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

Décembre 2011

1 Procédures

On rajoute au langage *Logo* la possibilité de définir et d'appeler des procédures. Par exemple :

```
PROC carré k =
    [
      MOVE k; TURN 90;
      MOVE k; TURN 90;
      MOVE k; TURN 90;
      MOVE k
    ];

CALL carré 17;
```

La syntaxe de définition d'une procédure est donnée par

DEC	::=	PROC ident IDENTS = PROG
IDENTS	::=	ident
	|	ident IDENTS

Une procédure est un morceau de programme nommé et paramétré.

Les commandes MOVE et TURN doivent maintenant pouvoir également être utilisées avec des variables (on n'avait jusqu'à présent que des constantes numériques – num)

CMD	::=	...
	|	MOVE ident
	|	TURN ident

L'invocation d'une procédure suivra la syntaxe suivante

CMD	::=	...
	|	CALL ident NUMS
NUMS	::=	num
	|	num NUMS

Une suite de commandes peut maintenant contenir des définitions de procédures :

CMDS	::=	CMD
	|	CMD ; CMDS
	|	DEC ; CMDS

On impose ici qu'un programme ne se termine pas par une déclaration, ce qui n'est pas déraisonnable.

Modélisation

Un appel de procédure est l'activation du morceau de programme invoqué avec les valeurs que doivent prendre la paramètres. Le modèle de l'invocation d'une procédure nous est fourni par la β-réduction du λ-calcul : $(\lambda x.t\ v) \rightsquigarrow t[v/x]$. La valeur de l'application est celle de t où x «vaut» (la valeur de) v. Ainsi, pour prendre un exemple simplissime, si la fonction appliquée est l'identité $\lambda x.x$ la valeur de l'application est la valeur de l'argument appliqué v, c'est-à-dire, la valeur associée au paramètre x par le mécanisme de réduction.

Pour obtenir un tel résultat, on peut soit effectuer la substitution, soit conserver, dans une structure associative, la *liaison* entre la variable x et sa valeur v pour la quérir lorsque l'on en a besoin. Dans notre exemple *Logo*, ce sera pour exécuter les `MOVE k` de la procédure `carré`. La structure associative qui conserve la liaison entre les variables (paramètres) et leur valeur est un *environnement*. Abstraitement, un environnement est une fonction de l'ensemble des identificateurs dans les valeurs : $Id \to I\!R$.

Il nous faut deux opérations sur les environnements : l'accès à une valeur et l'ajout d'une association.

L'accés à une valeur est «gratuit» si l'on modélise les environnements par des fonctions : si ρ est un environnement et `x` un identificateur, sa valeur est simplement le résultat de l'application $\rho(\texttt{x})$. Le seul point est que cette valeur peut ne pas être définie. On gère cette situation en ajoutant à l'ensemble des valeurs un élément distingué noté $\perp$. On note $I\!R_\perp$ cet ensemble. Un environnement est donc plus précisément une fonction : $Id \to I\!R_\perp$.

L'ajout d'une association entre un identificateur `x` et une valeur v à un environnement ρ est la fonction ρ' telle que

- $\rho'(\texttt{x}) = v$ et
- $\rho'(\texttt{y}) = \rho(\texttt{y})$ si $\texttt{x} \neq \texttt{y}$.

En utilisant la λ-notation, c'est la fonction $\lambda y.(\mathsf{if}\ (y = \texttt{x})\ v\ (\rho\ y))$. On notera $\rho[\texttt{x} := v]$.

Posons : $Env = Id \to I\!R_\perp$

Valeur sémantique d'une procédure Une procédure est un morceau de programme. Son résulat est donc analogue à celui des programmes : une suite états qu'elle calcule à partir de la valeur de l'état au moment de son appel. Également, une procédure est un morceau de programme paramètré qui est exécuté compte tenu des valeurs de ses paramètres d'appel. Une procédure est donc *fonction* de ces paramètres et de l'état courant. Par exemple, notre procédure `carré` est une fonction : $I\!R \to I\!R^3 \to (I\!R^3)^\star$.

