פרויקט - חלק 3

בפרויקט זה נממש את הקומפיילר משפת --C, שהכרנו בחלקים הקודמים של הפרויקט, לשפת המכונה Riski. נשתמש בלינקר כדי לשלב מספר קבצי שפת מכונה לקובץ ריצה. את קובץ הריצה נוכל להריץ על המכונה הוירטואלית (VM) RX.

לדוגמה, נניח שיש לנו תכנית המורכבת משני מודולים (קבצי קוד מקור): myprog.cmm, extra_funcs.cmm . הקובץ myprog.cmm הוא הקובץ הראשי המכיל את ה-main של התכנית ואולי כמה פונקציות עזר, והקובץ השני מכיל עוד מספר פונקציות שנדרשות בקובץ הראשי.

כדי לקמפל נקרא לקומפיילר (שאתם תבנו) עבור כל אחד מהקבצים:

```
rx-cc myprog.cmm
rx-cc extra_funcs.cmm
```

לאחר הקומפילציה הנ"ל נקבל שני קבצים בשפת המכונה Riski שיצר הקומפיילר: `myprog.rsk` ו- `extra_funcs.rsk` הכוללים, בנוסף לקוד המכונה, מידע לטובת שלב הלינקר.

נקרא ללינקר (שיסופק לכם) באופן הבא:

```
rx-linker myprog.rsk extra_funcs.rsk
```

הלינקר יקשר בין הקריאות לפונקציות למימוש הפונקציות ויפיק קובץ ריצה בשם הקובץ הראשון (הראשי) עם סיומת e. , כלומר במקרה הזה נקבל את myprog.e .

את קובץ הריצה נוכל להריץ על ידי קריאה למכונה הוירטואלית (שתסופק לכם):

```
rx-vm myprog.e
```

ממשק הקומפיילר:

```
rx-cc <input_file>
```

1. שם קובץ הריצה של הקומפיילר: `rx-cc`
2. פרמטר יחיד בשורת הפקודה: שם של קובץ קלט קוד מקור יחיד. הסיומת של שם הקובץ חייבת להיות `.cmm` עבור קבצי מקור בשפת --C .
3. פלט: הקומפיילר מייצר קובץ באותו השם אך עם סיומת rsk. במקום cmm. שמכיל את התוכנית המתורגמת לשפת ה Riski, אם התוכנית חוקית. אם התוכנית אינה חוקית , יש להציג הודעת שגיאה (אין לייצר קובץ פלט במקרה של תוכנית לא חוקית). כמו-כן, קובץ rsk. מכיל מידע נוסף בראשיתו (header) לטובת הלינקר, כפי שיפורט בהמשך.

תיאור השפה --C

שפת --C מוגדרת באופן זהה להגדרות בחלקים הקודמים של הפרויקט. בפרט, האסימונים מוגדרים בחלק 1, והתחביר מוגדר בחלק 2.

התאמות הדקדוק:

1. פתרון הדו-משמעויות בדקדוק ייעשה באמצעות מנגנון הגדרת קדימות/אסוציאטיביות האסימונים של Bison בלבד ואין לשנות את הדקדוק, כפי שנדרשתם בחלק 2.
2. עבור דו משמעות באופרטורים אריתמטיים/לוגיים/בוליאניים יש ליישם קדימויות ואסוציאטיביות כמקובל ב-C/C++ . ניתן להעזר בסיכום הקדימויות בקישור הבא:
http://en.cppreference.com/w/cpp/language/operator_precedence
3. בטיפול בדו-משמעות מסוג dangling else יש ליישם (ללא שינוי בדקדוק!) הצמדת ה-else ל-if הקרוב.
4. בטיפול בדו-משמעות בהקשר של פעולת explicit cast יש להחיל את ה-cast של הטיפוס המבוקש על ה-EXP הקרוב הבסיסי ביותר (כלומר, עדיפות גבוהה יותר לפעולת cast מאשר להרכבת EXP מורכב יותר מה-EXP הצמוד ל-cast).
5. <u>השינוי היחיד המותר בדקדוק המוגדר בחלק 2</u> הוא שימוש ב"מרקרים" (כללי אפסילון), בדומה לנלמד בכתה (למשל: `M→ε { M.quad := nextQuad }`).
6. מותר שימוש (זהיר) בכללים סמנטיים פנימיים, לדוגמה:
```
B: X {some_mid_action(); } Y {rule_action();}
```
לפרטים עיינו בתיעוד של Bison בפרק שנקרא Actions in Mid-Rule .

