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 Compilation de graphes (14 pts)

Le but de cet exercice est de programmer un compilateur de graphes représentés dans un format textuel. Le compilateur produit une représentation en mémoire des graphes décrits. Nous utiliserons ce compilateur pour produire un Makefile à partir d'un graphe de dépendances entre fichiers, en passant par une représentation intermédiaire du graphe en mémoire.
Afin de représenter un graphe sous forme textuelle, nous utiliserons le langage dot, qui est un langage général de description de graphes dans un format textuel. On trouvera, figure 1, un exemple de description d'un graphe simple au format dot.

```
digraph mon_graphe {
     b -> d;
     a -> b ->c;
     a [label=n1];
     b -> d [color=blue];
     b [label=a1, color=blue];
 }
```

FIGURE 1 – Un exemple de description de graphe en dot

dot permet de décrire des graphes orientés ou non orientés. Nous ne nous intéresserons ici qu'au premier cas. Un graphe orienté commence par le mot clef `digraph` suivi du nom du graphe, lui même suivi d'une accolade ouvrante. On trouve ensuite la description de sommets et d'arcs. Un sommet est représenté par une chaîne de caractères ne comportant pas d'espaces et un arc par une flèche (`->`). La deuxième ligne de notre exemple représente un arc entre les deux sommets `b` et `d`. On peut aussi décrire des chemins sur une seule ligne, comme dans la ligne trois de notre exemple.
Différents attributs peuvent être associés aux sommets, comme dans la ligne quatre ou aux arcs, comme dans la ligne cinq. Dans les deux cas, les attributs sont

représentés à l'aide de crochets. La liste des attributs et de leurs valeurs n'est pas déterminée, n'importe quelle chaîne de caractère peut constituer un attribut ou une valeur. L'attribut `label` a un statut spécial, c'est lui qui permet d'attribuer une étiquette à un sommet ou à un arc. On peut aussi associer une liste d'attributs à un sommet ou à un arc, auquel cas, ils sont séparés les uns des autres par des virgules, comme dans la ligne cinq.

Q.1.1. Ecrire une grammaire hors contexte non ambiguë G permettant de générer des descriptions de graphes au format dot.

Q.1.2. Dessiner l'arbre de dérivation que G associe à notre exemple.

Q.1.3. Ecrire en langage C un analyseur pour le langage $L(G)$. On suppose que l'on dispose d'un analyseur lexical implémenté sous la forme d'une fonction `yylex`. Vous indiquerez néanmoins la liste des symboles pouvant être renvoyés par cette fonction.

On définit les structures de donnée `graphe`, `sommet`, `maillon` et `arc` en langage C permettant de représenter des graphes, des sommets, des listes de sommets ainsi que des arcs en mémoire. Un maillon d'une liste de sommets est une structure comportant deux champs, un champ `elt` qui pointe vers un `sommet` et un champ `suiv`, qui pointe vers le maillon suivant de la liste.
Ces structures de donnée sont accessibles par l'intermédiaire des fonctions suivantes :

```
graphe *cree_graphe(void);
sommet *sommet_existe(graphe *g, char *label); /* retourne le sommet
                        étiqueté label, s'il existe, et NULL sinon */
sommet *ajoute_sommet(graphe *g, char *label);
void ajoute_attribut_sommet(sommet *s, char *attribut, char *valeur);
char *valeur_attribut_sommet(sommet *s, char *attribut);
arc *ajoute_arc(graphe *g, sommet *orig, sommet *dest);
void ajoute_attribut_arc(arc *a, char *attribut, char *valeur);
maillon *liste_successeurs(sommet *s);
```

La fonction `liste_successeurs` retourne la liste des sommets que l'on peut atteindre directement à partir du `sommet` passé en argument.

Q.1.4. Ajoutez à votre analyseur des appels aux fonctions de manipulation de graphes afin de construire le graphe correspondant au fichier traité.

Un Makefile est un fichier composé de règles de dépendances. Chaque règle commence par une ligne, appelée ligne de dépendances, décrivant un but suivi par deux points (:) et une séquence (éventuellement vide) de composants (des fichiers ou d'autres buts) desquels dépend le but. Une ligne de dépendances est suivie d'une séquence, éventuellement vide, de lignes de commandes. Ces lignes décrivent un certain nombre de commande à effectuer. Elles commencent toutes par le caractère de tabulation.
La première ligne de notre exemple, figure 2 représente une règle composée uniquement d'une ligne de dépendance, elle indique que le but `all` dépend des deux buts `prog1.exe` et `prog2.exe`. La deuxième règles indique que `prog1.exe` dépend de `main1.c` et `module.o`. La commande correspondant indique comment compiler `prog1.exe`.

```
all: prog1.exe prog2.exe
prog1.exe: main1.c module.o
      gcc -o prog1.exe main1.c module.o
prog2.exe: main2.c module.o
      gcc -o prog2.exe main2.c module.o
module.o: module.c module.h
      gcc -c module.c
```