Posons :

- $FProc_1 = I\!R \to I\!R^3 \to (I\!R^3)^\star$
- $FProc_{n+1} = I\!R \to FProc_n$
- $FProc = \bigcup_n FProc_n$

De manière générale, la valeur sémantique d'une procédure est une fonction de $FProc$.

La valeur sémantique des procédures est associée à son nom. Posons $EnvProc = Ident \to FProc_\perp$.

Sémantique

Avec l'introduction des procédures, il faut introduire la fonction sémantique d'interprétation de leurs déclarations. Il faut également ajouter la structure d'association entre nom de procédure et fonction sémantique associée. Enfin, il faut également ajouter la structure d'environnement pour les valeurs d'appel des procédures.

Résumons donc le langage que nous devons interpréter :

```
PROG    ::=  []
          |  [ CMDS ]

CMDS    ::=  CMD
          |  CMD ; CMDS
          |  DEC ; CMDS

CMD     ::=  MOVE ARG
          |  TURN ARG
          |  CALL ident ARGS

DEC     ::=  PROC ident IDENTS = PROG

IDENTS  ::=  ident
          |  ident IDENTS

ARG     ::=  ident
          |  num
ARGS    ::=  ARG
          |  ARG ARGS
```

On ajoute une nouvelle fonction sémantique **D** pour les définitions, ainsi qu'une fonction **A** pour l'évaluation des arguments. Parmi les commandes, on trouve maintenant les appels de procédures qui produisent non pas simplement un état, mais une suite d'états. Il faut donc modifier le type de la valeur de retour de la fonction sémantique **C** qui devient $(\mathbb{R}^3)^\star$. Comme le résultat de **C** est argument de la fonction **Cs**, il faut également que l'état que nous passions en paramètre à cette fonction soit modifié et devienne lui aussi une suite d'états. Il en va de même pour **C**. Le dernier état calculé est en tête de liste.

Les signatures des fonctions sémantiques deviennent donc :

$$
\begin{array}{lcl}
\mathbf{P} & : & Prog \to (\mathbb{R}^3)^\star \\
\mathbf{Cs} & : & Cmds \to (\mathbb{R}^3)^\star \to Env \to EnvProc \to (\mathbb{R}^3)^\star \\
\mathbf{C} & : & Cmd \to (\mathbb{R}^3)^\star \to Env \to EnvProc \to (\mathbb{R}^3)^* \\
\mathbf{D} & : & Dec \to Env \to EnvProc \to EnvProc \\
\mathbf{A} & : & Arg \to Env \to \mathbb{R}
\end{array}
$$

Équations sémantiques avec $cs \in Cmds$, $c \in Cmd$, $d \in Dec$, $z \in Arg$, $e \in \mathbb{R}^3$, $\rho_v \in Env$ et $\rho_p \in EnvProc$.