כללים סמנטיים:

מעבר לפעולות ליצירת הקוד, יש כמובן לאכוף בקומפיילר שלכם כללים סמנטיים. למעט הדגשים המפורטים להלן, הסמנטיקה זהה לסמנטיקה של שפת C :

1. לא ניתן לבצע פעולות אריתמטיות או השוואתיות בין טיפוסים מסוגים שונים. עבור הסוגים הבסיסיים (integer ו-real) התכנית צריכה להכיל המרות מפורשות (explicit cast) בשימוש בתחביר המתאים (כמו ב-C) על מנת שפעולות יתבצעו בין ביטויים מטיפוסים זהים.
2. לא ניתן לבצע המרות מטיפוסים מורכבים (defstruct), אך ניתן להמיר התייחסות לשדה מסוים בטיפוס מורכב שהינו מטיפוס בסיסי.
3. פעולות אריתמטיות או השוואתיות אפשריות רק עבור הטיפוסים הבסיסיים. פעולות על טיפוסים מורכבים הן שגיאה סמנטית, אולם השמות אפשריות גם בין טיפוסים מורכבים (זהים) – ראה הסבר על הסמנטיקה הרלוונטית לכך בהמשך המסמך.
4. בקריאות לפונקציה הפרמטרים המועברים צריכים להיות מאותו טיפוס של הפרמטרים המוגדרים. בהתאם, התכנית חייבת להכיל המרות מפורשות של ביטוי לטיפוס המתאים לפרמטר אם הם שונים, כנ"ל. כל הפרמטרים לפונקציה חייבים להיות מטיפוס בסיסי.
5. כל חוסר התאמה של טיפוס בין אופרנדים/פרמטרים שלא טופל כנ"ל מהווה שגיאה סמנטית.
6. קבועים מספריים שאין בהם נקודה הם מטיפוס integer, אחרת הם מטיפוס real.
7. יש לנהל טבלאות סמלים עם תיחום סטטי (static scoping), כלומר, משתנה מוגדר בתחום של ה-BLK שבו הוגדר.
   - תחילת פונקציה (או main) מאתחלת טבלת סמלים ריקה.
   - הפרמטרים של פונקציה שייכים לתחום (scope) של הבלוק הראשי של הפונקציה, כאילו היו חלק מה-var שבתחילתה.
   - הגדרת משתנה ב-BLK פנימי ממסכת משתנה בעל שם זהה שהוגדר בבלוק שמכיל אותו.
   - אסור, כמובן, להגדיר יותר ממשתנה אחד בעל אותו שם באותו הבלוק (זו שגיאה סמנטית).
   - אין משתנים גלובליים

טיפוסים מורכבים (defstruct):

1. ניתן להגדיר טיפוסי מידע מורכבים בדומה ל-struct ב-C, באמצעות defstruct. בהתאם להגדרת התחביר, יש להגדיר את כל הטיפוסים המורכבים בחלק הראשון של קובץ קוד המקור. הטיפוסים המורכבים מוכרים מסיום ההגדרה שלהם עד לסוף הקובץ.
2. השדות של טיפוס מורכב יכולים להיות מטיפוס בסיסי (integer או real) ויכולים להיות גם מטיפוס מורכב אחר שהוגדר לפני כן, אך לא מהטיפוס של עצמם.
3. התייחסות (reference) לשדה של משתנה מורכב נעשית באמצעות ציון שם השדה בסוגריים מרובעים. לדוגמא, אם הגדרנו:

```
defstruct {
    foo : integer;
    bar : real;
} fbar;
```

אזי עבור משתנה g מהטיפוס fbar ניתן לבצע השמה לשדה bar כך:

```
g[bar] = 2.5;
```

ניתן להתייחס לשדה במשתנה מורכב גם כ-Rvalue , כמובן.

