Exams Compila

Avec corrigés

Ce document contient les exams suivant :

Corrigé Rattrapage 2009 & 2012

Corrigé Session Normale 2010 & 2011 & 2013 + Corrigé Questions de QCM

Corrigé
Rattrapage 2009

Examen de rattrapage

Année Universitaire : 2008 - 2009
Filière : Ingénieur
Semestre : S3
Période : P2

Module : **M3.4** - Compilation
Elément de Module : **M3.4.1** - Compilation
Professeur : Karim BAÏNA

Date : 20/03/2009
Durée : 1H00

Nom :……
Prénom :……

Consignes aux élèves ingénieurs :
- *Aucun document n'est autorisé !!*
- *Le barème est donné seulement à titre indicatif !!*

Exercice I : Choix Exclusifs (10 pts)

Pour chaque concept/question, remplissez la case de la colonne des choix uniques correspondante par un choix qui soit le plus adéquat

Concept/Question	Choix unique	Choix possibles
(**1**) Toute grammaire peut être rendue LL(1)		**(a)** Vrai **(b)** Faux
(**2**) Flex peut totalement remplace Bison pour quelques langages particuliers		**(a)** Vrai **(b)** Faux
(**3**) Bison permet de programmer les grammaires attribuées avec héritage et synthèse d'attributs		**(a)** Vrai **(b)** Faux
(**4**) <**INST**> ::= IDF ":=" <**EXPR**> | IF '(' IDF '=' <**EXPR**> ')' THEN <**LISTE_INST**> ELSE <**LISTE_INST**> ENDIF | IF '(' IDF '=' <**EXPR**> ')' THEN <**LISTE_INST**> ENDIF | PRINT IDF ;		**(a)** est Ambiguë **(b)** n'est pas ambiguë
(**5**) <**ADDMIN**> ::= <**ADDMIN**> '+' IDF | <**ADDMIN**> '-' IDF | IDF		**(a)** est LL(1) **(b)** n'est pas LL(1)
(**6**) le 1-address code est choisi pour sa		**(a)**rapidité **(b)**taille du code **(c)**portabilité
(**7**) la fonction de hashage $h_1(s \in \Sigma^*) = \Sigma_{i=1..\|s\|}\ s_i$ répartit …… les identifiants s que la fonction $h_1(s \in \Sigma^*) = \Sigma_{i=1..\|s\|}\ s_i$ dans la table des symboles		**(a)** mieux **(b)** moins bien **(c)** similairement
(**8**) un automate NFA est analogue à une grammaire		**(a)** ambiguë **(b)** avec des règle à ε **(c)** non LL(1)
(**9**) Le langage $L = \{a^n b^n c^m d^m, n \geq 1,\ m \geq 1\} \cup \{a^n b^m c^m d^n, n \geq 1,\ m \geq 1\}$ est programmable en bison		**(a)** oui **(b)** non **(c)** avec des adaptations
(**10**) La grammaire <**INST1**> ::= IF '('<**EXPR**>')' THEN <**INST1**> ELSE <**INST1**> | IF '('<**EXPR**>**')' THEN** <**INST1**> et la grammaire <**INST2**> ::= IF '('<**EXPR**>')' THEN <**INST2**> ELSE <**INST**> ENDIF | IF '('<**EXPR**>')' THEN <**INST2**> ENDIF donne ….		**(a)** le même langage **(b)** différents langages

#define N 200
#define NBS 100
int NBVAR=0;
typedef enum {false=0, true=1} boolean;
typedef struct {
char *name; int nbdecl; int order;
} varvalueType;
varvalueType TS[NBS];
 	
Compléter La fonction de recherche d'un identifiant dans la tableau des symboles
boolean inTS(char * varname, int * rangvar){
int i =0;
(**11**) while ((i < …….)
&& (strcmp(TS[i].name, varname) != 0)) i++;
		(a) sizeof(TS) **(b)** N **(c)** NBS **(d)** NBVAR

(**12**) if (i == …..) return false;		**(a)** sizeof(TS) **(b)** N **(c)** NBS **(d)** NBVAR
(**13**) else { ……. = i; return true;} }		**(a)** TS[i].order **(b)** rangvar **(c)** &rangvar **(d)** *rangvar

(**14**) Que réalise la fonction process sur les mots du langage des alpha-numériques
typedef char * langage1 ;
void process(langage1 s){
int c, i, j;
for (i = 0, j = strlen(s)-1; i < j; i++, j--) {
c = s[i]; s[i] = s[j]; s[j] = c;
}
}
		(a) décale les mots à droite
		
		(b) décale les mots à gauche
		
		(c) renverse les mots
		
		(d) tranforme le mot en son palyndrôme

(**15**) Que réalise la fonction apply sur le langage des numériques
typedef int langage2;
langage1 apply (langage2 X){
int i = 0;
char langange1 s[100];
langage1 result;
do s[i++] = X % 10 + '0'; while ((X /= 10) > 0);
s[i] = '\0';
process(s);
result = (langage1) malloc(strlen(s) + 1);
strcpy(result, s);
return result;
}
		(a) calcule la chaîne décimale du mot binaire
		
		(b) calcule la chaîne binaire du mot décimale
		
		(c) renvoie l'image numérique du texte représentant le mot
		
		(d) renvoie l'image textuelle du mot

Exercice I : Réseau de concepts (10 pts)

Pour chaque concept/question, remplissez la case de la colonne des choix uniques correspondante par un choix qui soit le plus adéquat

Concept/Question	Choix unique	Choix possibles
(**1**) Bytecode Java	(**B**)	(**A**) démontrer qu' « une grammaire est ambiguë » est décidable mais l'inverse est non décidable
(**2**) ADDOP REG1, REG2		(**B**) Représentation
(**3**) Nombre de registres nécessaire pour un expression arithmétique		(**C**) Représentation
(**4**) Acorn RISC Machine-ARM		(**D**) Erreur Lexicale
(**5**) AST	(**C**)	(**E**) Attribut nécessaire à la génération de pseudo-code
(**6**) Commentaire C non fermé (/* sans */)		(**F**) two-address code
(**7**) Grammaire Ambiguë		(**G**) three-address code
(**8**) Récursivité Gauche		(**H**) Tri et tri inverse des feuilles
(**9**) Grammaire non LL		(**I**) Analyse floue
(**10**) Grammaire algébrique		(**J**) Analyse descendante non optimale
(**11**) *(null).suivant		(**K**)Erreur Syntaxique
(**12**) Select Pilote.Nom from * ;		(**L**) Analyse sans fin
(**13**) (n{m)n}m		(**M**) Analyse ascendante
(**14**) Grammaire LR		(**N**) Erreur Sémantique
(**15**) Dérivation droite et gauche		(**O**) Analyse impossible
(**16**) semi-décidabilité	(**A**) résolue	(**P**) Equation linéaire

Enoncé Exam 2010

Université Mohammed V – Souissi
Ecole Nationale Supérieure d'Informatique et d'Analyse des Systèmes

Examen

Année Universitaire : 2009 - 2010
Filière : Ingénieur
Semestre : S3
Période : P2

Module : **M3.4** - Compilation
Elément de Module : **M3.4.1** - Compilation
Professeur : Karim BAÏNA

- **Date : 15/01/2010**
- **Durée : 2H00**

Consignes aux élèves ingénieurs :
- *Seule la fiche de synthèse (A4 recto/verso) est autorisée !!*
- *Le barème est donné seulement à titre indicatif !!*
- *Les **réponses directes** et **synthétiques** seront appréciées*
- *Soignez votre **présentation** et **écriture** !!*

Exercice I : Syntaxe et Représentations intermédiaires *(20 pts)*

Soit la grammaire LALR du langage ZZ

PROG : LISTE_DECL LISTE_INST ;
LISTE_DECL : DECL | LISTE_DECL DECL;
DECL : **idf** TYPE CONST_IB ;
TYPE : **int** | **double** | **bool** ;
CONST_IB : **iconst** | **dconst** | TRUEFALSE ;
TRUEFALSE : **true** | **false** ;
LISTE_INST : INST | LISTE_INST INST ;
INST : **idf** "**:=**" EXPA /* *Affectation arithmétique*/ |
if '**(**' IDF '**=**' EXPA '**)** **then** LISTE_INST **endif** /* *Conditionnelle arithmétique*/ |
if '**(**' IDF '**=**' EXPA '**)** **then** LISTE_INST **else** LISTE_INST **endif** |
PRINT **idf** ; /* *Affichage d'une variable* */
EXPA : EXPA '**+**' EXPA | EXPA '**-**' EXPA | EXPA '*****' EXPA | EXPA '/' EXPA | '**(**' EXPA '**)**' | **iconst** | **dconst** | **idf** ;

Avec les priorités usuelles et associativités gauches des opérateurs arithmétiques '+', '-', '*' et '/'

1. Ajouter à la grammaire l'instruction d'affichage d'une chaîne de caractère (2pts)
Exemple, le programme : INT X 11 **PRINT** "# *X* = " **PRINT** X **PRINT** "#\n" produit : # X = 11 #

2. Ajouter à la grammaire l'instruction d'affectation booléenne complexe (2pts)
Exemple : x := (x and y or not z)
%left or
%left and
%left not

3. Après l'enrichissement de la question (2) (a) que remarquez – vous, (b) que proposez-vous ? (2pts)

4. Ajouter à la grammaire la conditionnelle booléenne (2pts)
Exemple : if (x = true)… if (x = false) … if (x = ((not x) and (y or z)))

5. Après l'enrichissement de la question (4) (a) que remarquez – vous, (b) que proposez-vous ? (2pts)

6. Enrichir les types suivants pour prendre en compte les enrichissements I.1, I.2 et I.4 (2pts)
On supposera défini ASTB (par analogie à ASTA type des arbres abstraits arithmétiques) le type des arbres abstraits booléens.

```
typedef struct INST {
  Type_INST typeinst;
  union  {
    // idf := EXPA
    struct  {
      int rangvar; // indice de l'idf (left exp), où il faut affecter, dans la table des symboles
      ASTA right; // l'expression arithmétique droite (right exp) à affecter
    } arithassignnode;
    // if ... then  … else
    struct  {
      int rangvar; // indice de l'idf (left exp) à comparer, dans la table des symboles
      ASTA right; // l'expression arithmétique (right exp) à comparer
      struct LIST_INST * thenlinst; // then list of instructions
      struct LIST_INST * elselinst; // else list of instructions
    } ifnode;
    // PRINT idf
    struct  {
      int rangvar; // indice de l'idf (à afficher) dans la table des symboles
    } printnode;
  } node;
} instvalueType;
```

```
typedef struct LIST_INST {
  struct INST first;
  struct LIST_INST * next;
} listinstvalueType;

typedef enum
{       PrintIdf,
        AssignArith,
        AssignBool,
        IfThenArith,
        IfThenElseArith
} Type_INST ;
```

7. Donner 4 erreurs sémantiques différentes engendrées par les enrichissements I.2 et I.4 (2pts)

8. Nous voudrions pouvoir exprimer des comparaisons riches et les utiliser dans les affectations et les conditionnelles
BOOL x FALSE
Exemples d'affectations booléennes : x := (l <= (50+ y * y)) ou x := ((25 * m) >= (50+ y * y)) ou x := (z = true) ou x := ((z or f) = true)
Exemples de conditionnelles : if (x) … ou if (l <= (50+ y * y)) ou if ((z or f) = true)
Les opérateurs de comparaisons supportés (=, <=, >=).
Modifier la grammaire pour prendre en compte cet enrichissement (2pts)

9. Enrichir les types de la question I.6 pour prendre en compte les enrichissements I.8 (2pts)

10. Les représentations intermédiaires graphiques produites à la fin de la phase d'analyse sont-elles vraiment indispensables puisque nous pouvons nous en passer pour générer le pseudo-code en même temps que l'analyse syntaxico-sémantique sans utiliser ni AST, ni DAG, ni CFG, … (*syntax driven translation*) (2pts)

Exercice II : Machine Virtuelle et Génération de pseudo-code *(10 pts, dont au max 4 de bonus TP)*

Soit l'instruction for dont la syntaxe est la suivante :
INST : **for idf ":="** nombre **to** nombre **loop** LIST_INST **end loop ;** | …

Son type d'instruction : typedef enum {…. forLoop } Type_INST ;

Et sa représentation intermédiaire (faisant part du type *node*)

```
typedef struct INST {
        Type_INST typeinst;
        union { …. // les autres types d'instructions
                // for idf ":=" nombre to nombre loop LIST_INST end loop ;
                struct  {
                int rangvar; // indice de l'idf (variable d'induction de la boucle à comparer) dans la table des symboles
                int min; // la valeur de la borne inférieure de l'intervalle d'itération
                int max; // la valeur de la borne supérieure de l'intervalle d'itération
                struct LIST_INST * forbodylinst; // la liste d'instructions corps de la boucle pour
                } fornode;
        } node;
} instvalueType;
```

Nous rappelons les structures de base :

```
typedef enum {ADD, DIV, DUPL, JMP, JNE, JG, LABEL, LOAD, MULT, POP, PRNT, PUSH, SUB, STORE, SWAP} CODOP;

typedef union {
  char * var;            // pour LOAD / STORE
  double _const;         // pour PUSH
  char * label_name; // pour JMP/JNE/JG/LABEL
} Param ;

typedef struct pseudocodenode * pseudocode;

struct pseudoinstruction{
        CODOP codop ;
        Param param ; // une opération possède un paramètre au maximum
} ;
struct pseudocodenode{
        struct pseudoinstruction first ;
        struct pseudocodenode * next ;
} ;
```

Comme vu en cours, la fonction **void interpreter_list_inst(listinstvalueType * plistinstattribute)** et
La fonction **pseudocode generer_pseudo_code_list_inst(listinstvalueType * plistinstattribute)** sont déjà définies.

