UPMC
Master informatique 2 – STL
NI503 – Conception de langages
Notes I

Novembre 2011

1 Évaluer

Un langage

Le langage *Logo* est composé commandes permettant de diriger le déplacement d'un point sur un plan cartésien discret. Le point est donné par ses coordonnées cartésiennes en abscisse et ordonnée (c'est-à-dire, un couple (x, y)).

Un movement est défini par une direction (c'est-à-dire un angle α) et une longueur. Les commandes de mouvement sont :

- `MOVE` : accompagné d'une valeur entière k, signifie le déplacement en ligne droite du point sur une longueur k ;
- `TURN` : accompagné d'une valeur entière d, signifie que l'orientation du mouvement voit son angle augmenté de d.

Exécuter un mouvement consiste donc à modifier sa position (c'est-à-dire la valeur des coordonnées (x, y)) en fonction de la valeur α de la direction et de la longueur k du déplacement. Si l'angle α est exprimé en *radians*, on a simplement :

- x devient $x + k \sin(\alpha)$;
- y devient $y + k \cos(\alpha)$.

Si le changement de direction est exprimé en degrés, la modification de la direction α s'écrit :

- α devient $\alpha + d\frac{\pi}{180}$.

Un programme *Logo* est une suite de commandes de mouvements. Par exemple, on déplace le point selon le périmètre d'un carré de côtés de longueurs 50 en enchaînant :

```
MOVE 50
TURN 90
MOVE 50
TURN 90
MOVE 50
TURN 90
MOVE 50
```

Sa syntaxe

Lexique

- les *symboles réservés* `[ ; ]`
- les *mots clé* `MOVE TURN`

– les *constantes numériques* `-?[0-9]+` (expression régulière). On note Num.

Grammaire

PROG	::=	`[]`
	|	`[` CMDS `]`
CMDS	::=	CMD
	|	CMD `;` CMDS
CMD	::=	`MOVE` Num
	|	`TURN` Num

Le programme *Logo* ci-dessus devient :

```
[ MOVE 50; TURN 90; MOVE 50; TURN 90; MOVE 50; TURN 90; MOVE 50 ]
```

Syntaxe abstraite définition *algébrique*, *types abstraits* $Prog$ et Cmd :

$[]$	:	$Prog$
$\_ :: \_$	:	$Cmd \times Prog \to Prog$
$Move$	:	$\mathbb{N} \to Cmd$
$Turn$	:	$\mathbb{N} \to Cmd$

Exemple : syntaxe abstraite du programme ci-dessus

$$Move(50) :: Turn(90) :: Move(50) :: Turn(90) :: Move(50) :: Turn(90) :: Move(50) :: []$$

Définition alternative : $A^\star$ suites génériques, pour un ensemble A

$[]$	:	$A^\star$
$\_ :: \_$	:	$A \times A^\star \to A^\star$
$Prog$	=	$Cmd^\star$
$Move$	:	$\mathbb{N} \to Cmd$
$Turn$	:	$\mathbb{N} \to Cmd$

2 Modéliser

Le *domaine* du langage *Logo* est celui d'un plan cartésien. Le *résultat* de l'exécution d'un programme *Logo* est le déplacement sur ce plan d'un point, à partir d'une position et d'une direction initiales.

Abstraitement, un déplacement est une suite de directions et de positions ; c'est-à-dire, une suite de triplets $(\alpha, x, y) \in \mathbb{R}^3$. Un tel triplet représente *l'état* du point guidé par le programme. On peut *modéliser* l'exécution d'un programme *Logo* comme une *fonction* des programmes munis d'un état initial vers une suite d'état

$$\mathbf{P} : Prog \times \mathbb{R}^3 \to (\mathbb{R}^3)^\star$$

On définit $\mathbf{P}$ par récurrence sur la suite de commandes qu'il contient :

$$\begin{array}{lcl} \mathbf{P}([], e) & = & e :: [] \\ \mathbf{P}(Move(k) :: p, (a, x, y)) & = & e :: \mathbf{P}(p, (a, x + k\sin(a), y + k\cos(a))) \\ \mathbf{P}(Turn(d) :: p, (a, x, y)) & = & e :: \mathbf{P}(p, (a + d\frac{\pi}{180}, x, y)) \end{array}$$

