Projet 1
ENS Lyon — L3IF
Construction de l'arbre et affichage

Sébastien Briais

13 octobre 2007

Dans cette partie, vous allez devoir définir une syntaxe abstraite pour le langage `drei` puis ensuite modifier votre analyseur syntaxique pour construire l'arbre de syntaxe abstraite. Pour tester le bon fonctionnement de tout ceci, vous ferez ensuite une fonction d'impression de l'arbre.

À rendre pour le 26 octobre minuit au plus tard.

Les groupes de couleur B sont invités à rendre ceci avant cette date afin de pouvoir commencer l'analyseur de types plus vite.

1 Consignes générales

Votre compilateur devra pouvoir être invoqué à l'aide de la commande `dreic` dans un terminal. Il prendra en argument un paramètre indiquant la phase du compilateur testée suivi du nom de fichier `drei` à "compiler". Un exemple typique d'utilisation est :

```
./dreic -printer Factorial.drei
```

Les noms des phases sont indiqués ci-après :

Nom de la phase	Nom de l'option
Analyse lexicale	`-scanner`
Analyse syntaxique	`-parser`
Construction de l'arbre	`-printer`
Analyse des types	`-analyser`
Génération de code	`-generator`

En outre, pour le rendu de votre projet,

- Vos sources devront être obligatoirement accompagnées d'un fichier `Makefile` permettant de construire votre projet. *Optionnellement*, vous pouvez aussi joindre un fichier `README` pour y reporter vos remarques éventuelles sur le fonctionnement ou la compilation de votre programme. Avec un peu de chance, il sera lu :p
- Vous devrez rendre vos sources sous forme d'une archive "tar gzippée". Le contenu de votre archive devra se décompresser dans un répertoire portant le nom de votre groupe (i.e. `B1`, `R3`, ...)

– L'archive devra m'être envoyée par mail. Le titre de votre mail sera de la forme "[PROJET] Etape 2"

Comme la plupart des groupes a repris le corrigé de la phase 1, cela devrait être facile à respecter.

2 Description de l'arbre

2.1 Définition de la syntaxe abstraite

Pour définir le type des arbres, il est conseillé de s'inspirer de la grammaire abstraite utilisé dans les documents décrivant la sémantique et le typage de `drei`. Néanmoins, vous êtes libres de vos choix.

Comme précisé en cours, n'oubliez pas d'annoter votre arbre avec la position correspondante dans le fichier source (pour pouvoir donner des messages d'erreur explicites par la suite).

2.2 en ocaml

Voici quelques indications pour réaliser cette partie en `ocaml`. Dans un module `Ast` (pour Abstract Syntax Tree), vous définirez plusieurs types algébriques permettant de représenter les arbres correspondant à la syntaxe abstraite.

```
type position = int * int

type name = Name of string

let mk_name str = Name(str)

type program =
  Program of decl list * statement
and decl =
  ClassDef of position * name * name option * member list
and member = ...
...
and ttype =
  | ClassType of name
  | IntType
  | NoType
and statement = ...
...
and expression = ...
...
```

3 Construction de l'arbre

Une fois le type des arbres défini, vous devrez modifier l'analyseur syntaxique pour construire un arbre de syntaxe abstraite.

Vous pouvez vous inspirer de l'exemple des expressions arithmétiques pour faire ceci.

3.1 Sucre syntaxique

Certaines constructions de la syntaxe concrète de `drei` n'ont pas besoin d'être représentées dans l'AST. Ce sont les constructions qui sont strictement équivalentes à d'autres constructions du langages, mais avec une syntaxe différente, et que l'on appelle «sucre syntaxique».

Equivalences pour les expressions sucrées :

Expr. sucrée	Expressions équivalente
`e1 || e2`	`!(!e1 && !e2)`
`false`	`0`
`true`	`1`
`"abc"`	`new Cons(A, new Cons(B, new Cons(C, new Nil())))` où `X` est la position sur la table ASCII du caractère `x`
`""`	`new Nil()`
`if (Expr) Statement`	`if (Expr) Statement else {}`

4 Impression de l'arbre

Pour vérifier que les arbres que vous construisez lors de l'analyse syntaxique sont corrects, vous allez ensuite écrire une fonction d'impression pour refléter la structure de vos arbres (comme expliqué en cours, n'hésitez pas à imprimer plus de parenthèses qu'il n'en faut).

Le résultat de l'impression du résultat de l'analyse d'un fichier source `drei` par votre compilateur doit être un programme `drei` valide, sans sucre syntaxique (ni commentaires) et avec la même signification sémantique que l'original. Vous pouvez valider certaines de ces propriétés de la manière suivante :

– Exécuter votre compilateur deux fois de suite, en se servant du résultat de la première exécution pour lancer la seconde. Les deux résultats doivent être identiques caractère pour caractère.

5 Exemple de fonctionnement sur Factorial.drei

La commande `./dreic -printer Factorial.drei` imprimera typiquement le programme de la page suivante.

```
class Factorial {
    def factorial1(x:Int) : Int = {
        var res:Int = 1;
        if ((x)>(0))
            set res = (x)*(this.factorial1((x)-(1)));
        else
            {
            }
        return res
    };
    def factorial2_aux(x:Int, acc:Int) : Int = {
        var res:Int = acc;
        if ((x)>(0))
            set res = this.factorial2_aux((x)-(1), (acc)*(x));
        else
            {
            }
        return res
    };
    def factorial2(x:Int) : Int = this.factorial2_aux(x, 1);
    def factorial3(x:Int) : Int = {
        var p:Int = 1;
        while ((x)>(0))
            {
                set p = (p)*(x);
                set x = (x)-(1);
            }
        return p
    };
}
{
    var fac:Factorial = new Factorial();
    var x:Int = 5;
    printInt(fac.factorial1(x));
    printChar(10);
    printInt(fac.factorial2(x));
    printChar(10);
    printInt(fac.factorial3(x));
    printChar(10);
}
```