אופטימיזציות גלובליות

תרגול 12

תזכורת: אופטימיזציות

- **זמן**
- זיכרון
- גישות לdisk
- power

תוך שימור ההתנהגות של התכנית

האופטימיזציות שנציע יהיו שמרניות
- לא נזהה את כל המקרים שניתן לשפר, אבל
- אם ביצענו שינוי כלשהו, מובטח שמשמעות התכנית לא תשתנה.

ככלל, עדיף תכנית פחות יעילה על פני תכנית לא נכונה.

תזכורת: אופטימיזציות

מתי לבצע אופטימיזציות?

- AST
  - יתרונות: לא תלוי מכונה
  - חסרונות: מעט מדי מידע מכדי להועיל
- קוד בשפת הביניים
  - יתרונות: לא תלוי מכונה, חושף מספיק כדי להועיל
  - חסרונות: לעיתים עדיין ניתן לבצע אופטימיזציות ספציפיות למכונה מסויימת, שלא ניתן לבצע בשלב זה.
- קוד המכונה
  - יתרונות: חושף הרבה הזדמנויות לייעול
  - חסרונות: תלוי מכונה, יש לממש לכל מכונה בנפרד
- במהלך הריצה (JIT)
  - יתרונות: ניתן לנצל מידע הקיים רק בזמן ריצה ולא קומפילציה (פרמטרים חוזרים, פונקציות פופולריות)
  - חסרונות: מתבצע על חשבון זמן ריצה

- בשבוע שעבר ראינו אופטימיזציות לוקאליות
  - ברמת הבלוק הבסיסי
  - עם הקשר לשאר התכנית
  - מבוצעות על קוד ביניים
- השבוע נראה:
  - אופטימיזציות גלובליות
  - אופטימיזציות זמן ריצה

אופטימיזציות גלובאליות

- לפעמים אין צורך להסתכל על הקוד של התכנית
- מספיק להסתכל על מבנה התכנית
- נבצע אופטימיזציה על ה-CFG
- נייצר קוד חדש שמייצג את ה-CFG המתוקן

דוגמא - Branch Chaining

הציעו אלגוריתם המבצע את האופטימיזציה Branch Chaining:

ביטול שרשראות של קפיצות שכל אחת מהן קופצת לבאה בתור והחלפתן בקפיצה לכתובת היעד של הקפיצה האחרונה בשרשרת.

הרעיון:

100: x:= y + z
200: goto 300
300: goto 400

→

100: x:= y + z
200: goto 400

Branch Chaining

מתי קפיצה (פקודות goto) מיועדת לסילוק?

- חייבת להיות קפיצה לא מותנית
- וגם, חייבת להיות פקודה יחידה בבלוק הבסיסי שלה
  - היא אחרונה כי תמיד אחרי goto מתחילים בלוק חדש.
  - היא leader כי יש פקודה שקופצת אליה.

בהינתן CFG, נציע אלגוריתם הבא:

1. מצא פקודות "goto x" שנמצאת לבד בצומת v כלשהו.
2. לכל צומת u ב-CFG שיש קשת מ-u ל-v, החלף את כתובת הקפיצה של u לכתובת x.
3. אם אין יותר קשתות נכנסות ל-v, מחק את v מהגרף.
4. חזור ל-1 כל עוד יש מועמדים מתאימים.

