תכנות מונחה עצמים – תרגיל בית 5

כללי

1. מועד ההגשה הוא 01/07/2021 בשעה 23:59.
2. המתרגל האחראי על התרגיל הוא מקסים.
3. קראו היטב את הוראות התרגיל.
4. שאלות יש לשלוח למקסים (maxim.barsky@campus.technion.ac.il) עם הכותרת 236703 HW5.
5. הקפידו על קוד ברור, קריא ומתועד ברמה סבירה. עליכם לתעד כל חלק שאינו טריוויאלי בקוד שלכם.
6. מהירות ביצוע אינה נושא מרכזי בתרגילי הבית הקורס. בכל מקרה של התלבטות בין פשטות לבין ביצועים, העדיפו את המימוש הפשוט.
7. הימנעו משכפול קוד והשתמשו במידת האפשר בקוד שכבר מימשתם.
8. כדי להימנע מטעויות, אנא עיינו ברשימת ה־**FAQ** המתפרסמת באתר באופן שוטף.

הקדמה

בתרגיל זה תתכנתו תכנות מתקדם בשפת ++C. התרגיל מורכב משני חלקים בלתי תלויים העוסקים בנושאים שונים ב־++C. האחד יעסוק בשימוש במערכת ה־templates ל־meta-programming והשני יעסוק ב־streams בדומה ל־streams כפי שראיתם בתרגול 3 (של Java).

למען הצלחתכם בקורס אנו ממליצים לשני השותפים לקחת חלק פעיל בשני חלקי תרגיל הבית (ולא לחלק את התרגיל חלק־חלק).

חלק א

כפי שלמדתם בקורס, מערכת ה־templates של ++C היא טיורינג־שלמה. כלומר, ניתן לממש בה כל תכנית מחשב בזמן הקומפילציה של התוכנית. בחלק א של התרגיל אנחנו נדגים את היכולת הזאת ע"י מימוש של מפרש (חלקי) של שפת ליספ.

חלק ב

חלק זה יעסוק במימוש של מנגנון Streams בדומה למנגנון בשפת Java הנלמד בתרגולים. המגנון יאפשר להפעיל פעולות על רצפים של איברים בדומה למנגנון בשפת Java. בנוסף, גם כאן המנגנון יפעל בצורה lazy, כלומר לא נבצע את הפעולות על ה־stream עד שלא נידרש להן ע"י פעולה כמו collect או forEach.

חלק א – Template meta-programming

הקדמה

בחלק זה תממשו מפרש פשוט לשפת LISP. המפרש יוגבל למספר מצומצם של פעולות, המפרש יקבל כקלט רצף של אסימונים – Tokens וישערך את הביטויים המיוצגים על ידי רצף האסימונים. בתרגיל זה אין צורך לעבוד עם מחרוזות או לחלק טקסט לאסימונים.

הביטויים בתרגיל הינם ביטויים חוקיים, אין צורך לבדוק תקינות קלט

אסימון

מהו אסימון?

אסימון הוא ערך בעל משמעות מיוחדת אשר נוצר מתוך מחרוזת קלט. בתרגיל נעבוד לא על קוד מקור בשפת ליספ אלא על רשימה של אסימונים של תוכנית בשפת ליספ. במקום לקבל קוד מקור של תוכנית, נקבל את רצף האסימונים שמתאים לו, לדוגמה:

```
// lisp source code:
(+ 1 (+ 3 5))

// translates to these tokens:
LPAR PLUS INT<1> LPAR PLUS INT<3> INT<5> RPAR RPAR
```

המשמעות של האסימונים האלה תוסבר בהמשך התרגיל.

שפת ליספ

כל ביטוי בשפת ליספ הוא או ערך (ליטרל, במקרה שלנו מספר שלם) או ביטוי מרוכב. ביטויים מרוכבים הם תוצאה של קריאות לפונקציה, כמו למשל פונקציית החיבור, לדוגמה: חיבור של שני מספרים.

```
(+ 1 2)
```

בגרסה שלנו לשפה אנחנו נתמוך במספר מצומצם של פונקציות. המשתמש לא יוכל להגדיר פונקציות בעצמו. בנוסף לכך, המשתמש לא יוכל להגדיר משתנים, לכן בגרסה שלנו של השפה אין צורך לשמור סביבה וכריכות (קישורים בין שמות וערכים) בשונה מהנלמד בקורס שפות תכנות.