1. Compléter l'interpréteur de représentations intermédiaire pour prendre en compte l'instruction for : (2pts)

```
void interpreter_inst(instvalueType instattribute){
        switch(instattribute.typeinst){
                case forLoop :
                        // à compléter ….
                break ;
        } // end switch

}
```

2. Compléter le générateur de code pour prendre en compte l'instruction for : (2pts)

```
pseudocode generer_pseudo_code_inst(instvalueType instattribute){
pseudocode  pc = (pseudocode)malloc(sizeof (struct pseudocodenode));
        switch(instattribute.typeinst){
                case forLoop :
                        // à compléter ….
                break ;
                } // end switch
        return pc ;
}
```

3. (i) Spécifier les profils des fonctions du type abstrait de données pile (empiler, dépiler, tête_pile, taille_pile, pile_vide) **sans les implémenter. (ii) Supposer l'existence d'une pile système globale** pile VM_STACK ; **liée à la machine virtuelle, (iii) Réaliser l'interpréteur du pseudo-code intégré à la machine virtuelle à travers les deux fonctions :** (2pts)

```
void interpreter_pseudo_code_inst(pseudoinstruction pc) ;
void interpreter_pseudo_code_list_inst(pseudocode pc) ;
```

Corrigé Exam 2010

Université Mohammed V – Souissi
Ecole Nationale Supérieure d'Informatique et d'Analyse des Systèmes

Examen

Année Universitaire : 2009 - 2010
Filière : Ingénieur
Semestre : S3
Période : P2

Module : **M3.4** - Compilation
Elément de Module : **M3.4.1** - Compilation
Professeur : Karim BAÏNA

- **Date : 15/01/2010**
- **Durée : 2H00**

Consignes aux élèves ingénieurs :
- *Seule la fiche de synthèse (A4 recto/verso) est autorisée !!*
- *Le barème est donné seulement à titre indicatif !!*
- *Les **réponses directes** et **synthétiques** seront appréciées*
- *Soignez votre **présentation** et **écriture** !!*

Exercice I : Syntaxe et Représentations intermédiaires *(20 pts)*

Soit la grammaire LALR du langage ZZ

PROG : LISTE_DECL LISTE_INST ;
LISTE_DECL : DECL | LISTE_DECL DECL;
DECL : **idf** TYPE CONST_IB ;
TYPE : **int** | **double** | **bool** ;
CONST_IB : **iconst** | **dconst** | TRUEFALSE ;
TRUEFALSE : **true** | **false** ;
LISTE_INST : INST | LISTE_INST INST ;
INST : **idf** "**:=**" EXPA */* Affectation arithmétique*/* |
if '**(**' IDF '**=**' EXPA '**)** **then** LISTE_INST **endif** */* Conditionnelle arithmétique*/* |
if '**(**' IDF '**=**' EXPA '**)** **then** LISTE_INST **else** LISTE_INST **endif** |
PRINT **idf** ; */* Affichage d'une variable */*
EXPA : EXPA '**+**' EXPA | EXPA '**-**' EXPA | EXPA '*****' EXPA | EXPA '/' EXPA | '**(**' EXPA '**)**' | **iconst** | **dconst** | **idf** ;

Avec les priorités usuelles et associativités gauches des opérateurs arithmétiques '+', '-', '*' et '/'

1. Ajouter à la grammaire l'instruction d'affichage d'une chaîne de caractère (2pts)
Exemple, le programme : INT X 11 **PRINT** "# *X* = " **PRINT** X **PRINT** "#\n" produit : # X = 11 #

On ajoutera un nouveau *terminal string représentant l'expression régulière des chaînes de caractères ["][^\n"]*["] qu'il n'est pas demandé de définir :*

INST : PRINT string ;

2. Ajouter à la grammaire l'instruction d'affectation booléenne complexe (2pts)
Exemple : x := (x and y or not z)
%left or
%left and
%left not

On ajoutera un nouveau non-terminal EXPB dérivant les expressions booléennes :

EXPB : EXPB or EXPB | EXPB and EXPB | not EXPB | '(' EXPB ')' | TRUEFALSE | idf ;
NB. *il n'est pas demandé de désambiguïser ces règles !*

3. Après l'enrichissement de la question (2) (a) que remarquez – vous, (b) que proposez-vous ? (2pts)

(a) La grammaire devient ambiguë du fait qu'un IDF peut être dérivé à partir des non-terminaux EXPA et EXPB. ….. (*1 pt*)
(b) démarche de désambiguisation….. ….. (*1 pt*)

4. Ajouter à la grammaire la conditionnelle booléenne (2pts)
Exemple : if (x = true)… if (x = false) … if (x = ((not x) and (y or z)))

On ajoutera deux règles à la grammaire

INST : **if** '**(**' IDF '**=**' EXPB '**)** **then** LISTE_INST **endif** */* Conditionnelles booléennes */* |
if '**(**' IDF '**=**' EXPB '**)** **then** LISTE_INST **else** LISTE_INST **endif**

5. Après l'enrichissement de la question (4) (a) que remarquez – vous, (b) que proposez-vous ? (2pts)

(a) La grammaire devient de nouveau ambiguë du fait qu'une conditionnelle if '(' IDF '=' IDF ')' peut être dérivé à partir des instructions : if '(' IDF '=' EXPA ')' et : if '(' IDF '=' EXPB ')' (*1 pt*)
(b) démarche de désambiguisation….. (*1 pt*)

6. Enrichir les types suivants pour prendre en compte les enrichissements I.1, I.2 et I.4 (2pts)
On supposera défini ASTB (par analogie à ASTA type des arbres abstraits arithmétiques) le type des arbres abstraits booléens.

```
typedef struct INST {
  Type_INST typeinst;
  union  {
    // idf := EXPA
```

```
typedef struct LIST_INST {
  struct INST first;
  struct LIST_INST * next;
} listinstvalueType;
```

```
  struct  {
    int rangvar; // indice de l'idf (left exp), où il faut affecter, dans la table des symboles
    ASTA right; // l'expression arithmétique droite (right exp) à affecter
  } arithassignnode;
  // if ... then  … else arithmétique
  struct  {
    int rangvar; // indice de l'idf (left exp) à comparer, dans la table des symboles
    ASTA right; // l'expression arithmétique (right exp) à comparer
    struct LIST_INST * thenlinst; // then list of instructions
    struct LIST_INST * elselinst; // else list of instructions
  } ifnode;
  // if ... then  … else booléenne
  struct  {
    int rangvar; // indice de l'idf (left exp) à comparer, dans la table des symboles
    ASTB right; // l'expression booléenne (right exp) à comparer
    struct LIST_INST * thenlinst; // then list of instructions
    struct LIST_INST * elselinst; // else list of instructions
  } ifnodebool;

  // PRINT idf
  struct  {
    int rangvar; // indice de l'idf (à afficher) dans la table des symboles
  } printnode;
  // PRINT string
  struct  {
    char * chaine ; // chaine de caractères à afficher
  } printnode;

 } node;
} instvalueType;
```

```
typedef enum
{       PrintIdf,
        PrintString,
        AssignArith,
        AssignBool,
        IfThenArith,
        IfThenElseArith,
        IfThenBool,
        IfThenElseBool
} Type_INST ;
```

7. Donner 4 erreurs sémantiques différentes engendrées par les enrichissements I.2 et I.4 (2pts)

En voici 6 erreurs sémantiques nouvelles (toutes *4 parmi ces 6 sont suffisantes*) :
1. Dans if '(' IDF '=' EXPB ')' la partie droite contient un identificateur non déclaré
2. Dans if '(' IDF '=' EXPB ')' la partie droite contient un identificateur déclaré d'un autre type que BOOL
3. Dans if '(' IDF '=' EXPB ')' la partie droite contient un identificateur non initialisé
4. Dans if '(' IDF '=' EXPB ')' la partie gauche est un identificateur non déclaré
5. Dans if '(' IDF '=' EXPB ')' la partie gauche est un identificateur déclaré d'un autre type que BOOL
6. Dans if '(' IDF '=' EXPB ')' la partie gauche est un identificateur non initialisé

8. Nous voudrions pouvoir exprimer des comparaisons riches et les utiliser dans les affectations et les conditionnelles
BOOL x FALSE
Exemples d'affectations booléennes : x := (I <= (50+ y * y)) ou x := ((25 * m) >= (50+ y * y)) ou x := (z = true) ou x := ((z or f) = true)
Exemples de conditionnelles : if (x) … ou if (I <= (50+ y * y)) ou if ((z or f) = true)
Les opérateurs de comparaisons supportés (=, <=, >=).
Modifier la grammaire pour prendre en compte cet enrichissement (2pts)

On ajoutera un non-terminal COMP (*expressions booléennes complexes*) dérivant les comparaisons arithmétiques et booléennes en plus des règles suivantes à la grammaire :

COMP : EXPA <= EXPA | EXPA => EXPA | EXPA = EXPA | EXPB = EXPB | EXPB

INST : **idf** ":=" COMP /* *Affectation booléennes* */ |
if '**(**' COMP '**)**' **then** LISTE_INST **endif** /* *Conditionnelles générales arithmétiques et booléennes* */ |
if '**(**' COMP '**)**' **then** LISTE_INST **else** LISTE_INST **endif**

On supprimera les règles suivantes de la grammaire :

~~**INST** : **if** '(' IDF '=' EXPA ')' **then** LISTE_INST **endif** /* *Conditionnelles arithmétiques* */ |~~
~~**if** '(' IDF '=' EXPA ')' **then** LISTE_INST **else** LISTE_INST **endif** |~~
~~**if** '(' IDF '=' EXPB ')' **then** LISTE_INST **endif** /* *Conditionnelles booléennes* */~~
~~**if** '(' IDF '=' EXPB ')' **then** LISTE_INST **else** LISTE_INST **endif**~~

9. Enrichir les types de la question I.6 pour prendre en compte les enrichissements I.8 (2pts)

On supposera l'existence du type ASTCOMP : un AST pour stocker les expressions booléennes complexes COMP.

```
typedef struct INST {
  Type_INST typeinst;
  union  {
   …

   // idf := COMP
   struct  {
     int rangvar; // indice de l'idf (left exp), où il faut affecter, dans la table des symboles
     ASTCOMP right; // l'expression booléenne complexe droite (right exp) à affecter
   } boolassignnode;
```

```
typedef struct LIST_INST {
  struct INST first;
  struct LIST_INST * next;
} listinstvalueType;

typedef enum
{       PrintIdf,
        PrintString,
        AssignArith,
        AssignBool,
```

```
                                                                    IfThen,
  // if ... then  … else arithmétique et booléen                    IfThenElse,
  struct  {                                                      } Type_INST ;
    ASTCOMP comparison; // l'expression booléenne complexe
   struct LIST_INST * thenlinst; // then list of instructions
   struct LIST_INST * elselinst; // else list of instructions
  } ifnode;

  …

 } node;
} instvalueType;
```

10. Les représentations intermédiaires graphiques produites à la fin de la phase d'analyse sont-elles vraiment indispensables puisque nous pouvons nous en passer pour générer le pseudo-code en même temps que l'analyse syntaxico-sémantique sans utiliser ni AST, ni DAG, ni CFG, … (*syntax driven translation*) (2pts)

C'est vrai, la production de représentations intermédiaires graphiques n'est pas indispensable pour des cas particuliers mais pas en général.

En effet, l'analyse syntaxico-sémantique suit un sens (descendant : famille de parseurs LL, descendant récursif, etc. ou ascendant famille de parseurs LR/SLR/LALR, shift-reduce, etc.). Cela limite le sens de calcul des attributs qui peuvent être selon le contexte sémantique hérités ou synthétisés.

