Travaux Dirigés et Pratiques d'interprétation et compilation GLIN604

Michel Meynard

23 janvier 2012

1 Analyse lexicale

Exercice 1 – Dessiner un AFD distinct pour chacun des langages suivants :

1. Langage de mots-clés : $L_{key} = \{if, then, else, throw\}$.
2. Langage des littéraux numériques entiers du C (ou C++, ou Java), décimaux L_{c10}, octaux L_{c8}, hexadécimaux L_{c16}.
3. Langage L_{id} des identificateurs composés d'une lettre au moins, éventuellement suivie de chiffres, de lettres et de "_".
4. Langage des littéraux numériques flottants décimaux L_f ; la suite de chiffres à gauche ou à droite du point décimal pouvant être vide ; l'exposant entier n'est pas obligatoire. Exemples : 13., 1.7e23, .89E-34
5. Langage L_{sep} des séparateurs composés de blancs (Espace, `\t, \n`), des commentaires à la C et à la C++.

– Dessiner un unique AFD à jeton reconnaissant une partie de ces langages. Vous reconnaitrez notamment : le mot-clé `if`, les identificateurs, les entiers décimaux, les flottants sans exposant, les séparateurs. Utiliser des jetons négatifs pour les lexèmes à filtrer (séparateurs).
– Ecrire dans le fichier `afd.h` la fonction `creerAfd()` correspondant à cet Afd à jeton filtrant.
– Implémenter l'analyseur lexical en C. Le `main()` appellera itérativement analex() et affichera une chaîne correspondant à l'entier retourné ainsi que le lexème, ceci jusqu'à la fin du fichier.

Exercice 2 Ecrire les expressions régulières correspondant aux langages réguliers suivants : $L_{key}, L_{c10}, L_{c8}, L_{c16}, L_{id}, L_f, L_{sep}$.

Exercice 3 Ecrire un analyseur lexical reconnaissant l'ensemble des expressions régulières des exercices précédents à l'aide de flex. L'action associée à chaque lexème reconnu consiste à retourner le jeton correspondant au lexème. Toute autre expression d'un caractère retournera un jeton correspondant au code ISO Latin-1 de ce caractère.

Exercice 4 Ecrire un source lex `delblancs.l+` filtrant un fichier en :
– supprimant les lignes blanches (lignes vides ou remplies de blancs (espace et tabul.)),
– suprimant les débuts et fins de ligne blancs,
– remplaçant tous les blancs `\t<espace>` multiples par un seul espace,
– remplaçant les tabulations `\t` par un espace.

Exercice 5 Ecrire en flex, un programme comptant le nombre de lignes, le nombre de mots et le nombre de caractères d'un fichier passé en argument (`man wc`). Un mot est une suite de caractères séparés par des blancs (tab, espace, retour ligne). Vérifiez que vos résultats sont les mêmes que ceux de `wc`.

Exercice 6 Soit l'expression régulière e suivante : $e = a((b|c)^{**}|cd)^*b$

1. Dessiner un automate fini équivalent à e.
2. Cet automate est-il déterministe ? Si oui indiquez pour quelles raisons, sinon déterminisez-le.
3. Minimisez cet AFD.

Exercice 7 Soit l'automate fini $B = (\{a, b, c\}, \{1, 2, 3, 4\}, \{1\}, \{2, 4\}, \{1a2, 1a3, 2b2, 2c2, 3b4, 4c3, 4b4\})$.

1. Dessiner un automate fini déterministe équivalent à B.
2. Minimisez cet AFD.

2 Analyse descendante récursive

Exercice 8 Soit la grammaire non récursive à gauche vue en cours $G_{ENR} = (\{0, 1, \ldots, 9, +, *, (,)\}, \{E, R, T, S, F\}, X, E)$ avec les règles de X suivantes :

$$\begin{aligned} E &\rightarrow TR \\ R &\rightarrow +TR|\varepsilon \\ T &\rightarrow FS \\ S &\rightarrow *FS|\varepsilon \\ F &\rightarrow (E)|0|1|\ldots|9 \end{aligned}$$

Soit le programme C vérifiant un mot du langage $L(G_{ENR})$:

```
/*============================================================================
Nom : analdesc.c        Auteur : Michel Meynard
Maj : 25/3/02           Creation : 8/1/98
Projet : Analyse descendante récursive d'expression arithmétique
----------------------------------------------------------------------------
Specification:
Ce fichier contient un reconnaisseur d'expressions arithmétiques composée de
littéraux entiers sur un car, des opérateurs +, * et du parenthésage ().
Remarque : soit rediriger en entrée un fichier, soit terminer par deux
caractères EOF (Ctrl-D), un pour lancer la lecture, l'autre comme "vrai" EOF.
============================================================================*/
#include <stdio.h>
                              /* les macros sont des blocs : pas de ';' apres */
#define AVANCER {jeton=getchar();numcar++;}
#define TEST_AVANCE(prevu) {if (jeton==(prevu)) AVANCER else ERREUR_SYNTAXE}
#define ERREUR_SYNTAXE {printf("\nMot non reconnu : erreur de syntaxe \
au caractère numéro %d, de jeton %d \n",numcar,jeton); exit(1);}

void E();void R();void T();void S();void F(); /* déclarations des fonctions */

int jeton;                         /* caractère courant du flot d'entrée */
int numcar=0;                      /* numero du caractère courant (jeton) */

void E(){
  T();                            /* regle : E->TR */
  R();
}
void R(){
  if (jeton=='+') {               /* regle : R->+TR */
    AVANCER
    T();
    R();
  }
  else ;                          /* regle : R->epsilon */
}
void T(){
  F();
  S();                            /* regle : T->FS */
}
void S(){
  if (jeton=='*') {               /* regle : S->*FS */
    AVANCER
    F();
    S();
  }
  else ;                          /* regle : S->epsilon */
}
void F(){
  if (jeton=='(') {               /* regle : F->(E) */
    AVANCER
    E();
    TEST_AVANCE(')')
  }
  else
    if (isdigit(jeton))           /* regle : F->0|1|...|9 */
      AVANCER
    else ERREUR_SYNTAXE
}
```

```
main(){                          /* Fonction principale */
  AVANCER                        /* initialiser jeton sur le premier car */
  E();                           /* axiome */
  if (jeton==EOF)                /* expression reconnue et rien après */
    printf("\nMot reconnu\n");
  else ERREUR_SYNTAXE            /* expression reconnue mais il reste des car */
}
```

1. Sur le modèle du vérificateur, écrire un programme évaluant la valeur d'une expression arithmétique. Par exemple, **evaldesc** sur la chaîne 1+2+(2+1)*3 retournera 12. Attention, on précisera pour chaque opérateur le type d'associativité, gauche ou droite, employée dans le programme.
2. Sur le modèle du vérificateur, écrire un programme traduisant une expression arithmétique en sa forme postfixée (polonaise inversée). Par exemple, **postdesc** sur la chaîne 1+2+(2+1)*3 retournera 12+21+3*+. On utilisera l'**associativité à gauche** pour l'addition et la multiplication.
3. Sur le modèle du vérificateur, et en utilisant le type abstrait Arbin implémenté en C, écrire un analyseur syntaxique produisant et affichant l'arbre abstrait correspondant à une expression. On utilisera l'associativité à gauche pour l'addition et la multiplication. Par exemple, **arbindesc** sur la chaîne 1+2+(2+1)*3 affichera :

```
+
 +
  1
  2
 *
  +
   2
   1
  3
```

Voici le fichier d'en-tête arbin.h :

```
/*============================================================================
Nom: arbin.h           auteur: Meynard Michel
Maj:  5/4/1995         Creation: 5/4/95
Projet:
----------------------------------------------------------------------------
Specification:
Ce fichier contient les declarations de structures, de types, de variables de
fonctions sur les arbres binaires d'entiers.
----------------------------------------------------------------------------
utilise par :
============================================================================*/

#ifndef _ARBINH
#define _ARBINH

#ifndef NULL
#define NULL 0
#endif /* NULL */

/*----------------------- Types Unions Structures -------------------------*/

typedef struct Noeudbin * Arbin;
/* pointeur sur la racine de l'arbre binaire d'entiers */

typedef struct Noeudbin {
    int val ;
    Arbin fg;
    Arbin fd ;
} Noeudbin ; /* noeud d'un arbre binaire d'entiers */

/*--------------------------FONCTIONS-----------------------------------------*/
#define creerarbin() (NULL) /* cree un arbin vide (retourne pointeur nul) */
```

```
#define sag(a) ((a)->fg) /* retourne le sous-arbre gauche de a */

#define sad(a) ((a)->fd) /* retourne le sous-arbre droit de a */

#define videa(a) (a==NULL) /* teste si arbin vide */

#define racine(a) ((a)->val) /* racine d'un arbin */

/* remarque : ces 5 pseudo-fonctions (macros) sont definies quel
que soit le type de l'arbin (entier, car, arbre,...) */


void greffergauche(Arbin pere, Arbin filsg);
/* remplace le sag(pere) par filsg et vide l'ancien sag */
/* operation MODIFIANTE :  ATTENTION aux effets de bord */

void grefferdroite(Arbin pere, Arbin filsd);
/* remplace le sad(pere) par filsd et vide l'ancien sad */
/* operation MODIFIANTE :  ATTENTION aux effets de bord */

Arbin construire(int rac, Arbin g, Arbin d); /* construit un nouvel arbin */

Arbin copiera(Arbin a); /* copie a */

void videra(Arbin * pa); /* vide *pa (NULL) et ses sous-arbres */
/* operation MODIFIANTE :  ATTENTION aux effets de bord */

void affichera(Arbin a,int indent); /* affiche a de manière indentée */

#endif /* _ARBINH */
```

Exercice 9 Ecrire un programme évaluant la valeur d'une expression arithmétique composée de nombres flottants non signés et des opérateurs `()`, `^`, `*`, `/`, `+`, `-`. Attention à respecter les associativités des opérateurs ! Vous utiliserez flex pour la partie analyse lexicale. Par exemple, **calcdesc** sur la chaîne `5.12-2.56-2^2^3/100` retournera 0.

Exercice 10 On désire écrire un traducteur d'expression régulière en automate d'état fini non déterministe par l'algorithme de Thompson. Une expression régulière de base est constituée d'une lettre minuscule comprise entre a et z (symboles terminaux), du symbole '@' pour représenter epsilon ε, du symbole '0' pour représenter le langage vide $\emptyset$. Une expression régulière (er) est de base ou est obtenue par concaténation (implicite) de deux er, par union de deux er (symbole |), par l'opération * appliquée à une er. Le parenthésage est permis pour modifier l'ordre de priorité des opérateurs qui est du plus petit au plus grand : |, concaténation, *, ().

1. Ecrire une grammaire "naturelle" engendrant des expressions régulières et prouver son ambiguïté.
2. Ecrire une grammaire récursive à gauche et non ambiguë des expressions régulières (S, E, T, F).
3. Ecrire une grammaire non récursive à gauche et non ambiguë des expressions régulières : dérécursivez la grammaire précédente (S, X, E, R, T, Y, F).
4. Ecrire un analyseur récursif descendant construisant l'arbre abstrait correspondant à une expression régulière.
5. Optimiser l'arbre abstrait construit en tenant compte des règles suivantes, e étant une er quelconque :
   - e**=e*,
   - @*=@=0*,
   - @e=e@=e, 0e=e0=0,
   - 0|e=e|0=e.
6. Fabriquer la table d'analyse de l'automate à pile en analyse descendante.
7. Dessiner les états successifs de la pile lors de la reconnaissance de l'expression $ab*|c$.
8. Ecrire une fonction NombreEtat(Arbin a) calculant le nombre d'état de l'automate fini non déterministe obtenu par l'algorithme de thompson.
9. Programmer l'algorithme de Thompson en implémentant l'automate non déterministe par un tableau dynamique d'entiers de NombreEtat(a) lignes et 3 colonnes. Chaque ligne correspond à un état. La première colonne de ce

tableau contient le symbole (@a-z) de la ou les transitions sortantes. Les deux autres colonnes contiennent le ou les états destinations de la ou des deux transitions possibles.

