[RAPPORT]

Mini-Project

Interpréteur pour les expressions régulières sur l'alphabet A = {a, b}

Université Mohamed Premier
Faculté des Sciences
Oujda

Réaliser par : KHIATI Wassim

CNE : 1129863367

Année Universitaire 2014-2015

Sommaire pages

I. Introduction :

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 va assurer beaucoup d'opérations, on note une petite partie :

- vérifier si une expression tapée est une expression régulière valide ;
- calculer la taille d'une telle expression ;
- afficher l'AST d'une expression régulière ;
- … ;

Le Code source est un peu volumineux, pour cela il est distribué sur quatre 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) ;

1) Flex :

Flex est Un analyseur lexical qui effectue l'analyse lexicale. Il prend en entrée une séquence de caractères individuels et les regroupe en lexèmes. Lex a été crée en1975 par A. E. Lesk, des laboratoires Bell. La version de GNU est Flex. Il existe une version de Flex pour Ada (Aflex), et une pour Eiffel (Flex/Eiffel). Quant à C++, une version de Lex est disponible avec YACC++.

Flex ajoute des fonctionnalités à Lex (analyse sur plusieurs fichiers, analyse à partir de chaînes en mémoire ... qui ne seront pas exposées ici. Une version acceptant des caractères sur 16 bits existe aussi, Zflex.

2)Bison

Bison est L'analyseur syntaxique qui prend en entrée une suite d'unités lexicales et vérifie qu'elle respecte la grammaire du langage source. Crée en 1974 par S. C. Johnson. La version de GNU est Bison. Il existe des versions pour Ada (Ayacc), une pour Eiffel (Bison/Eiffel). La version C++, avec certaines extensions, est YACC++. Zyacc et BTyacc offrent quant à eux des extensions intéressantes.

Yacc & Bison sont invoqués par les commandes :

yacc options fichier. Ou **Bison options fichier**

L'option –v produit les tables d'analyse sur le fichier *nom.output* L'option –d produit un fichier *nom.tab.h* (*y.tab.h* avec Yacc) qui contient les définitions des lexémes et le type des attributs, pour permettre l'interface avec l'analyseur lexical entre autre.

L'analyseur lexical n'est pas obligatoirement Flex. Il est fréquent qu'il soit programmé à la main, et inclus dans le fichier Bison ou Yacc (Selon l'utilisation).

II. Description du code Flex

J'ai essayé d'écrire le code Flex de façon qu'il prend les mots (lexèmes) nécessaires pour le programme et les renvoyer vert le fichiers Bison.

Pour éviter tout cas d'erreur simple comme oublier l'espace ou que le 1er caractère de chaque mot soit majuscule, le code Flex est programmé à prendre les deux.

Exemple :

```
[ \t]                ;
[Gg]etout       {  yylval.chaine = yytext;     return Getout;    }
[Hh]elp         {  yylval.chaine = yytext;     return Help;      }
```

- *Pour les Expressions Régulières :*

Le code Flex va pendre a les lettres de l'alphabet A = {a, b}, ou l'epsilon mentionné par 1, ou le vide mentionné par 0. Et à chaque instant qu'il trouve une il l'envoie, d'autre par il va envoyer aussi les symboles nécessaires : '|' qui dénote la réunion, '*' pour l'étoile, '(' ')' pour les parenthèses, et '^' pour la concaténation.

  - *La Concaténation :*

    La Concaténation est un opérateur invisible pour l'utilisateur, mais je l'ai manipulé qu'elle soit renvoyé sous le symbole '^' à chaque fois qu'une concaténation est mise.

L'exemple suivant donne une vue plus précise :

```
[ab10]                  {  yylval.car = yytext[0];
                           return Terminal;  }
a/[ab10"("]             {  unput('^'); unput('a'); }
b/[ab10"("]             {  unput('^'); unput('b'); }
1/[ab10"("]             {  unput('^'); unput('1'); }
0/[ab10"("]             {  unput('^'); unput('0'); }
")"/[ab10"("]           {  unput('^'); unput(')'); }
"*"/[ab10"("]           {  unput('^'); unput('*'); }
"*"                      |
"^"                      |
"|"                      |
"("                      |
")"                      |
"\n"                        return *yytext;
```

III. Description du code Bison

L'analyseur syntaxique est manipulé suivant la Grammaire Hors-Contexte « G » : G = (V, T, P, S) où :

