PPL 2020 - Moed A - Solutions
https://www.cs.bgu.ac.il/~ppl202

Q1a

```
;; Signature: scan(f, init, lst)
;; Type: [[T1 * T2 -> T1] * T1 * List(T2) -> List(T1)]
(define scan
  (lambda (f init lst)
    (if (empty? lst)
        (list init)
        (cons init (scan f (f init (car lst)) (cdr lst)))))
```

Q1b

```
const zipWith = <T1, T2, T3>(f: (x: T1, y: T2) => T3, lst1: T1[], lst2: T2[]):
Result<T3[]> =>
      isEmpty(lst1) && isEmpty(lst2) ? makeOk([]) :
      isEmpty(lst1) || isEmpty(lst2) ? makeFailure("Lists are not the same length")
:
      bind(zipWith(f, rest(lst1), rest(lst2)),
           xs => makeOk(cons(f(first(lst1), first(lst2)), xs)));
```

```
Grade key :
Q1a :
-1 wrong signature
-3 wrong type
-1 partially wrong type
-2 wrong base case
-2 missing initial value at cons
-3 no base case
-1 wrong order calling f
-2 wrong recursive call
-1 duplication at the output
-2 using auxiliary functions (letrec)
-1 missing list at base case
-2 missing (cons acc (scan …
this is not scheme-> 0

Q1b :
-4 wrong type
-2 Type not generic enough
-2 mutation
-2 loops
-2 using auxiliary functions not mentioned in the question
-2 not returning Result correctly
-2 wrong base case
-2 wrong recursive call
-2 no use of bind / wrong use of bind
```

Q2.1

```
(define den 2)
(define make-pair
  (lambda (a b)
    (lambda (msg)
      (cond ((eq? msg ‘first) a)
            ((eq? msg ‘second) b)
            ((eq? msg ‘f) (/ (+ a b) den)))
     )
   )
)
```

```
Grade key :
-1 wrong in returning ‘f tag
-1 wrong in checking ‘f tag
-1 cond/if is missing
-1 return value inside a lambda
```

Q2.2

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


export const makeClassExp = (args: VarDecl[], bindings: Binding[]): ClassExp
=> {tag: "ClassExp", args: args, bindings: bindings; }

export const isClassExp = (x: any): x is ClassExp => x.tag === ‘ClassExp’;
```

```
Grade key :
-1 tag is missing
-0.5 args type is wrong
-0.5 bindings type is wrong
```

Q2.3

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

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

-1 env is not mentioned
-0.5 argos type is wrong
-0.5 bindings type is wrong
-1 no call to makeClass
-1 env is not part of makeClass params
-0.5 exp.args or exp.bindings are not part of makeClass params

Q2.4

```
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);

