Théorie des Langages et Compilation
Séances de Travaux-Pratiques

Ph. Langevin

Année Universitaire 2015-16

Résumé

L'objectif de ces 6 séances de travaux-pratiques est de se familiariser avec quelques uns des outils, concepts et objet de la théorie des langages et de la compilation par le biais de : `gcc`, `regex`, `dot`, `grep`, `sed`, `awk`, `flex`, `bison`. Les séances devront faire l'objet d'une préparation à la maison sur la base d'indications données en travaux-dirigés. Idéalement, au terme de ces séances de travaux-pratiques, l'étudiant aura implanter un outil pour manipuler les automates.

dernières modifications :

```
   bison.tex 2016-03-16 11:31:16.208598905 +0100
    auto.tex 2016-03-02 16:01:24.524612678 +0100
 symbole.tex 2016-02-24 14:33:03.704833720 +0100
    eval.tex 2016-01-26 16:14:25.430313693 +0100
compil-tps.tex 2016-01-21 20:09:56.164020192 +0100
flexyacc.tex 2016-01-19 15:53:47.609118075 +0100
 projets.tex 2015-02-11 14:11:51.719712004 +0100
     pgn.tex 2015-01-30 16:13:10.721811790 +0100
   regex.tex 2015-01-29 17:53:27.148594873 +0100
     all.tex 2015-01-22 11:28:36.936479525 +0100
```

Table des matières

1 Expression rationnelle

1.1 Filtres

Aprés avoir déterminé l'origine des noms des filtres `grep`, `sed` et `awk`, vous réaliserez les opérations suivantes sur une source texte avec les filtres usuels :

1. extraire les adresses IPV4 ;
2. signaler les lignes qui apparaissent deux fois ;
3. mettre en majuscule les lettres qui suivent des espaces précédés d'un point ;
4. mettre en minuscule les identificateurs ;
5. supprimer les lignes 10,11 et 12.
6. inverser les nombres de 2 chiffres ;
7. extraire les lignes entre deux motifs ;
8. prénuméroter les lignes contenant un motif donné ; sans changer les autres lignes ;
9. postnuméroter les lignes correspondant à un motif donné ;
10. remplacer les chaines correspondant à un motif donné par son numéro d'ordre ;
11. signaler les lignes miroirs ;
12. mettre en forme une C-source.

Une page utile :

http://www.corporesano.org/doc-site/grepawksed.html

1.2 `regex`

Utiliser `regex` pour construire une commande en langage C qui met en évidence les correspondances d'un motif dans un fichier texte.

```
[pl]$ grep 'foo[^fb]*bar'       /tmp/file.txt
foo x bar y foo z foo bar
foo x bar y foo z foo bar
```

```
[pl]$ ./mygrep 'foo[^fb]*bar' /tmp/file/txt

foo x bar y foo z foo bar
foo x bar y foo z foo bar
```

Vous partirez du programme `regex.c` :

```c
#include <regex.h>
#include <stdlib.h>
#include <stdio.h>
int main(int argc, char *argv[])
{ regex_t regex;
  if (regcomp(&regex, argv[1], REG_EXTENDED) != 0)
    return 1;
  if ( 0 == regexec(&regex, argv[2], 0, NULL, 0) )
        printf ("matching!\n");
    else
        printf ("no match!\n");
  regfree (&regex);
  return (0);
}
```

regex.c

2 Descente récursive naive

Il s'agit d'implanter la fonction `int eval( char*e)` décrite dans les travaux-dirigés qui utilise une descente récursive naïve pour évaluer une expression arithmétique représentée par la chaine de caractère e.

```
./eval.exe '13'

eval = 13
./eval.exe '((13))'

eval = 13
./eval.exe '6-6-6'

eval = -6
./eval.exe '6-(6-6)'

eval = 6
```

```
./eval.exe '((1+2) * (3-5)+ 8)*(13+21)'

eval = 68
./eval.exe '1+2;3*4;5*6'
 [3] [12]
eval = 30
./eval.exe 'a=3-1;b=3+1;a*b'
 [2] [4]
eval = 8
```