- **V** = {*input, line, ER*}
- **T** = A ∪ E , tel que A c'est l'alphabet *A = {a, b}* et E c'est l'ensembles des Tokens utiliser *E = { Terminal, Q, Is, In, Word, Help, Show, Empty, Single, Finite, Getout, Symbol, Epsilon, Infinite, Nullable}*
- **P** ensemble des règles de production :

input → input line | *input → input Getout '\n'* | *input → Help '\n'*
input → **ε** | *ER → Terminal* | *ER → ER '*'*
ER → ER '|' ER | *ER → ER '^' ER* | *ER → '(' ER ')'*
line → '\n' | *line → Show ER '\n'* | *line → ER '\n'*
line → Is ER Nullable Q '\n' | *line → Is ER Empty Q '\n'*
line → Is ER Epsilon Q '\n' | *line → Is ER Single Q '\n'*
line → Is ER Finite Q '\n' | *line → Is ER Infinite Q '\n'*
line → Is Word In ER Q '\n' | *line → Is ER Symbol Q '\n'*

- **S** c'est l'axiome : *S = input*

Notre axiome est bien désigné par suite :

```
%start input
```

L'analyseur syntaxique produira un AST dont les nœuds sont définis par la structure présenté dans le fichier « ast.h »

J'ai bien déclaré les Token nécessaires qui vont être renvoyé par l'analyseur lexicale dont leur type incluse (la plus part c'est des chaines de caractère).

Noté bien que l'ensemble de typage nécessaire et utilisé est dénoté par l'Union :

```
%union {  struct ast* astval;
          char* chaine;
          char car;
       }
```

Les "ER" Expressions Régulières sont des pointeurs vert les structures **ast** et bien définie dans le fichier « ast.h »

```
%type <astval> ER
```

On note aussi que la concaténation '^' (opérateur invisible pour l'utilisateur), la réunion '|' et l''etoile '∗' sont associatives à gauche tel que l''etoile '∗' est plus prioritaire que le produit '^', lui-même plus prioritaire que la réunion '|'.

```
%nonassoc '(' ')'
%left '|'
%left '^'
%left '*'
```

IV. Définitions inclues dans le fichier « ast.h »

On a bien sûr inclut toute bibliothèque qui soit nécessaire au courant des fichiers « sca.flex » « par.y » « ast.c ». Et le fichier lui-même est inclut dans les fichiers précédemment annoncé.

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

Le fichier contient la structure de donnée qui va bien être utilisé :

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

Noté bien la création du tableau de chaines de caractère initialiser à 100, la variable statique qui sert à déterminer a qu'elle point le tableau 'tab[]' est rempli.

```c
char* tab[100];
static int M=0;
```

Il contient aussi la déclaration des fonctions nécessaires pour la création et la manipulation des arbres abstraits.

```c
struct ast* creer_a();                                  /* création de l'AST (a) */
struct ast* creer_b();                                  /* création de l'AST (b) */
struct ast* creer_epsilon();                            /* création de l'AST (ε) */
struct ast* creer_vide();                               /* création de l'AST (Ø) */
struct ast* creer_union(struct ast *, struct ast *);    /* création de l'AST Concaténation */
struct ast* creer_concat(struct ast *, struct ast *);   /* création de l'AST Union */
struct ast* creer_etoile(struct ast *);                 /* création de l'AST étoile */
struct ast* copier_ast(struct ast *);                   /* création d'une copie d'AST */
void remplir_elements_ast(struct ast *);                /* Remplir les éléments du tableau tab[] */
void imprimer_ast(struct ast *);                        /* Imprime les éléments dans le fichier ast.xml */
void liberer_ast(struct ast *);                         /*efface l'AST donner en argument & libère la mémoire*/
int est_x(struct ast* , int );                          /* test si l'AST est un élément (a) ou (b) */
void aide(void);                                        /* Affiche l'aide sur l'interpréteur */
void start(void);                                       /* Affiche un petit guide au début */
struct ast* residuel(char , struct ast *);              /* calcule l'AST du résiduel de e */
int appartient(char* , struct ast *);                   /* teste si un mot w appartient à  L(e) */
int contient_epsilon(struct ast *);                     /* teste si epsilon appartient à  L(e) */
int est_epsilon(struct ast*);                           /* teste si L(e) = {epsilon} */
int est_symbole(struct ast*);                           /* teste si L(e) = {a} ou L(e) = {b} */
int est_single(struct ast*);                            /* teste si L(e) contient un seul mot */
int est_vide(struct ast*);                              /* teste si L(e) est vide */
int est_fini(struct ast*);                              /* teste si L(e) est fini */
```

V. Explication des routines utilisées dans le programme

On a bien noté que le fichier « ast.c » contient la définition de ces routines (le code C). On va essayer dans ce qui suit donner une explication de l'algorithme utilisé pour chaque fonction.