En général, une grammaire peut contenir des attributs de tout genre ce qui se contredit avec le sens de l'analyse (exemple : une analyse LR ne permettra pas de calculer d'attributs hérités pendant la réduction de règles).

Il s'avère donc, plus pratique de stocker le résultat de l'analyse syntaxico-sémantique dans une représentation intermédiaire afin de pouvoir effectuer tous les calculs d'attributs nécessitant des parcours descendant ou ascendant et donc ne plus être lié au sens de l'analyse syntaxico-sémantique lui-même.

Exercice II : Machine Virtuelle et Génération de pseudo-code *(10 pts, dont au max 4 de bonus TP)*

Soit l'instruction for dont la syntaxe est la suivante :
INST : **for idf ":="** nombre **to** nombre **loop** LIST_INST **end loop ;** | …

Son type d'instruction : typedef enum {…. forLoop } Type_INST ;

Et sa représentation intermédiaire (faisant part du type *node*)

```
typedef struct INST {
        Type_INST typeinst;
        union { …. // les autres types d'instructions
                // for idf ":=" nombre to nombre loop LIST_INST end loop ;
                struct  {
                int rangvar; // indice de l'idf (variable d'induction de la boucle à comparer) dans la table des symboles
                int min; // la valeur de la borne inférieure de l'intervalle d'itération
                int max; // la valeur de la borne supérieure de l'intervalle d'itération
                struct LIST_INST * forbodylinst; // la liste d'instructions corps de la boucle pour
                } fornode;
        } node;
} instvalueType;
```

Nous rappelons les structures de base :

```
typedef enum {ADD, DIV, DUPL, JMP, JNE, JG, LABEL, LOAD, MULT, POP, PRNT, PUSH, SUB, STORE, SWAP} CODOP;

typedef union {
  char * var;            // pour LOAD / STORE
  double _const;         // pour PUSH
  char * label_name; // pour JMP/JNE/JG/LABEL
} Param ;

typedef struct pseudocodenode * pseudocode;

struct pseudoinstruction{
        CODOP codop ;
        Param param ; // une opération possède un paramètre au maximum
} ;
struct pseudocodenode{
        struct pseudoinstruction first ;
        struct pseudocodenode * next ;
} ;
```

Comme vu en cours, la fonction **void interpreter_list_inst(listinstvalueType * plistinstattribute)** et
La fonction **pseudocode generer_pseudo_code_list_inst(listinstvalueType * plistinstattribute)** sont déjà définies.

1. Compléter l'interpréteur de représentations intermédiaire pour prendre en compte l'instruction for : (2pts)

```
        void interpreter_inst(instvalueType instattribute){
                switch(instattribute.typeinst){
                        case forLoop :
                                set_value(instattribute.node.fornode.rangvar, instattribute.node.fornode.min) ;
                                // oubien TS[instattribute.node.fornode.rangvar] := instattribute.node.fornode.min

                                if  (get_value(instattribute.node.fornode.rangvar)  <=  instattribute.node.fornode.max)  {
                                // oubien if(TS[instattribute.node.fornode.rangvar]<= instattribute.node.fornode.max)
                                        interpreter_list_inst( forbodylinst ) ;
                                }
                        break ;
                } // end switch
```

```
}
```

2. Compléter le générateur de code pour prendre en compte l'instruction for : (2pts)

```
pseudocode generer_pseudo_code_inst(instvalueType instattribute){
pseudocode  pc = (pseudocode)malloc(sizeof (struct pseudocodenode));

// déclarer une variable static indice_boucle initialisée à 0
static int loopindex = 0 ;

        switch(instattribute.typeinst){
                case forLoop :
                // ALGORITHME :
                // 1. générer le code d'initialisation de la variable d'induction par la borne inf de l'intervalle
                // 2. générer le label de début de la boucle loop (le label doit être suffixé d'une clef unique
loop_1:.. loop2 :)
                // 3. générer la comparaison de la variable d'induction avec la borne sup de l'intervalle
                // 4. générer le saut vers le label de fin si la variable est supérieure à cette borne sup
                // 5. générer récursivement le code relatif au corps de la boucle
                // 6. générer le label de fin de la boucle loop (le label doit être suffixé d'une clef unique la même
que le label de début de la boucle fin_loop_1:.. fin_loop2 :)
                // 7. incrémenter l'indice de la boucle pour la prochaîne boucle loop
                indice_boucle ++ ;

                break ;
                } // end switch
        return pc ;
}
```

3. (i) Spécifier les profils des fonctions du type abstrait de données pile (empiler, dépiler, tête_pile, taille_pile, pile_vide) **sans les implémenter. (ii) Supposer l'existence d'une pile système globale** pile VM_STACK ; **liée à la machine virtuelle, (iii) Réaliser l'interpréteur du pseudo-code intégré à la machine virtuelle à travers les deux fonctions :** (2pts)

```
void interpreter_pseudo_code_inst(pseudoinstruction pci) ;
void interpreter_pseudo_code_list_inst(pseudocode pc) ;

void interpreter_pseudo_code_list_inst(pseudocode pc) ;
        if (pc != NULL) {
                // interpretation de la première l'instruction
                interpreter_pseudo_code_inst( pc->first ) ;

                // appel récursif sur la suite de pseudocode
                interpreter_pseudo_code_list_inst( pc-> next ) ;
        }
}

void interpreter_pseudo_code_inst(pseudocodeinstruction pci) {
        // SQUELETTE DU PROGRAMME A RAFFINER:

        switch(pci.codop){
        case ADD :
                op1 = VM_STACK.depiler() ;
                op2 = VM_STACK.depiler() ;
                VM_STACK.empiler(op1 + op2) ;
                break ;
        case DIV :
                op1 = VM_STACK.depiler() ;
                op2 = VM_STACK.depiler() ;
                VM_STACK.empiler(op1 / op2) ;
                break ;
        case DUPL :
                VM_STACK.empiler(VM_STACK.tetepile()) ;
                break ;
        case JMP :
                interpreter_pseudo_code_inst(rechercher_instruction_au_label(pci, pci.param.label_name)) ;
                break ;
        …
        case LOAD :
                op1 = VM_STACK.empiler(@pci.param.var) ;
                break ;
        case SWAP :
                op1 = VM_STACK.depiler() ;
                op2 = VM_STACK.depiler() ;
                VM_STACK.empiler(op2) ;
                VM_STACK.empiler(op1) ;
                break ;
        ….
        }
}
```

Enoncé Exam 2011

Université Mohammed V – Souissi
École Nationale Supérieure d'Informatique et d'Analyse des Systèmes

Examen

Année Universitaire : 2010 - 2011
Filière : Ingénieur
Semestre : S3
Période : P2

Module : **M3.4** - Compilation
Élément de Module : **M3.4.1** - Compilation
Professeur : Karim BAÏNA

- **Date : 12/01/2011**
- **Durée : 2H00**

Consignes aux élèves ingénieurs :

- *Le barème est donné seulement à titre indicatif !!*
- *Les **réponses directes** et **synthétiques** seront appréciées*
- *Soignez votre **présentation** et **écriture** !!*

Soit la grammaire du langage SQL simplifié :

<SELECT> ::= **select** <PROJECT> <FROM>
<PROJECT> ::= '*****' | <COLUMNS>
<FROM> ::= **from** <TABS> <FROMAUX>
<COLUMNS> ::= <COLUMN> <COLUMNAUX>
<COLUMNAUX> ::= '**,**' <COLUMNS> | **ε**
<COLUMN> ::= **idf** <POINTEDCOLUMN>
<POINTEDCOLUMN> ::= '**.**' **idf** | **ε**
<TABS> ::= **idf** | **idf** ',' <TABS>
<FROMAUX> ::= <WHERE> ';' | <ORDERBY> ';' | ';'
<WHERE> ::= **where** <EXPBOOL>
<EXPBOOL> ::= **not** <EXPBOOL>
| <EXPBOOL> **and** <EXPBOOL>
| <EXPBOOL> **or** <EXPBOOL>
| <COLUMN> <OP> <COLUMN>
<OP> ::= **lower** | **loweroreq** | **greater** | **greateroreq** | **eq** | **neq**
<ORDERBY> ::= **orderby** <COLUMNS>

1) Désambiguïser la grammaire en réécrivant les règles correspondant au non-terminal ou aux non-terminaux ambigus avec les priorités habituelles (2pts)

<EXPBOOL> ::= <OR>
<OR> ::= .....
<NOT> ::= .....
<AND> ::= .....
<AUX> ::= <COLUMN> <OP> <COLUMN>

2) Éliminer la récursivité à gauche (ne donner que les nouvelles règles) (2pts)

<OR> ::= .....
<ORAUX> ::= .....
<NOT> ::= .....
<NOTAUX> ::= .....
<AND> ::= .....
<ANDAUX> ::= .....

3) Rendre la grammaire LL(1) (ne donner que les nouvelles règles) (2pts)

<TABS> ::= .....
<TABSAUX> ::= .....

4) Calculer les directifs First et Follow des NT nullables (2pts)

Non-terminal	Les premiers (*First*)	Les suivants (*Follow*)
<COLUMNAUX>	.....	.....
<POINTEDCOLUMN>	.....	.....
<OR>	.....	.....
<NOTAUX>	.....	.....
<ANDAUX>	.....	.....
<TABSAUX>	.....	.....

5) Programmer en C le prédicat pointedcolumn faisant partie de l'analyseur syntaxique LL(1) (2pts)

.....
.....
.....
.....
.....
.....
.....
.....

6) Lister 4 erreurs sémantiques possibles (2pts)

erreur 1 - .....
erreur 2 - .....
erreur 3 - .....
erreur 4 - .....

7) Questions sur le code étudié en TP : (8pts)

7.1) Soit le type INST vu en TP, améliorer le pour prendre en compte l'instruction for. (2pts)

```
typedef struct INST {

  Type_INST typeinst;                    .....
  union  {                               .....
    // PRINT idftoprint                  .....
    struct  {                            .....
      int rangvar;                       .....
    } printnode;                         .....
    // left := right                     .....
    struct  {                            .....
      int rangvar;                       .....
      AST right;                         .....
    } assignnode;                        .....
    // IF ... THEN                       .....
    struct  {                            .....
      int rangvar;
      AST right;
      struct LIST_INST * thenlinst;
      struct LIST_INST * elselinst;
    } ifnode;
  } node;
} instvalueType;
```

7.2) Qu'est ce qui joue le rôle du tas dans la programmation de la mémoire virtuelle étudiée en TP ? **(2pts)**
.....
.....

7.3) Qu'est ce qui joue le rôle de la mémoire statique dans la programmation de cette mémoire virtuelle ? **(2pts)**
.....
.....

7.4) Donner deux limitations à la fonction interpreter_pseudo_code vue en TP (en justifiant) : **(2pts)**

```
void interpreter_pseudo_code(pseudocode pc){
 char ** next_label_name = (char **) malloc(sizeof (char*));
 if (pc != NULL){
    interpreter_pseudo_instruction(pc->first, next_label_name);
    if (*next_label_name == NULL) interpreter_pseudo_code(pc->next); // Il n y a pas de branchement !!
    else{ // JNE ou JMP ==> effectuer un branchement
            struct pseudocodenode * compteur_ordinal = pc->next;
            while ( (compteur_ordinal->first.codop != LABEL) ||
                        (strcmp(compteur_ordinal->first.param.label_name, *next_label_name) != 0) ) {
                        // (compteur_ordinal ne peut jamais == NULL) après JMP/JNE dans le code (par construction)
                        compteur_ordinal = compteur_ordinal->next;
            }
            interpreter_pseudo_code(compteur_ordinal); // branchement
    }
 }
}
```

limitation 1 – .....
.....

limitation 2 – .....
.....

Corrigé Exam 2011

Université Mohammed V – Souissi
École Nationale Supérieure d'Informatique et d'Analyse des Systèmes

Correction Examen

Année Universitaire : 2010 - 2011
Filière : Ingénieur
Semestre : S3
Période : P2

Module : **M3.4** - Compilation
Élément de Module : **M3.4.1** - Compilation
Professeur : Karim BAÏNA

- **Date : 12/01/2011**
- **Durée : 2H00**

Consignes aux élèves ingénieurs :
- *Le barème est donné seulement à titre indicatif !!*
- *Les **réponses directes** et **synthétiques** seront appréciées*
- *Soignez votre **présentation** et **écriture** !!*

Soit la grammaire du langage SQL simplifié :

<SELECT> ::= **select** <PROJECT> <FROM>
<PROJECT> ::= '*****' | <COLUMNS>
<FROM> ::= **from** <TABS> <FROMAUX>
<COLUMNS> ::= <COLUMN> <COLUMNAUX>
<COLUMNAUX> ::= '**,**' <COLUMNS> | **ε**
<COLUMN> ::= **idf** <POINTEDCOLUMN>
<POINTEDCOLUMN> ::= '**.**' **idf** | **ε**
<TABS> ::= **idf** | **idf** '**,**' <TABS>
<FROMAUX> ::= <WHERE> '**;**' | <ORDERBY> '**;**' | '**;**'
<WHERE> ::= **where** <EXPBOOL>
<EXPBOOL> ::= **not** <EXPBOOL>
| <EXPBOOL> **and** <EXPBOOL>
| <EXPBOOL> **or** <EXPBOOL>
| <COLUMN> <OP> <COLUMN>
<OP> ::= **lower** | **loweroreq** | **greater** | **greateroreq** | **eq** | **neq**
<ORDERBY> ::= **orderby** <COLUMNS>

1) Désambiguïser la grammaire en réécrivant les règles correspondant au non-terminal ou aux non-terminaux ambigus avec les priorités habituelles (2pts)

(i) priorités du cours : OR << NOT << AND, (ii) convention : associativité gauche (la plus utilisée et logique vu l'exercice 2 !!)