2.1 Mise en oeuvre

Pour mémoire, nous avons utilisé les fonctions suivantes :

- `int asso( int op )` : retourne 0 si l'opérateur est associatif à gauche, et 1 sinon.
- `int priorite( int op )` : retourne la priorité de l'opérateur.
- `int prec( int op , int oq)` : retourne 1 si la priorité du premier argument est inférieur à celle du second, 0 sinon.
- `int value( char *e )` : retourne la valeur représentée par e, un nombre, plus loin une lettre.
- `int getpos(char *e)` : retourne la position de l'opérateur de moindre priorité, -1 si pas d'opérateur.
- `void clean(char *e)` : supprime les parenthèses bordantes.

```
1  int eval( char* s)
2  {
3  int pos,  op;
4  char *g, *d;
5  pos = getpos( s );
6  if ( pos < 0 )
7          return value( s );
8  op = s[ pos ];
9  s[ pos ] = 0;
10 g = s;
11 d = & s[pos+1] ;
12 switch( op ) {
13         case '+': return eval(g)+eval(d);
14         case '*' : return eval(g)*eval(d);
15 }
16 return -1;
17 }
18 int main( int argc, char* argv[] )
19 {
20 printf ( "\neval = %d\n", eval( argv[1]) );
21 return 0;
22 }
```

2.2 Exercice

1. Télécharger expr.c, faire un `makefile`, compiler, vérifier la bonne gestion des expressions des opérateurs '+' et '*'.
2. Il y a un bug concernant la gestion des parenthèses, lequel ? Fixer le problème.
3. Modifier le code pour gérer les soustractions et les divisions.
4. Introduire un séparateur d'instructions qui trace les calculs, le résultat doit correspondre au dernier calcul effectué.

```
:> ./eval.exe '1+2;3*4;5*6'
 [3] [12]
eval=9
> eval/eval.exe '(1;2;3)*5'
 [1] [2]
eval = 15
```

5. Modifier le code pour gérer 26 cases mémoires représentées par les lettres de l'alphabet. Bien entendu, il faut introduire l'opérateur affectation représenté par le symbole '=' avec une sémantique analogue à celle du langage C.

```
:> ./eval.exe 'a=3-1;b=3+1;a*b'
 [2] [4]
eval = 8
:> ./eval.exe  'a = b = 23; a+b'
 [23]
eval = 46
```

6. Modifier le code pour gérer l'opérateur unaire factoriel.
7. Comment ajouter l'opérateur d'exponentiation qui sera représenté par le symbole '**' ?
8. Modifier le code pour gérer l'opposé représenté le symbole '-'.
9. Essayer de gérer quelques erreurs synxatiques !

A ce stade, nous devons avoir compris les limites de la descente récursive naïve ! Reprendre les mêmes question avec la calculette à pile pile.c qui a l'avantage de séparer les phases lexicales et syntaxiques.

3 Analyse lexicale

3.1 Introduction a `flex`

La structure d'un programme `flex` est similaire à celle d'une source `bison`. Elle est en effet découpée en 4 zones séparées par les balises %{ , %}, %%, %% : prologue, définition, règles et épilogue. La compilation d'une source `flex` produit une fonction `int yylex(void)` dans un fichier :

Prologue
Automate
Epilogue

L'analyseur lexical de l'exemple ci-dessous recherche le mot le plus long tout en calculant la somme des entiers rencontrés dans le fichier. Il utilise deux variables prédéfinies dans `flex` `yytext` et `yyleng` qui représentent respectivement le motif et la longueur du motif reconnu par l'analyseur.

```
%{
// Prologue en langage C
    #include <stdio.h>
    int total = 0;
    int score = 0;

%}

// Definitions lex

LETTRE    [a-zA-Z]
CHIFFRE   [0-9]
NOMBRE    {CHIFFRE}+

%%
// regles et action

{NOMBRE}   total+= atoi( yytext );
{LETTRE}+ if (yyleng > score){
                score = yyleng;
               ECHO;
              }
. printf("\nNi mot, \
        ni nombre :%s", yytext);
%%

// Epilogue en langage C

int main( void )
{
 yylex() ;
 printf("\nSomme des nombres %d\
        \nbye...\n", total);
 return 0;
}
```

