\
\mathbf{Ss}[[S]],\sigma & = & \mathbf{S}[[S]],\sigma \\
\mathbf{Ss}[[S\ Ss]],\sigma & = & \mathbf{Ss}[[Ss]],(\mathbf{S}[[S]],\sigma) \\
 & & \\
\mathbf{E}[[n]],\sigma & = & n \\
 & & etc.
\end{array}$$

Remarque : pour l'affectation, on a distingué une opération sur les environnements ($\sigma[x := v]$), elle est de type $\mathsf{atom} \rightarrow Val \rightarrow Env \rightarrow Env$. On pose :

$$\begin{array}{lcll}
\sigma[x := v] & = & Erreur & \text{si } \sigma(x) \text{ non défini} \\
\sigma[x := v](x) & = & v & \\
\sigma[x := v](y) & = & \sigma(y) & \text{si } y \neq x
\end{array}$$

4 Variation syntaxique et conséquence

Soit la syntaxe alternative

```
Prog  ::=  Stat
       |   Decs Stat

Dec   ::=  Cst id = Exp ;
       |   Var id ;
       |   Fun id (Ids )= Exp ;

Decs  ::=  Dec
       |   Dec Decs

Stat  ::=  id := Exp ;
       |   Begin Stats End ;

Stats ::=  Stat
       |   Stat Stats

Exp   ::=  num
       |   str
       |   id
       |   if(Exp : Exp, Exp )
       |   id(Exps )
       |   ...†

Exps  ::=  Exp
       |   Exp, Exps

Ids   ::=  id
       |   id, Ids
```

† : on omet ici les notations pour les opérateur préfixes, infixes, etc.

Il est clair que l'on peut donner à cette syntaxe la même sémantique qu'en 3.2 pour peu que l'on donne une valeur par défaut aux variables (`Var`).

Cependant, cette sémantique pêche sur deux points (au moins :)

- elle ne distingue pas variables et constantes : on peut modifier la valeur d'une constante. Pire, rien n'empêche d'affecter le nom d'une fonction !
- avec l'apparition de l'affectation, vient la notion de *dynamicité* du mécanisme de liaison qu'il vaut mieux expliciter.

4.1 Dinstinguer

Pour traiter le *distingo* entre constante et variable, deux solutions :

1. dinstinguer constantes et variables dans l'environnement.
2. dinstinguer les epaces de noms : environnement et *mémoire*.

1. Somme disjointe Pour appliquer la première solution, on «duplique» le domaine Val des valeurs en en faisant la somme disjointe notée $Val + Val$. Concrètement, les élements de $Val + Val$ sont des couples de la forme $(0, v)$ ou $(1, v)$. Ce qui permet de discriminer des valeurs d'une copie de Val de celles de l'autre.

Si l'on veut distinguer les constantes des variables, on note $inCst(v)$ pour $(0, v)$ et $inVar(v)$ pour $(1, v)$. On peut traiter une somme disjointe comme un type abstrait avec ses constructeurs (ici $inCst : Val \to Val+Val$ et $inVar : Val \to Val + Val$), des accesseurs ($valCst : Val + Val \to Val$ et $valVar : Val + Val \to Val$) et ses reconnaiseurs ($isCst : Val + Val \to Bool$ et $isVar : Val + Val \to Bool$). On peut, si on le désire, ajouter un *distingo* pour les fonctions ($inFun$, $isFun$, $valFun$).

$$
\begin{array}{lcll}
\ldots & & & \\
\mathbf{D}[[\texttt{Cst } x \texttt{ = } e \texttt{ ;}]], \sigma & = & (x, inCst(\mathbf{E}[[e]], \sigma)) : \sigma & \\
\mathbf{D}[[\texttt{Var } x \texttt{ ;}]], \sigma & = & (x, inVar(default)) : \sigma & \\
\mathbf{D}[[\texttt{Fun } f\texttt{(}x_1, \ldots\texttt{) = } e \texttt{ ;}]], \sigma & = & (x, inFun(\lambda v_1 \ldots (\mathbf{E}[[e]], (x_1, v_1) : \ldots : \sigma))) : \sigma & \\
\ldots & & & \\
\mathbf{S}[[x \texttt{ := } e \texttt{ ;}]], \sigma & = & \sigma[x := (\mathbf{E}[[e]], \sigma)] & \text{si } isVar(\sigma(x)) \\
 & = & Erreur & \text{sinon} \\
\ldots & & & \\
\mathbf{E}[[x]], \sigma & = & valOf(\sigma(x)) & \\
\ldots & & &
\end{array}
$$