4. התייחסות לשדה לא קיים/מוגדר הינה שגיאה סמנטית.
5. ניתן לבצע השמה ממשתנה מטיפוס מורכב למשתנה אחר מאותו טיפוס. פעולת השמה זו בעלת סמנטיקה של העתקה פשוטה, כלומר, כל השדות במקור מועתקים לשדות היעד המתאימים.
6. **לא** ניתן להעביר משתנה מטיפוס מורכב כפרמטר של פונקציה, ו**לא** ניתן להחזיר טיפוס מורכב מפונקציה כערך החזרה.

שפת המטרה: שפת ה – Riski

Riski היא שפה דמוית ASM. הפורמט המחייב של התוכנית הוא:

1. הוראה אחת בכל שורה.
2. ההוראות עצמן תמיד באותיות גדולות .
3. קוד ההוראה והארגומנטים מופרדים ע"י תו רווח אחד לפחות .
4. בכל תוכנית, הוראת ה HALT מופיעה בשורה האחרונה אותה יש לבצע.
5. יש לציין בפקודות קפיצה (BREQZ, BNEQZ, JLINK, UJUMP) את מספר הפקודה אליה קופצים (לא כתובת). מספרה של הפקודה הראשונה הוא **1**.

שפת ה RISKI תומכת בפקודות הבאות:

Opcode	Arg	Description
COPYI	A B	A=B
PRNTI	B	Prints the value of B
READI	A	Read an integer into A
SEQUI	A B C	If B=C then A=1 else A=0
SNEQI	A B C	If B!=C then A=1 else A=0
SLETI	A B C	If B<C then A=1 else A=0
SGRTI	A B C	If B>C then A=1 else A=0
ADD2I	A B C	A=B+C
SUBTI	A B C	A=B-C
MULTI	A B C	A=B*C
DIVDI	A B C	A=B/C
LOADI	A B C	A = IMem[B+C]
STORI	A B C	IMem[B+C] = A
COPYR	D E	D=E
PRNTR	E	Prints the value of E
READR	D	Read a real into D
SEQUR	A E F	If E=F then A=1 else A=0
SNEQR	A E F	If E!=F then A=1 else A=0
SLETR	A E F	If E<F then A=1 else A=0
SGRTR	A E F	If E>F then A=1 else A=0
ADD2R	D E F	D=E+F
SUBTR	D E F	D=E-F
MULTR	D E F	D=E*F
DIVDR	D E F	D=E/F
LOADR	D B C	D=RMem[B+C]
STORR	D B C	RMem[B+C] = D
CITOR	D B	D= real(B)
CRTOI	A E	A= integer(E)
UJUMP	L	goto L
JLINK	L	I0 =address of next instruction goto L
RETRN		goto I0
BREQZ	B L	If(B=0) goto L
BNEQZ	B L	If(B!=0) goto L
PRNTC	C	Prints a character of the ASCII value of C
HALT		Stop

הערות לטבלה הנ"ל :

1. A,B,C,D,E,F הם סימנים מופשטים , שיכולים לציין רגיסטר או קבוע כלשהו:
   - A מציין רגיסטר שלם (מטיפוס integer). B,C מציינים רגיסטרים או קבועים שלמים.
   - D מציין רגיסטר ממשי (מטיפוס real). E,F מציינים רגיסטרים או קבועים ממשיים.
2. כל הפקודות פרט ל-LOAD ו-STOR פועלות על רגיסטרים בלבד.
3. L מציין תווית (מספר שורה) .

