Examen

Documents interdits

Durée: 2h

Note Importante : il vous est demandé, dans certaines questions de cet examen, d'écrire du code en langage C. Veillez à écrire votre code de manière la plus lisible possible. Le code illisible ne sera pas lu !

1 Grammaires hors-contextes et automates à pile

L'objectif de cet exercice est de constuire un compilateur pour *grammaires hors-contexte*. Ce compilateur prend en entrée une grammaire hors-contexte g et produit un automate à pile a reconnaissant le même langage que la grammaire g.
Vous trouverez dans la partie gauche de la figure 1 une grammaire hors-contexte g_1, dans la partie centrale, l'automate a_1 lui correspondant, au format graphique, et dans la partie droite le même automate au format textuel. Le compilateur prend en entrée une grammaire hors-contexte g telle qu'elle se présente dans la partie gauche et produit l'automate a tel qu'il se présente dans la partie droite.
Une grammaire hors contexte se présente sous la forme d'une liste de règles. Une règle commence par le symbole de la partie gauche, suivi du symbole ' :' (on n'a pas utilisé la flèche pour séparer la partie gauche de la partie droite pour des questions de lisibilité), suivie des symboles de la partie droite suivis du point virgule. L'axiome est la partie gauche de la première règle. Une règle vide a une partie droite vide. Les symboles correspondent à des caractères simples (pas de chaînes de caractères). Le fichier d'entrée ne contient pas d'espace et pas de retour à la ligne.

Q.1.1. Ecrire une grammaire G qui permet de générer/reconnaître les grammaires hors-contexte g au format décrit ci dessus.

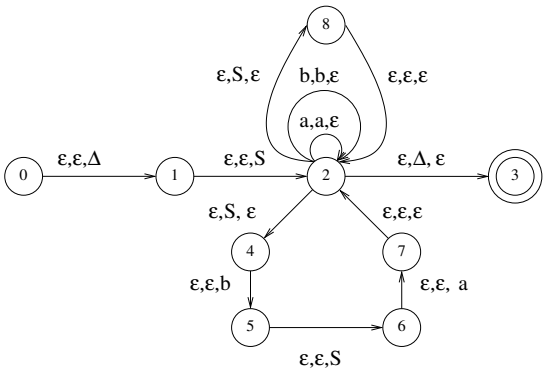
Q.1.2. Dessiner l'arbre de dérivation de la grammaire g_1 de la figure 1.

Q.1.3. Calculer les premiers et suivants de G et dessiner la table $LL(1)$. Si votre grammaire n'est pas $LL(1)$ modifiez la de manière qu'elle le devienne.

A l'issue de l'analyse syntaxique d'une grammaire, on construit un arbre abstrait la représentant. L'arbre abstrait est composé des trois types de nœuds suivants :

```
typedef struct liste_sym_ {char sym; struct liste_sym_ *suivant;} liste_sym;
typedef struct regle_ { char gauche; liste_sym *droite;} regle;
typedef struct liste_r_ { regle *regle; struct liste_r_ *suivant;} liste_regles;
```

S : $a\ S\ b$;
S : ;



(0,	1,	ε,	ε,	Δ)
(1,	2,	ε,	ε,	S)
(2,	3,	ε,	Δ ,	ε)
(2,	2,	a ,	a ,	ε)
(2,	2,	b ,	b ,	ε)
(2,	4,	ε,	S ,	ε)
(4,	5,	ε,	ε,	b)
(5,	6,	ε,	ε,	S)
(6,	7,	ε,	ε,	a)
(7,	2,	ε,	ε,	ε)
(2,	8,	ε,	S ,	ε)
(8,	2,	ε,	ε,	ε)

FIGURE 1 – Une grammaire hors-contexte et deux représentations d'un automate à pile reconnaissant le même langage

Q.1.4. Ecrire en C un analyseur syntaxique $LL(1)$ pour la grammaire G qui produit un arbre abstrait correspondant à la grammaire g analysée. Vous pourrez utiliser les fonctions suivantes : [1]

```
liste_sym *cree_liste_sym( char sym, liste_sym *suivant );
regle *cree_regle( char gauche, liste_sym *droite );
liste_regles *cree_liste_regles( regle *regle, liste_regles *suivant );
```

Un *automate à pile* se présente sous la forme d'une liste de *transitions*. Chaque transition est composée de cinq éléments : un état de départ, un état d'arrivée, un symbole à lire du mot à reconnaître, un symbole à dépiler et un symbole à empiler. Les trois derniers éléments peuvent être vides. Ils sont alors représentés par le mot vide, noté ε.