3 Analyse ascendante LR

Exercice 11 On reprend l'exercice 10 en analyse ascendante.

1. En utilisant la grammaire récursive à gauche non ambiguë, écrire un source bison qui produise et affiche l'arbre abstrait associé à une expression régulière (sans utiliser flex).
2. En utilisant la grammaire "naturelle" et les règles de précédences pour les opérateurs, écrire un source bison traduisant une expression régulière en un automate d'état fini par l'algorithme de Thompson. Attention, à la concaténation qui est implicite!

Exercice 12 En reprenant l'architecture de la calculette (calc.l et calc.y) vue en cours, écrire une calculette logique d'ordre 0. Une expression logique de base est constituée des constantes 0 ou false ou faux ou 1 ou true ou vrai. Une expression logique est de base ou est obtenue par conjonction (&), disjonction (|), implication (->), équivalence (==), ou exclusif (^) de deux expressions logiques, ou encore par négation (!) d'une expression logique. Le parenthésage est permis pour modifier l'ordre de priorité des opérateurs qui est du plus petit au plus grand : $\{|, ->, ==, \hat{}\}$, &, !, (). Les expressions logiques sont évaluées de gauche à droite.

1. Ecrire logic.l, logic.y pour évaluer une expression logique d'ordre 0.
2. Ajouter à cette calculette la fonctionalité suivante : 26 variables logiques, nommées a-z, permettent de conserver le résultat de calculs précédents. L'affectation de ces variables et leur utilisation ressemblera à la syntaxe C. Par exemple, $a = 0|1 -> 0$ puis $b = a -> (0|1)$. Les variables seront implantées dans un tableau global de 26 entiers, réalisant une table des symboles rudimentaire.

Exercice 13 Soit la grammaire $G_{ETF} = (\{0, 1, \ldots, 9, +, *, (,)\}, \{E, T, F\}, R, E)$ avec les règles de R suivantes :

$$\begin{aligned} E &\rightarrow E + T|T \\ T &\rightarrow T * F|F \\ F &\rightarrow (E)|0|1|\ldots|9 \end{aligned}$$

1. Calculer la collection canonique (AFD) des ensemble d'items LR(0) de la grammaire augmentée associée à G_{ETF}.
2. Construire les tables d'analyse Action et Successeur après avoir calculé les TabSuivants des non terminaux.
3. Analyser l'expression (5+3)*2$.

4 Analyse et compilation d'un langage Pasada

L'objectif à réaliser est un compilateur traduisant des programmes écrits en pasada (Pascal-Ada), dans le langage C. Le compilateur sera lui-même écrit en C++. Il est donc de la forme : $\text{pasada}_{\text{C++}}\text{C}$.

4.1 Description du langage pasada

4.1.1 Caractéristiques

- Pas de compilation séparée. Un programme pasada est entièrement localisé dans un fichier d'extension .pa.
- C'est un langage très fortement typé. Toutes les conversions de types doivent être explicites. Il existe 4 types : int, bool, string, float.
- Un programme pasada utilise un certain nombre de procédures. Chaque procédure déclare un certain nombre de variables locales, puis définit les instructions qui la réalisent. Chaque procédure possède un certain nombre de paramètres typés qui peuvent être soit d'entrée (in : lisibles mais non modifiables), soit de sortie (out : résultats), soit d'entrée/sortie (inout : lisibles et modifiables).
- Le point d'entrée du programme, déclare un certain nombre de variables globales, puis définit les instructions.
- Les entrées/sorties sont très basiques : lecture au clavier et écriture à l'écran d'une variable typée.
- Il n'y a pas de problèmes de références en avant : on peut utiliser une procédure avant qu'elle n'apparaisse dans le fichier.
- Il n'y a pas de surcharge de procédures : chaque procédure a un nom différent. Il n'y a pas de conflit de noms de variables : dans une proc, s'il y a conflit sur le nom local/global, c'est la variable locale qui est prioritaire.
- Les structures de contrôle sont : if then else, if then, while begin end, repeat until.
- Les opérateurs sont à la pascal : égal (=), différent (<>), et (and) évaluation complète, non (not), affectation (:=), ...

4.1.2 Exemple

```
/* les commentaires à la C et à la C++ sont possibles */
procedure fact(in int n, out int r) // factorielle : 2 paramètres
int j;                  // variable locale à fact()
begin                   // début du corps de procédure
  if n=0                // pas de parenthèse autour d'une condition
  then r:=1;            // affectation du résultat
  else
   begin                // bloc d'instructions
    fact(n-1,j);        // appel récursif à fact
    r:=n*j;             // affectation du résultat
   end                  // fin de bloc
  return;               // retour à l'appelant
end                     // fin de procédure

program                 // point d'entrée du programme
int i,j;                      // variable globale
begin                   // bloc d'instructions du programme
  writeString("Veuillez entrer un entier SVP : ");      // affichage
  readInt(i);           // proc prédéfinie : readInt(out int i)
  writeString("Voici le résultat de factorielle() : ");
  fact(i,j);            // calcul
  writeInt(j);          // proc prédéfinie : writeInt(in int i)
end
```

4.2 Développement du compilateur

Le compilateur pac (pasada compiler), sera développé en lex, yacc, C++.

Exercice 14 Ecrire la grammaire du langage pasada. N'oubliez pas les procédures prédéfinies telles que les conversions, la taille d'une chaîne, les entrées/sorties, ...

Exercice 15 Ecrire un analyseur lex/yacc qui réalise uniquement la vérification syntaxique sans génération de code.

Exercice 16 Ecrire un compilateur en utilisant les structures de données adéquates : arbres abstraits, table des symboles, ...

Exercice 17 Améliorer le langage pasada en ajoutant la notion de tableau statique. Un tableau est toujours déclaré avec sa taille : `int tab[10];`. Les tableaux sont indicés de 1 à n. Dans une procédure, un paramètre tableau est déclaré sans sa taille : `inout int tableau[];`. Dans ce cas, l'opérateur prédéfini `|tableau|` renvoie la taille (int) du tableau.

Exercice 18 Améliorer le langage pasada en ajoutant la notion de structure C.

5 Solutions des exercices

Solution 1 – unique AFD à jeton :

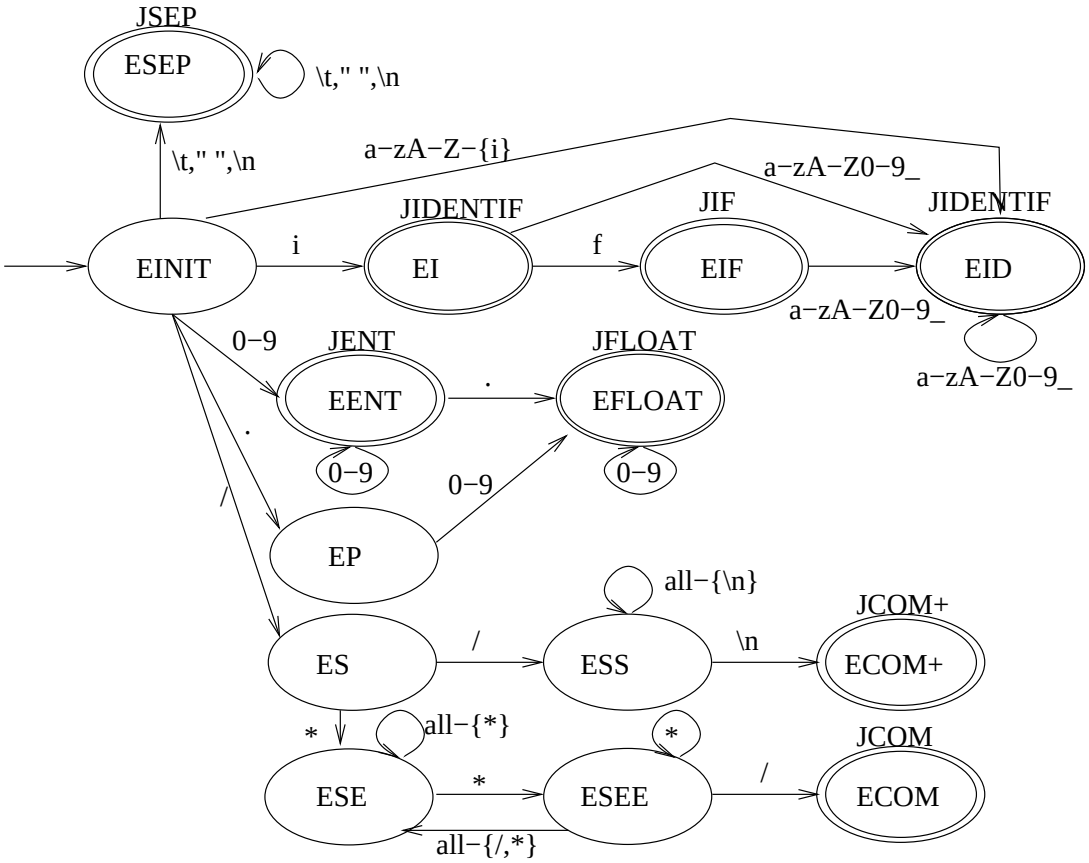

FIGURE 1 – AFD

– afd.h

```
/*=========================================================================
Nom : afd.h     Auteur : Michel Meynard | Définition d'un AFD
=========================================================================*/
#define EINIT 0
#define EI 1
#define EIF 2
#define EID 3
#define ES 4
#define ESS 5
#define ECOMP 6
#define ESE 7
#define ESEE 8
#define ECOM 9
#define ESEP 10
#define EENT 11
#define EP 12
#define EFLOAT 13
#define NBETAT 14

int TRANS[NBETAT][256]; /* table de transition */
int JETON[NBETAT]; /* tableau des jetons */

/** transitions de ed en ef pour tous les char de cd à cf */
void classe(int ed, int cd, int cf, int ef){
  int i;
```

```
  for(i=cd;i<=cf;i++)
    TRANS[ed][i]=ef;
}
void creerAfd(){ /* Construction de l'AFD */
  int i,j;
  for (i=0;i<NBETAT;i++){ /* init */
    for(j=0;j<256;j++) TRANS[i][j]=-1; /* pas de trans */
    JETON[i]=0; /* init tous états non finaux */
  }
  classe(EINIT,'0', '9', EENT); /* entiers et flottants */
  classe(EENT,'0', '9', EENT);
  TRANS[EENT]['.']=EFLOAT;TRANS[EINIT]['.']=EP;
  classe(EP,'0', '9', EFLOAT);
  classe(EFLOAT,'0', '9', EFLOAT);
  JETON[EENT]=300+EENT;JETON[EFLOAT]=300+EFLOAT; /* jetons à retourner */

  classe(EINIT,'a', 'z', EID); /* IF et ID */
  classe(EINIT,'A', 'Z', EID);
  TRANS[EINIT]['i']=EI; /* surcharge */
  classe(EI,'a', 'z', EID);
  classe(EI,'A', 'Z', EID);
  classe(EI,'0', '9', EID);
  TRANS[EI]['_']=EID;
  TRANS[EI]['f']=EIF; /* surcharge */
  classe(EIF,'a', 'z', EID);
  classe(EIF,'A', 'Z', EID);
  classe(EIF,'0', '9', EID);
  TRANS[EIF]['_']=EID;
  classe(EID,'a', 'z', EID);
  classe(EID,'A', 'Z', EID);
  classe(EID,'0', '9', EID);
  TRANS[EID]['_']=EID;
  JETON[EI]=JETON[EID]=300+EID;JETON[EIF]=300+EIF; /* états finaux */

  /* commentaires */
  TRANS[EINIT]['/']=ES;TRANS[ES]['/']=ESS;TRANS[ES]['*']=ESE;
  classe(ESE,0,255,ESE);
  classe(ESS,0,255,ESS);
  classe(ESEE,0,255,ESE);
  TRANS[ESE]['*']=ESEE;TRANS[ESEE]['/']=ECOM;TRANS[ESEE]['*']=ESEE;
  JETON[ECOM]=-1; /* commentaire C à filtrer */
  TRANS[ESS]['\n']=ECOMP;JETON[ECOMP]=-1; /* commentaire C++ */

  /* séparateurs */
  TRANS[EINIT][' ']=TRANS[EINIT]['\t']=TRANS[EINIT]['\n']=ESEP;
  TRANS[ESEP][' ']=TRANS[ESEP]['\t']=TRANS[ESEP]['\n']=ESEP;
  JETON[ESEP]=-1; /* séparateur à filtrer */
}
```

