Projet : traducteur C vers Ocaml

À rendre avant le 02/06 à 22h

Le but de ce projet est d'écrire un code en C permettant la traduction d'un fichier écrit dans le langage C en un fichier écrit dans le langage Ocaml.

De manière générale, une traduction de ce genre est toujours plus ou moins possible en passant par un arbre de syntaxe abstraite. Ici ce n'est **pas** ce qu'on va faire. Il est en fait possible de traduire le C en Ocaml en lisant linéairement le code, à condition d'interdire certaines fonctionnalités du langage.

Le processus se découpe en plusieurs morceaux :

- Ouvrir le fichier C (qu'on appelera s dans la suite) et créer un fichier Ocaml (qu'on appelera d dans la suite)
- Lire et décomposer le contenu de s en unités de sens : les lexèmes. Ceci via un programme appelé lexeur, un lexeur incomplet vous est fourni avec ce sujet.
- Pour chaque lexème en C, décider de comment on le traduit en Ocaml, et l'écrire dans d.

Quelques remarques générales :

- Votre code supposera toujours que le code de s compile, donc est bien formé.
- Le code Ocaml de d doit aussi compiler et donner le même résultat à l'exécution que l'exemple en C.
- Le code produit doit être lisible, donc passer régulièrement à la ligne.
- Testez votre code au fur et à mesure en écrivant des exemples simples, **puis** complexifiez.
- N'oubliez pas qu'il s'agit d'un projet et que vous devez faire preuve d'initiative et d'initiative **logique**.
- si néanmoins vous pensez qu'une partie du projet, ou un bout de code C, est trop difficile à traduire, faites en moi part. Je n'ai pas pu tout prévoir.

1 Le lexeur

Un lexeur est un algorithme qui découpe un morceau de code en des "unités de sens", c'est à dire qu'il sépare les mots-clés du langage des noms de variable des constantes entières, etc...

Par exemple le code suivant :

```
while(x>1){
 x=x-12;
}
```

contient les lexèmes `while`, `=` (l'affectation), `12` (constante entière), `x` (variable) et bien d'autres.

Le fichier **`lexer.c`** contient le code d'un lexeur capable de reconnaitre quelques lexèmes du langage C. Il y a une fonction principale et beaucoup de fonctions auxiliaires, leurs utilités sont précisées.

Le fichier définit un type `maillon`. Il s'agit d'une structure chaînée, dont chaque maillon peut contenir un char indiquant de quel type de lexème il s'agit et un argument de type `char*` (ou `void*` si vous en avez besoin) qui apporte des précisions. Par exemple le lexème `while` est représenté par le maillon `{lexeme='M'; argument = "while";}`, `'M'` indiquant que le lexème est un mot-clé.

La fonction lexeur en elle-même prend en entrée le descripteur de fichier, de type `FILE*`. Elle renvoie des maillons enchainés qui décrivent tous les lexèmes trouvés dans le fichier.

Le lexeur lit chaque caractère un par un. Selon le caractère, le lexème peut avoir plusieurs formes :

- Soit le caractère est un lexème, par exemple '('. On ajoute le maillon correspondant et on continue
- Soit le lexème est plus grand que le caractère. Par exemple pour ' " ', le lexème associé est l'intégralité de la chaine de caractère, donc on lit des caractère jusqu'à trouver l'autre ' " '. Ensuite on crée un maillon dont l'argument est toute la chaine.
- ...

Par exemple sur l'exemple précédent le lexeur renvoie (j'ai mis des guillemets autour des caractères qui ne se voient pas) :

```
D DEBUT
M while
P (
V x
B >
0 1
```

```
P )
P {
P "\n"
P " "
V x
E =
V x
O -
0 12
P ;
P "\n"
P }
P "\n"
```

2 Étape 1 : opérations élémentaires et variables

Dans un premier temps, on écrira un traducteur pour des codes qui ne comportent que la fonction `main`. On remarquera que l'analogue de la fonction `main` en Ocaml est de mettre le code en dehors de toute fonction.

Cette étape devra permettre la définition de variables, leur affectation et leur utilisation. Les types considérés pour les variables seront uniquement **`int`** et `bool`. Ceci interdit les pointeurs (pour le moment).

