Polytech ParisSud – 4ème année Informatique 2018-2019

Feuille de TD - III

Exercice 1 : On considère les définitions de types d'un langage inspiré de CAML. Les types peuvent être génériques. Les paramètres de généricité sont introduits par des « variables de type »[1] : `'a list` définit par exemple des listes d'éléments de type `'a`, où `'a` représente un type quelconque. On instancie un tel type en remplaçant la variable de type par une expression de type comme dans `int list` ou `(int list) list` pour obtenir respectivement des listes d'entiers ou des listes de listes d'entiers. Si un type prend plus d'une variable de type, celles-ci sont données entre parenthèses et séparées par des virgules. Le type `type ('a, 'b) paire = 'a*'b` définit des paires dont chaque composant est d'un type arbitraire, comme dans `(int, int list) paire`. Le type « paire homogène » s'écrit `type 'a monoPaire = 'a*'a` dont une instance possible est `int monoPaire`.

La syntaxe pour définir un type est : **`type`** *variablesOptionnelles nom-de-type* `=` *expression-de-type*

Un nom de type est un identificateur commençant par une minuscule. Une variable de type est formée d'une apostrophe suivie d'un identificateur. L'expression de type peut être :

- un nom de type (dont les types prédéfinis) éventuellement instancié: `int`, `float`, `'a list`, `int list`, ...
- si *type*$_1$ et *type*$_2$ représentent deux expressions de types, *type*$_1$ `*` *type*$_2$ représente le type de leur produit cartésien
- un type « fonction » : une fonction prend toujours un seul argument et renvoie toujours un résultat. Le constructeur d'un type fonction est de la forme *type*$_1$ → *type*$_2$. Le paramètre ou le résultat peut être lui-même de type *fonction*...
- un « type avec variantes » défini par l'utilisateur. Chaque variante est étiquetée par un « constructeur » avec ou sans paramètres, sous la forme *Constructeur* ou *Constructeur* **`of`** *expression-de-type*. Un constructeur est un identifiant débutant par une majuscule. Les variantes du type sont séparées par `'|'`. L'expression de type associée à un constructeur ne doit **pas** elle-même contenir de constructeur. Une expression avec constructeur ne peut **pas** être opérande de **`list`**, `*` ou → .
- une sous-expression de type entre parenthèses.

Exemples :

```
type paireInt = int*int;
type listeListeInt = (int list) list;
type fonctionPaireDeIntDansInt = (int*int) → int;
```

Fonction des entiers vers les fonctions des listes d'entiers vers les listes d'entiers :

```
type fonc = int → (int list → int list);
```

Type avec constructeurs:

```
type carte = As | Roi | Dame | Valet | Valeur of int;
```

Arbres binaires paramétrés par le type de l'étiquette

```
type 'a binTree = Nil | Noeud of 'a binTree * 'a * 'a binTree
```

L'instantiation d'un type générique a la même priorité que l'opérateur **`list`**; ils sont prioritaires sur `*` qui est prioritaire sur →. L'opérateur → associe à **droite**, l'opérateur `*` associe à **gauche**. Donc `int * int list → int → int binTree` est équivalent à la forme complètement parenthésée `(int * (int list)) → (int → (int binTree))`.

Q0. Donnez la liste des tokens (unités lexicales) que vous utiliseriez pour l'analyse lexicale de ce langage.

Q1. Donnez une grammaire **non ambiguë** pour les déclarations de type qui respecte les précédences et associativités demandées.

Q2. Donnez l'arbre syntaxique de l'expression de type `int * int list → int → int binTree`

Exercice 2 : On considère maintenant une sous-grammaire des expressions de ce langage[2]. Une grammaire (ambiguë) des expressions est la suivante:

```
E ::= E + E  | Id | Cste | ( E ) | E , E | E E
```

L'opérateur `+` est l'addition sur les entiers. L'opérateur `,` construit des n-uplets (n ≥ 1) de valeurs comme dans le triplet `x+3, x*y, 5` qui est de type `int*int*int`. Chaque composant d'un n-uplet peut être d'un type arbitraire. La règle `E E` correspond à l'application de fonctions : l'évaluation du premier `E` doit donner en résultat une fonction qu'on appliquera au résultat de l'évaluation du second `E`. Les fonctions prennent un unique argument et donnent un unique résultat. Ci-dessous on définit deux fonctions `add` et `somme` [3] qui se distinguent par la façon dont elles prennent leurs arguments (et donc aussi par le type de leur résultat)

```
let add = function x -> function y -> x + y
```

---

1 Une variable de type est un identificateur précédé du symbole `'` .

2 Java 8 permet des choses similaires avec les « *lambda expressions* ».

3 La construction `let` permet de donner un nom. La construction `function` permet de définir une fonction anonyme.

```
let somme = function (x, y) -> x + y
```

L'appel `add 3` est correct, de même que `add (3)`, et renvoie en résultat une fonction anonyme à un paramètre `y` qui calcule `y + 3`. Une expression comme `add 3 5`, ou avec des parenthèses explicites `(add 3) 5`, est correcte et donnera `8`. Un appel tel que `add(3, 5)` est incorrect car correspondrait à une fonction qui attend en argument un couple d'entiers, comme le fait `somme`. Un appel correct à `somme` s'écrit lui `somme(3, 5)` et non pas `somme 3 5`.

On suppose que l'application de fonction a la précédence la plus élevée, suivie de `'+'`, puis `','`. Tous les opérateurs sont associatifs à gauche.

1. Les expressions `somme 3, 5` et `add 3, 5` sont-elles correctes (syntaxe et typage) ?
2. Donnez une grammaire **non-ambiguë** pour le langage qui reflète les précédences et associativités indiquées.
3. Donnez les arbres syntaxiques des expressions `add 3 5 + 2` et `add 3 + 5 2` dans **votre** grammaire en considérant `add` comme une instance de `Id` et `3` et `5` comme des instances de `Cste`. Laquelle des deux correspondra à une expression correcte du point de vue du typage ?
4. Calculez les ensembles *First* et *Follow* pour la **grammaire de départ** augmentée de la règle `S ::= E $`.
5. A l'aide de l'automate LR(0) fourni en annexe, indiquez **tous** les conflits (état, type de conflit et caractère d'avance) et **montrez comment les résoudre** pour avoir un analyseur correct.