Travaux Dirigés et Pratiques d'interprétation et compilation HLIN604

Michel Meynard

14 janvier 2016

1 Analyse lexicale

TD/TP 1

Exercice 1 (TD1) — Dessiner un Automate à états Finis Déterministe (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).

Exercice 2 (TP1) On utilisera dans ce TP, la fonction `analex` définie dans le cours et qui doit être récupérée sur le site web `http://www2.lirmm.fr/~meynard/Ens2/article.php3?id_article=62`. 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 3 (TD2) Ecrire les expressions régulières correspondant aux langages réguliers suivants :

$$L_{key}, L_{c10}, L_{c8}, L_{c16}, L_{id}, L_f, L_{sep}$$

Exercice 4 (TP2) 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 5 (TP2) 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/TP 3

Exercice 6 (TD/TP) 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 7 (TD/TP) 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 8 (TD) 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 9 (TD) 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

TD/TP 4 et 5

Exercice 10 (TD/TP) 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 $L(G_{ENR})$:

```
/** @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'évalation 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.
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.

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. 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 :

```
/** @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 * pa);

/** 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 */
```

Exercice 11 (TD/TP 5) 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 12 (TD/TP) 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

TD/TP 8 et 9

Exercice 13 (TD/TP) On reprend l'exercice 12 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 14 (TD/TP) 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 15 (TD) 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 : pasada$_{\mathrm{C++}}$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 16 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 17 Ecrire un analyseur lex/yacc qui réalise uniquement la vérification syntaxique sans génération de code.

Exercice 18 Ecrire un compilateur en utilisant les structures de données adéquates : arbres abstraits, table des symboles, ...

Exercice 19 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 20 Améliorer le langage pasada en ajoutant la notion de structure C.