ביטויים

בגרסת התרגיל לשפת ליספ, הביטויים הפשוטים יהיו מספרים שלמים (int).
ביטויים מורכבים יהיו קריאות לפונקציות, שמות הפונקציות יהיו אסימונים אשר יסופקו בקבצים המצורפים.

הביטויים בתרגיל הינם ביטויים חוקיים, אין צורך לבדוק תקינות קלט

ייצוג האסימונים בתרגיל

המפרש שנממש בתרגיל הבית יקבל כקלט רצף של אסימונים ויצטרך לשערך אותם לערך כלשהו. לכן עולה השאלה: כיצד נייצג את האסימונים האפשריים בתרגיל?

לשם כך בקובץ המסופק לכם מוגדרות מחלקות (struct־ים בעצם), אשר כל אחת מהן מהווה טיפוס אשר מייצג את האסימון המתאים לה. (מגדירים מחלקות ולא enums למשל שכן בחישוב בעזרת templates אנחנו בד"כ מקבלים כפרמטרים טיפוסים ולא ערכים ולכן מייצגים את האסימונים באמצעות טיפוסים).
בנוסף לכך, אסימונים של מספרים ייוצגו ע"י מחלקה בשם Int אשר מכילה שדה יחיד מטיפוס int אשר יכיל את הערך המספרי של האסימון. נייצג אסימון של המספר 23 באמצעות הטיפוס Int<23>. (מימוש למחלקה נמצא בקובץ המצורף).

רשימת (חלקית) של האסימונים

Character Representation	Token	Type
(	LPAR	**struct** LPAR {};
)	RPAR	**struct** RPAR {};
+	PLUS	**struct** PLUS {};
-	MINUS	**struct** MINUS {};
=	EQ	**struct** EQ {};
COND	COND	**struct** COND {};
NOT	NOT	**struct** NOT {};
5	Int¡5¿	template<**int** n> **struct** Int<n> {};

ייצוג ביטויים בתרגיל

ראינו כיצד מייצגים אסימונים בתרגיל.
כעת נייצג ביטויים כרצף של אסימונים. בתרגיל נממש את הפונקציה eval אשר משערכת ביטויים בשפת ליספ, נעביר את הביטויים שהיא תקבל כקלט ותשערך באמצעות parameter pack. כלומר template <typename ...T>.

Expression	Representation
+ 1 2	PLUS Int<1> Int<2>
(+ 1 2)	LPAR Int<1> Int<2> RPAR
(= 1 2)	LPAR EQ Int<1> Int<2> RPAR

המפרש

המפרש ימומש באמצעות מימוש ה־struct Eval. למבנה זה יהיו שני שדות מסוג static constexpr int, האחד יהיה ערך תוצאת השערוך (המפרש שלנו תומך רק בפעולות על מספרים ולכן הערך יהיה מטיפוס int). השדה השני ישמש אתכם כחלק מתהליך ה־parsing.

```
template <typename ...T>
struct Eval<T...> {
    static constexpr int value; // the value of the evaluation result
    static constexpr int drop_amount; // used for parsing
};
```

פונקציות הנתמכות ע"י המפרש

עליכם לתמוך במפרש בפונקציות הבאות:

1. PLUS – הפונציה הזו מקבלת שני ביטויים ומחזירה את תוצאת החיבור של שני הביטויים (שומרת בתוך השדה value את ערך תוצאת החיבור). לדוגמה, חיבור של שני מספרים בליספ:

```
// result of the lisp expression "+ 3 5"
Eval<PLUS, Int<3>, Int<5>>::value == 8;
```

2. MINUS – הפונקציה הזו מקבלת שני ביטויים ומחזירה את תוצאת החיסור של שני הביטויים (שומרת בתוך השדה value את ערך תוצאת החיסור). לדוגמה, חיסור של שני מספרים:

```
// result of the lisp expression "- 3 7"
Eval<MINUS, Int<3>, Int<7>>::value == -4;
```