$$
\begin{array}{lcl}
\mathbf{P}[[\ \texttt{[}\ cs\ \texttt{]}\ ]] & = & \mathbf{Cs}[[cs]]\{(\frac{\pi}{2}, 0, 0)\} \\
\mathbf{Cs}[[\ []\ ]]es\ \rho_v\ \rho_p & = & es \\
\mathbf{Cs}[[d\ \texttt{;}\ cs]]es\ \rho_v\ \rho_p & = & \mathbf{Cs}[[cs]]es\ \rho_v\ \rho'_p \\
 & & \text{avec } \rho'_p = \mathbf{D}[[d]]\ \rho_v\ \rho_p \\
\mathbf{Cs}[[c\ \texttt{;}\ cs]]es\ \rho_v\ \rho_p & = & (\mathbf{Cs}[[cs]]es'\ \rho_v\ \rho_p) \\
 & & \text{avec } es' = \mathbf{C}[[c]]es\ \rho_v\ \rho_p \\
\mathbf{D}[[\texttt{PROC}\ f\ x_1 \ldots x_n\ \texttt{=}\ cs]]\ \rho_v\ \rho_p & = & \rho_p[f := \lambda v_1 \ldots v_n.\lambda es.(\mathbf{Cs}[[cs]]es\ \rho'_v\ \rho_p)] \\
 & & \text{avec } \rho'_v = \rho_v[x_1 := v_1; \ldots; x_n := v_n] \\
\mathbf{C}[[\texttt{MOVE}\ z]]((a, x, y) :: es)\ \rho_v\ \rho_p & = & (a, x + k\sin(a), y + k\cos(a)) :: (a, x, y) :: es \\
 & & \text{avec } k = \mathbf{A}[[z]]\rho_v \\
\mathbf{C}[[\texttt{TURN}\ z]]((a, x, y) :: es)\ \rho_v\ \rho_p & = & (a + d\frac{\pi}{180}, x, y) :: (a, x, y) :: es \\
 & & \text{avec } d = \mathbf{A}[[z]]\rho_v \\
\mathbf{C}[[\texttt{CALL}\ f\ z_1 \ldots z_n]]es\ \rho_v\ \rho_p & = & (\rho_p\ f)\ r_1 \ldots r_n\ es \\
 & & \text{avec } r_1 \ldots r_n = (\mathbf{A}[[z_1]]\rho_v) \ldots (\mathbf{A}[[z_n]]\rho_v) \\
\mathbf{A}[[s]]\rho_v & = & (\rho_v\ s) \\
\mathbf{A}[[n]]\rho_v & = & n
\end{array}
$$

Exemple :

```
[
  PROC rect h w  =
    [
      MOVE h; TURN 90;
      MOVE w; TURN 90;
      MOVE h; TURN 90;
      MOVE w
    ];
  PROC carré w =
    [
      CALL rect w w;
    ];
  CALL carré 100;
  CALL carré 75;
  CALL carré 50;
  CALL carré 25;
]
```

$\mathbf{P}$[[`PROC rect h w = [...]; PROC carré w = [...]; CALL carré ...`]]

$= \mathbf{Cs}$[[`PROC rect h w = [...]; PROC carré w = [...]; CALL carré ...`]]$\{(\frac{\pi}{2},0,0)\}\ \emptyset\ \emptyset$

$= \mathbf{Cs}$[[`PROC carré w = [...]; CALL carré ...`]]$\{(\frac{\pi}{2},0,0)\}\ \emptyset\ \rho_p^1$
avec $\rho_p^1 = [\texttt{rect} := \lambda v_1.v_2.\lambda es.(\mathbf{Cs}[[\texttt{[...]}]]es\ [\texttt{h} := v_1; \texttt{w} := v_2]\ \emptyset)]$

$= \mathbf{Cs}$[[`CALL carré 100; ...`]]$\{(\frac{\pi}{2},0,0)\}\ \emptyset\ \rho_p^2$
avec $\rho_p^2 = \rho_p^1[\texttt{carré} := \lambda v_1.\lambda es.(\mathbf{Cs}[[\texttt{[...]}]]es\ [\texttt{w} := v_1]\ \rho_p^1)]$

