Examen de Compilation

Licence Informatique 3

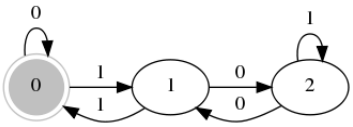
3 Mai 2013

Le sujet est composé de 4 exercices indépendants. Aucun document n'est autorisé. Durée de l'épreuve : 0x78 minutes. La note finale tiendra trés largement compte de la présentation.

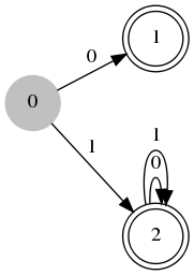
1 Automate

Dans l'ensemble $\{0,1\}^*$ un mot binaire $x_n x_{n-1} \dots x_1$ représente la valeur entière $\sum_{i=1}^{n} x_i 2^{i-1}$. Conformément à l'usage un nombre binaire est un mot commençant par la lettre 1 ou bien égal à 0. Contruire les automates pour reconnaitre :

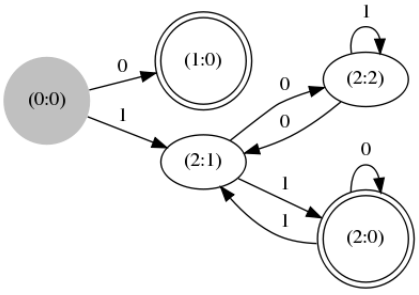
1. le langage des mots de valeur multiple de trois.



2. le langage des mots $0 + 1\{0,1\}^*$.



3. le langage des nombres multiple de 3.



2 Théorie des langages

Dans l'ensemble $\{a, b\}^*$, on condidère le langage X des mots contenant un nombre de a supérieur ou égal à celui de b.

1. Donner la définition de l'équivalence de Nérode entre deux états d'un automate.
   Si F désigne l'ensemble des états finaux d'un automate deux états accessibles x et y sont équivalents si et seulement
   $$\forall w \in \{a, b\}^*, \quad x.w \in F \Leftrightarrow y.w \in F$$
2. Montrer que X n'est pas régulier. Indication : étudier l'équivalence de Nérode entre les états q_i (i entier) obtenus par la lecture du mot a^i à partir de l'état initial.
   Avec ces notations, pour $i < j$, nous avons $q_j b^j \in F$ car $a^j b^j$ est un mot du langage, mais $q_i b^j \notin F$ car $a^i b^j$ contient plus de b que de a. Un hypothétique AFND aurait un nombre infini d'états : c'est absurde.

3 Ambiguité

On considère la grammaire

$$\begin{array}{lcl} \text{X} & \longrightarrow & \text{a X} \\ \text{X} & \longrightarrow & \text{a X b} \\ \text{X} & \longrightarrow & \epsilon \end{array}$$

1. Décrire le langage des mots reconnus.
   Le langage des mots de la forme $a^m b^n$ avec $m \geq n$.
2. Montrer que la grammaire est ambiguë.
   aab possède deux arbres syntaxiques.
3. Proposer une sémantique pour fixer l'ambiguité.
   Comme dans le cas du `si alors sinon`, il suffit de faire correspondre chaque b au a le plus proche sur la gauche.
4. En déduire une grammaire non ambiguë.
   $$\begin{array}{lcl} \text{X} & \longrightarrow & \text{O} \mid \text{F} \\ \text{O} & \longrightarrow & \text{a X} \\ \text{F} & \longrightarrow & \text{a F b} \\ \text{F} & \longrightarrow & \epsilon \end{array}$$

4 flex-bison

Le jeu d'instruction d'une machine similaire à MILXI est partiellement décrit ci-dessous. Un compilateur est implanté au travers des codes `scan.l` et `parse.y`, il produit du code d'assemblage à partir d'un langage évolué.

READ x	mem[x] := lecture	WRITE x	écrire mem[x]
LOAD x	ACC := mem[x]	STO x	mem[x] := ACC
ADD x	ACC += mem[x]	MUL x	ACC *=mem[x]
CONST x	ACC := x	STOP	arrêt

1. Préciser les phases de compilation effectivement présentes dans ces codes.
   Analyse lexicale, analyse syntaxique, gestion de symbole, production de code.

