Examen de Compilation

Document non autorisés

Partie I : QCM sur 6 pts

Question 1

Une action sémantique est un traitement qui permet de :

a. Un traitement qui permet de vérifier une règle sémantique.
b. Une procédure qui implémente une règle syntaxique.
c. Un traitement qui permet de générer des Mnémoniques de Pcode.
d. Un traitement qui permet de passer l'allocation mémoire.

Question 2

L'outil **Jflex** prend en entrée :

a. des expressions grammaticales
b. des lexèmes.
c. des expressions régulières.
d. le langage de programmation Java.

Question 3

La phrase « **toute variable utilisé doit être déclarer »**

a. est une règle syntaxique.
b. est une règle sémantique.
c. est une action sémantique.
d. est une règle lexicale.

Question 4

Le traitement de génération Pcode :

a. est une implémentation des règles syntaxique.
b. est une analyse lexicale.
c. est une vérification des règles sémantique.
d. est une implémentation des actions sémantique.

Question 5

En analyse sémantique :

a. les nombres sont des objets sémantiques.
b. les mots clés sont des objets sémantiques.
c. les identificateurs sont des objets sémantiques.
d. les Tokens sont des objets sémantiques.

Question 6

L'outil **Yacc** permet de :

a. transformer les Tokens en grammaire.
b. transformer une grammaire en action sémantique.
c. transformer les Tokens en analyseur lexicale.
d. transformer une grammaire en analyseur syntaxique.

Question 7

La phrase « **Réservation des emplacements mémoire pour les constantes et les variables** » **:**

a. présente une règle lexicale.
b. présente une procédure syntaxique.
c. présente une action sémantique.
d. présente une règle sémantique.

Question 8

L'outil **yacc** prend en entrée :

a. une grammaire et des règles sémantiques.
b. une grammaire et des règles syntaxiques.
c. une grammaire et des actions sémantiques.
d. une grammaire et des Tokens.

Partie II : Exercices (sur 9 points)

Enoncé :

On continue toujours dans le traitement de notre mini compilateur pascal et on ajoute a notre grammaire syntaxique l'instruction **Tant que ... faire** et l'instruction **Répéter.... Jusqu'à ce que.**

Pour cela, on prend la même grammaire vue dans le cours et on la modifie comme indiqué dans L'annexe I.

Les règles modifiées apparaissent en gras et elles sont :

```
TANT QUE   ::=  WHILE EXPR DO INST

RÉPÉTER    ::=  REPEAT
                INST 1;
                INST 2;
                ...ETC...
                INST N

                UNTIL COND
```

La procédure **Tant que** : permet de répéter l'instruction tant que l'expression est vraie.

Pour la procédure syntaxique **Répéter :** Les N instructions sont répétées jusqu'à ce que la condition soit vérifiée. Même si la condition est vraie dès le début, elles sont au moins exécutées une fois.

L'objectif est de faire l'analyse syntaxique et la génération de code pour ces deux nouvelles règles.
Pour COND et INST, vous supposez que les procédures associées sont définies.
On suppose que les tokens WHILE_TOKEN, DO_TOKEN, REPEAT_TOKEN, UNTIL_TOKEN déjà ajoutés. De même que les erreurs correspondantes WHILE _ERR, DO_ERR, REPEAT_ERR, UNTIL _ERR.
Toutes les méthodes non demandées sont considérées comme définies.

1- Définir la procédure syntaxique avec les actions sémantiques pour générer le code cible de la règle **TANT QUE.**
2- Définir la procédure syntaxique avec les actions sémantiques pour générer le code cible de la règle **RÉPÉTER.**

On suppose les méthodes suivantes : inst(), symbSuiv(), testeAccept(), generer1(), generer2() et les types nécessaires prédéfinies .

EXERCICES PCODE :

1- Ecrire le programme pcode généré pour l'exemple suivant, il faut présenter la pile : (**2 points)**

```
PROGRAM exam 1
VAR
  a : integer;
BEGIN
  REPEAT
    read(a)

  UNTIL a = 482;
  write(a)

END.
```

1- Ecrire le programme pcode généré pour l'exemple suivant, il faut présenter la pile : (**3 points)**

```
PROGRAM  exam 2;
VAR
  nombre, racine : REAL;
BEGIN
  read(nombre);
  racine := 0;
  WHILE racine * racine < nombre DO
    racine := racine + 0.01;
  write (nombre , racine)
END.
```

N.B : Le jeu d'instructions PCode fait l'objet de l'annexe III.

ANNEXE I : GRAMMAIRE AVEC LES NOUVELLES RÈGLE

PROGRAM ::= program ID ; BLOCK .
BLOCK ::= CONSTS VARS INSTS
CONSTS ::= const ID = NUM ; { ID = NUM ; } |ε
VARS ::= var ID { , ID } ; | ε
INSTS ::= begin INST { ; INST } end
INST ::= INSTS | AFFEC | SI | TANTQUE | ECRIRE | LIRE | POUR |ε.

