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 ;
        ….
        }
}
```