Travaux Dirigés et Pratiques d'Analyse Syntaxique et interprétation HAI601I

Michel Meynard

19 décembre 2023

Préambule

Les documents nécessaires, cours, énoncés des TDs et TPs, fichiers utiles pour TPs, sont disponibles sur Moodle : `https://moodle.umontpellier.fr/course/view.php?id=5914`.

1 Analyse lexicale

TD/TP 1

Exercice 1 (TD Théorie des langages)

1. Comment peut-on caractériser un langage rationnel (régulier) ?
2. Les langages de programmation (C, Java, Python) sont-ils réguliers ? Pourquoi ?
3. Comment peut-on caractériser un langage algébrique ?
4. Soit l'extrait suivant de grammaire EBNF de requête de consultation SQL :

```
SELECT
  [ALL | DISTINCT | DISTINCTROW ]
  select_expr [, select_expr] ...
  [into_option]
  [FROM table_references
    [PARTITION partition_list]]
  [WHERE where_condition]
  [GROUP BY {col_name | expr | position}
    [ASC | DESC], ... [WITH ROLLUP]]
  [HAVING where_condition]
  [ORDER BY {col_name | expr | position}
    [ASC | DESC], ...]
  [LIMIT {[offset,] row_count | row_count OFFSET offset}]
  ;
```

   Expliquer les différentes conventions de notation utilisées !

5. les requêtes suivantes sont-elles correctes ?

```
select 5 ;
SELECT distinct nom from etudiant
select ref from articles order by rayon asc, ref asc;
```

Exercice 2 (TD automate)

— Dessiner un Automate à états Finis Déterministe (AFD) distinct pour chacun des langages suivants :

1. Langage de dertains mots-clés du C : $L_{key} = \{if, then, else, throw\}$ (sensible à la casse).
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 bien 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).

Exercice 3 (TP1 AFD en C)

On utilisera dans ce TP, la fonction `analex` définie dans le cours et qui doit être récupérée sur l'ENT. Trois fichiers sont à télécharger :

— `afd.h` qui contient la définition d'un l'automate ;
— `analex.h` qui contient la définition de la fonction `analex()` ;
— `analex.c` qui contient un main appelant la fonction `analex()` itérativement ;

L'objectif de ce TP est d'implémenter l'unique AFD à jeton réalisé dans l'exercice précédent de façon à reconnaitre une partie des catégories lexicales du langage C. Pour cela, on modifiera la fonction `creerAfd` du fichier `afd.h`.

1. Lisez et testez les 3 fichiers téléchargés afin d'en comprendre le fonctionnement ;
2. On aura besoin de créer plusieurs transitions d'un état à un autre étiqueté par une classe de caractères comme les minuscules, ou les chiffres. Ecrire le corps de la fonction suivante :

```
/** construit un ensemble de transitions de ed à ef pour un intervale de char
 *@param ed l'état de départ
 *@param ef l'état final
 *@param cd char de début
 *@param cf char de fin
*/
void classe(int ed, int cd, int cf, int ef);
```

3. En utilisant, cette fonction `classe()`, modifiez la fonction `creerAfd()` dans le fichier `afd.h` ;
4. Dans `analex.c`, modifiez le `main()` afin qu'il n'affiche plus d'invite itérativement analex() et affichera une chaîne correspondant à l'entier retourné ainsi que le lexème, ceci jusqu'à la fin du fichier.

TD/TP 2

Exercice 4 (TD Expressions régulières)
Ecrire les expressions régulières correspondant aux langages réguliers suivants : $L_{key}, L_{c10}, L_{c8}, L_{c16}, L_{id}, L_f, L_{sep}$.

Exercice 5 (TP 2 Flex)
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 ASCII de ce caractère.

1. Ecrivez l'analyseur lexical `analflex.l`
2. Transformez-le en exécutable
3. Testez-le à la ligne de commande puis sur un programme C redirigé. Continuez à améliorer ce programme jusqu'à ce que tous les lexèmes soient reconnus.

