פרויקט - חלק 3

הקדמה

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 הווירטואלית rx-vm.

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

נקרא ללינקר (שיסופק לכם) באופן הבא:

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

הלינקר יקשר בין הקריאות לפונקציות לבין מימוש 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 הווירטואלית (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. המכיל 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.

התאמות הדקדוק

1. פתרון הרב-משמעויות בדקדוק ייעשה באמצעות מנגנון הגדרת קדימות/אסוציאטיביות האסימונים של Bison בלבד ואין לשנות את הדקדוק, כפי שנדרשתם בחלק 2.
2. עבור רב משמעות באופרטורים אריתמטיים/לוגיים/בוליאניים יש ליישם קדימויות ואסוציאטיביות כמקובל ב-C++/C . ניתן להיעזר בסיכום הקדימויות בקישור הבא:
http://en.cppreference.com/w/cpp/language/operator_precedence
3. בטיפול ברב-משמעות מסוג dangling else יש ליישם (ללא שינוי בדקדוק!) הצמדת ה-else ל-if הקרוב.
4. בטיפול ברב-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 בשיעור.
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. התוכנית צריכה להכיל המרות מפורשות (explicit cast) בשימוש בתחביר המתאים (כמו ב-C) על מנת שפעולות יתבצעו בין ביטויים מטיפוסים זהים. המרה מטיפוס שלם לממשי תשתמש בקוד מכונת RISKI המוגדר בחלק הבא.
2. טיפוס void משמש רק לטובת טיפוס החזרה מפונקציה. לא ניתן להצהיר על משתנה או פרמטר לפונקציה מסוג void. לא ניתן לבצע פעולות כלשהן בין ערכים (EXP) מטיפוס void.
void שנגזר ל-STMT חייב להיות מטיפוס void (במידת הצורך, יש להמירו באמצעות cast ל-void).
3. בקריאות לפונקציה הפרמטרים המועברים צריכים להיות מאותו טיפוס של הפרמטרים המוגדרים. אי התאמה בטיפוסים להצהרת ממשק הפונקציה גוררת שגיאה סמנטית. בהתאם, התוכנית חייבת להכיל המרות מפורשות של ביטוי לטיפוס המתאים לפרמטר אם הם שונים, כנ"ל.
4. ייתכן ובסוף פונקציה לא תהיה פקודת return בקוד המקור – הן מכיוון שמדובר בפונקציה המחזירה void והן בגלל שגיאת תוכניתן. אין צורך להפיק שגיאה במקרה כזה, אולם בכל מקרה יש להבטיח חזרה מהפונקציה לקורא, גם אם חסרה פקודת return בסוף הפונקציה. ערך ההחזרה במקרה כזה (אם אינו מוגדר void) לא מוגדר (יכול להיות כל ערך).

(המשך הכללים בעמוד הבא)

5. יש לנהל טבלאות סמלים עם תיחום סטטי (static scoping), כלומר, משתנה מוגדר בתחום של ה-BLK שבו הוגדר.
   1. תחילת פונקציה מאתחלת טבלת סמלים ריקה.
   2. הפרמטרים של פונקציה שייכים לתחום (scope) של הבלוק הראשי של הפונקציה.
   3. הגדרת משתנה ב-BLK פנימי ממסכת משתנה בעל שם זהה שהוגדר בבלוק שמכיל אותו.
   4. אסור, כמובן, להגדיר יותר ממשתנה אחד בעל אותו שם באותו הבלוק (זו שגיאה סמנטית).
   5. אין משתנים גלובליים.

6. לולאות:
בשפת --C, יש שני סוגים של לולאות.
   1. `while BEXP do STMT`
   2. `full_while BEXP do STMT`

אופן הביצוע של שתיהן זהה לחלוטין כמו שפת C, חוץ מהבדל אחד:
קטע קוד full_while מחזיר ערך בוליאני.
בתרגיל זה נממש את סכימת התרגום שהצעתם בתרגיל היבש 2 עבור חוק ה- `full_while`.
הערה: כאשר ערך ההחזרה של ה- full_while אינו בשימוש, צריך להתעלם ממנו.

