Projet AS & PP3 2016

Table des matières

1 Organisation du projet **1**

2 Conseils d'organisation du travail **2**

3 Description du projet **2**

4 Langage de programmation fonctionnel pour manipuler des documents **2**
4.1 Syntaxe pour les arbres 3
4.1.1 Quelques conventions pour la syntaxe 4

5 Structures de contrôle et fonctions **6**
5.1 Primitives de programmation fonctionnelle 6
5.1.1 Déclarations et définitions 6
5.1.2 Structures des expressions 8
5.1.3 Fonctionnalités à ajouter 11

1 Organisation du projet

- Le projet est à réaliser par groupes de 4 ou 5 étudiant(e)s. Chaque membre doit contribuer de façon significative au travail de programmation, avoir connaissance des options choisies, des difficultés rencontrées, et des solutions retenues, et exposera sa contribution individuellement lors d'une soutenance de 25mn (soutenances prévues la semaine du 2 mai).

- On demande également un rapport court (5 à 10 pages), présentant le travail réalisé, illustré par des exemples, exposant les problèmes rencontrés et les choix effectués. Idéalement, le rapport sera présenté au format `html`, et pourra avoir été généré partiellement par votre logiciel.

- Le projet doit être transmis par mail à l'adresse aspp3@labri.fr, avec comme sujet [**Projet ASPP3**] **Nom1 Nom2 Nom3 Nom4** (dans le cas d'un groupe de 4 personnes). Le message devra contenir une archive compressée nommée `Nom1-Nom2-Nom3-Nom4.tar.gz`. Cette archive une fois décompressée contiendra un **répertoire** nommé Nom1-Nom2-Nom3-Nom4, qui lui-même contiendra les fichiers.

- Vous fournirez un `makefile` pour compiler le projet.

- Vous indiquerez dans le rapport un dépôt `git` (ou à défaut `svn`). Seuls les membres du groupes auront accès en écriture à ce dépôt. Seuls les membres du groupes et les enseignants de l'UE auront accès en lecture.

- Dates de remise du projet, code et rapport (attention, date **stricte**) : 29 avril 2016, 18h00.

- La notation tiendra compte du degré de réalisation du sujet, de la qualité du code (portabilité, lisibilité, documentation), du jeu de tests présenté, et de la qualité du rapport et de la soutenance. Elle peut varier à l'intérieur d'un groupe en cas de travail trop inégal.

2 Conseils d'organisation du travail

La construction d'un langage de programmation pose de nombreux problèmes et nécessite une progression **par étapes** :

- Écrire d'abord l'analyseur lexical ; le projet pose des difficultés liées à l'analyse lexicale. Il vous faudra bien lire le sujet pour repérer quels sont les éléments lexicaux utiles au langage de programmation.
- Écrire l'analyseur syntaxique ; la grammaire du langage du projet est complexe. Il vaut mieux la construire **de façon incrémentale** en ajoutant petit à petit de constructions syntaxiques. Il sera parfois nécessaire de distinguer au sein d'une même construction syntaxique plusieurs situations mutuellement exclusives. Par exemple, il semble nécessaire de distinguer les forêts qui se terminent par une expression, des autres.
- Il faut écrire le rapport **au fur et à mesure** de l'avancement du projet afin d'y faire apparaître et de documenter les choix d'implémentation que vous avez faits.

3 Description du projet

La plupart des sites web sont désormais dynamiques : le contenu des pages est calculé par le serveur à chaque demande. Cependant, bien peu de sites web présentent un contenu qui évolue rapidement ou qui contiendrait un volume d'information nécessitant une évaluation dynamique de leurs contenus. La raison pour laquelle ces sites sont écrits de façon dynamique est qu'ils présentent la *même information sous plusieurs formes*.

Par exemple, un blog peut être vu comme une série d'articles, et un *site* de blog présente ce contenu en faisant des regroupements thématiques, en montrant les titres et début des derniers articles écrits, en présentant une chronologie générale des articles, etc. Il est par ailleurs bien rare qu'un blog publie régulièrement plus d'une dizaine d'articles par jour, aussi son contenu évolue-t-il généralement lentement. Présenter un tel site en utilisant une évaluation dynamique des pages l'expose plus facilement à des attaques qui viseraient les lacunes de configuration du serveur. Par ailleurs, si le blog est hébergé sur un serveur ayant une faible capacité de calcul, l'accès à toute page peut être ralenti par l'évaluation qu'elle nécessite.

