Licence d'info - Compilation/génération de code

Examen du 23 mai 2017

Tous documents sont autorisés. Durée : 2 h. Barème indicatif.

---

1 Fonctions en assembleur (5 points)

1. Écrivez en assembleur `nasm` 64 bits les fonctions suivantes. Les fonctions doivent être utilisables avec du C en 64 bits. Les `int` sont codés sur 4 octets. (Par défaut les entiers sont signés en C.)
   (a) `int int_cmp(int,int)` : la fonction renvoie un nombre strictement négatif si le premier argument est plus petit que le second, 0 si les arguments sont identiques et un nombre strictement positif sinon.
   (b) `void swap(int*,int*)` : la fonction intervertit les valeurs pointées par les adresses passées en argument.
2. Faites un appel en `nasm` 64 bits pour chacune des fonctions. Les valeurs ne sont pas importantes. Les arguments de `swap()` doivent être des adresses de variables locales allouées dans la pile.
3. Que faut-il modifier si les `int` sont sur 8 octets ?

2 Traduction en bison (5 points)

Le format de fichier SubRip (fichier avec l'extension `.srt`) permet d'ajouter des sous-titres à une vidéo. Le but de cet exercice est de modifier un fichier avec un format similaire au SubRip. On a simplifié le format pour alléger la grammaire et simplifier les calculs. Le format de fichier est décrit par la grammaire suivante :

S	→	suiteST
suiteST	→	suiteST ST
	|	ST
ST	→	NUM '-' NUM TEXT

Chaque nœud ST représente un sous-titre (TEXT) précédé des temps en millisecondes entre lesquels il doit être affiché à l'écran (exemple : figure 2).

500
1000 - 2000 *On est complètement authentique*
2000 - 3000 *quand on ressemble à ce qu'on a rêvé de soi-même.*

FIGURE 1 – Fichier en entrée

1500 - 2500 *On est complètement authentique*
2500 - 3500 *quand on ressemble à ce qu'on a rêvé de soi-même.*

FIGURE 2 – Fichier en sortie

On veut un programme qui décale tous les sous-titres d'un certain temps en ms. Ce programme prendra un fichier qui commence avec le temps de décalage en ms, suivi des sous-titres. Par exemple, le programme doit accepter le contenu du fichier de la figure 1 et créer celui de la figure 2. Le programme traitera donc des fichiers dans le format suivant :

S	→	NUM suiteST
suiteST	→	suiteST ST
	|	ST
ST	→	NUM '-' NUM TEXT

Sans utiliser de variable globale, écrivez du code `bison` qui permet de créer un fichier de sous-titres où tous les temps sont décalés. On se contentera d'afficher le résultat sur le terminal. On supposera qu'un programme `flex` renvoie les lexèmes, mais on ne demande pas d'écrire le code `flex`.

3 Génération de code en bison (5 points)

Le langage Turtle est un langage de programmation qui permet de dessiner. On demande de créer un compilateur qui prend du code Turtle et qui

crée un code qui dessine la même chose dans un certain langage cible. On a simplifié le langage Turtle pour alléger la grammaire et les calculs.

Les dessins avec Turtle se font en contrôlant les mouvements d'une tortue : on peut la diriger, et elle laissera des traces sur son trajet. Pour cela, vous pouvez dire à la tortue d'avancer d'une certaine distance dans la direction vers laquelle elle regarde ou de tourner sur elle même, ce qui change la direction. La tortue ne peut regarder que dans 4 directions : en haut (3), à droite (0), en bas (1) ou à gauche (2). Au début d'un programme Turtle, la tortue regarde à droite et commence à la position $(0, 0)$. Les directives de Turtle sont les suivantes : $AVANCER\ n$, $TOURNER\_A\_GAUCHE$ et $TOURNER\_A\_DROITE$, où n est un entier qui représente la distance à parcourir. Voir la figure 3 pour du code Turtle et la 4 pour le dessin associé.

```
AVANCER 3
TOURNER_A_DROITE
AVANCER 1
TOURNER_A_GAUCHE
AVANCER 1
TOURNER_A_GAUCHE
AVANCER 2
```

FIGURE 3 – Exemple de code Turtle

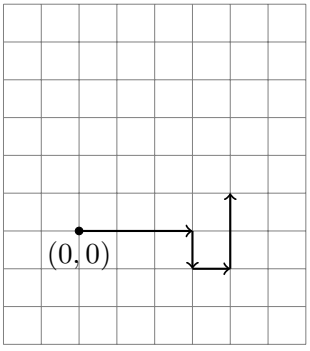

FIGURE 4 – Le dessin obtenu. Le quadrillage, le point et les flèches ne sont là qu'à titre informatif

