**תרגיל בית 6**

חלק 1 – שאלות כלליות

1. 1.1. הפקודות האטומיות ב-NIM הן הפקודה הריקה (שורה ריקה), וכל שורה היא פקודה אטומית (בדומה ל-; שיש בסוף כל פקודה ב-C או שורה), וsequencers.

1.2. בשפה ישנם 2 וריאנטים לפקודת ההשמה:

**var** x = 3  
**var** x, y = 3

אך האפשרות השנייה היא בעצם syntactic-sugar . אפשרות זאת בעצם תהפוך לשתי פקודות השמה שונות "מאחורי הקלעים".

1.3. ה-sequencers של השפה הם:, block , break continue ,return.

1.4. בנאי הפקודות ב-NIM הם ה-sequencers שציינו בסעיף קודם.

1.5. 1.5.1. שרשור: ירידת שורה או קריאה לפונקציה.

1.5.2. תנאי: (if,elif) , (case,of,else) , when , continue , break.

1.5.3. לולאה: while , (for,in).

1.6. כיוון ש-NIM בנויה על C, היא בעצם גם כן expression oriented רק שבשונה מ-C, כל ביטוי הוא פקודה ופקודה נגמרת בירידת שורה או נקודה פסיק.

2.

השאלה שבחרתי:

מה היא תכונת צ'רץ' רוסר, ומדוע היא רלוונטית לחישוב בשיטה הנורמלית לעומת השיטה התאוותנית, שדווקא היא נפוצה יותר.

תשובה:

לפי הבנתי, תוכנת צ'רצ רוסר אומרת כי אם קיימים 2 דרכים להשיג אותה תוצאה הע"י סדר שערוך שונה, אז קיימת דרך שלישית המשלבת בין השניים.

היא רלוונטית למושגים "שערוך נורמלי וראווטני" מכיוון שאלו 2 דרכים לשערך ביטוי, הגרסא הראווטנית משערכת את כל הערכים לפני השימוש, בעוד בשערוך נורמלי היא משערכת רק במידת הצורך (אך בכל פעם שנדרשת) וניתן לחשוב על העברת ביטוי והשערוך שלו רק כאשר צריך, בכך למשל אם שולחים אלגוריתם ארוך לפונ' אפס חוסכים את כל האלגוריתם הזה, ומכאן חוסכים בזמן ריצה, ערך נוסף OVERHEAD על הזכרון כדי לבצע שערוך זה.

מכל הנאמר, ניתן להבין שקיימת דרך שלישית לשערוך שהיא משלבת בין השניים. מהערכה שלי מדובר על

partial evalutaiton

3.

2) Eager or Normal-order

#include <stdio.h>  
#include <stdlib.h>

int changeFlag(int \*f){  
 \*f = 0;  
 return 0;  
}

int someFunc(int a){

return 0;

}

int main()

{

int\* flag;

\*flag = 1;

someFunc(changeFlag(flag));

printf("%d",\*flag);

return 0;

}

7) Short circuit of OR or not

fun shortOrNot() =  
let  
 exception notShort  
 fun isNotShort() = raise notShort  
in  
 if 1>0 orelse isNotShort() then 1 else 0  
end handle notShort => 0;

4.

2)

First class function:

הן פונקציות שהן "אזרחיות סוג א' " ניתן להעבירן כארגומנטים לפונקציות,להחזירן כערך ולבצע השמה שלהן למשתנים (או לאחסנן במבנים).

בשפות בהן הפונקציות הן "אזרחיות מסוג א'" , לשמיתוהם אין שום משמעות מיוחדת.

פונקציות מסוג זה הן הכרחיות בשפות פונקציונליות (כגון ML) בהן פונקציונליות מסדר גבוהה הכרחיות, ונקראים גם functorים.

דוגמה לשימוש זה ראינו בפונקציה map בML המקבלת פונקציה ורשימה, ומחזירה רשימה שהפונקציה הופעלה על כל איבר ברשימה.

דוגמאות לשפות התומכות בפונקציות מרמה ראשונה: ML,Haskell,

שפות שאינן תומכות בסוג זה של פונקציות נקראות second class ועל מנת לאפשר העברה של פונקציות משתמשים בטכניקות שונות כמו העברת פונקציה כמצביע לפונקציה , delegates.