Ce projet propose d'implémenter un compilateur de site web. Il s'agit de créer un langage de programmation qui permettra de décrire les pages web et d'éviter les redondances d'information, en séparant le contenu de la présentation elle-même. Ainsi, un blog pourra être présenté comme un ensemble d'articles et d'instructions de présentation à partir desquels les pages du site seront générées. Il suffira ensuite de les télécharger sur le serveur pour qu'elles soient publiées sous une forme statique.

4 Langage de programmation fonctionnel pour manipuler des documents

Le langage de programmation que nous proposons de construire manipulera deux types de données :

- un type numérique qui sera composé des entiers et représenté en machine par le type `int` de C,

- un type *document* qui sera composé d'arbres (plus précisément de forêts : une forêt est une séquence d'arbres) représentant des documents XML.

Ce langage sera fonctionnel et manipulera donc des structures persistantes (*i.e.*, qui ne seront pas modifiées au cours de l'exécution). Une partie importante du projet portera sur l'implémentation de méthodes de sélection et de manipulation avancées pour les documents arborescents.

4.1 Syntaxe pour les arbres

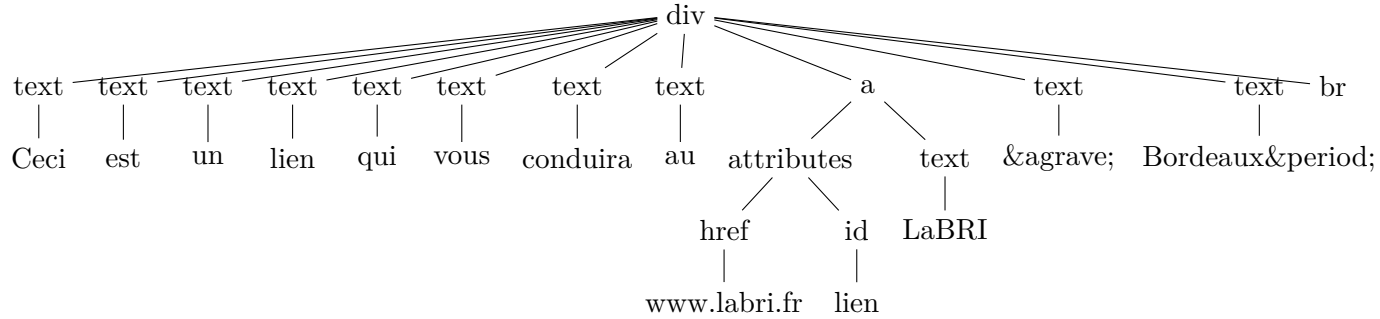
Les documents HTML et plus généralement les documents XML représentent des arbres étiquetés. Pour représenter ce type de documents, nous adopterons la syntaxe illustrée par l'exemple suivant :

```
div{
    "Ceci est un lien qui vous conduira
    au " a[href="www.labri.fr" id="lien"]{"LaBRI"} "
    à Bordeaux."
    br/
   }
```

qui représente le code HTML suivant :

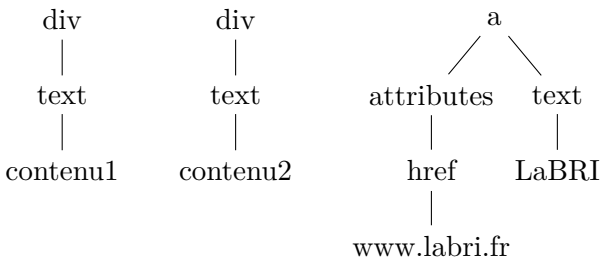
```
<div>
  Ceci est un lien qui vous conduira
  au <a href="www.labri.fr" id="lien">LaBRI</a>
  à Bordeaux&period;
  <br/>
</div>
```

et qui aura la représentation interne suivante pour le langage de programmation que nous construisons :



