CS 410 : Langages et Compilation

Durée : 1h45
Documents autorisés

Le barème est donné à titre indicatif et peut être amené à changer.

Exercice 1 (10 points)

Dans les formules mathématiques du langage LaTeX, une partie en indice est indiquée par le caractère souligné `'_'`. Par exemple `x_i` représente x_i. Si l'indice n'est pas réduit à un seul caractère, il faut le mettre entre accolades. Par exemple `x_{i+1}` représente x_{i+1}. Les indices peuvent être imbriqués : `x_{y_i}` représente x_{y_i}.

Dans un document HTML, un indice est mis entre les balises `<SUB>` et `</SUB>`. Par exemple, x_i s'écrit `x<SUB>i</SUB>`, x_{i+1} s'écrit `x<SUB>i+1</SUB>` et x_{y_i} s'écrit `x<SUB>y<SUB>i</SUB></SUB>`.

On considère la grammaire G suivante qui engendre les formules du langage LaTeX:

S	→	FormuleLatex `EOF`
FormuleLatex	→	ε | Primaire FormuleLatex
Primaire	→	`id` | `'_'` Indice
Indice	→	`'{'` FormuleLatex `'}'` | `id`

avec $V_T = \{$ `id`, `'_'`, `'{'`, `'}'` $\}$, où `id` représente un caractère quelconque autre que `'_'`, `'{'` et `'}'`.

Question 1 La grammaire G est-elle LL(1) ? Justifier.

Question 2 Transformer la grammaire G en une grammaire équivalente G' sans ε.

Question 3 La grammaire G' est-elle LR(0) ? Justifier.

Question 4 La grammaire G' est-elle SLR(1) ? Justifier.

Question 5 Décrire à l'aide d'attributs sur G le calcul d'une expression HTML équivalente à une formule LaTeX. On utilisera des attributs de type chaîne de caractères, et on notera "ε" la chaîne vide, et "$\cdot$" la concaténation des chaînes.

Exercice 2 (10 points)

On souhaite compiler un langage comprenant des instructions de sortie de boucle *break* et *continue*. La grammaire du langage d'entrée est la suivante :

```
instrs ::= instrs instr
| instr
;
expr ::= expr PLUS expr
| expr MULT expr
| expr > expr
```

```
| INTEGER
| BOOL
| IDF
;
instr ::= IF LPAR expr RPAR instr
| IF LPAR expr RPAR instr ELSE instr
| WHILE LPAR expr RPAR do instr
| BREAK PV
| IDF = expr PV
;
```

La représentation intermédiaire de notre compilateur vérifie la grammaire d'arbre suivante :

```
INSTRS -> Noeud_Instrs(INSTR, INSTRS) |
       Noeud_Vide

EXP -> Noeud_Plus(EXP, EXP) |
       Noeud_Mult(EXP, EXP) |
       Noeud_Sup(EXP, EXP) |
       Noeud_Integer |
       Noeud_Boolean |
       IDF

INSTR -> Noeud_If_else(EXP,INSTR,INSTR) |
         Noeud_If(EXP,INSTR) |
         Noeud_While(EXP,INSTR) |
         Noeud_Affect(IDF,EXP) |
         Noeud_Break |
         Noeud_Continue |

IDF -> Noeud_Idf
```

Une instruction *break* termine l'exécution de la boucle et passe donc au code qui suit. Une instruction *continue* termine le tour de boucle et saute directement au test de la condition d'entrée du tour suivant.

A propos des types :

- On suppose que les identificateurs sont nécessairement de type entier ou booléen.
- La somme et le produit de deux expressions doit s'effectuer nécessairement entre deux expressions de même type (entier ou booléen). Dans le cas des booléens, la somme correspond au *ou* logique et le produit au *et* logique.
- La comparaison de deux expressions est de type booléen. Elle peut s'effectuer entre deux entiers ou deux booléens.
- Les expressions dans les conditions des boucles *while* et *if* doivent être de type booléen.

Partie I : Analyse contextuelle (5 points)

Question 1 Attribuer la grammaire d'arbre de manière à ce que les expressions soient décorées de leur type.

Question 2 Pour chacun des types de noeud de l'arbre abstrait (`Noeud_Plus, ..., Noeud_if_else, ..., Noeud_Idf`, donner les vérifications contextuelles à faire pour vérifier le bon typage du programme.

Question 3 On veut également vérifier que chacune des instructions *break* et *continue* sont bien à l'intérieur d'une boucle. Attribuer la grammaire d'arbre et donner les vérifications contextuelles à faire pour vérifier la bonne utilisation de ces deux instructions.

Partie II : Génération de code (5 points)

Dans cette partie, on utilise le langage d'assemblage vu en TD.

On suppose qu'on a défini une fonction coder_Exp(Arbre A, Registre R) qui génère le code associé à l'expression d'arbre A et stocke le résultat dans le registre R. On suppose qu'on a défini les en-têtes des fonctions suivantes :

- coder_If_Else (Arbre A, [...])
- coder_If (Arbre A, [...])
- coder_While(Arbre A, [...])
- coder_Affect (Arbre A, [...])
- coder_Break(Arbre A, [...])
- coder_Continue(Arbre A, [...])

[...] signifie que vous pouvez ajouter des arguments à ces fonctions.

On se donne les même fonctions que lors des TD, à savoir toutes les méthodes permettant de connaître le type des noeuds Arbre, de récupérer des registres, de générer des instructions en langage d'assemblage, etc.

Question 4 Ecrire en pseudo-code les fonctions coder_While(Arbre A, [...]) , coder_Break(Arbre A, [...]) , ainsi que coder_Continue(Arbre A, [...]) .

Si besoin, vous pouvez décorer les noeuds de l'arbre des informations que vous souhaitez (en attribuant la grammaire d'arbre).

Vous pouvez par exemple définir un décor nommé *nombreRegistre* (que vous définissez dans votre copie) et y accéder dans votre code en écrivant : A.getDecor().nombreRegistre

Vous pouvez également rajouter des paramètres aux fonctions si besoin.

Remarque : on écrit du pseudo-code! On a donc le droit d'écrire par exemple
Registre R = getRegistre(); generer(LOAD 2(GB) R). Simplement, il faut que l'algorithme soit correct et que le pseudo-code soit compréhensible.