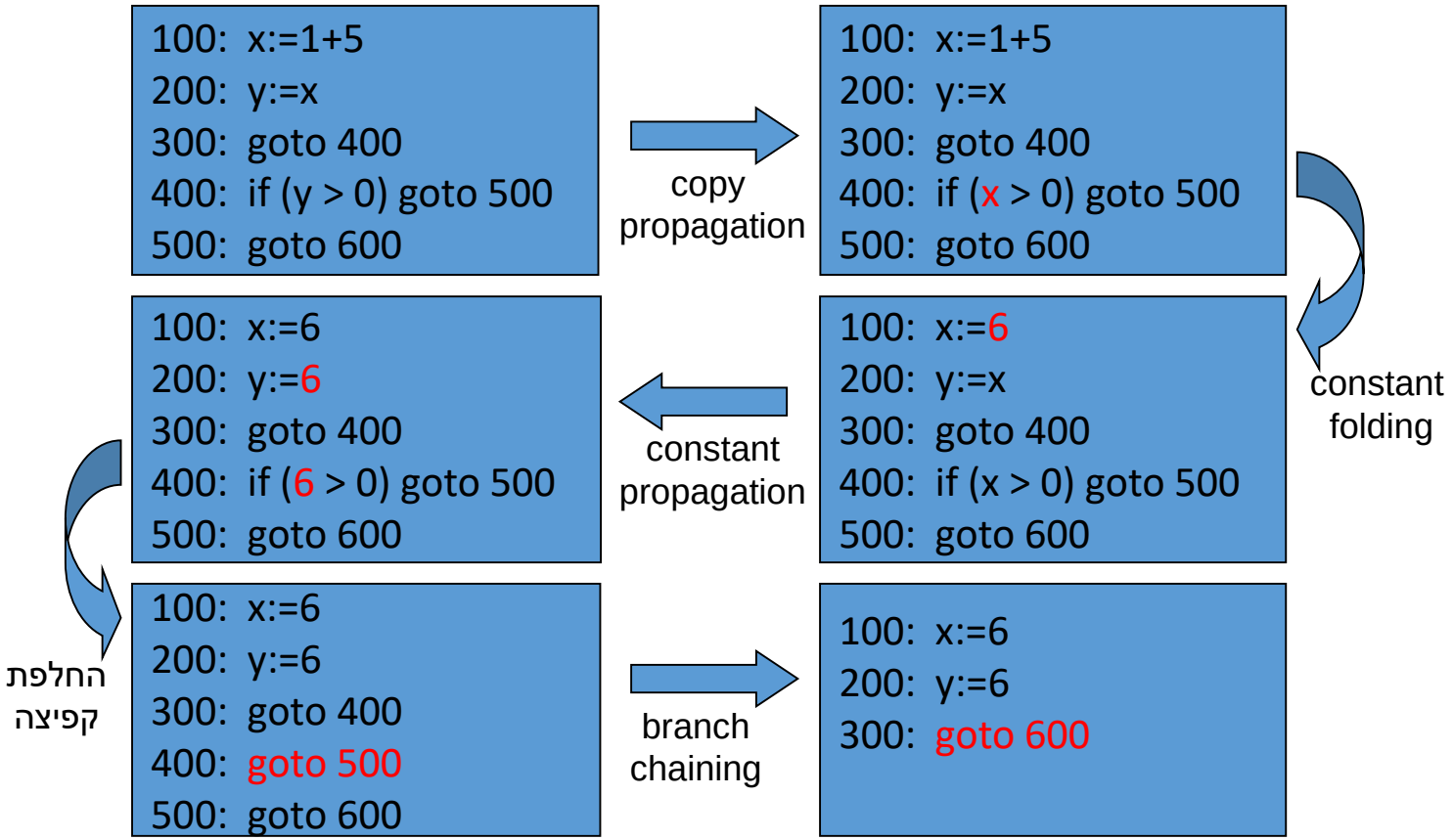
Branch Chaining

שאלה: אילו אופטימיזציות כדאי להריץ לפני Branch Chaining על מנת להגדיל את יעילותה?

תשובה: אופטימיזציות שיעזרו להחליף קפיצות מותנות בקפיצות לא מותנות:

- Copy & Constant Propagation
  - החלפת הופעות משתנה יעד של פקודת השמה במשתנה מקור / קבוע.
- Constant Folding
  - פישוט של ביטויים המורכבים מקבועים בלבד.
- החלפת קפיצות מותנות עם תנאי שיש בו רק קבועים בקפיצות לא מותנות.

דוגמה



ואם אין יותר שימוש ב-x וב-y, ניתן למחוק גם את שתי הפקודות הראשונות
(Useless Code Elimination)

דוגמה ממבחן

פקודה u תיקרא בלתי ישיגה (unreachable) אם u אינה מתבצעת באף הרצה של תכנית.

הציעו אלגוריתם למחיקת פקודות בלתי ישיגות מה-CFG.

