Université Mohamed Premier
Faculté des Sciences
Oujda

Filière SMI-S5
Année universitaire 2014-2015

Module Compilation
Mini-Projet

Problème

L'objectif de ce mini-projet est la réalisation (en Flex et Bison) d'un interpréteur pour les expressions régulières sur l'alphabet $A = \{a,\ b, \}$. Cet interpréteur doit assurrer au minimum les opérations suivantes :

- vérifier si une expression tapée est une expression régulière valide ;
- calculer la taille d'une telle expression ;
- représenter en mémoire une expression régulière sous la forme d'un AST ;
- afficher l'AST d'une expression régulière ;
- calculer l'AST du résiduel d'une expression régulière ;
- tester si le langage dénoté par une expression régulière contient le mot vide, contient un mot donné, est vide, fini, infini, est un simple symbole, ou est réduit à un seul mot ;

Tout d'abord, il faut chercher une grammaire hors-contexte non-ambiguë G qui engendre les expressions régulières sur l'alphabet $A = \{a,\ b\}$, en supposant que :

- la concaténation (opérateur invisible), la réunion '|' et l'étoile '$*$' sont associatives à gauche ;
- l'étoile '$*$' est plus prioritaire que le produit, lui-même plus prioritaire que la réunion '|' ;
- le symbole 1 désigne ε et le symbole 0 représente l'ensemble vide $\emptyset$;

On rappelle qu'un **AST** (**arbre de syntaxe abstrait** en Français et *Abstract Syntax Tree* en Anglais) est une représentation **réduite** de l'arbre de dérivation : il ne stocke que les éléments necéssaires à l'analyse sémantique. Votre analyseur syntaxique produira un **AST** dont les nœuds sont définis par la structure suivante :

```
struct ast
{
   enum { CONCAT, UNION, ETOILE, A, B, EPSILON, VIDE } type;
   struct ast *fg;  /* fils gauche */
   struct ast *fd;  /* fils droit */
};
```

Le champ *type* indique le type de l'opération dans l'expression régulière. Si le type d'un nœud est **ETOILE**, le champ fd doit être à $NULL$; si c'est A, B, **EPSILON** ou **VIDE**, alors fg

et fd sont tous les deux à $NULL$.

Le code de l'analyseur sera un peu volumineux. Pour cela, il est très efficace (et c'est obligatoire) de le distribuer sur plusieurs fichiers sources :

- *par.y* : qui va contenir le code Bison de l'analyseur syntaxique ;
- *sca.lex* : qui va contenir le code Flex de l'analyseur lexical ;
- *ast.h* : qui va contenir les structures de données et les routines nécessaires pour la création et la manipulation des arbres abstraits ;
- *ast.c* : qui va contenir la définition de ces routines (le code C) ;

Voici comment compiler l'ensemble :

```
bison -d par.y
flex sca.lex
gcc -o reg par.tab.c lex.yy.c ast.c -ly -lfl
```

Voici les fonctions à écrire pour créer l'**AST** :

```
struct ast* creer_a();
struct ast* creer_b();
struct ast* creer_epsilon();
struct ast* creer_vide();
struct ast* creer_union(struct ast *e1, struct ast *e2);
struct ast* creer_concat(struct ast *e1, struct ast *e2);
struct ast* creer_etoile(struct ast *e);
struct ast* copier_ast(struct  ast *e);
```

Écrire également une fonction qui permet de libérer l'espace mémoire occuppée par un **AST** :

```
void liberer_ast(struct ast *e);
```

Dans le but de vérifier qu'un **AST** a été correctement crée, il faut aussi écrire une fonction qui imprime cet arbre. Je vous propose une solution pour ce problème (une solution de votre part sera la bienvenue) : créer un fichier XML à partir de l'arbre de syntaxe. La balise de l'élément racine est <ast> ...< /ast>. Un élément du premier niveau est l'une des trois balises : $< concat > ... < /concat >$, $< union > ... < /union >$, ou $< etoile > ... < /etoile >$. Une feuille est l'une des quatre balises : $< a/ >$, $< b/ >$, $< epsilon/ >$, ou $< vide/ >$.

Par exemple, pour l'expression régulière ab^*, la fonction doit créer le fichier XML suivant :

```
<ast>
   <concat>
      <a/>
      <etoile>
         <b/>
      </etoile>
   </concat>
</ast>
```

Une fois le fichier XML crée, on peut le visualiser en lancant un navigateur Web (IE, Mozilla, Google crome, ...etc) :

Écrire tout d'abord une fonction qui collecte les éléments d'un **AST** dans un tableau de chaînes de caractères :

```
void remplir_elements_ast(struct ast *e);
```

Puis écrire la fonction qui imprime un AST :

```
void imprimer_ast(struct ast *e);
```

Cette dernière créera un fichier XML nommé "ast.xml", puis lui ajoute la balise d'ouverture de l'élément racine $< ast >$, copie le contenu du tableau précédent, puis ajoute la balise de fermeture de l'élément racine $< /ast >$.

