Question 1: Theoretical Questions [30 points]

Q1.1 Is a function-body with multiple expressions required in a pure functional programming? In which type of languages is it useful? [3 points]

בתכנות פונקציונאלית טהורה ניתן לכתוב מספר ביטויים אך לרוב נסתפק בביטוי יחיד, מאחר וביטויים נוספים יכתבו כחלק מגוף הפונק'. פונק' עם מספר ביטויים שימושיות בתכנות פרוצדורלי, שמתמקד בהרצה נפרדת של קטעי קוד או בתכנות מונחה עצמים שמבדיל בין הצהרה על משתנה ופעולות עליו.

Q1.2 a. Why are special forms required in programming languages? Why can't we simply define them as primitive operators? Give an example [3 points]

Special forms אינם עוברים evaluation עד שלב הרצת הקוד. לכל special form תהיה דרך ייחודית לחישובה, וכך בעזרת special forms, לא נהיה חייבים לבצע בדיקה לכל חלקי הפקודה, אלא רק לחלק שרלוונטי בעת הרצת הקוד תוך כדי שמירה על חוקי הסינטקסט. בprimitive operators, כל חלקי הביטוי עוברים הערכה וחישוב, בין אם הקוד רלוונטי להרצה או לא.
לדוגמא, ניקח את הform של תנאי הif –

(if test then alt)

בעזרת תנאי הif, בהתאם לערך שמתקבל בtest, רק אחד מהביטויים של then/alt עוברים הערכה, והחלק הנוסף נשאר קומפקטי ללא חישוב. כך ניתן לבנות קוד יותר יעיל, שרק החלקים הרלוונטים לביטוי עוברים הערכה.

b. Can the logical operation 'or' be defined as a primitive operator, or must it be defined as a special form? Refer in your answer to the option of shortcut semantics. [4 points]

תנאי הor יכול להיכתב כאופרטור פרימטיבי באופן בו יתבצע evaluation לכל הביטויים שעושים עליו את האופרטור. כאשר הor נכתב כפרימיטיבי הוא אינו מהווה shortcut semantic אלא פעולת "או" פשוטה בין כל הביטויים הבוליאניים כחלק מביטוי בוליאני גדול (בג'אווה נכתוב: "exp1 | exp2").
את הor ניתן לכתוב כspecial form באופן בו הוא מבצע evaluation רק על חלק מהביטויים:
כאשר נתקל בביטוי שמחזיר true, תנאי הor יעצור ולא ימשיך להעריך את שאר חלקי הביטוי כי בכל מקרה יחזיר true. כלומר, כאשר or נכתב כspecial form, הוא מהווה shortcut semantic לפעולה "or while not true".
(בג'אווה נכתוב: "exp1 || exp2").

Q1.3 What is a syntactic abbreviation? Give two examples [4 points] 11

קיצור תחבירי שנועד לקצר ולייפות את הקוד, על ידי יצירת ביטויים מקוצרים בעלי אותה משמעות, כלומר קיצורים תחביריים ינקו את הקוד משימוש באופרטורים פרימיטיביים רבים –

לדוגמא:

((lambda (x) (* x x))(3+5)) ↔ (let (x 3+5) (* x x))

(+ (+ (+ 3 5) 8) 4) ↔ (+ 3 5 8 4)

Q1.4 a. What is the value of the following L3 program? Explain. [2 points]

(define x 1) (let ((x 5) (y (* x 3))) y)

סדר פעולות התכנית:

1. x=1
2. במקביל: x=5, y=x*3=1*3
3. החזרת y

let מבצע binding בנפרד לכל משתנה ולכן ההשמה של y "אינה מכירה" את ההשמה של "x" שבה עודכן ערכו ל-5. מאחר שזו שפת תכנות פונקציונאלית הערך שיוחזר הוא הערך האחרון, כלומר: $1 * 3 = 3$

b. Read about let* here. What is the value of the following program? Explain. [2 points]

(define x 1) (let* ((x 5) (y (* x 3))) y)

סדר פעולות התכנית:

1. $x = 1$
2. במסגרת פעולת let^*:
   a. $x = 5$
   b. $y = x * 3 = 5 * 3 = 15$
3. החזרת הערך *y*

*Let** מבצעת *binding* לכל המשתנים ב*clause* משמאל לימין. כלומר, ההשמה של *y* מתבצעת לאחר עדכון הערך של *x*. לכן, הערך של *y* מתעדכן ל15. התכנית פונקציונאלית אז הערך שיוחזר הינו האחרון ולכן יוחזר $3 * 5 = 15$

c. Annotate lexical addresses in the given expression [6 points]

```
(define [x: 2 0] [2 free])                                                  ;; (define x 2)
(define [y: 2 1] [5 free])                                                  ;; (define y 5)
(let                                                                        ;; (let
    ( ([x: 1 0] [1 free])                                                   ;;    ((x 1)
    ([f: 1 1] (lambda ([z: 0 0]) ([+ free] [x: 2 0] [y: 2 1] [z: 0 0]) ) ) )  ;;     (f (lambda (z) (+ x y z))))
([f: 1 1] [x: 1 0])                                                         ;;   (f x)
)                                                                           ;; )
(let*                                                                       ;; (let*
    ( ([x: 1 0] [1 free])                                                   ;;    ((x 1)
    ([f: 1 1] (lambda ([z: 0 0]) ([+ free] [x: 1 0] [y: 2 1] [z: 0 0]) ) ) )  ;;     (f (lambda (z) (+ x y z))))
([f: 1 1] [x: 1 0])                                                         ;;   (f x)
)                                                                           ;; )
```

d. Define the let* expression in section c above as an equivalent let expression [3 points]

