תורת הקומפילציה

תרגיל 5

מתרגל אחראי על התרגיל: שי גנדלמן

שאלות בנוגע לתרגיל יש לשאול בפיאצה של הקורס. לפני ששואלים שאלה יש לבדוק שהיא לא הופיעה בעבר. שאלה שהופיעה בעבר לא תקבל תשובה. הערות והבהרות המופיעות בפיאצה מחייבות.

במידה ויהיו שינויים \ עדכונים בתרגיל הבית, הודעה תפורסם עם עדכון מסמך זה, והשינויים יסומנו בצהוב.

התרגיל יבדק בבדיקה אוטומטית. הקפידו למלא אחר ההוראות במדויק.

1. כללי

בתרגיל זה תממשו תרגום לשפת ביניים LLVM IR, עבור השפה FanC מתרגילי הבית הקודמים. בין היתר תממשו השמה של משתנים מקומיים במחסנית, ומימוש מבני בקרה באמצעות backpatching.

2. LLVM IR

בתרגיל תשתמשו בשפת הביניים של LLVM שריאתם בשיעורים. התיעוד לשפה זמין ב-
https://llvm.org/docs/LangRef.html

ניתן למצוא דוגמאות קוד ב-LLVM בקובץ המצורף llvm_examples.zip.

2.1 פקודות

בשפת LLVM יש מספר רב של פקודות. מומלץ להשתמש בפקודות המדמות את שפת הרביעיות שנלמדה בשיעורים. להלן הפקודות:

1. טעינה לרגיסטר: load
2. שמירת תוכן רגיסטר: store
3. פעולות חשבוניות: add, sub, mul, udiv, sdiv
4. פעולת השוואה: icmp
5. קפיצה מותנית ולא מותנית: br
6. קריאה לפונקציה: call
7. חזרה מפונקציה: ret
8. הקצאת זיכרון: alloca
9. חישוב כתובת: getelementptr
10. צומת phi: phi. הפקודה phi מקבלת רשימת זוגות של ערכים ותוויות, ומשימה לרגיסטר את הערך המתאים לתווית של הבלוק שקדם לבלוק הנוכחי של צומת ה-phi בזמן ריצת התוכנית. במידה ומשתמשים ב-phi היא חייבת להיות הפקודה הראשונה בבלוק.

ניתן להשתמש בכל פקודה אחרת של LLVM בתנאי שניתן להריץ אותה באמצעות lli על שרת הקורס.

2.2 רגיסטרים

ב-LLVM ניתן להשתמש באינסוף רגיסטרים. השפה היא Single Static Assignment (SSA) כך שניתן לבצע השמה יחידה לרגיסטר.

שימו לב שלא קיימת פקודה להשמה של קבוע ברגיסטר, ניתן לעשות זאת למשל על ידי חיבור הערך המבוקש עם 0. **אין להשתמש ברגיסטרים לאחסון ערכי ביטויים בוליאנים בזמן שערוך הביטוי**. (דרישה זו תובהר בפרק הסמנטיקה).

2.3 תוויות

ב-LLVM יעדים של קפיצות מיוצגים בתור תוויות. מחרוזות אלפא-נומריות (+קו תחתון, נקודה, דולר) שאחריהן מופיעות נקודתיים. לדוגמה:

```
label_42:
%t6 = load i32, i32* %ptr
```

קפיצה אל label_42 תקפוץ אל הבלוק הבסיסי המתחיל בשורה שאחריה, **וכל לייבל מתחיל בלוק בסיסי חדש וכל בלוק בסיסי צריך להסתיים בפקודת br או ret הקובע את מבנה ה-CFG (גרף הבקרה) של התוכנית.**

את התוויות בפקודות קפיצה של מבני בקרה יש לייצר ולהשלים בשיטת ה-backpatching כפי שנלמד בשיעור. נתונה לכם המחלקה CodeBuffer בקובץ bp.hpp, עם מימוש של buffer ופונקצית backpatching. אין חובה להשתמש במחלקה, אך מומלץ.

<u>עבודה עם CodeBuffer:</u>

המחלקה ממשת מתודות הדומות לאלו שנלמדו בשיעור – backpatch, merge, makelist, emit עם שינויים קלים – אנא קראו את התיעוד.