Testez cet analyseur

Exercice 6 (TP2 supplément)
Améliorer l'analyseur lexical précédent à l'aide de flex en affectant à une variable globale `yylval` une valeur sémantique dépendant de la catégorie du lexème reconnu :

mots-clés rien ;

LITENT valeur entière longue (long int) ;

ID valeur chaîne de caractères ;

LITFLOT flottant double précision ;

TD/TP3

Exercice 7 (TD/TP : Flex nettoyage de texte)
Ecrire un source flex `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 8 (TD/TP : wc en flex)
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 9 (TD/TP : convertisseur MarkDown (.md) To HTML)
Markdown est un langage de balisage léger destiné à offrir une syntaxe facile à lire et à écrire utilisé par github et gilab (read.md). Un document balisé par Markdown peut être converti facilement en HTML. Pour réaliser des listes non numérotées (`<ul><li>`) :

- une ligne vide précède et suit la liste de 1er niveau
- chaque item est précédé d'une étoile (*) ou d'un tiret (-) en début de ligne
- on peut emboîter des listes en indentant avec au moins 2 espaces de plus que le niveau parent

L'exemple suivant

```
*1
  *1,1
    -1,1,1
      *1,1,1,1
      *1,1,1,2
  -1,2
*2
```

produira :

```
<ul><li>1
</li><ul><li>1,1
</li><ul><li>1,1,1
</li><ul><li>1,1,1,1
</li><li>1,1,1,2
</li></ul></ul><li>1,2
</li></ul><li>2
</li></ul>
```

Ecrire le source flex permettant d'effectuer la traduction.

TD/TP4

Exercice 10 (TD algorithmique des automates d'états finis)
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 11
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.

Exercice 12 (TP Amélioration des tps précédents)
Vous pouvez reprendre les exercices précédents afin d'améliorer la reconnaissance des jetons du langage C (analflexsem.l) ...

2 Analyse descendante récursive

TD/TP5

Exercice 13 (Analyse descendante récursive)
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 $L(G_{ENR})$:

```c
/** @file analdesc.c
 *@author Michel Meynard
 *@brief Analyse descendante récursive d'expression arithmétique
 *
 * 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>
#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 \n",numcar); exit(1);}

void E(void);void R(void);void T(void);void S(void);void F(void); /* déclars */

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

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

1. Dessiner l'arbre de dérivation associé au mot : `1+2+3*4`. Dessiner l'arbre des appels récursifs des fonctions E, R, T, S, F lorsqu'on reconnaît ce même mot. Que remarquez-vous ?
2. On souhaite implémenter l'évaluation de la valeur d'une expression arithmétique pendant sa vérification syntaxique. La multiplication sera prioritaire par rapport à l'addition et l'associativité des deux opérateurs sera à gauche (de gauche à droite). Annoter l'arbre obtenu à la question précédente pour indiquer où sont effectuées les 3 opérations et comment les fonctions se transmettent leurs résultats.(TD)
3. 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.
4. 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.
5. 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` :

```c
/** @file arbin.h
 * @brief en-tête définissant les structures, types, fonctions sur les arbres
 * binaires d'entiers.
 * @author Meynard Michel
*/
#ifndef _ARBINH
#define _ARBINH

#ifndef NULL
#define NULL 0
#endif

/*----------------------- Types Unions Structures ----------------------------*/
/** type pointeur sur la racine de l'arbre binaire d'entiers */
typedef struct Noeudbin * Arbin;

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

/** macro fonction créant un Arbin vide (retourne pointeur nul) */
#define ab_creer() (NULL)

/** macro fonction retournant le sous-arbre gauche d'un Arbin
 *@param a un Arbin
 *@return le sous-arbre gauche de a
*/
#define ab_sag(a) ((a)->fg)