– fonction d'analyse lexicale : `analex.h`.

```
/*=========================================================================
Nom : analex.h  Auteur : Michel Meynard  Définition de la fon : analex()
retourne un entier négatif si erreur, positif si OK, 0 si fin de fichier
le filtrage est permis gâce aux jetons négatifs
=========================================================================*/
#include <string.h>
char lexeme[1024]; /* lexème courant de taille maxi 1024 */
int DEBUG=0; /* débogage */

int analex(){ /* reconnaît un mot sur l'entrée standard */
```

```
  int  etat=EINIT; /* unique état initial */
  int efinal=-1; /* pas d'état final déjà vu */
  int lfinal=0; /* longueur du lexème final */
  int c;char sc[2];int i; /* caractère courant */
  lexeme[0]='\0'; /* lexeme en var globale (pour le main)*/
  while ((c=getchar())!=EOF && TRANS[etat][c]!=-1){ /* Tq on peut avancer */
    sprintf(sc,"%c",c); /* transforme le char c en chaine sc */
    strcat(lexeme,sc); /* concaténation */
    if (DEBUG) printf("%i -- %c --> %i ;",etat,c,TRANS[etat][c]);
    etat=TRANS[etat][c]; /* Avancer */
    if (JETON[etat]){ /* si état final */
      efinal=etat; /* s'en souvenir */
      lfinal=strlen(lexeme); /* longueur du lexeme egalement */
    } /* fin si */
  } /* fin while */
  if (JETON[etat]){ /* état final */
    ungetc(c,stdin); /* rejeter le car non utilisé */
    return (JETON[etat]<0 ? analex() : JETON[etat]);/* ret le jeton ou boucle*/
  }
  else if (efinal>-1){ /* on en avait vu 1 */
    ungetc(c,stdin); /* rejeter le car non utilisé */
    for(i=strlen(lexeme)-1;i>=lfinal;i--)
      ungetc(lexeme[i],stdin); /* rejeter les car en trop */
    lexeme[lfinal]='\0'; /* voici le lexeme reconnu */
    return (JETON[efinal]<0 ? analex() : JETON[efinal]);/* ret jeton ou boucle*/
  }
  else if (strlen(lexeme)==0 && c==EOF)
    return 0; /* cas particulier */
  else if (strlen(lexeme)==0){
    lexeme[0]=c;lexeme[1]='\0'; /* retourner (c,c) */
    return c;
  }
  else {
    ungetc(c,stdin); /* rejeter le car non utilisé */
    for(i=strlen(lexeme)-1;i>=1;i--)
      ungetc(lexeme[i],stdin); /* rejeter les car en trop */
    return lexeme[0];
  }
}
```

– Principal : `analex.c`.

```
/*=========================================================================
Nom : analex.c  Auteur : Michel Meynard  Prog principal appelant analex()
=========================================================================*/
#include <stdio.h>
#include "afd.h" /* Définition de l'AFD et des JETONS */
#include "analex.h" /* Définition de la fon : int analex() */

int main(){ /* Construction de l'AFD */
  int j; /* jeton retourné par analex() */
  char *invite="Saisissez un(des) mot(s) parmi : un entier, un flottant, un identif., le mot clef if, un
  creerAfd(); /* Construction de l'AFD à jeton */
  printf("%s",invite); /* prompt */
  while((j=analex())){ /* analyser tq pas jeton 0 */
    printf("\nRésultat : Jeton = %d ; Lexeme = %s\n%s",j,lexeme,invite);
  }
  return 0;
}
```

Solution 2 Expressions régulières correspondant aux langages réguliers :

1. Langage des mots-clés : $L_{key} = \{if, then, else, throw\} = L(if|then|else|throw)$
2. Langage des littéraux numériques entiers : $L_{c10} = L((1|\ldots|9)(0|1|\ldots|9)*)$, $L_{c8} = L(0(0|1|\ldots|7)*)$, $L_{c16} = L(0x(0|1|\ldots|9|A|B|\ldots|F)+)$. Le signe éventuel est pris en compte lors de l'analyse syntaxique.
3. Langage L_{id} des identificateurs : $L_{id} = L((a|b|\ldots|z|A|B|\ldots|Z)(a|b|\ldots|z|A|B|\ldots|Z|0|1|\ldots|9|\_)*)$.
4. Langage des littéraux flottants décimaux : $L_f = L(((0|1|\ldots|9) + .(0|1|\ldots|9) * |.(0|1|\ldots|9)+)(((e|E)(+|-|\varepsilon)(0|1|\ldots|9)+)|\varepsilon))$
5. Langage des séparateurs : $L_{sep} = L((ESPACE|\backslash t|\backslash n|//[\hat{}\backslash n]*\backslash n|"/*"([\hat{}*]|"*"+[\hat{}*/])*"*"+/)+)$

Solution 3 Source lex :

```
                                /* analflex.l+ */
%{
#include <iostream.h>           /* pour cout */
%}
chiffre         ([0-9])
lettre          ([a-zA-Z])
%%
if              {return 300;}
then            {return 301;}
else            {return 302;}
throw           {return 303;}
0[0-7]+         {return 304;}
0x[0-9A-Fa-f]+  {return 305;}
[1-9]{chiffre}* {return 306;}
{lettre}({lettre}|{chiffre}|_)*                                 {return 307;}
({chiffre}+\.{chiffre}*|\.{chiffre}+)([eE][-+]?{chiffre}+)?     {return 308;}
[ \t\n]+        {/* ne rien faire : filtrer les blancs */}
"//".*\n        {/* ne rien faire : filtrer les commentaires C++ */}
"/*"([^*]|"*"+[^*/])*"*"+"/" {/* ne rien faire : filtrer les commentaires C */}
.               {return yytext[0];}
%%
int main(){
  int j;
  while ((j=yylex())!=0)
    cout<<"Jeton : "<<j<<" de lexème : "<<yytext<<endl;
}
/* autre façon de traiter les commentaires C gérant la non fin de fichier !
 int i,j;
do {
   while ((i=yyinput())!=EOF && i!='*'); // input si C
   if (i==EOF){
     cout<<"Erreur lexicale : commentaires à la C non terminé !\n"; exit(1);
   }
   if ((j=yyinput())!='/') unput(j);
} while (j!='/');
*/
```

Solution 4 Source lex supprimant les lignes vides, et les blancs multiples :

```
        /* delblancs.l+ */
%{
        #include <iostream.h>   /* pour cout */
%}
%%
^[\t ]*\n       {/* filtrer les lignes blanches */}
^[\t ]+         {/* filtrer les débuts blancs et les fin de fic blancs */}
[\t ]+$         {/* filtrer les fins blanches en fin de ligne (\n) */}
[\t ]+          {int c; /* $ ne matche pas avec EOF ! */
                 if ((c=yyinput())!=EOF){
                        unput(c);
```

```
                        cout<<'_'; // à changer en espace !
                }
                else
                  return 0;
                }
%%
int main(){int i; while (i=yylex());}      // tq pas fin de fichier
```

Solution 5 Comptage des caractères, des mots et des lignes :

```
%{  /* comptage.l */
  int nbcar=0; /* nombre de car */
  int nbmot=0; /* nombre de mots */
  int nbligne=0; /* nombre de lignes */
%}
%%
[^\n\t ]+ {nbmot++;nbcar+=yyleng;}
. {nbcar++;}
\n {nbcar++;nbligne++;}
%%
int main(int nbarg, char* argv[]){
  if (nbarg==2){FILE *f=fopen(argv[1],"r");yyin=f;} /* ouverture du fichier */
  yylex();
  printf("\nnbcar : %d ; nbmot : %d ; nbligne : %d\n",nbcar,nbmot,nbligne);
}
```

Solution 6 Soit l'expression régulière e suivante : $e = a((b|c)^{**}|cd)^*b$

1. Un automate fini équivalent à e : $B = (\{a,b,c,d\},\{1,2,3,4\},\{1\},\{4\},\{1a2,2b2,2c2,2c3,2b4,3d2\})$.
2. B non déterministe. $B_{detEtComplet} = (\{a,b,c,d\},\{1,2,23,24,p\},\{1\},\{24\},T)$ avec

$$T = \{1a2, 1(b|c|d)p, 2b24, 2c23, 2(a|d)p, 23b24, 23d2, 23c23, 23ap, 24b24, 24c2324(a|d)p, p(a|b|c|d)p\})$$

.

3. $B_{mini} = B_{detEtComplet}$.

Solution 7 Soit l'automate fini $B = (\{a,b,c\},\{1,2,3,4\},\{1\},\{2,4\},\{1a2,1a3,2b2,2c2,3b4,4c3,4b4\})$.

1. Automate fini déterministe équivalent à B : $B_d = (\{a,b,c\},\{1,2,23,24\},\{1\},\{2,23,24\},T)$ avec

$$T = \{1a23, 23b24, 23c2, 24b24, 24c23, 2b2, 2c2\})$$

.

2. AFD minimal : $B_m = (\{a,b,c\},\{1,2\},\{1\},\{2\},\{1a2,2b2,2c2\})$.

Solution 8 evaldesc avec associativité à droite des opérateurs `+, *`.

```
/*============================================================================
Nom : evaldesc.c        Auteur : Michel Meynard
Maj : 16/06/2005        Creation : 8/1/98
Projet : Evaluation descendante récursive d'expression arithmétique
----------------------------------------------------------------------------
Specification:
Ce fichier contient un évaluateur d'expressions arithmétiques composée de
littéraux entiers sur un car, des opérateurs +, * et du parenthésage ().
On utilise l'associativité à doite : 1+2+3=1+(2+3); 1*2*3=1*(2*3)
Remarque : soit rediriger en entrée un fichier, soit terminer par deux
caractères EOF (Ctrl-D), un pour lancer la lecture, l'autre comme "vrai" EOF.
============================================================================*/
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
```

```
                            /* les macros sont des blocs : pas de ';' apres */
#define AVANCER {jeton=getchar();numcar++;}
#define TEST_AVANCE(prevu) {if (jeton==(prevu)) AVANCER else ERREUR_SYNTAXE}
#define ERREUR_SYNTAXE {printf("\nMot non reconnu : erreur de syntaxe \
au caractère numéro %d, de jeton %d \n",numcar,jeton); exit(1);}
#define INVITE "Veuillez saisir une expression numérique SVP (q pour quitter) : "
void X();int E();int R();int T();int S();int F(); /* déclarations des fonctions */

int jeton;                        /* caractère courant du flot d'entrée */
int numcar=0;                     /* numero du caractère courant (jeton) */

void X(){ /* AXIOME */
  int r;
  if (jeton=='q'){ /* regle : X -> 'q' '\n' */
    AVANCER;
    if (jeton=='\n')
      return; /* OK */
    else
      ERREUR_SYNTAXE; /* q suivi d'autre chose */
  }
  else {
    r=E(); /* regle : X -> E '\n' X */
    if (jeton=='\n'){
      printf("Mot reconnu de valeur : %d\n",r);
      printf(INVITE);
      numcar=0; /* réinit */
      AVANCER; /* avance au prochain jeton */
      X(); /* boucle tq pas 'q' */
    }
    else ERREUR_SYNTAXE;          /* expression reconnue mais reste des car */
  }
}
int E(){                          /* regle : E->TR */
  return R(T());
}
int R(int g){
  if (jeton=='+') {               /* regle : R->+TR */
    AVANCER;
    return R(g+T()); /* associativité à gauche */
/* return g+R(T()); asso à droite */
  }
  else                            /* regle : R->epsilon */
    return g;                     /* ret la partie gauche */
}
int T(){                         /* regle : T->FS */
  return S(F());
}
int S(int g){
  if (jeton=='*') {               /* regle : S->*FS */
    AVANCER;
    return S(g*F()); /* associativité à gauche */
/* return g*S(F()); asso à droite */
  }
  else                            /* regle : S->epsilon */
    return g;                     /* ret la partie gauche */
}
int F(){
  int r;                          /* resultat */
  if (jeton=='(') {               /* regle : F->(E) */
```

```
    AVANCER;
    r=E();
    TEST_AVANCE(')');
  }
  else
    if (isdigit(jeton)) {       /* regle : F->0|1|...|9 */
      r=jeton-'0';              /* valeur comprise entre 0 et 9 */
      AVANCER;
    }
    else ERREUR_SYNTAXE;
  return r;
}
int main(){                         /* Fonction principale */
  printf(INVITE);
  numcar=0; /* init */
  AVANCER;                      /* initialiser jeton sur le premier car */
  X(); /* axiome */
  return 0; /* ne retourne jamais que par X */
}
```