תזכורת מתרגיל בית יבש 2:

הסמנטיקה של מבנה הבקרה הינה של לולאה המחזירה חיווי אם ביצועה החל והסתיים שלא ע"י פקודת break:

1. יש לבצע BEXP בעזרת חישוב מקוצר
   a. אם BEXP הוא true יש לעבור לשלב 2 להלן.
   b. אם BEXP הוא false יש לסיים, ולהחזיר true אם STMT התבצע לפחות פעם אחת, אחרת יש להחזיר false.
2. יש לבצע את STMT
   a. אם בתוך STMT התבצעה פקודת break, יש לסיים את לולאת ה- full_while ולהחזיר false.
   b. אחרת יש לחזור לשלב 1 לעיל.

הערה: אפשר להניח שהלולאות תמיד עוצרות.

7. פקודת break יכולה להופיע רק בבלוק של לולאה, אחרת יש לדווח על שגיאה סמנטית.
הסמנטיקה של הפקודה היא יציאה מהלולאה הכוללת הקרובה ביותר. (כמו בשפת C).

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
COPYF/COPYI	A B	A=B
SEQUF/SEQUI	A B C	If B=C then A=1 else A=0
SNEQF/SNEQI	A B C	If B!=C then A=1 else A=0
SLETF/SLETI	A B C	If B<C then A=1 else A=0
SGRTF/SGRTI	A B C	If B>C then A=1 else A=0
ADD2F/ADD2I	A B C	A=B+C
SUBTF/SUBTI	A B C	A=B-C
MULTF/MULTI	A B C	A=B*C
DIVDF/DIVDI	A B C	A=B/C
LOADF/LOADI	A B C	A = Mem[B+C]
STORF/STORI	A B C	Mem[B+C] = A

- שימו לב כי פעולות בעלות סיומות I מתבצעות על רגיסטרים שלמים (int) ופעולות בעלות סיומת F מתבצעות על רגיסטרים ממשיים (float).

CITOF	A B	A = (float)B
CFTOI	A B	A = (int) B

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
PRNTI	B	Prints the integer value of B
READI	A	Read an integer into A

PRNTF	B	Prints the float value of
READF	A	Read a float into A

HALT		Stop

הערות לטבלה הנ"ל :

1. A,B,C הם סימנים מופשטים , שיכולים לציין רגיסטר או קבוע כלשהו:
   - A מציין רגיסטר שלם בן 32 ביט.
   - B,C מציינים רגיסטרים או קבועים.
2. כל הפקודות פרט ל-LOAD ו-STOR פועלות על רגיסטרים בלבד.
3. L מציין תווית (מספר שורה*).

*ראו הסבר לגבי מספור השורות בפרק "מרחב זיכרון הפקודות וחישוב יעדי הקפיצה" בהמשך.

משאבי זיכרון ורגיסטרים

1. כתובת זיכרון נתונים (הערך של B+C עבור Mem[B+C] במִפרט הפקודות) הינה מספר שלם בין 0 ל-1023. הזיכרון הוא ממוען בתים, כלומר כתובת X היא הבית בהיסט X מתחילת הזיכרון. גודל הנתון שנקרא הוא 32 ביט בין אם נקרא מספר שלם או ממשי.
   * זיכרון הפקודות נפרד מזיכרון הנתונים ומרחבי הכתובות בלתי תלויים. ראו פרטים בפרק "מרחב זיכרון הפקודות וחישוב יעדי הקפיצה" בהמשך.
2. המעבד מכיל 1024 רגיסטרים מסוג שלם ו-1024 מסוג ממשי בני 32 ביט כל אחד. הרגיסטרים בעלי המספרים הסידוריים 0-1023. נסמנם באות I (איי) ומספר סידורי החל מאפס, למשל: I12 מציין את הרגיסטר השלוש-עשרה בקובץ הרגיסטרים של המעבד. בדומה עבור רגיסטרים ממשיים נסמנם באות F (אף).