/** macro fonction retournant le sous-arbre droit d'un Arbin
 *@param a un Arbin
 *@return le sous-arbre droit de a
```

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

/** macro fonction  testant si un Arbin est vide
 *@param  a un Arbin
 *@return vrai si a est vide, faux sinon
*/
#define ab_vide(a) (a==NULL)

/** macro fonction retournant la racine d'un Arbin
 *@param a un Arbin
 *@return l'entier situé à la racine
*/
#define ab_racine(a) ((a)->val)

/** @remark Ces 5 pseudo-fonctions (macros) sont definies quel
 * que soit le type de l'Arbin (entier, car, arbre,...)
*/

/** fonction remplaçant le sag(pere) par filsg et vidant l'ancien sag. Attention,
 * opération MODIFIANTE !
 *@param pere un Arbin
 *@param filsg l'Arbin qu'il faut greffer à la place du sag actuel de pere
 */
void ab_greffergauche(Arbin pere, Arbin filsg);

/** fonction remplaçant le sad(pere) par filsd et vidant l'ancien sag. Attention,
 * opération MODIFIANTE !
 *@param pere un Arbin
 *@param filsd l'Arbin qu'il faut greffer à la place du sad actuel de pere
 */
void ab_grefferdroite(Arbin pere, Arbin filsd);

/** fonction construisant un nouvel Arbin à partir d'une valeur entière qui
 * deviendra la racine et de deux sous arbres.
 *@param rac l'entier racine
 *@param g un Arbin qui devient sag
 *@param d un Arbin qui devient sad
 *@return l'Arbin construit
 */
Arbin ab_construire(int rac, Arbin g, Arbin d);


/** fonction copiant (clone) la structure d'un Arbin
 *@param a un Arbin
 *@return l'Arbin copié
 */
Arbin ab_copier(Arbin a);

/** fonction vidant un Arbin (désallocation). Attention, opération MODIFIANTE !
 *@param pa un pointeur sur Arbin
 */
void ab_vider(Arbin a);

/** fonction affichant un Arbin de manière indentée. Attention, racine(a) est
 * affichée comme un char
 *@param a l'Arbin à afficher
 */
void ab_afficher(Arbin a);


#endif /* _ARBINH */
```

TD/TP6

Exercice 14 (Calculette CLI)
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.

TD/TP 7

Exercice 15 (TD et TP Langage des expressions régulières)
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.

Exercice 16 (TP supplémentaire : minimisation d'AFD)
Développez le programme de minimisation d'un AFD par partitionnements successifs. Un exemple d'exécution suit :

```
$ expreg
Veuillez saisir une expression régulière suivie de <Entrée> S.V.P.
ab|ac|ab*

Mot reconnu dont l'arbre abstrait est :
|
  |
    .
      a
      b
    .
      a
      c
  .
    a
    *
      b

Construction de l'AFN de 18 états ...

AFDM : Afd : l'état initial est 2
Etat : 0 FINAL : --b--> 3, --c--> 1,
Etat : 1 FINAL :
Etat : 2 --a--> 0,
Etat : 3 FINAL : --b--> 3,
```

3 Analyse ascendante LR

TD/TP 8

Exercice 17 (TD/TP Bison)
On reprend l'exercice 15 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 prenant le source bison suivante (attention, à la concaténation qui est implicite !) :

```
%left '|'
%left CONCAT
%left '*'
%%
expr :  '(' expr ')'    {$$ = $2;}
|       expr expr  %prec CONCAT { $$= ab_construire('.',$1,$2);}
|       expr '|' expr   {$$= ab_construire('|',$1,$3);}
|       expr '*'        {$$= ab_construire('*',$1,ab_creer());}
|       SYMBOLE    {$$=ab_construire(yylval.i, ab_creer(), ab_creer());}
;
```

   La compilation bison avertit : "4 conflits par décalage/réduction" ! Examinez le fichier y.output afin de résoudre ce problème !
3. En utilisant la grammaire "intuitive" précédente et les règles de priorités et associativités trouvées à la question précédente, écrire un source bison traduisant une expression régulière en son arbre abstrait.

TD/TP 9 bison avec flex

Exercice 18 (Calculette logique)
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.

TD/TP 10

Exercice 19 (Collection canonique SLR)
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$.

Exercice 20 (TD collection canonique grammaire intuitive des exp. rég.)
Soit la grammaire $G_{regexp} = (\{S, |, *, (,)\}, \{e\}, R, e)$ avec les règles de R suivantes (Attention, le symbole | faisant partie du vocabulaire terminal, on utilisera + pour représenter l'alternative) :

$$e \rightarrow e|e + ee + e * +(e) + S$$

Le jeton S (Symbole) représente une lettre minuscule. On peut, grâce à cette grammaire ambigüe, générer des exp.rég. telles que : `ab|c*, (ab)*, (a(a(ab)*b)*)*`.

1. Dessinez la collection canonique de cette grammaire puis calculez les premiers et les suivants de e
2. Calculez la table d'analyse SLR de cette grammaire
3. Discutez des conflits et résolvez-les avec Bison afin que les opéraeurs soient associatifs à gauche et que les priorités ascendantes soient : |, CONCAT, *
4. Analysez le texte d'entrée $a|cd*$

Exercice 21 (TP supplémentaire)
En utilisant les fichiers fournis sur Moodle, implémenter le sprint 2 de l'interpréteur C permettant l'exécution d'une séquence de déclarations et d'instructions.