postdesc avec associativité à gauche des opérateurs +, *.

```
/*==============================================================================
Nom : postdesc.c        Auteur : Michel Meynard
Maj : 8/3/99            Creation : 8/1/98
Projet : Traduction postfixe descendante récursive d'expression arithmétique
------------------------------------------------------------------------------
Specification:
Ce fichier contient un traducteur d'expressions arithmétiques infixées
en expressions arithmétiques postfixées. Les expressions d'entrée sont
composées de littéraux entiers sur un car, des opérateurs +, * et du
parenthésage (). Associativité à gauche des opérateurs +, *.
Remarque : soit rediriger en entrée un fichier, soit terminer par deux
caractères EOF (Ctrl-D), un pour lancer la lecture, l'autre comme "vrai" EOF.
==============================================================================*/
#include <stdio.h>
                            /* les macros sont des blocs : pas de ';' apres */
#define AVANCER {jeton=getchar();numcar++;}
#define TEST_AVANCE(prevu) {if (jeton==(prevu)) AVANCER else ERREUR_SYNTAXE}
#define ERREUR_SYNTAXE {printf("\nMot non reconnu : erreur de syntaxe \
au caractère numéro %d \n",numcar); exit(1);}

void E();void R();void T();void S();void F(); /* déclarations des fonctions */

int jeton;                       /* caractère courant du flot d'entrée */
int numcar=0;                    /* numero du caractère courant (jeton) */
char r[100];                     /* resultat */
char *pc=r;                      /* pointeur courant sur la chaine resultat */

void E(){
  T();                           /* regle : E->TR */
  R();
}
void R(){
  if (jeton=='+') {              /* regle : R->+TR */
    AVANCER
    T();
    *pc='+';pc++;                /* écriture du + */
    R();
  }
  else ;                         /* regle : R->epsilon */
```

```
}
void T(){
  F();
  S();                          /* regle : T->FS */
}
void S(){
  if (jeton=='*') {             /* regle : S->*FS */
    AVANCER
    F();
    *pc='*';pc++;               /* écriture du + */
    S();
  }
  else ;                        /* regle : S->epsilon */
}
void F(){
  if (jeton=='(') {             /* regle : F->(E) */
    AVANCER
    E();
    TEST_AVANCE(')')
  }
  else
    if (isdigit(jeton)) {       /* regle : F->0|1|...|9 */
      *pc=(char)jeton;pc++;     /* écriture du chiffre */
      AVANCER
    }
    else ERREUR_SYNTAXE
}
main(){                         /* Fonction principale */
  AVANCER                       /* initialiser jeton sur le premier car */
  E();                          /* axiome */
  if (jeton==EOF) {             /* expression reconnue et rien après */
    *pc='\0';                   /* pour pouvoir être affiché par printf */
    printf("\nMot reconnu d'écriture postfixe : %s\n",r);
  }
  else ERREUR_SYNTAXE           /* expression reconnue mais il reste des car */
}
```

arbindesc avec associativité à gauche des opérateurs `+`, `*`.

```
/*============================================================================
Nom : arbindesc.c       Auteur : Michel Meynard
Maj : 8/3/99            Creation : 11/01/98
Projet :
----------------------------------------------------------------------------
Specification:
Ce fichier analyse une expression arithmétique et affiche l'arbre abstrait
correspondant. Asociativité à gauche.
============================================================================*/
#include <stdio.h>

#include "arbin.h"
#define AVANCER {jeton=getchar();numcar++;}
#define TEST_AVANCE(prevu) {if (jeton==(prevu)) AVANCER else ERREUR_SYNTAXE}
#define ERREUR_SYNTAXE {printf("\nMot non reconnu; erreur de syntaxe \
au caractère numéro %d, de jeton %d \n",numcar,jeton); exit(1);}

Arbin E();Arbin R();Arbin T();Arbin S();Arbin F(); /* déclarations des fonctions */

int jeton;                      /* caractère courant du flot d'entrée */
int numcar=0; /* numero du caractère courant (jeton) */
```

```
Arbin E(){
  return R(T()); /* regle : E->TR */
}
Arbin R(Arbin tg){ /* tg est l'arbin du T de gauche */
  if (jeton=='+') {             /* regle : R->+TR */
    AVANCER;
    return R(construire('+',tg,T()));
  }
  else
    return tg;                  /* regle : R->epsilon */
}
Arbin T(){                   /* regle : T->FS */
  return S(F());
}
Arbin S(Arbin fg){ /* fg est l'arbin du F de gauche */
  if (jeton=='*') {             /* regle : S->*FS */
    AVANCER;
    return S(construire('*',fg,F()));
  }
  else
    return fg;                  /* regle : S->epsilon */
}
Arbin F(){
  Arbin r;                      /* resultat */
  if (jeton=='(') {             /* regle : F->(E) */
    AVANCER;
    r=E();
    TEST_AVANCE(')');
  }
  else
    if (isdigit(jeton)){        /* regle : F->0|1|...|9 */
      r=construire(jeton,creerarbin(),creerarbin());
      AVANCER;
    }
    else ERREUR_SYNTAXE;
  return r;
}
main(){                         /* Fonction principale */
  Arbin r;                      /* resultat */
  AVANCER;                      /* initialiser jeton sur le premier car */
  r=E();                        /* axiome */
  if (jeton==EOF){              /* expression reconnue et rien après */
    printf("\nMot reconnu dont l'arbre abstrait est : \n");
    affichera(r,0);
  }
  else ERREUR_SYNTAXE;          /* expression reconnue mais il reste des car */
}
```

Solution 9 Calculette flottante descendante en C avec flex :

Fichier C

```
/*============================================================================
Nom : calcdesc.c        Auteur : Michel Meynard
Maj : 13/6/2005         Creation : 8/3/99
Projet : Evaluation descendante récursive d'expression arithmétique flottante
----------------------------------------------------------------------------
Specification:
Ce fichier contient un évaluateur d'expressions arithmétiques (calculette)
composée de littéraux flottants, des 5 opérateurs et du parenthésage ().
On utilise l'associativité à doite pour la puissance, à gauche pour les autres.
Remarque : la lecture d'expression est répétitive, en cas d'erreur toute la
```

```
ligne est supprimée.
================================================================================*/
#include <stdio.h>
#include <math.h>

#define AVANCER {jeton=yylex();}
#define TEST_AVANCE(prevu) {if (jeton==(prevu)) AVANCER else ERREUR_SYNTAXE}
#define ERREUR_SYNTAXE {printf("\nMot non reconnu : erreur de syntaxe \
au caractère numéro %d \n",numcar);while (jeton!='\n') AVANCER;erreur=1;}
#define INVITE "Veuillez saisir une expression arithmétique infixée ou q pour quitter S.V.P. :\n"

double valeur;                  /* var globale utilisee par flex */
int jeton;                      /* caractère courant du flot d'entrée */
int numcar;                     /* numero du caractère courant (jeton) */
int erreur;                     /* booléen indiquant s'il y a eu erreur */

int yylex();
void X();double E();double R(double);double T();double S(double);double F();
double G();double C(); /* déclarations des fonctions */

void X(){
  numcar=1;erreur=0;double r;
  if (jeton=='q'){              /* regle : X -> 'q' '\n' */
    AVANCER;
    if (jeton=='\n')
      return;                   /* OK */
    else {
      ERREUR_SYNTAXE;           /* q suivi d'autre chose */
      X();
    }
  }
  else {
    r=E();                      /* regle : X -> E '\n' X */
    if (!erreur && jeton=='\n'){
      printf("Résultat : %.2f\n\n",r);
    }
    else  if (!erreur) {
      ERREUR_SYNTAXE;           /* expression reconnue mais il reste des car */
    }
    printf(INVITE);
    AVANCER;
    X();
  }
}
double E(){
  return R(T());                /* regle : E->TR */
}
double R(double tg){            /* tg est le double du T de gauche */
  if (jeton=='+') {             /* regle : R->+TR */
    AVANCER;
    return R(tg+T());
  }
  else
    if (jeton=='-') {           /* regle : R->-TR */
      AVANCER;
      return R(tg-T());
    }
  return tg;                    /* regle : R->epsilon */
}
```

```
double T(){                     /* regle : T->FS */
  return S(F());
}
double S(double fg){            /* fg est le double du F de gauche */
  if (jeton=='*') {             /* regle : S->*FS */
    AVANCER;
    return S(fg*F());
  }
  else
    if (jeton=='/') {           /* regle : S->/FS */
      AVANCER;
      return S(fg/F());
    }
    else
      return fg;                /* regle : S->epsilon */
}
double F(){                     /* regle : F->GC */
  return C(G());                /* associatif de droite a gauche */
}
double C(double gg){
  if (jeton=='^') {             /* regle : C->^GC */
    AVANCER;
    return pow(gg,C(G()));
  }
  else
    return gg;                  /* regle : C->epsilon */
}
double G(){
  double e=0;
  if (jeton=='(') {             /* regle : G->(E) */
    AVANCER;
    e=E();
    TEST_AVANCE(')');
  }
  else if (jeton==300){         /* regle : G-><flottant> */
    e=valeur;
    /* printf("jeton : %f",e);*/
    AVANCER;
  }
  else ERREUR_SYNTAXE;
  return e;
}
int main(){                         /* Fonction principale */
  printf(INVITE);
  AVANCER;                      /* initialiser jeton sur le premier car */
  X();                          /* axiome */
  return 0;                     /* ne retourne jamais que par X */
}
```

Fichier flex

```
/* calcdesc.fl */
chiffre         ([0-9])
%{
extern double valeur; /* valeur flottante du lexème */
extern int numcar; /* position du car */
%}
%%
{chiffre}+\.{chiffre}*|\.{chiffre}+|{chiffre}+ {
  valeur=atof(yytext);numcar+=yyleng;return 300;
}
[ \t]      {numcar+=1; /* filtrer les blancs */}
```

```
.|\n {numcar+=1;return (int)yytext[0];}
%%
int yywrap(){return 1;}
```

makefile

```
calcdesc : calcdesc.o lex.calc.o
        @echo debut edition de liens de $@
        $(CC) $(CFLAGS) -o $@ $^ -lm
        @echo fin édition de liens : vous pouvez executer $@

lex.calc.o : calcdesc.fl
        @echo debut $(LEX)-compil : $^
        $(LEX) $^
        @echo fin $(LEX)-compil de : $^
        @echo debut compil c de lex.yy.c
        $(CC) $(CFLAGS) -c -o $@ lex.yy.c
        @echo fin compil c et obtention de $@
```

Solution 10 Traducteur d'expression régulière en automate d'état fini.