1 - void main(void) /* *La fonction principale* */

La fonction main ne fait de grand-choses, elle donne premièrement un petit affichage en appelant la fonction *start()* , et juste en suite elle donne démarrage a l'analyseur syntaxique en appelant sa fonction *yyparse()* pour que le programme fait marche.

2 - void start(void) ;

Cette fonction donne un affichage citant une petite manière de l'utilisation de l'interpréteur (c'est seulement un gros *printf()*).

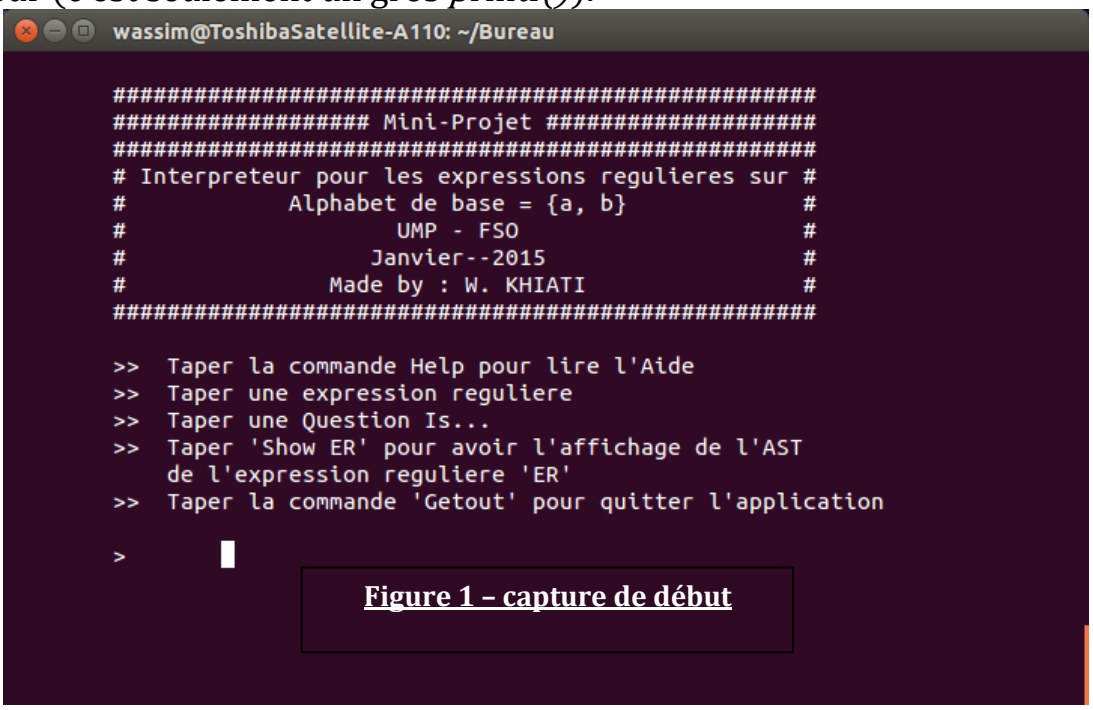

Figure 1 – capture de début

3 - void aide(void) ;

La fonction aide donne un guide d'utilisation de l'interpréteur. On peut bien lire ce guide en Tapant la commande **Help**

4 - struct ast* creer_a();

La fonction déclare un pointeur vert structure ast nommé '**a**' puis on alloue un espace mémoire et on la crée, ensuite on l'affecte le type « A » au champ type et on le ferme aux fils gauche & droite.

```
(*a).type = A;     (*a).fg = NULL;    (*a).fd = NULL;
```

On retourne finalement le pointeur **a**.

5 - struct ast* creer_b();

La fonction déclare un pointeur vert structure ast nommé '**b**' puis on alloue un espace mémoire et on la crée, ensuite on l'affecte le type « B » au champ type et on le ferme aux fils gauche & droite.

```
(*b).type = B;    (*b).fg = NULL;    (*b).fd = NULL;
```

On retourne finalement le pointeur **b**.

6 - struct ast* creer_epsilon();

La fonction déclare un pointeur vert structure ast nommé '**e**' puis on alloue un espace mémoire et on la crée, ensuite on l'affecte le type « EPSILON » au champ type et on le ferme aux fils gauche & droite.

```
e->type = EPSILON;      e->fg = NULL;      e->fd = NULL;
```

On retourne finalement le pointeur **e**.

