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.

On calcule les directeurs:
directeur(FormuleLatex $\to \varepsilon$) = Suivant(FormuleLatex) = $\{'\}', EOF\}$
directeur(FormuleLatex → Primaire FormuleLatex) = Premier(Primaire) = {`id`,`'_'`}
Les autres directeurs sont clairement disjoints. La grammaire est donc LL(1).

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

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

La nouvelle grammaire n'est pas LL(1) mais peu importe...

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

Dans l'automate LR, il est facile de voir qu'on obtient un état contenant les règles suivantes:
FormuleLatex → Primaire •
FormuleLatex → Primaire • FormuleLatex
Cet état a un conflit *shift/reduce* et le langage n'est donc pas LR(0).

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

En dessinant l'automate LR, on se rend compte qu'il y a 2 états qui posent problème.

1. FormuleLatex → Primaire •
   FormuleLatex → Primaire • FormuleLatex
   Peut-on résoudre ce conflit ? On pourrait appliquer la règle FormuleLatex → Primaire • si le caractère suivant est dans Suivant(FormuleLatex) = { EOF, '}'}. On pourrait appliquer la règle FormuleLatex → Primaire • FormuleLatex si le caractère suivant est dans Premier(FormuleLatex) = { `id`, '_'}. Le conflit est résolu.
2. Primaire → `id` •
   Indice → `id` •
   Le conflit se résout si Suivant(Primaire) et Suivant(Indice) sont disjoints. Or, Suivant(Indice) = Suivant(Primaire). La grammaire n'est donc pas SLR(1).

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.

S$\uparrow^{html}$	→	FormuleLatex$\uparrow^{html}$ `EOF`
FormuleLatex$\uparrow^{\varepsilon}$	→	ε
FormuleLatex$\uparrow^{s1.s2}$	→	Primaire$\uparrow^{s1}$ FormuleLatex$\uparrow^{s2}$
Primaire$\uparrow^{\texttt{"id"}}$	→	`id`
Primaire$\uparrow^{\texttt{"_{"}.html.\texttt{"}"}}$	→	'_' Indice$\uparrow^{html}$
Indice$\uparrow^{html}$	→	'{' FormuleLatex$\uparrow^{html}$ '}'
Indice$\uparrow^{\texttt{"id"}}$	→	`id`

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.

Les questions 1 et 2 sont corrigées simultanément.

EXP$\uparrow^t$	$\rightarrow$	Noeud_Plus(EXP$\uparrow^{t1}$, EXP$\uparrow^{t2}$)	$t = t1$, condition$(t1 = t2)$
		Noeud_Mult(EXP$\uparrow^{t1}$, EXP$\uparrow^{t2}$)	$t = t1$, condition$(t1 = t2)$
		Noeud_Sup(EXP$\uparrow^{t1}$, EXP$\uparrow^{t2}$)	$t =$`booleen`, condition$(t1 = t2)$
		Noeud_Integer	$t =$ `entier`
		Noeud_Boolean	$t =$ `booleen`
		IDF$\uparrow^{t1}$	$t = t1$
INSTR	$\rightarrow$	Noeud_If_else(EXP$\uparrow^t$,INSTR,INSTR)	
		Noeud_If(EXP$\uparrow^t$,INSTR)	condition$(t =$ `booleen`$)$
		Noeud_While(EXP$\uparrow^t$,INSTR)	condition$(t =$ `booleen`$)$
		Noeud_Affect(IDF$\uparrow^{t1}$,EXP$\uparrow^{t2}$)	condition$(t1 = t2)$
		Noeud_Break	
		Noeud_Continue	
IDF$\uparrow^t$	$\rightarrow$	Noeud_Idf	

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.

INSTRS	$\rightarrow$	Noeud_Instrs(INSTR$\downarrow_{\texttt{false}}$, INSTRS)	
		Noeud_Vide	
EXP	$\rightarrow$	Noeud_Plus(EXP, EXP)	
		Noeud_Mult(EXP, EXP)	
		Noeud_Sup(EXP, EXP)	
		Noeud_Integer	
		Noeud_Boolean	
		IDF	
INSTR$\downarrow_{boucle}$	$\rightarrow$	Noeud_If_else(EXP,INSTR$\downarrow_{boucle}$,INSTR$\downarrow_{boucle}$)	
		Noeud_If(EXP,INSTR$\downarrow_{boucle}$)	
		Noeud_While(EXP,INSTR$\downarrow_{\texttt{true}}$)	
		Noeud_Affect(IDF,EXP)	
		Noeud_Break	condition(boucle = `true`)
		Noeud_Continue	condition(boucle = `true`)
IDF	$\rightarrow$	Noeud_Idf	

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.

Il fallait voir qu'il était nécessaire de savoir ou brancher lorsqu'on atteint un `break` ou un `continue`. Il faut donc attribuer la grammaire de manière à ce que les Noeuds break et continue héritent des étiquettes auxquelles elles doivent brancher. Le code résultant est le suivant:

```
void coder_While(Arbre A) {
        Etiquette etiq_fin = getEtiquette();
        Etiquette etiq_test = getEtiquette();
        Registre R = getRegistre();
        coder_Exp(A.fils1, R);
        generer_etiquette(etiq_test);
        generer(CMP #0 R);
        generer(BEQ etiq_fin);
        coder_instr(A.fils2, etiq_fin, etiq_test);
        generer(BRA etiq_test);
        generer_etiquette(etiq_fin);
}

void coder_Break(Arbre A, Etiquette etiq_fin) {
        generer(BRA etiq_fin);
}

void coder_Continue(Arbre A, Etiquette etiq_test) {
        generer(BRA etiq_test);
}

// j'ai defini une fonction coder_instr pour simplifier coder_While
// je dois donc la coder
void coder_instr(Arbre A, Etiquette etiq_fin, Etiquette etiq_test) {
        if (is_If_Else(A)) coder_If_Else(A,etiq_fin, etiq_test);
        if (is_If(A)) coder_If(A,etiq_fin, etiq_test);
        if (is_While(A)) coder_while(A);
        if (is_Affect(A)) coder_Affect(A);
        if (is_Break(A)) coder_Break(A);
        if (is_Continue(A)) coder_Continue(A);
```

```
}
```