הפונקציות מטפלות ברשימת כתובות בבאפר קוד, ניתן להשתמש בכתובות אלו לצורך דיבוג הבאפר. שימו לב, ערך החזרה של emit הוא הכתובת אליה כתבתם. זו הכתובת שבה עליכם להשתמש לצורך backpatching.

שימו לב, בשונה משפת הביניים התיאורתית שלמדתם בשיעור, שבו יכלתם לכתוב במפורש את הערך המספרי של הכתובת אליה תרצו לקפוץ, ב-LLVM ניתן לקפוץ אך ורק לתוויות שהוגדרו (לא לכתובת מספרית). זה נובע מכך שהמיקום בזיכרון אליו יכתבו הפקודות ידוע רק כאשר הוא יתורגם לשפת מכונה. לכן יעדי הקפיצות שתעבירו למתודה backpatch יהיו המחרוזת של התווית, ורשימת המיקומים ב-buffer שצריך להשלים.

2.4 משתנים גלובלים

ניתן לשמור ליטרל מחרוזת כמשתנה גלובלי.
את ליטרל המחרוזת יש להגדיר עם null בסופה (להוסיף לה את התו \00,כמו בדוגמאות שבהרצאה ובתרגול). ניתן להניח כי המחרוזות בתוכניות הבדיקה לא יכילו תוים מיוחדים כמו \n,\r,\t.
המחלקה CodeBuffer מכילה מתודה להדפסת באפר הקוד, ומכילה בנוסף לכך גם שתי מתודות לטיפול במשתנים הגלובלים של התוכנית: המתודה emitGlobal כותבת שורות לבאפר נפרד, והמתודה printGlobalBuffer מדפיסה את תוכן הבאפר הנפרד.

3. מחסנית

בתרגיל אתם לא נדרשים לנהל את המחסנית עם רשומות ההפעלה של הפונקציות הנקראות.

את המשתנים הלוקלים של הפונקציות יש לאחסן על מחסנית, לפי ה-offsets שחושבו בהתאם לתרגיל 3. מומלץ להקצות בתחילת הפונקציה מקום לכל משתנים הלוקלים על המחסנית באמצעות הפקודה alloca ובה נתייחס לכל משתנה ללא תלות בטיפוסו כ-i32.
בכדי לאחסן טיפוס בוליאני או byte כ-i32 ניתן להשתמש בפקודה zext המשלימה את הביטים העליונים עם אפסים, ו-trunc שעושה את הפעולה ההפוכה. כדי ללמוד יותר על פקודות אלו מומלץ לבקר בתיעוד הרשמי של LLVM.

- **ניתן להניח כי מספר המשתנים הלוקלים בכל פונקציה קטן מ-50.**

4. סמנטיקה

יש לממש את ביצוע כל ה - statements בפונקציה ברצף בסדר בו הוגדרו. הסמנטיקה של ביטויים אריתמטיים ושל קריאות לפונקציות מוגדרת כמו הסמנטיקה שלהם בשפת C. ההרצה תתחיל בפונקציה main, ותסתיים כשהקריאה החיצונית ביותר לפונקציה main חוזרת. עבור מבני הבקרה יש להשתמש ב - backpatching. ניתן להיעזר בדוגמאות מהתרגולים.

4.1 משתנים

4.1.1 אתחול משתנים

יש לאתחל את כל המשתנים בתכנית כך שיכילו ערך ברירת מחדל במידה ולא הוצב לתוכם ערך.
הטיפוסים המספריים יאותחלו ל-0.
הטיפוס הבוליאני יאותחל ל-false.

4.1.2 גישה למשתנים

כאשר מתבצעת פניה בתוך ביטוי למשתנה מטיפוס פשוט, יש לייצר קוד הטוען מן המחסנית את הערך האחרון שנשמר עבור המשתנה. כאשר מתבצעת השמה לתוך משתנה, יש לייצר קוד הכותב למחסנית את ערך הביטוי במשפט ההשמה.

4.2 ביטויים חשבוניים

יש לממש פעולות חשבוניות לפי הסמנטיקה של שפת C.

הטיפוס המספרי int הינו signed, כלומר מחזיק מספרים חיוביים ושליליים.
הטיפוס המספרי byte הינו unsigned, כלומר מחזיק מספרים אי-שליליים בלבד.
חילוק יהיה חילוק שלמים.