משאבי זכרון ורגיסטרים

1. כתובת זיכרון (הערך של B+C עבור IMEM[B+C], RMEM[B+C]) הינה מספר שלם בין 0 ל-999.
2. אין לקרוא מהזיכרון באמצעות פקודת LOAD שאינה תואמת לסוג ה-STOR ששימש לכתיבה לאותה כתובת (למשל, אסור שימוש ב-LOAD**R** עבור כתובת שנכתבה באמצעות STOR**I**). קריאת ערך מהזיכרון בפקודה שאינה תואמת לסוג הנכתב תחזיר **ערך לא מוגדר**.
3. המעבד מכיל 1000 רגיסטרים מכל סוג, בעלי המספרים הסידוריים 0-999. רגיסטרים עבור integer יסומנו באות **I** ורגיסטרים מסוג real יסומנו באות **R**. למשל: I29, R17 .
   שימו-לב: הרגיסטרים צריכים לשרת רק את המשתנים הזמניים שמקצה הקומפיילר. אין להשתמש בהם (באופן קבוע) לטובת משתנים שמוצהרים בתכנית המקומפלת.
4. הרגיסטר השלם הראשון I0 שמור לטובת שמירת כתובת החזרה בפקודות JLINK + RETRN.
5. יש לשמור (להקצות) רגיסטרים שלמים (מטיפוס integer) נוספים על מנת לנהל את המחסנית ואת רשומת ההפעלה. רצוי להקצות את הרגיסטרים השמורים ברגיסטרים הנמוכים ואת ההקצאות המשתנות למשתנים הזמניים לבצע מעל התחום השמור, בהתאם לתוכנית המקומפלת.
6. אין צורך לטפל בהקצאה יעילה של רגיסטרים. התוכניות ירוצו כל עוד לא יחרגו ממספר הרגיסטרים הנ"ל. ניתן להניח שבמהלך פונקציה אחת (כולל main) לא יהיה צורך ביותר מ-900 רגיסטרים (משתנים זמניים) עבור הקוד של אותה הפונקציה, אולם שימו-לב שפונקציה יכולה לקרוא לעצמה או לכל פונקציה אחרת.
7. הניחו כי יש מספיק מקום בזיכרון המחסנית לצורך אחסון רשומות הפעלה ושמירת רגיסטרים במהלך שרשרת קריאה לפונקציות. אין צורך לבדוק גלישה של המחסנית מעבר לקצה מרחב הזיכרון.

מימוש write בשפת --C:

פקודת write בשפת --C משמשת להדפסת ערכים לפלט הסטנדרטי. יש לממש אותה באמצעות הפקודות *PRNT המתאימות. במקרה של הדפסת מחרוזת, יש להשתמש במספר קריאות מתאים לפקודה PRNTC. כמו-כן, יש לטפל בהמרה של הצירופים המיוחדים "`\n`" (שורה חדשה : LF) ו-"`\t`" (טאב : TAB) לערך הASCII המתאים. כלומר, רצף של שני תווים כאלו במחרוזת יש להמיר לתו המתאים בפלט ולא להדפיס לוכסן ואות – בדומה לשימוש בצירופים הללו ב-`printf` בשפת C.

חישוב יעדי הקפיצה והטלאה לאחור:

בקוד ה - Riski שמייצר המהדר עשויות להופיע פקודות קפיצה, כאשר יעד הקפיצה הוא מספר השורה. זכרו כי השורה הראשונה **בתכנית** המיוצרת מקובץ המקור הנוכחי היא שורה מספר 1.

לצורך חישוב יעדי הקפיצה , יש להשתמש בהטלאה לאחור (backpatching) ובמרקרים מסוג M→ε , N→ε.

קריאות לפונקציות:

השפה מאפשרת קריאה לפונקציות המוגדרות וממומשות בראש קובץ התוכנית הראשי (היכן שנמצא main) או בקובץ נפרד. הפונקציות מקבלות מספר כלשהו של פרמטרים (בהתאם להגדרה) עפ"י ערכם (by value) ומחזירות ערך החזרה יחיד.

להזכירכם, הדקדוק מכיל הצהרות על פונקציות מהצורה:

```
FDEFS -> FDEFS TYPE id ( FUNC_ARGLIST_FULL ) BLK
      | FDEFS extern TYPE id ( FUNC_ARGLIST_FULL ) ; | ε
```

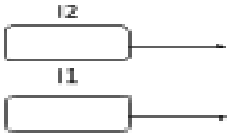
שימו לב שלמרות שהדקדוק מאפשר להעביר פרמטרים מכל סוג שהוא, השפה (סמנטיקה) מתירה העברה של פרמטרים מסוגי הבסיס בלבד (integer או real).