$= \mathbf{Cs}$[[`CALL carré 75; ...`]]$es_1\ \emptyset\ \rho_p^2$
avec $es_1 = \mathbf{C}$[[`CALL carré 100`]]$\{(\frac{\pi}{2},0,0)\}\ \emptyset\ \rho_p^2$
$= ((\lambda v_1.\lambda es.(\mathbf{Cs}[[\texttt{[CALL rect w w]}]]es\ [\texttt{w} := v_1]\ \rho_p^1))\ (\mathbf{A}[[\texttt{w}]][\texttt{w} := 100])\ \{(\frac{\pi}{2},0,0)\})$
$= ((\lambda v_1.\lambda es.(\mathbf{Cs}[[\texttt{[CALL rect w w]}]]es\ [\texttt{w} := v_1]\ \rho_p^1))\ 100\ \{(\frac{\pi}{2},0,0)\})$
$= \mathbf{Cs}$[[`[CALL rect w w]`]]$\{(\frac{\pi}{2},0,0)\}\ [\texttt{w} := 100]\ \rho_p^1$
$= ((\lambda v_1.v_2.\lambda es.(\mathbf{Cs}[[\texttt{[...]}]]es\ [\texttt{h} := v_1; \texttt{w} := v_2]\ \emptyset))\ (\mathbf{A}[[\texttt{w}]][\texttt{w} := 100])\ (\mathbf{A}[[\texttt{w}]][\texttt{w} := 100])\ \{(\frac{\pi}{2},0,0)\})$
$= ((\lambda v_1.v_2.\lambda es.(\mathbf{Cs}[[\texttt{[...]}]]es\ [\texttt{h} := v_1; \texttt{w} := v_2]\ \emptyset))\ 100\ 100\ \{(\frac{\pi}{2},0,0)\})$
$= \mathbf{Cs}$[[`MOVE h; TURN 90; ...`]]$\{(\frac{\pi}{2},0,0)\}\ [\texttt{h} := 100; \texttt{w} := 100]\ \emptyset$
$= \mathbf{Cs}$[[`TURN 90;...`]]$es_{11}\ [\texttt{h} := 100; \texttt{w} := 100]\ \emptyset$
avec $es_{11} = \mathbf{C}$[[`MOVE h`]]$\{(\frac{\pi}{2},0,0)\}\ [\texttt{h} := 100; \texttt{w} := 100]\ \emptyset$
$= \{(\frac{\pi}{2}, 0 + 100\sin(\frac{\pi}{2}), 0 + 100\cos(\frac{\pi}{2})); (\frac{\pi}{2},0,0)\}$
$= \{(\frac{\pi}{2},100,0); (\frac{\pi}{2},0,0)\}$
$= \ldots$
$= \{(2\pi,0,0); (2\pi,-100,0); (\frac{2\pi}{3},-100,0); (\frac{2\pi}{3},-100,100); (\pi,-100,100); (\pi,0,100); (\frac{\pi}{2},0,100); (\frac{\pi}{2},0,0)\}$
$= \mathbf{Cs}$[[`...`]]$es_2\ \emptyset\ \rho_p^2$
avec $es_2 = \mathbf{C}$[[`CALL carré 75`]]$es_1\ \emptyset\ \rho_p^2$
$= \ldots$

2 Expressions

Arithmétique, comparaison, opérateurs booléens avec leur syntaxe infixe usuelle.

```
EXPR  ::=  ident

       |   true
       |   false
       |   not EXPR
       |   EXPR and EXPR
       |   EXPR or EXPR

       |   EXPR = EXPR
       |   EXPR < EXPR

       |   num
       |   EXPR + EXPR
       |   EXPR - EXPR
       |   EXPR * EXPR
       |   EXPR / EXPR
       |   ( EXPR )
```

La sémantique des expressions est quasiment tautologique : l'addition (symbole `+`) est interprétée par l'addition (que l'on écrit $+$) ; le test d'égalité (symbole `=`) par l'égalité (que l'n écrit $=$), etc.