Un appel de `yylex()` déclenche une analyse lexicale du flux `yyin`. Au cours traitement, l'analyseur tente de satisfaire la première règle, puis la seconde etc... La plus longue correspondance est chargée dans la variable `yytext`, sa longueur dans `yyleng`.

Le nom d'une source `flex` termine souvent par le suffixe `.l` . Éditez un fichier scan.l. Méfiez vous des copier/coller d'un fichier `pdf` vers un fichier texte ! Compilez. Si tout se passe bien, `flex` construit une source `lex.yy.c` qu'il faut compiler en incluant la bibliothèque `flex` :

```
$> gcc -Wall lex.yy.c -o scan.exe -lfl
```

Testez l'exécutable `scan.exe` en lançant quelques commandes comme `./scan.exe < scan.l`, ou encore `cat lex.yy.c | ./scan.exe`, voire `./scan.exe` ! Dans ce dernier cas, CTRL-d ferme le fichier `stdin`.

1. Faire un fichier `makefile`.
2. Modifiez la fonction `main()` pour affecter la variable `yyin`. Vous pouvez jeter un coup d'oeil `man flex` ou `info flex`.
3. Modifiez `scan.l` pour préciser la ligne contenant le mot le plus long.
4. Modifiez `scan.l` pour préciser l'adresse (ligne, colonne) du mot le plus long.

3.2 Gestion de symboles

Il s'agit d'utiliser `flex` pour écrire une commande `scan.exe` qui dresse un rapport sur les mots les plus fréquents d'un fichier texte. Un peu comme le fait le script word.sh basé sur la commande `awk` :

```
#!/bin/bash
function help {
echo "usage $0 : [-n number ]" file
exit
}
qte=3
while getopts "hn:" opt; do
  case $opt in
    n) qte=$OPTARG;;
   \?) echo "option invalide $OPTARG";exit 1 ;;
    :) echo "argument requis $OPTARG";exit 2 ;;
    h) help; exit 1 ;;
  esac
done

shift $((OPTIND-1))
echo distribution des mots de $1
if [ -n "$1" ]; then
awk -F'[^a-zA-Z]*' \
'{for(i=1;i<=(NF);i++){t[$i]++}}END{for(x in t){print t[x],x}}' $1 \
 | sort -nr | head -$qte
else help
fi
```

word.sh

Le programme doit être écrit en langage `flex`, et vous utiliserez une gestion de symboles basée sur une liste chainée

```
typedef struct _list_ {
    char *key;
    int    count;
    struct _list_ * next;
} enrlist , *list ;
```

Les mots trouvés au cours de l'analyse lexicale sont recherchés dans une liste chainée globale pour maintenir à jour le nombre d'occurrences des mots rencontrés le fichier.

```
  int inserer ( char *k, list  *ptr );
```

La fonction renvoie *faux* si la clé k est dans la liste globale sinon elle retourne *vrai* aprés avoir inséré la nouvelle clé. Dans les deux cas, *ptr* donne accès aux attributs correspondants.

```
1 if ( inserer ( "toto", & ptr ) )
2     printf ( "%s : nouveau\n", ptr->key );
3 printf ( "%s %d occurrences\n", ptr->key, ptr->count );
4 }
```

1. Ecrire `int inserer(char* k, list* ptr)`
2. Ecrire une fonction `void plist( list l )`, pour afficher le contenu de la liste l.
3. Modifier l'analyseur lexical pour obtenir la distribution des mots.

