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