Normal order evaluation:

היא אסטרטגיית הערכה של ארגומנטים לפונקציות כך שהשערוך הארגומנטים נעשה רק במידת הצורך (אם נעשה בה שימוש) בתוך הפונקציה, אז נעשה שערוך של הביטוי. ניתן להתייס לצורה זו כהעברת ביטוי לפונקציה בלי לשערכה עד מידת הצורך, זאת בניגוד לEager evaluation שהחישוב נעשה לפני השימוש , ולמעשה נעשה תמיד , וכן בניגוד לLazy evaluation השערוך אינו נשמר ונעשה בכל פעם מחדש, היתרון בכך שאין OVERHEAD על הזכרון, החסרון בכך כמובן הוא שצריך לבצע חישוב חוזר.

ניתן לממש normal evaluation ע"י שימוש בfirst class functions ובכך שנעביר פונקציה המכילה ביטוי שישעורך רק כשמפעילים את הפונקציה, או בשימוש בclosures שמכילים את כל האינפורמציה הרצויה וכל המידע הנחוץ מהסביבה שקראה לפונקציה הנחוצה לחישוב הפרמטר וכך החישוב יתבצע רק כאשר משתמשים בclosure.

לכן ניתן להסיק כי המושגים בלתי תלויים מכיוון ששימוש באחד אינו כופה שימוש באחר, אם כי כפי שהסברנו ניתן לממש את המושג השני באמצעות המושג הראשון, אך לא ניתן לממש את המושג הראשון באמצעות השני, שהרי זמן שיערוך לא יכול להשפיע על אם השפה אוורטוגנולית או לא, כלומר לא נוכל להעביר פונקציות באופן ישיר כפרמטר אם השפה אינה תומכת באפשרות זו, אלא ע"י "תחכום" כמו מצביע לפונקציה ובכך ליצור normal evaluation בכך שאיננו משערכים את הפונקציה ובו זמנית מעבירים את הפונקציה כפרמטר, או שימוש בdelegates (הכרזה על פונקציה כאחת שתוכל להתקבל כפרמטר).

7)

Static typing:

בדיקה סטטית היא בדיקה הנעשית בזמן קומפלציה ע"י אנליזה וניתוח הטקסט של התוכנית (הקוד מקור) ע"י נסיון להוכיח את מערכת הטיפוסים בצורה דטרמניסטית. אם התוכנית עוברת בדיקה סטטית זו אז ניתן להגיד שהיא בטוחה מבחינת קבוצת הטיפוסים בה היא תומכת ובכך נמנעים משגיאות זמן ריצה שיכלו לקרות לתוכנית בעקבות טיפוסיות, ובכך ניתן לטפל בשגיאות אלו עוד בשלב מוקדם במחזור הפיתוח, גם למקומות בהם איננו בהכרח מגיעים אליהם בזמן ריצה, ובכך מבטיחים את נכונות התוכנית גם עבור מקרים ש"קל" לפספס.  
 זהו למעשה מנגנון או כלי המקטין את הסבירות להתכל ב"בעיית העצירה".

Dynamic Typing:

בדיקה דינמית היא בדיקה הנעשית בזמן ריצה על מנת לבדוק את טיפוסיות המשתנים שלא ניתן לבדוק (או לא רוצים לבדוק) באמצעות static typing . בדר"כ ממשים זאת ע"י הצמת "טאגים"(הפניה לטיפוס) עבור אווביקטים המכיל מידע על הטיפוסיות, המכונה RTTI המאפשר גם לממש dynamic dispatch , reflections , late –binding ,downcasting.

אם התוכנית אינה עוברת בדיקה דינמית זו , נזרקת חריגה בזמן ריצה המונעת את המשך התוכנית עם פירוט מהיכן נובעת השגיאה. מכאן שבא יתרונה בכך שאנו מזהים שגיאות בזמן ריצה ויודעים בדיוק מדוע קרו ובכך מונעים שגיאות לוגיות (כמו בשפת C המאפשר type punning), וכן איננו חייבים שהתוכנית תעבוד במלואה בזמן שתרוץ, בדוגמה למערכות web שאינן מצריכות כי כל התוכנית תעבוד במלואה אלא רק החלקים בהם נעשים שימוש. לעומת זאת מכאן גם בא חסרונה שהרי יש לשמור מידע זה בזכרון, ובכך נוצר overhead נוסף על התוכנית ע"מ להבטיח את נכונותה ובטיחותה.

שני המושגים אינם תלויים בשני, ורוב השפות נוקטות בשיטה המאפשרת חפיפה בין השניים (שיטה היברידית).