On devra pouvoir reconnaitre les valeurs entières (comme 4 ou 456) et les valeurs booléennes (`false` et `true`).

Les opérations à supporter sont les opérations arithmétiques sur les entiers ($+, -, x, /, \%$), ainsi que les opérations booléennes (&&, ||, !). On traduira aussi les comparaisons ($<, <=, >, >=, !=, ==$).

Les variables fonctionnant différemment en C et en Ocaml, on fixe des normes sur le fichier s :

- les noms de variables ne contiennent que des lettres minuscules.
- toute définition de variable se fait avec une initialisation. On a donc toujours **`int`** `x = 0;` et jamais **`int`** `x;`.
- on considère que toute variable définie est susceptible d'être modifiée plus loin dans le code.
- il n'y a pas de définition de variables en dehors de `main`.
- on n'utilise pas les arguments de la fonction `main`.
- les casts (changements de type) sont interdits.
- les opérateurs `+=` et autres sont interdits.

On remarquera que s commence nécessairement par des lignes de la forme **`#include`** `...` qu'on ignorera, elles n'ont pas d'analogue en Ocaml.

Voici un exemple de fichier (on a sauté les **`#include`** `...`) et sa traduction attendue :

```
int main(){
 int x = 0;
 int y = 3+x;
}
```

```
 let x = ref 0 in
 let y = ref 3+!x;;
```

3 Étape 2 : les commentaires

Le but est simple : traduire les commentaires.

En C il existe deux types de commentaires : avec `/*... */` et avec `//...` . On veut traduire les deux.

Le lexeur ne permet pas de repérer les commentaires. Donc vous devez le modifier et faire en sorte qu'il renvoie un nouveau type de lexème spécialement pour les commentaires, dont l'argument sera le texte du commentaire. (on utilisera 'C' comme code de lexème)

4 Étape 3 : la fonction printf

Pour cette étape, on va s'intéresser à la traduction de la fonction `printf`.

On remarquera que `Printf.printf` est curryfiée en Ocaml. On ne peut donc pas se contenter de recopier les arguments de C vers Ocaml.

Les arguments sont entourés par des parenthèses en C, ce qui permet de les reconnaitre. Trouver la parenthèse ouvrante est facile, elle est juste à côté de `printf` (modulo des espaces). Trouver la parenthèse fermante est plus difficile, on peut avoir des cas comme suit : `printf("%d %d\n", 3, (3+x))` où la première parenthèse fermante n'est pas la bonne.

Il vous est demandé de trouver une solution linéaire à ce problème, c'est à dire que vous devez traduire en ne faisant qu'une seule passe de la liste chaînée des lexèmes.

Par exemple :

```
int main(){
int i=10;
printf("%d\n", (i+10));
}
```

```
 let i = ref 10 in
 Printf.printf "%d\n" (!i+10);;
```

5 Étape 4 : boucles et conditionnelles

Dans un troisième temps on va ajouter la possibilité de traduire les conditionnelles, et les boucles **`while`**.

Ici une difficulté est de pouvoir identifier la fin d'une boucle ou conditionnelle de la fin de `main` (ou plus tard d'une autre fonction). On rappelle qu'en Ocaml, à défaut de {}, on doit séparer le contenu d'une boucle du reste du code par **`do ... done`** et le contenu des deux branches de la conditionnelle par **`begin ... end`**.

Pour cette partie, on garantit :

- l'absence d'instructions **`break`** ou **`continue`** dans les boucles.
- on garantit également l'utilisation d'accolades autour du corps d'instructions et des parenthèses autour de la condition : on écrira jamais **`if`** `(x>0) x+=1;` mais plutôt **`if`** `(x>0){x+=1;}`.
- un bloc {} n'apparait jamais sauf pour une conditionelle ou une boucle (ou plus tard une fonction).

Par exemple :

```
int main(){
int i=10;
 while(i>0){
   printf("%d\n", i);
   i=i-1;
 }
}
```

```
 let i = ref 10 in
 while (!i>0) do
   Printf.printf "%d\n" !i;
   i:=!i-1
done;;
```

6 Étape 5 : fonctions

Dans cette partie, petit changement, au lieu d'avoir tout dans `main`, on peut définir d'autres fonctions. Ces fonctions doivent aussi être définies, avec le même nom et les mêmes arguments, en Ocaml.

