PPL 2020 - Moed B - Solutions

https://www.cs.bgu.ac.il/~ppl202

Q1a

```
;; Signature: drop-while(pred?, lst)
;; Type: [[T -> Boolean] * List(T) -> List(T)]
(define drop-while
  (lambda (pred? lst)
    (cond ((empty? lst) ‘())
          ((not (pred? (car lst))) lst)
          (else (drop-while pred? (cdr lst))))))
```

Grading Key:
- -1 Wrong signature
- -3 Wrong type
- -2 Wrong recursive case
- -2 No case for empty list
- -1 Returning (cdr lst) instead of lst when the predicate doesn’t hold

Q1b

```
const uncurry = <T1,T2,T3>(f:(x: T1) => (y: T2) => T3):(x: T1, y: T2) => T3
=>
    (x, y) => f(x)(y)
```

Grading Key:
- -4 Wrong type
- -6 Wrong implementation
- -1 Incorrect function declaration
  (immediately returning f(x)(y) instead of (x, y) => f(x)(y))
- -2 Use of auxiliary function not mentioned in the question
- -2 Type can be more generic

Q2a

1.

```
export interface Class {
   tag: "Class";
   args: VarDecl[];
   bindings: Binding[];
}

const evalClass = (exp: ClassExp, env: Env): Result<Class> =>
  makeOk(makeClass(exp.args, exp.bindings));


const applyClass = (cls: Class, args: Value[]): Result<Closure> => {
   const cases : LitExp[] = map((b: Binding) =>
                              makeLitExp(b.var.var), cls.bindings);
   const actions : CExp[] = map((b: Binding) => b.val, cls.bindings);
   const params : VarDecl[] = [makeVarDecl('msg')];
   const body : CExp[] = renameExps(
                         [makeCondExp(makeVarRef('msg'),cases, actions]);
   const litArgs = map(valueToLitExp,args);
   return makeOk(
      makeClosure(params,substitute(body,map((v: VarDecl) => v.var, cls.args),litArgs)));
}
```

2.

במודל הסביבות ערכי השדות נשמרים בפריים שנוצר ב applyClass

במודל ההצבה ערכי השדות נשמרים בגוף הפרוצדורה שנוצרה ב applyClass

Q2b

1.

```
const L4normalApplyProc = (op: Value, args: CExp[], env: Env): Result<Value> => {
   if (isPrimOp(op)) {
      const argVals: Result<Value[]> = mapResult((arg) => L4normalEval(arg, env), args);
      return bind(argVals, (args: Value[]) => applyPrimitive(op, args));
   } else if (isClosure(op)) {
      const vars = map((p) => p.var, op.params);
      return L4normalEvalSeq(op.body, makeExtEnv(vars, args, op.env));
   }  else if (isClass(op))
         applyClass(op, args)
```

```
  } else {
     return makeFailure(`Bad proc applied ${proc}`);
  }
};

const applyClass = (cls: Class, args: CExp[]): Result<Closure> => {
    const cases : LitExp[] = map((b: Binding) =>
        makeLitExp(b.var.var), cls.bindings);
    const actions : CExp[] = map((b: Binding) => b.val, cls.bindings);
    const params : VarDecl[] = [makeVarDecl('msg')];
    const body : CExp[] = [makeCondExp(makeVarRef('msg'),cases, actions];
    const extEnv : Env =
         makeExtEnv(map((v: VarDecl) => v.var, cls.args), args, cls.env)))
    return makeOk(makeClosure(params,body, extEnv));
}

export interface NonEmptyEnv {
  tag: "Env";
  var: string;
  val: CExp ;
  nextEnv: Env;
}

export const makeEnv = (v : string, val : CExp, env : Env): NonEmptyEnv =>
  ({tag: "Env", var: v, val: val, nextEnv: env});

export const applyEnv = (env: Env, v: string): Result<Value> =>
  isEmptyEnv(env) ? makeFailure("var not found " + v) :
  env.var === v ? makeOk(L4normalEval(env.val)) :
  applyEnv(env.nextEnv, v);
```

2.

- מקרה בו החישוב ב applicative order יעיל יותר מהחישוב ב normal order:

(define m1 (Math (+ 3 4) 5))
(m1 'square)

- מקרה בו החישוב ב normal order מסתיים בהצלחה, בעוד החישוב ב applicative order גורר שגיאת זמן ריצה:

(define m2 (Math (/ 1 0) 5))
(m2 'normalize)

- מקרה בו החישוב ב normal order מסתיים בהצלחה, בעוד החישוב ב applicative order אינו מסתיים:

```
(define m3 (Math (letrec ((f (lambda() (f)))) (f)) 5))
(m3 ‘normalize)
```

Q3a

Material about typing statements is in
https://www.cs.bgu.ac.il/~ppl202/wiki.files/class/notebook/3.1TypeChecking.html#Type-Statements
In particular, pay attention to this quote:

> The typing statement:
>
> `{f:[T1 -> T2], g:T1} |- (f g):T2`
>
> states that **for every consistent replacement of T1, T2**, under the assumption that the type of f is [T1 –> T2], and the type of g is T1, the type of (f g) is T2.

When a typing statement includes type variables, it must be true for all possible replacements of these variables.

{f:[T1->T2]} |- (f 7): T2 No - T1 could be bound to a type not compatible with Number

{f:[T->T]} |- (f x):T No - there is no assumption that warrants the fact that x is of type T.

{f:[T1->T2], g:[T2->T3], x:T1} |- (g (f x)):T3 Yes - all assumptions are met.

{g:[Empty -> T], x: T} |- (g x):T No - g does not accept arguments

{x:[T->T]} |- (lambda (x) (x x)):[T->T]
No - the expression (x x) is not well typed according to the type of x for all possible bindings of T.