Définition alternative initialisation

Trois fonctions : une pour les programmes, une pour les suites de commandes, une pour les commandes (*cf* grammaire). Signatures des fonctions :

$\mathbf{P}$	:	$Prog \to (\mathbb{R}^3)^\star$
$\mathbf{Cs}$	:	$Cmds \times \mathbb{R}^3 \to (\mathbb{R}^3)^\star$
$\mathbf{C}$	:	$Cmd \times \mathbb{R}^3 \to \mathbb{R}^3$

Équations : on choisit un état initial

$$\begin{array}{lcl}
\mathbf{P}(p) & = & \mathbf{Cs}(p, (\frac{\pi}{2}, 0, 0)) \\
\mathbf{Cs}([], e) & = & e :: [] \\
\mathbf{Cs}(c :: cs, e) & = & e :: \mathbf{Cs}(cs, \mathbf{C}(c, e)) \\
\mathbf{C}(Move(k), (a, x, y)) & = & (a, x + k\sin(a), y + k\cos(a)) \\
\mathbf{C}(Turn(d), (a, x, y)) & = & (a + d\frac{\pi}{180}, x, y)
\end{array}$$

Mathématiser On peut voir suite d'éléments pris dans un ensemble A comme un élément du produit A^n (un n-uplet). Le ≪zéro-uplet≫ est la suite vide, élément de A^0.

On peut également donner une autre vision des suites qui n'utilise que le produit cartésien entre deux ensembles (les *paires*). On pose

$$\begin{array}{lcl}
A^0 & = & \{\emptyset\} \\
A^{n+1} & = & A \times A^n \\
A^\star & = & \bigcup_{n \in I\!N} A^n
\end{array}$$

La suite $x_1, \ldots, x_n$ est le couple $(x_1, (\ldots (x_n, \emptyset) \ldots))$. Nos opérations algébriques sur les suites correspondent à des objets mathématiques

$$\begin{array}{lcl}
[] & = & \emptyset \\
x :: xs & = & (x, xs)
\end{array}$$

On peut aussi formaliser la syntaxe abstraite en termes ensemblistes. Par exemple, on *code* chacune des deux instructions `MOVE` et `TURN` par 0 et 1 respectivement. L'ensemble des expressions de la forme `MOVE` k est l'ensemble de couples $\{(0, k) \mid k \in I\!N\}$ et l'ensemble des expressions de la forme `TURN` d est l'ensemble $\{(1, d) \mid d \in I\!N\}$. L'ensemble des commandes *Logo* est l'union

$$\{(0, k) \mid k \in I\!N\} \cup \{(1, d) \mid d \in I\!N\}$$

On a construit ainsi *l'union disjointe* notée $I\!N \oplus I\!N$: c'est-à-dire, deux ≪copies≫ de $I\!N$ avec possibilité de discriminer les éléments de chacune d'elle.

On pose :

$$\begin{array}{lcl}
Cmd & = & I\!N \oplus I\!N \\
Cmds & = & Cmd^\star \\
Prog & = & Cmds
\end{array}$$

Notre programme devient alors

$$((0, 50), ((1, 90), ((0, 50), ((1, 90), ((0, 50), ((1, 90), ((0, 50), \emptyset)))))))$$

On ne s'intéressera plus aux détails de codage de la syntaxe abstraite. Savoir qu'il est possible nous suffira. On utilisera par la suite la notation entre doubles crochets ([[ ]]) pour désigner la syntaxe abstraite des commandes et des programmes. Par exemple :

- [[`MOVE` k]]
- [[`TURN` d]]
- [[c ; cs]]

Le langage des fonctions sémantiques

Nous utiliserons, pour noter les fonctions, une λ-notation : $\lambda x.e$ est la fonction de paramètre x et de corps e ; $\lambda x.\lambda y.e$ est la fonction à deux paramètres x et y, et de corps e.

La λ-notation des fonctions est à la base d'une modélisation très générale de l'écriture et l'évaluation des expressions fonctionnelles : le λ-calcul.

Les expressions, on dit les *termes* du λ-calcul (ou λ-termes) sont définis ainsi : on considère un ensemble infini dénombrable $\mathcal{X}$ de symboles dits de *variables* ; l'ensemble des termes est le plus petit ensemble qui satisfasse

1. les variables sont des termes.
2. si x est une variable et t un terme alors $\lambda x.t$ est un terme.
3. si t et u sont deux termes alors $(t\ u)$ est un terme.