<EXPBOOL> ::= <OR>
<OR> ::= <OR> **or** <NOT> | <NOT>
<NOT> ::= **not** <NOT> | <AND>
<AND> ::= <AND> **and** <AUX> | <AUX>
<AUX> ::= <COLUMN> <OP> <COLUMN

2) Éliminer la récursivité à gauche (ne donner que les nouvelles règles) (2pts)

<OR> ::= <NOT> <ORAUX>
<ORAUX> ::= **or** <NOT><ORAUX> | ε
<NOT> ::= ***(cette règle ne change pas ne pas pénaliser si répétée !!)***
<NOTAUX> ::= ***(ni ce terminal, ni cette règle n'existe !!)***
<AND> ::= <AUX> <ANDAUX>
<ANDAUX> ::= **and** <AUX> <ANDAUX> | ε
<AUX> ::= <COLUMN> <OP> <COLUMN> | (<OR>) règle optionnelle à bonifier, sans pénaliser si omise !!

3) Rendre la grammaire LL(1) (ne donner que les nouvelles règles) (2pts)

<TABS> ::= **idf** <TABSAUX>
<TABSAUX> ::= '**,**' <TABS> | ε

4) Calculer les directifs First et Follow des NT nullables (2pts)

Non-terminal	Les premiers (*First*)	Les suivants (*Follow*)
<COLUMNAUX>	**,**	**from**, ';'
<POINTEDCOLUMN>	'.'	',', ';', **lower**, **loweroreq**, **greater**, **greateroreq**, **eq**, **neq**
<OR> **N'EST PAS NULLABLE (*) (bonifier 2 cas : (*) et ligne tableau vide OU ligne tableau correcte !!**	**idf, not**	';'
<NOTAUX> N'EXISTE PAS !!	VIDE	VIDE
<ANDAUX>	**and**	**or**, ';'
<TABSAUX>	','	**where**, **orderby**, ';'

5) Programmer en C le prédicat pointedcolumn faisant partie de l'analyseur syntaxique LL(1) (2pts)

On supposera que l'appel à lire_token() se fait avant chaque prédicat et que lire_token est exactement celle programmée en TP

```
boolean pointedcolumn(){
boolean result;
if((token==virgule)||(token==pointvirgule)||(token==lower)||(token==loweroreq)||(token==greater)||(token==greteroreq)||(token==eq)||(token==neq){
        follow_token = true;
        result = true;
else if (token == point) {
        token = lire_token();
        if (token = idf)        result = true;
        else result = false;
}else result = false;
return result;}
```

6) Lister 4 erreurs sémantiques possibles (2pts)
erreur 1 - Table (T) non existante dans la base de donnée
erreur 2 - Champs (C) non existant dans aucune table de la clause FROM
erreur 3 - Champs (T.C) pointé par une table qui n'est pas la sienne
erreur 4 - Comparaison entre deux champs de types incompatibles
autres erreurs *– à vous de juger si d'autres erreurs valent le coup (si pas d'intersection avec erreurs 1-4)*

7) Questions sur le code étudié en TP : (8pts)

7.1) Soit le type INST vu en TP, améliorer le pour prendre en compte l'instruction for. (2pts)

```
typedef struct INST {

  Type_INST typeinst;        .....
  union  {                   .....
    // PRINT idftoprint      Il faut ajouter à l'union la structure suivante
    struct  {                // for (index:= exp_min..exp_max) loop list_inst end loop;
      int rangvar;           struct  {.....
    } printnode;                     int rangvar; // indice de l'index de la boucle
    // left := right                 AST  borneinf; // l'expression borne inf      (int borneinf; est acceptable !!!)
    struct  {                        AST bornesup; // l'expression borne sup     (int bornesup; est acceptable !!!)
      int rangvar;                   struct LIST_INST * forbodylinst; // for body list of instructions
      AST right;             } fornode;.....
    } assignnode;            .....
    // IF ... THEN           // la première solution avec les AST est la meilleure à distinguer par rapport à la 2ème.
    struct  {                .....
      int rangvar;           // améliorer le type Type_INST for est optionnel à bonifier mais ne pas pénaliser si omis !!
      AST right;             .....
      struct LIST_INST * thenlinst;
      struct LIST_INST * elselinst;
    } ifnode;
  } node;
} instvalueType;
```

7.2) Qu'est ce qui joue le rôle du tas dans la programmation de la mémoire virtuelle étudiée en TP ? **(2pts)**
le langage ZZ n'offrant pas d'instruction d'allocation dynamique de type (malloc), le tas n'est pas géré par la mémoire virtuelle (la pile peut donc prendre toute la mémoire non consommée par le mémoire code et la mémoire donnée (statique)).

7.3) Qu'est ce qui joue le rôle de la mémoire statique dans la programmation de cette mémoire virtuelle ? **(2pts)**
Nous avons réutilisé la table des symbôles comme solution simple de gestion de la mémoire des données (statique).

7.4) Donner deux limitations à la fonction interpreter_pseudo_code vue en TP (en justifiant) : **(2pts)**

```
void interpreter_pseudo_code(pseudocode pc){
 char ** next_label_name = (char **) malloc(sizeof (char*));
 if (pc != NULL){
   interpreter_pseudo_instruction(pc->first, next_label_name);
   if (*next_label_name == NULL) interpreter_pseudo_code(pc->next); // Il n y a pas de branchement !!
   else{ // JNE ou JMP ==> effectuer un branchement
         struct pseudocodenode * compteur_ordinal = pc->next;
         while ( (compteur_ordinal->first.codop != LABEL) ||
                 (strcmp(compteur_ordinal->first.param.label_name, *next_label_name) != 0) ) {
                 // (compteur_ordinal ne peut jamais == NULL) après JMP/JNE dans le code (par construction)
                 compteur_ordinal = compteur_ordinal->next;
         }
         interpreter_pseudo_code(compteur_ordinal); // branchement
   }
 }
}
```

limitation 1 – le branchement arrière à des labels se trouvant avant l'instruction JMP qui déclenche ce branchement n'est pas possible (*struct pseudocodenode * compteur_ordinal = **pc->next**;*)

limitation 2 – effectuer un branchement s'effectue en coût de la boucle (au pire des cas en O(n)) et peut être optimisé par un accès direct via une table de hashage des labels en O(1))

Université Mohammed V – Souissi
École Nationale Supérieure d'Informatique et d'Analyse des Systèmes

Examen

Année Universitaire : 2011 - 2012
Filière : Ingénieur
Semestre : S3
Période : P2

Module : **M3.4** - Compilation
Élément de Module : **M3.4.1** - Compilation
Professeur : Karim BAÏNA

- **Date : 12/01/2011**
- **Durée : 2H00**

Consignes aux élèves ingénieurs :

- *Le barème est donné seulement à titre indicatif !!*
- *Les **réponses directes** et **synthétiques** seront appréciées*
- *Soignez votre **présentation** et **écriture** !!*

Exercice I : Questions de cours *(10 pts)*

Concept/Question	Choix unique	Choix possibles
(**1**) Bytecode Java	(**B**)	**(A)** démontrer qu' « une grammaire est ambiguë » est décidable mais l'inverse est non décidable
(**2**) **ADDOP** REG1, REG2	**J**	(**B**) Représentation.....**Linéaire**
(**3**) Nombre de Registres nécessaires pour une expression arithmétique	**I**	(**C**) Analyseur Ascendant
(**4**) Grammaire LL	**F**	(**D**) Erreur Syntaxique
(**5**) **A**corn **R**ISC **M**achine-ARM	**U**	(**E**) **L**ook **A**head **L**eft-to-right with **R**ightmost parse
(**6**) Select * From * ;	**D**	(**F**) Analyseur Descendant
(**7**) LALR	(**E**)	(**G**) Erreur Sémantique
(**8**) *(null).suivant	**G**	(**H**) Erreur Lexicale
(**9**) Grammaire LR	**C**	(**I**) Attribut nécessaire à la génération de pseudo-code
(10) Génération de code à une adresse	**P**	(**J**) Two-address code
(**11**) Grammaire Ambiguë	**K**	(**K**) Analyse floue
(**12**) Récursivité Gauche	**N**	(**L**) Analyse descendante non optimale
(**13**) Grammaire Héréditairement-ambiguë	**Q**	(**M**) Analogie avec les epsilon-NFA
(**14**) Grammaire non LL	**L**	(**N**) Analyse sans fin
(**15**) Grammaire algébrique	**S**	(**O**) Union de deux langages hors-contexte
(**16**) Grammaire à Terminaux nullables	**M**	(**P**) Stack Machine code
(**17**) n'est pas hors-contexte	**O**	(**Q**) Analyse impossible
(**18**) $({}^{n}\{{}^{m})^{n}\}^{m}$	**R**	(**R**) Analyse hors contexte impossible
(**19**) Commentaire C avec /* sans */	**H**	(**S**) Equation linéaire
(**20**) Dérivation droite et gauche	**T**	(**T**) Tri et tri inverse des feuilles
(11) semi-décidabilité	**(A)** « ***résolue*** »	(**U**) Three-address code

Exercice II : Syntaxe et Représentations intermédiaires *(5 pts)*

1) Améliorer la grammaire LL(1) des instructions ZZ pour prendre en compte l'instruction switch entière :
switch (x) case 1 : … break ; case 20 : … break ; …. default : … break ; endswitch
On notera que le case n'est pas obligatoire, mais, le default est toujours obligatoire à la fin. On supposera que la partie lexicale est déjà réalisée pour les nouveaux terminaux nécessaires.

```
INST : IDF = ADDSUB ';'
     | IDF = TRUE ';'
     | IDF = FALSE ';'
     | if '(' idf == ADDSUB ')' then LISTE_INST IF_INSTAUX
     | print IDF ';'
     | for IDF = inumber to inumber do LISTE_INST endfor
     | switch '(' IDF ')' SWITCH_BODY endswitch

SWITCH_BODY  : case INUMBER ':'  LISTE_INST break ';' SWITCH_BODY
             | default ':' LISTE_INST break ';'
```

** une autre solution peut être de rendre SWITCH_BODY nullable et gérer le default dans la règle appelante.

```
IF_INSTAUX :  endif  | else LISTE_INST endif
```

2) Compléter le type INST pour stocker la représentation intermédiaire du switch :
Soit le nouveau type d'instruction :

```
typedef enum {AssignArith, AssignBool, IfThenArith, IfThenElseArith, PrintIdf, For, Switch} Type_INST ;

typedef struct INST {
  Type_INST typeinst;
  union  {
 //...
```

```
  // for (index:= exp_min..exp_max) loop list_inst end loop;
  struct {
    int rangvar; // indice de l'index de la boucle
    int borneinf; // l'expression borne inf
    int bornesup; // l'expression borne sup
    struct LIST_INST * forbodylinst; // for body list of instructions
  } fornode ;
   // switch ( x ) case 1 :  ... break ; case 20 : ... break ; .... default : ... break ; endswitch
   struct {
     int rangvar;              // indice de la variable du switch
     struct case *cases        // pour les cases (SWITCH_BODY), tableau dynamique non trié de couples val- liste
     struct LIST_INST * defaultbodylinst ; // la liste d'instructions par défaut du switch
   } switchnode ;
} node;
} instvalueType ;

typedef struct case {
          int value ;                          // la valeur du cas (doit être >= 0)
          struct LIST_INST * casebodylinst; // la liste d'instructions du cas
} casevaluelinst;
```

* Une autre solution est de coder le default comme dernier élément de la liste cases avec une valeur (value) impossible (négative, ex. -1). Pour cette solution, le tableau **cases** va toujours contenir au moins un couple !
** la structure **case** peut également contenir un pointeur struct case * nextcase ; (si l'on ne veut pas, à chaque découverte d'un cas, effectuer des realloc de tout le tableau **cases**, mais seulement une allocation du **nextcase**.