Le langage Turtle est décrit par la grammaire suivante :

S	$\rightarrow$	suiteD
suiteD	$\rightarrow$	suiteD D
	|	D
D	$\rightarrow$	AVANCER NUM
	|	TOURNER_A_DROITE
	|	TOURNER_A_GAUCHE

Le langage cible n'a qu'une directive : $TRACE\_TRAIT\ n_1\ n_2\ n_3\ n_4$ qui trace un trait depuis le point de coordonnées (n_1, n_2) jusqu'au point de coordonnées (n_3, n_4).

$TRACE\_TRAIT$ 0 0 3 0
$TRACE\_TRAIT$ 3 0 3 -1
$TRACE\_TRAIT$ 3 -1 4 -1
$TRACE\_TRAIT$ 4 -1 4 1

FIGURE 5 – Le fichier obtenu après la compilation

Indication : lisez les 3 questions ci-dessous avant de commencer. Vous pouvez répondre en une seule fois à 2 ou 3 questions.

1. Écrivez en `bison` un compilateur qui prend du code Turtle et crée du code équivalent dans le langage cible. On se contentera d'afficher le code résultat sur le terminal. Par exemple, votre programme doit pouvoir prendre le code de la figure 3 et créer celui de la 5.
2. Le langage Turtle permet maintenant de donner une couleur au trait avec la directive $COULEUR\ n$, où n est un entier représentant une couleur. Tous les traits tracés après cette directive seront de cette couleur jusqu'à ce qu'on rechange de couleur. Au début les traits sont noirs (la valeur 0). La directive $TRACE\_TRAIT$ prend un 5^e argument entier qui représente la couleur du trait.

   Apportez toutes les modifications nécessaires à votre code pour traduire cette nouvelle directive.
3. On donne maintenant la possibilité à la tortue de décoller, ce qui permet de la déplacer sans laisser de traces. Elle peut atterrir pour recommencer à dessiner. Ces deux actions sont faites par les directives $DECOLLER$ et $ATTERRIR$ respectivement. Décoller plusieurs fois de suite ou atterrir sans avoir décollé n'ont aucun effet particulier.

   Apportez toutes les modifications nécessaires à votre code pour traduire ces nouvelles directives.

4 Notation polonaise inverse (5 points)

La notation polonaise inverse permet de représenter des opérations arithmétiques. Le but de cet exercice et de créer une calculatrice polonaise inverse. Avec la notation habituelle, les opérations arithmétiques s'écrivent de la manière suivante : $operande_1$ *operateur* $operande_2$, où *operateur* est $+$, $-$, $*$ ou $/$, et où $operande_1$ et $operande_2$ sont des nombres ou des expressions. La notation polonaise inverse, elle, commence par les opérandes puis écrit l'opérateur : $operande_1$ $operande_2$ *operateur*. Par exemple, 5 + 10 s'écrit 5 10 + en notation polonaise inverse.

Une grammaire possible pour les expressionns arithmétiques en notation polonaise inverse est :

S	→	E
E	→	E T ADDSUB
	|	T
T	→	T F DIVSTAR
	|	F
F	→	NUM

1. Quel est l'avantage de la notation polonaise inverse par rapport à la notation habituelle ?
2. Écrivez un programme `bison` qui prend une expression arithmétique en notation polonaise inverse et qui affiche son résultat sur le terminal. On ne demande pas d'écrire le programme `flex`.
3. On permet maintenant d'utiliser des constantes nommées. La grammaire devient :

S	→	DeclConst E
DeclConst	→	...
E	→	E T ADDSUB
	|	T
T	→	T F DIVSTAR
	|	F
F	→	NUM
	|	Const
Const	→	IDENT

   Écrivez un programme `bison` qui prend des déclarations de constantes éventuelles (on ne demande pas d'écrire les actions qui traduisent les déclarations) puis une expression en notation polonaise inverse, et qui produit du code en `nasm` 64 bits qui calcule la valeur de l'expression.

Les entiers seront codés sur 8 octets. Vous avez accès à une fonction `int getOffset (char *)` qui renvoie :
— 0 si l'identifiant passé en argument n'est pas déclaré, et
— son adresse relative par rapport à `rbp` s'il l'est.
L'adresse relative renvoyée est strictement négative.