Le terme $\lambda x.t$ est appelé une *abstraction*, le terme $(t\ u)$ est une *application*.

On pourra utiliser les abréviations suivantes :
- $(t\ u_1\ u_2)$ pour $((t\ u_1)\ u_2)$
- $\lambda x_1.x_2.t$ pour $\lambda x_1.\lambda x_2.t$

On pourra également mettre des parenthèses autour d'une abstraction si cela peut faciliter la lecture, sans qu'il faille y voir une application.

Outre ces constructions de base, on peut ajouter à la syntaxe des λ-termes des fonctions ou opérations pédéfinies. Par exemple $\lambda n.n^2$ est la notation de la fonction d'élévation au carré. Naturellement, on ajoutera également au λ-termes les constantes numériques. Ainsi, on pourra écrire l'application $(\lambda x.x^2\ 42)$.

α-équivalence Dans l'écriture $\lambda x.t$, la variable x est *liée* à la manière dont une variable x est liées dans la formule $\forall x.\Phi$. Les deux fonctions $\lambda x.t$ et $\lambda y.t[y/x]$ sont deux fonctions *équivalentes*
- si $t[y/x]$ est le terme obtenu en remplaçant x par y dans t ;
- et si y n'a pas d'autre occurence dans t.

β-réduction La β-réduction est la modélisation du principe d'évaluation des applications de fonctions par la substitution :
- $(\lambda x.t\ u)$ (lire : $\ll\lambda x.t$ appliqué à $u\gg$) se réduit en $t[u/x]$;
- si t se réduit en t' alors $(t\ u)$ se réduit en $(t'\ u)$;
- si u se réduit en u' alors $(t\ u)$ se réduit en $(t\ u')$;
- si t se réduit en t' alors $\lambda x.t$ se réduit en $\lambda x.t'$.

La règle principale est la première. Une application de la forme $(\lambda x.t\ u)$ est appelée un *redex*.

On notera $t \rightsquigarrow v$ pour $\ll t$ se réduit en $v\gg$

Comme dans le renommage des variables liées, il faut prendre garde, lors de la substitution d'un terme à une variable, qu'aucune des variables du terme u ne devienne liée dans le résultat de la substitution. Techniquement, on obtient ce résulat en renommant systématiquement les variables liées du terme où l'on remplace x lors du processus de susbtitution :

$$(\lambda y.t)[u/x] = \lambda z.(t[z/y][t/x])$$

en choisissant z n'ayant aucune occurence ni dans t ni dans u.

Formes spéciales, structures de contrôles Comme on s'est autorisé à pouvoir utiliser la λ-notation en y mèlant des expressions arithmétiques $(\lambda n.n^2)$, on s'autorisera à introduire d'autre opérateurs tels l'alternative, le if. Toutefois, à la différence des fonctions arithmétiques usuelles, on précise, pour l'alternative, ses règles de réduction.

$$\begin{array}{lcl} (\mathsf{if}\ true\ t_1\ t_2) & \rightsquigarrow & t_1 \\ (\mathsf{if}\ false\ t_1\ t_2) & \rightsquigarrow & t_2 \end{array}$$

Implicitement, on a donc rajouté les valeurs booléennes a nos valeurs sémantiques : $\mathbb{B} = \{true, false\}$. Mais on a rajouté plus : pour pouvoir réduire une expression alternative $(\mathsf{if}\ t\ t_1\ t_2)$, il faut avoir déterminé *au préalable* si t se réduit sur $true$ ou $false$. On a ici fixé une *stratégie d'évaluation* où l'ordre d'évaluation des arguments d'une application n'est pas indifférent : on a fixé qu'il faut évaluer le premier argument en premier. En conséquence, pour évaluer l'application $(\mathsf{if}\ t\ t_1\ t_2)$, si $t \rightsquigarrow true$, on ne cherche pas à évaluer t_2 et si $t \rightsquigarrow false$, on ne cherche pas à évaluer t_1.

Arité d'une fonction L'arité d'une fonction est le nombre d'arguments qu'elle attend. Par exemple, l'addition est d'arité 2, c'est un opérateur (fonction) binaire. Cette arité est reflétée par le type de l'addition : $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$.