השוואות רלציוניות בין שני טיפוסים מספריים שונים יתייחסו לערכים המספריים עצמם (כלומר, כאילו הערך הנמצא ב - byte מוחזק על ידי int (לכן, למשל, הביטוי 8b==8 יחזיר אמת).

יש לממש שגיאת חלוקה באפס. במידה ועומדת להתבצע חלוקה באפס, תדפיס התכנית "Error division by zero" באמצעות הפונקציה print ותסיים את ריצתה.

4.2.1 גלישה נומרית

יש לדאוג שתתבצע גלישה מסודרת של ערכים נומריים במידה ופעולה חשבונית חורגת (מלמעלה או מלמטה) מהערכים המותרים לטיפוס.

טווח הערכים המותר ל- int הוא 0-0xffffffff, (כך ש- 0-0x7fffffff חיוביים ו- 0x80000000-0xffffffff שליליים).
גלישה נומרית עבור int אמורה לעבוד באופן אוטומטי במידה ומימשתם את התרגיל לפי ההנחיות (כלומר, תתקבל תמיד תוצאה בטווח הערכים המותר, ללא שגיאה)

טווח הערכים המותר עבור byte הוא 0-255. יש לוודא כי גם תוצאות פעולה חשבונית מסוג byte תניב תמיד ערך בטווח הערכים המותר על ידי truncation של התוצאה, כלומר איפוס הביטים הגבוהים.

4.3 ביטויים בוליאנים

יש לממש עבור ביטויים בוליאניים short-circuit evaluation, באופן הזהה לשפת C: במידה וניתן לקבוע בשלב מסוים בביטוי בוליאני את תוצאתו, אין להמשיך לחשב חלקים נוספים שלו. כך למשל בהינתן הפונקציה printfoo:

```
bool printfoo() {
      printi(1);
      return true;
}
```

והביטוי הבוליאני:

```
true or printfoo()
```

לא יודפס דבר בעת שערוך הביטוי.

בנוסף, **אין להשתמש ברגיסטרים לתוצאות או תוצאות ביניים של ביטויים בוליאניים** . יש לתרגם אותם לסדרת קפיצות כפי שנלמד בתרגול. לדוגמה, במידה והביטוי הבוליאני הוא ה-Exp במשפט השמה למשתנה, יש להשתמש רק ברגיסטר אחד לתוצאה הסופית כמשתנה ביניים לצורך ביצוע store (שמירה לזיכרון).

רמז – שימוש בפקודה phi יוכל להקל על המימוש במקרים מסוימים, אך איננו מחייב.

- **ניתן לשמור ערך של ביטוי בוליאני ברגיסטר במקרים הבאים:**
  - **כתיבה וקריאה למשתנים**
  - **חישוב ערך של relop על ידי הפקודה icmp**
  - **העברה של ערך בוליאני לפונקציה**
  - **החזרה של ערך בוליאני מפונקציה**
- **עם זאת, אין לבצע חישובים בוליאנים כדי לחשב תוצאה של פעולות לוגיות - not, and, or, רק עבור התוצאה הסופית.**

4.4 קריאה לפונקציה

בעת קריאה ל - Call, ישוערכו קודם כל הארגומנטים של הפונקציה לפי הסדר (משמאל לימין) ויועברו לפונקציה הנקראת. קוד הפונקציה יקרא באמצעות הפקודה call. בסוף ביצוע הפונקציה תקרא הפקודה ret. סוף הפונקציה הוא סוף רצף הפקודות בבלוק של הפונקציה, גם אם אינו כולל אף פקודת return.

במידה והפונקציה מחזירה ערך מטיפוס כלשהו (int, byte, bool) והפקודה האחרונה שמתבצעת בה אינה פקודת return, הערך שיוחזר יהיה ערך ברירת המחדל עבור אתחול משתנים מטיפוס זה.

- **ניתן להניח שבגוף הפונקציה לא תתבצע השמה לתוך פרמטר של הפונקציה.**

4.5 if

בראשית ביצוע משפט if משוערך התנאי הבוליאני Exp. במידה וערכו true, יבוצע ה – Statement בענף הראשון, ואחריו ה – Statement שנמצא בקוד אחרי ה - if. במידה וערכו false ומדובר ב-if-else יבוצע ה – Statement בענף השני, ואחריו ה – Statement שנמצא בקוד אחרי ה – if.