avec $valOf(inCst(v)) = v$, $valOf(inVar(v)) = v$ et $valOf(inFun(v)) = v$ ou $Erreur$ selon que l'on veut des valeurs fonctionnelles ou non dans les expressions.

2. Mémoire Pour appliquer la seconde solution, on introduit $\mu : Mem$ avec $Mem = \mathsf{id} \rightarrow Val$ telle que $\mu(x)$ donne la valeur du nom x et l'opération de mise à jour $\mu[x := v]$ décrite plus haut.

Les déclarations peuvent affecter soit l'environnement σ soit la mémoire μ. L'instruction d'affectation ne s'intéresse qu'à la mémoire : $\mathbf{D}[[]] : Env \rightarrow Mem \rightarrow Env \times Mem$ et $\mathbf{S}[[]] : Env \rightarrow Mem \rightarrow Mem$.

$$
\begin{array}{lcl}
\ldots & & \\
\mathbf{D}[[\texttt{Cst } x \texttt{ = } e \texttt{ ;}]], \sigma, \mu & = & ((x, (\mathbf{E}[[e]], \sigma)) : \sigma), \mu \\
\mathbf{D}[[\texttt{Var } x \texttt{ ;}]], \sigma, \mu & = & \sigma, ((x, default) : \mu) \\
\mathbf{D}[[\texttt{Fun } f\texttt{(}x_1, \ldots\texttt{) = } e \texttt{ ;}]], \sigma, \mu & = & (x, inFun(\lambda v_1 \ldots (\mathbf{E}[[e]], (x_1, v_1) : \ldots : \sigma, \mu))) : \sigma, \mu \\
\ldots & & \\
\mathbf{S}[[x \texttt{ := } e \texttt{ ;}]], \sigma, \mu & = & (\mu[x := (\mathbf{E}[[e]], \sigma)]) \\
\mathbf{S}[[\texttt{Begin } Ss \texttt{ End}]], \sigma, \mu & = & \mathbf{Ss}[[Ss]], \sigma, \mu \\
 & & \\
\mathbf{Ss}[[S]], \sigma, \mu & = & \mathbf{S}[[S]], \sigma, \mu \\
\mathbf{Ss}[[S\ Ss]], \sigma, \mu & = & \mathbf{Ss}[[Ss]], \sigma, (\mathbf{S}[[S]], \sigma, \mu)
\end{array}
$$

L'accès à la valeur d'un identificateur doit gérer la possibilité d'une erreur :

$$
\begin{array}{lcll}
\mathbf{E}[[x]], \sigma, \mu & = & \sigma(x) & \\
 & = & \mu(x) & \text{si } \sigma(x) = Erreur
\end{array}
$$

Solution mixte Pour éviter l'essai/erreur de l'accés à la valeur d'un nom, on introduit un domaine d'adresses Adr, on pose $Mem = Adr \rightarrow Val$. Tant qu'on y est, on peut aussi disntinguer les fonctions : domaine Fun. On pose $Env = \mathsf{id} \rightarrow Val + Fun + Adr$.