פתרון: נבחין שאם פקודה u אינה נמצאת על אף מסלול מהצומת התחילי ב-CFG, לא ייתכן שהיא מתבצעת בריצה כלשהי ⇦ u בלתי ישיגה.

דוגמה ממבחן - המשך

אלגוריתם:

1. בצע סריקה ב-CFG (BFS או DFS) וסמן את כל הצמתים ב-CFG שבהם נתקלת בסריקה.
2. כל צומת שאינו מסומן - מחק מהגרף.

האם הפתרון אופטימלי? כלומר, האם כל הפקודות בלתי-ישיגות יימחקו?

תשובה: לא, למשל:

```
Int x=10
If(x<5) print “Will never Happen”
```

איך אפשר לשפר?

אופטימיזציות בזמן ריצה

JIT

מבנה סכמתי של קומפיילר

- עד עכשיו הנחנו שכל התהליך מתבצע לפני ריצת התכנית
- זה לא בהכרח נכון!

- אנחנו יכולים לבחור האם לבצע מראש או בזמן ריצה

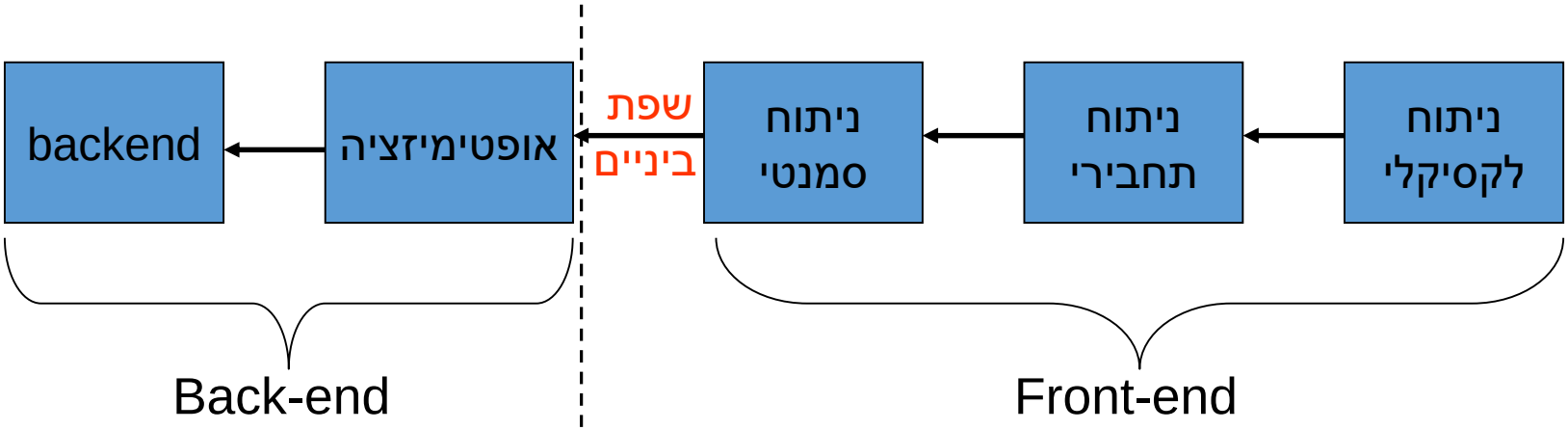
תזכורת: אינטרפרטר (מפרש\מפענח)

- תוכנה הקוראת תוכנית מחשב הכתובה בשפת תכנות ומבצעת אותה ישירות, פקודה אחר פקודה
- במקום קומפיילר שמייצר קוד מכונה – המפענח קורא את הקוד בזמן ריצה ומתרגם אותו לקוד מכונה דינאמית
  - כל חלק ה-frontend עדיין חייב להתבצע (למה?)
- נהוג בשפות סקריפט (python, javascript)
- התוצאה: קוד ללא אופטימיזציות (למה?)