1. Grammaire "naturelle" :

$$E \rightarrow a|b|\ldots|z|@|0|EE|E*|E'|'E$$

   Cette grammaire est ambiguë : $a|b|c$ peut être générée de 2 façons (associativité à gauche ou à droite).
2. Grammaire récursive à gauche et non ambiguë :

$$\begin{aligned} S &\rightarrow S'|'E|E \\ E &\rightarrow ET|T \\ T &\rightarrow T*|F \\ F &\rightarrow (S)|a|b|\ldots|z|@|0 \end{aligned}$$

3. Grammaire non récursive à gauche et non ambiguë des expressions régulières :

$$\begin{aligned} S &\rightarrow EX \\ X &\rightarrow '|'EX|\varepsilon \\ E &\rightarrow TR \\ R &\rightarrow TR|\varepsilon \\ T &\rightarrow FY \\ Y &\rightarrow *Y|\varepsilon \\ F &\rightarrow (S)|a|b|\ldots|z|@|0 \end{aligned}$$

4. Analyseur récursif descendant construisant l'arbre abstrait et l'automate non déterministe.

```
/*=============================================================================
Nom : expreg.c          Auteur : Michel Meynard
Maj : 11/04/2000        Creation : 11/03/98
Projet :
-------------------------------------------------------------------------------
Specification:
Ce fichier analyse une expression régulière, affiche l'arbre abstrait
correspondant, puis l'automate non déterministe construit par l'algo de
Thompson. Associativité à gauche des opérateurs | et concat
=============================================================================*/
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include "arbin.h"
#include "afn.h"
#include "afd.h"
#include "afdm.h"
#define AVANCER {jeton=getchar();numcar++;}
```

```
#define TEST_AVANCE(prevu) {if (jeton==(prevu)) AVANCER else ERREUR_SYNTAXE}
#define ERREUR_SYNTAXE {printf("\nMot non reconnu; erreur de syntaxe \
au caractère numéro %d \n",numcar); exit(1);}

Arbin S();Arbin X(Arbin);Arbin E();Arbin R(Arbin);Arbin T();Arbin Y(Arbin);Arbin F();
/* déclarations en avant des fonctions */
afn a; /* l'afn */
int jeton;                      /* caractère courant du flot d'entrée */
int numcar=0;                   /* numero du caractère courant (jeton) */

Arbin S(){ /* union | */
  Arbin g; /* resultat de E */
  g=E(); /* regle : S->EX */
  return X(g);
}
Arbin X(Arbin g){ /* suite de union */
  Arbin d;
  if (jeton=='|') { /* regle : X->|EX */
    AVANCER;
    d=E();
    if (racine(g)=='0') /* 0|e=e */
      g=d;
    else if (racine(d)!='0')
      g=construire('|',g,d);  /* g et d non vides 0|e=e|0=e */
    return X(g);
 }
  return g; /* regle : X->epsilon */
}
Arbin E(){ /* concaténation */
  Arbin g; /* resultat de T */
  g=T(); /* regle : E->TR */
  return R(g);
}
Arbin R(Arbin g){ /* suite de concat */
  Arbin d; /* resultat de T */
  if (jeton=='(' || islower(jeton) || jeton=='0' || jeton=='@'){
    d=T(); /* regle : R->TR : premiers(T)= premiers(F) */
    if (racine(g)=='@' || racine(d)=='0')
      g=d; /* @.e=e e.0=0 */
    else if (racine(d)!='@' && racine(g)!='0')
      g=construire('.',g,d); /* e.z */
    return R(g);
  }
  return g; /* regle : R->epsilon */
}

Arbin T(){
  Arbin g; /* resultat de F */
  g=F();
  return Y(g); /* regle : T->FY */
}
Arbin Y(Arbin g){ /* ret vrai (1) si au moins 1 étoile */
  Arbin d; /* resultat de Y */
  if (jeton=='*') { /* regle : Y->*Y */
    AVANCER;
    if (racine(g)=='@'||racine(g)=='0')
      d=construire('@',creerarbin(),creerarbin()); /* @*=0*=@ */
    else if (racine(g)=='*')
      d=g; /* e**=e* */
```

```
    else
      d=construire('*',g,creerarbin()); /* ni epsilon ni ensemble vide ni * */
    return Y(d);
  }
  else return g; /* regle : Y->epsilon */
}
Arbin F(){
  Arbin r;                      /* resultat */
  if (jeton=='(') {             /* regle : F->(S) */
    AVANCER;
    r=S();
    TEST_AVANCE(')');
  }
  else
    if (islower(jeton)||jeton=='0'||jeton=='@'){ /* regle : F->a|b|...|z|@|0 */
/* @ epsilon, 0 ensemble vide */
      r=construire(jeton,creerarbin(),creerarbin());
      AVANCER;
    }
    else ERREUR_SYNTAXE;
  return r;
}

int nombreEtat(Arbin a){ /* calcul du nombre d'état de l'afn */
  if (videa(a))
    return 0;
  else
    if (racine(a)=='.') /* la concat ne rajoute pas d'état */
      return nombreEtat(sag(a))+nombreEtat(sad(a));
    else /* union, *, symbole nécessitent 2 nouveaux états */
      return 2+nombreEtat(sag(a))+nombreEtat(sad(a));
}

int main(){                         /* Fonction principale */
  Arbin r;                      /* resultat */
  int nbetat; /* nombre d'états */
  afn n;
  afde d;
  afd d1,d2;
  printf("\nVeuillez saisir une expression régulière suivie de <Entrée> S.V.P.\n");
  AVANCER;                      /* initialiser jeton sur le premier car */
  r=S();                        /* axiome */
  if (jeton=='\n'){              /* expression reconnue et rien après */
    printf("\nMot reconnu dont l'arbre abstrait est : \n");
    affichera(r,0);
    if (racine(r)=='0'){
      printf("\nLe langage de l'expression régulière est vide !\n");
      exit(0);
    }
    printf("\nConstruction de l'AFN de %i états ...\n", nbetat=nombreEtat(r));
    n=afn_creer(nbetat,r);
    printf("\nAFN : %s",afn_chaine(n));
    d=afde_creer(n);
    printf("\nAFD : %s\n",afde_chaine(d));
    d1=afd_creer(d);
    printf("\nAFD : %s",afd_chaine(d1));
    afn_detruire(n);
    d2=minimiser(d1);
    printf("\nAFDM : %s\n",afd_chaine(d2));
```

```
  }
  else ERREUR_SYNTAXE; /* expression reconnue mais il reste des car */
  return 0;
}
```

5. Optimisation déjà réalisée dans le programme précédent.
6. Table d'analyse de l'automate à pile en analyse descendante :

V_N	*premiers*	*TabSuivants*		a	$($	$)$	$\|$	$*$	\$
S	$a,($	\$, $)$	S	$S \to EX$	$S \to EX$				
X	$\varepsilon, \|$	\$, $)$	X			$X \to \varepsilon$	$X \to \|EX$		$X \to \varepsilon$
E	$a,($	$\|$, \$, $)$	E	$E \to TR$	$E \to TR$				
R	$\varepsilon, a,($	$\|$, \$, $)$	R	$R \to TR$	$R \to TR$	$R \to \varepsilon$	$R \to \varepsilon$		$R \to \varepsilon$
T	$a,($	$\|$, \$, $), a,($	T	$T \to FY$	$T \to FY$				
Y	$\varepsilon, *$	$\|$, \$, $), a,($	Y	$Y \to \varepsilon$	$Y \to \varepsilon$	$Y \to \varepsilon$	$Y \to \varepsilon$	$Y \to *Y$	$Y \to \varepsilon$
F	$a,($	$\|$, \$, $), a,($	F	$F \to a$	$F \to (S)$				

7. états successifs de la pile sur $ab*|c$.

Pile	*Flotd'entre*	*Action*
\$$S$	$ab*\|c$\$	$S \to EX$
\$$XE$	$ab*\|c$\$	$E \to TR$
\$$XRT$	$ab*\|c$\$	$T \to FY$
\$$XRYF$	$ab*\|c$\$	$F \to a$
\$$XRYa$	$ab*\|c$\$	$AVANCER$
\$$XRY$	$b*\|c$\$	$Y \to \varepsilon$
\$$XR$	$b*\|c$\$	$R \to TR$
\$$XRT$	$b*\|c$\$	$T \to FY$
\$$XRYF$	$b*\|c$\$	$F \to a$
\$$XRYa$	$b*\|c$\$	$AVANCER$
\$$XRY$	$*\|c$\$	$Y \to *Y$
\$$XRY*$	$*\|c$\$	$AVANCER$
\$$XRY$	$\|c$\$	$Y \to \varepsilon$
\$$XR$	$\|c$\$	$R \to \varepsilon$
\$$X$	$\|c$\$	$X \to \|EX$
\$$XE|$	$\|c$\$	$AVANCER$
\$$XE$	c\$	$E \to TR$
\$$XRT$	c\$	$T \to FY$
\$$XRYF$	c\$	$F \to a$
\$$XRYa$	c\$	$AVANCER$
\$$XRY$	\$	$R \to \varepsilon$
\$$XR$	\$	$Y \to \varepsilon$
\$$X$	\$	$X \to \varepsilon$
\$	\$	$ACCEPTER$