$\mathbf{E} \;:\; \mathit{Expr} \to \mathit{Env} \to I\!R$

$$
\begin{array}{lcl}
\mathbf{E}[[x]]\ \rho_v & = & (\rho_v\ x) \\
\mathbf{E}[[\texttt{true}]]\ \rho_v & = & 1 \\
\mathbf{E}[[\texttt{false}]]\ \rho_v & = & 0 \\
\mathbf{E}[[\texttt{not}\ e]]\ \rho_v & = & (\mathsf{if}\ ((\mathbf{E}[[e]]\ \rho_v) = 0)\ 1\ 0) \\
\mathbf{E}[[e_1\ \texttt{and}\ e_2]]\ \rho_v & = & (\mathsf{if}\ ((\mathbf{E}[[e_1]]\ \rho_v) = 0)\ 0\ (\mathbf{E}[[e_2]]\ \rho_v)) \\
\mathbf{E}[[e_1\ \texttt{or}\ e_2]]\ \rho_v & = & (\mathsf{if}\ ((\mathbf{E}[[e_1]]\ \rho_v) = 0)\ (\mathbf{E}[[e_2]]\ \rho_v)\ 1) \\
 & & \\
\mathbf{E}[[e_1\ \texttt{=}\ e_2]]\ \rho_v & = & (\mathsf{if}\ ((\mathbf{E}[[e_1]]\ \rho_v) = (\mathbf{E}[[e_2]]\ \rho_v)\ 1\ 0) \\
\mathbf{E}[[e_1\ \texttt{<}\ e_2]]\ \rho_v & = & (\mathsf{if}\ ((\mathbf{E}[[e_1]]\ \rho_v) < (\mathbf{E}[[e_2]]\ \rho_v)\ 1\ 0) \\
 & & \\
\mathbf{E}[[n]]\ \rho_v & = & n \\
\mathbf{E}[[e_1\ \texttt{+}\ e_2]]\ \rho_v & = & (\mathbf{E}[[e_1]]\ \rho_v) + (\mathbf{E}[[e_2]]\ \rho_v) \\
\mathbf{E}[[e_1\ \texttt{-}\ e_2]]\ \rho_v & = & (\mathbf{E}[[e_1]]\ \rho_v) - (\mathbf{E}[[e_2]]\ \rho_v) \\
\mathbf{E}[[e_1\ \texttt{*}\ e_2]]\ \rho_v & = & (\mathbf{E}[[e_1]]\ \rho_v) \times (\mathbf{E}[[e_2]]\ \rho_v) \\
\mathbf{E}[[e_1\ \texttt{/}\ e_2]]\ \rho_v & = & \frac{(\mathbf{E}[[e_1]]\ \rho_v)}{(\mathbf{E}[[e_2]]\ \rho_v)}
\end{array}
$$

Les expressions introduisent la possibilité de désigner une valeur numérique plus complexe qu'avec une simple constante numérique ou un identificateur. Là où nous attendions la désignation d'une valeur numérique, nous pouvons donc utiliser une expression. Dans notre langage, cela concerne les primitves `MOVE` et `TURN` aisni que les paramètres d'appel de `CALL`. On peut en conséquence modifier la syntaxe des commandes :

```
CMD    ::=  MOVE EXPR
         |  TURN EXPR
         |  CALL ident EXPRS

EXPRS  ::=  EXPR
         |  EXPR EXPRS
```

Le non-terminal ARG et son *alter ego* ARGS sont devenus caduques.

Au niveau sémantique, la fonction $\mathbf{A}[[]]$ d'évaluation des arguments est remplacée par $\mathbf{E}[[]]$:

$$
\begin{array}{lcl}
\mathbf{C}[[\texttt{MOVE } e]](a,x,y) :: es\ \rho_v\ \rho_p & = & (a, x + k\sin(a), y + k\cos(a)) :: (a,x,y) :: es \\
 & & \text{avec } k = \mathbf{E}[[e]]\rho_v \\
\mathbf{C}[[\texttt{TURN } e]](a,x,y) :: es\ \rho_v\ \rho_p & = & (a + d\frac{\pi}{180}, x, y) :: (a,x,y) :: es \\
 & & \text{avec } d = \mathbf{E}[[e]]\rho_v \\
\mathbf{C}[[\texttt{CALL } f\ e_1 \dots e_n]]es\ \rho_v\ \rho_p & = & (\rho_p\ f) r_1 \dots r_n\ es\ \rho_v\ \rho_p \\
 & & \text{avec } r_1 \dots r_n = (\mathbf{E}[[e_1]]\rho_v) \dots (\mathbf{E}[[e_n]]\rho_v)
\end{array}
$$

3 Alternative

L'alternative *if-then-else* est la structure de contrôle obligée de tout langage de programmation. Ne nous en privons donc pas.