במקרה שנעשתה קריאה לפונקציה שאינה מוצהרת או מוגדרת בקובץ עד לשורת הקוד הקורא, על הקומפיילר להכריז על שגיאת קומפילציה (סמנטית). גם קריאה לפונקציה עם פרמטרים מסוג לא מתאים להגדרת הממשק שלה מהווה שגיאת קומפילציה (סמנטית). שימו-לב כי פונקציה יכולה לקרוא לעצמה, כלומר, פונקציה מוכרת מרגע שהוצהר על הממשק שלה, גם אם לא הסתיים הבלוק שמכיל את קוד המימוש שלה. מומלץ להשתמש בטבלת סמלים מתאימה על מנת לשמור את כתובת ההתחלה של הפונקציה (מספר שורה בתוכנית) ונתונים נוספים הנדרשים לשימוש בפונקציה בקוד ואימות שימוש נכון בפרמטרים (בדומה לטבלת סמלים עבור משתנים).

ניתן לקרוא לפונקציה מספר פעמים במהלך התוכנית, כולל קריאות רקורסיביות. בהתאם, השתמשו ברשומת הפעלה בדומה לנלמד בכתה וברגיסטרים שמורים על מנת לנהל אותה. ניתן לתכנן את רשומת ההפעלה כרצונכם בתנאי שהיא עומדת בדרישות הנ"ל. ***שימו-לב***: בעת קריאה לפונקציה, כל רגיסטר יכול להשתנות. לכן עליכם להתכונן היטב למעבר לפונקציה.

להלן המלצה לאפשרות אחת לניהול רשומת ההפעלה, אולם היא אינה מחייבת:

רגיסטר I0 יכיל את כתובת החזרה מהפונקציה (מחויב מפקודת JLINK). רגיסטר I1 יכיל את כתובת בסיס רשומת ההפעלה (stack frame) ו- I2 יכיל את המצביע לראש המחסנית. לדוגמה:



Callee locals…
Callee saved registers…
Function return value
Arg1
Arg2
…
Caller’s saved registers…
…

הצבות בשפת Riski:

בשפת Riski יש צורך בהמרה כאשר מציבים מרגיסטר שלם לתוך רגיסטר ממשי, למשל אם נייצר פקודה כזו:

```
COPYR R1 I1
```

נקבל **שגיאת ריצה** מסימולטור ה-RX. יש צורך בהמרה מפורשת:

```
CITOR R2 I1
COPYR R1 R2
```

ובאופן דומה יש לעשות שימוש בפקודת CRTOI.

יש לזכור שהבדיקות הנ"ל רלוונטיות רק בין רגיסטרים. גישות לזיכרון לא יכולות לבדוק את הסוג שנשמר בכתובת מסוימת ושני הטיפוסים תופסים תא בודד בזיכרון. קריאה מהזיכרון עם פקודה של רגיסטר מטיפוס שונה מהטיפוס ששימש לכתיבה לזיכרון תחזיר ערך לא מוגדר מהזיכרון ולא תגרום לשום שגיאת ריצה. המרת סוגים מחייבת קריאה לרגיסטר מהסוג שנכתב ושימוש בפקודת המרת סוג בין רגיסטרים.

התמודדות עם שגיאות קומפילציה והודעות שגיאה:

אין הנחת קלט תקין בפרויקט, כלומר יש להתמודד עם שגיאות. במקרה של גילוי שגיאה בזמן הקומפילציה, יש לעצור את הקומפילציה, להוציא לפלט השגיאות הסטנדרטי (stderr/cerr) הודעת שגיאה, ולצאת מהקומפיילר עם החזרת קוד יציאה/שגיאה כמפורט להלן:

1. עבור שגיאה לקסיקלית, קוד שגיאה 1 והודעה במבנה הבא:

```
Lexical error: ’<lexeme>’ in line number <line_number>
```

2. עבוד שגיאה תחבירית, קוד שגיאה 2 והודעה במבנה הבא:

```
Syntax error: ‘<lexeme >’ in line number <line_number>
```

כאשר `<lexeme>` הינה הלקסמה הנוכחית בעת השגיאה.