מבנה סכמתי של קומפיילר

- עד כמה הקוד מקומפל? יש ספקטרום.
- בצד אחד: קומפיילר קלאסי
  - הכל מתבצע מראש
  - נהוג בשפות C, C++, וכו'
- בצד השני: הכל מתבצע בזמן ריצה, interpreter
- ריצה של קוד מקומפל מהירה יותר אבל קימפול הקוד דורש זמן ומשאבים

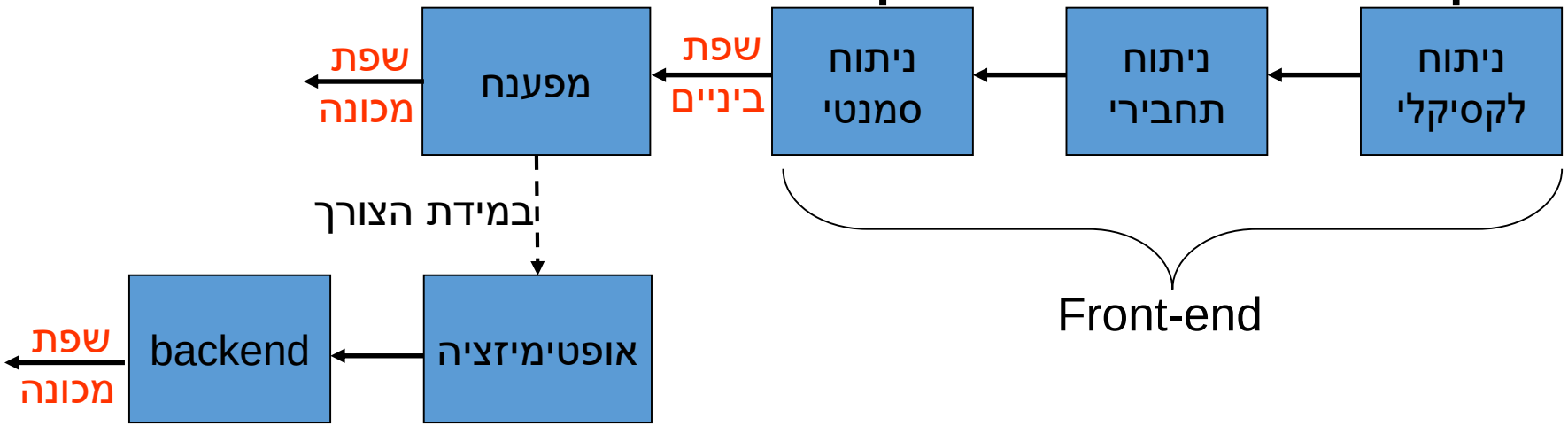
מבנה סכמתי של קומפיילר



- אפשר גם אחרת! להתמקם באמצע הספקטרום.
- נבצע חלק מהעבודה מראש וחלק נשאיר לזמן ריצה
  - כך מקומפל קוד בשפת ג'אווה
  - compiler JIT

Just in time compilation

- הקומפילציה נעשית תוך כדי ריצת התוכנית.



- **מטרה עיקרית:**

  שילוב המהירות של קוד שקומפל מראש עם הגמישות של תרגום תוך כדי ריצה

  - מאפשר אופטימיזציות חזקות יותר
    - כחלק מתהליך הקומפילציה, לא בפענוח
    - למשל בהתבסס על טיפוסים וערכים של משתנים שידועים רק בזמן ריצה

JIT for Java

- **לפני הריצה:**
  - הקומפיילר של java מקמפל את הקוד לשפת ביניים (java bytecode)
    - מבצע בדיקות לקסיקליות, תחביריות וסמנטיות
    - לא נהוג שהקומפיילר מבצע אופטימיזציות על הקוד
    - למרות שיכול לבצע לפחות חלק מהן
  - הקוד המיוצר לא תלוי במערכת עליה ירוץ (מדוע?)

JIT for Java

- **בזמן ריצה:**
  - את הbytecode ניתן להריץ ישירות (interpreted)
  - נעשה זאת כל עוד זה משתלם
- תזכורת: האיזון הוא בין הנוחות של אינטרפרטר ליעילות של קוד מקומפל

מאוד משתלם
זמן לביצוע ה-backend: 5sec
זמן לביצוע ללא אופטימיזציות: 20sec
מספר הפעמים שהפונקציה תרוץ: 50

???

