Programmation - TP3: Vers le projet

Simon Halfon

September 28, 2015

1 Objectif du TP

Le dernier TP était majoritairement consacré aux types produits, dans ce TP on va surtout jouer avec des types sommes. L'idée principale est de voir toutes les fonctions qui vous seront utiles pour le projet. Comme je l'ai dit les semaines passées, l'objectif des TP de OCaml est que lors du projet, vous n'ayez pas à vous soucier de OCaml. Les difficultées que vous rencontrerez doivent être relatives à la sémantique du C et au code assembleur (et des exceptions).

Pour garantir que vous ayez vu toutes les fonctionnalités d'OCaml nécessaires à la création d'un compilateur, quoi de mieux que ... de créer un compilateur ?

Exercice - 1 *Compilation d'un mini langage de programmation*

L'objet de l'exercice est d'écrire un compilateur pour un langage de programmation impératif simplissime (mais Turing complet): `IMP`. Un compilateur est un traducteur automatique (bien plus performant que Gogole traduction) d'un premier langage de programmation $\mathcal{L}_1$ vers un second langage $\mathcal{L}_2$. Par ailleurs, un compilateur est un programme, il est donc écrit dans un troisième langage de programmation $\mathcal{L}_3$. Pour votre projet compilation, $\mathcal{L}_1$ = C−−, $\mathcal{L}_2$ = Assembleur et $\mathcal{L}_3$ = OCaml. Dans cet exercice, on prendra $\mathcal{L}_1$ = `IMP` et $\mathcal{L}_2 = \mathcal{L}_3$ =OCaml.

Le langage `IMP` est constitué de trois types d'objets:

- Les expressions arithmétiques, qui comprennent: les nombres entiers, les variables (globales), ainsi que l'addition, la soustraction et la multiplication de deux expressions arithmétiques.
- Les expressions booléens, qui comprennent: les booléens "vrai" et "faux", l'égalité et la comparaison de deux expressions arithmétiques, et les 3 opérateurs logiques (négation, conjonction, disjonction) appliquées à des expressions booléennes.
- Les programmes: "skip" (le programme qui ne fait rien), l'affectation d'une expression arithmétique à une variable, la séquence, la conditionnelle (if then else) et la boucle while.

1- Les types:
a- Ecrire en OCaml le type des expressions arithmétiques.
b- Ecrire en OCaml le type des expressions booléennes.
c- Ecrire en OCaml le type des programmes de `IMP`.

Dans `IMP`, toutes les variables sont globales. Un environnement est une valuation de ces variables globales. Formellement, c'est une fonction qui à chaque variable associe une valeure entière (oui en `IMP` on ne manipule que des variables entières, c'est ce qui en fait un langage simplissime). L'exécution d'un programme est définit comme une fonction des environnements dans les environnements: tout ce que font les programmes en `IMP`, c'est modifier les valeurs des variables. Par exemple, le programme suivant

```
a := 0 ;
s := 1 ;
t := 1 ;
while (s < n)
{
```

```
  a := a + 1 ;
  s := s + t + 2 ;
  t := t+2
}
```

manipule (modifie) quatre variables: a, s, t et n. Il vérifie qu'après l'exécution du programme, s contient la partie entière de la racine carré de n.
2- Ecrire le type des environnements.
3- Ecrire en OCaml un programme `IMP` qui calcule la factoriel d'une certaine variable (par exemple "x").
4- Ecrire une fonction qui étant donné un environnement e et un programme `IMP` p calcule l'exécution de p depuis l'environnement e. Autrement dit, la fonction que vous devez écrire est de type

```
environnement -> program -> environnement
```

Une telle fonction s'appelle un interpréteur.
5- Testez votre fonction factoriel

On va maintenant compiler `IMP` en OCaml, c'est à dire écrire un programme `compile` (en OCaml) qui prend en entrée un programme `IMP` p et qui écrit dans un fichier un programme OCaml *équivalent* à p, c'est à dire "qui fait la même chose".

Vous allez vouloir et devoir tester votre compilateur. Vous avez sans doute remarqué en écrivant la fonction factorielle qu'il est très pénible d'écrire du `IMP` directement avec la représentation qu'on en a donné dans OCaml, ce qui va vous dissuader de tester votre compilateur sur de gros exemples. Il serait particulièrement agréable de pouvoir écrire un programme `IMP` comme dans l'exemple de la racine carré plus haut. C'est ce que j'ai fait pour vous !
6- Allez récupérer l'archive `IMP.zip` sur http://www.lsv.ens-cachan.fr/~halfon/. Elle contient les fichiers suivants:

- `parser.ml`, `lexer.mll`. Ne regardez pas ces deux fichiers, ils ont été générés automatiquement par les lexer et parser `Ocamllex` et `Menhir` depuis les fichiers `parser.mly` et `lexer.mll`. Ils sont par conséquent illisibles. Par contre, si vous êtes curieux, vous pouvez aller voir les fichiers `parser.mly` et `lexer.mll`. Vous n'avez cependant pas à y toucher pour l'exercice. Vous verrez peut être ces outils au second semestre dans le cours de Langages Formels.

