Principles of Programming Languages

Exam of 2014.07.25

Notes

Total available time: 2h.

You may use any written material you need.

You cannot use computers or phones during the exam.

GIVEN NAME _____

SURNAME _____

SIGNATURE _____

Scheme

Exercise 1.1 (4 points)

Define a procedure (called *vecstrings*) that accepts two parameters: a vector V and a list L of strings. *vecstrings* is used to put every string s in L in V, depending on its length: s is placed at position V[|s|], while strings too long are discarded. If more than one strings have the same length, they are collected in a list.

Example:

(define ex '("hi" "there" "have" "an" "interesting" "day"))

(define v1 (make-vector 7 #f))

(vecstrings v1 ex) is the vector #(#f #f (“an” “hi”) “day” “have” “there” #f)

```
(define (vecstrings V strls)
  (let ((top (- (vector-length V) 1)))
   (for-each (lambda (s)
               (let ((sl (string-length s)))
                (when (<= sl top)
                  (vector-set! V sl
                               (let ((old (vector-ref V sl)))
                                (cond
                                  ((string? old) (list s old))
                                  ((list? old)   (cons s old))
                                  (else s)))))))
             strls)
   V))
```

Exercise 1.2 (6 points)

Define the procedure *make-vecstring*, which is a variant of *vecstrings* returning a closure over V. Such closure has one parameter that must be a string *s* and works like *vecstrings*, by putting *s* in V. When the closure is called with the parameter *'return*, it must return the current value of V.

Example:

(define my-v (make-vecstring v1)) ; the definition of v1 is in Ex. 1.1

(my-v "another")

(my-v "member")

(my-v "no")

(my-v 'return) is the vector #(#f #f (“no” “an” “hi”) “day” “have” “there” “member”)

```
(define (make-vecstring V)
  (let ((top (- (vector-length V) 1)))
   (lambda (s)
     (if (eq? s 'return)
       V
       (let ((sl (string-length s)))
        (when (<= sl top)
          (vector-set! V sl
                       (let ((old (vector-ref V sl)))
                        (cond
                          ((string? old) (list s old))
                          ((list? old)   (cons s old))
                          (else s)))))))))))
```

Haskell

Exercise 2.1 (1+2+2 points)

Consider this data definition: `data Valn a = Valn a (a -> Bool)`

where *a* is a generic type, and the function: *a* -> *Bool* is a predicate that checks the validity of the stored value.

1) *Valn* cannot derive *Eq* or *Show*, why?

```
Because value equality for functions is undecidable; there is not a standard representation of
functions in Haskell.
```

2) Make *Valn* an instance of *Eq*.

```
instance Eq a => Eq (Valn a) where
  (Valn x f) == (Valn x' f') = (x == x') && (f x) == (f' x')
```

3) Make *Valn* an instance of *Show*.

```
instance Show a => Show (Valn a) where
  show (Valn x f) = "Valn "++ show x ++ " " ++ show (f x)
```

Exercise 2.2 (5 points)

Make *Valn* an instance of *Num*, considering that the predicate for two argument functions (e.g. (+)) must be the logical “and” of the two predicates; for one argument functions, say *abs*, the predicate remains the same.

```
instance Num a => Num (Valn a) where
  (Valn a f) + (Valn b g) = Valn (a+b) (\x -> (f x) && (g x))
  (Valn a f) - (Valn b g) = Valn (a-b) (\x -> (f x) && (g x))
  (Valn a f) * (Valn b g) = Valn (a*b) (\x -> (f x) && (g x))
```

```
  negate (Valn a f) = Valn (negate a)  f

  abs (Valn a f)    = Valn (abs a)     f

  signum (Valn x f) = Valn (signum x)  f

  fromInteger i = Valn (fromInteger i) (\x -> True)
```

Prolog

Exercise 3.1 (5 points)

Define the *remove* predicate, knowing that `remove(Elem, List1, List2)` is true when *List1*, with *Elem* removed, results in *List2*.

Example:

?- remove(3,[2,3,1,3],X).

X = [2, 1, 3] ; X = [2, 3, 1]

```
remove(X,[X],[]).

remove(X,[X|Xs],Xs).

remove(X,[Y|Xs],[Y|Ys]) :- remove(X,Xs,Ys).
```

Exercise 3.2 (3+1+2 points)

Consider this code:

```
proc0(L,S) :- proc1(L,S), proc2(S).


proc2([]).

proc2([_]).

proc2([X,Y|ZS]) :- X =< Y, proc2([Y|ZS]).


proc1([],[]).

proc1([X|XS],YS) :- proc1(XS,ZS), remove(X,YS,ZS).
```

1) For what can be proc0 used? What is it?

```
It is a sorting algorithm, considering all the permutations of the input list L.
```

2) Give reasonable names to proc0, proc1, proc2.

```
proc0 = permutation_sort

proc2 = sorted

proc1 = permutation
```

3) Is a good idea to use proc0 in a program? Why?

```
No, it is probably the world's possible sorting algorithm available. It is much better to use,
e.g. the quicksort implementation seen in class.
```