IN 213 **Solution**

ENSTA IP PARIS

Sémantique opérationnelle

Réalisation d'un interprète du langage PCF

Introduction

Le but de ce TP est de programmer la partie sémantique d'un interprète du langage PCF du TP précédent. Les principaux composants du programme, qui ont formé le corrigé du TP précédent, vous sont donnés : analyseurs lexical et syntaxique, boucle principale, ainsi que quelques éléments de la partie sémantique.

Les fichiers desquels vous devez partir sont les suivants :

— `pcflex.mll` : l'analyseur lexical
— `pcfast.ml` : le type des arbres de syntaxe abstraite, et leur imprimeur (*pretty-printer*)
— `pcfparse.mly` : l'analyseur syntaxique
— `pcfsem.ml` : (nouveau) des éléments de sémantique opérationnelle du langage :
  — le type `pcfval` des valeurs sémantiques
  — des indications relatives aux différents éléments à produire (environnements, fonction sémantique).
— `pcfloop.ml` : la boucle principale.

Compiler et recompiler

La commande `make` permet de reconstruire l'exécutable `pcfloop`, après chacune de vos modifications.

Note : La toute première fois, après récupération des fichiers, vous devrez créer un fichier (vide) `.depend` dans le répertoire où vous aurez mis les fichiers. Cela permettra une gestion automatisée des dépendances de compilation (quoi recompiler quand un fichier a été modifié). Vous invoquerez alors `make depend` afin que ces dépendances soient calculées initialement. Lorsque vous modifierez vos fichiers, si vous utilisez une fonction d'une unité de compilation non encore utilisée dans un fichier, la compilation pourra échouer car les dépendances entre fichiers ne seront peut-être pas à jour (en fait, c'est très peu probable car le squelette impose déjà les dépendances nécessaires). Dans ce cas, re-invoquez `make depend`.

1 Traitement des constantes et des opérateurs

Q1 Complétez `pcfsem.ml` de sorte que le traitement des constantes entières (`PCFast.EInt _`), booléennes (`EBool _`), de la conditionnelle (`PCFast.EIf(_, _, _)`), et des opérateurs unaires et binaires soit effectif.

La fonction sémantique `eval` devra garder son paramètre `rho` même s'il n'est jamais utilisé à ce stade (il le sera à partir de la question suivante).

À chaque ajout, recompilez et testez.

Solution

```
let rec eval e rho =
  match e with
  | EInt n →  Intval n
```

```
  | EBool b → Boolval b
  | EString s → Stringval s
  | EMonop ("-", e) → (
      match eval e rho with
      | Intval n → Intval (−n)
      | _ → error "Opposite of a non-integer"
    )
  | EMonop (op, _) → error (Printf.sprintf "Unknown unary op: %s" op)
  | EBinop (op, e1, e2) → (
      match (op, eval e1 rho, eval e2 rho) with
      | ("+", Intval n1, Intval n2) → Intval (n1 + n2)
...
      | (("+"|"-"|"*"|"/"), _, _) →
          error "Arithmetic on non-integers"
      | ("<", Intval n1, Intval n2) → Boolval (n1 < n2)
...
      | (("<"|">"|"="|"≤"|"≥"), _, _) →
          error "Comparison of non-integers"
      | _ → error (Printf.sprintf "Unknown binary op: %s" op)
    )
;;
```

2 Traitement des identificateurs

On traitera les environnements (`rho`) comme des listes des couples (*identificateur*, *valeur*). L'environnement initial (vide) est la liste vide, et la fonction `lookup` produit la valeur associée à un identificateur, lorsque c'est possible.

Q2 Testez les fonctions `lookup` et `extend` sur de petits exemples. (Demandez à OCaml quel est leur type.)

Q3 Complétez le traitement des identificateurs et du `let` dans la fonction `eval`.

Solution

```
let rec eval e rho =
  match e with
...
  | ELet (x, e1, e2) →
      let v1 = eval e1 rho in
      let rho1 = extend rho x v1 in
      eval e2 rho1
 | EIdent v → lookup v rho
;;
```

3 Fonctions et applications

Q4 Les valeurs sémantiques des fonctions de PCF sont essentiellement celles décrites durant le cours. Complétez le traitement des fonctions et des applications dans la fonction `eval`.

Solution

```
let rec eval e rho =
  match e with
...
  | EFun (a, e) →  Funval { param = a; body = e; env = rho }
  | EApp (e1, e2) →  (
      match (eval e1 rho, eval e2 rho) with
      | (Funval { param; body; env }, v2) →
          let rho1 = extend env param v2 in
          eval body rho1
      | (Funrecval { fname; param; body; env } as fval, v2) →
          let rho1 = extend env fname fval in
          let rho2 = extend rho1 param v2 in
          eval body rho2
      | (_, _) →  error "Apply a non-function"
    )
;;
```

4 Déclarations récursives

La syntaxe des définitions récursives permet de définir des **fonctions** récursives locales. Vous avez vu, durant le cours, qu'une fonction récursive disposait de son propre nom qu'elle associait à sa propre valeur lors de l'évaluation de chacun de ses appels.

Q5 Complétez votre fonction sémantique en donnant la sémantique de ces fonctions récursives locales.

Solution

```
let rec eval e rho =
  match e with
...
  | ELetrec (f, x, e1, e2) →
      let fval = Funrecval { fname = f; param = x; body = e1; env = rho } in
      let rho1 = extend rho f fval in
      eval e2 rho1
;;
```

5 S'il vous reste du temps ...

... enrichissez votre langage à votre gré : traitement des chaînes de caractères, environnement primitif plus riche, définitions globales, *etc.*