3. MUL – הפונקציה הזו מקבלת שני ביטויים ומחזירה את תוצאת הכפל של שני הביטויים (שומרת בתוך השדה value את ערך תוצאת הכפל). לדוגמה, כפל של שני מספרים:

```
// result of the lisp expression "* 4 2"
Eval<MUL, Int<4>, Int<2>>::value == 8;
```

4. DIV – הפונקציה הזו מקבלת שני ביטויים ומחזירה את תוצאת החילוק של שני הביטויים (שומרת בתוך השדה value את ערך תוצאת החילוק). לדוגמה, חילוק של שני מספרים:

```
// result of the lisp expression "/ 9 3"
Eval<DIV, Int<9>, Int<3>>::value == 3;
```

שימו לב שזוהי חלוקת שלמים (מתנהג כמו חלוקת int־ים בשפת C).

בכל הפעולות האריתמטיות, אין צורך לדאוג לסדר פעולות חשבון שכן בשפת ליספ, אין קדימות של פעולות אלא חוקי הקדימות נובעים ממבנה הביטוי. יש לשערך את ביטויים באופן רקורסיבי לפי המבנה שלהם. כלומר הביטויים הבאים שקולים:

```
* + 1 2 3 == * (+ 1 2) 3 // == (1 + 2) * 3, same expression in C syntax
```

5. EQ – הפונקציה מקבלת שני ביטויים ומחזירה את הערך 1 אם תוצאת השיערוך שלהם זהה ו־0 אחרת (שומרת בתוך השדה value את הערך 0 או 1). לדוגמה:

```
// these two expressions are equal
// the equivalent lisp expression is "EQ (+ 1 2) + 2 1"
Eval<EQ, LPAR, PLUS, Int<1>, Int<2>, RPAR, PLUS, Int<2>, Int<1>>::value == 1;

// these two expressions are not equal
// the equivalent lisp expression is "EQ 1 MUL 1 2"
Eval<EQ, Int<1>, MUL, Int<1>, Int<2>>::value == 0;
```

6. NOT – הפונקציה מקבלת ביטויי ומחזירה 1 אם תוצאת השערוך שלו שווה ל־0 ו־1 אחרת (שומרת בתוך השדה value את הערך 0 או 1).

```
// the equivalent lisp expression is "NOT + 1 0"
Eval<NOT, PLUS, Int<1>, Int<0>>::value == 0;

// the equivalent lisp expression is "NOT 0"
Eval<NOT, Int<0>>::value == 1;
```

7. COND – הפונקציה מקבלת שלושה ביטויים, הביטויי הראשון הוא ביטוי תנאי, אם ערכו שונה מ־0 אז הפונקציה תחזיר את ערך הביטויי השני, אחרת את ערך הביטויי השלישי. הפונקציה דומה לביטויים בוליאניים בשפת Squeak.

```
// the general form is "COND condition if_true if_false

// the equivalent lisp expression is "COND (EQ 1 0) 3 5"
// since 1 + 0 is not zero, we return the second expression (if_true)
Eval<COND, LPAR, EQ, Int<1>, Int<0>, RPAR, Int<3>, Int<5>>::value == 3;

Eval<COND, Int<0>, Int<9>, Int<10>>::value == 10;
```

8. FACT – הפונקציה הזו תקבל ביטויי אחד כקלט ותחזיר את תוצאת חישוב פונקציית העצרת על הקלט (שומרת בתוך השדה value את ערך תוצאת פונקציית העצרת).

```
// equivalent to 4!
Eval<FACT, Int<4>>::value == 24;

// equivalent to (2 + 3)!
Eval<Fact, PLUS, Int<2>, Int<3>>::value == 120;
```