- `main.ml`, que vous ne devez pas modifier non plus. Il s'occupe de l'interraction avec l'utilisateur que je décris un peu plus bas.

- `compile.ml`: c'est le fichier que vous devez compléter. Il faut y écrire une fonction

  ```
  compile: out_channel -> string -> decl -> decl -> program -> unit
  ```

  Le type `out_channel` en OCaml est le type d'un canal de sortie. C'est comme cela que l'on écrit dans un fichier, avec la fonction `Printf.fprintf`. Un exemple est donné dans le fichier `compile.ml`. L'argument de type `string` est simplement le nom du fichier (sans l'extension) que l'on vous a donné à compiler, il vous sert à nommer la fonction de votre fichier `.ml`. Les deux autres types sont décris dans le fichier `type.ml`

- `type.ml`, contient la définition des types des programmes `IMP`. C'est exactement la réponse à la question 1. Il contient aussi la définition du type

  ```
  decl = variable list
  ```

  Ces deux listes en argument de la fonction `compile` vous donne le nom des variables de votre programme qui doivent être considérées comme des arguments (resp. des output). Les autres variables doivent être traitées comme des variables globales. Lisez le sujet jusqu'au bout, ça devrait s'éclaircir.

  Bien sûr vous avez pu choisir des noms de constructeurs différents des miens. Vous pouvez donc soit apprendre les miens, soit écrire une fonction de conversion de type:

```
program -> my_program
```

et travailler avec votre type dans les fonctions auxiliaires.

- `Makefile`: c'est le fichier magique qui vous permet de compiler tout ça sans vous poser de questions. Il suffit de taper `$ make` dans le terminal. Ceux qui ont fait l'exercice sur la calculatrice s'en souviennent sans doute. On vous expliquera plus tard comment fonctionne les fichiers makefile (c'est pas ma guerre).

Utilisation: comme précisé au dessus, commencez par faire `$ make` pour compiler le tout. Ensuite, utilisez `$ ./Compile fichier-1 ... fichier-n` pour compiler les fichiers 1 à n. Si vous essayez cela, vous allez rapidement voir que mon programme exige un fichier d'extension `.imp`. Ecrivez donc quelques broutilles dans un fichier `test.imp`. Si vous exécutez `$ ./Compile test.imp`, vous allez recevoir une erreur de Parsing ou de Lexing. Il faut que votre fichier contienne du code syntaxiquement correct, et la syntaxe est la suivante:

- Comme dans la plupart des langages, les espaces et retour à la lignes sont ignorées.
- Comme dans l'exemple de programme `IMP` donné plus haut, l'affectation est notée `:=`, la séquence `;`, et les opérations arithmétiques de manières usuelles.
- Les opérateurs booléens doivent être notés `!`, `&&`, `||`. Seuls `=`, `<` et `<=` sont reconnus.
- Dans les boucles while et les conditionnelles, la condition booléennes doit être entre parenthèse, et les corps de boucle entre accolades. Par exemple:

```
if (x <= z && y < z)
{
    x := y
}
else
{
    x := z - y ;
    z := y
}
```

- Le programme qui ne fait rien est noté `skip`.

Enfin, comme `IMP` ne contient que des variables globales, j'ai ajouté à la syntaxe la possibilité de spécifier parmi les variables du programmes, lesquelles sont des variables d'entrées, lesquelles sont des variables de sorties, et lesquelles sont des variables locales. Vous pouvez donc ajouter en début de fichier les lignes optionnelles:

```
ARG x,y,z
OUT t,u.
```

Exercice: Je vous demande donc de générer du code OCaml selon les spécifications suivantes:

- Le fichier `.ml` contient deux fonctions. La première est nommé d'après le nom du fichier `.imp`, qui prend autant d'argument que le nombre de variables déclarée après le mot clé `ARG` et qui retourne un k-uplet d'entiers, où k est le nombre de variables déclarée après le mot clé `OUT`.
- Pas de variables globales, la première ligne du fichier `.ml` est:

```
let nom_fonction arg_1 ... arg_n =
```

où n est le nombre de variables déclarées après le mot clé `ARG` dans le fichier `.imp`.

- La seconde fonction est nommée `()` et est de type:

```
(): unit -> unit
```

qui doit demander n arguments à l'utilisateur, soit lors de l'appel dans le terminal (utilisez `Sys.argv`), soit à l'aide des fonctions `read_int` ou `read_string` (consultez `Persvasives`); et qui affiche le résultat de la fonction dans le terminal (`Printf.printf`).

Par exemple, si j'ai un fichier `nadine.imp`:

```
ARG x,y,z
OUT t.

t := x+y+z
```

alors, je veux pouvoir exécuter les commandes suivantes:

- `$ ./Compile nadine.imp`
- `$ ocamlc -o Nadine nadine.ml`
- `$ ./Nadine 12 5 66`

et obtenir dans le terminal le nombre 83. Ou, dans le second cas, lorsque je tape `$ ./Nadine`, le terminal me demande d'entrer 3 entiers, et me répond 83. Je veux également que votre programme crash **proprement** si le nombres d'arguments donnés n'est pas le bon, ou si ce ne sont pas des entiers, etc.