8. fonction NombreEtat(Arbin a) dans le programme suivant.
9. Construction de Thompson :

```
/*=========================================================================
Nom : afn.h             Auteur : Michel Meynard
Maj :                   Creation : 17/2/2003
Projet : expreg.c
-------------------------------------------------------------------------
Specification:
Ce fichier déclare les AFN de Thompson.
=========================================================================*/
/* un afn : tableau nbetat-1 lignes par 3 colonnes (nbetat>1)
   col 1 : symbole de trans
   si symbole=epsilon alors 1 ou 2 transitions par epsilon
   col 2 : etat destination
   col 3 : etat destination
*/
#ifndef _AFN
```

```
#define _AFN
#include "arbin.h"

typedef struct afn_struct {
  int **trans;       /* trans : un tableau dynamique de nbetat*3 int */
  int nbetat; /* nombre d'états */
  int etat; /* état courant */
  int etatInit;
  int etatFin; /* etat initial et final de l'afn */
} afn_struct;
typedef struct afn_struct * afn;

afn afn_creer(int nbe,Arbin r); /* constructeur */
void afn_detruire(afn a); /* désallouer */
/* fonction récursive */
void afn_construire(afn a,int *ei,int *ef,Arbin r);
/* construction d'un afn non vide à partir d'un arbre binaire (exp reg) */
char * afn_chaine(afn a);
#endif
/*============================================================================
Nom : afn.c             Auteur : Michel Meynard
Maj :                   Creation : 10/06/99
Projet : expreg.c
----------------------------------------------------------------------------
Specification:
Ce fichier permet de gérer un AFN de Thompson.
============================================================================*/
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>
#include "arbin.h"
#include "afn.h"

afn afn_creer(int nbe,Arbin r){ /* constructeur */
  afn a;
  int i;
  a=(afn)malloc(sizeof(afn_struct));
  a->trans=(int**) malloc(((a->nbetat=nbe)-1)*sizeof(int));
  for (i=0;i<a->nbetat-1;i++)
    a->trans[i]=(int*) malloc(3*sizeof(int));
  a->etat=a->etatFin=a->etatInit=-1;
  afn_construire(a,&(a->etatInit),&(a->etatFin),r);
  return a;
}
void afn_detruire(afn a){
  int i;
  for (i=0;i<a->nbetat-1;i++)
    free(a->trans[i]); /* désallouer */
  free(a->trans);
}
/* fonction récursive modifiant l'afn pointé */
void afn_construire(afn a,int *ei,int *ef,Arbin r){
  /* construction d'un afn non vide à partir d'un arbre binaire (exp reg) */
  int ig,fg,id,fd; /* états initiaux et finaux de g et d */
  if (islower(racine(r))||racine(r)=='@'){ /* symbole ou epsilon */
    a->etat++; /* nouvel état */
    *ei=a->etat; /* état initial */
    a->trans[*ei][0]=racine(r); /* affectation du symbole */
```

```
    a->etat++; /* nouvel état */
    a->trans[*ei][1]=a->etat; /* état de transition */
    *ef=a->etat; /* état final */
    a->trans[*ei][2]=-1; /* pas de 2ème transition */
  }
  else if (racine(r)=='.'){ /* concat */
    afn_construire(a,&ig,&fg,sag(r)); /* récursif sur sag() */
    afn_construire(a,&id,&fd,sad(r)); /* récursif sur sad() */
    a->trans[fg][0]='@'; /* symbole : epsilon */
    a->trans[fg][1]=id; /* transition epsilon vers init de droite */
    a->trans[fg][2]=-1; /* pas de 2ème transition */
    *ei=ig;*ef=fd; /* l'état final reste celui de droite */
  }
  else if (racine(r)=='|'){ /* union */
    afn_construire(a,&ig,&fg,sag(r)); /* récursif sur sag() */
    afn_construire(a,&id,&fd,sad(r)); /* récursif sur sad() */
    a->etat++; /* nouvel état initial */
    a->trans[a->etat][0]='@'; /* affectation du symbole epsilon */
    a->trans[a->etat][1]=ig; /* transition epsilon vers init de gauche */
    a->trans[a->etat][2]=id; /* transition epsilon vers init de droite */
    *ei=a->etat; /* nouvel état initial */
    a->etat++; /* nouvel état final */
    a->trans[fg][0]='@'; /* symbole : epsilon */
    a->trans[fg][1]=a->etat; /* transition epsilon vers nouveau final */
    a->trans[fg][2]=-1; /* pas de 2ème transition */
    a->trans[fd][0]='@'; /* symbole : epsilon */
    a->trans[fd][1]=a->etat; /* transition epsilon vers nouveau final */
    a->trans[fd][2]=-1; /* pas de 2ème transition */
    *ef=a->etat; /* état final */
  }
  else if (racine(r)=='*'){ /* étoile */
    afn_construire(a,&ig,&fg,sag(r)); /* récursif sur sag() */
    a->etat++; /* nouvel état initial */
    a->trans[a->etat][0]='@'; /* affectation du symbole epsilon */
    a->trans[a->etat][1]=ig; /* transition epsilon vers init de gauche */
    *ei=a->etat; /* nouvel état initial */
    a->etat++; /* nouvel état final */
    a->trans[*ei][2]=a->etat; /* transition epsilon vers nouveau final */
    a->trans[fg][0]='@'; /* symbole : epsilon */
    a->trans[fg][1]=a->etat; /* transition epsilon vers nouveau final */
    a->trans[fg][2]=ig; /* retour vers l'état initial pour bouclage */
    *ef=a->etat; /* état final */
  }
}
char * afn_chaine(afn a){
  int etat;
  char *sd=(char *)malloc(2048),s[128]; /* ATTENTION */
  sprintf(sd,"état initial : %4d; état final : %4d\n",a->etatInit,a->etatFin);
  for(etat=0;etat<a->nbetat-1;etat++){ /* le dernier état est final ! */
    sprintf(s,"(%4d %c %4d),",etat,a->trans[etat][0],a->trans[etat][1]);
    strcat(sd,s);
    if (a->trans[etat][2]!=-1){ /* 2 transitions */
      sprintf(s,"(%4d %c %4d),",etat,a->trans[etat][0],a->trans[etat][2]);
      strcat(sd,s);
    }
  }
  sd[strlen(sd)-1]='\n';
  return sd;
}
```

```
/*==========================================================================
Nom: afd.h              Auteur: Michel Meynard
Maj: 16/06/2005         Creation: 17/2/2003
Projet:
--------------------------------------------------------------------------
Specification: déclaration utiles à la gestion d'un afd

==========================================================================*/
#ifndef _AFD
#define _AFD

#include "afn.h"
#include "../../../../../C/Sd/sd.h"

/* ----------------------AFDE---------------------------------------------- */
/** Un AFDE est un AFD dont les états dont des Ensembles d'états de l'AFN */
typedef struct afde_struct { /* Autom d'états finis déterministe */
  ensg einit; /* état initial (ens d'états) */
  ensg efinaux; /* ens des ens d'états finaux */
  assog graphe; /* de clé : un ens d'états;
   de valeur : une assog(char, ens d'états)  */
} afde_struct;
typedef struct afde_struct * afde;

void afde_eps_ferme(ensg e,afn a);
/* EpsilonFermeture de e : ajoute les états accessibles par epsilon */

afde afde_creer(afn n); /* constructeur */
char* afde_chaine(afde a); /* convertit en chaine */

/* ----------------------AFD----------------------------------------------- */
/** Un AFD est un AFDE dont on a renuméroté chaque état par un entier 0..n-1 */
typedef struct afd_struct { /* Autom d'états finis entiers déterministes */
  int einit; /* état initial (entier) */
  ensg efinaux; /* ens d'états finaux */
  assog graphe; /* de clé : un état;
   de valeur : une assog(char, état entier)  */
} afd_struct;
typedef struct afd_struct * afd;

afd afd_creer(afde a); /* constructeur */

char* afd_chaine(afd a);
#endif
/*==========================================================================
Nom: afd.c              Auteur: Michel Meynard
Maj: 20/2/2003          Creation: 17/2/2003
Projet:
--------------------------------------------------------------------------
Specification: définitions des fonctions

==========================================================================*/
#include <stdio.h>
#include <stdlib.h>
#include "../../../../../C/Sd/sd.h"

#include "afd.h"

int DEBUG=0;
```

```
afde afde_creer(afn n){
  afde a; /* l'afde formé d'ens */
  assog trans; /* transition (char, ens), */
  ensg atraiter,etat,nouvetat; /* ensemble des ens d'états à traiter */
  ensg_it it; /* itérateur sur l'ens des états à traiter */
  char c,*pc; /* char de transition */
  int *pi,*pj; /* états de l'afn */
  a=(afde)malloc(sizeof(afde_struct)); /* allocation */
  a->einit=ensg_entier_creer(10); /* ens d'états initiaux */
  ensg_ajouter(a->einit,entier_creer(n->etatInit)); /* unique état init de afn */
  afde_eps_ferme(a->einit,n);
  if (DEBUG) printf("\nEtat initial : {%s}\n",ensg_chaine(a->einit));
  a->efinaux=ensg_creer(10,/* ensemble d'ensemble d'entiers */
     (int(*)(void *,void *))ensg_egal,
     (unsigned int(*)(void *))ensg_hash,
     (char*(*)(void *))ensg_chaine
     ); /* ensg */
  a->graphe=assog_creer(10, /* à chaque état (ensg),une assog (char, état) */
      (int(*)(void *,void *))ensg_egal,/* clé = ens */
      (unsigned int(*)(void *))ensg_hash,
      (char*(*)(void *))ensg_chaine,
      (char*(*)(void *))assog_chaine /* valeur = asso trans */
      ); /* assog */
  atraiter=ensg_creer(10,/* ensg d'ens d'entier */
   (int(*)(void *,void *))ensg_egal, /* clé ens */
   (unsigned int(*)(void *))ensg_hash, /* clé entière */
   (char*(*)(void *))ensg_chaine
   ); /* ensg */
  ensg_ajouter(atraiter,a->einit); /* au début, que l'état initial */

  while ((etat=ensg_extraire(atraiter))){ /* BOUCLE PRINCIPALE */
    /* tq il reste des états non traités */
    if (DEBUG) printf("Etat courant : %s\n",ensg_chaine(etat));
    trans=assog_creer(10,/* ens des transitions */
    (int(*)(void *,void *))car_egal,/* clé = car */
    (unsigned int(*)(void *))car_hash,
    (char*(*)(void *))car_chaine,
    (char*(*)(void *))ensg_chaine /* valeur = ensg etats */
    );
    assog_ajouter(a->graphe,etat,trans); /* ajouter l'état et des trans vide */
    if (ensg_present(etat,entier_creer(n->etatFin))){ /* si l'état est final*/
      ensg_ajouter(a->efinaux,etat);
    }
    /* pour chaque état de l'afn */
    it=ensg_it_creer(etat); /* parcourir chaque état de l'afn */
    while((pi=ensg_it_prochain(it))){ /* *pi est un état afn */
      if (*pi!=n->etatFin && (c=n->trans[*pi][0])!='@'){
/* si pas final et pas epsilon */
pj=entier_creer(n->trans[*pi][1]); /* état afn destinataire */
nouvetat=assog_valeur(trans,&c); /* état déjà présent ? */
if (nouvetat==NULL){ /* char pas encore vu */
  nouvetat=ensg_entier_creer(10); /* alloc */
  ensg_ajouter(nouvetat,pj);
  /* le nouvel état dest */
  assog_ajouter(trans,car_creer(c),nouvetat);
  if (DEBUG) printf("  trans (%c) --> %s\n",c,ensg_chaine(nouvetat));
}
else{ /* etat - c -> nouvetat dans trans */
```

```
  ensg_ajouter(nouvetat,pj); /* 1 état afn */
}
      } /* pour les epsilon, rien à faire */
    } /* fin du parcours des états de l'afn */
    assog_it_detruire(it);
    it=assog_it_creer(trans); /* parcourir les trans */
    while(assog_it_prochain(it,(void**)&pc,(void**)&nouvetat)){
      /* *pc est un char, nouvetat est un ens d'entier */
      afde_eps_ferme(nouvetat,n); /* fermer epsilon par l'afn */
      if (DEBUG) printf("epsilon fermeture : %s\n",ensg_chaine(nouvetat));
      if(ensg_absent(a->graphe,nouvetat)){ /* nouvetat pas encore traité */
ensg_ajouter(atraiter,nouvetat); /* même s'il y était déjà */
      }
    }
    assog_it_detruire(it);
  } /* atraiter est vide */
  ensg_detruire(atraiter);
  return a;
}

/* afde_chaine(afde a) */
char* afde_chaine(afde a){
  chaine ch;
  assog_it it,it2; /* itérateur */
  void* etat,*trans;
  void* pc,*dest;

  ch=chaine_creer("Afd avant renumérotation ; l'état initial est : {");
  ch=chaine_cat(ch,ensg_chaine(a->einit));
  ch=chaine_cat(ch,"}");

  it=assog_it_creer(a->graphe);
  while(assog_it_prochain(it,(void**)&etat,(void**)&trans)){
    ch=chaine_cat(ch,"\nEtat : {");
    ch=chaine_cat(ch,a->graphe->cle_chaine(etat));
    ch=chaine_cat(ch,"} ");
    if (ensg_present(a->efinaux,etat)){
      ch=chaine_cat(ch,"FINAL : ");
    }else{
      ch=chaine_cat(ch,": ");
    }
    it2=assog_it_creer(trans);
    while(assog_it_prochain(it2,(void**)&pc,(void**)&dest)){
      ch=chaine_cat(ch,"--");
      ch=chaine_cat(ch,car_chaine(pc));
      ch=chaine_cat(ch,"--> {");
      ch=chaine_cat(ch,ensg_chaine(dest));
      ch=chaine_cat(ch,"}, ");
    }
    assog_it_detruire(it2);
  }
  assog_it_detruire(it);
  return ch;
}

/* EpsilonFermeture de e : ajoute les états accessibles par epsilon */
void afde_eps_ferme(ensg e,afn n){
  int *etat,i,*succ,j; /* état courant (etat,i), suivant (succ,j) */
  ensg atraiter; /* ens des états a traiter */
```

```
  atraiter=ensg_clone(e); /* init avec chaque état de l'afn */

  while((etat=ensg_extraire(atraiter))){ /* traiter 1 fois chaque état */
    i=*etat;
    if (i!=n->etatFin && n->trans[i][0]=='@'){ /* epsilon */
      j=n->trans[i][1]; /* au moins 1 successeur */
      if (ensg_absent(e,&j)){ /* pas encore traité */
ensg_ajouter(e,succ=entier_creer(j));
/* ajouter l'état accessible */
ensg_ajouter(atraiter,succ);
/* atraiter est inclus ou égal à e */
      }
      if ((j=n->trans[i][2])!=-1){
ensg_ajouter(e,succ=entier_creer(j));
/* ajouter l'état accessible */
ensg_ajouter(atraiter,succ);
/* atraiter est inclus ou égal à e */
      }
    }
  }
}

afd afd_creer(afde a){ /* NUMEROTATION de a dans a1 */
  assog trans,nouvtrans; /* transition (char, ens), */
  ensg atraiter,etat,nouvetat; /* ensemble des ens d'états à traiter */
  char *pc; /* char de transition */
  int *dest,*src,numetat; /* états de l'afd */
  assog renum;
  afd a1;

  a1=(afd)malloc(sizeof(afd_struct)); /* puis renuméroté et retourné */
  a1->efinaux=ensg_entier_creer(10);
  a1->graphe=assog_creer(10,  /* à chaque état ent, une assog (char, ent) */
       (int(*)(void *,void *))entier_egal, /* clé entière */
       (unsigned int(*)(void *))entier_hash,
       (char*(*)(void *))entier_chaine,
       (char*(*)(void *))assog_chaine /* valeur asso trans */
       ); /* assog */
  renum=assog_creer(10,/* à chaque état de a, un entier de a1 */
  (int(*)(void *,void *))ensg_egal,/* clé ens */
  (unsigned int(*)(void *))ensg_hash,
  (char*(*)(void *))ensg_chaine,
  (char*(*)(void *))entier_chaine /* valeur entière */
  ); /* assog */
  numetat=0; /* état initial */
  atraiter=ensg_creer(10,/* ens d'ens d'entier */
   (int(*)(void *,void *))ensg_egal, /* clé ens */
   (unsigned int(*)(void *))ensg_hash, /* clé entière */
   (char*(*)(void *))ensg_chaine
   ); /* ens des états de l'afd à traiter */
  /* ce qui est dans atraiter est deja numéroté */
  ensg_ajouter(atraiter,a->einit);
  a1->einit=numetat;
  assog_ajouter(renum,a->einit,(entier_creer(numetat++)));/* {8,0,2} --> 0 */
  /* pour chaque état source de trans */
  while((etat=ensg_extraire(atraiter))){ /* etat ens */
    src=assog_valeur(renum,etat); /* état src dans afd */
    /*printf("debug:afd_creer:renum : %s\nsrc=%i\n",assog_chaine(renum),*src); */
```

```
    /*     printf("debug:atraiter:%s\n",assog_chaine(atraiter)); */
    if (ensg_present(a->efinaux,etat)){
      ensg_ajouter(a1->efinaux,src); /* src est final */
    }
    nouvtrans=assog_creer(10,/* ens des transitions */
(int(*)(void *,void *))car_egal,/* clé ens */
(unsigned int(*)(void *))car_hash,
(char*(*)(void *))car_chaine,
(char*(*)(void *))entier_chaine /* valeur asso trans */
);
    trans=assog_valeur(a->graphe,etat); /* transitions actuelles */
    while(assog_extraire(trans,(void**)&pc,(void**)&nouvetat)){
      if ((dest=assog_valeur(renum,nouvetat))==NULL){ /* la dest n'existe pas */
assog_ajouter(renum,nouvetat,(dest=entier_creer(numetat++)));
ensg_ajouter(atraiter,nouvetat);
/*  printf("debug:atraiter:%s\n",assog_chaine(atraiter)); */
      }
      assog_ajouter(nouvtrans,pc,dest); /* nouv transition 0 --a--> 1 */
    }
    assog_ajouter(a1->graphe,src,nouvtrans);
  }
  return a1;
}

char* afd_chaine(afd a){
  chaine ch;
  assog_it it,it2; /* itérateur */
  void* etat,*trans;
  void* pc,*dest;

  ch=chaine_creer("Afd : l'état initial est ");
  ch=chaine_cat(ch,entier_chaine(&a->einit));

  it=assog_it_creer(a->graphe);
  while(assog_it_prochain(it,(void**)&etat,(void**)&trans)){
    ch=chaine_cat(ch,"\nEtat : ");
    ch=chaine_cat(ch,a->graphe->cle_chaine(etat));
    ch=chaine_cat(ch," ");
    if (ensg_present(a->efinaux,etat)){
      ch=chaine_cat(ch,"FINAL : ");
    }else{
      chaine_cat(ch,": ");
    }
    it2=assog_it_creer(trans);
    while(assog_it_prochain(it2,(void**)&pc,(void**)&dest)){
      ch=chaine_cat(ch,"--");
      ch=chaine_cat(ch,car_chaine(pc));
      ch=chaine_cat(ch,"--> ");
      ch=chaine_cat(ch,entier_chaine(dest));
      ch=chaine_cat(ch,", ");
    }
    assog_it_detruire(it2);
  }
  assog_it_detruire(it);
  ch=chaine_cat(ch,"\n");
  return ch;
}
```