7 - struct ast* creer_vide();

La fonction déclare un pointeur vert structure ast nommé '**v**' puis on alloue un espace mémoire et on la crée, ensuite on l'affecte le type « VIDE » au champ type et on le ferme aux fils gauche & droite.

```
v->type = VIDE;    v->fg = NULL;      v->fd = NULL;
```

On retourne finalement le pointeur **v**.

8 - struct ast* creer_union(struct ast *e1, struct ast *e2);

On déclare un pointeur vert structure ast nommé '**u**' puis on alloue un espace mémoire et on la crée, ensuite on l'affecte le type « UNION » au champ type et on affecte au fils gauche le paramètre qui est à gauche 'e1' et au fils droite le paramètre qui est à droite 'e2'.

```
u->type = UNION;  u->fg = e1; u->fd = e2;
```

On retourne finalement le pointeur **u**.

9 - struct ast* creer_concat(struct ast *e1, struct ast *e2);

On déclare un pointeur vert structure ast nommé '**c**' puis on alloue un espace mémoire et on la crée, ensuite on l'affecte le type « CONCAT » au champ type et on affecte au fils gauche le paramètre qui est à gauche 'e1' et au fils droite le paramètre qui est à droite 'e2'.

On retourne finalement le pointeur **c**.

```
c->type = CONCAT; c->fg = e1; c->fd = e2;
```

10 - struct ast* creer_etoile(struct ast *e);

On déclare un pointeur vert structure ast nommé '**s**' puis on alloue un espace mémoire et on la crée, ensuite on l'affecte le type « ETOILE » au champ type et on affecte au fils gauche le paramètre 'e' et on le ferme au fils droite.

```
(*s).type = ETOILE;     (*s).fg = e;      (*s).fd = NULL;
```

On retourne finalement le pointeur **s**.

11 - struct ast* copier_ast(struct ast *e);

On déclare un pointeur vert structure ast nommé '**cp**' puis on alloue un espace mémoire et on la crée, ensuite on l'affecte le même type du paramètre 'e' au champ type.

Pour le champ 'fg' (récursivement 'fd') on test si le paramètre 'e' est fermé au fils gauche (rec. Fils droite).

- si oui on ferme 'cp' au fils gauche (rec. Fils droite)
- sinon on lui affecte l'ast retourné par la fonction *copier_ast(e->fg)* (rec. *copier_ast(e->fd)*) d'une façon récursive dont le but d'affecter au fils gauche (rec. Fils droite) de 'cp' ce qui est dans son parallèle au paramètre 'e'.

Au dernier moment, on retourne le pointeur **cp**.

12 - void remplir_elements_ast(struct ast *e) ;

La fonction remplit les éléments de l'AST dans le tableau tab[], elle se base sur une boucle *switch* suivant le type du paramètre rentré. On va aussi utiliser la variable statique M qui est initialisé par 0.

- Si le type est un ***A***, ***B***, ***EPSILON*** où ***VIDE***: on va premièrement allouer une place mémoire pour la (M+1)éme case du tableau « **tab[]** » par une chaine de caractère de longueur 'n' {tq n réfère à la longueur de la chaine de caractère suivant le typage}, puis on remplit **tab[M]** par la chaine convenable {"*<a/>*" pour A, "*<b/>*" pour B, "*<epsilon/>*" pour EPSILON & "*<vide/>*" pour VIDE] et on incrémente le **M**, ensuite en sort de la boucle par un *break ;*
- Si le type est un ***ETOILE***: on va remplir le tableau suivant la même méthode de début par la chaine qui déduit l'ouverture de l'étoile « *<etoile>* » mais en va pas sortir de la boucle, on va suivant remplir le tableau suivant se qui est dedans le fils gauche par un appelle récursive de la fonction elle-même par le fils gauche comme paramètre [*remplir_elements_ast(e->fg);*]. après on va remplir le tableau suivant la même méthode de début par la chaine qui déduit la fermeture de l'étoile « *</etoile>* », ensuite en sort de la boucle par un *break ;*
- Si le type est un ***UNION*** où ***CONCAT***: on va remplir le tableau suivant la même méthode de début par la chaine qui déduit l'ouverture de l'union (res. Concaté.) « *<union>* » (res. « *<concat>* ») mais en va pas sortir de la boucle, on va suivant remplir le tableau suivant se qui est dedans le fils gauche par un appelle récursive de la fonction elle-même par le fils gauche comme paramètre [*remplir_elements_ast(e->fg);*] puis on va faire la même chose pour le fils droite [*remplir_elements_ast(e->fd);*]. après on va remplir le tableau suivant la même méthode de début par la chaine qui déduit la fermeture de l'union (res. Concaté.) « *</union>* » (res. « *</concat>* »), ensuite en sort de la boucle par un *break ;*

13 - void imprimer_ast(struct ast *e) ;

Cette fonction sert à remplir le fichier XML nommé ast.xml représentant l'AST de '**e**' et d'affiche également le contenu de ce fichier dans la sortie standard (l'écran). Et pour cela on utilise la commande **Show ER** (tq ER réfère à l'expression régulière qu'on désigne).