On suppose une fonction $new : Mem \rightarrow Adr$ telle que $new(\mu)$ fournisse une adresse non encore définie pour μ.

$$
\begin{array}{lcll}
\ldots & & & \\
\mathbf{D}[[\texttt{Var } x \texttt{ ;}]], \sigma, \mu & = & (x, inAdr(new(\mu)))\sigma, \mu) & \\
\mathbf{D}[[\texttt{Cst } x \texttt{ = } e \texttt{ ;}]], \sigma, \mu & = & ((x, inVal(\mathbf{E}[[e]], \sigma)) : \sigma), \mu & \\
\mathbf{D}[[\texttt{Fun } f\texttt{(}x_1, \ldots\texttt{) = } e \texttt{ ;}]], \sigma, \mu & = & (x, inFun(\lambda v_1 \ldots (\mathbf{E}[[e]], (x_1, v_1) : \ldots : \sigma, \mu))) : \sigma, \mu & \\
\ldots & & & \\
\mathbf{S}[[x \texttt{ := } e \texttt{ ;}]], \sigma, \mu & = & \mu[valOf(\sigma(x)) := (\mathbf{E}[[e]], \sigma)] & \text{si } isAdr(\sigma(x)) \\
\ldots & & & \\
\mathbf{E}[[x]], \sigma, \mu & = & \mu(\sigma(x)) & \text{si } isAdr(\sigma(x)) \\
 & = & valOf(\sigma(x)) & \text{sinon} \\
\ldots & & &
\end{array}
$$

4.2 Statique *vs* dynamique

Quid des variables dans les définitions de constantes et de fonctions ?

La suite `Var x ; Cst y = x+1 ;` provoque une *Erreur* lorsque l'on essaie de calculer $\mathbf{E}$[[`x+1`]] dans la définition de `y` (cf. $\mathbf{D}$[[`Var x ;`]]). C'est souhaitable.

La suite `Var x0 ; Fun f(x) = x+x0 ;` provoquera également une *Erreur* si l'on veut calculer $\lambda v.(\mathbf{E}[[\texttt{x+x0}]], ..)$. Est-ce souhaitable ? L'habitude dit «non» : une fonction doit pouvoir dépendre de la valeur d'une *variable globale* : liaison dynamique.

$$\mathbf{D}[[\texttt{Fun } f\texttt{(}x_1, \ldots\texttt{) = } e\texttt{ ;}]], \sigma, \mu \quad = \quad (x, inFun(\lambda m.\lambda v_1 \ldots (\mathbf{E}[[e]], (x_1, v_1) : \ldots : \sigma, m))) : \sigma, \mu$$

avec, pour l'application de fonction

$$\mathbf{E}[[f\texttt{(}e_1, \ldots\texttt{)}]], \sigma, \mu \quad = \quad ((\mathbf{E}[[f]], \sigma, \mu)\ \mu\ (\mathbf{E}[[e_1]], \sigma, \mu] \ldots)$$

Prix à payer le lambda-calcul gére *de facto* l'environnement (cf. fonction *eval*) mais pas la mémoire. Si l'on veut une sémantique à mémoire explicite – qui est proche de l'intuition – il faut l'intégrer au lambda-calcul. On peu :

- soit étendre le lambda-calcul aux effets de bord : opérateur primitif *set*! ;
- soit faire de la mémoire une valeur codée par des lambda-termes : liste d'association avec ses fonctions d'ajout, accès.

4.3 Complément d'impératif

On enrichit l'ensemble des instructions : alternative, boucles, blocs avec déclarations locales, procédures. Au domaine des valeurs s'ajoute le constructeur *inProc* et ses accolytes.

Syntaxe

DEC ::= ...
| `Proc id (` IDS `)=` STAT

STAT ::= ...
| `id (` EXPS `)`
| `If` EXP `Then` STAT `Else` STAT
| `While` EXP `Do` STAT
| `Begin` DECS STATS `End ;`

EXP ::= ...
| { DECS EXP }
...