Plus généralement, on pourra représenter les *forêts* (c'est-à-dire, les séquences d'arbres) en utilisant simplement des accolades. Ainsi, l'expression

```
{div{"contenu1"} div{"contenu2"}
    a[href="www.labri.fr"]{"LaBRI"}
}
```



dénote simplement la forêt

Il sera possible d'écrire les forêts en incluant d'autres forêts (éventuellement vides) :

```
{{div{"contenu1"} div{"contenu2"}}
   a[href="www.labri.fr"]{"LaBRI"}
   {{} {p{"contenu3"}}
}
```

Cette écriture est équivalente à la notation plus simple :

```
{div{"contenu1"} div{"contenu2"}
   a[href="www.labri.fr"]{"LaBRI"}
   p{"contenu3"}
}
```

4.1.1 Quelques conventions pour la syntaxe

Les étiquettes qui peuvent être mises sur les nœuds doivent obéir aux conventions de XML :

- Elles doivent commencer par une lettre ou un caractère '_' (mais '_' seul n'est pas accepté comme étiquette possible).
- Elles ne doivent pas commencer par une séquence de la forme xml, XML, Xml, etc.
- Elles peuvent contenir des lettres, des chiffres, des apostrophes, des points, des caractères '_'.
- Elles ne doivent pas contenir d'espace.

Cette convention s'applique également aux noms des attributs.

Par ailleurs, dans la syntaxe qui permet d'écrire un arbre, nous imposons que :

- Le nom de l'étiquette soit immédiatement accolé (sans espace ou saut de ligne) soit à :
  - un crochet ouvrant '[' au cas où l'on souhaite associer une liste d'attributs au nœud,
  - une accolade ouvrante '{' pour écrire le contenu du nœud,
  - un symbole '/' si l'on souhaite décrire un nœud qui ne peut avoir de contenu (NB : `foo/` représente la balise `<foo/>` alors que `foo{}` représente la balise `<foo></foo>`).
- Si un nom d'étiquette est suivi d'une liste d'attributs entre crochets, alors :
  - il peut y avoir des espaces et des sauts de lignes entre la fin de la déclaration des attributs et l'accolade ouvrante qui permet de décrire le contenu du nœud,
  - s'il s'agit d'un nœud qui ne peut pas avoir de contenu, on impose que le '/' soit directement positionné après le crochet fermant.
- On autorise également d'écrire une forêt entre accolades sans qu'il y ait besoin d'une balise pour ouvrir la racine.

Pour résumer, les formes **licites** sont les suivantes :

```
div{
"contenu..."
}
div[att1="v1"
    att2="v2"]
```

```
    {
    "contenu..."
    }

br[att1="v1"
   att2="v2"]/

{"contenu ... " div[class="nb"]{...}
 "suite ..."}
```

et voici un ensemble de formes **illicites** :

```
div

  {"contenu ..."}

div    [att1="v1" att2="v2"]{"contenu"}

br[att1="v1" att2="v2"]   /
```

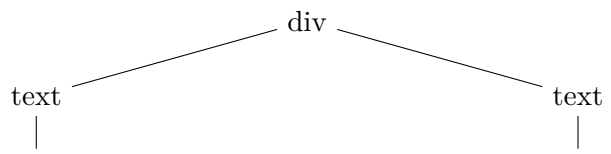
Comme vous pouvez le noter dans tous ces exemples :

- Les valeurs associées aux attributs sont toujours entourées par une paire de guillemets doubles (`"..."`).
- On peut utiliser des espaces et des sauts de lignes de façon libre dans la déclaration des attributs.
- Les contenus textuels d'une balise sont toujours entre guillemets doubles. Les espaces et les sauts de lignes servent seulement à séparer deux mots consécutifs et ont donc peu d'importance.

On utilisera les séquences d'échappement usuelles pour les caractères spéciaux dans les contenus textuels des balises. Il faudra également traduire automatiquement les caractères représentés par une entité de caractère XML/HTML. Par exemple, le code suivant :

```
div{"<a href=\"www.labri.fr\"></a>"}
```

représente l'arbre suivant :



Vous pourrez trouver une liste exhaustive des entités XML/HTML ici.