Rappels sur la construction d'un automate à pile correspondant à une grammaire hors-contexte
Soit la grammaire $g = \langle N, \Sigma, R, S\rangle$ où N est l'alphabet non terminal Σ l'alphabet terminal, R l'ensemble de règles et S l'axiome.
On peut construire un automate à pile a correspondant à une telle grammaire de la façon suivante :

1. a possède toujours les quatre états 0, 1, 2 et 3 et les trois transitions (0, 1, ε, ε, Δ), (1, 2, ε, ε, S) et (2, 3, ε, Δ, ε). Les deux premières empilent successivement le symbole de fond de pile Δ et l'axiome S, tandis que la troisième dépile le symbole de fond de pile. L'état 3 est l'unique état d'acceptation.
2. pour tout symbole terminal x, a possède une transition (2, 2, x, x, ε)
3. A toute règle $X \to x_1, \ldots x_n$ de R correspond une séquence de transitions $t_0 \ldots t_{n+1}$. La transition t_0 dépile la partie gauche de la règle. La transition t_1 empile le *dernier* symbole de la partie droite, t_2 empile l'avant dernier symbole ... La transition t_{n+1} est une transition vide qui permet de retourner à l'état 2.

1. Attention : il n'est pas démandé d'écrire les corps de ces fonctions.

Q.1.5. Ecrire la fonction `void alphabet_terminal( liste_regles *lr, char *sigma )` qui prend en paramètre la racine d'un arbre abstrait correspondant à une grammaire g et un tableau de 256 caractères. Cette fonction remplit le tableau `sigma` de la manière suivante : étant donné le caractère `x`, `sigma[x]` vaut `1` si `x` est un symbole terminal de g. Il vaut `0` sinon.

Q.1.6. Ecrire la fonction `void genere_automate( liste_regles *lr, char *sigma )` qui prend en paramètres la racine d'un arbre abstrait correspondant à une grammaire hors contexte ainsi que l'alphabet terminal de cette grammaire et génère l'automate correspondant.

2 Compilation d'un programme $L \rightarrow$ MIPS

```
1  entier $num1, entier $num2, entier $reste;
2
3  calcule( entier $a, entier $b ) {
4    retour $a - ($a / $b ) * $b;
5  }
6
7  main() {
8    $num1 = lire();
9    $num2 = lire();
10   si !( $num2 = 0 ) alors {
11     $reste = calcule($num1, $num2);
12     ecrire( $reste );
13   }
14 }
```

Q.2.1. Donner le résultat de la fonction `calcule(7,2)`. Que fait cette fonction ? Que fait le programme ?

Q.2.2. Dessiner un arbre abstrait correspondant aux lignes 8 à 13.

Q.2.3. Dessiner la table des symboles après traduction de la ligne 3.

Q.2.4. Traduire en assembleur MIPS la ligne 1.

Q.2.5. Traduire en assembleur MIPS les lignes 8 à 13, sauf la ligne 11.

Q.2.6. Traduire en assembleur MIPS la ligne 11, en expliquant les actions relatives à l'appel de fonction. Utiliser le passage d'arguments et valeur de retour par la pile du système, comme dans le projet.

Annexe : liste partielle d'opérations MIPS

`lw`	reg, adresse	charge la valeur pointée par l'adresse (4 octets)
`sw`	reg, adresse	sauvegarde le registre à l'adresse
`li`	reg, valeur	met la valeur dans le registre
`move`	reg, reg	copie le registre de droite vers celui de gauche
`not`	résultat, arg	met dans résultat la négation de arg
op	résultat, arg1, arg2	où *op* $\in$ {`add, sub, and, or, xor`}
opi	résultat, arg1, valeur	où *opi* $\in$ {`addi, andi, ori, xori`}
`mult`	arg1, arg2	met le produit dans `Hi` et `Lo`
`mflo`	reg	récupère la valeur de `Lo`
`div`	arg1, arg2	met le quotient dans `Lo`
		et le reste dans `Hi`
`j`	adresse	saute à l'adresse
`jal`	adresse	idem, enregistre l'adresse de retour dans `$ra`
`jr`	reg	saute à l'adresse contenue dans le registre
`beq`	arg1, arg2, adresse	saute si `arg1` = `arg2` ;
		même format pour {`bne, bgt, bge, blt, ble`},
		respectivement $\{\neq, >, \geq, <, \leq\}$.
`syscall`		effectue un appel système selon la valeur de `$v0` :
	si `$v0` = 1,	affiche l'entier `$a0`,
	si `$v0` = 5,	lit un entier et le met dans `$v0`,
	si `$v0` = 10,	termine le programme.