RÉPÉTER ::= REPEAT INST UNTIL COND

AFFEC ::= ID := EXPR
SI ::= if COND then INST[else INST | ε]

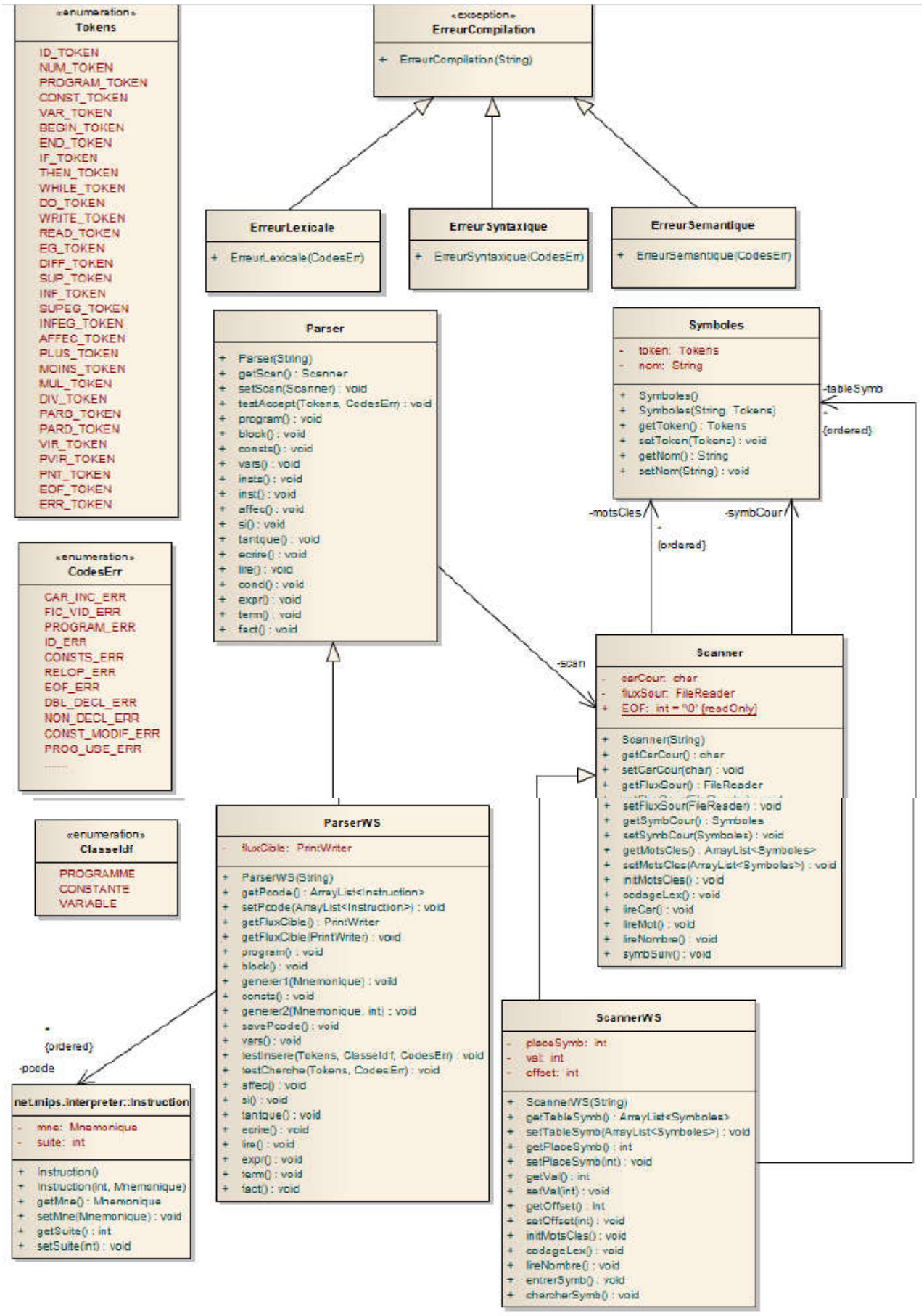
TANTQUE ::= WHILE COND DO INST

ECRIRE ::= write(EXPR { , EXPR })
LIRE ::= read(ID { , ID })
COND ::= EXPR RELOP EXPR
RELOP ::= = | <> | < | > | <= | >=
EXPR ::= TERM { ADDOP TERM }
ADDOP ::= + | -
TERM ::= FACT { MULOP FACT }
MULOP ::= * | /
FACT ::= ID | NUM | (EXPR)

ANNEXE II : LE JEU D'INSTRUCTIONS PCODE

ADD	additionne le sous-sommet de pile et le sommet, laisse le résultat au sommet (idem pour SUB, MUL, DIV)
EQL	laisse 1 au sommet de pile si sous-sommet = sommet, 0 sinon (idem pour NEQ, GTR, LSS, GEQ, LEQ)
PRN	imprime le sommet, dépile
INN	lit un entier, le stocke à l'adresse trouvée au sommet de pile, dépile
INT c	incrémente de la constante c le pointeur de pile (la constante c peut être négative)
LDI v	empile la valeur v
LDA a	empile l'adresse a
LDV	remplace le sommet par la valeur trouvée à l'adresse indiquée par le sommet (déréférence)
STO	stocke la valeur au sommet à l'adresse indiquée par le sous-sommet, dépile 2 fois
BRN i	branchement inconditionnel à l'instruction i
BZE i	branchement à l'instruction i si le sommet = 0, dépile
HLT	Halte