3. עבור שגיאה סמנטית, קוד שגיאה 3 והודעה במבנה הבא:

```
Semantic error: <error description> in line number <line_number>
```

כאשר `<error description>` הינה הודעת תיאור השגיאה כרצונכם, ו-
`<line_number>` הוא מספר שורת השגיאה בקוד המקור.

4. עבור שגיאות אחרות בזמן ריצת הקומפיילר (למשל, שם קובץ קלט לא קיים, כישלון בהקצאת זיכרון וכו'), קוד שגיאה 9 והודעה במבנה הבא:

```
Operational error: <error description>
```

אין לייצר קובץ פלט במקרה של שגיאה!

תמיכה ב-Linker

כמו בשפת C, נרצה לאפשר הפרדה של הקוד למספר קבצים ולקמפל כל אחד מהם בנפרד. ה-linker מאפשר לחבר את הקבצים הללו לקובץ ריצה (executable) בודד אשר ירוץ במחשב הנדרש. בתרגיל זה נשתמש ב-linker המסופק לכם (ראו הסבר בסוף התרגיל) אשר יצור קובץ ריצה בודד עבור ה-RX ונאפשר שימוש בפונקציות המוגדרות בקבצים שונים.

לצורך תמיכה ב-Linker על הקומפיילר להוסיף header בעל חמש שורות בדיוק (כמתואר בהמשך) שיכיל מספר פרטים נחוצים לצורך חיבור הקבצים.

ה-linker ישתמש במידע זה כדי לקשר בין הקריאות לפונקציה לבין מיקום הפונקציה בקובץ הריצה המאוחד. בנוסף, ה-linker יתקן את כל הקפיצות האבסולוטיות שנעשו בקוד ויוסיף את ההיסט הדרוש לפי המיקום החדש של הקוד בקובץ המאוחד.

שימו לב:
הלינקר "מתקן" כתובות קפיצה קיימות, ולא מוסיף כתובות בקוד, כמו בתהליך הטלאה. לכן, בכל פקודת קפיצה לפונקציה חייבת כבר להיות כתובת קפיצה. במקרה של קפיצה לפונקציה באותו הקובץ, זו תהיה השורה של יעד הקפיצה כאילו זה היה הקובץ היחיד (השורה שבה מתחילה פונקציית היעד באותו קובץ). במקרה שהקפיצה לפונקציה הממומשת בקובץ אחר, ניתן לרשום כל כתובת יעד שלינקר יוכל להחליף (רצוי שזו תהיה כתובת לא חוקית, על מנת לזהות תקלות בתהליך שמבצע הלינקר).

מבנה ה-header

```
<header>
<main> OR <empty>
<unimplemented> func1,L1,L2,L3,... func2,L1,L2,L3,... …
<implemented> func1,L1 func2,L1 …
</header>
```

כאשר `func` הוא שם פונקציה ו-`L` היא השורה בה הפונקציה ממומשת או נקראת.

שורה ראשונה: כותרת של תחילת ה- header.

שורה שניה: ציון האם הקובץ נוכחי מכיל את שגרת ההפעלה (main). על הקומפיילר להוסיף שורה `<main>` במידה שהקובץ הנוכחי מכיל את ה-main של התוכנית, או לחילופין `<empty>` , במידה שלא. שימו לב ה- linker יתריע על שגיאה במידה שיש יותר מ-main אחד או שאין בכלל main.

שורה שלישית: ציון כל הפונקציות הלא ממומשות בקובץ `<unimplemented>` - מכיל את שמות הפונקציות שהוצהרו ולא מומשו בקובץ יחד עם מיקום כל הקריאות לפונקציות בקובץ המקומפל (מספרי שורות). לפני כל מספר של מיקום יש פסיק. בסוף הרשימה אין פסיק.

שורה רביעית: ציון כל הפונקציות הממומשות בקובץ `<implemented>` - מכיל את שמות כל הפונקציות שנוצרו ומומשו בקובץ, פסיק, ומיקומן היחסי בקובץ המקומפל (מספר שורה).