לא משתלם
זמן הרצת ה-backend: 5sec
זמן לביצוע ללא אופטימיזציות: 1sec
מספר הפעמים שהפונקציה תרוץ: 1

משתלם לקמפל

לא משתלם לקמפל

JIT for Java

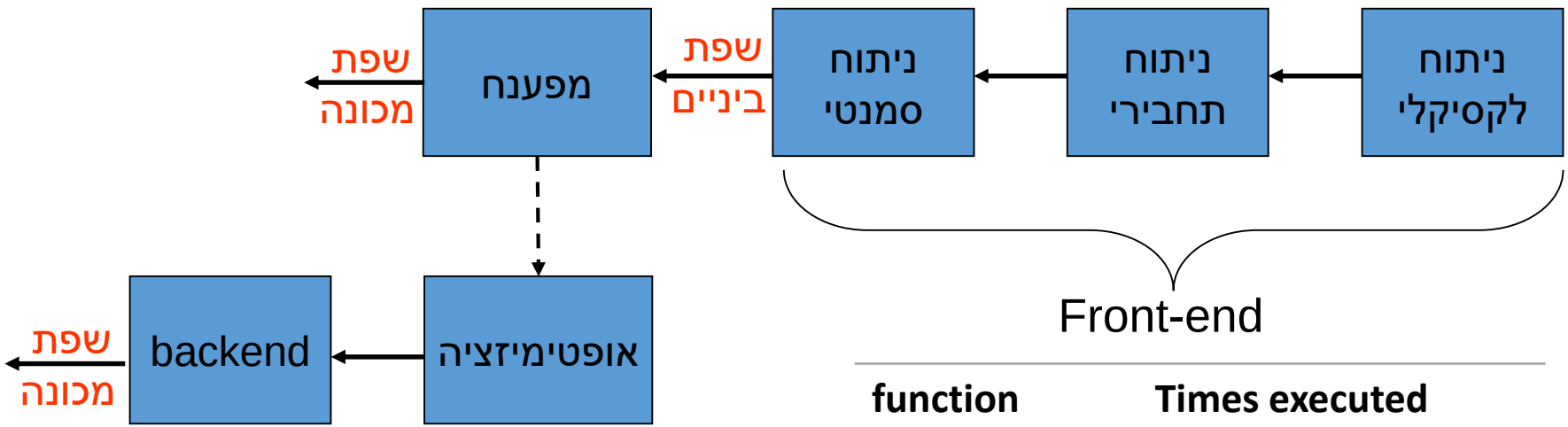
- **כיצד נחליט אם משתלם לקמפל?**
  - סביבת הריצה זוכרת כמה פעמים השתמשנו בכל פונקציה
  - קוד שמשתמשים בו הרבה (מעל רף קבוע מראש) מקומפל לקוד מכונה והמונה עבורו מאופס (למה בהמשך)

JIT for Java

```
void f(int x) { … }
void g(int n) {
    f(n);
    f(n-1);
    f(n-2);
      ⋮
```

```
_call_f:
#get x
t1 = (-1)fp
…
```

```
void f(int x)
{ … }
```




```
_call_f:
0010001010
…
```

function	Times executed
void f(int x)	1 2
void g(int x)	

JIT for Java

- **מה עכשיו?**
  - נשתמש בקוד שקימפלנו ונמשיך למנות שימושים
  - בכל פעם שהמונה עבור קוד מסויים עובר את הרף, הקוד מקומפל שוב ברמת אופטימיזציה גבוהה יותר (עד הרמה המקסימלית)

JIT for Java

- פונקציות לא מקומפלות בפעם הראשונה שהן נקראות
  - מאד יקר לקמפל את כל הפונקציות בעליית התכנית
- נשקיע את המשאבים הכרוכים בקימפול ואופטימיזציות רק עבור קוד שמבוצע הרבה
  - לרוב פונקציות נפוצות יקומפלו קרוב לתחילת הריצה
- על פונקציות שמקומפלות ניתן להפעיל אופטימיזציות חזקות יותר מאשר אם היינו מקמפלים הכל מראש
- מצד אחד מאט את התכנית, מצד שני יכול גם להאיץ
  - באיזה סוג קומפיילר כדאי להשתמש תלוי בהיכן ואיך תרוץ התכנית

שאלה ממבחן – אביב 2013 א

בשאלה זו הנך מתבקשת להסביר איך הקומפיילר יכול לנצל תוספת לשפת התכנות על מנת ליצור **קוד יעיל יותר**.

