LU2IN119 2022-2023

Projet de programmation fonctionnelle

Le but de ce problème est de réaliser un interprète du langage **PF23***, langage à pile à la PostScript.*

Description du langage

PF23 est un langage à pile simple où l'on peut définir de nouveaux opérateurs (ou mots). Les arguments d'un opérateur sont empilés dans la pile des données. Ils seront recherchés dans la pile par l'opérateur au moment de son exécution.

Exemple

L'expression `2+(3*4)` qui s'écrit en polonaise inversée en `2 3 4 * +` est considérée comme une suite de mots dont l'évaluation agit sur la pile de données de la manière suivante :

— d'empiler successivement les valeurs `2`, `3`, puis `4`,
— de remplacer ensuite les 2 nombres du sommet de la pile (`3` et `4`) par leur produit `12`,
— et enfin de remplacer les 2 nombres en haut de pile (`2` et `12`) par leur somme `14`.

Opérateurs

PF23 fournit des opérateurs de pile, des opérateurs arithmétiques et des opérateurs de comparaison.

Les règles suivantes, de la forme `OP (pile1 -- pile2)`, spécifient l'effet qu'a un opérateur `OP` sur la pile `pile1`. Après l'évaluation de `OP`, la pile `pile1` devient `pile2`. Par convention la pile croît vers la droite.

opérateurs de pile

— `DUP (n -- n n)` (dupliquer le sommet de pile)
— `DROP ( n -- )` (supprimer le sommet de pile)
— `SWAP ( a b -- b a)` (échanger des deux éléments du sommet)
— `ROT (a b c -- b c a)` (rotation des trois éléments du sommet)

opérateurs arithmétiques et de comparaison

— `*`, `/`, `+`, `-` `(a b -- op(a,b))`, où `op(a,b)` dénote un entier.
— `=`, `<>`, `<`, `>` `(a b -- op(a,b))`, où `op(a,b)` dénote un booléen (`TRUE` ou `FALSE`).

Les arguments sont d'abord évalués et placés dans la pile, puis l'opérateur est évalué immédiatement avec ses arguments, puis le résultat est placé en sommet de pile.

L'opérateur `<>` est l'opérateur *non-égal*.

Une erreur se produit s'il n'y a pas assez d'arguments sur la pile, ou en cas de division par zéro.

Exemple 1 :
`1 DUP +` (on obtient `2` en sommet de la pile)
Exemple 2 :
`1 2 + 10 *` (on obtient `30` en sommet de la pile)
Exemple 3 :
`1 2 + 5 <` (on obtient `TRUE` en sommet de la pile)

Définitions

Le programmeur construit le vocabulaire de son application en définissant (avec des noms `<var>`) ses propres mots. La définition d'un nouveau mot est introduite par l'opérateur "`:`" (*deux-points*) et se termine par "`;`" (*point virgule*) comme décrit dans la grammaire suivante où `<liste_de_mots>` correspond à une suite possiblement vide de mots :

```
définition := : <var> <liste_de_mots> ;
```

L'appel à un mot défini x exécute la liste de mots associée à x dans le dictionnaire. La définition de mots en **PF23** est l'équivalent des sous-programmes, fonctions ou procédures dans les autres langages.

Les arguments sont d'abord évalués et placés dans la pile, comme pour les opérateurs arithmétiques et de comparaisons.

Exemple 4 : le programme **PF23** suivant définit une fonction `CARRE` et l'appelle :

```
: CARRE DUP * ;    (définition de CARRE)
11 CARRE           (on obtient 121 en sommet de la pile)
```

Exemple 5 : le programme **PF23** suivant définit une fonction `CARRE`, puis une fonction `CUBE` qui appelle `CARRE`, puis appelle `CUBE` avec l'argument `2` :

```
: CARRE DUP * ;          (définition de CARRE)
: CUBE DUP CARRE * ;     (usage de CARRE dans une définition)
1 1 + CUBE               (on obtient 8 en sommet de la pile)
```

Exemple 6 : une difficulté est de poursuivre l'exécution après l'appel d'un nouveau mot.

```
: CARRE DUP * ;    (définition de CARRE)
4 CARRE 1 +        (on obtient 17 en sommet de la pile)
```

Enfin on introduit une structure de contrôle conditionnelle décrite par la règle syntaxique suivante qui définit deux cas :

```
conditionnelle := <liste_de_mots>_1 IF <liste_de_mots>_2 THEN
                | <liste_de_mots>_1 IF <liste_de_mots>_2 ELSE <liste_de_mots>_3 ENDIF
```

Dans le cas `<liste_de_mots>_1 IF <liste_de_mots>_2 THEN <liste_de_mots>_3`, si l'exécution de `<liste_de_mots>_1` retourne la valeur `TRUE`, `<liste_de_mots>_2` et exécutée.

Dans le cas `<liste_de_mots>_1 IF <liste_de_mots>_2 ELSE <liste_de_mots>_3 ENDIF`, si l'exécution de `<liste_de_mots>_1` retourne la valeur `TRUE`, alors `<liste_de_mots>_2` est exécutée. Sinon, `<liste_de_mots>_3` est exécutée.