La gestion de symboles basée sur liste chainée n'est pas performante, pour vous en rendre compte, appliquez votre scanner à un gros fichier texte, un dictionnaire par exemple. Les mécanismes de dispersion (hachage) permettent d'accélérer le code d'un facteur arbitraire à condition d'avoir suffisamment de mémoire.

4 Automate

Au cours des séances de cours et travaux dirigés, nous avons décrit quelques opérations fondamentales sur les automates : suppression des ϵ-transitions, déterminisation, minimisation. L'objectif de l'exercice est d'implanter les algorithmes de minimisation et de déterminisation. Comme je n'ecris pas de cours sur le sujet, je vous suggère les transparents de Sandrine Julia à Nice. Facile à trouver par un moteur de recherhe avec les clés :

```
filetype:pdf sandrine julia automate
```

4.1 représentation

Dans cette séance, on s'intéresse aux automates finis non déterministes avec un nombre d'états $n < 64$ sur l'alphabet des 36 caractères alphanumériques. Les états seront numérotés de 0 à $n-1$ un tel automate est représentable par la structure

```
1 typedef struct {
2         ullong etat;              // etats utiles
3         ullong initial , final ;
4         ullong delta[64][37];     // transition
5         char * ident[64];         // identificateur des etats
6 } afn;
```

Vous partirez des sources de ce répertoire http://langevin.univ-tln.fr/cours/COMPIL/tps/afnd, et vous implanterez les algorithmes dans des fichiers séparés.

4.2 entrée/sortie

Un fichier source pour représenter un automate fini non déterministe sera un fichier texte composé d'une succession de lignes au format : NB LETTRE NB, INIT NB (suite), FINAL NB (suite). Le caractère spécial '-' sera utilisé pour représenter les ϵ-transitions. Les tokens INIT et FINAL seront utilisés pour préciser la nature des états.

```
// exemple d'automate a trois etats
// avec une epsilon transition.
INIT 0
0 a 0
0 b 1
1 a 2
1 b 0
FINAL  2
2 a 1
2 b 2
2 - 0
```

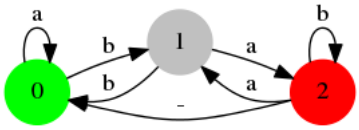
sample.afn

1. L'analyseur lexical `auto.l` permet de charger un automate à partir d'un fichier source.
2. Le `dotify.c` construit une représentation au format `dot`, voir la galerie du projet graphviz.

```
digraph  {
rankdir=LR;
0 [style=filled, shape=circle, color=green, label="0"]
0->0 [label="a"]
0->1 [label="b"]
1 [style=filled, shape=circle, color=gray, label="1"]
1->2 [label="a"]
1->0 [label="b"]
2 [style=filled, shape=circle, color=red, label="2"]
2->0 [label="-"]
2->1 [label="a"]
2->2 [label="b"]
}
```

sample.afn.dot

Observer les sources, le fichier `makefile` et les compilations qui permettent d'obtenir la représentation sagittale :



4.3 synchronisation

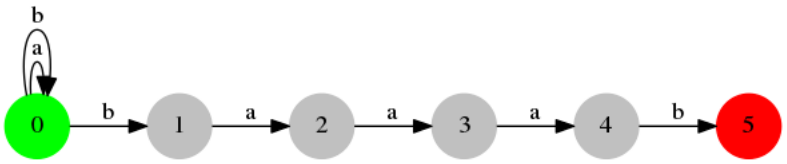
Le calcul d'ϵ-fermeture `ullong eps(ullong x, afnd A)` d'un ensemble d'états de l'automate A est implanté dans `epsilon.c`.

1. Implanter une procédure `void synchronisation( afnd *A)` qui supprime les ϵ-transitions de A.
2. Implanter le calcul des états accessibles et celui des états co-accessibles.

4.4 déterminisation

L'algorithme de déterminisation construit un automate déterministe complet en $O(2^n)$ étapes par un parcours en profondeur à partir de l'état initial.