כמובן שלא ניתן לממש את האחד באמצעות האחר שהרי שהם נבדקים בזמני מחזור שונים בהליך הפיתוח (קומפלציה מול זמן ריצה).

5.

int \* func(){

int var=5;

return &var;

}

בתוכנית להעיל המשנה var "גווע" עם יציאתנו מהscope של הפונקציה func, ולכן אנו מחזירים מצביע לDangling pointer (הצבעה למקום לא חוקי ששוחרר). לא ניתן לכתוב תוכנית דומה בפסקל, מכיוון שבפסקל אין מצביעים, אלא הפניות- ולכן אם נבצע תוכנית דומה(ע"י השמה של משתנה מקומי למשנה מסוג הפניה) הערך יועתק לפני שיגווע (יוצא מן המחסנית).

6.

חלק 2 – איטרטורים ב-JAVA

**לצורך תרגיל זה מימשנו ארבע מחלקות.**

**Qube הוא איטרטור המאפשר לעבור על "קוביות" של מספרים.**

**QubePairs הוא איטרטור היוצר את סכום כל סכומי החזקות השלישיות בסדר אינסופי "אלכסוני".**

**LongPair היא מחלקה המתארת זוג מספרים שלמים.**

**TaxiCabIt היא מחלקה הראשית המדפיסה את המס' מונית השני.**

import java.util.Iterator;

/\*\*

\* iterator of cubes.

\* @author ashiber

\*/

public class Cube implements Iterator<Long>{

private Long i;

public Cube(Long init){

i=init;

}

@Override

public boolean hasNext() {

return i<=Math.cbrt(Long.MAX\_VALUE);

}

@Override

public Long next() {

long temp= getCube();

i++;

return temp;

}

@Override

public void remove() {

throw new UnsupportedOperationException("Not supported yet."); //To change body of generated methods, choose Tools | Templates.

}

public long getCube(){

return i\*i\*i;

}

}

import java.util.Iterator;

/\*\*

\* iterator of sum of diagonal cubes.

\* @author ashiber

\*/

public class CubePairs implements Iterator<LongPair> {

private Cube firstCube;

private Cube secondCube;

public CubePairs() {

firstCube = new Cube((long) 0);

secondCube = new Cube((long) 0);

}

@Override

public boolean hasNext() {

return firstCube.hasNext() && secondCube.hasNext();

}

@Override

public void remove() {

throw new UnsupportedOperationException("Not supported yet."); //To change body of generated methods, choose Tools | Templates.

}

@Override

public LongPair next() {

LongPair temp= new LongPair(this.firstCube.getCube(), this.secondCube.getCube());

if (this.secondCube.getCube() < this.firstCube.getCube()) {

this.secondCube.next();

} else {

this.firstCube.next();

this.secondCube = new Cube((long) 0);

}

return temp;

}

}

/\*\*

\* represent a pair of two long numbers.

\* @author ashiber

\*/

class LongPair {

private long a;

private long b;

public LongPair(long a,long b){

this.a=a;

this.b=b;

}

/\*\*

\* @return the a

\*/

public long getA() {

return a;

}

/\*\*

\* @param a the a to set

\*/

public void setA(long a) {

this.a = a;

}

/\*\*

\* @return the b

\*/

public long getB() {

return b;

}

/\*\*

\* @param b the b to set

\*/

public void setB(long b) {

this.b = b;

}

public long getSum(){

return a+b;

}

}

import java.util.HashMap;

import java.util.Iterator;

/\*\*

\* the main program will calculate the second taxi cab number.

\* @author ashiber

\*/

public class TaxiCabIt implements Iterable<LongPair> {

/\*\*

\* @param args the command line arguments

\*/

public static void main(String[] args) {

TaxiCabIt it = new TaxiCabIt();

HashMap<Long, Integer> taxiMap = new HashMap<>();

int loopCounter=0;

int ifCounter=0;

for (LongPair pair : it) {

loopCounter++;

ifCounter+=2;

if(taxiMap.get(pair.getSum())==null){

taxiMap.put(pair.getSum(),0);

}

taxiMap.put(pair.getSum(), taxiMap.get(pair.getSum()) + 1);

if (taxiMap.get(pair.getSum()) > 1) {

System.out.println(String.format("%d is the taxi cab number , total if's: %d , total iterations:%d", pair.getSum(),ifCounter,loopCounter));

break;

}

}

}

@Override

public Iterator<LongPair> iterator() {

return new CubePairs();