On peut alors définir le mot `FACTORIELLE`, et l'appeler avec l'argument `6`, de la manière suivante :

```
: FACTORIELLE DUP 1 > IF DUP 1 - FACTORIELLE * THEN ;
  6 FACTORIELLE
```

Exécution

La syntaxe de **PF23** est tellement simple qu'elle ne nécessite pas d'analyse syntaxique. En fait un système **PF23** évalue des listes de mots de la manière suivante :

1. lire un mot
2. si le mot est une constante (entier ou booléen), l'empiler
3. si le mot est un opérateur prédéfini, l'évaluer,
4. si le mot est "`:`", ajouter au dictionnaire la définition qui suit (jusqu'à la fin "`;`" fermant la définition),
5. si c'est un `IF`, l'évaluer,
6. sinon chercher la définition du mot dans le dictionnaire : si le mot est défini récupérer sa définition et l'exécuter,
7. puis passer à la suite.

Sur papier

— trace 1 : Soit le mot `TUCK` suivant : `: TUCK DUP ROT SWAP ;` Indiquer ce qu'il fait en décrivant l'état de la pile `a b c` – (`c` est le sommet de pile) avant et après son appel dans l'évaluation de la phrase suivante : `9 12 15 TUCK`.
— trace 2 : Même question pour le mot `CUBE` (voir exemple 5), sur l'appel suivant : `6 CUBE`.

1 Définition de type

1. Définir un type OCaml `element` pour les opérateurs de pile, les opérateurs arithmétiques, les opérateurs `<`, `>`, `=` et `<>` (*non égal*), les mots, les constantes entières et les constantes booléennes.
2. définir les fonctions :

   ```
   to_string : element -> string
   of_string : string -> element
   ```

   Indice : utiliser la fonction `int_of_string_opt : string -> int option` de la bibliothèque standard d'OCaml.

3. Définir la fonction `parse : string -> element list` qui étant donné le texte d'un programme **PF23** retourne la liste d'éléments correspondante.

Définir également la fonction `text : element list -> string` qui étant donnée une liste d'éléments restitue le programme sous forme de chaîne de caractères.
On pourra utiliser la fonction `split : string -> string list` fournie.

2 Réduction

Ecrire une fonction `step` qui prend une pile et un opérateur ou une constante et retourne une pile modifiée suivant les règles suivantes :
- si c'est une constante, alors l'empiler,
- si c'est un opérateur prédéfini l'évaluer
- si c'est autre chose, lancer l'exception `Invalid_argument "step"`.

3 Calculette

On définit les types des piles et des programmes :

```
type stack = element list
type prog = element list
```

Ecrire une fonction `calc` qui prend une pile et un programme (i.e., une suite d'éléments) sans définitions de noms ni conditionnelles et retourne la pile obtenue à la fin de l'exécution du programme.
Par exemple :
- `calc [] []` ⇝ `[]`
- `text (calc [] (parse "2 3 4 * +"))` ⇝ `"14"`
- `text (calc [] (parse "7 10 * 8 + dup 42"))` ⇝ `"42 78 78"`

4 Dictionnaire

Implémenter une structure de dictionnaire, associant des noms (de type `name = string`) à des listes de mots. Pour cela, définir le type `dico` des dictionnaires (au choix), le dictionnaire vide `empty` et les fonctions de manipulation de dictionnaires :
- `add : name -> prog -> dico -> dico`
- `lookup : name -> dico -> prog`

Indice : On peut par exemple implémenter le dictionnaire sous la forme d'une *liste d'associations*, c'est à dire une liste de couple (*clé*,*valeur*). La fonction `add` ajoute une association en tête de la liste. La fonction `lookup` cherche la valeur associée à une certaine clé dans la liste d'associations.

5 Evaluateur

Ecrire une fonction `eval : dico -> stack -> prog -> stack` qui prend un dictionnaire, une pile et un programme et retourne la pile obtenue à la fin de l'exécution du programme.
Par exemple :

— `eval empty [] []` $\rightsquigarrow$ `[]`
— `text (eval empty [] (parse ": CARRE DUP * ; 4 CARRE"))` $\rightsquigarrow$ `"16"`

Pour cela, implémenter l'évaluation des constantes et des opérateurs arithmétiques, des opérateurs de pile (e.g., `DUP`), des les déclarations de nouveaux noms dans le dictionnaire, et des deux constructions conditionnelles (*if-then* et *if-else-endif*).

6 Programmation en PF23

Définir une fonction `fib : int -> string` telle que `fib n` construit un programme **PF23** calculant le `n`ème nombre de Fibonacci :

```
fib(n) = 1                      si n < 1
fib(n) = fib(n-1) + fib(n-2)    sinon
```

On donne un exemple, plus simple qui pourra servir de modèle. La fonction `carre n` suivante construit le programme **PF23** calculant le carré de `n` :

```
let carre (n:int) : string =
   Printf.sprintf ": CARRE DUP * ; %d CARRE" n ;;
```

Indice : on pourra aussi s'inspirer de la définition de `FACTORIELLE` donnée plus haut dans l'énoncé.

7 Ecriture de jeux de test

Pour tester votre implémentation, définir une liste de couples :

```
let jeux_de_test : (string * string) list = [ ... ]
```

Chaque couple associe au texte d'un programme **PF23** le résultat de son exécution.

Par exemple : `("1 2 +", "3")`.

On cherchera notamment à tester des calculs récursifs et des manipulations complexes de la pile (en utilisant `SWAP` et `ROT` par exemple).