ANNEXE III : DIAGRAMME UML DE PROJET


«enumeration»
Tokens
ID_TOKEN
NUM_TOKEN
PROGRAM_TOKEN
CONST_TOKEN
VAR_TOKEN
BEGIN_TOKEN
END_TOKEN
IF_TOKEN
THEN_TOKEN
WHILE_TOKEN
DO_TOKEN
WRITE_TOKEN
READ_TOKEN
EG_TOKEN
DIFF_TOKEN
SUP_TOKEN
INF_TOKEN
SUPEG_TOKEN
INFEG_TOKEN
AFFEC_TOKEN
PLUS_TOKEN
MOINS_TOKEN
MUL_TOKEN
DIV_TOKEN
PARG_TOKEN
PARD_TOKEN
VIR_TOKEN
PVIR_TOKEN
PNT_TOKEN
EOF_TOKEN
ERR_TOKEN
«exception»
ErreurCompilation
+ ErreurCompilation(String)
ErreurLexicale
+ ErreurLexicale(CodesErr)
ErreurSyntaxique
+ ErreurSyntaxique(CodesErr)
ErreurSemantique
+ ErreurSemantique(CodesErr)
Parser
+ Parser(String)
+ getScan() : Scanner
+ setScan(Scanner) : void
+ testAccept(Tokens, CodesErr) : void
+ program() : void
+ block() : void
+ consts() : void
+ vars() : void
+ insts() : void
+ inst() : void
+ affec() : void
+ si() : void
+ tantque() : void
+ ecrire() : void
+ lire() : void
+ cond() : void
+ expr() : void
+ term() : void
+ fact() : void
Symboles
- token: Tokens
- nom: String
+ Symboles()
+ Symboles(String, Tokens)
+ getToken() : Tokens
+ setToken(Tokens) : void
+ getNom() : String
+ setNom(String) : void
-tableSymb
{ordered}
-motsCles
{ordered}
-symbCour
«enumeration»
CodesErr
CAR_INC_ERR
FIC_VID_ERR
PROGRAM_ERR
ID_ERR
CONSTS_ERR
RELOP_ERR
EOF_ERR
DBL_DECL_ERR
NON_DECL_ERR
CONST_MODIF_ERR
PROG_USE_ERR
-scan
Scanner
- carCour: char
- fluxSour: FileReader
+ EOF: int = '\0' {readOnly}
+ Scanner(String)
+ getCarCour() : char
+ setCarCour(char) : void
+ getFluxSour() : FileReader
+ setFluxSour(FileReader) : void
+ getSymbCour() : Symboles
+ setSymbCour(Symboles) : void
+ getMotsCles() : ArrayList<Symboles>
+ setMotsCles(ArrayList<Symboles>) : void
+ initMotsCles() : void
+ codageLex() : void
+ lireCar() : void
+ lireMot() : void
+ lireNombre() : void
+ symbSuiv() : void
«enumeration»
ClasseIdf
PROGRAMME
CONSTANTE
VARIABLE
ParserWS
- fluxCible: PrintWriter
+ ParserWS(String)
+ getPcode() : ArrayList<Instruction>
+ setPcode(ArrayList<Instruction>) : void
+ getFluxCible() : PrintWriter
+ getFluxCible(PrintWriter) : void
+ program() : void
+ block() : void
+ generer1(Mnemonique) : void
+ consts() : void
+ generer2(Mnemonique, int) : void
+ savePcode() : void
+ vars() : void
+ testInsere(Tokens, ClasseIdf, CodesErr) : void
+ testCherche(Tokens, CodesErr) : void
+ affec() : void
+ si() : void
+ tantque() : void
+ ecrire() : void
+ lire() : void
+ expr() : void
+ term() : void
+ fact() : void
*
{ordered}
-pcode
net.mips.interpreter::Instruction
- mne: Mnemonique
- suite: int
+ Instruction()
+ Instruction(int, Mnemonique)
+ getMne() : Mnemonique
+ setMne(Mnemonique) : void
+ getSuite() : int
+ setSuite(int) : void
ScannerWS
- placeSymb: int
- val: int
- offset: int
+ ScannerWS(String)
+ getTableSymb() : ArrayList<Symboles>
+ setTableSymb(ArrayList<Symboles>) : void
+ getPlaceSymb() : int
+ setPlaceSymb(int) : void
+ getVal() : int
+ setVal(int) : void
+ getOffset() : int
+ setOffset(int) : void
+ initMotsCles() : void
+ codageLex() : void
+ lireNombre() : void
+ entrerSymb() : void
+ chercherSymb() : void