}

}

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Prog-Lang** | **Average time** | **Standard deviation** | **Ifs count** | **Loops count** |
| **AWK** | 0.00045146 | 6.82362689484e-05 | 55 | 65 |
| **BASH** | 0.00057947 | 7.90626909484e-05 | 666 | 523 |
| **Java** | 0.00070549 | 3.63214798707e-05 | 160 | 80 |
| **ML** | 0.00172946 | 0.000319284870296 | 1381 | 563 |
| **Nim** | 0.00045245 | 5.63669007486e-05 | 0 | 13229 |
| **Pascal** | 0.00047774 | 4.88574702579e-05 | 0 | 13238 |
| **Python** | 0.00075707 | 7.08501594917e-05 | 13229 | 13229 |

חלק 3 – מספר המונית: תרגיל מסכם

השתמשנו בשלושה אלגוריתמים שונים:  
  
\*הראשון היה מעבר מספר על כל המספרים מ1 עד 1729 ובדיקה ע"י איטרטורים האם יש שני זוגות של מספרים שהסכום שלהם הוא 1729 והשורש השלישי שלהם הוא שלם.  
\*השני היה באמצעות 4 איטרטורים. IT1 שעולה מ-1, IT2 עולה מ-1 עד ל-IT1. באופן דומה IT3 עולה מ-IT2 עד IT4 שיורד מ-IT1. כלומר מציאת שני זוגות, אחד חיצוני ואחד פנימי שחיבור החזקות שלהם מחזיר את אותו הסכום.  
\*אלגוריתם אחרון היה באמצעות מערך אסוציאטיבי arr. באמצעות זוג איטרטורים a,b יצרנו זוגות של מספרים בשלישית והגדלנו את הערך במערך האסוציאטיבי arr[a^3+b^3]+=1. כשגילינו arr[a^3+b^3]==2, החזרנו את a^3+b^3.  
  
השתמשנו באלגוריתם הראשון בשפות Nim,Pascal,Python. ניתן לראות שכיוון שהאלגוריתם עובר על יותר מספרים (הרבה בדיקות מיותרות), ספירת הלולאות בשלושתם גבוהה מכולם.

האלגוריתם השני בשימוש בשפות BASH,Java,ML. ניתן לראות שיפור משמעותי מבחינת כמות הלולאות שהתוכניות מבצעות. ב-JAVA אנו עובדים עם גנרטורים, לכן הרבה עבודה מתבצעת "מאחורי הקלעים" וספירת הלולאות והתנאים נמוכה מהשאר. למרות השיפור באלגוריתם, הביצועים אינם יותר טובים ואפילו יותר גרועים מהשפות שעובדות עם האלגוריתם הראשון. BASH הוא המהיר מביניהם, אך הוא עובד עם פרשן, לכן הוא עדיין יותר איטי מ-Nim ו-Pascal. על השפות האחרות נפרט בהמשך.

השפות בהן הקוד התפרש בפרשן הן Pyhon ו-ML (למרות שניתן להדר את הקוד ב-Python). ניתן לראות שזמן הריצה של תכניות אלה היה הכי אורך. אנו חושבים שנוסף על כך שהפרשן מפרש בזמן אמת ובודק את התוכנית תוך כדי עבודה, ישנן גם ההדפסות רבות אשר מעקבות את התוכנית.

כמו כן שמנו לב כי JAVA רצה די לאט, למרות שהיא עובדת באלגוריתם השני. אנו מסיקים שדבר זה נובע מכך שהיא עוברת דרך שכבת ה-JVM ובודקת באופן דינמי את הטיפוסיות.

לבסוף המנצחת מכל הבחינות בתחרות הלא הוגנת היא AWK. כיוון שרק בה השתמשנו באלגוריתם האחרון, כמות הלולאות והבדיקות קטנה בה משמעותית מהשאר. הזמן שלה הכי טוב, אם כי לא בהפרש גדול מ-Nim. כלומר המסקנה היא שהשפות אשר מהודרות ועובדות עם טיפוסיות סטטית (Pascal ו-Nim במקרה זה) הן המהירות ביותר, כיוון שהן היו כמעט הכי מהירות למרות שהאלגוריתם שלהן היה הכי פחות טוב.  
  
  
סביבת העבודה שלנו הייתה ב-Ubuntu 16.04 ב-VMware על המחשב האישי. התקנת הקומפיילרים הייתה:

sudo apt-get install nim   
sudo apt-get install fp-compiler-3.0.0   
sudo apt-get install smlnj   
sudo apt-get install default-jre   
sudo apt-get install default-jdk