התנאי הבוליאני של המשפט עשוי לכלול ביטויים מורכבים, כפי שהוגדר בתרגיל 3.

4.6 while

בראשית ביצוע משפט while משוערך התנאי הבוליאני ב-Exp. במידה וערכו true, יבוצע ה-Statement, והריצה תחזור לשיערוך ה-Exp. במידה וערכו false, יבוצע ה-Statement שנמצא בקוד אחרי ה-while.

התנאי הבוליאני עשוי לכלול ביטויים מורכבים, כפי שהוגדר בתרגיל 3.

4.7 break

משפט break יכול להיות משוייך ל-case ב-switch או ללולאת while. משפט ה-break ישוייך למבנה בקרה הפנימי ביותר אליו הוא שייך, לדוגמה בקטע קוד הבא, ה-break משוייך למבנה המסומן באותו הצבע:

```
while (a < 10) {
      switch (x) {
            case 1:
                  while (z < 10) {
                        z = z + 1;
                        break;
                  }
            case 2: a = a + 1; break;
```

```
        }
        break;
}
```

כאשר break משוייך ל-while, אז ביצוע משפט break יגרום לביצוע הפקודה הבאה אחרי גוף ה-while. כאשר break משוייך ל-case של switch, ביצוע משפט ה-break יגרום לביצוע הפקודה שבאה אחרי גוף ה-switch אליו ה-case משוייך. פירוט נוסף נמצא בסעיף על switch בהמשך.

4.8 continue

ביצוע משפט continue בגוף הלולאה יגרום לקפיצה לתנאי הלולאה הפנימית ביותר בה ה-continue נמצא. תנאי הלולאה ייבדק ובמידה והתנאי מתקיים, המשפט הבא שיתבצע הוא המשפט הראשון בתוך אותה לולאה. אחרת יתבצע המשפט הבא אחרי לולאה זו.

4.9 return

במידה וזהו משפט return Exp, יש לקרוא ל-ret כך שיחזיר את Exp.

4.10 switch

בביצוע משפט switch, תחילה ישוערך ערך ה-Exp שמופיע בכלל. לאחר מכן נעבור לבצע את הפקודות המופיעות לאחר ה-case שערכו שווה לערך Exp, ונמשיך לבצע את הפקודות עד למופע של break, או עד סוף גוף ה-switch. יש לשים לב שמתבצע fall-through בדומה לשפת C, כלומר כל עוד לא מופיע break, נמשיך לבצע את הפקודות גם ב-case המופיעים אחרי ה-case המתאים.

במידה והערך שחושב על ידי Exp שונה מכל הערכים ב-case'ים השונים, אז במידה ויש default נעבור לבצע את הפקודות שמופיעות אחריו, ובמידה ואין default, נעבור לבצע את הפקודות המופיעות אחרי גוף ה-switch.

דוגמה ל-fall-through:

(שימו לב שהדוגמה שונתה)

```
switch (x) {
      case 1: printi(1); break;
      case 2: printi(2);
      case 3: printi(3); break;
      case 4: printi(4);
      default: printi(0);
}
```

עבור x=1 יודפס 1, עבור x=2 יודפס 2,3 עבור x=3 יודפס 3 ועבור x=4 יודפס 4,0, ועבור כל x אחר יודפס 0.

- **ניתן להניח שכל case מכיל ערך יחודי, כלומר אין case 2'ים שונים באותו switch עם אותו ערך.**
- **ניתן להניח שלא יופיע default יותר מפעם אחת ב-switch יחיד.**
- **לפי הדקדוק שהוצג בת"ב 3, default יכול להופיע רק בסוף משפט switch.**

5. שימוש בפונקציות ספרייה

ניתן להשתמש בפונקציות printf, exit מהספרייה הסטנדרטית, על ידי הכרזה שלהם:

```
declare i32 @printf(i8*, …)

declare void @exit(i32)
```

יש להוסיף הכרזות אלו לקוד המיוצר על מנת שיעבוד כראוי.

6. פונקציות פלט

קיימות 2 פונקציות פלט בשפת FanC. הראשונה printi, המקבלת מספר, והשנייה print, המקבלת מחרוזות. עליכם לכלול את המימוש שלהן בקוד שתייצרו. שימו לב שיש לכלול את ההגדרות של @.str_specifier, @.int_specifier.

