Projet :
Une extenstion du langage C par les types inductifs

(Enoncé provisoire, mis à jour le 22 mars 2010)

Contexte

Dans des langages tels que Caml il est possible gérer la syntaxe abstraite des langages par ce qu'on appelle les types inductifs. Plus généralement, les types inductifs donnent la possibilité de représenter de façon directe les structures arborescentes.

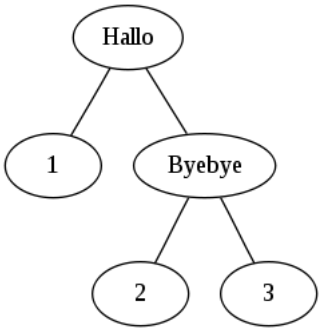
Exemple. La définition de type :

```
type arbre =  Feuille of int | Noeud of string*arbre*arbre;
```

permettra d'écrire des expressions dénotant des arbres – dont les feuilles sont étiquetés par des entiers, et les noeuds internes sont étiquetés par des chaînes de caractères.
L'expression de type `arbre`

```
Noeud("Hallo",Feuille(1),Noeud("Byebye",Feuille(2),Feuille(3)))
```

dénotera l'arbre



Représenter de telles structures arborescentes dans le langage C demande d'abord de donner des définitions de types souvent assez complexes, en donnant à l'avance des définitions de structures et unions.

Exemple. Le type `arbre` définit ci-dessus nécessitera les définitions suivantes :

```
/* Contructeur Noeud */
struct Noeud {
        char * champ_1;
        union arbre *champ_2;
        union arbre *champ_3;
};
/* Contructeur Feuille */
struct Feuille {
        int champ_1;
};
/** Type arbre **/
union arbre {
        struct Feuille *champ_1;
        struct Noeud *champ_2;
};
/* Typedefs */
typedef struct Noeud Noeud;
typedef struct Feuille Feuille;
typedef union  arbre  arbre;
```

Ensuite, nous souhaitons avoir un mécanisme pour pouvoir construire des tels arbres.

Exemple. Nous allons définir un certain nombre de fonctions :

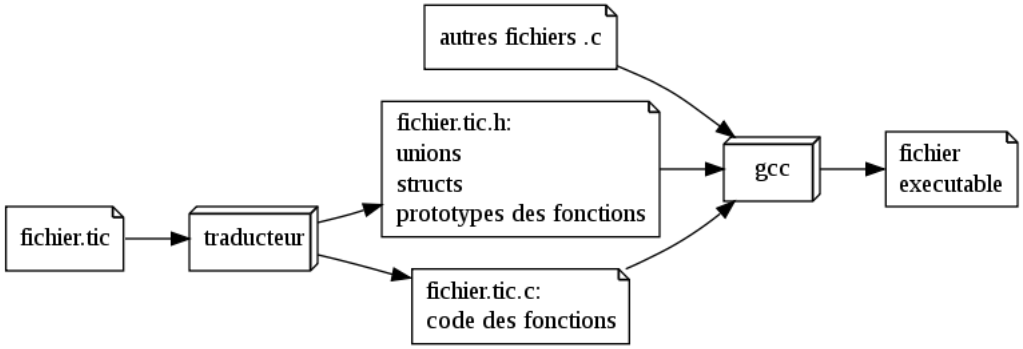
```
/* Prototypes */
extern Noeud *make_Noeud(char *,arbre *,arbre *);
extern Feuille *make_Feuille(int);
extern arbre *make_arbre_of_Feuille(Feuille *);
extern arbre *make_arbre_of_Noeud(Noeud *);
```

L'implantation des ces fonctions sera (par exemple) la suivante :

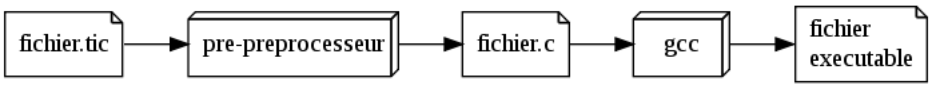
```
/* Contructeur Noeud */
Noeud *make_Noeud(char * par_1,arbre * par_2,arbre * par_3)
{
        Noeud* new = malloc(sizeof(Noeud));
        if(new ==NULL) return NULL;
        new->champ_1 = par_1;
        new->champ_2 = par_2;
        new->champ_3 = par_3;
        return new;
}
/* Contructeur Feuille */
Feuille *make_Feuille(int par_1)
{
        Feuille* new = malloc(sizeof(Feuille));
        if(new ==NULL) return NULL;
        new->champ_1 = par_1;
        return new;
}
/* Type arbre */
arbre *make_arbre_of_Feuille(Feuille * par)
{
        arbre* new = malloc(sizeof(arbre));
        if(new == NULL) return NULL;
        new->champ_1 = par;
        return new;
}
arbre *make_arbre_of_Noeud(Noeud * par)
{
        arbre* new = malloc(sizeof(arbre));
        if(new == NULL) return NULL;
        new->champ_2 = par;
        return new;
}
```