Syntaxe

CMD ::= ...
| IF EXPR PROG PROG

Sémantique

$$
\begin{array}{lcl}
\mathbf{C}[[\texttt{IF } t\ p_1\ p_2]]es\ \rho_v\ \rho_p & = & (\mathsf{if}\ ((\mathbf{E}[[t]]\ \rho_v) = 0) \\
 & & (\mathbf{P}[[p_2]]es\ \rho_v\ \rho_p) \\
 & & (\mathbf{P}[[p_1]]es\ \rho_v\ \rho_p))
\end{array}
$$

4 Procédures récursives

Avec l'alternative, il est possible d'envisager l'introduction dans le langage la définition de fonctions récursives. Nous aurions pu le faire avant, mais une procédure récursive sans possibilité d'exprimer un test d'arrêt eût été d'un intérêt assez limité.

Rappelons la sémantique des déclarations de procédures :

$$
\begin{array}{lcl}
\mathbf{D}[[\texttt{PROC } f\ x_1 \dots x_n\ \texttt{=}\ cs]]\ \rho_v\ \rho_p & = & \rho_p[f := \lambda v_1 \dots v_n.\lambda es.(\mathbf{Cs}[[cs]]es\ \rho'_v\ \rho_p)] \\
 & & \text{avec } \rho'_v = \rho_v[x_1 := v_1; \dots; x_n := v_n]
\end{array}
$$

Le corps $(\mathbf{Cs}[[cs]]es\ \rho'_v\ \rho_p$ de la fonction associée à l'identificateur f activera l'évaluation des commandes cs de f dans l'environnement de procédures ρ_p. Si donc, figure dans les commandes cs un appel récursif à f, la valuer associée à f ne pourra être trouvée. Le modèle actuel ne convient pas aux procédures récursives.

On peut palier ce problème de deux manières :

1. modifier notre gestion des environnements de procédures de manière à pouvoir disposer, au moment de l'appel d'une procédure, de la bonne référence.
2. modifier l'interprétation des procédures elles-mêmes de manière à ce qu'elles sachent gérer la récurrence.

Environnements Dans la version actuelle de notre sémantique, lorsque l'on crée la valeur d'une procédure (au moment de sa déclaration), on *capture* dans celle-ci l'état de l'environnement tel qu'il est ≪juste avant≫ la déclaration. Ensuite, lorsque la procédure est invoquée, c'est cet environnement qui est utilisé pour l'évaluation de la suite de commandes de la procédure. Dans ce cas, on dit que l'on utilise des *environnements statiques*. La valeur d'une procédure est une structure englobant le code à exécuter et son environnement ; ce que l'on appelle une *fermeture*.

Si l'on procède différemment, c'est-à-dire, si au moment de l'appel d'une procédure on ne restaure pas l'environnement qui existait au moment de sa déclaration, mais qu'au contraire, on utilise l'environnement qui existe au moment de l'appel, alors on trouvera dans celui-ci une référence à la procédure appelée. Faisant cela, on utilise des *environnements dynamiques* : ils varient possiblement selon les divers moments des appels.

Au niveau sémantique, il est très facile de modéliser cela. La valeur d'une procédure est une fonction qui aura également comme paramètre une environnement ρ_p de procédure (il n'est donc pas encore fixé) :

$$\mathbf{D}[[\texttt{PROC } f\ x_1 \ldots x_n \texttt{ = } cs]]\ \rho_v\ \rho_p \quad = \quad \rho_p[f := \lambda v_1 \ldots v_n.\lambda es.\lambda\rho'_p(\mathbf{Cs}[[cs]]es\ \rho'_v\ \rho'_p)] \text{ avec } \rho'_v = \rho_v[x_1 := v_1; \ldots; x_n := v_n]$$