(let ((x 1)) (let ((f (lambda (z) (+ x y z)))) (f x)))

e. Define the let* expression in section c above as an equivalent application expression (with no let) [3 points]

((lambda (x) ((lambda (z) (+ x y z)) x)) 1)

DBC

; Signature: (make-ok anyValue)
; Type: [N -> '(Ok N)]
; Purpose: Wrap successful program value into ok object, marking it good to work with
; Pre-conditions: true

; Tests: (make-ok 3) -> '(Ok 3), (make-ok ‘(0 1))->’(0 1)
make-ok

; Signature: (make-error messageString)
; Type: [N -> '(Error (message N))]
; Purpose: Create Error object representing an error of a program run with relevant message
; Pre-conditions: true
; Tests: (make-error "test successful")->'(ERROR, "test successful")
make-error

; Signature: (ok? anyParam)
; Type: [N -> Boolean]
; Purpose: test if a parameter is of ok type
; Pre-conditions: any
; Tests: (define r1 (make-ok 3))(ok? r1)→ #t, (define r3 ‘error)(ok? r3)→ #f
ok?

; Signature: (error? anyParam)
; Type: [N -> Boolean]
; Purpose: Check if the parameter is of type error
; Pre-conditions: true
; Tests: (define r2 (make-error "Error: key not found"))(ok? r2)→ #f,(error? r2)→ #t
error?

; Signature: (result? anyParam)
; Type: [N -> Boolean]
; Purpose: Check if the parameter is of type makeOk or error
; Pre-conditions: true
; Tests: (define r1 (make-ok 3))(result? r1)->#t,(define r3 'ok)(ok? r3)->#f
result?

; Signature: (result->val anyResult)
; Type: [Result<N>->Error.msg|Val]
; Purpose: get value of ok\error object
; Pre-conditions: anyResult needs to be a pair
; Tests: (define r2 (make-ok "kay"))(result->val r2)->"kay", (result->val (make-error "err"))->"err"
result->val
; Signature: (bind makesNonResultOutputFunction)
; Type: [(f: N->result) -> (result -> (f N |error("Binded function received none result var"))]
; Purpose: "wraps" a function output with a result (ok | error object)
; Pre-conditions: makesNonResultOutputFunction is a function that returns a result
; Tests:
(define pipe (lambda (fs) (if (empty? fs) (lambda (x) x) (compose (pipe (cdr fs)) (car fs))))) (define square (lambda (x) (make-ok (* x x))))
(define inverse (lambda (x) (if (= x 0) (make-error "div by 0") (make-ok (/ 1 x))))) (define inverse-square-inverse (pipe (list inverse (bind square) (bind inverse)))) (result->val (inverse-square-inverse 2))->4
bind

; Signature: (make-dict)
; Type: [None -> dict]
; Purpose: create a new empty dict
; Pre-conditions: true
; Tests: make-dict -> '("dictionary")
make-dict

; Signature: (dict? anyParam)
; Type: [N -> Boolean]
; Purpose: Check if the parameter is of type Dict
; Pre-conditions: true
; Tests: (define dict (make-dict))(dict? dict)->#t, (dict? '(2 4))->#f
dict?

; Signature: (put dict key val)
; Type: [Dict N U -> ok<Dict>| error("Error: not a dictionary")]
; Purpose: add key and value to an existing dictionary, or update an existing key with a given value

; Pre-conditions: true
; Tests:
(result->val (put '(1 2) 3 4))-> "Error: not a dictionary", (put (make-dict) 3 4)->'("ok" "dictionary" 3 . 4)
put

; Signature: (get dict key)
; Type: [Dict N -> ok(Dict.N)|error("Key not found")|error("Error: not a dictionary")]
; Purpose: get the value in a dictionary matching to the inputed key
; Pre-conditions: true
; Tests:
(result->val (get '(1 2) 1))->"Error: not a dictionary", (result->val (get (result->val (put dict 3 4)) 3))-> 4
get
; Signature: (map-dict dict lambda)
; Type: [Dict -> Dict]
; Purpose: Apply inputed function on each value of the dictionary
; Pre-conditions: lambda - function accepts keys of dictionary
; Tests:
(result->val (get (result->val (map-dict (result->val (put (result->val (put (make-dict) 1 #t)) 2 #f)) (lambda (x) (not x)))) 2))->#t
map-dict

; Signature: (filter-dict dict pred)
; Type: [Dict pred (key pair) -> Dict]
; Purpose: Get a dictionary and a pred function key pair -> Boolean and returns a dict only with
(key, value) that matched pred
;Pre-conditions: pred function accepts key and value and returns Boolean
; Tests:
(result->val (get (result->val (filter-dict (result->val (put (result->val (put (make-dict) 2 3)) 3 4)) even-key-odd-value?)) 3)) -> "Key not found"
filter-dict