On va ouvrir le flot et on lui associe le fichier « ast.xml » sous le mode « w » (write), ensuite on ajoute la balise d'ouverture de l'élément racine « *< ast >* », puis on copie à l'aide d'une boucle *for* le contenu du tableau *tab[]* précédemment remplit depuis le 1er élément jusqu'à le Méme, et après on ajoute la balise de fermeture de l'élément racine « *</ast>* » et on ferme le flot. On fait la même chose sûr la sortie standard (l'écran).

Comme on a finis de l'utilisation du tableau, on libère tout ses case mémoire à l'aide de la fonction *free()* dans une boucle *for* depuis le 1er élément jusqu'à le Méme, et Finalement on rend la variable M a sa valeur initiale 0. *M = 0 ;*

Exemple Expliquant :

```
wassim@ToshibaSatellite-A110: ~/Bure
        >
        >       a|(b|a*)*
        Valide, taille = 7
        >       Show a|(b|a*)*
<ast>
<union>
<a/>
<etoile>
<union>
<b/>
<etoile>
<a/>
</etoile>
</union>
</etoile>
</union>
</ast>
        >
```

Figure 3 – Capture de Show

```
-<ast>
 -<union>
    <a/>
   -<etoile>
    -<union>
       <b/>
      -<etoile>
         <a/>
       </etoile>
     </union>
   </etoile>
 </union>
</ast>
```

Figure 2 – Le fichier ast.xml

14 - int contient_epsilon(struct ast *e) ;

L'utilisateur peut demander si le langage d'une expression régulière contient le mot ε à travers une question de la forme : **Is ER Nullable ?**
(Où 'ER' réfère à l'expression régulière qu'on désigne).

En tapant cette commande il sera appelle a la fonction *contient_epsilon()*. La fonction se base dans son travail sur un *switch* de cas selon le type de la variable structuré dont '**e**' pointe sûr. Pour ce cas on va avoir, si le type était :

```
wassim@ToshibaSatellite-A110: ~/Bureau
        >       Is 0 Nullable ?
        => No
        >       Is a|b* Nullable ?
        => Yes
        >       Is 1 Nullable ?
        => Yes
        >       Is (b|a) Nullable ?
        => No
        >
```

Figure 4 - Capture de Nullable

- **EPSILON** : Donc le langage contient le mot ε. *=> return 1 ; /*vrai*/*
- **CONCAT** : À l'aide d'un appelle récursive de la fonction elle-même, on doit tester que les deux fils gauche & droite contiennent le mot ε.
  => return contient_epsilon(e->fg) && contient_epsilon(e->fd);
- **UNION** : Dans ce cas on si l'un des deux fils contient le mot ε c'est bien.
  => return contient_epsilon(e->fg) || contient_epsilon(e->fd);
- **ETOILE** : On sait que quelle que soit le fils gauche, l'étoile d'un langage L sur A, notée L^* admet toujours ε dans l'ensemble de ces mots car ($L^0 = \varepsilon$).
  => return 1 ; / vrai */*
- **A**, **B**, ou **VIDE** : Alors le langage ne contient pas le mot ε. *=> return 0 ; /* faux */*

15 - int est_vide(struct ast* e) ;

L'utilisateur peut demander si le langage d'une expression régulière est vide L(e) = Ø à travers une question de la forme : **Is ER Empty ?**
(Où 'ER' réfère à l'expression régulière qu'on désigne).

En tapant cette commande il sera appelle a la fonction *est_vide()*.