שימו-לב: הרגיסטרים צריכים לשרת רק את המשתנים הזמניים שמקצה הקומפיילר. אין להשתמש בהם (באופן קבוע) לטובת משתנים שמוצהרים בתוכנית המקומפלת.

3. הרגיסטר השלם הראשון I0 שמור בארכיטקטורה לטובת שמירת כתובת החזרה בפקודות JLINK + RETRN.
4. יש לשמור (להקצות) רגיסטרים נוספים, לבחירתכם, על מנת לנהל את המחסנית ואת רשומת ההפעלה. מומלץ להקצות את הרגיסטרים השמורים ברגיסטרים הנמוכים ואת ההקצאות המשתנות למשתנים הזמניים לבצע מעל התחום השמור, בהתאם לתוכנית המקומפלת.
5. אין צורך לטפל בהקצאה יעילה של רגיסטרים. ניתן להניח שבמהלך פונקציה אחת לא יהיה צורך ביותר מ-900 משתנים זמניים עבור הקוד של אותה הפונקציה, אולם שימו-לב שפונקציה יכולה לקרוא לעצמה או לכל פונקציה אחרת (כלומר, הקוד הנוצר עבור כל פונקציה זקוק כמעט לכל מרחב הרגיסטרים).
6. 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. גישה לזיכרון מעבר למרחב המוגדר תגרום לאירוע "חריגה" ועצירה של מכונת ה-Riski.

מרחב זיכרון הפקודות וחישוב יעדי הקפיצה

זיכרון הפקודות במכונת Riski נפרד מזיכרון הנתונים וממוען באופן שונה ממנו. כתובת של פקודה היא מספר השורה (מספר סידורי) של פקודה בקוד המכונה המיוצר, החל מ-1 עבור הפקודה הראשונה בזיכרון. קובץ הריצה (.e) נטען לתחילת זיכרון הפקודות (כתובת 1) וריצת מכונת Riski מתחילה מפקודה (שורה) מספר 1 (כלומר, הפקודה הראשונה בקובץ הריצה היא הפקודה הראשונה שתרוץ ב-VM של ה-Riski).

בקוד ה - Riski שמייצר הקומפיילר עשויות להופיע פקודות קפיצה, כאשר יעד הקפיצה הוא מספר שורת פקודת היעד כנ"ל. לצורך חישוב יעדי הקפיצה בזמן קומפילציה, יש להשתמש בהטלאה לאחור (backpatching) ובמרקרים מסוג $N \to \varepsilon, M \to \varepsilon$. בתהליך זה הקומפיילר מניח שקובץ המקור שהוא מקמפל הוא היחיד בקובץ הריצה, כלומר שהפקודה הראשונה בקוד שהוא מייצר תהיה בכתובת (שורה) 1 בזיכרון הפקודות.

במידה וקובץ ה-rsk שמייצר הקומפיילר משולב עם עוד קבצי rsk לטובת קובץ הריצה (.e), הלינקר ידאג לתקן את כתובות היעד הרלוונטיות בהתאם למיקום היחסי של כל קובץ rsk בקובץ הריצה שנוצר. ראה פרטים על תהליך זה בפרק "תמיכה ב-Linker" בהמשך.

מימוש write בשפת --C

פקודת write בשפת --C משמשת להדפסת ערכים לפלט הסטנדרטי. יש לממש אותה באמצעות הפקודות PRNTI/PRNTF/PRNTC המתאימות.

פקודת PRNTI מניחה שהערך שניתן לה בארגומנט הוא בן 32 ביט. בהתאם, יש להמיר את ה-EXP שבארגומנט של פקודת write לגודל של 32 ביט על מנת שיודפס ערך נכון במקרה שה-EXP הוא של ערך מטיפוס שלם קטן יותר.

במקרה של הדפסת מחרוזת, יש להשתמש במספר קריאות מתאים לפקודה PRNTC. כמו-כן, יש לטפל בהמרה של הצירופים המיוחדים:

- $\backslash n$ (שורה חדשה בלינוקס).
- $\backslash$" (גרשיים באמצע מחרוזת)
- $\backslash t$ (טאב).