L'appel d'une procédure utilise l'environnement présent au moment de l'appel pour fournir l'argument correspondant de la valeur de la procédure :

$$\mathbf{C}[[\texttt{CALL } f\ e_1 \ldots e_n]]es\ \rho_v\ \rho_p \quad = \quad (\rho_p\ f)\ r_1 \ldots r_n\ es\ \rho_p \text{ avec } r_1 \ldots r_n = (\mathbf{E}[[e_1]]\rho_v) \ldots (\mathbf{E}[[e_n]]\rho_v)$$

Exemple idiot :

```
PROC f x = [ CALL f x ];
CALL f 10;
```

$\mathbf{P}[[\texttt{PROC f x = [CALL f x]; CALL f 10}]]$
$= \mathbf{Cs}[[\texttt{PROC f x = [CALL f x]; CALL f 10}]]\{(\frac{\pi}{2}, 0, 0)\}\ \emptyset\ \emptyset$
$= \mathbf{Cs}[[\texttt{CALL f 10}]]\{(\frac{\pi}{2}, 0, 0)\}\ \emptyset\ \rho_p^1$
avec $\rho_p^1 = [\texttt{f} := \lambda v.\lambda es.\lambda\rho_p.(\mathbf{Cs}[[\texttt{CALL f x}]]es\ [\texttt{x} := v]\ \rho_p)]$
$= ((\lambda v.\lambda es.\lambda\rho_p.(\mathbf{Cs}[[\texttt{CALL f x}]]es\ [\texttt{x} := v]\ \rho_p))\ (\mathbf{E}[[\texttt{10}]]\emptyset\ \rho_p^1)\ \{(\frac{\pi}{2}, 0, 0)\}\ \rho_p^1$
$= ((\lambda v.\lambda es.\lambda\rho_p.(\mathbf{Cs}[[\texttt{CALL f x}]]es\ [\texttt{x} := v]\ \rho_p))\ 10\ \{(\frac{\pi}{2}, 0, 0)\}\ \rho_p^1$
$= \mathbf{Cs}[[\texttt{CALL f x}]]\{(\frac{\pi}{2}, 0, 0)\}\ [\texttt{x} := 10]\ \rho_p^1$
$= ((\lambda v.\lambda es.\lambda\rho_p.(\mathbf{Cs}[[\texttt{CALL f x}]]es\ [\texttt{x} := v]\ \rho_p))\ (\mathbf{E}[[\texttt{x}]][\texttt{x} := 10])\ \{(\frac{\pi}{2}, 0, 0)\}\ \rho_p^1$
$= ((\lambda v.\lambda es.\lambda\rho_p.(\mathbf{Cs}[[\texttt{CALL f x}]]es\ [\texttt{x} := v]\ \rho_p))\ 10\ \{(\frac{\pi}{2}, 0, 0)\}\ \rho_p^1$
$= \mathbf{Cs}[[\texttt{CALL f x}]]\{(\frac{\pi}{2}, 0, 0)\}\ [\texttt{x} := 10]\ \rho_p^1$

Fonctions récursives La seconde solution consiste donc à conserver des environnements de procédures statiques en associant aux procédures des valeurs récursives de fonction. Pour obtenir *l'effet d'itération* des fonction récursive en λ-calcul on utilise un *combinateur de point fixe*. On écrit $!f.t$ où f est une variable. On pose

$$!f.t \text{ se réduit en } t[!f.t/f]$$

Avec ce combinateur, on peut, par exemple exprimer la fonction factorielle

$$fac = !f.\lambda n.(\mathsf{if}\ (n = 0)\ 1\ (n \times (f\ (n - 1))))$$

Notez que cette définition *n'est pas récursive*.

