11 + 3 + 7 = 21 → 20

Universtité Côte d'Azur
Polytech Nice Sophia - SI 4

2021 – 2022

NOM: NIGET
PRÉNOM: Tom

Programmation Fonctionnelle

23 mars 2022

Durée : 1h30

Vous apporterez un très grand soin à la présentation car elle interviendra dans la notation. Par exemple, les réponses très peu lisibles ou contenant du code non indenté ***seront considérées comme fausses.*** *Par ailleurs, la qualité du code proposé et la complexité des solutions interviendront dans la notation.* ***Documents non autorisés.***

Question 1

On veut écrire une version de la commande `make` en Scheme où les dépendances seront exprimées sous la forme de listes. Ainsi, le fichier `Makefile` suivant :

```
prog: m1.o m2.o
    gcc -o prog m1.o m2.o
m1.o: m1.c common.h m1.h
    gcc -c m1.c
m2.o: m2.c common.h m2.h
    gcc -c m2.c
all.tgz: prog
    tar cvfz all.tgz prog m1.* m2.* common.h
```

pourrait être représenté par la liste de dépendances suivante :

```
(define deps '(("prog" ("m1.o" "m2.o")
                       "gcc -o prog m1.o m2.o")
               ("m1.o" ("m1.c" "common.h" "m1.h")
                       "gcc -c m1.c")
               ("m2.o" ("m2.c" "common.h" "m2.h")
                       "gcc -c m2.c")
               ("all.tgz"  ("prog")
                       "tar cvfz all.tgz prog m1.* m2.* common.h")))
```

Notes :

1. Pour répondre à cette question, il n'est pas utile d'être un *gourou* de la commande `make`. Toutefois, au cas où vous auriez oublié comment celle-ci fonctionne, la première règle du `Makefile` indique que la construction du programme `prog` dépend des fichiers `m1.o` et de `m2.o` ; la commande permettant de le fabriquer, si l'un de ces deux fichiers a changé, est donnée dans la deuxième ligne de la règle.
2. Pour simplifier vos fonctions, vous **devez** utiliser le "vocabulaire" suivant :

   ```
   (define target       car)
   (define dependancies cadr)
   (define command      caddr)
   ```

   Ainsi, `(command (assoc "m1.o" deps))` permet de trouver la commande associée à la compilation de la cible `"m1.o"` dans la liste de dépendances `deps` et `(dependancies (car deps))` renvoie la liste `("m1.o" "m2.o")`.

Question 1.1

Écrire la fonction `(unique lst)` qui renvoie une copie de la liste `lst` sans doublon. Cette fonction renvoie les éléments dans l'ordre où ils apparaissent dans la liste originale :

```
> (unique '("m1.o" "m2.o" "prog" "prog" "all.tgz" "m1.o" "all.tgz"))
("m1.o" "m2.o" "prog" "all.tgz")
> (unique '("m1.o" "m2.o"))
("m1.o" "m2.o")
```

member plutôt

```
(define (unique lst)
  (letrec ((contains (lambda (x lst) (if (null? lst) #f
                                         (or (eq? x (car lst)) (contains x (cdr lst)))))))
    (letrec ((aux (lambda (l r)
                    (cond
                      ((null? r) l)
                      ((contains (car r) l) (aux l (cdr r)))
                      (else (aux (append l (list (car r))) (cdr r)))))))
      (aux (list) lst))))
```

ou mem si disponible

3

↑ évitable ici - préférer cons -

Question 1.2

Écrire la fonction `(direct-targets deps f)` qui permet de trouver la liste des cibles directes du fichier `f`, c'est-à-dire la liste des cibles qui dépendent directement de `f` (et qui devront donc être reconstruites si on touche au fichier `f`) :

```
(direct-targets deps "m2.c")      ⟶ ("m2.o")          ; m2.o doit être reconstruit si m2.c change
(direct-targets deps "common.h") ⟶ ("m1.o" "m2.o")    ; reconstruire m1.o et m2.o si commun.h change
```

Vous pouvez utiliser ici la fonction `(member elem lst)`.

```
(define (direct-targets deps f)
  (map target (filter (lambda (t) (member f (cdr t))) deps)))
```

dépendances

3

Rap.

