Université Bordeaux
Licence, module Compilation, 2018/2019

TD5 – Contrôle de type

Exercice 5.1

Une première grammaire d'expression typée Soit la grammaire suivante écrite décrivant une expression typée. Les terminaux de cette grammaire sont les identifiants, les constantes entières, les chaînes de caractères, les mots clés `List`, `int`, `string`, le symbole "`->`" et les symboles d'un caractère "`;:(),+-[]`".

```
expression_typee : declaration_list expression

declaration_list:
                | declaration_list declaration

declaration: identifiant ':' type ';'

type: type_simple
    | "List" '(' type ')'
    | '(' type_list ')' '->' type

type_simple: "int"
           | "string"

type_list:
         | type
         | type_list ',' type

expression: identifiant
          | entier
          | chaine
          | expression '+' expression
          | expression '-' expression
          | '(' expression ')'
          | '[' expression_list ']'
          | expression '(' expression_list ')'

expression_list: %empty
               | expression
               | expression_list ',' expression
```

1. On s'intéresse tout d'abord au langage des types, c'est-à-dire le langage engendré par le non-terminal `type`.

   (a) Vérifier si oui ou non les termes suivants sont des types valides :

```
(List(int), int)                    string
() -> string                        (int) -> (List(string)) -> string
List(int,List(int))                 ((int) -> (List(string))) -> string
```

(b) Donner deux autres types valides faisant intervenir chaque règle de production de `type`.

2. On s'intéresse maintenant au langage engendré par le non-terminal `expression_typee`. On retrouve dans ce langage les expressions typées suivante :

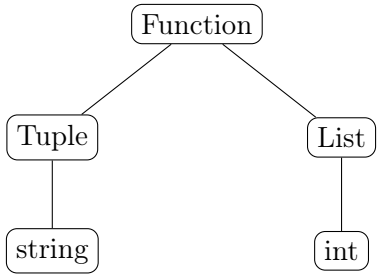
```
42
a: int; b: int; a-b-1
head: (List(int)) -> int; head(["un","deux"]) + 1
item: (int) -> (List(string)) -> string; i: int; item(i)(["un","deux","trois"])
find: (List(string)) -> (string) -> int; find(["un","deux","trois"])
```

Dans ces exemples, `a`,`b` et `i` sont des variables, et `head` et `item` sont des fonctions.

(a) Réaliser un programme reconnaissant le langage des expressions typées.

(b) En utilisant une grammaire attribuée, représenter pour chaque déclaration `identifiant: type;` le type associé à l'aide d'une classe `Type`.

Indication : Une structure arborescente est fortement recommandée. Par exemple, on s'attend à produire pour le type "`(string)->List(int)`" un type de la forme :



(c) Stocker dans une `HashMap<String, Type>` le type de chaque identifiant.

3. On reconnaît maintenant les expressions typées, cependant on souhaite que celles-ci soient *bien* typées. Par exemple, l'expression "`1+"deux"`" est mal typée car on ne peut pas additionner un entier avec une chaîne de caractères. Lors de l'appel d'une fonction, "`exp_1 (exp_2, exp_3)`", `exp_1` doit être de type fonction à deux arguments, `exp_2` doit avoir le type de son premier argument et `exp_3` celui de son second.

(a) Parmi les expressions proposées à la question 2, quelles sont celles bien typées ? Donner leurs types.

(b) Chaque expression a un type qui se construit par induction sur sa structure. Implémenter la construction du type d'une expression.

(c) En plus des opérations arithmétiques + et − sur les entiers, on s'autorise à utiliser le symbole + pour la concaténation de chaînes de caractères. Modifier la grammaire pour refléter ce changement.

Exercice 5.2

Les types templates et l'inférence de type Dans les langages haut niveau comme Java ou C++, la construction de type autorise la déclaration de type variable, par exemple dans `HashMap<K,V>`, `K` et `V` sont des types quelconques. En d'autres termes, on s'autorise l'utilisation de variables dans la déclaration de type. En piochant par exemple dans les fonctions standard OCaml, la fonction `head` est de type `List(A) -> A`.

