TD Mips (fonctions et mémoire)

Exercice 1 (Fonctions) Traduire en Mips les fonctions suivantes, en respectant cette convention d'appel : les deux premiers arguments sont passés respectivement via les registres `$a0` et `$a1` et le résultat renvoyé via le registre `$v0`. *Indication : il vous faudra identifier les registres qui ont besoin ou non d'être sauvegardés. Le cas échéant, n'hésitez pas à adapter la stratégie générale du cours à ces cas particuliers. Autre indication : cela vaut le coup de tester votre code dans le simulateur MARS !*

1. Comptage de bits.

```
let rec compte n =
  if n = 0
  then 0
  else n mod 2 + compte (n/2)
```

2. Variante récursive terminale.

```
let rec aux n acc =
  if n = 0
  then acc
  else compte (n/2) (acc + n mod 2)

let compte n = aux n 0
```

3. Fonction 91 de McCarthy.

```
let rec f91 n =
  if n > 100
  then n - 10
  else f91(f91(n+11))
```

□

Exercice 2 (Valeurs optionnelles) On s'intéresse à des expressions admettant une forme de valeur optionnelle similaire au type option de Caml. Ainsi, Some e désigne une valeur optionnelle calculée par e, et None désigne une valeur optionnelle absente. Pour accéder à la valeur d'une option e, on se donne une construction ?e : d, qui évalue l'expression e et teste la valeur obtenue :

- si e s'évalue en Some v alors le résultat est v,
- si e s'évalue en None alors le résultat est la valeur de l'expression d (qu'on appelle le résultat par défaut).

Pour représenter de telles valeurs optionnelles en mémoire, on propose de procéder ainsi :

- La valeur `None` est représentée comme l'entier `0`.
- La valeur `Some v` est représentée par un pointeur vers un bloc alloué dans le tas, formé d'une part d'un entête contenant le nombre d'éléments dans le bloc (en l'occurrence, 1), et d'autre part d'un champ par élément du bloc (en l'occurrence, un unique champ pour l'unique élément v).

Note : la valeur `0` n'est jamais reconnue comme un pointeur valide.

Questions

1. Décrire l'état des registres et du tas après l'exécution du code suivant, et préciser la valeur contenue dans le registre `$v0`.

```
li   $a0, 8
li   $v0, 9
syscall
li   $t0, 1
sw   $t0, 0($v0)
li   $t0, 2
sw   $t0, 4($v0)
move $t0, $v0
li   $v0, 9
syscall
sw   $t0, 4($v0)
li   $t0, 1
sw   $t0, 0($v0)
```

2. Supposons que le registre `$a0` contienne la valeur `Some (Some (Some 3))`. Donner une configuration possible du tas, en précisant les adresses des blocs représentés et leur contenu, sachant que la première adresse du tas est `0x10040000`.
3. Supposons que le registre `$a0` contienne une valeur de la forme `Some (Some` n`)`. Écrire un fragment de code MIPS qui place la valeur n dans le registre `$v0`.
4. Donner le code MIPS pour une fonction f qui prend en entrée une valeur de type `int option` et est telle que :

$$\begin{aligned} f(\text{None}) &= -1 \\ f(\text{Some } n) &= n+1 \end{aligned}$$

   Le paramètre est passé par le registre `$a0` et le résultat est passé par le registre `$v0`.
5. Supposons que les registres `$a0` et `$a1` contiennent chacun une valeur de type `int option`. Écrire un fragment de code MIPS qui écrit `1` dans le registre `$v0` si les deux valeurs sont égales, et `0` sinon.

□

Exercice 3 (Exceptions) On s'intéresse à un petit langage impératif avec des mécanismes pour déclencher et rattraper des exceptions, similaires à raise et try/with en Caml ou à throw et try/catch en Java. Outre quelques formes d'instructions standards, ce langage propose donc des exceptions, chacune identifiée par un nombre entier positif. La construction

try { i }
catch k { i' }

permet d'exécuter l'instruction i tout en permettant de rattraper une éventuelle exception déclenchée pendant cette exécution.

- Si l'exécution de l'instruction i se termine sans déclencher d'exception, il ne se passe rien d'autre et on passe à l'instruction suivante.

— Si l'exécution de l'instruction i produit l'exception numéro n, alors :
  — si $n = k$ alors on exécute i',
  — sinon on propage l'exception n.