Exercice III : Machine Virtuelle, Génération et Interprétation de pseudo-code *(5 pts)*

Nous souhaitons optimiser le temps de branchement de l'interpréteur du pseudo-code.

Soient les nouvelles structures de stockage des représentations intermédiaires linéaires du pseudo-code

```
// structure des nom-valeur DATA
struct namevalue {
          char * name;
          double value;
};

// nouvelle structure pour les branchements
struct jump {

          char * label_name;// nom du label
          pseudocodenode * jmpto;// @ de ce label
};

// structure linéaire du pseudocode
struct pseudocodenode{
  struct pseudoinstruction first;
  struct pseudocodenode * next;
};
```

```
// nouvelle structure pour les opérandes
typedef union {
  char * var;
  double _const;
  struct jump jp;
  struct namevalue nv;
} Param;


// structure pour les pseudoinstructions 1 adress
struct pseudoinstruction{
  CODOP codop;
  Param param;
};

typedef struct pseudocodenode * pseudocode;
```

1) Compléter la nouvelle fonction interpréteur d'un pseudocode

```
// precondition pc <> NULL
void interpreter_pseudo_code(pseudocode pc){
struct pseudocodenode** next_label_adress=(struct pseudocodenode**)malloc(sizeof(struct pseudocodenode *));

 if (pc != NULL){
    interpreter_pseudo_instruction(pc->first, next_label_address);
    if (*next_label_adress == NULL) interpreter_pseudo_code(pc->next); // Il n y a pas de branchement !!
    else interpreter_pseudo_code(*next_label_address); // effectuer un branchement en O(1) si // JNE, JG ou
JMP
    }}
```

2) Compléter la fonction interpréteur d'une pseudo instruction

```
void interpreter_pseudo_instruction(struct pseudoinstruction pi, struct pseudocodenode ** next_label_adress){
  Element op1, op2 ;
  *next_label_adress = NULL;

  switch(pi.codop){
     case JNE:       op1 = depiler(VM_STACK); op2 = depiler(VM_STACK);
                     if (op1 != op2)  (*next_label_adress) = pi.param.jp.jmpto ; break;

    case JG:         op1 = depiler(VM_STACK); op2 = depiler(VM_STACK);
                     if (op1 > op2) (*next_label_adress) = pi.param.jp.jmpto ; break;

    case JMP:   (*next_label_adress) = pi.param.jp.jmpto ; break ;
     // interprétation des autres pseudo-instructions (hors scope)
  }}
```

Corrigé
Rattrapage 2012

Examen de rattrapage

Année Universitaire : 2012 - 2013
Filière : Ingénieur, **Semestre :** S3, **Période :** P2

Module : **M3.4** – Compilation
Élément de Module : **M3.4.1** - Compilation
Professeur : Karim BAÏNA

- **Date : 27/03/2012**
- **Durée : 1H**

Consignes aux élèves ingénieurs :

- *Aucun document n'est autorisé !!*
- *Le barème est donné seulement à titre indicatif !!*
- *Soignez votre **présentation** et **écriture** !!*

Exercice I : Réseaux de Concepts[1] — 8 pts

Concept/Question	Choix unique	Choix possibles
(I.**1**) Grammaire Héréditairement ambiguë	**B**	**(A)** démontrer qu'« une grammaire est ambiguë » est décidable mais l'inverse est non décidable
(I.**2**) **ADDOP** REG1, REG2	**J**	(**B**) Analyse Hors Contexte impossible
(I.**3**) Nombre de registres nécessaires pour une expression arithmétique	**I**	(**C**) Représentation …..**GRAPHIQUE**
(I.**4**) Grammaire LL	**F**	(**D**) Erreur Syntaxique
(I.**5**) **A**corn **R**ISC **M**achine-ARM	**K**	(**E**) Représentation …..**HYBRIDE**
(I.**6**) select * from * ;	**D**	(**F**) Analyse Descendante
(I.**7**) CFG	**E**	(**G**) Erreur Sémantique détectable
(I.**8**) select T1.A1 from T2 ;	**G**	(**H**) Tri et tri inverse des feuilles
(I.**9**) AST	**C**	(**I**) Attribut nécessaire à la génération de pseudo-code
(I.**10**) Dérivation droite et gauche	**H**	(**J**) two-address code
(I. **11**) Grammaire Ambiguë	**L**	(**K**) three-address code
(I. **12**) Récursivité Gauche	**N**	(**L**) Analyse floue
(I. **13**) Grammaire non LL	**M**	(**M**) Analyse descendante non optimale
(I. **14**) p = NULL ; *(p).suivant	**O**	(**N**) Analyse sans fin
(I. **15**) $(^n\{^m)^n\}^m$	**P**	(**O**) Erreur Sémantique non détectable
(I.16) semi-décidabilité	**A** ***«résolue »***	(**P**) Analyse impossible

Exercice II : Choix alternatifs — 12 pts

Concept/Question	Choix unique	Choix possibles
(II.**1**) S=<**Expression**> ::= IDF | '{' <**Fields**> '}' | <**Expression**> '.' IDF <**Fields**> ::= <**Field**> | <**Fields**> ',' <**Field**> <**Field**> ::= IDF '=' <**Expression**>	**(A)**	**(A)** . << {} << , << = **(B)** . << , << {} << = **(C)** {} << . << = << , **(D)** . >> {} >> , >> = avec >> signifie est plus prioritaire que
(II.**2**) S=<**INST**> ::= IDF ":=" <**EXPR**> | IF '(' IDF '=' <**EXPR**> ')' THEN <**LISTE_INST**> ELSE <**LISTE_INST**> ENDIF | IF '(' IDF '=' <**EXPR**> ')' THEN <**LISTE_INST**> ENDIF | PRINT IDF ;	**(B)**	**(A)** est Ambiguë **(B)** n'est pas ambiguë
(II.**3**) S=<**ADDMIN**> ::= <**ADDMIN**> '+' IDF | <**ADDMIN**> '-' IDF | IDF	**(B)**	**(A)** est LL(1) **(B)** n'est pas LL(1)
(II.**4**) le 1-address code est choisi pour sa	**(C)**	**(A)** rapidité **(B)** taille de code **(C)** portabilité
(II.**5**) la fonction de hashage h1($s\in\Sigma^*$) = $\Sigma i=1..\|s\|$ si répartit …… les identifiants s que la fonction h2($s\in\Sigma^*$) = $\Sigma i=1..\|s\|$ i * si dans la table des symboles	**(B)**	**(A)** mieux **(B)** moins bien **(C)** similairement
(II.**6**) un automate NFA est analogue à une grammaire	**(C)**	**(A)** ambiguë **(B)** avec des règle à ε **(C)** non LL(1)
(II.**7**) Le langage $L = \{a^n b^n c^m d^m\} \cup \{a^n b^m c^m d^n\}$ $n>=1$, $m>=1$ admet une grammaire	**(C)**	**(A)** non ambigüe **(B)** ambigüe mais désambiguisable **(C)** héréditairement ambigüe
(II.**8**) Les deux grammaires de starts S1/S2 S1=<**INST1**> ::= IF '(<**EXPR**>')' THEN <**INST1**> ELSE <**INST1**> | IF '(<**EXPR**>**')' THEN** <**INST1**> S2=<**INST2**> ::= IF '(<**EXPR**>')' THEN <**INST2**> ELSE <**INST**> ENDIF | IF '(<**EXPR**>')' THEN	**(B)**	**(A)** le même langage **(B)** différents langages

[1] Astuce générale : Pour chaque concept/question (de la colonne 1), remplissez la case de la colonne des choix uniques (colonne 2) correspondante par un choix qui soit le plus adéquat (de la colonne 3). Il y a des relations 1 – 1 (i.e. à chaque élément de la colonne 1 correspond 1 et 1 seul élément de la colonne 3). Le cas échéant compléter les pointillés.

<**INST2**> ENDIF donne ….		

#define N 200
#define NBS 100
int NBVAR=0;
typedef enum {false=0, true=1} boolean;
typedef struct {
char *name; int nbdecl; int order;
} varvalueType;
varvalueType TS[NBS];
 	
Compléter La fonction de recherche d'un identifiant dans la tableau des symboles
boolean inTS(char * varname, int * rangvar){
int i =0;
(II.**9**) while ((i < …….)
&& (strcmp(TS[i].name, varname) != 0)) i++;
(II.**10**) if (i == …..) return false;
(II.**11**) else { ……. = i; return true;}
}
	(D)	**(A)** sizeof(TS) **(B)** N **(C)** NBS **(D)** NBVAR

	(D)	**(A)** sizeof(TS) **(B)** N **(C)** NBS **(D)** NBVAR
	(D)	**(A)** TS[i].order **(B)** rangvar **(C)** &rangvar **(D)** *rangvar

(II.**12**) Que réalise la fonction process sur les mots du langage des alpha-numériques
typedef char * langage1 ;
void process(langage1 s){
int c, i, j;
for (i = 0, j = strlen(s)-1; i < j; i++, j--) {
c = s[i]; s[i] = s[j]; s[j] = c;
}
}
	(C)	**(A)** décale les mots à droite
		
		(B) décale les mots à gauche
		
		(C) renverse les mots
		
		(D) transforme le mot en son palindrome

(II.**13**) Que réalise la fonction apply sur le langage des numériques
typedef int langage2;
langage1 apply (langage2 X){
int i = 0;
char langange1 s[100];
langage1 result;
do s[i++] = X % 10 + '0'; X = X / 10 ;
while ((X /= 10) > 0);
s[i] = '\0';
process(s);
result = (langage1) malloc(strlen(s) + 1);
strcpy(result, s);
return result;
}
	(D)	**(A)** calcule la chaîne décimale du mot binaire
		
		(B) calcule la chaîne binaire du mot décimale
		
		(C) renvoie l'image numérique du texte représentant le mot
		
		(D) renvoie l'image textuelle du mot

(II.**14**) <**Route**> ::= <**Inst**> <**Suite**> <**Suite**> ::= ε | <**Inst**> <**Suite**> <**Inst**> ::= GO | <**Panneau**> <**Turn**> <**Turn**> ::= TL | TR <**Panneau**> ::= ε | PAN	**(B)**	(**A**) ambiguë, (**B**) LL(1), (**C**) non LL(1)
(II.**15**) Le langage $a^n b^n$ pour $n < 42^{51} - 1$	**(B)**	(**A**) infini, (**B**) suit CFG linéaire, (**C**) n'admet pas 1 DFA, (**D**) vide
(II.**16**) L'expression [-+]?[0-9]+,[0-9]* n'engendre pas	**(A)**	(**A**) 42 (**B**) 42, (**C**) 42,4 (**D**) 42,42
(II.**17**) L'expression [a-zA-Z][a-zA-Z0-9_]* n'engendre pas	**(A)**	(**A**) __STDC__ (**B**) main (**C**) eval_expr (**D**) exit_42
(II.**18**) Quel rôle ne jouent pas les représentations intermédiaires ?	**(A)**	(**A**) résolution de la surcharge, (**B**) factorisation de certaines optimisations, (**C**) décomposition en plusieurs étapes de la traduction (**D**) indépendance des parties frontales et terminales
(II.**19**) Pour une compilation vers un système embarqué, il est plus important d'avoir	**(C)**	(**A**) une grammaire compacte, (**B**) une compilation rapide, (**C**) un code optimisé, (**D**) un code riche, (**E**) un code portable
(II.**20**) le three-address code par rapport au one-address code est plus	**(A)**	(**A**) rapide, (**B**) portable, (**E**) lisible

Corrigé Exam 2013

Université Mohammed V – Souissi
École Nationale Supérieure d'Informatique et d'Analyse des Systèmes

EXAMEN

Année Universitaire : 2012 - 2013
Filière : Ingénieur
Semestre : S3 - **Période :** P2

Module : **M3.4** - Compilation
Élément de Module : **M3.4.1** - Compilation
Professeur : Karim BAÏNA

- **Date : 09/01/2013**
- **Durée : 1H30**

Consignes aux élèves ingénieurs :
- *Seule la fiche de synthèse (A4 recto/verso) est autorisée !!*
- *Le barème est donné seulement à titre indicatif !!*
- *Soignez votre **présentation** et **écriture** !!*

Exercice I : Question de cours *(5pts)*

Concept/Question	Choix unique	Choix possibles
(I. **1**) Une grammaire LL(1) est forcément récursive droite	**F**	(**V**) Vrai (**F**) Faux
(I. **2**) Une grammaire récursive gauche peut coder une associativité droite	**V**	(**V**) Vrai (**F**) Faux
(I. **3**) Une grammaire attribuée récursive droite peut produire à un attribut AST gauche	**V**	(**V**) Vrai (**F**) Faux
(I. **4**) Une grammaire attribuée récursive droite pour produire un attribut AST gauche doit être S-attribuée	**V**	(**V**) Vrai (**F**) Faux
(I. **5**) Une grammaire arithmétique LL(1) L-attribuée est structurée pour produire un attribut AST gauche	**F**	(**V**) Vrai (**F**) Faux