שורה חמישית: סגירת ה-header.

כל המיקומים של פונקציות או קריאות להן יתייחסו למספרי השורות בקוד שפת האסמבלי הנוצר אשר מתחילים במספר 1, **החל מהשורה שלאחר ה-`</header>`**.

לדוגמה:

```
<header>
<main>
<unimplemented> foo,2,10,30, goo,3,11,21 boo
<implemented> woo,2 voo,9
</header>
```

זהו header של קובץ שבו הפונקציה foo (חיצונית) נקראת בשורה 2,10 ו-30. הפונקציה goo (חיצונית) נקראת בשורה 3,11 ו-21 והפונקציה boo מוצהרת אבל לא נקראת משום מקום. הפונקציות woo ו-voo מומשו בשורות 2 ו-9 בהתאמה. לפי השורה השניה ניתן לדעת שזהו קובץ שמכיל את ה-main.

* ראו דוגמאות לתכניות בשפת --C והפלט המתאים של הקומפיילר באתר הקורס, תחת קבצי עזר לחלק זה של הפרויקט. קחו בחשבון כי הדוגמאות מתאימות לרשומת ההפעלה והקצאות הרגיסטרים שנבחרו באותו מימוש וייתכן שבמימוש שלכם יהיה הבדל בפלט, גם אם הוא נכון ומתאים לדרישות הפרויקט. לכן, הבדיקה של המימוש שלכם בפועל צריכה להיות מבוססת על הרצת קובץ הריצה (.e) על גבי ה-VM של ה-RX והשוואת הפלט לפלט המצופה.

כלים נוספים:

- יש להשתמש בכלים שנלמדו בקורס (Flex, Bison) לצורך כתיבת הפרויקט – במיוחד לצורך מימוש המנתח הלקסיקלי והמנתח התחבירי.
- יש לכתוב את הפרויקט בשפת C או ++C בלבד.

ניתן להעזר במבני נתונים סטנדרטים (רשימות, טבלאות "האש" וכו') המסופקים על ידי ספריות חיצוניות סטנדרטיות הזמינות במכונה הוירטואלית של לינוקס המסופקת לטובת הפרויקט (למשל, STL ++C). במידה ותשתמשו ב-STL ++C ניתן להעזר בפרטים בקישור הבא לגבי מבני נתונים שימושיים: http://en.cppreference.com/w/cpp/container

ניתן גם להשתמש במימושים "פתוחים" של מבני נתונים כנ"ל, בתנאי שהשימוש בהם דורש הוספה של קבצים בודדים של המימוש לחומר ההגשה. אין להשתמש בספריות שאינן מותקנות במכונה הוירטואלית של לינוקס המסופקת. בכל מקרה של שימוש במימוש חיצוני של מבנה נתונים כלשהו, יש לציין בתיעוד המצורף להגשה את המקור לאותו מימוש שצירפתם.

הלינקר: rx-linker

באתר הקורס, תחת קבצי עזר לחלק זה של הפרויקט, ניתן למצוא את הלינקר בשם rx-linker.

גם אם יש רק קובץ rsk. אחד בתכנית, יש צורך להפעיל עליו את הלינקר לשם יצירת קובץ ריצה. במקרה זה הלינקר רק ימחק את ה-header ויצור קובץ חדש עם סיומת e. . כאשר מפעילים את הלינקר עם יותר מקובץ אחד, חייבים לתת את שם הקובץ הראשי, שמכיל את ה-main, כארגומנט ראשון. שם הקובץ שייווצר הוא שם הקובץ הראשון ללא סיומת ה-rsk ועם סיומת e.

המכונה הוירטואלית : rx-vm

באתר הקורס ניתן למצוא את המכונה הוירטואלית rx-vm, שהיא בעצם סוג של מפרש לשפת Riski. בעזרתו אתם יכולים "להריץ" את קובץ הריצה e. שיצר הלינקר. rx-vm מצפה לארגומנט בודד בשורת הפקודה - שם של קובץ ה-e. כאשר RX נתקל בפקודות READR או READI, הוא מדפיס סימן "?" על המסך ומחכה לקלט. ניתן גם להעביר אליו קלט מקובץ בעזרת שימוש ב-redirection pipe לקלט הסטנדרטי, כמקובל ביוניקס.