Dans cette partie préliminaire nous vous proposons d'utiliser la structure de donnée `C` suivante pour modéliser les arbres :

```c
#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>

struct tree;
struct attributes;
enum type {tree, word};        //typage des nœuds: permet de savoir si un nœud construit
```

```
                                //un arbre ou s'il s'agit simplement de texte

struct attributes{
    char * key;               //nom de l'attribut
    char * value;             //valeur de l'attribut
    struct attributes * next; //attribut suivant
};

struct tree {
    char * label;              //étiquette du nœud
    bool nullary;              //nœud vide, par exemple <br/>
    bool space;                //nœud suivi d'un espace
    enum type tp;              //type du nœud. nullary doit être true s tp vaut word
    struct attributes * attr;  //attributs du nœud
    struct tree * daughters;   //fils gauche, qui doit être NULL si nullary est true
    struct tree * right;       //frère droit
};
```

Vous devez réaliser les tâches suivantes :

- Écrire un analyseur qui permette de lire un fichier utilisant la notation convenue pour les arbres, de façon à représenter le fichier dans la structure de données C proposée ci-dessus,
- Écrire une fonction qui permette de transcrire cette structure de données dans un fichier au format XML.

NB : dans la description que nous avons donnée, nous n'avons pas précisé comment différencier les deux arbres suivants :

```
div{"contenu " a[href="www.labri.fr"]{"LaBRI"}"."}
div{"contenu" a[href="www.labri.fr"]{"LaBRI"}"."}
```

qui représentent les deux codes HTML suivants :

```
<div>contenu <a href="www.labri.fr">LaBRI</a></div>
<div>contenu<a href="www.labri.fr">LaBRI</a></div>
```

Le champ `space` de la structure de donnée doit être utilisé à cet effet.

5 Structures de contrôle et fonctions

Nous allons maintenant enrichir la syntaxe pour qu'elle permette de manipuler les arbres et les nombres afin d'avoir des moyens puissants d'extraire et de présenter les informations nécessaires à la construction d'un site web.

5.1 Primitives de programmation fonctionnelle

5.1.1 Déclarations et définitions

Les noms de variables et de fonctions qui pourront être utilisés dans ce langage de programmation suivent les mêmes conventions que les étiquettes des arbres à **l'exception notable du fait qu'ils peuvent commencer par les lettres** `xml`.

Un ficher pourra contenir un ensemble de déclarations de variables et de fonctions qui seront définies par une succession de déclarations de la forme :

```
let v1 = ...;
let v2 = ...;
```

pour les variables. Les déclarations de fonctions seront de l'une des formes suivantes :

```
let f v1 v2 = ...;
let g  = fun z t -> ...;
let f x1 ... xn = fun y1 ... yp ->...;
let rec h x1 ... xn = fun y1 ... yp -> ...;
```

La notation `let rec` permet de définir des fonctions/structures récursives. Bien entendu, ces déclarations pourront faire appel aux déclarations précédentes et pourront s'écrire sur plusieurs lignes.

Comme le langage doit permettre de générer des fichier `html`, il est également possible d'écrire des *actions* et ainsi parmi les déclarations, on pourra également écrire des expressions qui ne sont par précédées par un construction `let`. Ainsi, il est possible d'écrire :

```
let v = e1;
let f g = e2;
e3;
let u = e4;
```

où `e1`, `e2`, `e3` et `e4` sont des expressions du langage.

La seule action que l'on s'autorise pour le moment est donnée par la fonction `emit` qui prend deux arguments, un nom de fichier et une expression et qui écrit le contenu du document sous la forme de document XML dans le fichier. Par exemple, l'action (pour la syntaxe de l'application de fonction voir la section Application de fonctions) :

```
emit "index.html" {html{header{} body{h1{"Nouvelle Page"}}}}
```

écrit le document suivant :

```
<html>
  <header>
  </header>
  <body>
    <h1>Nouvelle page</h1>
  </body>
</html>
```

dans le fichier `index.html`.

Un fichier sera composé de deux parties :

1. une séquence de déclarations de variables et de fonctions,
2. une séquence d'arbres qui constituent une forêt, qui pourra utiliser les variables et les fonctions précédemment définies.