Q3b

Material about this question is in
https://www.cs.bgu.ac.il/~ppl202/wiki.files/class/notebook/3.2TypeInference.html#Type-Inference-using-Type-Equations
In particular, most errors were due to the wrong application of typing rules for app-exp and if-exp.

```
(lambda (n) : boolean (if (= n 0) #f (even? (- n 1))))

Expression                                                  Variable
=============================                               ========
   1. (lambda (n) : boolean (if (= n 0) #f (even? (- n 1))))      T0
   2. (if (= n 0) #f (even? (- n 1)))                             T1
   3. (= n 0)                                                     T2
   4. #f                                                          T#f
   5. (even? (- n 1))                                             T3
   6. (- n 1)                                                     T4
   7. even?                                                       Teven?
   8. =                                                           T=
   9. -                                                           T-
   10.   n                                                           Tn
   11.   1                                                           Tnum1
   12.   0                                                           Tnum0

Expression                                  Equation
=============================               ==========================
(lambda (n) : boolean (if (= n 0) #f (even? (- n 1))))
   1.                                   T0 = [Tn -> boolean]
   2.                                   // Because of the annotation :bool
                                        T1 = boolean
   3. (if (= n 0) #f (even? (- n 1)))  T#f = T3
   4.                                   T1 = T#f
   5.                                   T2 = boolean
   6. (= n 0)                           T= = [Tn * Tnum0 -> T2]
   7. (even? (- n 1))                   Teven? = [T4 -> T3]
   8. (- n 1)                           T- = [Tn * Tnum1 -> T4]
   9. =                                 T= = [Number * Number -> Boolean]
   10.   even?                          Teven? = [Number -> Boolean]
   11.   -                              T- = [Number * Number -> Number]
   12.   1                              Tnum1 = Number
   13.   0                              Tnum0 = Number
```

Q4a
https://www.cs.bgu.ac.il/~ppl202/wiki.files/class/notebook/4.1AsyncProgramming.html#Promises-Summary

- **The type** of functions that return promises are clearer: for a synchronous function f(x:T1):T2 the corresponding Promise version will have type fp(x:T1):Promise<T2>.
  This is in contrast with a callback-based version which would have type fc(x:T1, (err:Error, res:T2)->T3): void.
- **Composition** of functions returning promises is simplified: chaining .then(handler) vs. embedded call in the callback.
- **Error handling** can be specified in a single place, as errors are cascaded through promises in a chain.

Q4b

1.
```
;; Signature: map-filter(f pred? lst)
;; Purpose: collect the values (f x) for x in l such that (pred x) is true.
;; Type: [[T1 -> T2] * [T1 -> Boolean] * List(T1) -> List(T2)] (1)
;; Example: (map-filter square even? '(1 2 3 4)) → '(4 16)
(define map-filter
  (lambda (f pred? lst)
    (cond ((empty? lst) '()) (1)
          ((pred? (car lst))
           (cons (f (car lst)) (map-filter f pred? (cdr lst)))) (1)
          (else (map-filter f pred? (cdr lst)))))) (1)
```

2.
```
;; Signature: map-filter-iter(f pred? lst)
;; Purpose: collect the values (f x) for x in l such that
;; (pred x) is true in an iterative manner
;; Example: (map-filter-iter square even? '(1 2 3 4)) → '(4 16)
(define map-filter-iter
  (lambda (f pred? lst)
    (letrec ((iter (lambda (lst acc) (1)
                     (cond ((empty? lst) acc) (1)
                           ((pred? (car lst))
                            (iter (cdr lst)
                                  (append acc (list (f (car lst))))))(1)
                           (else (iter (cdr lst) acc)))))) (1)
      (iter lst '()))))
```

3.

```
;; Signature: map-filter$(f$ pred?$ lst cont)
;; Purpose: CPS version of map-filter
;; Type: [[T1 * [T2 -> T3] -> T3] *
;;        [T1 * [Boolean -> T3] -> T3] *
;;        List(T1) *
;;        [List(T2) -> T3] -> T3] # type (1)
;; Example: (map-filter$ square$ even?$ '(1 2 3 4) id) → '(4 16)
(define map-filter$
    (lambda (f$ pred?$ lst cont)
      (cond ((empty? lst) (cont '())) ;# base-case (1)
            (else (pred?$ (car lst)
                          (lambda (pred-ok)
                            ; # True-case (2)
                            (if pred-ok
                                (f$ (car lst)
                                    (lambda (f-car)
                                      (map-filter$
                                        f$ pred?$ (cdr lst)
                                        (lambda (m-cdr)
                                          (cont (cons f-car m-cdr))))))
                                ; # false-case (2)
                                (map-filter$ f$ pred?$ (cdr lst) cont)))))))

;; Example of invocation
(define square$ (lambda (n cont) (cont (* n n))))
(define even?$ (lambda (n cont) (cont (even? n))))
(define id (lambda (x) x))
(map-filter$ square$ even?$ '(1 2 3 4) id)
```

Q5a

See material in https://www.cs.bgu.ac.il/~ppl202/wiki.files/class/notebook/5.2LogicProgramming.html#Church-Numeral-Encoding

```
sum_to(0, 0).

sum_to(s(X), Sum) :-
    sum_to(X, SumX),
    plus(s(X), SumX, Sum).
```

Q5b

See material in https://www.cs.bgu.ac.il/~ppl202/wiki.files/class/notebook/5.1RelationalLogicProgramming.html#Unification-for-Logic-Programming

1. { X = 1, Y = 2, Z = [3, 4] }
2. Fails in occurs check: X = f(g(X))
3. { X = T, T = 2, Y = [], Z = [4] }
4. { X = [1], Y = [[1]|W] }