Parsing

בתרגיל לא תצטרכו לבצע parsing מסובך, שכן ליספ היא שפה שקל (יחסית) לפרסר.
על מנת לא לזרוק אתכם לבד למים עמוקים, נצרף הסבר כיצד לפרסר מתוך רצף האסימונים פעולת חיבור ולחשב את ערכה. בחלק זה גם יוסבר מה משמעות השדה **drop_amount** במבנה Eval.
כאשר מקבלים רשימה של אסימונים שבראשה עומד האסימון PLUS יש תחילה לפרסר את האיבר הראשון בפעולת החיבור, בין אם הוא מספר או ביטויי מורכב בעצמו. לאחר מכן יש לפרסר את הביטויי השני שבא אחריו, שגם ערכו ישמש לפעולת החיבור. לשם כך יש לדלג על האסימונים ששייכים לביטוי הראשון. לשם כך לכל פעולת Eval נשמור את מספר האסימונים שמרכיבים אותה בשדה drop_amount כך נוכל לדעת על כמה אסימונים יש לדלג בקלט על מנת להגיע לביטויי השני.
בפסודוקוד, אלגוריתם הפרסור והחישוב של פעולת החיבור ייראה כך:

```
template <typename ...TokenStream> // this list of tokens starts with the token PLUS
struct Eval <parameters here> {
    first = Eval<TokenStream after PLUS>;
    second = Eval<TokenStream after PLUS and after first::drop_amount>;

    static constexpr int value = first::value + second::value;
    static constexpr int drop_amount = ...;
};
```

טסטים

באתר יסופקו מספר קבצי בדיקה המדגימים את השימוש ב־Eval אשר יאפשר לכם לבדוק את עצמכם. שימו לב, מעבר ה־"טסטים" האלה לא מבטיח קבלת ציון מלא בתרגיל הבית.

הערות והנחיות

- ניתן ומומלץ לפצל את הקוד למספר קבצי header ולשמור על הקוד מאורגן.
- בכל מקום בו כתוב מבנה הכוונה היא ל־struct.
- ניתן להגדיר את המבנים באמצעות struct או class, לבחירתכם.
- שימו לב שאם אתם מפצלים את הקוד למספר קבצי header, יש לספק קובץ בשם "lisp.h" אשר יכלול/ייבא את יתר קבצי ה־header שלכם.

חלק ב – מימוש מערכת Streams

מבוא

בחלק זה של התרגיל תממשו מערכת Streams הדומה למערכת שלמדתם עליה בתרגולי Java.
כזכור, Stream הוא רצף של איברים התומך בפעולות סדרתיות ומקביליות (בגרסה שתממשו לא תתמכו בפעולות מקביליות) המושפעות מהפעולות שקדמו להן, כאשר נקודת ההתחלה היא יצירת ה־Stream מאוסף איברים, כלומר Collection.

פעולות על **Streams**

הפעולות על streams מתחלקות לשלושה סוגים:

- פעולות יצירה – פעולות היוצרות Stream חדש. למשל (...)of, המתודה ()stream של Collection.
- פעולות ביניים – פעולות המבצעות מניפולציות על Stream נוכחי ומחזירות Stream (כלומר לא ניתן להשתמש באיברים השמורים בו). למשל ()map, ()sorted.
- פעולות טרמינליות – פעולות הסוגרות את ה־Stream ומחזירות Collection חדש)בהתאם לפעולות שבוצעו) או סקלר. למשל ()reduce, ()collect.

תכונה מיוחדת של Stream היא שהם עובדים בצורה lazy – כלומר כל פעולות הביניים שהופעלו במהלך בניית ה־Stream מתבצעות בפועל רק לאחר ביצוע פעולה טרמינלית. מתכונה זו נובע שכל בדיקות הטיפוסים ובדיקות השימוש ב־Stream נעשות רק ב־"רגע האחרון", רק שיש בהן צורך.
הגרסה שתממשו בתרגיל הבית תעבוד על מבני נתונים סטנדרטיים של ה־STL – std::list, std::vector, std::set ועוד. (עבור std::map תהיה הנתנהגות מיוחדת שתוגדר בהמשך).

C++ Lambdas – Anonymous Functions

כזכור ממימוש מנגנון ה־Streams המקורי ב־Java, כדי לתת התנהגות מיוחדת לכל פעולה ב־Stream מועברת כפרמטר מתודה או פונקצייה אנונימית. גם במימוש שלנו נאפשר שימוש בפונקציות אנונימות, פונקציות רגילות (ומתודות סטטיות שלא מקושרות למופע ספציפי של מחלקה). בעזרת מנשק ה־**functional** שנוסף ב־C++11 ומאפשר תכנות (טיפה) יותר פונקציונאלי בשפה.
המנשק מגדיר את הטיפוס std::function $< R(T_1, T_2, ..., T_n)$ שמהווה טיפוס מעטפת לפונקציות ולפונקציות אנונימיות. כאשר R הוא טיפוס החזרה של הפונקציה ו־$T_1, T_2, ..., T_n$ הם טיפוסי הפרמטרים של הפונקציה. עבור פונקציה שמקבלת 0 פרמטרים ומחזירה R נקבל את הטיפוס std::function $< R() >$. נשתמש במעטפת זו בכל מתודה שתוגדר בהמשך שתקבל פונקציה כפרמטר.
כמו כן, ב־C++11 נוספו פונקציות אנונימיות שזוכרות את הסביבה שלהן (נקראות גם closures) הדומות ל־**BlockClosure** משפת Squeak.
מבנה פונקציה אנונמית (lambda) בשפת C++:

```
[ <captures> ]( <params> ) -> <ret> { <body> }
```