Question 1.3

Écrire la fonction `(targets deps f)` qui permet de trouver la liste des dépendances du fichier `f`, c'est à dire la liste de **toutes** les cibles à reconstruire si le fichier `f` est modifié :

```
> (targets deps "m1.c")          ;; si on touche "m1.c" ...
("m1.o" "prog" "all.tgz")        ;;     ... liste des fichiers à reconstruire
> (targets deps "prog")
("all.tgz")
> (targets deps "common.h")
("m1.o" "m2.o" "prog" "all.tgz")
```

```
(define (targets deps f)
  (let ((d (direct-targets deps f)))
    (unique (apply append d (map (lambda (x) (targets deps x)) d)))))
```

Ok

Bien

3

Question 1.4

Enfin, pour finir, écrire la fonction `(build deps f)` qui permet d'afficher les commandes qui doivent être déclenchées lorsque le fichier `f` est modifié :

```
> (build deps "prog")                    ;; si on touche "prog"
tar cvfz all.tgz prog m1.* m2.* common.h  ;; commande(s) à lancer
> (build deps "common.h")
gcc -c m1.c
gcc -c m2.c
gcc -o prog m1.o m2.o
tar cvfz all.tgz prog m1.* m2.* common.h
```

```
(define (build deps f)
  (for-each displayln (map (lambda (d) (command (assoc d deps)))
                           (targets deps f))))
```

ou for-all, j'ai oublié lequel est la fonction et lequel est la macro faite en TP. bref, le "map" sans résultat.

2

Question 2

La fonction `hashmap` prend comme arguments deux listes de même longueur E et F, représentant deux ensembles $E = \{e_1, e_2, ..., e_n\}$ et $F = \{f_1, f_2, ..., f_n\}$.

L'évaluation de l'expression `(hashmap E F)` renvoie une fonction à un seul paramètre qui à tout e_k de E associe l'élément f_k de F et à tout autre objet associe la valeur booléenne *faux*.

```
(define h (hashmap '(1 2 3)  '(un deux trois)))
(h 1)                ⇒ un
(h 3)                ⇒ trois
(h 4)                ⇒ #f
```

```
(define (hashmap E F)
  (if (null? E)
      (λ(x)#f)
      (λ(x) (if (eq? x (car E))
                (car F)
                ((hashmap (cdr E) (cdr F)) x)))))
```

3

NOM: NIGET

PRÉNOM: Tom

Question 3

On désire pouvoir construire des fonctions qui acceptent des **arguments nommés**. Un argument nommé a toujours une valeur par défaut, et il peut être précisé lors de l'appel en citant son nom suivi de sa valeur (ce qui permet de passer les paramètres dans n'importe quel ordre).

Ces fonctions sont définies avec la forme spéciale `lambda-opt`. Lors de la définition d'une telle fonction, les paramètres nommés sont représentés par une liste formée d'un symbole et d'une valeur par défaut pour le paramètre (s'il n'est pas cité lors de l'appel).

```
> (define win (lambda-opt ((title "Window") (width 1000) (height 600))
                  (printf "Title=~s width=~s height=~s\n" title width height)))

> (win)                                ;; utilisation de toutes les valeurs par défaut
Title="Window" width=1000 height=600
> (win 'title "my-app")                ;; appel de win en précisant la valeur de title
Title="my-app" width=1000 height=600
> (win 'width 300 'height 100)
Title="Window" width=300 height=100
> (win 'height 400 'title "my-app" 'width 1200)
Title="my-app" width=1200 height=400
```

Question 3.1

Pour réaliser `lambda-opt`, on commence par construire la fonction `(find-value x lst default)` qui cherche la valeur associée au symbole `x` dans la liste `lst`. Si `x` apparaît dans la liste `lst`, `find-value` renvoie la valeur qui la suit immédiatement dans `lst`. Sinon, `find-value` renvoie `default`.

```
(find-value 'height (list 'title "mywin" 'height 700 'width 1000)   50)   ⟶ 700
(find-value 'height (list 'title "mywin" 'width 1000)               50)   ⟶ 50
```

Écrire la fonction `find-value` :

```
(define (find-value x lst default)
  (cond
    ((null? lst) default)
    ((eq? x (car lst)) (cadr lst))
    (else (find-value x (cddr lst) default))))
```

3

Question 3.2

1. En utilisant la fonction `find-value`, définie précédemment, écrire la macro-expansion de l'utilisation suivante de `lambda-opt` :

```
(lambda-opt ((title "Window") (width 1000) (height 600))
   (list title width height))
```

```
(lambda args
  (let ((title (find-value 'title args "Window"))
        (width (find-value 'width args 1000))
        (height (find-value 'height args 600)))
    (list title width height)))
```

2. Écrire la macro `lambda-opt`

```
(define-macro (lambda-opt params . body)
  `(lambda args
     (let ,(map (lambda (a) `(,(car a) (find-value ',(car a) args ,(cadr a))))
                params)
       ,@body)))
```