Sémantique

$$\mathbf{D}[[\texttt{Proc } p\texttt{(}x_1, \ldots\texttt{)=}S]], \sigma, \mu \quad = \quad (x, inProc(\lambda m.\lambda v_1 \ldots.(\mathbf{S}[[S]], (x_1, v_1) : \ldots : \sigma, m))) : \sigma, \mu$$

$$\begin{array}{lcll}
\mathbf{S}[[p\texttt{(}e_1, \ldots\texttt{)}]], \sigma, \mu & = & (valOf(\sigma(p))\ \mu\ (\mathbf{E}[[e_1]], \sigma, \mu) \ldots) & \text{si } isProc(\sigma(p)) \\
\mathbf{S}[[\texttt{If } e\texttt{ Then } S_1\texttt{ Else } S_2]], \sigma, \mu & = & (?\ (\mathbf{E}[[e]], \sigma, \mu)\ \langle \mathbf{S}[[S_1]], \sigma, \mu \cdot \mathbf{S}[[S_1]], \sigma, \mu \rangle) & \\
\mathbf{S}[[\texttt{While } e\texttt{ Do } S]], \sigma, \mu & = & (!w.\lambda m.(?\ (\mathbf{E}[[e]], \sigma, m) \langle (w\ (\mathbf{S}[[S]], \sigma, m)) \cdot m \rangle)\ \mu) & \\
\mathbf{S}[[\texttt{Begin } Ds\ Ss\texttt{ End ;}]], \sigma, \mu & = & \mathbf{Ss}[[Ss]](\mathbf{Ds}[[Ds]], \sigma, \mu) &
\end{array}$$

$$\mathbf{E}[[\{Ds\ E\}]], \sigma, \mu \quad = \quad \mathbf{E}[[E]](\mathbf{Ds}[[Ds]], \sigma, \mu)$$

Un mot d'explication sur la boucle :

$$\begin{aligned}
\underbrace{\mathbf{S}[[\texttt{While } e \texttt{ Do } S]]}_{w}, \sigma, \mu &= (\lambda m.(?\ (\mathbf{E}[[e]], \sigma)\ \langle(\mathbf{S}[[\texttt{Begin } S \texttt{ While } e \texttt{ Do } S \texttt{ End}]], \sigma, m) \cdot m\rangle)\ \mu) \\
&= (\lambda m.(?\ (\mathbf{E}[[e]], \sigma, \mu)\ \langle(\underbrace{\mathbf{S}[[\texttt{While } t \texttt{ Do } S]]}_{w}(\mathbf{S}[[S]], \sigma, m)) \cdot m\rangle)\ \mu) \\
&= !w.\lambda m.(?\ (\mathbf{E}[[e]], \sigma, \mu)\ \langle(w\ (\mathbf{S}[[S]], \sigma, m)) \cdot m\rangle)
\end{aligned}$$

5 Continuations

Expliciter le flux de contrôle : sémantique avec pile.

$$\begin{aligned}
\mathbf{S}[[\texttt{Begin } S\ Ss \texttt{ End}]], \pi, \sigma, \mu &= \mathbf{S}[[S]], (\mathbf{S}[[\texttt{Begin } Ss \texttt{ End}]] : \pi), \sigma, \mu \\
\mathbf{S}[[x \texttt{ := } e]], (C : \pi), \sigma, \mu &= C, \pi, \sigma, (\mu[x := (\mathbf{E}[[e]], \sigma, \mu)]) \\
\mathbf{S}[[\texttt{While } e \texttt{ Do } S]], (C : \pi), \sigma, \mu &= (?\ (\mathbf{E}[[e]], \sigma, \mu)\ \langle\mathbf{S}[[S]], (\mathbf{S}[[\texttt{While } e \texttt{ Do } S]] : C : \pi), \sigma, \mu \cdot C, \pi, \sigma, \mu\rangle)
\end{aligned}$$

Pour être complet, on peut rajouter

$$\mathbf{S}[[\texttt{Begin End}]], (C : \pi), \sigma, \mu = C, \pi, \sigma, \mu$$

Le sommet de pile C prend toujours en argument la suite de la suite π : le calcul en cours prend en argument le calcul à suivre. Faire de la pile une fonction : continuation $\kappa : Cont = Mem \to Mem$. Une instruction donne une fonction de la mémoire dans la mémoire.