       return makeOk(
         makeClosure(
           [makeVarDecl('msg')],
           [makeCondExp(makeVarRef('msg'),cases, actions],
           makeExtEnv(map((v: VarDecl) => v.var, cls.args), args, cls.env)));
}
```

Grade key
Correct CondExp definition: 3 points
Correct Closure definition: 2 points
Environment definition and extension: 4 points
General: 1 point

Q2.5

ביצירת מופע של class הסביבה של המופע היא הסביבה שהיתה בזמן הגדרת ה class (בתוספת הפריים עם ערכי השדות של המופע). בקוד של סעיף א, המופע מוגדר תמיד יחד עם המחלקה, כך שלכל מופע יהיה את הסביבה הנוכחית. לא ניתן להגדיר מחלקה בנקודת זמן מסוימת ואחר כך להגדיר מופעים שונים בעלי אותה סביבה נושנה. במובן זה הצורה המיוחדת class אינה סתם קיצור תחבירי.
אך ברמה העקרונית אמרנו ש L2 היא Turing-Complete, כלומר שניתן לממש איתה כל סמנטיקה של שפות תכנות, בפרט הסמנטיקה של הגדרת מחלקה ומופעים בנפרד. כלומר, כל צורה מיוחדת מ L3 והלאה היא בהגדרה קיצור תחבירי.

Grade key

Understanding of what is a syntactic abbreviation: 2 points
Concretization of syntactic abbreviation to our case: 1 point
The issue of the environment of each instance w.r.t. the environment of the class: 1 point
The principle issue of L4 as Turing-Complete w.r.t. Syntactic abbreviation: 1 point

Q3.1 Difference between 2 methods to verify a function correctness: Precondition verification vs. Type Correctness.
Which tools are used for each / Which one is dynamic vs static.

The following definitions underlie the answer - they were not necessary as part of the answer:

a. **Preconditions** are the conditions specified as part of the function contract and that must hold for a call to a function to be valid. It is the responsibility of the caller to enforce preconditions. In the functional programming paradigm, preconditions only concern the function parameters and - if they are relevant - the value of global variables. When a function is called that violates its preconditions, the behavior of the function is unpredictable (it can throw errors or return unspecified values).
b. **Type correctness** for a function is the condition that all values passed to a function as parameters belong to the expected types; in return, the function guarantees that the value it will return belongs to the specified return type.

Differences:

a. **Preconditions can be more precise**: preconditions include the condition on the type of the parameters but they can include additional constraints which cannot be specified within the type system of the language. For example, if the type system includes a type for "numbers" - preconditions can specify tighter conditions such as "positive even integer numbers".
b. Type correctness implies guarantees both for the incoming argument values and for the type of the return value - preconditions only concern incoming arguments.
c. **Preconditions are verified by the programmer by writing code** - the responsibility is on the caller side (to verify that preconditions hold before calling the function). Precondition verifications can be added in the body of the functions as guards before the function is executed as "defensive programming" that either throw errors or return special values (like in the case of the Result style).
d. **Type correctness is enforced by the type checker** on the basis of type annotations. Programmers do not write code to ensure type correctness, instead, they define appropriate types and annotate functions.
e. In languages that have no type checker (no type annotations - for example Scheme or JavaScript), the type of the arguments must be tested by the programmer at runtime using type predicates. This implies that such "dynamic languages" (no static type checking) must provide

inspection primitives to test values at runtime (predicates like isNumber, isString etc).

f. **Preconditions are tested in a dynamic manner** (at runtime, during program execution).

g. **Type correctness is tested in a static manner** (by the type checker, based on the AST of the program, without executing the program) when the language supports type checking. Else it must be tested at runtime like all other preconditions.

Q3.2

```
type NonEmptyList = number[];
const isNonEmptyList = (x: any): x is NonEmptyList =>
      Array.isArray(x) && x.length > 0;
const first2 = (lst: NonEmptyList): number => first(lst) + first(lst);
```

The call:

```
first2([]);
```

passes type checking without error because [] is recognized by the type checker of TypeScript as a member of the NonEmptyList type - which is defined as “number[]”. (Remark: The fact that [] is recognized as a member of number[] is shown in the example at the beginning of the question.)

The fact that a user-defined type predicate (isNonEmptyList) is provided does not change the way the type checker behaves. User-defined type predicates are only used by the type checker when they are invoked by the programmer -- the type checker can infer that a parameter belongs to a type. But the type checker will NOT invoke a user defined type predicate on its own.

Q3.3

```
const first3 = (l: number[]): Result<number> =>
    isEmpty(l) ? makeFailure("empty") : makeOk(first(l) + first(l));
```

Q3.4

```
type List2 = EmptyListI2 | NonEmptyListI2;
type EmptyListI2 = [];
type NonEmptyListI2 = {first: number; rest: number[]};
```

(Note: the question in the exam had rest: List2 which is not convenient to use - I updated it to make caller4 easier to read for future generations - it had no impact on grading this year).

```
const first4 = (l: NonEmptyListI2): number => l.first + l.first;
```

Q3.5

For functions that are not type safe, the caller must verify the precondition.
For functions that are type safe, the caller must pass an appropriate value.

