TP7: premier contact avec un squelette

David Baelde

4 et 5 novembre 2015

On commence à travailler avec la structure du projet. La partie "frontale" du compilateur est déja écrite : il s'agit du *lexer* (analyse lexicale) et du *parser* (analyse syntaxique). Cette partie restera probablement assez mystérieuse pour vous pour l'instant, et ne s'éclaircira qu'au second semestre avec le cours de langages formels. Ce qu'il faut retenir est que la base de code fournie permet d'extraire un "arbre de syntaxe abstraite" à partir d'un code source C--. L'objectif de ce TP est d'exploiter ce code pour afficher correctement cet arbre sous la forme d'un docuement XML. Vous pourrez ensuite, par exemple, visualiser cet arbre dans un navigateur web.

Exercice - 1 *Premier contact*

1- Récupérer sur la page des TP l'archive contenant le squelette de code du compilateur :
`http://www.lsv.fr/~baelde/prog1/projet.tar.gz`.
Décompressez l'archive au moyen de la commande `tar zxvf projet.tar.gz`. Les fichiers sont dans le sous-répertoire `ProjetMiniC` dans lequel vous devrez travailler…jusqu'à la fin du semestre ! Faites donc attention à ne pas trop le salir, faire des sauvegardes, etc.

2- Allez voir sur la page du projet chez Jean Goubault, cf. liens à partir de la page du TP. En particulier la Section 2 (fonctionnement général) et Section 4 (architecture, détails de l'AST). (Ces pages peuvent encore contenir des traces d'assembleur 32 bit, anciens noms de chargés de TP, etc. c'est normal. Il y a aussi de légères différences entre la base de code décrite et celle que je vous fournis.)

3- La base de code fournie comprend un fichier `cprint.ml` qui définit notamment la fonction print_declarations . Celle-ci est appelée quand on invoque le compilateur avec l'option `-D`, par exemple :
`./mcc -D Exemples/sieve.c`
Commencez par écrire le corps de cette fonction, pour qu'elle affiche les noms des différentes déclarations de fonction. L'output pourra par exemple ressembler à ça :

```
add (x, y)
sub (x, y)
print_tab (tab, size)
```

Exercice - 2 *Le module `Format`*

Le module `Format` est en quelque sorte un `Printf` très enrichi, permettant d'afficher du texte structuré via les concepts de boîtes et coupures. Pour l'apprivoiser, un tutoriel est disponible ; on pourra aussi voir la page de référence.

1- Écrivez (dans un fichier quelconque) une fonction pp_nested (pour "pretty-print nested") qui affiche des listes imbriquées de chaînes de caractères, en indentant de façon raisonnable. Le type des listes imbriquées que vous utiliserez doit être :

```
type nested =
  | F of string
  | I of nested list
```

Votre fonction devra prendre un formatter et une liste imbriquée, et s'utilisera par exemple ainsi :

```
let bc = I [F "bbbbb"; F "ccccccccccccc"]
let abcbc = I [F "aaa"; bc; bc]
let l = I [abcbc;bc;abcbc]
let () =
  pp_nested Format.std_formatter l ;
  Format.print_newline () ;
  (* Ou encore, de facon equivalente *)
  Format.printf "Voila:_%a@." pp_nested l
```

Exercice - 3 *AST vers XML*

Dans cet exercice, vous devez générer un document XML représentant l'arbre de syntaxe abstraite d'un programme. XML est un langage générique utilisé massivement dans différents domaines, par exemple pour écrire des pages internets (XHTML), des dessin vectoriel (SVG), des textes formattés (OpenDocument), etc.

En XML, on représente un noeud d'un arbre en utilisant des tags : `<tag>contenu</tag>`. Le contenu de ce noeud peut être une succession d'autres noeuds XML ou de texte. Vous devez faire attention à bien refermer les balises (les balises *fermantes* commencent par un "/"). Vous devez refermer les balises en suivant l'ordre (inverse) d'ouverture. Par exemple `<a><b>Coucou</a></b>` est mal formé. Pour utiliser les symboles <,> et & dans du texte vous devez utiliser respectivement &lt, &lg et &amp.

Un noeud XML peut avoir des *attributs* (en plus de som nom) :

```
<chapitre class="bibliographie">contenu</chapitre>.
```

1- Écrivez le corps de la fonction print_locator du fichier `cprint.ml` qui affiche les attributs d'un locator (cf. premier type dans le fichier `error.ml`). L'output attendu est de la forme :

```
file="tests/cat.c" first-line= "26" first-column="8"
                last-line="26" last-column="9"
```

2- Ecrivez le corps de la fonction print_ast qui affiche l'arbre en XML indenté correctement via `Format`. Un exemple de document est donné en Figure 1. Testez avec l'option `-A`. Par exemple, vous pouvez enregistrer votre document XML avec `./mcc -A Exemple\slash cat.c > doc.XML` et l'ouvrir avec un navigateur internet.