La fonction se base dans son travail sur un *switch* de cas selon le type de la variable structuré dont '**e**' pointe sûr. Pour ce cas on va avoir, si le type était :

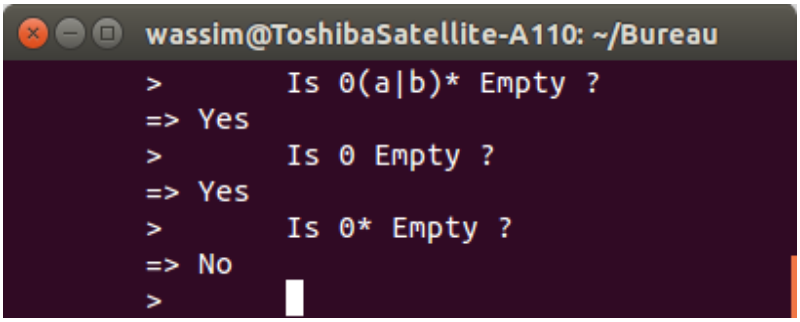

Figure 5 - Capture de Empty

- **VIDE** : Donc le langage est vide L(e) = Ø. *=> return 1 ; /*vrai*/*
- **CONCAT** : À l'aide d'un appelle récursive de la fonction elle-même, on doit tester que l'un des deux fils soit vide (car on sait que ØE = EØ = Ø). *=> return est_vide(e->fg) || est_vide(e->fd);*
- **UNION** : On sait que Ø est l'élément neutre pour la réunion. En conséquent pour que l'Union soit vide, ces deux fils gauche & droite doivent être vide. *=> return est_vide(e->fg) && est_vide(e->fd);*
- **(Les autres types)** : Pour tout autre type le langage ne peut pas être vide L(e) ≠ Ø. *=> return 0 ; /* faux */*

16 - int est_epsilon (struct ast* e) ;

L'utilisateur peut demander si le langage d'une expression régulière égal ou réduit au mot vide L(e) = ε, en utilisant la question de la forme : **Is ER Epsilon ?**

(Où 'ER' réfère à l'expression régulière qu'on désigne).

En tapant cette commande il sera appelle a la fonction *est_epsilon()*.

La fonction se base dans son travail sur un *switch* de cas selon le type de la variable structuré dont '**e**' pointe sûr. Si le type était :

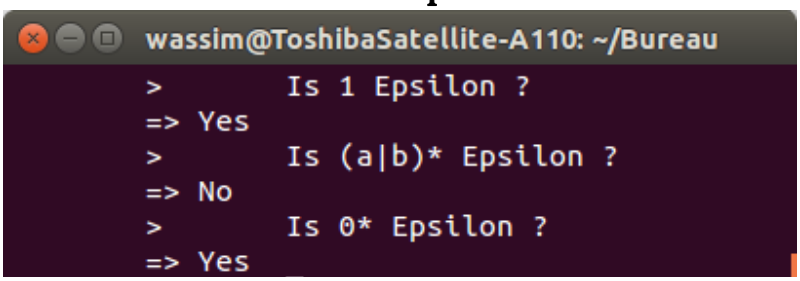

Figure 6 - Capture d'Epsilon

- **EPSILON** : C'est bien vue que c'est le vide L(e) = ε. *=> return 1 ; /*vrai*/*
- **CONCAT** : À l'aide d'un appelle récursive de la fonction elle-même, on doit tester que les deux fils gauche & droite soient égal ou réduit au mot vide L(e) = ε. *=> return est_epsilon(e->fg) && est_epsilon(e->fd);*
- **UNION** : Pour que la réunion soit égal ou réduit au mot vide, il ya deux cas.
  - ◊ Le 1er cas consiste à avoir le même analyse expliquer pour la Concaténation. *=> return est_epsilon(e->fg) && est_epsilon(e->fd);*
  - ◊ Le 2émé cas se base sûr le test que l'un des deux fils soient vide (à l'aide de la fonction *est_vide*()) et l'autre doit être égal ou réduit au mot vide. *=> return est_epsilon(e->fg) || est_epsilon(e->fd);*
- **ETOILE** : On sait que l'étoile d'un langage L peut être égal ou réduit au mot vide L(e) = ε, si et seulement si le fils gauche soit vide L(e→fg) = Ø, ou qu'il soit égal ou réduit au mot vide L(e→fg) = ε. Sinon c'est faux.
- **A, B, ou VIDE** : Alors le langage ne peut pas être égal ou réduit au mot vide. L(e) ≠ ε. *=> return 0 ; /* faux */*

17 - int est_fini(struct ast* e) ;

L'utilisateur peut demander si le langage d'une expression régulière est fini ou est infini, en utilisant les deux questions de la forme :

Is ER Finite ?

```
>
>       a|bab|0*
Valide, taille = 9
>       Is a|bab|0* Finite ?
=> Yes
>       Is (a|bab|0)* Finite ?
=> No
>       Is (aab0)* Finite ?
=> Yes
```

Figure 7 - Capture de Finite

Is ER Infinite ?

```
>
>       b(a|bab)aa
Valide, taille = 13
>       Is b(a|bab)aa Infinite ?
=> No
>       Is 1|ab* Infinite ?
=> Yes
>       Is (aab0)* Infinite ?
=> No
>
```

Figure 8 - Capture de Infinite

En tapant cette commande il sera appelle a la fonction *est_epsilon()*.