FIGURE 2 – Un exemple de Makefile

Nous allons nous servir du langage dot pour représenter des dépendances entre fichiers et générer automatiquement un Makefile à partir du fichier dot, en passant par une représentation intermédiaire sous forme de graphe, produite par votre compilateur de la question 1.4.
Dans la représentation du graphe de dépendances en dot, nous associerons aux sommets un attribut `type` qui indique la nature de l'élement associé au sommet. Cinq types seront distingués, le type `executable`, pour les fichiers exécutables , le type `source`, associé aux fichiers source, le type `header` associé aux fichiers `.h`, le type `objet` associé aux fichiers `.o` et le type `abstrait` pour les buts qui ne sont pas associés à un fichier.
Nous ajouterons aux fonctions de manipulation de graphes la fonction `sommet *tri_topologique(graphe *g)` qui effectue un tri topologique du graphe `g` et renvoie la racine du graphe. Si `g` comporte des cycles, la fonction retourne `NULL`. Cette fonction a pour but de s'assurer que le graphe de dépendances ne contient pas de cycles et retourne la racine du graphe, qui est le but de plus haut niveau.

Q.1.5. Représenter en dot les dépendances de l'exemple de la figure 2

Q.1.6. Ecrire la fonction `graph2makefile(graphe *g)` cette dernière prend comme argument un graphe de dépendances et produit un Makefile. En fonction du type des sommets, votre fonction devra générer les commandes correctes.

Attention : Un élément partagé par plusieurs buts ne sera parcouru qu'une seule fois.

2 Compilation (6 pts)

Soit le programme suivant p écrit en langage L :

```
main()
entier $a;
{$a = 0;
tantque ($a < 10) faire{
ecrire($a);
$a = $a + 1;
}}.
```

Q.2.1. Dessiner l'arbre abstrait que votre compilateur associe au programme p.

Q.2.2. Ecrire la séquence d'instructions pour la machine M produite à l'issue de la compilation du programme p. On suppose que la fonction `main` se trouve à l'adresse 5.

Annexe : Les instructions de la machine M

opcode	*opér.*	*explication*
EMPC	val.	Empile la valeur indiquée.
EMPL	adr.	Empile la valeur de la variable locale déterminée par le déplacement relatif à BEL donné par adresse (entier relatif).
DEPL	adr.	Dépile la valeur qui est au sommet et la range dans la variable locale déterminée par le déplacement relatif à BEL donné par adresse (entier relatif).
EMPG	adr.	Empile la valeur de la variable globale déterminée par le déplacement (relatif à BEG) donné par adresse.
DEPG	adr.	Dépile la valeur qui est au sommet et la range dans la variable globale déterminée par le déplacement (relatif à BEG) donné par adresse.
EMPT	adr.	Dépile la valeur qui est au sommet de la pile, soit i cette valeur. Empile la valeur de la cellule qui se trouve i cases au-delà de la variable globale déterminée par le déplacement (relatif à BEG) indiqué par adresse.
DEPT	adr.	Dépile une valeur v, puis une valeur i. Ensuite range v dans la cellule qui se trouve i cases au-delà de la variable globale déterminée par le déplacement (relatif à BEG) indiqué par adresse.
PILE	val.	Ajoute val, qui est un entier positif ou négatif, à SP
ADD		Dépile deux valeurs et empile le résultat de leur addition.
SOUS		Dépile deux valeurs et empile le résultat de leur soustraction.
MUL		Dépile deux valeurs et empile le résultat de leur multiplication.
DIV		Dépile deux valeurs et empile le quotient de leur division euclidienne.
MOD		Dépile deux valeurs et empile le reste de leur division euclidienne.
EGAL		Dépile deux valeurs et empile 1 si elles sont égales, 0 sinon.
INF		Dépile deux valeurs et empile 1 si la première est inférieure à la seconde, 0 sinon.
INFEG		Dépile deux valeurs et empile 1 si la première est inférieure ou égale à la seconde, 0 sinon.
NON		Dépile une valeur et empile 1 si elle est nulle, 0 sinon.
SAUT	adr.	L'exécution continue par l'instruction ayant l'adresse indiquée.
SIVRAI	adr.	Dépile une valeur et si elle est non nulle, l'exécution continue par l'instruction ayant l'adresse indiquée. Si la valeur dépilée est nulle, l'exécution continue normalement.
SIFAUX	adr.	Comme ci-dessus, en permutant nul et non nul.
APPEL	adr.	Empile l'adresse de l'instruction suivante, puis fait la même chose que SAUT.
RETOUR		Dépile une valeur et continue l'exécution par l'instruction dont c'est l'adresse.
ENTREE		Empile la valeur courante de BEL, puis copie la valeur de SP dans BEL.
SORTIE		Copie la valeur de BEL dans SP, puis dépile une valeur et la range dans BEL.
STOP		La machine s'arrête.
LIRE		Obtient de l'utilisateur un nombre et l'empile
ECRIV		Extrait la valeur qui est au sommet de la pile et l'affiche