Objectifs

Nous souhaitons automatiser ce processus de traduction, d'une syntaxe de type en style Caml vers la syntaxe C. Le premier objectif est donc celui de programmer un traducteur capable de produire à partir d'un fichier `fichier.tic` – contenant des définitions de types dans la syntaxe Caml, où `tic` signifie types-inductifs C – un fichier `fichier.tic.h`, qui contiendra les définitions des structures et unions nécessaires, les typedefs et les prototypes des fonctions pour construire les arbres, et aussi un fichier `fichier.tic.c` où ces fonctions sont définies. Le schéma est alors le suivant :



L'objectif final sera de construire un pre-preprocesseur : dans le fichier `fichier.tic` on pourra inclure (par un mécanisme de mots clés) de telles définitions de types, qui seront traduite par le pre-preprocesseur dans un fichier `fichier.c` dans le langage C prêt à être compilé, selon le schéma suivant :



Étape I : la grammaire des types, un analyseur

Nous allons définir d'abord quel est le langage en style OCaml qui nous permets de définir les types. Nous allons le faire bien évidemment à l'aide d'une grammaire.

$$
\begin{aligned}
\mathit{definitions} &\rightarrow \mathit{definition} \mid \mathit{definition}\ \mathit{definitions} \\
\mathit{definition} &\rightarrow \texttt{type}\ \mathit{defs\_mutuelles}\ \texttt{';'} \\
\mathit{defs\_mutuelles} &\rightarrow \mathit{une\_def}\ \mathit{autres\_defs} \\
\mathit{autres\_defs} &\rightarrow \epsilon \mid \texttt{and}\ \mathit{une\_def}\ \mathit{autres\_defs} \\
\mathit{une\_def} &\rightarrow \mathit{def\_simple} \mid \mathit{def\_inductive} \\
\mathit{def\_simple} &\rightarrow \text{IDENT}\ \texttt{'='}\ \mathit{type\_simple} \\
\mathit{def\_inductive} &\rightarrow \text{IDENT}\ \texttt{'='}\ \mathit{type\_inductif} \\
\mathit{type\_simple} &\rightarrow \text{IDENT} \mid \text{TYPE\_ATOMIQUE} \\
\mathit{type\_inductif} &\rightarrow \mathit{somme} \\
\mathit{somme} &\rightarrow \mathit{terme}\ \mathit{reste\_somme} \\
\mathit{reste\_somme} &\rightarrow \epsilon \mid \texttt{'|'}\ \mathit{terme}\ \mathit{reste\_somme} \\
\mathit{terme} &\rightarrow \text{CONSTRUCTEUR}\ \mathit{arg\_of\_constr} \\
\mathit{arg\_of\_constr} &\rightarrow \epsilon \mid \texttt{of}\ \mathit{produit} \\
\mathit{produit} &\rightarrow \mathit{facteur}\ \mathit{reste\_produit} \\
\mathit{reste\_produit} &\rightarrow \epsilon \mid \texttt{'*'}\ \mathit{facteur}\ \mathit{reste\_produit} \\
\mathit{facteur} &\rightarrow \text{IDENT} \mid \mathit{type\_atomique}
\end{aligned}
$$

On vous demande d'abord d'écrire un analyseur syntaxique capable de reconnaître les définitions de types bien formés par rapport à cette grammaire. Implantez une procédure de récupération sur erreur syntaxique : si une définition n'est pas correcte, l'analyseur après avoir signalez l'erreur continuera son analyse à partir de la définition suivante.

Étape II : le traducteur

Le principe est simple : chaque définition introduite par le mot clé `type` donnera éventuellement lieu à un suite de définitions de types dans le langage C (introduites par des `typedef`s). Plus précisement on traduira chaque *somme* par une `union` du langage C, et chaque *produit* par une `struct` du langage C. Nous introduisons des `typedefs` pour aiser l'utilisation des types.