2. Donner un `makefile` pour compiler une commande `minias` à partir de ces sources.

```
all : minias
minias      : scan.o parse.o
        gcc scan.o parse.o -o minias -lfl

scan.o      : parse.c scan.l
        flex -o scan.c scan.l
        gcc -Wall -g -c scan.c

parse.c : parse.y
        bison -dv parse.y -o parse.c

parse.o : parse.c
        gcc -Wall -g -c parse.c

demo    : prog.as minias
        ./minias < prog.as

clean :
        rm -f *.o *.c *~
```

3. Quel sera le résultat de `./minias < prog.as` où `prog.as` est un fichier texte dont le contenu, écrit ici sur trois colonnes, est :

```
var x                   input x                 output z
var y                   input y
var z                   z := (x+y) * (x+y)
```

READ 0	STO 4
READ 1	LOAD 3
LOAD 0	MUL 4
ADD 1	STO 3
STO 3	LOAD 3
LOAD 0	STO 2
ADD 1	WRITE 2

4. Quelles modifications faut-il effectuer pour gérer une instruction d'arrêt STOP.
   Ligne 40 de `scan.l`, ajouter `STOP return STOP`. Dans le parseur ajouter STOP dans la déclaration des tokens. En ligne 45, ajouter la production : `STOP { printf("STOP") ; }`
5. Quelles modifications faut-il effectuer pour gérer les constantes.
   Ligne 44 de `scan.l`, ajouter `yylval=atoi(yytext) ;`. Dans le parseur ajouter CONST dans la déclération des tokens. En ligne 45, ajouter la production : `CONST NB { printf("CONST %d", $2) ; }`

scan.l

```
%{
#include "parse.h"

int value = 0;
typedef struct _symb_ {
  int    val;
  char* key;
  struct _symb_ * next;
} enrs, *symb;
symb ts   = NULL;
int insert( char * k )
  {
    symb x;
    x = ts;
    while ( x ) {
if ( strcmp( x->key, k ) == 0 )
     return x->val;
     x = x -> next;
    }
    x = malloc( sizeof( enrs) );
    x->key = strdup( k );
    x->next = ts;
    x->val = value;
    value++;
    ts = x;
    return x->val;
  }

%}

%%
":="          return AFF;
"+"           return ADD;
";"           return NL;
"*"           return MUL;
"("           return PG;
")"           return PD;
"var"         return VAR;
"input"       return IN;
"output"      return OUT;
[a-z]+     {  yylval = insert( yytext );
              return ID;
           }
[0-9]+        return NB;
\n            return NL;
.             ;
%%
```

parse.y

```
%{
#include <stdio.h>
int used[64] = {0};
int getreg( void ) {
  int r = 0;
  while ( used[r] ) r++;
  used[ r ] = 1;
  return r;
}
void ungetreg( int n ) {
 if ( used[n] == 1 )
        used[n] = 0;
}
extern int yylex( void );
int yyerror( char *msg );

%}

%token ID NB VAR NL AFF
%token IN OUT PG PD
%left  MUL
%left  ADD


%%

PROG  : DECL LIST
DECL  : VAR ID NL
   { used[$2] = 2;} DECL
       | /* empty */
       ;
LIST  : INSTR NL LIST
       | /* empty */
       ;
INSTR : ID AFF EXP  {
    printf("\nLOAD  %d", $3);
    printf("\nSTO   %d", $1);
    ungetreg( $3 );
     }
    | IN  ID   {
      printf("\nREAD  %d", $2);
     }
    | OUT ID   {
      printf("\nWRITE %d", $2);
     }
          ;
```

```
46 EXP : EXP ADD EXP {
47         printf("\nLOAD␣␣%d", $1);
48         printf("\nADD␣␣␣%d", $3);
49         ungetreg( $1 );
50         ungetreg( $3 );
51         $$ = getreg( );
52         printf("\nSTO␣␣␣%d", $$ );
53       }
54       | EXP MUL EXP {
55         printf("\nLOAD␣␣%d", $1);
56         printf("\nMUL␣␣␣%d", $3);
57         ungetreg( $1 );
58         ungetreg( $3 );
59         $$ = getreg( );
60         printf("\nSTO␣␣␣%d", $$ );
61       }
62       | PG EXP PD { $$ = $2;}
63       | ID { $$ = $1;}
64       ;
65
66 %%
67
68 int main( void )
69 {
70 yyparse();
71 return 0;
72 }
73
74 int yyerror(char *msg) {
75 printf("%s", msg);
76 return 0;
77 }
```