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;}
```