Remarque. Pour simplifier le processus de traduction (ainsi que le processus vérification de la cohérence des définitions) il sera outil d'utiliser une table des symboles : le lexeur ajoutera chaque $IDENT$ et chaque $CONSTRUCTUER$ dans cette table. Chaque symbole pourra avoir un nombre d'attributs que l'on utilisera pour la traduction et la vérification.

Étape III : vérification de la cohérence des définitions

On mettra en ouvre une étape de vérification de la cohérence des définitions, visée à sélectionner les types et constructeurs qui respectent les critères suivants :

1. Chaque type et chaque constructeur sera défini une seule fois.

   Exemple. Considérez les définitions suivantes :

   ```
   type couple = Couple of int*int;
   type complexe = Couple of real*real;
   ```

   Le type `complexe` n'est pas bien défini car le constructeur `Couple` apparaissant dans sa définition est défini auparavant.

2. Dans le corps d'une définition inductive chaque identificateur doit être défini avant, ou être défini par la même définition mutuelle.

   Exemple. Dans la suite de définitions

   ```
   type abc = int;
   type def = Triplet of abc*def*ghi;
   type pair = Zero | Succ of impair
      and
        impair = Autre_succ of pair;
   ```

   la définition du type `def` n'est pas valide, car le type `ghi` n'a pas été défini avant, donc sa définition n'est pas complète. Par contre le type `pair` est bien défini, car le type `impair` dans le corps de sa définition est défini dans la même définition mutuelle.

Le traducteur traduira seulement les types dont la cohérence a été vérifiée, et imprimera des messages d'erreur autrement.

Étape IV : construction du pre-preprocesseur

Un fichier `.tic` sera un fichier C où on pourra inclure des définition de types par des mots clés, par exemple :

```
/* BEGIN TIC */
     .
     .
     .
/* END TIC */
```

À l'intérieur de ces mots clés on trouvera les définitions de types.

Étape V : discussion sur le travail à suivre

Proposez un discussion (2 pages au plus en pdf) sur le travail nécessaire pour compléter le projet et se servir de l'outil dans la pratique. Notamment, des expressions telles que

```
Noeud("Hallo",Feuille(1),Noeud("Byebye",Feuille(2),Feuille(3)))
```

peuvent se traduire maintenant par

```
make_arbre_of_Noeud(
  make_Noeud("Hallo",
    make_arbre_of_Feuille(make_Feuille(1)),
    make_arbre_of_Noeud(
      make_Noeud("Byebye",
        make_arbre_of_Feuille(make_Feuille(2)),
        make_arbre_of_Feuille(make_Feuille(3))))));
```

mais ce processus reste assez complexe à se faire à la main.
Détaillez donc un projet de travail pour porter à terme le projet, ce qui veut dire pouvoir se servir des outils programmé dans la pratique quotidienne de la programmation. Si vous avez le temps, débutez de projet.

Modalités d'évaluation

- Le travail se déroulera en binôme. Afin de stimuler la collaboration entre étudiants différents, un bonus (2 pts) sera donné aux binômes différents de toute autre binôme du 6ème semestre.
- Bien que le travail soit en binôme, la note est individuelle et elle n'est pas nécessairement la même pour les deux membres du binôme.
- Les membres du binômes seront communiqués au responsable de la formation au plus tôt (avant le vendredi 12/3/2010), pour pouvoir mettre en place des répertoires svn. Depuis la mise en place de ces répertoires, les étudiants sauveront leurs projets dans ces répertoires (l'utilisation du dépôt svn est obligatoire).
- Le projet sera évalué aussi par rapport à l'assiduité et la constance. L'histoire du projet, disponible par l'outil svn, servira à ce fin.
- Chaque étape fera partie d'une évaluation. Préparez du matériel pour tester chaque étape lors de la soutenance. Pour chaque étape, préparez une demo (code source et autre matériel) dans le dépôt svn, avec une tag.
- Date de la soutenace : à préciser. (Autour du 25 mai).

Rappel : La présence à la soutenance est obligatoire. Tout étudiant ne se présentant pas à la soutenance, sans justification, se verra attribué la note « absence injustifiée ». La note du projet n'est pas rattrapable.