Exercice II : Syntaxe, Sémantique et Représentations intermédiaires *(5 pts)*

1) Soit la nouvelle grammaire LL(1) des instructions ZZ :

INST : ... | for IDF = INUMBER to INUMBER do LISTE_INST endfor | switch '(' IDF ')' SWITCH_BODY endswitch	**SWITCH_BODY** : case INUMBER ':' LISTE_INST break ';' SWITCH_BODY | default ':' LISTE_INST break ';' **IF_INSTAUX** : endif | else LISTE_INST endif

Et soit le nouveau types Type_INST et INST pour stocker la représentation intermédiaire d'une instruction :
typedef enum {AssignArith, AssignBool, IfThenArith, IfThenElseArith, PrintIdf, For, Switch} Type_INST ;

```
typedef struct INST {
  Type_INST typeinst;
  union  { //...
    // for index = nb_min..nb_max do list_inst endfor
    struct {
      int rangvar; // indice de l'index de la boucle
      int borneinf; // l'expression borne inf
      int bornesup; // l'expression borne sup
      struct LIST_INST * forbodylinst; // for body list of instructions
    } fornode ;
    // switch ( x ) case 1 :  list_inst break ; case 20 : list_inst break ; …. default : list_inst break ; endswitch
    struct {
      int rangvar;                  // indice de la variable du switch
      int nbcases ;                 // taille du tableau dynamique cases suivant
      struct case *cases            // pour les cases (SWITCH_BODY), tableau dynamique non trié de couples (value, list_inst)
      struct LIST_INST * defaultbodylinst ; // la liste d'instructions par défaut du switch
    } switchnode ;
} node;
} instvalueType ;

typedef struct case {
            int value ;                                 // la valeur du cas (doit être >= 0)
            struct LIST_INST * casebodylinst; // la liste d'instructions du case
} casevaluelinst;
```

Compléter la fonction suivante d'interprétation de la représentation intermédiaire graphique (noeud du CFG) d'une instruction :

```
void interpreter_inst(instvalueType instattribute){
  double rexp;
  switch(instattribute.typeinst){
...
  case For :
  int i ;
  for (i = instattribute.node.fornode.borneinf ;        i < = instattribute.node.fornode.bornesup                    ; i++) {
            set_valinit(instattribute.node.fornode.rangvar, i) ;
            interpreter_list_inst( instattribute.node.fornode.forbodylinst ) ;                                  }  break;
  case Switch :
  int i = 0 ;
  while( (i < instattribute.node.switchnode.nbcases) &&
         (valinit(instattribute.node.switchnode.rangvar) != instattribute.node.switchnode.cases[i].value) i++ ;

  if (i < instattribute.node.switchnode.nbcases) { interpreter_list_inst( instattribute.node.switchnode.cases[i].casebodylinst ) ; }
  else { interpreter_list_inst( instattribute.node.switchnode.defaultbodylinst ) ; }  break;
 } // fin switch
}// fin interpreter_inst
```

Exercice III : Machine Virtuelle, Génération et Interprétation de pseudo-code *(10 pts)*

Code source ZZ	Pseudo-code 1 adresse généré pour un AST d'associativité gauche (**à compléter**)	Pseudo-code 1 adresse généré pour un AST d'associativité droite (**à compléter**)
demiRayon INT 4; Perimetre DOUBLE 0; Surface DOUBLE 0.0; Pi DOUBLE 3.14; **BEGIN** Perimetre = 2.0 * Pi * 2 * demiRayon; print Perimetre; Surface = ((Pi * (4 * demiRayon)) * demiRayon); print Surface; **END**	demiRayon 4.000000 Perimetre 0.000000 Surface 0.000000 Pi 3.140000 **begin:** PUSH 2.000000 **(1 pt)** LOAD Pi MULT PUSH 2.000000 MULT LOAD demiRayon MULT STORE Perimetre LOAD Perimetre PRINT LOAD Pi **(1 pt)** PUSH 4.000000 LOAD demiRayon MULT MULT LOAD demiRayon MULT STORE Surface LOAD Surface PRINT **end:**	demiRayon 4.000000 Perimetre 0.000000 Surface 0.000000 Pi 3.140000 **begin:** PUSH 2.000000 **(1 pt)** LOAD Pi PUSH 2.000000 LOAD demiRayon MULT MULT MULT STORE Perimetre LOAD Perimetre PRINT LOAD Pi **(1 pt)** PUSH 4.000000 LOAD demiRayon MULT MULT LOAD demiRayon MULT STORE Surface LOAD Surface PRINT **end:**

Code source ZZ d'un calcul de fibonacci	Pseudo-code 1 adresse généré (**à compléter**)

REM fibonacci
 	
REM grand pere Fibo(i=0) = 1
gp int 1;
 	
REM pere Fibo(i=1) = 1
p int 1;
 	
REM petit fils
pf int 0;
 	
i int;
 	
correct bool;
 	
begin
 	
REM calcul de Fibo(i=10)
for i = 2 to 10 do
 	
pf = p + gp;
 	
gp = p;
 	
p = pf;
 	
rem print pf;
 	
endfor
 	
print pf;
 	
if (pf == 89) then
  correct = true;
else
  correct = false;
endif
 	
print correct;
 	
end
	gp 1.000000
	p 1.000000
	pf 0.000000
	i 0.000000
	correct 0.000000 **(1 pt)**
	begin:
	PUSH 2.000000
	STORE i
	for0:
	PUSH 10.000000 **(1 pt)**
	LOAD i
	JG endfor0
	LOAD p
	LOAD gp
	ADD
	STORE pf
	LOAD p
	STORE gp
	LOAD pf
	STORE p
	PUSH 1.000000
	LOAD i
	ADD **(1 pt)**
	STORE i
	JMP for0
	endfor0:
	LOAD pf
	PRINT
	PUSH 89.000000 **(1 pt)**
	LOAD pf
	JNE else1
	PUSH 1.000000
	STORE correct
	JMP endif1
	else1: **(1 pt)**
	PUSH 0.000000
	STORE correct
	endif1: **(1 pt)**
	LOAD correct
	PRINT
	end:

Corrigé des QCMs Fréquents

Control Flow Graph→(Représentation……..)
bytecode J2EE→ one-address code
Grammaire attribuée→ actions sémantiques
Grammaire LL →Analyseur descendant
Acorn RISC Machine-ARM →three-address code
select * from * ;→Erreur Syntaxique
bytecode →(Représentation……..)
select T1.A1 from T2 →Erreur Sémantique
Grammaire LR → Analyseur Ascendant
Commentaire C non fermé (/* sans */)→Erreur Lexicale
DAG→ Représentation……..
bytecode J2ME →one-address code
Grammaire LALR →Analyse Bottom-up
Terminal $t \in T$ →Classe d'expression régulière de Σ^*
Représentation intermédiaire linéaire→ Code à 2-adresses
Automate à Piles → Langage irrégulier
Grammaire régulière →Grammaire linéaire
Récursivité gauche →Bouclage du parseur LL(1)
Ambiguïté →2 Arbres syntaxiques
Erreur de parenthésage non équilibré →Analyse syntaxique
Analyse sémantique →Erreur de type
LALR →Parseur bottom-up
LL(1)→Parseur Top-down
Automate d'état finis →Langage régulier
Déterminisme →Grammaire linéaire LL(1)
Représentation intermédiaire graphique →DAG
Identificateur erroné →Analyse lexicale
lemme de l'étoile →Pompage
$A = < S, \Sigma, \delta, s_0, F >$ où $S \cap F \neq \varnothing \rightarrow \varepsilon \in L$
$card(\varepsilon\text{-fermeture}(s_0)) > 1 \rightarrow \delta(s_0, \varepsilon) = s_1$, où $s_0 \neq s_1$
typedef void * Vector ;→fermeture de Kleene
Automate d'état finis →Langage régulier
Token →lexème

card(ε-fermeture(s_0)) > 1→$\delta(s_0, \varepsilon) = s_1$, où $s_0 \neq s_1$
typedef void * Vector ;→fermeture de Kleene
Automate d'état finis →Langage régulier
Token →lexème
Système d'équations →Grammaire linéaire
ε-fermeture(s_0) \ { s_0} ≠ Ø→$\delta(s_0, \varepsilon) = s_1$, où $s_0 \neq s_1$
Problème semi-décidable → l'ambiguïté d'une grammaire
Erreur : if sans endif (en csh)→Analyse syntaxique
Erreur : /* sans */ (en C)→Analyse lexicale
Optimisation en mémoire →Minimiser un automate
L1G→Langage binaire
Java →Représentation…….Linéaire
ADDOP REG1, REG2 →Two-address code
Nombre de Registres nécessaires pour une expression arithmétique →Attribut nécessaire à la génération depseudo-code
Grammaire LL →Analyseur Descendant
LALR → Look Ahead Left-to-right with Rightmost parse
*(null).suivant →Erreur Sémantique
Grammaire LR →WAnalyseur Ascendant
Génération de code à une adresse → Stack Machine code
Grammaire Ambiguë →Analyse floue
Récursivité Gauche →Analyse sans fin
Grammaire Héréditairement-ambiguë →Analyse impossible

Concept/Question	Choix unique	Choix possibles
(1) A= < S, Σ, δ, s_0, F > où $s_0 \notin F$	I	**(a) Langage binaire**
(2) Automate à Piles	E	(b) Analyse syntaxique
(3) Système d'équations	F	(c) deux arbres syntaxiques
(4) ε-fermeture(s_0) \ { s_0} ≠ ∅	K	(d) Analyse lexicale
(5) Problème semi-décidable	j	(e) Langage hors contexte
(6) Erreur : if sans endif (en csh)	d	(f) Grammaire linéaire
(7) Automate d'état finis	G	(g) Langage régulier
(8) Erreur : /* sans */ (en C)	B	(h) Minimiser un automate
(9) Grammaire ambiguë	c	(i) ε ∉ L
(10) Optimisation en mémoire	H	(j) Vérifier l'ambiguïté d'une grammaire
(11) L1G	**(a) « résolu »**	(k) $\delta(s_0,\varepsilon) = s_1$, où $s_0 \neq s_1$

Concept/Question	Choix unique	Choix possibles
(1) Control Flow Graph	(b)	**(a)** démontrer qu' « une grammaire est ambiguë » est décidable mais l'inverse est non décidable
(2) bytecode J2EE	j	(b) Représentation……..
(3) Grammaire attribuée	i	(c) Analyseur Ascendant
(4) Grammaire LL	f	(d) Erreur Syntaxique
(5) **A**corn **R**ISC **M**achine-ARM	k	(e) Représentation……..
(6) select * from * ;	d	(f) Analyseur Descendant
(7) bytecode	(e)	(g) Erreur Sémantique
(8) select T1.A1 from T2 ;	g	(h) Erreur Lexicale
(9) Grammaire LR	c	(i) actions sémantiques
(10) Commentaire C non fermé (/* sans */)	**h**	(j) one-address code
(11) semi-décidabilité	**(a) « *RESOLUE* »**	(k) three-address code

Concept/Question	Choix unique	Choix possibles
(1) Représentation intermédiaire linéaire	Code à 2-adresses	(a) DAG
(2) Automate à Piles	Langage irrégulier	(b) Parseur bottom-up
(3) Grammaire régulière	Grammaire linéaire	(c) Assembleur
(4) Récursivité gauche	Bouclage du parseur LL(1)	(d) Langage régulier
(5) Ambiguïté	2 Arbres syntaxiques	(e) Pompage
(6) Erreur de parenthésage non équilibré	Analyse syntaxique	(f) Parseur Top-down
(7) LL(1)	Parseur Top-down	(g) Bouclage du parseur LL(1)
(8) LALR	Parseur bottom-up	(h) Langage irrégulier
(9) Analyse sémantique	Erreur de type	(i) Grammaire linéaire LL(1)
(10) Automate d'état finis	Langage régulier	(j) Analyse lexicale
(11) Déterminisme	Grammairelinéaire LL(1)	(k) Grammaire linéaire
(12) Représentation intermédiaire graphique	DAG	(l) Code à 2-adresses
(13) Identificateur erroné	Analyse lexicale	(m) Analyse syntaxique
(14) lemme de l'étoile	Pompage	(n) Erreur de type
(15) L2G	Assembleur	(o) 2 Arbres syntaxiques

