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))