La fonction se base dans son travail sur un *switch* de cas selon le type de la variable structuré dont '**e**' pointe sûr. Si le type était :

- **UNION** ou **CONCAT** : À l'aide d'un appelle récursive de la fonction elle-même, Les deux fils Gauche et Droite doivent être finies.
  => return est_fini(e->fg) && est_fini(e->fd);
- **ETOILE** : Dans ce cas le langage peut être finis si et seulement si le fils gauche soit vide L(e) = Ø ou égal ou réduit au mot vide L(e) = ε.
  => return est_epsilon(e->fg) || est_vide(e->fg);
- **(**Les autres types**)** : Pour tout autre type c'est bien convaincant que le langage est fini (c'est des symboles fixe). => return 1 ; /* Vrai */

18 - void liberer_ast(struct ast *e);

Cette fonction permet de libérer l'espace mémoire occupée par un AST, et on l'appel a chaque fois qu'on a fini d'utilisé une Expression Régulière.

Elle se base dans son travail sur un *switch* de cas selon le type de la variable structuré dont '**e**' pointe sûr. Mais le fonctionnement principale se base sur :

- Si l'ast n'est pas fermé au fils gauche on lance un appel récursif de la fonction elle-même dont le paramètre soit le fils gauche [*liberer_ast(e→fg)*] pour libérer se qui est dedans .
- Si l'ast n'est pas fermé au fils droite on lance un appel récursif de la fonction elle-même dont le paramètre soit le fils droite [*liberer_ast(e→fd)*] pour libérer se qui est dedans .
- Si l'ast est fermé aux deux fils Gauche & Droite donc c'est un symbole fixe, alors on le libère. [*liberer_ast(e)*].

19 - int est_x(struct ast *x, int T) ;

L'utilisation de cette routine sert à s'avoir si l'élément '**x**' pointe sur un AST **A** ou **B** ou peut être réduit en un seul mot **a** ou **b**, en faisant la distinction à travers l'entier '**T**' qui réfère sur l'AST **A** si [T = 3], ou sûr l'AST **A** si [T = 4].

La fonction se base dans son travail sur un petit test par i*f()* suivi d'un *switch* de cas selon le type de la variable structuré dont '**x**' pointe sûr.

- Si l'AST est **A** avec [T=3] : elle retourne vrai *=> return 1 ; /*vrai*/*
- Si l'AST est **B** avec [T=4] : elle retourne vrai *=> return 1 ; /*vrai*/*

- Sinon on rentre sûr les différentes cas du type de la variable structuré dont '**x**' pointe sûr.
  - **UNION** : on doit faire une suite de test :
    - Si les deux fils gauche et droite ont la même chose (se qu'on cherche le A ou le B), c'est Bien. *=> return 1 ; /*vrai*/*
    - Si l'une des deux soit vide et l'autre contient se qu'on cherche (le A ou le B), c'est Bien. *=> return 1 ; /*vrai*/*
    - Sinon on retourne faux. *=> return 0 ; /*faux*/*
  - **CONCAT** : Ce cas peut être vrai si l'une des deux soit le mot vide et l'autre contient se qu'on cherche (le A ou le B).
  - **(Les autres types)** : Si ça passe a ce cas c'est faux. *=> return 0 ; /*faux*/*

20 - int est_symbole(struct ast *e) ;

L'utilisateur peut demander si le langage d'une expression régulière égal ou réduit à un symbole L(e) = {a, b}, en utilisant la question de la forme : **Is ER Symbol ?**

En tapant cette commande il sera appelle a la fonction *est_symbole()*.

La fonction se base sur l'appelle de fonction *est_x()* expliqué au dessus.