ההמרה היא 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.

המרת טיפוסים

כפי שצוין לעיל, הרגיסטרים בארכיטקטורת Riski הם שלמים או ממשיים בני 32 ביט.

יש צורך בהמרה כאשר מציבים מרגיסטר שלם לתוך רגיסטר ממשי (ולהיפך). לדוגמא, הפקודה הבאה תחזיר שגיאת זמן ריצה ע"י הסימולטור RX:

COPYF F1 I1

במקום זאת יש להשתמש בהמרה מפורשת בעזרת פקודת RISKI הבאה:

CITOF R2 I1

COPYF R1 R2

יש לשים לב שבזיכרון משתנים ממשיים ושלמים שניהם נשמרים ב32 ביט. לכן קריאה וכתיבה מהזיכרון של משתנה שלם או ממשי בעזרת פקודת LOAD או STORE אשר אינו מתאימה לו תבצע פעולה על ערך שאינו מוגדר.

הצהרות, הגדרות וקריאות לפונקציות

השפה מאפשרת קריאה לפונקציות המוצהרות ו/או מוגדרות (ממומשות) לפני מקום הקריאה בקוד (כולל קריאה של פונקציה לעצמה, ללא הצהרה מקדימה). הגדרת הפונקציה (המימוש שלה) יכולה להיות באותו קובץ מקור (לפני השימוש, או אחרי השימוש אם לפני השימוש הייתה הצהרה מתאימה) או בקובץ מקור אחר. הפונקציות מקבלות מספר כלשהו של פרמטרים (בהתאם להגדרה) ע"פ ערכם (by value) ומחזירות ערך החזרה יחיד מהטיפוס שהוצהר עבור הפונקציה.

להזכירכם, הדקדוק מכיל הצהרות על פונקציות מהצורה:

```
FDEFS -> FDEFS FUNC_API BLK | FDEFS FUNC_API ; | ε
FUNC_API -> TYPE id ( FUNC_ARGS )
```

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). על מנת לאפשר קריאה לפונקציה שרק הוצהרה אבל טרם מומשה, יש לשמור את המקומות בהם יש קריאה לפונקציה מוצהרת אך לא מוגדרת. במידה והפונקציה תוגדר בהמשך הקובץ, באחריות הקומפיילר להטליא את המקומות בהם הייתה קריאה לאותה פונקציה בקוד המיוצר. אם פונקציה לא תוגדר עד לסיום אותו קובץ מקור, יש לכלול את רשימת מקומות הקריאה לפונקציה בתחילת קובץ ה-rsk בהתאם להגדרות בפרק "תמיכה ב-Linker" בהמשך (הרשימה תשמש את הלינקר להטלאת קריאות בין מודולים).

אין תמיכה ב-Function overloading, כלומר, לכל פונקציה קיימת רק הגדרה יחידה וממשק (הגדרת פרמטרים וערך החזרה) יחידים. הגדרת פונקציה ששונה מההצהרה עליה (אם קיימת) או קיום הצהרות לאותה פונקציה עם סוגי פרמטרים/ערכי החזרה שונים או ריבוי הגדרות של פונקציה הן שגיאות סמנטיות.