Concept/Question	Choix unique	Choix possibles
(1) Représentation intermédiaire linéaire	l	(a) DAG
(2) Automate à Piles	H	(b) Parseur bottom-up
(3) Grammaire régulière		(c) Assembleur
(4) Récursivité gauche	g	(d) Langage régulier
(5) Ambiguïté	o	(e) Pompage
(6) Erreur de parenthésage non équilibré	M	(f) Parseur Top-down
(7) LL(1)	f	(g) Bouclage du parseur LL(1)
8) LALR (	b	(h) Langage irrégulier
(9) Analyse sémantique	N	(i) Grammaire linéaire LL(1)
(10) Automate d'état finis	d	(j) Analyse lexicale
(11) Déterminisme		(k) Grammaire linéaire
(12) Représentation intermédiaire graphique	a	(l) Code à 2-adresses
(13) Identificateur erroné	J	(m) Analyse syntaxique
(14) lemme de l'étoile	e	(n) Erreur de type
(15) L2G	c	(o) 2 Arbres syntaxiques

Concept/Question	Choix unique	Choix possibles
(1) $A = < S, \Sigma, \delta, s_0, F >$ où $S \cap F \neq \varnothing$	i	**(a) Assembleurs**
(2) Automate à Piles	e	//(b) Analyse syntaxique
(3) Grammaire régulière	f	//(c) Pompage
(4) card(ε-fermeture(s_0)) > 1	j	//(d) Analyse lexicale
(5) typedef void * Vector ;	k	//(e) Langage irrégulier
(6) Erreur de parenthésage non équilibré	b	//(f) Grammaire linéaire
(7) Automate d'état finis	g	//(g) Langage régulier
(8) Identificateur erroné	d	//(h) Lexème
(9) Lemme de l'étoile	c	//(i) $\varepsilon \in L$
(10) Token	**h**	//(j) $\delta(s_0, \varepsilon) = s_1$, où $s_0 \neq s_1$
(11) L2G	**(a) « Question résolue »**	(k) fermeture de Kleene

Concept/Question	Choix unique	Choix possibles
(**1**) A= < S, Σ, δ, s_0, F > où S ∩ F ≠ ∅	I	**(a) Assembleurs**
(**2**) Automate à Piles	E	(**b**) Analyse syntaxique
(**3**) Grammaire régulière	F	(**c**) Pompage
(**4**) card(ε-fermeture(s_0)) > 1	J	(**d**) Analyse lexicale
(**5**) typedef void * Vector ;	K	(**e**) Langage irrégulier
(**6**) Erreur de parenthésage non équilibré	B	(**f**) Grammaire linéaire
(**7**) Automate d'état finis	G	(**g**) Langage régulier
(**8**) Identificateur erroné	D	(**h**) Lexème
(**9**) Lemme de l'étoile	C	(**i**) ε ∈ L
(**10**) Token	H	(**j**) δ(s_0, ε)= s_1, où $s_0 \neq s_1$
(11) L2G	**(a) « Question résolue »**	(**k**) fermeture de Kleene

Exemple de A4

AST.h

```
typedef enum {NB=0, _IDF = 1, BOOLEAN = 2, OP=3}Type_Exp
typedef enum {Int, Bool, Double} Type;
typedef enum {plus, moins, mult, _div} Type_Op;
typedef enum {false, true} boolean;
struct Exp ; typedef struct Exp * AST;
typedef union {double nombre ;char *idf;boolean bool;
 struct {Type_Op top;AST expression_gauche ;
         AST expression_droite ; } op;
             } ExpValueTypeNode;
typedef struct Exp {
  Type_Exp typeexp ;Type typename;
  ExpValueTypeNode noeud ;
                  }expvalueType;
```

AST.C

- `AST arbre_gauche(AST a){return a->noeud.op.expressior`
- `Type_Op top(AST a){return a->noeud.op.top;}`
- `Type type(AST a){return a->typename;}`
- `boolean est_feuille(AST a){return(a->typeexp != OP);`
- `AST creer_feuille_booleen(boolean b){AST result result->typeexp=BOOLEAN;result->noeud.bool = b; result->typename = Bool;return result;}`

exam 2010 - 2011

<EXPBOOL> ::= **not** <EXPBOOL>
| <EXPBOOL> **and** <EXPBOOL>
| <EXPBOOL> **or** <EXPBOOL>
| <COLUMN> <OP> <COLUMN>

- **Désambiguïser la grammaire**

<EXPBOOL> ::= <OR>
<OR> ::= <OR> **or** <NOT> | <NOT>
<NOT> ::= **not** <NOT> | <AND>
<AND> ::= <AND> **and** <AUX> | <AUX>
<AUX> ::= <COLUMN> <OP> <COLUMN

- **Éliminer la récursivité à gauche**

<OR> ::= <NOT> <ORAUX>
<ORAUX> ::= **or** <NOT><ORAUX> | ε
<NOT> ::= ***(cette règle ne change pas ne pas pénaliser si répétée !!)***
<NOTAUX> ::= ***(ni ce terminal, ni cette règle n'existe !!)***
<AND> ::= <AUX> <ANDAUX>
<ANDAUX> ::= **and** <AUX> <ANDAUX> | ε
<AUX> ::= <COLUMN> <OP> <COLUMN> | (<OR>)

Rendre la grammaire LL(1)

<TABS> ::= **idf** <TABSAUX>
<TABSAUX> ::= ',' <TABS> | ε

```
typedef struct INST {
  Type_INST typeinst;
  union  { // PRINT
       struct  { int rangvar;} printnode;
          // left := right
   struct  { AST right; } assignnode;
        // IF ... THEN
   struct  { int rangvar; AST right;
             struct LIST_INST * thenlinst;
             struct LIST_INST * elselinst;
           } ifnode;
  } node;
} instvalueType;
```

```
// for (index:=exp_min..exp_max) loop list_inst end loop

struct  { int rangvar ; AST borneinf; AST bornesup;

          struct LIST_INST * forbodylinst } fornode;
```

Non-terminal	(*First*)	Les suivants (*Follow*)
<COLUMNAUX>	,	from, ';'
<POINTEDCOLUMN>	'.'	',', ';', lower, lowereq, greater, greatereq, eq, neq
<OR> **N'EST PAS NULLABLE (*) (bonifier 2 cas : (*) et ligne tableau vide OU ligne tableau correcte !!**	idf, not	';'
<NOTAUX> **N'EXISTE PAS !!**		VIDE
<ANDAUX>	and	or, ';'
<TABSAUX>	','	where, orderby, ';'

le langage ZZ n'offrant pas d'instruction d'allocation dynamique de type (malloc), le tas n'est pas géré par la mémoire virtuelle (la pile peut donc prendre toute la mémoire non consommée par le mémoire code et la mémoiredonnée (statique)).

Qu'est ce qui joue le rôle de la mémoire statique dans la programmation de cette mémoire virtuelle ?
Nous avons réutilisé la table des symbôles comme solution de gestion de la mémoire des données (statique).

QCM

Control Flow Graph→(Représentation……..)
bytecode J2EE→ one-address code
Grammaire attribuée→ actions sémantiques
Grammaire LL →Analyseur descendant
Acorn RISC Machine-ARM →three-address code
select * from * ;→Erreur Syntaxique
bytecode →(Représentation……..)
select T1.A1 from T2 →Erreur Sémantique
Grammaire LR → Analyseur Ascendant
Commentaire C non fermé (/* sans */)→Erreur Lexicale
DAG→ Représentation……..
bytecode J2ME →one-address code
Grammaire LALR →Analyse Bottom-up
Terminal t ∈ T →Classe d'expression régulière de Σ*
Représentation intermédiaire linéaire→ Code à 2-adresses
Automate à Piles → Langage irrégulier
Grammaire régulière →Grammaire linéaire
Récursivité gauche →Bouclage du parseur LL(1)
Ambiguïté →2 Arbres syntaxiques
Erreur de parenthésage non équilibré →Analyse syntaxique
Analyse sémantique →Erreur de type
LALR →Parseur bottom-up
LL(1)→Parseur Top-down
Automate d'état finis →Langage régulier
Déterminisme →Grammaire linéaire LL(1)
Représentation intermédiaire graphique →DAG
Identificateur erroné →Analyse lexicale
lemme de l'étoile →Pompage
A= < S, Σ, δ, s_0, F > où S ∩ F ≠ ∅ →ε ∈ L
card(ε-fermeture(s_0)) > 1→δ(s_0, ε)= s_1, où $s_0 \neq s_1$
typedef void * Vector ;→fermeture de Kleene
Automate d'état finis →Langage régulier
Token →lexème
Système d'équations →Grammaire linéaire
ε-fermeture(s_0) \ { s_0} ≠ ∅→δ(s_0, ε)= s_1, où $s_0 \neq s_1$
Problème semi-décidable → l'ambiguïté d'une grammaire
Erreur : if sans endif (en csh)→Analyse syntaxique
Erreur : /* sans */ (en C)→Analyse lexicale
Optimisation en mémoire →Minimiser un automate
L1G→Langage binaire
Java →Représentation…….Linéaire
ADDOP REG1, REG2 →Two-address code
Nombre de Registres nécessaires pour une expression arithmétique →Attribut nécessaire à la génération depseudo-code
Grammaire LL →Analyseur Descendant
LALR → Look Ahead Left-to-right with Rightmost parse
*(null).suivant →Erreur Sémantique
Grammaire LR →WAnalyseur Ascendant
Génération de code à une adresse → Stack Machine code
Grammaire Ambiguë →Analyse floue
Récursivité Gauche →Analyse sans fin
Grammaire Héréditairement-ambiguë →Analyse impossible
Grammaire non LL →Analyse descendante non optimale
Grammaire nullables →Analogie avec les epsilon-NFA
n'est pas hors-contexte →U deux langages hors-contexte
Dérivation droite et gauche →Tri et tri inverse des feuilles

exam 2009 – 2010

. Ajouter à la grammaire l'instruction d'affichage d'une chaîne de caractère
INST : PRINT string ; (avec string non terminal==*["][^\n"]*["])
. Ajouter à la grammaire l'instruction d'affectation booléenne complexe
EXPB : EXPB or EXPB | EXPB and EXPB | not EXPB | '(' EXPB ')' | TRUEFALSE | idf
3. Après l'enrichissement de la question (2)
(a) que remarquez – vous,
(b) que proposez-vous ?
(a) La grammaire devient ambiguë du fait qu'un IDF peut être dérivé à partir des non-terminaux EXPA et EXPB. ….. (*1 pt*)
(b) démarche de désambiguisation….. ….. (*1 pt*)
4. Ajouter à la grammaire la conditionnelle booléenne
INST : **if** '(' IDF '=' EXPB ')' **then** LISTE_INST **endif**
if '(' IDF '=' EXPB ')' **then** LISTE_INST **else** LISTE_INST **endif**
5. Après l'enrichissement de la question (4)
(a) que remarquez – vous,
(b) que proposez-vous ?
(a) La grammaire devient de nouveau ambiguë du fait qu'une conditionnelle if '(' IDF '=' IDF ')' **peut être dérivé à partir des instructions** :
if '(' IDF '=' EXPA ')' et : **if** '(' IDF '=' EXPB ')' **(*1 p***
(b) démarche de désambiguisation

Exam 2012

switch '(' IDF ')' SWITCH_BODY endswitch

SWITCH_BODY : case INUMBER ':' LISTE_INST break ';' SWITCH_BODY
| default':' LISTE_INST break ';'

AST

```
struct {
int rangvar;
struct case *cases ;
struct LIST_INST * defaultbodylinst ;
    } switchnode ;
```