Elle retourne vrai si :

- C'est un AST **A** ou réduit en **A**, sans qu'il soit ou peut être réduit en un AST **B**.
- C'est un AST **B** ou réduit en **B**, sans qu'il soit ou peut être réduit en un AST **A**.

```
>       a|a
Valide, taille = 3
>       Is a|a Symbol ?
=> No
>       Is 0|b Symbol ?
=> Yes
>       Is 1 Symbol ?
=> No
>       Is a Symbol ?
=> Yes
>       Is 0 Symbol ?
=> No
>
```

Figure 9 - Capture de Symbole

21 - int est_single(struct ast *e);

L'utilisateur peut demander si le langage d'une expression régulière égal ou réduit à un seul mot, en utilisant la question de la forme : **Is ER Single ?**
(Où 'ER' réfère à l'expression régulière qu'on désigne).

En tapant cette commande il sera appelle a la fonction *est_single()*.

Dans son travail la fonction se base sur un *switch* de cas selon le type de la variable structuré dont '**e**' pointe sûr. Si le type était :

```
>       Is baa Single ?
=> Yes
>       Is ba|a* Single ?
=> No
>       Is 0|a* Single ?
=> No
>       Is 0|a Single ?
=> Yes
>
```

Figure 10 - Capture de Single

- **CONCAT** : À travers un appelle récursive de la fonction elle-même, Les deux fils Gauche et Droite doivent être aussi single pour qu'elle retourne vrai
- **UNION** : Pour qu'elle soit vrai dans ce cas l'un des fils doit être vide est l'autre single.
- **ETOILE** : Ce cas est vrai si et seulement si le fils gauche admet le langage vide ou qu'il soit le mot vide {ε} ou réduit en un.
- **VIDE** : C'est pas un single, c'est faux. *=> return 0 ; /*faux*/*
- **(Les autres types)** C'est bien évident que les autres types vont retourner vrai. *=> return 1 ; /*vrai*/*

22 - struct ast* residuel(char s, struct ast *e);

On calcul à travers cette fonction le résiduel du langage L de l'expression régulière dénoté par l'AST '**e**' par rapport à la lettre '**s**', noté L/a est le langage formé par les mots de L ayant a pour préfixe, auxquels on a supprimé ce préfixe.

Formellement : $L/a = \{w \in A* | aw \in L\}$

Dans son travail la fonction se base sur un *switch* de cas selon le type de la variable structuré dont '**e**' pointe sûr et Grâce à des règles. Si le type était :

- ⌘ **VIDE** : On retourne le vide car $\emptyset/x = \emptyset$.
- ⌘ **CONCAT** : sur les différents cas on va avoir :
  - Si l'un des deux fils est égal ou réduit au mot vide on retourne le résiduel de l'autre.
  - Si l'un des deux fils est vide, on retourne le vide.
  - Sinon on suivra l'une des deux règles suivantes :
    - $(rs)/x = (r/x)s$, si $\varepsilon \notin L(r)$.
    - $(rs)/x = (r/x)s \mid s/x$, si $\varepsilon \in L(r)$.
- ⌘ **UNION** : sur les différents cas on va avoir :
  - Si l'un des deux fils est vide on retourne le résiduel de l'autre.
  - Sinon on suivra la règle suivante : $(r \mid s)/x = (r/x) \mid (s/x)$.
- ⌘ **ETOILE** : les différents cas on va avoir :
  - Si le caractère rentré est l'epsilon, on retournera '**e**' elle-même.
  - Sinon on suivra la règle suivante : $r*/x = (r/x)r*$.
- ⌘ **(Les autres types)** : On retourne l'epsilon si c'est les même $x/y = \varepsilon$, si $x = y$, et le vide sinon $x/y = \emptyset$, si $x \neq y$.

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

23 - int appartient(char *w, struct ast *e) ;

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

Is <Word> In ER ?

(Où 'ER' réfère à l'expression régulière qu'on désigne, et 'Word' au mot).

En tapant cette commande il sera appelle a la fonction *appartient()*.

La fonction se base une boucle *while* qui parcourt le mot caractère par caractère par un appel de la fonction *residuel()* définit au dessus.

```
>       a(a|b)*a
Valide, taille = 8
>       Is <aba> In a(a|b)*a ?
=> Yes
>       Is <1> In (b|0)* ?
=> Yes
>       Is<b>Inb(a|b)*b?
=> No
>
```

Figure 11 - Capture d'appartient

- Si le résiduel retourné était vide on sort de la fonction par une raiponce négatif. *=> return 0 ; /*faux*/*
- Qu'on finit de parcourir la chaine et par ensuite on sort de la boucle, on doit tester que le dernier résiduel retourné admet le mot vide {ε} dans son langage pour qu'il soit vrai. *=> return 1 ; /*vrai*/*
- Sinon c'faux le mot n'appartient pas au langage de expression régulière. *=> return 0 ; /*faux*/*