1. Implanter l'algorithme de déterminisation qui produit une sortie au format `dot`.
2. Détermininiser l'automate :



4.5 minimisation

L'algorithme de Moore détermine les classes d'équivalence de Nérode par des raffinements successifs de la non-Nérode équivalence. Il permet de construire l'automate minimal en $O(An^2)$ étapes.

1. Ecrire l'algorithme de Minimisation de Moore.

5 Analyse syntaxique

5.1 calculette `bison`

On considère le langage des expressions arithmétiques de la forme :

$$A = 3 * 5;\ B = C = (A - 5) * 7 + 3;\ ?C; \tag{1}$$

c'est-à-dire une suite d'instructions séparées par le symbole " ;". Les espaces seront ignorés, les nombres réduits à un chiffre, et les identificateurs réduits à une lettre, une instruction d'affichage, une affectation, 4 opérateurs et des expressions parenthèsées.

1. Identifiez bien le rôle des différents lexèmes, les priorités et ordres d'associations des opérateurs.
2. Définir la syntaxe du langage avec la notation de Backus-Naur.

Pour construire un premier analyseur avec `bison`, on commence par quelque chose de trés élémentaire : le langage des expressions non parenthésées construites à partir de trois symboles terminaux NB, PLUS, FOIS, et un symbole non-terminal EXP et des règles :

EXP := NB | EXP PLUS EXP | EXP FOIS EXP

Pour construire un analyseur syntaxique avec la commande `bison` on édite un fichier de suffixe `.y`, disons le fichier `calc.y` incluant la définition d'un analyseur lexical, obligatoirement identifié par `int yylex(void)`, la description des lexèmes (tokens) et des règles. Les actions sémantiques écrites en langage C entre accolades calculent l'attribut du père, représenté par $$, en fonction des attributs des fils de gauche à droite représentés par $1, $2, ...

```
%{
    int yylval ;
%}

%token NB PLUS FOIS FIN

%start PROG

%%

PROG : EXP FIN { printf("%d", $1 ) ; return 1;}
EXP  : NB { $$ = $1 }
     | EXP PLUS EXP { $$ = $1 + $3 ;}
     | EXP FOIS EXP { $$ = $1 * $3 ;}
     ;
%%

int main( void )
{
     yyparse() ;
}

int yylex( )
{
  // a definir !
}

int yyerror(char *msg)
{
printf("\n%s\n", msg);
}
```

La fonction `main` du programme est réduit à sa plus simple expression : un appel de l'analyseur syntaxique `yyparse()` qui utilise implicitement la variable `yylval` et l'analyseur lexical `yylex()`, une fonction qui renvoie la valeur du lexème (token) courant dont l'attribut est transmis par la variable `yylval`. Les erreurs de syntaxes provoquent l'appel de la fonction `yyerror()`. Ci-dessous, un exemple d'analyseur lexical rudimentaire que vous ne manquerez pas d'améliorer !

```
1 int yylex()
2 {
3     int car;
4     car = getchar();
5     if (car == EOF)
6         return 0;
```

```
7       if (isdigit (car)) {
8           yylval = car - '0';
9           return NB;
10      }
11      switch (car) {
12      case '+':
13          return PLUS;
14      case '*':
15          return FOIS;
16      case '!':
17          return FIN;
18      }
19
20  }
```

5.2 Tests

Lancez la commande

```
bison -v calc.y
```

L'analyseur échoue :-<...

La présence d'un conflit "décalage/reduction" montre que la grammaire ne peut pas être traitée par `bison`. En effet, seules certaines grammaires LR sont admises par `bison`. Dans notre cas, il s'agit d'un problème d'ambiguité relatif aux caractères associatifs des opérateurs. Une déclaration plus précise permet de fixer les priorités :

```
%left PLUS
%left FOIS
```

L'ordre des déclarations indique que l'opérateur FOIS (associatif à gauche) est de priorité supérieure à l'opérateur PLUS (associatif à gauche).