L'instruction throw(n) déclenche l'exception numéro n : les instructions suivantes sont ignorées et l'exception n est propagée au bloc try/catch englobant. Si une exception est déclenchée ou se propage en dehors de tout bloc try/catch alors l'exécution du programme est interrompue.

On s'intéresse à la compilation des mécanismes d'exceptions en suivant la stratégie dite de *stack cutting*, dans laquelle on stocke en mémoire un ensemble de gestionnaires d'exception correspondant chacun à un try/catch en cours d'exécution. Les différents gestionnaires sont chaînés entre eux, du plus récent vers le plus ancien. On maintient également un pointeur vers le gestionnaire courant (c'est-à-dire le plus récent encore actif).

Un gestionnaire d'exceptions est un bloc contenant, au minimum :
— l'adresse du gestionnaire précédent,
— le numéro de l'exception rattrapée par ce gestionnaire,
— l'adresse du code de rattrapage,
— les sauvegardes des valeurs des registres $fp et $sp au moment de la création du gestionnaire d'exceptions.

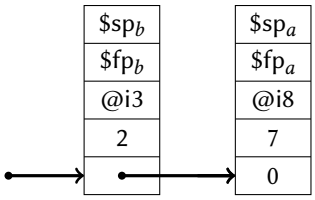
En l'absence de gestionnaire précédent, on indique par convention l'adresse 0. Les gestionnaires d'exceptions sont manipulés comme suit.
— À l'entrée dans un bloc try/catch on crée un nouveau gestionnaire g' contenant les informations de rattrapage de ce bloc et les sauvegardes de $fp et $sp. Le gestionnaire g' pointe sur l'ancien gestionnaire courant g et on met à jour le pointeur vers le gestionnaire courant.
— À la sortie d'un bloc try/catch on défausse le gestionnaire courant g'. Cela se fait par une mise à jour du pointeur vers le gestionnaire courant, qui va maintenant pointer vers le gestionnaire qui précédait g.
— Lorsqu'une exception est déclenchée on compare son numéro avec le numéro de l'exception rattrapée par le gestionnaire courant.
  — si les numéros sont égaux on restaure les valeurs sauvegardées dans le gestionnaire courant pour $fp et $sp, puis on exécute le code pointé par ce gestionnaire,
  — sinon on propage l'exception au gestionnaire précédent.

  Dans ces deux cas, on défausse de plus le gestionnaire courant.

Considérons le programme suivant.

```
try { try { i1 }
      catch 2 { i3 }
      try { i4 }
      catch 5 { i6 } }
catch 7 { i8 }
```

Voici une représentation des gestionnaires d'exceptions lors de l'exécution de l'instruction i1. On y note respectivement @i3 et @i8 les adresses des blocs de code i3 et i8, et $\$sp_a$, $\$sp_b$, $\$fp_a$ et $\$fp_b$ les valeurs sauvegardées de $sp et $fp à différents moments.



Questions

1. Parmi les candidats suivants, dire lesquels sont adaptés ou non pour stocker les gestionnaires d'exceptions, et lesquels sont adaptés ou non pour stocker le pointeur vers le gestionnaire courant. Justifier, et éventuellement préciser les particularités de certains choix possibles.
   (a) les données statiques (b) la pile (c) le tas (d) les registres
2. Dessiner des représentations des gestionnaires d'exceptions aux moments suivants :
   (a) Pendant l'exécution de i4, (b) Pendant l'exécution de i6, (c) Pendant l'exécution de i8.
3. Écrire du code MIPS réalisant les actions suivantes. Précisez quelle stratégie vous choisissez parmi celles citées à la question 1.
   (a) Défausser le gestionnaire d'exception courant.
   (b) Exécuter le code de rattrapage du gestionnaire d'exception courant (sans tester le numéro de l'exception).
   (c) Créer un nouveau gestionnaire d'exception rattrapant l'exception numéro 3 avec un code de rattrapage désigné par l'étiquette rat.
   (d) En supposant que le registre $t0 contient le numéro d'une exception, restaurer $fp et $sp et exécuter le code de rattrapage du gestionnaire d'exception courant si celui-ci porte le bon numéro (sans propager aux gestionnaires suivants si ce n'est pas le cas).
   (e) Le mécanisme complet de throw 1 (avec propagation aux gestionnaires d'exceptions aussi loin que nécessaire).
4. Décrire une stratégie permettant d'assurer qu'une exception non rattrapée interrompt proprement le programme, avec un appel système exit. *On ne demande pas de code MIPS.* □