Exercice - 4 *(Bonus) ASCII art - c'est (aussi) joli*

1- Ecrivez une autre version de print_ast qui affiche l'arbre en ASCII art. Un exemple de bel arbre est donné en Figure 2, un autre en Figure 3. Laissez libre cours à votre créativité et imagination ;)

Fichier source :

```
int
main (int argc, char **argv)
{
  int a;
  return a + 1;
}
```

Rendu :

```
<cfun name="main" file="tests/test.c" first-line="2" first-column="0"
      last-line="2" last-column="4">
  <args>
    <cdecl name="argc" file="tests/test.c" first-line="2" first-column="10"
           last-line="2" last-column="14"/>
    <cdecl name="argv" file="tests/test.c" first-line="2" first-column="23"
           last-line="2" last-column="27"/>  </args>
  <cblock
    file="tests/test.c" first-line="3" first-column="0" last-line="6"
    last-column="1">
    <cdecl name="a" file="tests/test.c" first-line="4" first-column="6"
           last-line="4" last-column="7"/>
    <creturn
      file="tests/test.c" first-line="5" first-column="2" last-line="5"
      last-column="14">
      <add
        file="tests/test.c" first-line="5" first-column="9" last-line="5"
        last-column="14">
        <var name="a" file="tests/test.c" first-line="5" first-column="9"
             last-line="5" last-column="10"/>
        <cst value="1" file="tests/test.c" first-line="5" first-column="13"
             last-line="5" last-column="14"/></add></creturn></cblock>
  </cfun>
```

Figure 1 – Rendu attendu pour un programme simple

```
[ Seq ]
   +-----[ Seq ]
   |         +-----[ Seq ]
   |         |         +-----[ Label ]-- func1
   |         |         \-----[ Seq ]
   |         |                 +-----[ Seq ]
   |         |                 |         +-----[ Nop ]
   |         |                 |         \-----[ Seq ]
   |         |                 |                 +-----[ Label ]-- __RUN__func1__
   |         |                 |                 +-----[ Move ]
   |         |                 |                 |       +-----[ Temp ]--  0
   |         |                 |                 |       +-----[ Const ]--  1
   |         |                 |                 |       \-----[ Temp ]-- 2
   |         |                 |                 +-----[ Label ]-- __RUN__func1__
   |         |                 |                 \-----[ Move ]
   |         |                 |                         +-----[ Temp ]--  0
   |         |                 |                         \-----[ Const ]--  1
   |         |                 +-----[ Seq ]
   |         |                 |         +-----[ Nop ]
   |         |                 |         \-----[ Seq ]
   |         |                 +-----[ Seq
   |         |                 |         +-----[ Nop ]
   |         |                 |         \-----[ Seq ]
   |         |                 \-----[ Fin_fun ]

  ...       ...
```

Figure 2 – Exemple de rendu attendu pour l'arbre en ASCII

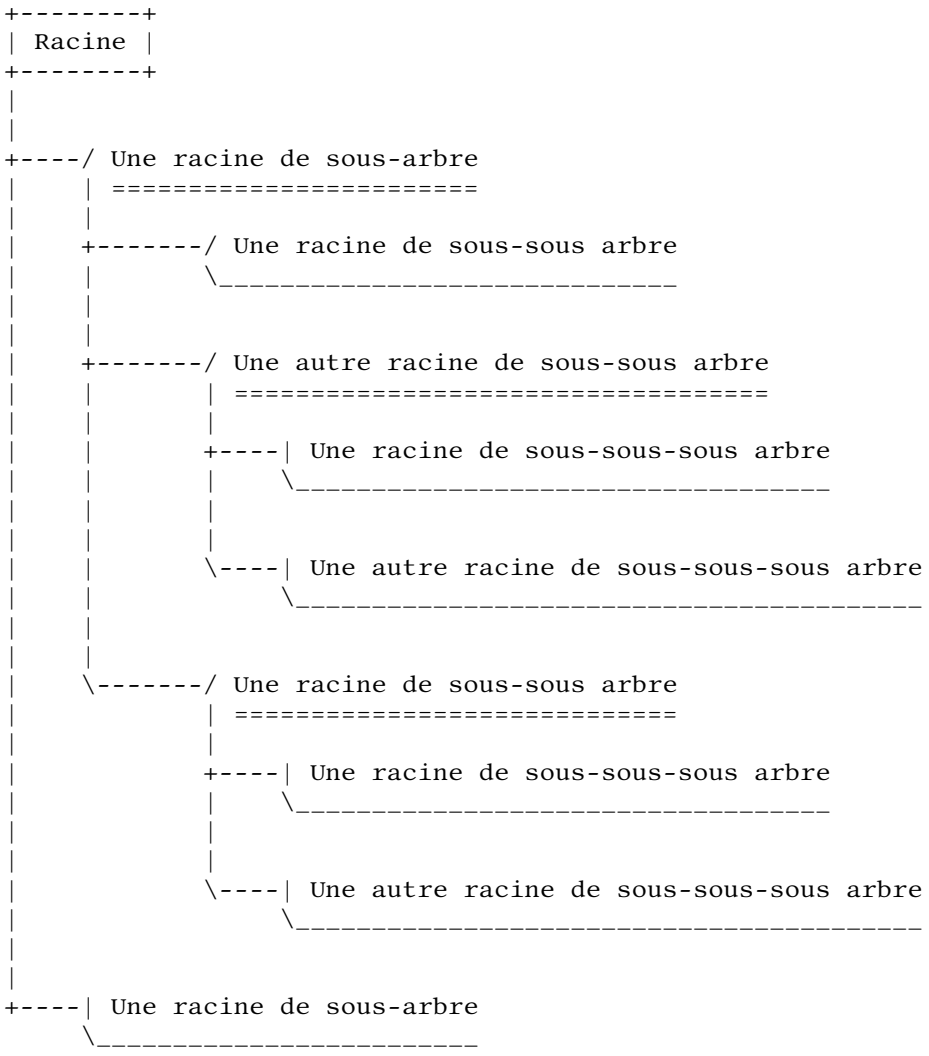

Figure 3 – Autre exemple de rendu pour l'arbre en ASCII