Avec la λ-notation, cette notion n'est plus tout-à-fait valable : la fonction notée $\lambda x.\lambda y.x + y$ est en apparence une fonction d'arité 2, toutefois, la syntaxe du λ-calcul ne force pas l'application à 2 termes. Par exemple la terme $(\lambda x.\lambda y.x + y\ 1)$ est correctement construit, il se réduit en $\lambda y.1 + y$. Ce résultat est une *fonction* ; la fonction *successeur*. Une telle application peut-être appelée *application partielle*. Le type de $\lambda x.\lambda y.x + y$ est noté $\mathbb{N} \to (\mathbb{N} \to \mathbb{N})$ que l'on peut abréger en $\mathbb{N} \to \mathbb{N} \to \mathbb{N}$. En vérité, avec la λ-notation, toute fonction est d'arité 1.

Ajoutons la structure de couple au λ-calcul en notant (u_1, u_2) le couple formé de u_1 et u_2 ; ainsi que les deux projections fst et snd telles que $(fst\ (u_1, u_2))$ se réduit en u_1 et $(snd\ (u_1, u_2))$ se réduit en u_2. On peut alors écrire la fonction d'addition comme $\lambda c.(fst\ c) + (snd\ c)$. On n'obtiendra un résultat correct que si on l'applique à un couple : $(\lambda c.(fst\ c) + (snd\ c)\ (3, 4))$. Dans un tel cas, pour alléger l'écriture en évitant l'usage des projections, on s'autorisera à faire porter l'abstraction sur un couple en écrivant : $\lambda(x, y).x + y$.

Tests et unions disjointes Parmi les opérations primitives que l'on ajoute au λ-termes, figure les opérateurs de comparaison : égalité, ordre, etc. Pour ne pas alourdir inutilement la lecture, nous adopterons la notation infixe usuelle : $(x = 0)$. Le résultat de l'application de ces opérateurs sera une valeur booléenne : $true$ ou $false$.

3 Sémantique

La fonction **P** associe à un programme une *valeur* : la suite des points parcourus (on pourrait oublier les directions). Cette valeur est un objet mathématique défini en termes de nombres, d'ensembles, de produit cartésiens d'ensembles. C'est ce que l'on appelle la *dénotation* ou le *sens* d'un programme. Il est défini en composant le sens des commandes du programme :

- `MOVE` est la fonction qui au nombre k et à l'état (a, x, y) associe l'état $(a, x + k\sin(a), y + k\cos(a))$;
- `TURN` est la fonction qui au nombre d et à l'état (a, x, y) associe l'état $(a + d, x, y)$.

Ces fonctions sont aussi des objets mathématiques.

Équations sémantiques On reformule les signatures et définitions des fonctions d'évaluation en utilisant la notation symbolique pour la syntaxe abstraite et le typage inspiré du λ-calcul :

$$
\begin{array}{lcl}
\mathbf{P} & : & Prog \to (\mathbb{R}^3)^\star \\
\mathbf{Cs} & : & Cmds \to \mathbb{R}^3 \to (\mathbb{R}^3)^\star \\
\mathbf{C} & : & Cmd \to \mathbb{R}^3 \to \mathbb{R}^3 \\
\mathbf{P}[[\texttt{[}cs\texttt{]}]] & = & \mathbf{Cs}[[cs]](\frac{\pi}{2}, 0, 0) \\
\mathbf{Cs}[[\ \texttt{[]}\ ]]e & = & e :: [] \\
\mathbf{Cs}[[c\texttt{;}cs]]e & = & e :: \mathbf{Cs}[[cs]](\mathbf{C}[[c]]e) \\
\mathbf{C}[[\texttt{MOVE}\ k]](a, x, y) & = & (a, x + k\sin(a), y + k\cos(a)) \\
\mathbf{C}[[\texttt{TURN}\ d]](a, x, y) & = & (a + d\frac{\pi}{180}, x, y)
\end{array}
$$

Exercices

1. Redéfinir les équations sémantiques de manière à ne conserver que la trace des positions de la tortue. C'est-à-dire que la signature de la fonction d'interprétation des programmes devient :

$$\mathbf{P} : Prog \to (\mathbb{N}^2)^\star$$

2. Rajouter au langage la possibilité de gérer un ≪crayon≫ qui selon qu'il est baissé ou non produira ou ne produira pas de trace.