עבודה 4 בעקרונות שפות תכנות

# חלק 1

## שאלה 1

נבצע Type inference לפונקציה הנתונה על פי שלבי האלגוריתם המתאים.

((lambda (x1 y1) (if (> x1 y1) #t #f)) 8 3)

שלב 1: Renaming – אין צורך בשינוי שמות המשתנים, כבר יש להם שמות ייחודיים.

שלב 2: התאמת משתני טיפוס לכל תתי הביטויים.

|  |  |
| --- | --- |
| **Var** | **Expression** |
| T0 | ((lambda (x1 y1) (if (> x1 y1) #t #f)) 8 3) |
| T1 | (lambda (x1 y1) (if (> x1 y1) #t #f)) |
| Tif | (if (> x1 y1) #t #f) |
| Ttest | (> x1 y1) |
| T> | > |
| Tx1 | x1 |
| Ty1 | y1 |
| T#t | #t |
| T#f | #f |
| Tnum8 | 8 |
| Tnum3 | 3 |

שלב 3: בניית משוואות טיפוסים.

|  |  |
| --- | --- |
| **Equation** | **Expression** |
| T1 = [Tnum8 \* Tnum3 → T0] | ((lambda (x1 y1) (if (> x1 y1) #t #f)) 8 3) |
| T1 = [Tx1 \* Ty1 → Tif] | (lambda (x1 y1) (if (> x1 y1) #t #f)) |
| Tif = T#t  Ttest = boolean  T#t = T#f | (if (> x1 y1) #t #f) |
| T> = [Tx1 \* Ty1 → Ttest] | (> x1 y1) |
| T> = [number \* number → boolean] | > |
| T#t = boolean | #t |
| T#f = boolean | #f |
| Tnum8 = number | 8 |
| Tnum3 = number | 3 |

שלב 4: פתרון המשוואות.

|  |  |
| --- | --- |
| **Substitution** | **Equation** |
|  | T1 = [Tnum8 \* Tnum3 → T0] |
| T1 = [Tx1 \* Ty1 → Tif] |
| Tif = T#t |
| Ttest = boolean |
| T#t = T#f |
| T> = [Tx1 \* Ty1 → Ttest] |
| T> = [number \* number → boolean] |
| T#t = boolean |
| T#f = boolean |
| Tnum8 = number |
| Tnum3 = number |

|  |  |
| --- | --- |
| **Substitution** | **Equation** |
| T1 = [Tnum8 \* Tnum3 → T0] | T1 = [Tx1 \* Ty1 → Tif] |
| Tif = T#t |
| Ttest = boolean |
| T#t = T#f |
| T> = [Tx1 \* Ty1 → Ttest] |
| T> = [number \* number → boolean] |
| T#t = boolean |
| T#f = boolean |
| Tnum8 = number |
| Tnum3 = number |

|  |  |
| --- | --- |
| **Substitution** | **Equation** |
| T1 = [Tnum8 \* Tnum3 → T0] | Tif = T#t |
| Ttest = boolean |
| T#t = T#f |
| T> = [Tx1 \* Ty1 → Ttest] |
| T> = [number \* number → boolean] |
| T#t = boolean |
| T#f = boolean |
| Tnum8 = number |
| Tnum3 = number |
| Tx1 = Tnum8 |
| Ty1 = Tnum3 |
| Tif = T0 |

|  |  |
| --- | --- |
| **Substitution** | **Equation** |
| T1 = [Tnum8 \* Tnum3 → T0]  Tif = T#t  Ttest = boolean | T#t = T#f |
| T> = [Tx1 \* Ty1 → Ttest] |
| T> = [number \* number → boolean] |
| T#t = boolean |
| T#f = boolean |
| Tnum8 = number |
| Tnum3 = number |
| Tx1 = Tnum8 |
| Ty1 = Tnum3 |
| Tif = T0 |

|  |  |
| --- | --- |
| **Substitution** | **Equation** |
| T1 = [Tnum8 \* Tnum3 → T0]  Tif = T#f  Ttest = boolean  T#t = T#f | T> = [Tx1 \* Ty1 → Ttest] |
| T> = [number \* number → boolean] |
| T#t = boolean |
| T#f = boolean |
| Tnum8 = number |
| Tnum3 = number |
| Tx1 = Tnum8 |
| Ty1 = Tnum3 |
| Tif = T0 |