Modifiez `calc.y` jusqu'à ce que l'exécution :

```
bison calc.y
```

se passe correctement. Vous devez obtenir une source C `calc.tab.c` qu'il faut compiler par

```
gcc -Wall calc.tab.c -o calc
```

pour obtenir un analyseur nommé `calc`, prévoir un fichier `Makefile` pour la suite.

```
calc.exe : calc.y
 bison  calc.y -o calc.c
 gcc -Wall -g  calc.c -o calc.exe
```

Testez l'analyseur `calc`, il est sensé évaluer une formule terminée par un point d'exclamation. Est-ce bien le cas ?

5.3 Améliorations

Il ne reste plus qu'à améliorer les analyseurs en incorporant pas à pas les ingrédients nécessaires de sorte à obtenir une calculette capable de manipuler des suites d'instructions un peu plus complexes à base de 26 registres mémoires A, B, ..., Z. Pour écrire des programmes comme (1).

1. Améliorez l'analyseur lexical pour filtrer les espaces et tabulations.
2. Modifiez la grammaire et `calc.y` pour gérer les autres opérations : division, soustraction.
3. Intégrez une fonction `int myexp( int x, int n)` pour gérer les exponentiations, l'opérateur sera représenté par deux étoiles.
4. Gérez les parenthèses.
5. Consultez le manuel pour gérer correctement les opposés (moins unitaire). Vous ferez le choix de `bc`.
6. Modifier `yylex()` pour manipuler des nombres de plusieurs chiffres : la fonction `ungetchar()` est particulièrement utile !
7. Ajouter des règles syntaxiques pour effectuer des calculs séparés par des " ; "
8. Ajoutez un token `MEM` dont les attributs possibles seront A, B, ...dans l'analyseur lexical. Incorporez les règles syntaxiques pour traiter les expressions contenant des variables.
9. Ajoutez un lexème `AFFECT`. Incorporer les règles syntaxiques pour reconnaître les affectations `MEM AFFECT EXP`.
10. Modifiez les actions sémantiques pour afficher la liste des règles utilisées.
11. Testez le mode DEBUG, de `bison`.
12. Utilisez le symbole prédéfini `error` pour faire de la récupération d'erreur.

6 Utilisation conjointe de `bison` et `flex`

Dans cette partie, il s'agit décrire un analyseur lexical en `flex` pour un analyseur syntaxique en `bison`. Nous reprenons l'exemple de la calculette en gérant les identificateurs de variables dans une table de symboles.

6.1 makefile

```
CFLAGS=-Wall -g
calc : calc.o scan.o
    gcc $(CFLAGS) calc.o scan.o  -o calc -lfl
```

```
calc.c  : calc.y
    bison  -d -ocalc.c calc.y

scan.c  : calc.c scan.l
    flex   scan.l -o scan.c

scan.o  : scan.c
    gcc $(CFLAGS) -c scan.c

calc.o  : calc.c
    gcc $(CFLAGS) -c calc.c

clean   :
   rm -f *.o
```

6.2 gestion des identificateurs

1. L'option -d de `bison` fournit un fichier de définitions utilisable par l'analyseur lexical.
2. Observer le fichier `calc.h` pour comprendre son rôle.
3. Prévoir un fichier de définition `scan.h` utilisable par les deux analyseurs.
4. Le typage des symboles ternimaux et non-terminaux se fait par une déclaration `%union`, pour la calculette,

```
1   %union {
2       PTR symb;
3       int  val
4   }
```

   Le fichier `scan.h` contenant la définition du symbole `PTR`.
5. La gestion des champs au niveau des symoles grammaticaux peut se faire explicitement sous la forme :

```
AFFECT : ID AFF EXP     { $<symb>1 -> valeur = $<val>3 }
```

   La déclaration `%type` permet de définir des types implicites :

```
%type <val>  EXP
%type <symb> ID

...

AFFECT : ID AFF EXP     { $1 -> valeur = $3 }

...
```

les actions sémantiques réfèrent par défaut au champ correspondant lors de l'utilisation de l'attribut du symbole EXP.