Les fonctions peuvent être appelées dans d'autres fonctions, ou dans `main`. En revanche une fonction n'est jamais définie à l'intérieur d'une autre fonction.

Le type de retour des fonctions sera **`int`**, `bool` ou **`void`**. Évidemment le mot-clé **`return`** n'a pas d'analogue en Ocaml, donc il faudra réfléchir à comment faire. On oubliera pas le cas où il n'y a pas de **`return`**.

On ne curryfiera pas les fonctions. Par exemple, si on a **`int`** `f(`**`int`** `x, `**`int`** `y){...}`, on le traduira par **`let`** `f(x,y) = ...`.

On garantit que :

- les fonctions ne sont jamais récursives.

- les variables données en entrée d'une fonction ne sont pas modifiées. Par exemple si on a `int f(int x){...}`, alors la valeur de la variable `x` n'est pas modifiée dans le corps de la fonction.
- les noms de fonctions ne contiennent que des lettres minuscules.

7 Étape 6 : personnalisation

Dans cette partie vous devez ajouter une nouvelle fonctionnalité à votre traducteur :

- les tableaux dynamiques à une dimension (création, utilisation y compris avec effet de bord).
- les types structurés (création, utilisation y compris avec effet de bord).
- les boucles `for`.

8 Attendus

À la fin du projet vous devrez remettre un code qui compile et qui tourne sans erreur de segmentation, pour que je puisse tester. Bien sûr il est idéal que ce code fonctionne sur des exemples associés à chacune des étapes. Le code devra être commenté, mais pas trop.

Remarque : un vrai projet se découpe en fichiers distincts pour plus de lisibilité. Ici j'ai découpé en un fichier `lexer.c` et un fichier `traducteur.c`. Vous pouvez utiliser toutes les fonctions de `lexer.c` dans `traducteur.c`. (mais pas l'inverse ! !) Ne touchez pas à ce découpage à moins de savoir ce que vous faites.

Remarque : vous serez amenés à modifier le fichier `lexer.c`. Il est interdit de rendre le lexeur non linéaire (c'est à dire de revenir en arrière sur un caractère précédent). Il est également interdit de faire faire au lexeur des étapes de la traduction. Globalement si votre modification ne consiste pas à imiter ce qui est déjà fait, réfléchissez encore un peu.

En parallèle du code vous devrez rendre un rapport d'une (minimum !) ou deux (maximum !) pages expliquant les points principaux de l'implémentation et les choix techniques (et limitations supplémentaires vis à vis de l'énoncé ?) réalisés lors de votre implémentation. Le rapport vient s'inscrire en complément des commentaires.

9 Annexe

Pour utiliser C et pouvoir manipuler les fichiers chez vous on passera par une machine virtuelle (VM). La machine virtuelle nécessite **beaucoup** de mémoire, prévoyez 20 Go d'espace.

Télécharger Virtualbox sur cette page : `https://www.virtualbox.org/wiki/Downloads`. Sélectionner le lien "windows host".

Installer Virtualbox en double-cliquant sur le .exe.

Télécharger l'environnement Ubuntu de CCINP sur cette page : `https://www.concours-commun-inp.fr/fr/epreuves/les-epreuves-orales.html`, onglet MPI, "Télécharger l'environnement technique (fichier .ova) sur WeTransfer".

Lancer le logiciel et importer le fichier .ova. À cette étape si cela échoue avec une erreur du type "Invalid args", libérez plus d'espace.

Une fois le fichier importé, lancez la VM et attendez que ça démarre. Une fois la VM démarrée, vous aurez un système Linux devant vous. Pour avoir l'utilisation la moins saccadée possible, éteignez les processus en arrière-plan pour libérer le cpu (réseaux sociaux, launcher de jeux, cortana, mise à jour automatique de windows, ...)

La VM contient VScode, emacs et un terminal avec gcc et ocamlc. Pour mettre les fichiers du projet sur la VM, utilisez soit une clé USB (activez l'option dans Virtualbox), ou modifiez le dossier mémoire de votre VM. Si besoin il y a des tutos sur internet.