כאשר **params** הם הפרמטרים שהפונקציה האנונימית מקבלת, **ret** הוא טיפוס החזרה של הפונקציה (אפשר לוותר על החץ ועל **ret** במידה והקומפיילר ידע להסיק לבד את טיפוס החזרה של הפונקציה). body הוא הגוף של הפונקציה.
captures הם רשימה (יכולה להיות ריקה) של משתנים שה־lambda זוכרת, בעצם זוהי הסביבה שה־closure שומרת. תיאור לסינטקס של closure נמצא בלינק הבא:
http://en.cppreference.com/w/cpp/language/lambda
דוגמת שימוש לפונקציה אנונימית המקבלת שני מספרים ומחזירה את סכום ריבועיהם:

```
std::function<int(const int, const int)> lambda_example = [](const int a, const int b) -> int {
    int a_squared = a * a;
    int b_squared = b * b;
    return a_squared + b_squared;
};

int res = lambda_example(42, 24); // res = 42 * 42 + 24 * 24
```

נעבור כעת למימוש מערכת ה־Streams. את אובייקט ה־Stream נייצג בעזרת המחלקה **Stream**. את עקרון ה־laziness נממש בעזרת lambdas ויכולות ה־closures שלהם. באופן דומה למימוש PolyStream בתרגיל בית 1. נשמור במחלקה שדה שיהיה פונקציה שלא מקבלת פרמטרים ומחזירה אוסף המחזיק מצביעים של הטיפוס שה־Stream מייצג (הטיפוס יהיה טיפוס גנרי). בהפעלת הפונקציה נחשב את כל פעולות ה־Stream שהצטברו על אובייקט ה־Stream הנוכחי. כל פעולת ביניים תשנה את השדה הזה כך שיועדכן "לעטוף" את השדה הישן (בעזרת ה־closure) ואת הפעולה הנוכחית.

הממשק למערכת יהיה נגיש מהקובץ **Stream.h** יש לדאוג לכך שכל הפעולות שמוגדרות בתרגיל נגישות מקובץ זה, שכן כל הבדיקות ישתמשו ב־include יחיד אליו.

טיפ

במהלך המימוש ייתכן ותתקלו במצבים שלא תדעו במדויק מה הטיפוס של משתנה מסוים (למשל, הוא יקבע רק בזמן קומפילציה ולא בעת כתיבת הקוד, לדוגמה שדה של פרמטר גנרי), לכן מומלץ להשתמש במשתנים "מטיפוס auto" שהקומפיילר יסיק את טיפוסם בעצמו.

המחלקה Stream

המחלקה תקבל את T כטיפוס גנרי (והוא יהיה טיפוס האיברים של כל ה־Stream) ותשמור שדה מטיפוס std::function¿ (כפי שהוגדר לעיל, הפונקציה מחזירה אוסף כלשהו של *T, סוג האוסף נתון לבחירתכם).

כמו כן, אתם רשאים להוסיף מחלקות נוספות שיורשות מ־Stream ובנאים כרצונכם.

שימו לב, בדוגמאות בהמשך התרגיל יופיע הוקטור intPointerVector. זהו וקטור של מצביעים ל־int המאותחל באופן הבא:

```
int array[10] = {0,1,2,3,4,5,6,7,8,9};
std::vector<int*> intPointerVector;
for (size_t i = 0; i < 10; ++i) {
    intPointerVector.push_back(array + i);
}
```

על המחלקה לתמוך בפעולות הבאות:

פעולות יצירה:

- of(TContainer&) מתודה סטטית גנרית של המחלקה Stream<T> המקבלת כפרמטר גנרי (TContainer) אוסף של מצביעים ל־T ומחזיר עצם של Stream (לא מצביע ל־Stream) שמאותחל להכיל את האיברים של האוסף שהועבר.