```
const caller1 = (l: number[]): void => {
    // Use first1
    if (! isEmpty(l)) console.log(first1(l));
    return;
}

const caller2 = (l: number[]): void => {
    // Use first2
    if (isNonEmptyList(l)) console.log(first2(l));
    return;
}
```

The way to compose Result<T> values is to use bind:

```
const caller3 = (l: number[]): Result<void> => {
    // Use first3
    return bind(first3(l), (res: number) => makeOk(console.log(res)));
}

const caller4 = (l: number[]): void => {
    // Use first4
    if (! isEmpty(l)) console.log(first4({first: first(l), rest: rest(l)}));
}
```

Q3.6
What is the recommended version of this function to write safe code?

The best version is first4 - because one cannot invoke first4 with a wrong parameter - the programmer cannot “forget” to check the precondition, else the code will not pass type checking. This is the safest version.

The second best version is first3 - in this function, the fact that the function returns a Result<T> value shows explicitly that the function is unsafe, and forces the programmer to deal with the consequences of not checking the preconditions (using bind). It is still not as safe as first4 which prevents the unsafe calls altogether.

Q4.1

Which of these are in tail form:

```
(cond ((f x) (+ x x))
      ((> x 0) 0))              No because after (f x) one must
                                continue the computation of (+ x x)

(cond ((+ x x) (f x))
      ((> x 0) (- x)))          Yes

(lambda (x) (* x (f x)))        No because the sub-expression
                                (* x (f x)) is not in tail-form
                                because after (f x) one must compute
                                (* x <res>)

(lambda (x) (f (* x x)))        Yes
```

Q4.2

```
; Signature: split$(pred$ lst c)
; Type: [[T1 * [Boolean -> T2] -> T2] * List<T1> *
;        [List<T1> * List<T1> -> T2] -> T2]
; Purpose: Returns the application of the continuation c on two lists:
; 1. A list of members for which the predicate pred$ holds.
; 2. A list of members for which it doesn't.
; Examples:
; (split$ even?$ '(1 2 3 4 5 6 7) (lambda (x y) (list x y)))
; => '((2 4 6) (1 3 5 7))
(define split$
  (lambda (pred$ lst c)
    (if (empty? lst)
        (c '() '())
        (pred$ (car lst)
            (lambda (pred_res)
                (if pred_res
                    (split$ pred$ (cdr lst)
```

```
                        (lambda (l1 l2)
                            (c (cons (car lst) l1) l2)))
                  (split$ pred$ (cdr lst)
                        (lambda (l1 l2)
                            (c l1 (cons (car lst) l2)))))))))
```

The following version was also acceptable:

```
(define split$
  (lambda (pred$ lst c)
    (if (empty? lst)
        (c lst lst)
        (split$ pred$ (cdr lst)
                (lambda (yes-lst no-lst)
                  (pred$ (car lst)
                         (lambda (res_pred)
                           (if res_pred
                               (c (cons (car lst) yes-lst) no-lst)
                               (c yes-lst (cons (car lst) no-lst))))
     ))))))
```

Q5.1

1. עשו הוא רוצח
2. אבי ושרה התחתנו
3. מי שהורג מישהו הוא רוצח
4. יצחק אוכל כל דבר שהוא טעים <u>או</u> מתוק
5. מי שנרצח מת

```
%1 killer(esau).
%2 married(avi, sarah).
%3 killer(X) :- kills(X, _).
%4 eats(itshak, X) :- tasty(X).
   eats(itshak, X) :- sweet(X).
%5 dead(X) :- kills(_, X).
```

Q5.2

1.	food(bread,X) = food(Y,sausage)	{Y=break, X=sausage}
2.	food(bread,X,beer) = food(Y,sausage,X)	Fails X cannot be both sausage and beer.
3.	[X,1|Z] = [1,X,2,4]	{X=1, Z = [2,4]}
4.	food(X) = X	Fails occur-check

Q5.3

```
swap(leaf(X), leaf(X)).

swap(tree(Left1, Right1), tree(Left2, Right2)) :-
      swap(Left1, Right2), swap(Right1, Left2).
```