makefile :

```
expreg : expreg.o arbin.o afn.o afd.o ../../../../../C/Sd/libsd.a
```

```
	$(CC) $(CFLAGS) -o expreg $^
	@echo fin édition de liens : vous pouvez executer $@
```

Solution 11 Exercice 10 en analyse ascendante.

1. source bison de la grammaire récursive à gauche non ambiguë :

```
  /* expregasrecg.y */
%{
#include "arbin.h"
void yyerror(char*);
int yylex();
char * invite="\nVeuillez saisir une expression régulière suivie de <Entrée> S.V.P.\n";
%}
%union { /* YYSTYPE */
Arbin typeArbin;
int typeInt;
}
/* definition des jetons */
%token<typeInt> SYMBOLE
/* definition des types des non terminaux */
%type<typeArbin> s e t f
%%

liste   :       /* chaine vide sur fin de fichier Ctrl-D */
        |       liste  ligne {printf(invite);}
        ;
ligne   :       '\n'/* ligne vide : expression vide */
        |       error '\n'{yyerrok; /* après la fin de ligne */}
        |       s '\n'{affichera($1,0);}
        ;
s       :       s '|' e {$$= construire('|',$1,$3);}
        |       e       {$$=$1;}
        ;
e       :       e t     {$$= construire('.',$1,$2);}
        |       t       {$$=$1;}
        ;
t       :       t '*'   {$$= construire('*',$1,creerarbin());}
        |       f       {$$=$1;}
        ;
f       :       '(' s ')'  {$$=$2;}
        |     SYMBOLE {$$=construire(yylval.typeInt,creerarbin(),creerarbin());}
        ;
%%
int yylex(){int i=getchar();
if ((i>='a' && i<='z')||i=='@'||i=='0')
  {yylval.typeInt=i;return SYMBOLE;}
else return i;
      }
void yyerror(char *s) {fprintf(stderr,"%s\n",s);}
int main(){/*yydebug=1;*/printf(invite);return yyparse();}
```

2. source bison de la grammaire naturelle :

```
/* expregas.y */
%{
#include <ctype.h>
#include "arbin.h"
#include "afnThomp.h"
void yyerror(char*);
int yylex();
int nombreEtat(Arbin);
```

```
char * invite="\nVeuillez saisir une expression régulière suivie de <Entrée> S.V.P.\n";
%}
%union { /* YYSTYPE */
Arbin a;
int i;
}

%token<i> LETTRE         /* definition des jetons */

%type<a> expr           /* definition des types des non terminaux */

%left '|'     /* traitement des priorites sur les operateurs du moins au plus */
%left CONCAT LETTRE '('  /* symbole virtuel pour la concaténation (implicite) */
                         /* nécessaires pour conflits avec CONCAT */
%left '*'
%%

liste   : /* chaine vide sur fin de fichier Ctrl-D */
        |       liste  ligne {printf(invite);}
        ;
ligne   :       '\n'            /* ligne vide : expression vide */
        |       error '\n'      {yyerrok; /* après la fin de ligne */}
        |       expr '\n'
        {   int nbetat;
            affichera($1,0);
            if (racine($1)=='0')
                printf("\nLe langage de l'expression régulière est vide !\n");
            else{
                printf("\nConstruction de l'Automate de %i états ...\n", nbetat=nombreEtat($1));
                initAfn(nbetat);
                construireAfn($1);
                afficherAfn();
            }
        }
        ;
expr    :       '(' expr ')'    {$$ = $2;}
        |       expr expr  %prec CONCAT
        {Arbin g,d;g=$1;d=$2; /* CONCAT */
         if (videa(d))
           $$= g;
         else { /* g et d non vides */
           if (racine(g)=='@'){ /* @.e=e.@=e */
             videra(&g);
             $$= d;
           }
           else if (racine(d)=='@'){
             videra(&d);
             $$= g;
           }
           else if (racine(g)=='0'){    /* 0.e=e.0=0 */
             videra(&d);
             $$= g;
           }
           else if (racine(d)=='0'){
             videra(&g);
             $$= d;
           }
           else $$= construire('.',g,d);
         }
```

```
       }
        |       expr '|' expr
        {Arbin g,d;g=$1;d=$3; /* UNION */
         if (videa(d))
           $$= g;
         else { /* g et d non vides */
           if (racine(g)=='0'){ /* 0|e=e|0=e */
             videra(&g);
             $$= d;
           }
           else if (racine(d)=='0'){    /* e|0=e */
             videra(&d);
             $$= g;
           }
           else
             $$= construire('|',g,d);
         }
       }
        |       expr '*'
        {Arbin g=$1; /* ETOILE */
         if (racine(g)=='@'||racine(g)=='0'){   /* @*=0*=@ */
           videra(&g);
           $$= construire('@',creerarbin(),creerarbin());
         }
         else                   /* ni epsilon ni ensemble vide */
           if (racine(g)=='*')
             $$=g;
           else
             $$= construire('*',g,creerarbin());
       }
        | LETTRE     {$$=construire($1,creerarbin(),creerarbin());}
        ;
%%
int yylex(){int i=getchar();
        if (islower(i)||i=='@'||i=='0')
          {yylval.i=i;return LETTRE;}
        else return i;
      }
void yyerror(char *s) {fprintf(stderr,"%s\n",s);}
int nombreEtat(Arbin a){        /* calcul du nombre d'état de l'afn */
  if (videa(a))
    return 0;
  else
    if (racine(a)=='.')         /* la concat ne rajoute pas d'état */
      return nombreEtat(sag(a))+nombreEtat(sad(a));
    else                /* union, *, symbole nécessitent 2 nouveaux états */
      return 2+nombreEtat(sag(a))+nombreEtat(sad(a));
}

int main(){/*yydebug=1;*/printf(invite);return yyparse();}
/*============================================================================
Nom : afnThomp.h        Auteur : Michel Meynard
Maj : 13/6/2005         Creation : 13/6/2005
Projet : expregas
----------------------------------------------------------------------------
Specification:
Fichier d'en-tête pour gérer un AFN de Thompson.
============================================================================*/
/** retourne un afn : tableau nbetat-1 lignes par 3 colonnes (nbetat>1)
```

```
 *  col 1 : symbole de trans
 *   si symbole=epsilon alors 1 ou 2 transitions par epsilon
 *  col 2 : etat destination
 *  col 3 : etat destination
 */
int ** initAfn(int nbetat);

/** construction d'un afn non vide à partir d'un arbre binaire (exp reg) */
void construireAfn(Arbin);

/**  affiche l'afn */
void afficherAfn();
/*=============================================================================
Nom : afnThomp.c        Auteur : Michel Meynard
Maj : 13/6/2005         Creation : 10/06/99
Projet : expregas
-------------------------------------------------------------------------------
Specification:
Ce fichier permet de gérer un AFN de Thompson.
=============================================================================*/
#include <stdio.h>
#include <stdlib.h>
 #include <ctype.h>
#include "arbin.h"

static int **afn=NULL; /* afn : un tableau dynamique de nbetat*3 int */
static int nbetat; /* nombre d'états */
static int etat; /* état courant */
static int etatInit,etatFin; /* etat initial et final de l'afn */

int ** initAfn(int nbe){
  /* retourne un afn : tableau nbetat-1 lignes par 3 colonnes (nbetat>1)
     col 1 : symbole de trans
     si symbole=epsilon alors 1 ou 2 transitions par epsilon
     col 2 : etat destination
     col 3 : etat destination
  */
  int i;
  if (afn!=NULL){ /* pas la 1ere fois */
    for (i=0;i<nbetat-1;i++)
      free(afn[i]); /* désallouer */
    free(afn);
  }
  afn=(int**) malloc(((nbetat=nbe)-1)*sizeof(int));
  for (i=0;i<nbetat-1;i++)
    afn[i]=(int*) malloc(3*sizeof(int));
  etat=-1;
  return afn;
}

/* construction d'un afn non vide à partir d'un arbre binaire (exp reg) */
void construireAfn(Arbin r){
  int ig,fg,id,fd; /* états initiaux et finaux de g et d */
  if (islower(racine(r))||racine(r)=='@'){ /* symbole ou epsilon */
    etat++; /* nouvel état */
    afn[etat][0]=racine(r); /* affectation du symbole */
    etatInit=etat; /* état initial */
    etat++; /* nouvel état */
    afn[etatInit][1]=etat; /* état de transition */
```

```
    etatFin=etat; /* état final */
    afn[etatInit][2]=-1; /* pas de 2ème transition */
  }
  else if (racine(r)=='.'){ /* concat */
    construireAfn(sag(r)); /* récursif sur sag() */
    ig=etatInit;fg=etatFin; /* états initiaux et finaux de gauche */
    construireAfn(sad(r)); /* récursif sur sad() */
    id=etatInit;fd=etatFin; /* états initiaux et finaux de gauche */
    afn[fg][0]='@'; /* symbole : epsilon */
    afn[fg][1]=id; /* transition epsilon vers init de droite */
    afn[fg][2]=-1; /* pas de 2ème transition */
    etatInit=ig; /* l'état final reste celui de droite */
  }
  else if (racine(r)=='|'){ /* union */
    construireAfn(sag(r)); /* récursif sur sag() */
    ig=etatInit;fg=etatFin; /* états initiaux et finaux de gauche */
    construireAfn(sad(r)); /* récursif sur sad() */
    id=etatInit;fd=etatFin; /* états initiaux et finaux de gauche */
    etat++; /* nouvel état initial */
    afn[etat][0]='@'; /* affectation du symbole epsilon */
    afn[etat][1]=ig; /* transition epsilon vers init de gauche */
    afn[etat][2]=id; /* transition epsilon vers init de droite */
    etatInit=etat; /* nouvel état initial */
    etat++; /* nouvel état final */
    afn[fg][0]='@'; /* symbole : epsilon */
    afn[fg][1]=etat; /* transition epsilon vers nouveau final */
    afn[fg][2]=-1; /* pas de 2ème transition */
    afn[fd][0]='@'; /* symbole : epsilon */
    afn[fd][1]=etat; /* transition epsilon vers nouveau final */
    afn[fd][2]=-1; /* pas de 2ème transition */
    etatFin=etat; /* état final */
  }
  else if (racine(r)=='*'){ /* étoile */
    construireAfn(sag(r)); /* récursif sur sag() */
    ig=etatInit;fg=etatFin; /* états initiaux et finaux de gauche */
    etat++; /* nouvel état initial */
    afn[etat][0]='@'; /* affectation du symbole epsilon */
    afn[etat][1]=ig; /* transition epsilon vers init de gauche */
    etatInit=etat; /* nouvel état initial */
    etat++; /* nouvel état final */
    afn[etatInit][2]=etat; /* transition epsilon vers nouveau final */
    afn[fg][0]='@'; /* symbole : epsilon */
    afn[fg][1]=etat; /* transition epsilon vers nouveau final */
    afn[fg][2]=ig; /* retour vers l'état initial pour bouclage */
    etatFin=etat; /* état final */
  }
}
void afficherAfn(){
  int etat;
  printf("\nL'état initial est l'état %4d et l'état final est %4d\n",etatInit,etatFin);
  for(etat=0;etat<nbetat-1;etat++){ /* le dernier état est final ! */
      printf("transition : %4d %c %4d\n",etat,afn[etat][0],afn[etat][1]);
    if (afn[etat][2]!=-1) /* 2 transitions */
      printf("transition : %4d %c %4d\n",etat,afn[etat][0],afn[etat][2]);
    }
}
```