שימו לב שהפעלת הפעולה טרמינלית על Stream חדש (שלא הופעלו עליו פעולות ביניים) היא מצב חוקי וצריכה לעבוד על האיברים המקוריים (ולפעול לפי הגדרתה, למשל collect תחזיר את איברי האוסף ללא שינוי).

לדוגמה:

```
Stream<int> stream = Stream<int>::of(intPointerVector);
```

ניתן להניח שהאוסף הגנרי יהיה אוסף מ־STL (למשל vector, list) ויכיל את המתודות ()begin ו־()end.

עבור המקרה המיוחד של map המקבל שני פרמטרים גנריים (מפתח וערך) יש להחזיר Stream של הערכים ולא של המפתחות. ניתן להניח שהמבנה היחיד בעל יותר מפרמטר גנרי אחד שיועבר יהיה map.

רמז למימוש יופיע בהמשך.

פעולות ביניים:

כל הפעולות הבאות מחזירות Stream, והוא ישמש לשרשור פעולות (כמו ב־Java). האובייקט שיוחזר לא חייב להיות אובייקט חדש, החלטה זו נתונה בידיכם. (פעולת map שתוגדר בהמשך מחייבת החזרת אובייקט חדש).

חשוב מאוד: הפעולות צריכות לממש התנהגות עצלה! כלומר רק בעת ביצוע פעולה טרמינלית יש לבצע אותן בפועל (המתודות יכינו את ה־Stream לכך).

- filter() הפעולה מקבלת כפרמטר פונקציית פרדיקט const T* → bool ומחזירה את אובייקט ה־Stream לאחר שפעולת ה־Filter "עטפה" אותו. לאחר ביצוע פעולת ה־Filter האיברים שימחקו יהיו האבירים שהפקידט החזיר עבורם false.

לדוגמה:

```
auto stream = Stream<int>::of(intPointerVector).filter([](const int* val) { return *val });
```

בדוגמה האיברים שיישארו הם המספרים ששונים מ־0.

- map() הפועלה מקבלת כפרמטר גנרי R את הטיפוס אליו האיברים הנוכחים ימופו, בעזרת פרמטר שהיא תקבל – פונקציית מיפוי const T* → R*. המתודה תחזיר Stream<R> כך שהוא יכיר את פעולת ה־Stream<T> שקדם לו, ויעטוף אותו ביחד עם פעולת ה־map עליו (כמובן שהטיפוס R יכול להיות גם T אך אין הכרח לכך).

שימו לב! לכאורה ניתקנו את הרצף בכך שהחזרנו Stream<R>. יש לדאוג לכך שכל הפעולות שקדמו לביצוע ה־map עדיין יבוצעו באופן עצל, לאחר לאחר ביצוע פעולה טרמינלית.

רמז למימוש מופיע בהמשך.

לדוגמה (הניחו שהטיפוס Cell קיים):

```
auto stream = Stream<int>::of(intPointerVector)
    .map<Cell<int>>([](const int* val) { return new Cell<int>(val); });
```

בדוגמה כל המספרים מופו לטיפוס Cell (והוחזר Stream<Cell<int>>).

- distinct() – 2 פעולות המבצעות פעולה דומה. פעולה שתמחק איברים דומים עוקבים מה־Stream ותשאיר רק איברים ייחודיים, הפעולה תעטוף את ה־Stream שקדם לה.

הפעולות יהיו:

  - פעולה מלאה המקבלת פונקציית השוואה (const T*, const T*) → bool המקבלת 2 מצביעים ו־T. הפונקציה תחזיר true אם האיברים "דומים" ו־false אחרת.
  - פעולה דיפולטית המשתמשת ב־operator== של T ולא של T*. נצלו את הפעולה מלאה וכתבו lambda הממשת התנהגות זו ומתאימה לה. ניתן להניח שבעת שימוש בסוג זה של הפעולה ב־T קיים operator<.

הפעולות יחזירו Stream<T>.

פעולות טרמינליות:

הפעולות הבאות "סוגרות" את ה־Stream הקיים ומפעילות את כל פעולות הביניים שקדמו להן.