יודגש כי יתכן ו- rx-vm יקבל ויריץ קוד החורג מן ההנחיות הנ"ל. בכל מקרה עליכם לייצר קוד העומד בכל ההנחיות והדרישות המפורטות במסמך זה.

הוראות הגשה:

- מועד אחרון להגשה: יום ב' 23/01/2017 בשעה 23:55 .
- שימו-לב למדיניות בנוגע לאיחורים בהגשה המפורסמת באתר הקורס. במקרה של נסיבות המצדיקות איחור, יש לפנות **מראש** לצוות הקורס לתיאום דחיית מועד ההגשה.
- ההגשה בזוגות. הגשה בבודדים תתקבל רק באישור מראש מצוות הקורס.
- יש להגיש בצורה מקוונת באמצעות ה-Moodle מחשבונו של אחד הסטודנטים.

הקפידו לוודא כי העלתם את הגירסה של ההגשה אותה התכוונתם להגיש. לא יתקבלו טענות על אי התאמה בין הקובץ שנמצא ב-Moodle לבין הגירסה ש"התכוונתם" להגיש ולא יתקבלו הגשות מאוחרות במקרים כאלו.

- יש להגיש קובץ ארכיב מסוג Bzipped2-TAR בשם מהצורה (שרשור מספרי ת.ז – 9 ספרות):
  `proj-part3-<student1_id>_<student2_id>.tar.bz2`
- בארכיב יש לכלול את הקבצים הבאים:
  - את כל קבצי הקוד בהם השתמשתם (Bison, Flex, headers, וכל קובץ קוד מקור הנדרש לבניית המנתח) .

  *** יש להקפיד שהקוד שלכם יהיה קריא ומתועד פנימית ברמה סבירה כך שגם "זר" יוכל להבין את המימוש שלכם!**
  - יש לכלול גם קבצי מימוש מבני נתונים ממקור חיצוני שנעזרתם בהם, במידה ואינם חלק מהספריות הסטנדרטית של C++/C המותקנות במכונת לינוקס המסופקת. במקרה זה יש לתעד את המקור של אותם קבצים. אין לדרוש התקנה של שום קובץ או ספריה נוספים, מעבר למה שכלול בהגשה שלכם ולמה שמותקן במכונה הוירטואלית, על מנת לבנות את הקומפיילר שלכם.
  - `makefile` הבונה את קובץ הריצה של הקומפיילר

  * תזכורת: שם קובץ הריצה של הקומפיילר יהיה `rx-cc`
- מסמך תיעוד חיצוני (PDF) המכיל:
  - הסבר כללי על מימוש הקומפיילר שלכם
  - מבני הנתונים ששימשו לניהול מצב הקומפיילר, כגון: טבלאות הסמלים השונות.
  - תיאור ה-backpatching (פריסת הקוד) לכל אחד ממבני הבקרה
  - תיאור המודולים השונים בקוד
  - מבנה רשומת ההפעלה
  - אופן הקצאת רגיסטרים מיוחדים (שמורים)
  - הסבר על אופן המימוש של התמיכה במבנה נתונים מורכב, בפרט: אופן הגישה לשדה במבנה נתונים מורכב ושיטת העתקה (השמה) בין משתנים מסוג מורכב.
- קובץ הארכיב צריך להיות "שטוח" (כלומר, שלא ייצור ספריות משנה בעת הפתיחה אלא הקבצים ייווצרו בספריה הנוכחית).
- כמו בחלקים הקודמים, סביבת הבדיקה הרישמית הינה המכונה הוירטואלית של לינוקס המסופקת לכם. ניתן לפתח במחשב אחר מהמכונה הוירטואלית, אולם חובה לוודא לפני ההגשה שהתרגיל המוגש נבנה ורץ היטב במכונה הוירטואלית.

<u>תרגיל שלא יצליח להתקמפל יקבל 0.</u>

בהצלחה!