עליך להסביר האם תמיד ניתן להפעיל את האופטימיזציה שלך או שהיא תלויה בתכונות נוספות כלשהן. כמו כן, הסבירי אם הטכניקה שלך תמיד תשפר את זמן הריצה או שיתכן והיא תשפיע עליו לרעה – ובאילו תנאים.

בשאלה זו נתמקד ב-**Just-in-time compiler (JIT)**.

שאלה ממבחן – אביב 2013 א

1. (15 נק') נניח שלשפת התכנות מוסיפים מילה שמורה חדשה **oftensame** שניתן להוסיף לפרמטרים של פונקציה. למשל:

```
int foo(float f, oftensame int k, int x)
```

המשמעות של המילה השמורה היא שהמתכנת מצפה שהפונקציה foo תיקרא מספר רב של פעמים כאשר הפרמטר k יכיל **תמיד** ערך מתוך מספר קטן של **ערכים אפשריים**. אבל, **לא ניתן** להניח שקבוצת הערכים של k ידועה בזמן קומפילציה, ואין להניח חסם כלשהו על מספר הערכים האפשריים. כאשר ערך מסוים של k מופיע בחלק גדול של הקריאות ל-foo, נגיד שהערך הזה הוא נפוץ. תארי כיצד קומפיילר JIT יכול להשתמש בידע של הגדרת oftensame על מנת **לשפר את זמן הריצה של התכנית** כאשר היא נקראת עם אחד **הערכים הנפוצים** של k. אין צורך לשפר את זמן הריצה במקרה שהקריאה ל-foo היא עם ערך k שאינו נפוץ (ותתכן אפילו פגיעת ביצועים במקרה כזה).

שאלה ממבחן – אביב 2013 א - פתרון

1. על הקומפיילר להגדיר מפה, שהמפתח שלה הוא ערך k שחוזר על עצמו, ממופה אל קוד מקומפל של הפונקציה עבור אותו הk.

כיצד משתפר זמן הריצה?

- מכיוון שנתון שהפונקציה חוזרת מספר רב של פעמים, יהיה יותר יעיל להריץ קוד מקומפל, מאשר להשתמש בinterpreter.
- בקוד המקומפל k קבוע, ולכן נוכל להפעיל יותר אופטימיזציות כגון Constant Propogation.

שאלה ממבחן – אביב 2013 א

2. (15 נק') עכשיו הניחי שניתן להגדיר את **oftensame** על כל הפרמטרים של פונקציה. למשל:

```
int foo(float f, int k, int x) : oftensame
```

כאשר הכוונה היא שהמתכנת מצפה שהפונקציה foo תיקרא מספר רב של פעמים כאשר יהיה סט אחד נפוץ של ערכים לכל הפרמטרים שיופיע **ברוב הקריאות**. למשל ערכים $\{v_1,v_2,v_3\}$ שיופיעו ברוב הקריאות (כלומר רוב הקריאות יהיו מהצורה $foo(v_1,v_2,v_3)$). הערכים האלה **אינם ידועים בזמן קומפילציה**. כמו בסעיף הקודם, גם כאן כל אחד מהערכים נבחר מתוך מספר קטן של ערכים אפשריים שאינם ידועים בזמן קומפילציה.

שאלה ממבחן – אביב 2013 א - פתרון

2. מכיוון שהשלשה f,k,x מגדירה באופן חח"ע את ערך הפונקציה, הקומפיילר יוכל לשמור במפה את השלשה f,k,x כמפתח אשר ממופה לערך ההחזרה של הפונקציה.

במקרה שלפונקציה יש side effects כגון print, ניצור קוד מקומפל באמצעות JIT.