- collect() פעולה המקבלת פרמטר גנרי TContainer ומחזירה מופע שלו. הפרמטר הגנרי יהיה אוסף כלשהו של T* ויכיל את איברי ה־Stream לאחר ביצוע כל פעולות הביניים שקדמו לפעולה הזו. ניתן להניח שהאוסף יהיה אוסף המקבל פרמטר גנרי יחיד.

לדוגמה:

```
std::vector<int*> v = Stream<int>::of(intPointerVector)
    .sorted()
    .collect<std::vector<int*>>();
```

בדוגמה נוצר Stream של שלמים הממיין את המספרים ואוסף אותם בחזרה לוקטור.

- forEach() פעולה המקבלת פונקציה T* → void ולא מחזירה דבר. הפעולה תבצע את כל פעולות הביניים על ה־Stream ותפעיל את הפונקציה על כי אבירי ה־Stream הסופיים.

לדוגמה:

```
Stream<int>::of(intPointerVector)
    .map<Cell<int>>([](const int* val) { return new Cell<int>(val); })
    .forEach(&Cell<int>::print);
```

בדוגמה יצרנו Stream של int, מיפינו אותו לטיפוס Cell<int> ולכל איבר כזה הפעלנו את המתודה print. שימו לב: כאן יכולנו להפעיל מתודה המקושרת למופע מכיוון שמתודות מחלקה ב־C++ מוסיפות באופן אוטומטי פרמטר של מצביע לטיפוס של המחלקה (המצביע this) וכך המתודה print שלא מקבלת פרמטרים מתאימה להגדרת הפעולה, שכן בפועל חתימתה void → Cell<int> *.

- reduce() פעולה המזכירה את הפונקציה foldl בשפת ML מהקורס שפות תכנות. הפעולה מקבלת איבר התחלתי מהטיפוס T* ופונקציה (const T*, const T*) → T* ומחזירה T*. הפעולה תפעיל את את כל פעולות הביניים על ה־Stream, ובהינתן האיברים הסופיים $a_1, a_2, ..., a_n$ הפונקציה תחזיר את הערך: $reduce\_func(a_n, reduce\_func(a_{n-1}, (..., reduce\_func(initialT, a_1)))$.

לדוגמה:

```
int initial = 0;
int* sum = Stream<int>::of(intPointerVector)
    .reduce(&initial, [](const int* init, const int* val) {
        int* temp = new int;
        *temp = *init + *val;
        return temp;
    });
```

בדוגמה, בסוף ריצת התוכנית המצביע sum יצביע על משתנה שיכיל את סכום הוקטור.

- max() / min() 2 הפעולות תפעיל את פעולות הביניים על ה־Stream ויחזירו מצביע לאיבר המינימלי/מקסימלי מבין איברי ה־Stream. הניחו שלטיפוס T קיים operator<.

דוגמה:

```
int* min = Stream<int>::of(intPointerVector).min();
```

- count() פעולה המפעילה את כל פעולות הביניים על ה־Stream ומחזירה את מספר האיברים שנמצאים ב־Stream לאחר ביצוע כל הפעולות.

דוגמה:

```
int count = Stream<int>::of(intPointerVector).count();
```

- allMatch() / anyMatch() 2 פעולות המקבלות פרדיקט const T* → bool ומחזירות bool. שתי הפונקציות מבצעות את פעולות הביניים על ה־Stream.
  - anyMatch – מחזירה true אם קיים איבר המקיים את הפרדיקט.
  - allMatch – מחזירה true אם כל האיברים מקיימים את הפרדיקט.

```
bool result = Stream<int>::of(intPointerVector)
    .anyMatch([](const int* a) { return *a == 2; });
```

בדוגמה זו, result יהיה true אמ"מ קיים ב־intPointerVector איבר שערכו 2.

- firstFirst() פעולה המקבלת פרדיקט const T* → bool המבצעת את כל פעולות הביניים על ה־Stream ומחזירה מצביע לאיבר הראשון (שימו לב שיש חשיבות לסדר האיברים) שמקיים את הפרדיקט. במידה ולא קיים איבר כנדרש, הפונקציה תחזיר nullptr.