|  |  |
| --- | --- |
| **Substitution** | **Equation** |
| T1 = [Tnum8 \* Tnum3 → T0]  Tif = T#f  Ttest = boolean  T#t = T#f | T> = [Tx1 \* Ty1 → boolean] |
| T> = [number \* number → boolean] |
| T#t = boolean |
| T#f = boolean |
| Tnum8 = number |
| Tnum3 = number |
| Tx1 = Tnum8 |
| Ty1 = Tnum3 |
| Tif = T0 |

|  |  |
| --- | --- |
| **Substitution** | **Equation** |
| T1 = [Tnum8 \* Tnum3 → T0]  Tif = T#f  Ttest = boolean  T#t = T#f  T> = [Tx1 \* Ty1 → boolean] | T> = [number \* number → boolean] |
| T#t = boolean |
| T#f = boolean |
| Tnum8 = number |
| Tnum3 = number |
| Tx1 = Tnum8 |
| Ty1 = Tnum3 |
| Tif = T0 |

|  |  |
| --- | --- |
| **Substitution** | **Equation** |
| T1 = [Tnum8 \* Tnum3 → T0]  Tif = T#f  Ttest = boolean  T#t = T#f  T> = [Tx1 \* Ty1 → boolean]  T> = [number \* number → boolean] | T#t = boolean |
| T#f = boolean |
| Tnum8 = number |
| Tnum3 = number |
| Tx1 = Tnum8 |
| Ty1 = Tnum3 |
| Tif = T0 |
| Tx1 = number |
| Ty1 = number |
| boolean = boolean |

|  |  |
| --- | --- |
| **Substitution** | **Equation** |
| T1 = [Tnum8 \* Tnum3 → T0]  Tif = T#f  Ttest = boolean  T#t = T#f  T> = [Tx1 \* Ty1 → boolean]  T> = [number \* number → boolean]  T#t = boolean | T#f = boolean |
| Tnum8 = number |
| Tnum3 = number |
| Tx1 = Tnum8 |
| Ty1 = Tnum3 |
| Tif = T0 |
| Tx1 = number |
| Ty1 = number |
| boolean = boolean |

|  |  |
| --- | --- |
| **Substitution** | **Equation** |
| T1 = [Tnum8 \* Tnum3 → T0]  Tif = boolean  Ttest = boolean  T#t = boolean  T> = [Tx1 \* Ty1 → boolean]  T> = [number \* number → boolean]  T#t = boolean  T#f = boolean | Tnum8 = number |
| Tnum3 = number |
| Tx1 = Tnum8 |
| Ty1 = Tnum3 |
| Tif = T0 |
| Tx1 = number |
| Ty1 = number |
| boolean = boolean |

|  |  |
| --- | --- |
| **Substitution** | **Equation** |
| T1 = [number \* number → T0]  Tif = boolean  Ttest = boolean  T#t = boolean  T> = [Tx1 \* Ty1 → boolean]  T> = [number \* number → boolean]  T#t = boolean  T#f = boolean  Tnum8 = number  Tnum3 = number | Tx1 = Tnum8 |
| Ty1 = Tnum3 |
| Tif = T0 |
| Tx1 = number |
| Ty1 = number |
| boolean = boolean |

|  |  |
| --- | --- |
| **Substitution** | **Equation** |
| T1 = [number \* number → T0]  Tif = boolean  Ttest = boolean  T#t = boolean  T> = [Tx1 \* Ty1 → boolean]  T> = [number \* number → boolean]  T#t = boolean  T#f = boolean  Tnum8 = number  Tnum3 = number | Tx1 = number |
| Ty1 = Tnum3 |
| Tif = T0 |
| Tx1 = number |
| Ty1 = number |
| boolean = boolean |

|  |  |
| --- | --- |
| **Substitution** | **Equation** |
| T1 = [number \* number → T0]  Tif = boolean  Ttest = boolean  T#t = boolean  T> = [number \* Ty1 → boolean]  T> = [number \* number → boolean]  T#t = boolean  T#f = boolean  Tnum8 = number  Tnum3 = number  Tx1 = number | Ty1 = Tnum3 |
| Tif = T0 |
| Tx1 = number |
| Ty1 = number |
| boolean = boolean |