Solution 12 Calculette logique d'ordre 0 :

source lex `/* logic.l */`

```
var             ([a-z])
%{
#include "y.tab.h"      /* JETONS crees par yacc et definition de yylval */
#include <stdio.h>      /* pour printf */
%}
%%
[ \t]+                  {/* filtrer les blancs */}
0|false|faux            { yylval=0; return BOOL; }
1|true|vrai             { yylval=1; return BOOL; }
{var}                   { yylval=yytext[0]-'a'; return VAR;}
&                       { return AND; }
\|                      { return OR; }
\^                      { return XOR; }
->                      { return IMP; }
==                      { return EQ; }
!                       { return NOT; }
exit|quit               { return (QUIT); }
aide|help|\?            { return (HELP); }
.                       { return yytext[0]; /* indispensable ! */}
[\n]                    { return yytext[0]; /* indispensable egalement ! */}
%%
int yywrap(){return 1;} /* pas de continuation sur un autre fichier */
```

source yacc

```
                                /*logic.y  */
%{
 char * invite="\nVeuillez saisir une expression logique suivie de <Entrée> S.V.P.\n";
 int var[26];                    /* les 26 variables a-z */
 int yylex();
 void yyerror(char *);
%}
                                /* YYSTYPE est un entier par défaut */
%token BOOL VAR AND OR NOT IMP EQ XOR QUIT HELP /* definition des jetons */

%left OR IMP EQ XOR             /* traitement des priorites des operateurs */
%left AND
%right NOT

%%

liste   :       /* chaine vide sur fin de fichier Ctrl-D */
        |       liste ligne {printf(invite);}
        ;
ligne   :       '\n'            /* ligne vide : expression vide */
        |       error '\n'      {yyerrok; /* après la fin de ligne */}
        |       expr '\n'
        {
          printf("Résultat : %i\n",$1); /* une valeur entière (bool) */
        }
        |       VAR '=' expr '\n'
        {
          var[$1]=$3;
        }
        |       QUIT '\n'       {return 0; /* fin de yyparse */}
        |       HELP '\n'       {
        printf("        Aide de la calculette logique\n");
        printf("        ============================\n");
        printf("Taper une expression constituée de booléens, d'opérations,\n");
        printf("de variables (de a à z), de parenthèses puis taper <Entrée> \n");
        printf("Ou taper une commande suivie de <Entrée>\n\n");
        printf("Syntaxe des booléens : 0 ou false ou faux  \n");
```

```
            printf(" ou 1 ou true ou vrai \n");
            printf("Opérations infixes : & (et), | (ou), -> (implication),\n");
            printf("                     == (équivalence), ^ (ou exclusif)\n");
            printf("Opérations préfixes : ! (négation)\n");
            printf("Commandes : exit ou quit pour quitter la calculette\n");
            printf("            aide ou help ou \? pour afficher cette aide\n");
            printf("            variable = expression\n");
            }
            ;
    expr    :       '(' expr ')'    {$$ = $2;}
            |       expr AND expr   {$$ = $1 && $3;}
            |       expr OR expr    {$$ = $1 || $3;}
            |       expr IMP expr   {$$ = !$1 || $3;}
            |       expr EQ expr    {$$ = $1 == $3;}
            |       expr XOR expr   {$$ = $1 != $3;}
            |       NOT expr        {$$ = !$2;}
            |       BOOL            {$$ = $1;}
            |       VAR             {$$ = var[$1];}
            ;
    %%
    void yyerror(char *s) {fprintf(stderr,"%s\n",s);}
    int main(){/*yydebug=1;*/printf(invite);return yyparse();}
```

makefile

```
logic : logic.y logic.l
            # d'abord yacc pour y.tab.h puis lex puis gcc
            @echo debut $(YACC)-compil : logic.y
            $(YACC) $(YACCFLAGS) logic.y
            @echo debut $(LEX)-compil : logic.l
            $(LEX) logic.l
            @echo debut compil c avec edition de liens
            $(CC) -o logic y.tab.c lex.yy.c
            @# attention l'option -g (debug) fait planter
            @echo fin compil : vous pouvez executer logic
```

Solution 13 (cf exercice 13)

1. Collection canonique (AFD) des ensemble d'items LR(0) de la grammaire augmentée $(S \to E)$ associée à G_{ETF} :
   $I_0 = Fermeture(\{S \to .E\}) = \{S \to .E, E \to .E + T, E \to .T, T \to .T * F, T \to .F, F \to .(E), F \to .d\}$
   $T = \{(I_0, E, I_1)\}$
   $I_1 = Fermeture(\{E \to E. + T\}) = \{E \to E. + T, S \to E.\}$
   $T+ = \{(I_0, T, I_2)\}$
   $I_2 = Fermeture(\{E \to T.\}) = \{E \to T., T \to T. * F\}$
   $T+ = \{(I_0, F, I_3)\}$
   $I_3 = Fermeture(\{T \to F.\}) = \{T \to F.\}$
   $T+ = \{(I_0, (, I_4)\}$
   $I_4 = Fermeture(\{F \to (.E)\}) = \{F \to (.E), E \to .E + T, E \to .T, T \to .T * F, T \to .F, F \to .(E), F \to .d\}$
   $T+ = \{(I_4, (, I_4), (I_4, d, I_5), (I_4, T, I_2), (I_0, d, I_5)\}$
   $I_5 = Fermeture(\{F \to d.\}) = \{F \to d.\}$
   $T+ = \{(I_1, +, I_6)\}$
   $I_6 = Fermeture(\{E \to E + .T\}) = \{E \to E + .T, T \to .T * F, T \to .F, F \to .(E), F \to .d\}$
   $T+ = \{(I_6, F, I_3), (I_6, (, I_4), (I_6, d, I_5), (I_2, *, I_7)\}$
   $I_7 = Fermeture(\{T \to T * .F\}) = \{T \to T * .F, F \to .(E), F \to .d\}$
   $T+ = \{(I_7, (, I_4), (I_7, d, I_5), (I_4, E, I_8)\}$
   $I_8 = Fermeture(\{F \to (E.), E \to E. + T\}) = \{F \to (E.), E \to E. + T\}$
   $T+ = \{(I_8, +, I_6), (I_6, T, I_9)\}$
   $I_9 = Fermeture(\{E \to E + T., T \to T. * F\}) = \{E \to E + T., T \to T. * F\}$
   $T+ = \{(I_9, *, I_7), (I_7, F, I_{10})\}$
   $I_{10} = Fermeture(T \to T * F.\}) = \{T \to T * F.\}$
   $T+ = \{(I_8,), I_{11})\}$
   $I_{11} = Fermeture(F \to (E).\}) = \{F \to (E).\}$

2. Tables d'analyse Action et Successeur après avoir calculé les TabSuivants des non terminaux : $TabSuivants(E) = \{+, \$\}; TabSuivants(T) = \{*, +, \$\}; TabSuivants(F) = \{*, +, \$\};$

	Action						Succ.		
	d	+	*	(	)	$	E	T	F
0	S5			S4			1	2	3
1		S6				Accepter			
2		R($E \to T$)	S7		R($E \to T$)	R($E \to T$)			
3		R($T \to F$)	R($T \to F$)		R($T \to F$)	R($T \to F$)			
4	S5			S4			8	2	3
5		R($F \to d$)	R($F \to d$)		R($F \to d$)	R($F \to d$)			
6	S5			S4				9	3
7	S5			S4					10
8		S6			S11				
9		R($E \to E + T$)	S7		R($E \to E + T$)	R($E \to E + T$)			
10		R($T \to T * F$)	R($T \to T * F$)		R($T \to T * F$)	R($T \to T * F$)			
11		R($F \to (E)$)	R($F \to (E)$)		R($F \to (E)$)	R($F \to (E)$)			

3. Analyse de l'expression (5+3)*2$.

Pile	Flot d'entrée	Action
$0	(5+3)*2$	S4
$0(4	5+3)*2$	S5
$0(455	+3)*2$	R($F \to d$)
$0(4F3	+3)*2$	R($T \to F$)
$0(4T2	+3)*2$	R($E \to T$)
$0(4E8	+3)*2$	S6
$0(4E8+6	3)*2$	S5
$0(4E8+635	)*2$	R($F \to d$)
$0(4E8+6F3	)*2$	R($T \to F$)
$0(4E8+6T9	)*2$	R($E \to E + T$)
$0(4E8	)*2$	S11
$0(4E8)11	*2$	R($F \to (E)$)
$0F3	*2$	R($T \to F$)
$0T2	*2$	S7
$0T2*7	2$	S5
$0T2*725	$	R($F \to d$)
$0T2*7F10	$	R($T \to T * F$)
$0T2	$	R($E \to T$)
$0E1	$	Accepter