הערות ורמזים למימוש:

- חלק זה של התרגיל נראה ארוך ועמוס מאוד, אך לא כך המצב – כמעט כל פעולות הביניים והפעולות הטרמינליות קיימות בשפה (בשמות אחרים) ב־STL ותחת ה־header **algorithm**. אתם רשאים להשתמש בהם כרצונכם. החלק הקשה והמסובך במימוש המערכת הוא הפיכתה ל־lazily evaluated.

  פירוט הפעולות בנמצאות ב־algorithm נמצא בלינק

http://en.cppreference.com/w/cpp/algorithm

- השתמשו בפונקציה std::copy המאפשרת העתקת מבני STL ומצביעים רגילים בעזרת איטרטורים.
- בעת מימוש פעולת of היזכרו בחלק הראשון של התרגיל, בו השתמשתם בתכונה חזקה מאוד שהשפה מציעה ולמדתם בתרגול, השתמשו בה גם כאן.
- כלל הנראה תזדקקו ל־closure פשוט של משתנים רגילים. להלן דוגמת קוד המשתמשת ב־closure כזה:

```
void function(int a, std::function<int(int, int)>& func) {
    auto lambda = [a, func]() {
        int b = func(a);
    };

    int c = lambda();
}
```

- מומלץ לייצר מחלקות שיורשות מ־Stream לכל אחת מפעולות הביניים. התייחסו לכל מחלקה כזו כאל מעטפת של הפעולה, ובאתחול מופע שלה שנו את שדה ה־lambda כך שיתאים לפעולה. במקרה של map (וגם שאר הפעולות) העבירו לבנאי המחקלה את עצם ה־Stream הישן כטיפוס גנרי ובכך תמנעו מצורך "לדעת" את הטיפוס של ה־Stream הקודם. במקרה הזה השתמשו במשתנים מסוג auto.

הערות

- בדיקת התרגיל תכלול הן את הפונקציונליות של המערכת כולה והן את הפונקציונליות של כל רכיב בנפרד. ודאו שכל רכיב עומד בדרישות התרגיל, לשם כך מומלץ מאוד לכתוב ולהריץ טסטים.
- עליכם לדאוג לשחרר את כל האובייקטים שהקצתם בחלק ב של התרגיל. על המשתמש לדאוג שלחרור עצמים שהוא יוצר בתוך הפונקציות האנונימיות שהוא מספק. עליכם לדאוג להקצאות שלכם בלבד.
- הקוד יקומפל בעזרת הפקודה:

```
g++ -std=c++11 *.cpp
```

יש לקמפל ולבדוק את ההגשה שלכם על שרת ה־csl3.

במידה וגרסת ה־GCC המותקנת על csl3 לא תומכת ב־C++11 ניתן לעדכן את הקומפיילר באופן הבא:

```
gcc --version # if version is 4.7 or higher, stop
bash
./usr/local/gcc4.7/setup.sh
cd ~
echo ./usr/local/gcc4.7/setup.sh >> .bashrc # this makes the change take effect on every login
```

הוראת הגשה

- בקשות לדחייה מכל סיבה שהיא, יש לשלוח למתרגל האחראי על הקורס (מקסים) במייל עם הכותרת 236703 HW5. שימו לב שבקורס יש מדיניות איחורים, כלומר ניתן להגיש באיחור גם בלי אישור דחייה, פרטים באתר הקורס תחת general info.
- הגשת התרגיל תתבצע אלקטרונית בלבד (יש לשמור את אישור ההגשה).
- יש להגיש זיפ בשם OOP5_<ID1> _ <ID2>.zip המכיל:
  - קובץ בשם readme.txt המכיל את שמות המגישים, מספרי הת"ז וכתובות הדוא"ל בפורמט הבא:

```
name1 id1 email1
name2 id2 email2
```

- תיקייה בשם part1 שתכיל את כל הקבצים שמימשתם בחלק הראשון של התרגיל כולל הקבצים שסופקו לכם.
- תיקייה בשם part2 שתכיל את כל הקבצים שמימשתם בחלק השני של התרגיל.

```
OOP5_123456789_123456789.zip/
    |_ readme.txt
    |_ part1/
        |_ lisp.h
        |_ additional files ...
    |_ part2/
        |_ Stream.h
        |_ additional files ...
```