Par exemple, on pourra avoir le document suivant :

```
let LaBRI = a[href="www.labri.fr"]{"LaBRI"};
let bdx = a[href="http://www.u-bordeaux.fr/"]{"Université de Bordeaux"};
let email= a[href="mailto:author@bdx.fr"]{"email"};

div{"Le " LaBRI, " est un laboratoire de l'" bdx, "."}

div{"Pour m'écrire utilisez l'adresse " email, "."}
```

Notez la *virgule* qui sert à séparer la variable de la suite du contenu. Cette *virgule* est optionnelle s'il n'y a pas de contenu à la suite de la variable.

5.1.2 Structures des expressions

Nous allons maintenant aborder la structure des expressions qui permettent de définir les variables, les fonctions ou encore un document en utilisant des variables ou des fonctions définies par ailleurs.

Les expressions auront les propriétés suivantes :

- toute expression arithmétique est une expression,
- on peut mettre autant de parenthèses que l'on souhaite autour d'expressions,
- toute description d'arbre est une expression,
- toute construction doit pouvoir être utilisée pour définir une expression.

1. Variables locales

   Dans les expressions, il sera possible d'utiliser des variables locales afin de réutiliser fonctions et expressions au sein même de la définition d'une expression.

   Pour les déclarations de variables, on utilisera des expressions de la forme :

   ```
   let ubx = a[href="https://www.u-bordeaux.fr/"]{"Université de Bordeaux"} in
       div{"L'" ubx, " a été créée en 1441 par le pape Pape Eugène IV à
                     l'initiative de Pey Berland - archevêque de Bordeaux de
                     1430 à 1456 - l'"
                ubx, "comptait quatre facultés : art, médecine,
                     droit et théologie."}
   ```

   ou encore de la forme :

   ```
   div{"L'" ubx, " a été créée en 1441 par le pape Pape Eugène IV à
                   l'initiative de Pey Berland - archevêque de Bordeaux de
                   1430 à 1456 - l'"
           ubx, "comptait quatre facultés : art, médecine,
                   droit et théologie."}
     where ubx = a[href="https://www.u-bordeaux.fr/"]{"Université de Bordeaux"}
   ```

   Des déclarations pourront être celles de fonctions voire de fonctions récursives en utilisant le mot clef `rec` à l'endroit convenable (après le `let` ou le `where`).

2. Application de fonctions

   Pour appliquer une fonction à ses arguments, nous utiliserons la syntaxe suivante :

   ```
   (...((f a1) a2) ... an)
   ```

   Cela signifie que le premier argument de `f` est `a1` et que l'on passe au résultat de cette application l'argument `a2`, etc… jusqu'à l'argument `an`.

   Il faudra également pouvoir appliquer les fonctions sans avoir à mettre toutes ces parenthèses et permettre d'écrire des expressions de la forme :

   ```
   f a1 a2 ... an
   ```

Dans l'exemple ci-dessous, on s'attend à ce que `(fun x1 x2 -> ...)` ait pour premier argument `(fun y -> ...) a`, et, respectivement, pour second, troisième et quatrième arguments `b`, `c`, et `d`.

```
(fun x1 x2 -> ...) ((fun y -> ...) a) b c d
```

Par ailleurs, on souhaite que la priorité maximale soit donnée à l'application par rapport à toutes les autres constructions. Par exemple dans le code suivant :

```
f a e + g b c d
```

on effectue la somme des résultats de l'application de `f` à `a` et `e` et de l'application de `g` à `b`, `c` et `d`.

De façon similaire, l'expression suivante

```
div{"contenu ..." f p{"contenu'...." ...}}
```

se parenthèse de la sorte :

```
div{"contenu ..." (f p{"contenu'...." ...})}
```

3. Conditionnelles et expressions booléennes

   Nous allons adopter la syntaxe suivante pour les conditionnelles :

   ```
   if e then ... else ...
   ```

   L'expression `e` qui sert à choisir la branche de la conditionnelle à exécuter devra s'évaluer ou bien en un entier ou alors en un arbre. Si l'entier est différent de 0 ou si l'arbre est non vide, alors la branche "`then`" sera exécutée, sinon ce sera la branche "`else`".

   Nous allons maintenant donner des expressions, et en dessous la façon dont on souhaite qu'elles soient implicitement parenthésées par l'analyseur.