On vérifie que cela fonctionne correctement :

$$\begin{array}{lcll}
(fac\ 0) & = & (!f.\lambda n.(\mathsf{if}\ (n = 0)\ 1\ (n \times (f\ (n - 1))))\ 0) & \textit{expansion} \\
& \rightsquigarrow & (\lambda n.(\mathsf{if}\ (n = 0)\ 1\ (n \times (fac\ (n - 1))))\ 0) & \textit{réduction du !f} \\
& \rightsquigarrow & (\mathsf{if}\ (0 = 0)\ 1\ (0 \times (fac\ (0 - 1)))) & \textit{réduction du } \lambda n \\
& \rightsquigarrow & (\mathsf{if}\ \mathit{true}\ 1\ (0 \times (fac\ (0 - 1)))) & \\
& \rightsquigarrow & 1 &
\end{array}$$

$$\begin{array}{lcll}
(fac\ 1) & \rightsquigarrow & (\mathsf{if}\ (1 = 0)\ 1\ (1 \times (fac\ (1 - 1)))) & \textit{expansion, réductions de !f et } \lambda n \\
& \rightsquigarrow & (\mathsf{if}\ \mathit{false}\ 1\ (1 \times (fac\ (1 - 1)))) & \\
& \rightsquigarrow & (1 \times (fac\ (1 - 1))) & \\
& \rightsquigarrow & (1 \times (fac\ 0)) & \\
& \rightsquigarrow & 1 \times 1 & \\
& \rightsquigarrow & 1 &
\end{array}$$

Au niveau sémantique, les définitions récursives auront un traitement particulier. Nous leur donnerons donc une syntaxe distinguée.

DEC ::= ...
| PROCREC ident IDENTS = PROG

La sémantique de déclaration d'une procédure récursive ne demande qu'un léger changement : un point fixe qui permet de lier récursivement le nom de la procédure à sa valeur au moment de l'appel

$$\mathbf{D}[[\texttt{PROCREC } f\ x_1 \ldots x_n\ \texttt{=}\ cs]]\ \rho_v\ \rho_p \quad = \quad \rho_p[f := !w.\lambda v_1 \ldots v_n.\lambda es.(\mathbf{Cs}[[cs]]es\ \rho'_v\ \rho'_p)]$$
avec $\rho'_v = \rho_v[x_1 := v_1; \ldots; x_n := v_n]$
et $\rho'_p = \rho_p[f := w]$

Pour reprendre l'exemple idiot :

```
PROCREC f x = [ CALL f x ];
CALL f 10;
```

$\mathbf{P}[[\texttt{PROCREC f x = [CALL f x]; CALL f 10}]]$
$= \mathbf{Cs}[[\texttt{PROCREC f x = [CALL f x]; CALL f 10}]]\{(\frac{\pi}{2}, 0, 0)\}\ \emptyset\ \emptyset$
$= \mathbf{Cs}[[\texttt{CALL f 10}]]\{(\frac{\pi}{2}, 0, 0)\}\ \emptyset\ \rho_p^1$
avec $\rho_p^1 = [\texttt{f} := !w.\lambda v.\lambda es.(\mathbf{Cs}[[\texttt{CALL f x}]]es\ [\texttt{x} := v][f := w])]$
$= ((!w.\lambda v.\lambda es.(\mathbf{Cs}[[\texttt{CALL f x}]]es\ [\texttt{x} := v]\ [f := w]))\ (\mathbf{E}[[\texttt{10}]]\emptyset)\ \{(\frac{\pi}{2}, 0, 0)\}\ \emptyset\ \rho_p^1$
$= ((!w.\lambda v.\lambda es.(\mathbf{Cs}[[\texttt{CALL f x}]]es\ [\texttt{x} := v]\ [f := w]))\ 10\ \{(\frac{\pi}{2}, 0, 0)\}\ \emptyset$
$= ((\lambda v.\lambda es.(\mathbf{Cs}[[\texttt{CALL f x}]]es\ [\texttt{x} := v]\ [f := !w. \ldots])\ 10\ \{(\frac{\pi}{2}, 0, 0)\}\ \emptyset$
$= \mathbf{Cs}[[\texttt{CALL f x}]]\{(\frac{\pi}{2}, 0, 0)\}\ [\texttt{x} := 10]\ [f := !w. \ldots]$
$= etc.$