En voici 6 erreurs sémantiques nouvelles (toutes *4 parmi ces 6 sont suffisantes*) :
1. Dans if '(' IDF '=' EXPB ')' la partie droite contient un identificateur non déclaré
2. Dans if '(' IDF '=' EXPB ')' la partie droite contient un identificateur déclaré d'un autre type que BOOL
3. Dans if '(' IDF '=' EXPB ')' la partie droite contient un identificateur non initialisé
4. Dans if '(' IDF '=' EXPB ')' la partie gauche est un identificateur non déclaré
5. Dans if '(' IDF '=' EXPB ')' la partie gauche est un identificateur déclaré d'un autre type que BOOL
6. Dans if '(' IDF '=' EXPB ')' la partie gauche est un identificateur non initialisé

```
// if ... then  … else booléenne
struct  {
  int rangvar; // indice de l'idf (left exp) à comparer,
  dans la table des symboles
  ASTB right; // l'expression
  booléenne (right exp) à
  comparer struct LIST_INST *
  thenlinst; // then list of
  instructions
  struct LIST_INST * elselinst; // else list of
  instructions
 } ifnodebool;
```

```
// PRINT string
struct  {
        char * chaine ;
} printnode;
```

```
typedef enum
{PrintIdf, PrintString,
 AssignArith, AssignBool,
IfThenArith, IfThenElseArith,
IfThenBool, IfThenElseBool
} Type_INST
```

```
void interpreter_pseudo_code_list_inst(pseudocode pc) ;
if (pc != NULL) {
// interpretation de la première l'instruction
interpreter_pseudo_code_inst( pc->first ) ;

// appel récursif sur la suite de pseudocode
interpreter_pseudo_code_list_inst( pc-> next );}
}
void
interpreter_pseudo_code_inst(pseudocodeinstructio
n pci) {
// SQUELETTE DU PROGRAMME A RAFFINER:
switch(pci.codop){
case ADD :
op1 = VM_STACK.depiler() ;op2 = VM_STACK.depiler() ;
VM_STACK.empiler(op1 + op2) ; break ;
case DIV :
op1 = VM_STACK.depiler() ; op2 =
VM_STACK.depiler() ;
VM_STACK.empiler(op1 / op2) ; break ;
case DUPL :
VM_STACK.empiler(VM_STACK.tetepile()); break ;
case JMP :
interpreter_pseudo_code_inst(
rechercher_instruction_au_label(pci, pci.param.label_name)) ;
                                    break ;
case LOAD :
op1 = VM_STACK.empiler(@pci.param.var) ;
break ;
case SWAP :
op1 = VM_STACK.depiler() ; op2 =
VM_STACK.depiler() ;
VM_STACK.empiler(op
2) ; VM_STACK.empiler(op1) ; break
;
                              ….
                              }
```

8. Nous voudrions pouvoir exprimer des comparaisons riches et les utiliser dans les affectations et les conditionnelles
On ajoutera un non-terminal COMP dérivant les comparaisons arithmétiques et booléennes en plus des règles suivantes à la grammaire :
COMP : EXPA <= EXPA | EXPA => EXPA | EXPA = EXPA | EXPB = EXPB | EXPB
INST : **idf ":="** COMP
if '(' COMP ')' **then** LISTE_INST **endif**
if '(' COMP ')' **then**
LISTE_INST **else**
LISTE_INST **endif**

```
// idf := COMP
struct  {
  int rangvar; // indice de l'idf (left exp), où il faut affecter, dans la table des symboles
  ASTCOMP right; // l'expression booléenne complexe droite (right exp) à affecter
} boolassignnode;
```

```
// if ... then  … else arithmétique et booléen
struct  {
   ASTCOMP comparison;
   struct LIST_INST * thenlinst;
  struct LIST_INST * elselinst;
} ifnode;
```

Compléter l'interpréteur de représentations intermédiaire pour prendre en compte l'instruction for

```
void interpreter_inst(instvalueType
instattribute){
switch(instattribute.typeinst){
                                  case forLoop :
set_value(instattribute.node.fornode.rangvar, instattribute.node.fornode.min) ;
// oubien TS[instattribute.node.fornode.rangvar] := instattribute.node.fornode.min

if  (get_value(instattribute.node.fornode.rangvar)  <=  instattribute.node.fornode.max)  {
// oubien if(TS[instattribute.node.fornode.rangvar]<= instattribute.node.fornode.max)
interpreter_list_inst( forbodylinst ) ;
}
```

```
// structure des nom-valeur DATA
struct namevalue { char * name ;double va };

// nouvelle structure pour les branchements struct jump {

        char * label_name;// nom du label pseudocodenode * jmpto
};

// structure linéaire du pseudocode
struct pseudocodenode{
  struct pseudoinstruction first;
  struct pseudocodenode * next;
};
```

```
// nouvelle structure pour les
opérandes typedef union {
  char * var;
  double _const;
  struct jump jp;
  struct namevalue nv;
} Param;
```

1) Compléter la nouvelle fonction interpréteur d'un pseudocode

```
// precondition pc <> NULL
void interpreter_pseudo_code(pseudocode pc){
struct pseudocodenode** next_label_adress=(struct pseudocodenode**)malloc(sizeof(struct pseudocodenode *));

 if (pc != NULL){
    interpreter_pseudo_instruction(pc->first, next_label_address);
    if (*next_label_adress == NULL) interpreter_pseudo_code(pc->next); // Il n y a pas de branchement !!
    else interpreter_pseudo_code(*next_label_address); // effectuer un branchement en O(1) si // JNE, JG ou
JMP
```

2) Compléter la fonction interpréteur d'une pseudo instruction

```
 void interpreter_pseudo_instruction(struct pseudoinstruction pi, struct pseudocodenode ** next_label_adress){ Element op1, op2 ;
   *next_label_adress = NULL;

   switch(pi.codop){
     case JNE:      op1 = depiler(VM_STACK); op2 = depiler(VM_STACK);
                        if (op1 != op2)  (*next_label_adress) = pi.param.jp.jmpto ; break;

     case JG:       op1 = depiler(VM_STACK); op2 = depiler(VM_STACK);
                        if (op1 > op2) (*next_label_adress) = pi.param.jp.jmpto ; break;

     case JMP:   (*next_label_adress) = pi.param.jp.jmpto ; break ;
```

```
Pile * VM_STACK;
void inintialiser_machine_abstraite(){VM_STACK = creer_pile();}
void interpreter_pseudo_instruction(struct pseudoinstruction pi, char ** next_label_name){
   Element op1, op2, resultat; int* rangvar ; *next_label_name = NULL;
switch(pi.codop){
   case DATA: varvalueType nv; strcpy(nv.name, pi.param.nv.name);nv.valinit = pi.param.nv.value;
              ajouter_nouvelle_variable_a_TS(nv);break;
   case ADD: op1 = depiler(VM_STACK); op2 = depiler(VM_STACK);resultat = op1 + op2;
              empiler(VM_STACK, resultat)break;
   case _DIV: op1 = depiler(VM_STACK);op2 = depiler(VM_STACK);resultat = op1 / op2;
              empiler(VM_STACK, resultat);break;
   case _MULT: op1 = depiler(VM_STACK); op2 = depiler(VM_STACK);resultat = op1 * op2;
               empiler(VM_STACK, resultat);break;
   case SUB:op1 = depiler(VM_STACK); op2 = depiler(VM_STACK);
                resultat = op1 - op2; empiler(VM_STACK, resultat);break;

    case LOAD:
              if (inTS(pi.param.var, rangvar) != true) if (debug) printf("%s n'est pas :\n", pi.param.var);
               else if (debug) printf("%s est Ã  l'indice %d :\n",pi.param.var, *rangvar);
              if (debug) printf("LOAD = %s %lf\n", pi.param.var,valinit(*rangvar));
              empiler(VM_STACK, valinit(*rangvar));break;
    case STORE: op1 = depiler(VM_STACK); inTS(pi.param.var, rangvar);set_valinit(*rangvar, op1); break;
    case DUPL: op1 = depiler(VM_STACK); empiler(VM_STACK, op1);empiler(VM_STACK, op1); break;
    case PUSH: empiler(VM_STACK, pi.param._const); break;
    case SWAP:op1 = depiler(VM_STACK); op2 = depiler(VM_STACK);empiler(VM_STACK, op1); empiler(VM_STACK, op2); break;
    case JNE: op1 = depiler(VM_STACK); op2 = depiler(VM_STACK);
             if (op1 != op2) strcpy(*next_label_name,pi.param.label_name)
             else {;} break;
    case JMP: strcpy(*next_label_name, pi.param.label_name);
    case PRNT:op1 = depiler(VM_STACK); printf("%lf", op1); break;
    case LABEL: break;
```

```
void interpreter_pseudo_code(pseudocode pc){char ** next_label_name = (char **) malloc(sizeof (char*));
if (pc != NULL){interpreter_pseudo_instruction(pc->first, next_label_name);
if (*next_label_name == NULL) interpreter_pseudo_code(pc->next); // Il n y a pas de branchement !!
else{ // JNE ou JMP ==> effectuer un branchement (O(n)
struct pseudocodenode * compteur_ordinal = pc->next;
       while ( (compteur_ordinal->first.codop != LABEL) ||
              (strcmp(compteur_ordinal->first.param.label_name, *next_label_name) != 0) ) {
              compteur_ordinal = compteur_ordinal->next;}
       interpreter_pseudo_code(compteur_ordinal)
```

```
typedef enum {DATA, ADD, _DIV, DUPL, JMP, JNE, LABEL, LOAD, _MULT, POP, PRNT, PUSH, SUB, STORE, SWAP} CODOP;
typedef struct pseudocodenode * pseudocode;
struct namevalue {char * name;double value;};
typedef union {char * var;double _const;char * label_name;
               struct namevalue nv;}Param;
struct pseudoinstruction{CODOP codop;Param param;};
struct pseudocodenode{struct pseudoinstruction first;
```

```
void  afficher_pseudo_code(pseudocode
pc){if (pc != NULL){
    afficher_pseudo_instruction(pc->first);
    afficher_pseudo_code(pc->next);}}
```

```
void inserer_code_en_queue(pseudocode  pc1, pseudocode pc2){
if (debug){afficher_pseudo_code(pc1); afficher_pseudo_code(pc2);}
if (pc1->next == NULL) { pc1->next = pc2;}
else{pseudocode pc = pc1;while(pc->next != NULL) {pc = pc->next;}
pc->next = pc2;}
if (debug) { afficher_pseudo_code(pc1); printf("\n");}}
```

```
typedef enum {BeginExpected} SyntacticErrorType;
typedef enum {NonDeclaredVar,AlreadyDeclared,
              BadlyInitialised,IncompatibleAssignType,
              IncompatibleCompType,IncompatibleOperationType
             } SemanticErrorType;
typedef struct {char *name;int line;
                SemanticErrorType errort;
               } smerror;
typedef struct {int line;SyntacticErrorType errort;
               } sxerror;
```

```
smerror * creer_sm_erreur(SemanticErrorType et, int line
  smerror * e = (smerror*) malloc (sizeof (smerror) );
  e->name = (char *) malloc (strlen(name));
  strcpy(e->name, name); e->line = line;e->errort = et;
  return e;}
```

```
void creer_sx_erreur(SyntacticErrorType et, int line){
sxerror * e = (sxerror*) malloc (sizeof (sxerror) );
e->line = line;e->errort = et; ERSX[NBERRSX++]= e; }
```

TYPE

```
typedef enum {PrintIdf, PrintString, AssignArith, AssignBool, IfThenArith, IfThenElseArith} Type_INST ;
```

```
typedef struct {char *name;
                int  nbdecl;Type typevar;boolean
initialisation;
                double valinit;int line;
```

```
typedef struct {inline;}tokenvalueType;
```

```
typedef struct {Type typename; } typevalueType;
```

```
typedef struct {Type typename; double valinit; }constvalueType;
```

```
struct INST; // pré déclaration de la structure
                de stockage d'une instruction
struct LIST_INST;// pré déclaration dela structure
                    de stockage d'une liste d'instruction
```

```
typedef struct INST {Type_INST typeinst;
  union  {// PRINT idftoprint
    struct  {int rangvar; } printnode;
    // left := right
    struct  {int rangvar;AST right;} assignnode;
    // IF ... THEN
    struct  {int rangvar;AST right;
             struct LIST_INST * thenlinst;
             struct LIST_INST * elselinst;} ifnode;
  } node;
} instvalueType;
typedef struct LIST_INST {struct INST first;
     struct LIST_INST * next;} listinstvalueType;
typedef union {
  varvalueType varattribute;
  constvalueType constattribute;
  Type typename;
  instvalueType instattribute;
  listinstvalueType listinstattribute;} valueType;
```

```
instvalueType* creer_instruction_print(int rangvar){
instvalueType * printinstattribute = (instvalueType *) malloc
(sizeof(instvalueType));
 printinstattribute->typeinst = PrintIdf;
printinstattribute->node.printnode.rangvar = rangvar;
return printinstattribute;}
```

```
instvalueType* creer_instruction_affectation(int rangvar, AST *
past){instvalueType * pinstattribute = (instvalueType *) malloc
(sizeof(instvalueType));
pinstattribute->typeinst =
(type(*past)==Bool)?AssignBool:AssignArith;
       pinstattribute->node.assignnode.rangvar = rangvar;
       pinstattribute->node.assignnode.right = * past;
       return pinstattribute;
}
```

```
instvalueType* creer_instruction_if(int rangvar,AST* past,listinstvalueType *plistthen,listinstvalueType * plistelse){
instvalueType * pinstattribute = (instvalueType *) malloc (sizeof(instvalueType));
pinstattribute->typeinst = ((plistelse != NULL)?IfThenElseArith:IfThenArith);
pinstattribute->node.ifnode.rangvar = rangvar;
pinstattribute->node.ifnode.right = * past;
pinstattribute->node.ifnode.thenlinst = plistthen;
pinstattribute->node.ifnode.elselinst = plistelse;
return pinstattribute;}
```