$\mathbf{S}[[S]] : (Mem \to Mem) \to Env \to Mem \to Mem$.

$$\begin{aligned}
\mathbf{S}[[x \texttt{ := } e]], \kappa, \sigma, \mu &= (\kappa\ (\mu[x := \mathbf{E}[[e]], \sigma, \mu])) \\
\mathbf{S}[[\texttt{If } e \texttt{ Then } S_1 \texttt{ Else } S_2]], \kappa, \sigma, \mu &= (?\ (\mathbf{E}[[e]], \sigma, \mu)\ \langle\mathbf{S}[[S_1]], \kappa, \sigma, \mu \cdot \mathbf{S}[[S_2]], \kappa, \sigma, \mu\rangle) \\
\mathbf{S}[[\texttt{While } e \texttt{ Do } S]], \kappa, \sigma, \mu &= (!\kappa_w.\lambda m.(?\ (\mathbf{E}[[e]], \sigma, m)\ \langle\mathbf{S}[[S]], \kappa_w, \sigma, m \cdot (\kappa\ m)\rangle)\ \mu) \\
\mathbf{S}[[\texttt{Begin } Ss \texttt{ End}]], \kappa, \sigma, \mu &= \mathbf{Ss}[[Ss]], \kappa, \sigma, \mu \\
\\
\mathbf{Ss}[[S\ Ss]], \kappa, \sigma, \mu &= \mathbf{S}[[S]], (\mathbf{Ss}[[Ss]], \kappa, \sigma), \sigma, \mu
\end{aligned}$$

Faire un `Break`

$$\mathbf{S}[[\texttt{Break}]], \kappa, \sigma, \mu = \mu$$

Exemple :

$$\begin{aligned}
&\mathbf{Ss}[[\texttt{x :=1 ; Break ; x :=2 ;}]], \kappa, \sigma, [] \\
&\mathbf{S}[[\texttt{x :=1}]], (\mathbf{Ss}[[\texttt{Break ; x :=2 ;}]], \kappa, \sigma), \sigma, [] \\
&\mathbf{Ss}[[\texttt{Break ; x :=2 ;}]], \kappa, \sigma, [x, 1] \\
&\mathbf{S}[[\texttt{Break}]], (\mathbf{Ss}[[\texttt{x :=2}]], \kappa, \sigma), \sigma, [x, 1] \\
&[x, 1]
\end{aligned}$$

Exercice : $\mathbf{S}[[\texttt{While (x>0) do If (x=1) Then Break ; Else x :=x-1 ;}]], \kappa, \sigma, []$

Rupture et rattrapage modélisation des exceptions.

On rajoute à la syntaxe

```
...
STAT ::= ...
       | Try STAT With Id Do STAT
       | Raise Id ;
...
EXP  ::= ...
       | Try EXP With Id Do EXP
       | Raise Id
...
```

On rajoute à la sémantique

$$\begin{array}{lcl}
\mathbf{S}[[\texttt{Try } S_1 \texttt{ With } X \texttt{ Do } S_2]], \kappa, \sigma, \mu & = & \mathbf{S}[[S_1]], \kappa, (X, inCont(\lambda m.\mathbf{S}[[S_2]], \kappa, \sigma, m) : \sigma, \mu \\
\mathbf{S}[[\texttt{Raise } X]], \kappa, \sigma, \mu & = & (valOf(\sigma(x))\ \mu)
\end{array}$$

$$\begin{array}{lcl}
\mathbf{E}[[\texttt{Try } E_1 \texttt{ With } X \texttt{ Do } E_2]], \kappa, \sigma, \mu & = & \mathbf{E}[[E_1]], \kappa, (X, inCont(\lambda m.\mathbf{E}[[E_2]], \kappa, \sigma, m) : \sigma, \mu \\
\mathbf{E}[[\texttt{Raise } X]], \kappa, \sigma, \mu & = & (valOf(\sigma(x))\ \mu)
\end{array}$$

Une référence sur l'usage de la sémantique dénotationnelle.

– D. Schmidt, *Denotational Semantics : A Methodology for Language Development*, 1986, disponible à `http://www.cis.ksu.edu/~schmidt/text/densem.html`