Voici les routines de manipulation de l'**AST** (à programmer) :

```
int contient_epsilon(struct ast *e);
/* teste si epsilon appartient à L(e) */
struct ast* residuel(char s, struct ast *e);
/* calcule l'AST du résiduel de e */
int appartient(char *w, struct ast *e);
/* teste si un mot w appartient à L(e) */
int est_vide(struct ast*);
/* teste si L(e) est vide */
int est_epsilon(struct ast*);
/* teste si L(e) = {epsilon} */
int est_symbole(struct ast*);
/* teste si L(e) = {a} ou {b} */
int est_single(struct ast*);
/* teste si L(e) contient un seul mot */
int est_fini(struct ast*);
/* teste si L(e) est fini */
```

Définition du résiduel d'un langage :

Le résiduel d'un langage L par rapport à une lettre a, noté L/a est le langage formé par les mots de L ayant a pour préfixe, auxquels on a supprimé ce préfixe. Formellement :

$$L/a = \{w \in A^* \mid aw \in L\}$$

Par exemple : si $L = \{a,\ abc,\ b\}$, alors $L/a = \{\varepsilon,\ bc\}$. Si r est une expression régulière qui dénote le langage L et x est un symbole, alors le langage L/x peut aussi être décrit par une expression régulière, grâce aux règles suivantes :

- $\emptyset/x = \varepsilon/x = \emptyset$.
- $x/y = \varepsilon$, si $x = y$.
- $x/y = \emptyset$, si $x \neq y$.
- $(rs)/x = (r/x)s$, si $\varepsilon \notin L(r)$.
- $(rs)/x = (r/x)s \mid s/x$, si $\varepsilon \in L(r)$.
- $(r \mid s)/x = (r/x) \mid (s/x)$.
- $r^*/x = (r/x)r^*$.

où r et s sont des expressions régulières sur A, et x et y sont des symboles de A.

Dialogue avec l'utilisateur :

Il faut absolument écrire un analyseur qui respecte les règles suivantes :

- l'utilisateur peut taper une expression régulière tout court :

```
> a|(b|a*)*
> valide, taille = 7
> *a
syntax error
```

- l'utilisateur peut demander si le langage d'une expression régulière contient le mot ε à travers une question de la forme :

```
> Is 0 Nullable ?
> No
> Is 1 Nullable ?
> Yes
> Is a Nullable ?
> No
> Is a* Nullable ?
> Yes
```

- l'utilisateur peut demander si un mot appartient au langage d'une expression régulière à travers une question de la forme :

```
> Is <aba> In a(a|b)*a ?
> Yes
> Is <b> In b(a|b)*b ?
> No
```

Attention, il y aura un problème au niveau de l'analyse lexicale : comment différencier un mot sur $A = \{a,\ b\}$ d'une expression régulière sur le même alphabet ? une solution consiste à entourer un mot par les symboles '<' et '>'.

- l'utilisateur peut demander si le langage d'une expression régulière est vide, est égal réduit au mot vide, est réduit à un seul mot, est fini ou est infini, à travers les questions :

```
> Is 0(a|b)* Empty ?
> Yes
> Is 0* Empty ?
> No
> Is (1|0)* Epsilon ?
> Yes
> Is a(1|0)* Epsilon ?
> No
> Is baa Single ?
> Yes
> Is a* Single ?
> No
> Is a|bab|0* Finite ?
> Yes
> Is (a|bab|0)* Finite ?
> No
> Is a* Infinite ?
> Yes
> Is b(a|bab)aa Infinite ?
> No
```

Pièces à rendre :

Ce mini-projet est individuel et à finir à la maison et à rendre le Jeudi 22 Janvier 2015 à 9H du matin avec les pièces suivantes :

- un CD contenant le code source ;
- un rapport contenant la description de votre travail : il faut expliquer l'algorithme utilisé pour chaque fonction (je dis l'algorithme, et non pas le code ou le pseudo-code).
- Les captures écrans des tests (à joindre avec le rapport).

N.B : Ce mini-projet est noté et représente 30% de la note globale du module. Un travail non rendu à la date et à l'heure précisées sera automatiquement noté par la note 0.