|  |  |
| --- | --- |
| **Substitution** | **Equation** |
| T1 = [number \* number → T0]  Tif = boolean  Ttest = boolean  T#t = boolean  T> = [number \* Ty1 → boolean]  T> = [number \* number → boolean]  T#t = boolean  T#f = boolean  Tnum8 = number  Tnum3 = number  Tx1 = number | Ty1 = number |
| Tif = T0 |
| Tx1 = number |
| Ty1 = number |
| boolean = boolean |

|  |  |
| --- | --- |
| **Substitution** | **Equation** |
| T1 = [number \* number → T0]  Tif = boolean  Ttest = boolean  T#t = boolean  T> = [number \* number → boolean]  T> = [number \* number → boolean]  T#t = boolean  T#f = boolean  Tnum8 = number  Tnum3 = number  Tx1 = number  Ty1 = number | Tif = T0 |
| Tx1 = number |
| Ty1 = number |
| boolean = boolean |
|  |

|  |  |
| --- | --- |
| **Substitution** | **Equation** |
| T1 = [number \* number → boolean]  Tif = boolean  Ttest = boolean  T#t = boolean  T> = [number \* number → boolean]  T> = [number \* number → boolean]  T#t = boolean  T#f = boolean  Tnum8 = number  Tnum3 = number  Tx1 = number  Ty1 = number  T0 = boolean | Tx1 = number |
| Ty1 = number |
| boolean = boolean |

|  |  |
| --- | --- |
| **Substitution** | **Equation** |
| T1 = [number \* number → boolean]  Tif = boolean  Ttest = boolean  T#t = boolean  T> = [number \* number → boolean]  T> = [number \* number → boolean]  T#t = boolean  T#f = boolean  Tnum8 = number  Tnum3 = number  Tx1 = number  Ty1 = number  T0 = boolean | ~~T~~~~x1~~ ~~= number~~ |
| ~~T~~~~y1~~ ~~= number~~ |
| ~~boolean = boolean~~ |

## שאלה 2

### סעיף a

נכון.

{f:[T1->T2], x: T1} |- (f x): T2

ניצור משתני טיפוס לכל אחד מתתי הביטויים:

|  |  |
| --- | --- |
| **Var** | **Expression** |
| Tapp | (f x) |
| Tf | f |
| Tx | x |

בהינתן f:[T1->T2], ניתן להסיק כי Tf = [T1 -> T2].

מכלל הגזירה של App Expression נובע Tf = [T1 -> Tapp]. כלומר Tapp = T2.

עוד נתון כי x: T1 ולכן הפעלת f על x תקינה ותניב את ערך מטיפוס T2 כאמור.

לסיכום, (f x): T2.

### סעיף b

לא נכון.

{f:[T1->T2], g: [T2->T3]}, x: T2} |- (f g x): T3

על פי הנתון f:[T1->T2] ניתן להסיק כי f היא פונקציה המקבלת פרמטר אחד (מסוג T1).

לכן ודאי הביטוי (f g x) לא יכול להיות מטיפוס T3, מכיוון שהוא לא ביטוי תקין (הפעלת f על 2 פרמטרים).

### סעיף c

נכון.

{f:[T2->T1], g: [T1->T2], x: T1} |- (f (g x)): T1

על פי הנתונים x: T1 ו- g: [T1->T2] ניתן להסיק כי הפעלת g עם הפרמטר x תקינה, ותניב ערך מטיפוס T2. כלומר (g x): T2.

כמו כן, מהנתון f:[T2->T1] נובע שהפעלת f על הערך שמתקבל מ- (g x) תקינה ותחזיר ערך מטיפוס T1. כלומר (f (g x)): T1.

### סעיף d

לא נכון.

{f:[T2->Number], x: Number} |- (f x x): Number

על פי הנתון f:[T2->Number] ניתן להסיק כי f היא פונקציה המקבלת פרמטר אחד (מסוג T2).

לכן ודאי הביטוי (f x x) לא יכול להיות מטיפוס Number, מכיוון שהוא לא ביטוי תקין (הפעלת f על 2 פרמטרים).

## שאלה 3

### סעיף a

הטיפוס של cons הוא [T1 \* T2 -> Pair(T1, T2)].

cons היא פונקציה המקבלת שני פרמטרים, ייתכן כי מטיפוסים שונים, ולכן היא מהצורה  
[T1 \* T2 -> T3].

cons יוצרת משני הפרמטרים שהיא מקבלת זוג סדור המכיל את שני האיברים האלה על פי סדר הפרמטרים, לכן טיפוס הערך המוחזר מ- cons הוא Pair כך שטיפוס האיבר הראשון בו זהה לטיפוס הפרמטר הראשון שניתן לפונקציה cons (כלומר T1), ובדומה טיפוס האיבר השני הוא T2. כלומר, T3 = Pair(T1, T2).