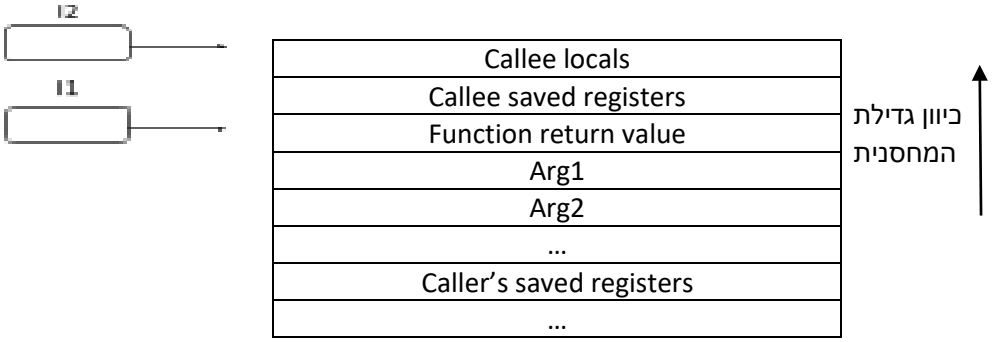
באחריות הקומפיילר שלכם לאכוף את הדרישה בתוך קובץ מקור. הלינקר ידאג לאכיפת מגבלה זו בין מודולים (שישנה הגדרה אחת לפונקציה בכל המודולים). הלינקר לא יאמת שה-API שהוצהר עבור פונקציה במודול אחד תואם ל-API שהוצהר עבור פונקציה באותו שם במודול אחר.

ניתן לקרוא לפונקציה מספר פעמים במהלך התוכנית, כולל קריאות רקורסיביות, ישירות או עקיפות. בהתאם, השתמשו ברשומת הפעלה בדומה לנלמד בשיעור 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 מהגדרה ארכיטקטונית של פקודת JLINK). רגיסטר I1 יכיל את כתובת בסיס רשומת ההפעלה (stack frame) ו- I2 יכיל את המצביע לראש המחסנית. לדוגמה:



התמודדות עם שגיאות קומפילציה והודעות שגיאה

אין הנחת קלט תקין לקומפיילר, 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 כנדרש. בתרגיל זה נשתמש ב-linker המסופק לכם (ראו הסבר בסוף התרגיל) אשר יצור קובץ ריצה בודד עבור מכונת ה-RX ונאפשר שימוש בפונקציות המוגדרות בקבצים שונים.

לצורך תמיכה ב-Linker על הקומפיילר להוסיף header בעל ארבע 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).

פונקציית main וספריית זמן ריצה

מכונת RX מתחילה את הריצה מכתובת 1 בזיכרון הפקודות. הלינקר שם בתחילת קובץ הריצה (החל מכתובת 1) את ספריית זמן הריצה rx-runtime.rsk. ספריית זמן הריצה מאתחלת מספר רגיסטרים לאפס וקוראת לפונקציית main. פונקציית main צריכה להיות מוגדרת באחד מקבצי rsk המסופקים ללינקר בשורת הפקודה וחייבת להיות מהטיפוס הבא:

```
void main();
```

העדר הגדרה של פונקציית בשם main באחד מהמודולים או הגדרה של יותר מפונקציית main אחת יגרמו לשגיאת לינקר. הגדרה של פונקציית main בעלת טיפוס ממשק שונה מהנ"ל לא תגרום לשגיאת לינקר, אולם ההתנהגות בזמן ריצה תהיה בלתי מוגדרת. החזרה מפונקציית main תביא לסיום הריצה (חייב להיות RETRN בסיום main, כמו בכל פונקציה אחרת).

ספריית זמן הריצה rx-runtime.rsk מסופקת לכם עם חומרי הפרויקט. מכיוון שאין העברת פרמטרים וערך חזרה מ-main, האופן שבו נקראת main מאתחול זמן הריצה לא מחייב דבר לגבי מבנה רשומת ההפעלה שלכם ואופן העברת פרמטרים לפונקציות, ולכן ניתן להשתמש בקובץ המסופק כפי שהוא ללא שינויים ללא קשר למבנה רשומת ההפעלה שתבחרו.

מבנה ה-header

```
1) <header>
2) <unimplemented> func1,L1,L2,L3,... func2,L1,L2,L3... …
3) <implemented> func1,L func2,...L …
4) </header>
```

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

שורה ראשונה: כותרת של תחילת ה- header.

שורה שניה: ציון כל הפונקציות הלא ממומשות בקובץ <unimplemented> - הרשומות מופרדות ע"י רווח.

כל רשומה בשורה מורכבת משני חלקים

1) שם הפונקציה
2) מיקומי כל הקריאות לפונקציה בקובץ המקומפל (מספרי שורות) מופרדים ע"י פסיק.