   ```
   if e1 then e2 else if e3 then e4 else e5
   (if e1 then e2 else (if e3 then e4 else e5))

   if e1 then e2 else e3+e4
   (if e1 then e2 else (e3+e4))

   if e1 then e2 else e3 e4
   (if e1 then e2 else (e3 e4))
   ```

   N'oubliez pas que l'on souhaite pouvoir écrire des choses comme celles-ci :

   ```
   div{"contenu ..." let a = (if e then f else g) b in
                         if e' then h a else k a,
                     foo 4,
       "suite ..."
     }
   where foo n = ...
   ```

   NB : il n'y a pas la possibilité d'écrire une expression conditionnelle sans la clause `else`. Ainsi les expressions de la forme :

   ```
   if e then f
   ```

ne sont pas licites.

Afin de pouvoir construire des expressions conditionnelles, il nous faut pouvoir construire des expressions booléennes. Pour cela, il est possible de comparer des expressions numériques, des arbres ou des forêts. Les opérateurs de comparaisons sont `<`, `>`, `>=`, `<=`, et `==`. On peut également comparer des expressions booléennes entre elles. Enfin, on peut utiliser des connecteurs logiques comme la conjonction `&&`, la disjonction `||` ou la négation `!`. Les priorités et l'organisation de ces opérateurs sont les mêmes que celles du langage `C`.

4. Opérations sur les arbres et filtrage

   Nous allons maintenant introduire un ensemble de constructions qui vont permettre de manipuler les arbres.

   La première construction que nous allons utiliser permet de destructurer les arbres et les forêts afin d'en extraire certaines parties. La syntaxe a la forme suivante :

```
match exp with
| p1 -> e1
| p2 -> e2
...
| pn -> en
end
```

   `exp` est l'arbre que l'on cherche à déstructurer et pour cela on utilise les *filtres* `p1`, `p2`, …, `pn`. Les filtres permettent de choisir quelle branche, `e1`, `e2`, …, `en`, exécuter en fonction de la forme de `exp`. De plus ils permettent d'extraire des sous-éléments de `exp` qui peuvent être utilisés dans l'évaluation du `e1`, …, `en` à exécuter.

   Nous allons maintenant décrire la syntaxe des filtres :

   - Le symbole '_' permet de filtrer n'importe quel arbre où chaîne
   - En utilisant une construction de la forme `a{ ... }` on filtre tout arbre dont la balise de sommet est `a`, et on peut continuer à filtrer à l'intérieur de la forêt sous-jacente. On peut également utiliser `_{ ... }` pour filtrer un arbre avec une balise de sommet quelconque.
   - On peut utiliser l'expression `*_*` pour filtrer toute la partie textuelle au début d'une forêt, et `/_/` pour filtrer tous les arbres au début d'une forêt.
   - enfin on peut utiliser des variables qui peuvent filtrer un arbre ou une forêt. En particulier on peut utiliser `*x*` et `/x/` pour assigner dans `x` ou bien toute la partie textuelle au début d'une forêt, ou alors tous les arbres au début d'une forêt.

   Par exemple, dans le programme suivant

```
match exp with
| a{z} -> c{"contenu" z}
| b{_ _{x} {y}} -> {x, y}
end
```

   Si `exp` est de la forme

```
a{"texte"}
```

   alors le résultat sera

```
c{"contenu" "texte"}
```

Si, en revanche, `exp` est de la forme

```
b{"mot " c{"contenu de c" a{contenu}}
b{contenu} e{contenu}}
```

alors elle s'évaluera en la forêt

```
"contenu de c" a{contenu} b{contenu} e{contenu}
```

Il faut bien noter ici que `x` a pour valeur

```
"contenu de c" a{contenu}
```

et que `y` a pour valeur

```
b{contenu} e{contenu}
```

5.1.3 Fonctionnalités à ajouter

Il manque quelques fonctionnalités à ajouter au langage :

- Un moyen de gérer les liens internes entre les fichiers générés qui garantissent leur existence,
- Une extension du langage pour pouvoir générer et manipuler des fichiers CSS,
- Un moyen d'extraire l'information contenue dans les attributs afin de pouvoir, par exemple, utiliser leur valeur dans des conditionnelles,
- Toute autre chose qui vous semblerait pertinente.

Implémenter ces fonctionnalités dans le langage et justifier les choix syntaxiques et l'implémentation que vous avez effectués.