### סעיף b

הטיפוס של car הוא [Pair(T1, T2) -> T1].

car היא פונקציה המקבלת פרמטר אחד שהוא זוג המכיל שני איברים, ייתכן כי מטיפוסים שונים, ולכן היא מהצורה [Pair(T1, T2) -> T3].

car מחזירה את האיבר הראשון בזוג. לכן טיפוס הערך המוחזר מ- car זהה לטיפוס של האיבר הראשון בזוג (T1). כלומר T3 = T1.

### סעיף c

הטיפוס של cdr הוא [Pair(T1, T2) -> T2].

cdr היא פונקציה המקבלת פרמטר אחד שהוא זוג המכיל שני איברים, ייתכן כי מטיפוסים שונים, ולכן היא מהצורה [Pair(T1, T2) -> T3].

cdr מחזירה את האיבר השני בזוג. לכן טיפוס הערך המוחזר מ- cdr זהה לטיפוס של האיבר השני בזוג (T2). כלומר T3 = T2.

## שאלה 4

## הטיפוס של הפונקציה המוגדרת בביטוי הבא:

## (define f (lambda (x) (values x x x)))

הוא:

[T1 -> (T1 \* T1 \* T1)]

## שאלה 5

### סעיף a

עבור {x: T1}, {x: T2} ה-MGU הוא {T1=T2}.

זאת משום ש- {x: T1} o {T1=T2} = {x: T1} o {T1=T2}, וה-unifier הזה מבצע substitution רק עבור הטיפוסים הנחוצים מבלי להניח הנחות נוספות על האיברים המוחלפים. בנוסף, כל שאר ה-unifiers מתקבלים ממנו ע"י הפעלת substitutions נוספים, לכן הוא ה-MGU (עד כדי renaming).

### סעיף b

עבור {x: Number}, {x: Number} ה- MGUהוא { }, כלומר unifier ריק.

זאת משום ש- {x: Number } o { } = {x: Number } o { }, וכל שאר ה-unifiers מתקבלים ממנו ע"י הפעלת substitutions נוספים, לכן הוא ה-MGU.

### סעיף c

עבור

{x: [T1\*[T1->T2]->Number]}, {x: [[T3->Number]\*[T4->Number]->Number]}

ה-MGU הוא **{T1=[T3->Number], T4=[T3->Number], T2=Number}**. זאת משום ש-

{x: [T1\*[T1->T2]->Number]} o

**{T1=[T3->Number], T4=[T3->Number], T2=Number}**

= {x: [[T3->Number]\*[T4->Number]->Number]} o

**{T1=[T3->Number], T4=[T3->Number], T2=Number}**

= {x: [[T3->Number]\*[[T3->Number]->Number]->Number]}

כמו כן, ה-unifier הזה מבצע substitution רק עבור הטיפוסים הנחוצים מבלי להניח הנחות נוספות על האיברים המוחלפים. בנוסף, כל שאר ה-unifiers מתקבלים ממנו ע"י הפעלת substitutions נוספים, לכן הוא ה-MGU (עד כדי renaming).

### סעיף d

עבור {x: [T1->T1]} , {x: [T1->[Number->Number]]}

ה-MGU הוא {T1=[Number->Number]}. זאת משום ש-

x: [T1->T1]} o {T1=[Number->Number]}  
= {x: [T1->[Number->Number]]} o {T1=[Number->Number]}  
= {x: [[Number->Number]->[Number->Number]]}

כמו כן, ה-unifier הזה מבצע substitution רק עבור הטיפוסים הנחוצים מבלי להניח הנחות נוספות על האיברים המוחלפים. בנוסף, כל שאר ה-unifiers מתקבלים ממנו ע"י הפעלת substitutions נוספים, לכן הוא ה-MGU.

# חלק 2

הערות

* מימשנו את values כ- PrimOp.
* לא תמכנו ב nested tuples.