והרכיבים הנ"ל מופרדים ע"י פסיק.

שורה שלישית: ציון כל הפונקציות הממומשות בקובץ <implemented> - הרשומות מופרדות ע"י רווח.

1) שם הפונקציה
2) המיקום היחסי להגדרת הפונקציה בקובץ המקומפל (מספרי שורה) .

והרכיבים הנ"ל מופרדים ע"י פסיק.

שורה רביעית: 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:

```
1) <header>
2) <unimplemented> foo,2,10,30 goo,3,11,21 boo
3) <imlemented> woo,2 voo,9 main,24
4) </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. כמו-כן, הקובץ מכיל את פונקציית ה-main בשורה 24.

גם כן, הרשומות של הפונקציות ב- header מופרדים על ידי רווח.

דוגמאות

- ראו דוגמאות לתוכניות 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.
- חשוב לעקוב אחרי לוח המודעות באתר הקורס. **בתאריך 06.01.22 יפורסמו עוד כמה דוגמאות לבדיקה עצמית.**

כלים נוספים

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

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

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 הווירטואלית 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.

ללינקר יש לתת בשורת הפקודה את רשימת קבצי rsk (תוצרי הקומפיילר) שמעוניינים לצרף לטובת קובץ הריצה. אין צורך לתת את קובץ ספריית זמן הריצה rx-runtime.rsk – הלינקר יצרף אותו באופן אוטומטי, כפי שהוסבר בסעיף "פונקציית main וספריית זמן ריצה" – אולם, יש להקפיד שקובץ זה יהיה בתיקייה הנוכחית בעת הפעלת הלינקר.

גם אם יש רק קובץ rsk אחד בתוכנית, יש צורך להפעיל עליו את הלינקר לשם יצירת קובץ ריצה הכולל את ספריית זמן הריצה. כאשר מפעילים את הלינקר עם יותר מקובץ אחד, שם קובץ הריצה שייווצר הוא שם הקובץ הראשון בשורת הפקודה עם סיומת e במקום rsk.

המכונה הווירטואלית rx-vm

באתר הקורס ניתן למצוא את המכונה הווירטואלית rx-vm, שהיא בעצם סוג של מפרש לשפת Riski. בעזרתה אתם יכולים "להריץ" את קובץ הריצה e. שיצר הלינקר. rx-vm מצפה לארגומנט בודד בשורת הפקודה - שם של קובץ ה-;.e כאשר RX נתקל בפקודה 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 ה' 2022/01/27 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 אתר ה-Moodle של הקורס, מחשבונו של אחד הסטודנטים. הקפידו לוודא כי העלתם את הגרסה של ההגשה אותה התכוונתם להגיש. לא יתקבלו טענות על אי התאמה בין הקובץ שנמצא ב-Moodle לבין הגרסה 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 במסמך התיעוד החיצוני. אין לדרוש התקנה של שום קובץ או ספריה נוספים, מעבר למה שכלול בהגשה שלכם ולמה שמותקן במכונה הווירטואלית, על מנת לבנות ולהריץ 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
  - מבנה רשומות ההפעלה
  - אופן הקצאת רגיסטרים מיוחדים (שמורים)
  - תיאור המודולים השונים בקוד של הקומפיילר שלכם
  - תיעוד המקור של מימוש מבני נתונים חיצוני (שאינו חלק מהספרייה הסטנדרטית) שהשתמשתם בו במימוש שלכם.
- קובץ הארכיב צריך להיות "שטוח" (כלומר, שלא ייצור ספריות משנה בעת הפתיחה אלא הקבצים ייווצרו בספריה הנוכחית).
- כמו בחלקים הקודמים, סביבת הבדיקה המחייבת הינה המכונה הווירטואלית של לינוקס המסופקת לכם. הקפידו לבדוק את ההגשה שלכם במכונה זו ואל תשאירו זאת לרגע האחרון. <u>תרגיל שלא יצליח להתקמפל יקבל 0.</u>

בהצלחה!