1. Modifier la grammaire précédente pour tenir compte des types variables.
2. Est-il possible de donner un type aux expressions suivantes ?
   `head: List(A) -> A; head(["un";"deux"])`
   `x: int; f: A -> B; head: List(C)->C; head(x)+f(x)`
   `x: int; f: A -> B; head: List(C)->C; x+f(head(x))`
   `x: A; cons: (B,List(B)) -> List(B); cons(x,x)`
3. L'exemple de la question précédente montre qu'on a besoin de nouveaux outils pour déterminer le type d'un expression. Dans l'exercice précédent, il suffisait de tester des égalités entre des types pour vérifier qu'une expression est bien typée. À cause de l'introduction de types variables, on doit tester un nouveau critère : si t et t' sont deux types, on dit que t et t' sont unifiables ($t \simeq t'$) si on peut trouver une substitution σ des types variables apparaissant dans t et t' telle que $\sigma(t) = \sigma(t')$. Une telle substitution est un *unificateur*. Parmi les types suivants, déterminer lesquels sont unifiables et donner un unificateur le plus général possible lorsque c'est le cas.

   (a) `int` et `string`
   (b) `int` et `A`
   (c) `int` et `List(A)`
   (d) `A` et `B`
   (e) `List(A)` et `List(int)`
   (f) `List(A)` et `(A) -> B`
   (g) `List(A)` et `A`
   (h) `(List(A), B) -> C` et `(int, List(D)) -> string`

4. Expliquer comment l'algorithme page 4 permet de déterminer si $t \simeq t'$.
5. Implémenter cet algorithme.
6. Construire un analyseur qui prend en entrée des expressions typées (avec des types variables) et qui en détermine le type (le plus général possible en cas d'ambiguïté), et échoue si l'expression est mal typée.
7. Modifier cet analyseur pour qu'on puisse se passer de définir le type d'une ou plusieurs expressions. Par exemple, dans l'expression `f:int -> A; f(x)`, `x` est nécessairement un entier. Que se passe-t-il quand aucun type n'est déclaré ?
8. Modifier l'analyseur pour qu'il renvoie :
   — Lorsque l'expression d'entrée est bien typée, la liste des types de toutes les variables y intervenant.
   — Lorsque l'expression d'entrée est mal typée, un message d'erreur expliquant où il y a un problème.

Entrées: t, t' deux types
Sorties: Une substitution σ ou un échec

$\sigma \leftarrow$ substitution vide
$E \leftarrow \{(t, t')\}$
tant que $E \neq \varnothing$ **faire**
  $(a, b) \leftarrow$ un élément de E
  $E' \leftarrow E \setminus \{(a, b)\}$
  si $a = b$ **alors**
    $E \leftarrow E'$
  fin
  si a *ou* b *est* `int` *ou* `string`*, et* $b \neq a$ **alors**
    retourner *Échec*
  fin
  si $a = \texttt{List}(a')$ **alors**
    si $b = \texttt{List}(b')$ **alors**
      $E \leftarrow E' \cup \{(a', b')\}$
    sinon
      retourner *Échec*
    fin
  fin
  si $a = (a'_1, \ldots, a'_n)$ `->` a' **alors**
    si $b = (b'_1, \ldots, b'_p)$ `->` b' *et* $n = p$ **alors**
      $E \leftarrow E' \cup \{(a', b'), (a'_1, b'_1), \ldots, (a'_n, b'_n)\}$
    sinon
      retourner *Échec*
    fin
  fin
  si $(a, b) = (x, t)$ *ou* (t, x) *avec* t *un terme et* x *une variable de type* **alors**
    si x *n'apparaît pas dans* t **alors**
      $\sigma \leftarrow \sigma \cup [x := u]$
      $E \leftarrow E'[x := u]$
    sinon
      retourner *Échec*
    fin
  fin
fin
retourner σ