## שאלה 3

(define f: [number -> (number \* number)] (lambda (x: number): (number \* number) (values x (+ x 1))))

(define g: [T1 -> (string \* T1)] (lambda (x: T1): (string \* T1) (values “x” x)))

# חלק 3

import { isBoolean } from "../shared/type-predicates";

// Return a generator that combines generator1 and generator2 by interleaving their values

export function\* braid(generator1: () => Generator, generator2: () => Generator): Generator {

    let generators = [generator1(), generator2()];

    while (generators.length > 0) {

        for (let i = 0; i < generators.length; i++) {

            const { value, done } = generators[i].next();

            if (done) {

                generators.splice(i, 1);

            }

            else {

                yield value;

            }

        }

    }

}

// returns a generator that combines both generators by taking two elements from gen1

// and one from the gen2.

export function\* biased(generator1: () => Generator, generator2: () => Generator): Generator {

    let gen1 = generator1();

    let gen2 = generator2();

    let gen1Done = false;

    let gen2Done = false;

    while (!(gen1Done && gen2Done)) {

        for (let i = 0; i < 2; i++) {

            const { value, done } = gen1.next();

            isBoolean(done) ? gen1Done = done : gen1Done = true;

            if (!gen1Done) {

                yield value;

            }

        }

        const { value, done } = gen2.next();

        isBoolean(done) ? gen2Done = done : gen2Done = true;

        if (!gen2Done) {

            yield value;

        }

    }

}

# חלק 4

## שאלה 1

### סעיף a

import { KeyValuePair } from "ramda";

// Part 4 ; Q1.a

export const divisionByZero = new Error("Division By Zero");

export function f(x: number): Promise<number> {

    return new Promise<number>((resolve, reject) => {

        if (x === 0) {

            reject(divisionByZero);

        } else {

            resolve(1 / x);

        }

    });

}

export function g(x: number): Promise<number> {

    return new Promise<number>((resolve, reject) => {

        try {

            resolve(x \* x);

        } catch (err) {

            reject(err);

        }

    });

}

export function h(x: number): Promise<number> {

    return new Promise<number>((resolve, reject) => {

       g(x)

        .then((x) => f(x) )

        .then((x) => resolve(x) )

        .catch((err) => reject(err) );

    });

}

### סעיף b

לשימוש בממשק של Promises ישנם מספר יתרונות על פני השימוש בממשק של Callbacks:

1. החתימה של הפונקציות בממשק של Promises דומות מאד לממשק הסינכרוני (למעט ערך ההחזרה שהוא מהצורה Promise<T>). לעומת זאת, הטיפוסים של הפונקציות בממשק Callbacks מסובכים הרבה יותר.
2. Error Handling פשוט יותר במנגנון Promises, ומתבצע על ידי שימוש ב ".catch" יחיד בסוף שרשור הPromises. לעומת זאת, טיפול בשגיאות במנגנון Callbacks הוא מסובך הרבה יותר, ודורש טיפול נפרד בשגיאות בכל callback.
3. שימוש סדרתי ב Promises מאפשר הרכבה נוחה של קריאות לפונקציות על ידי שרשור נוח של רצף הפעולות שאמורות להתבצע בתוכנית (בעזרת ".then"). לעומת זאת, ניהול סדר הקריאות של callbacks הוא מסובך יותר ודורש קינון של callbacks אחד בתוך השני שהוא אינטואיטיבי פחות.

## שאלה 2

// Part 4 ; Q2

export type SlowerResult<T> = KeyValuePair<number, T>;

const wrapPromise = <T>(promise: Promise<T>, index: number): Promise<SlowerResult<T>> =>

    new Promise<SlowerResult<T>>((resolve, reject) =>

        promise.then((res) => resolve([index, res]))

            .catch((e) => reject(e)));

// If both promises succeed, the return value is (0, value) or (1, value) where 0 indicates that

// the first promise was ​slower​, and 1 indicates that the second promise was slower,

// value is the return value of the promise that was resolved last.

export const slower = <T>(promises : Promise<T>[]): Promise<SlowerResult<T>> => {

    const w1 = wrapPromise(promises[0], 0);

    const w2 = wrapPromise(promises[1], 1);

    return new Promise<SlowerResult<T>>((resolve, reject) =>

        Promise.race([w1, w2])

            .then((fasterValue) => {

                Promise.all([w1, w2])

                    .then((values) => resolve(values.find(element =>  element[0] != fasterValue[0])))

                    .catch((e) => reject(e))

            })

            .catch((e) => reject(e))

    );

};