מימוש מומלץ לפונקציות הללו ניתן למצוא בקובץ print_functions.llvm.

- **ניתן להניח שלא יופיעו escape sequence במחרוזות המודפסות כדוגמת \n,\r,\t.**

7. טיפול בשגיאות

תרגיל זה מתמקד בייצור קוד ביניים ולא מוסיף שגיאות קומפילציה מעבר לאלה שהופיעו בתרגיל 3. יש לדאוג שהקוד המיוצר יטפל בשגיאות חלוקה באפס.

8. קלט ופלט המנתח

קובץ ההרצה של המנתח יקבל את הקלט מ-stdin.

את תוכנית LLVM השלמה יש להדפיס ל-stdout (באמצעות הפונקציות המתאימות במחלקה CodeBuffer). הפלט ייבדק על ידי הפניה לקובץ של stdout ו-stderr והרצה על ידי התוכנית lli.

9. הדרכה

כדאי לממש את התרגיל בסדר הבא:

1. קוד להקצאת רגיסטרים
2. חישובים אריתמטים – להתחיל מהפשוט למורכב
3. חישובים לביטויים בוליאני מורכבים
4. שמירת וקריאת משתנים למחסנית
5. רצף statements
6. מבני בקרה
7. קריאה לפונקציות הפלט
8. קריאה לפונקציות

מומלץ להתחיל בכתיבת תוכניות פשוטות ב-LLVM ולהרגיש בנוח עם השפה.

מומלץ ליצור template אליהם תוכלו להעתיק את הקוד המיוצר על ידי התוכנית, בלי לייצר תוכנית שלמה, על מנת שתוכלו לבדוק את ייצור הקוד שלכם בשלבים מוקדמים.

ניתן ומומלץ להשתמש במבני הנתונים של הספרייה הסטנדרטית.

10. הוראות הגשה

שימו לב כי קובץ ה-makefile מאפשר שימוש ב-stl. אין לשנות את ה-makefile.

יש להגיש קובץ אחד בשם ID1-ID2.zip עם מספרי ת"ז של המגישים. על הקובץ להכיל:

- קובץ flex בשם scanner.lex.
- קובץ bison בשם parser.ypp.
- כל הקבצים הדרושים לבניית המנתח, כולל output.* שסופקו כחלק מתרגיל 3, ב-bp.* שסופקו כחלק מתרגיל זה, במידה והשתמשתם בהם.

בנוסף יש להקפיד שהקובץ לא יכיל:

- את קובץ ההרצה
- קבצי פלט של flex ו-bison
- את ה-makefile שסופק כחלק מהתרגיל.

יש לוודא כי בביצוע unzip לא נוצרת תיקיה נפרדת. **על המנתח להיבנות על השרת csComp ללא שגיאות באמצעות קובץ makefile שסופק עם התרגיל.** הפקודות הבאות יגרמו ליצירת קובץ הרצה hw5:

```
unzip id1-id2.zip
cp path-to/makefile .
make
```

פלט המנתח צריך להיות ניתן להרצה על ידי הסימולטור, כך למשל יש לוודא כי תוכניות הדוגמה באתר מייצרות פלט זהה לפלט הנדרש. ניתן לבדוק את עצמכם כך:

```
./hw5 < path-to/t1.in > t1.llvm
lli path-to/t1.llvm > t1.res
diff path-to/t1.res path-to/t1.out
```

יריץ את המנתח, ייצר קובץ LLVM, יריץ את lli עליו ללא שגיאות, ו-diff יחזיר 0.

בדקו היטב שההגשה שלכם עומדת בדרישות בסיסיות הללו לפני ההגשה עצמה. מומלץ לכתוב גם טסטים נוספים שיבדקו את נכונות המימוש עבור מבני הבקרה שונים.

שימו לב כי באתר יש סקריפט לבדיקה עצמית לפני ההגשה בשם selfcheck. תוכלו להשתמש בו על מנת לוודא כי ההגשה שלכם תקינה.

הגשות שלא יעמדו בדרישות לעיל (ובפרט שלא עוברות את ה-selfcheck) יקבלו ציון 0.

בתרגיל זה ייבדקו העתקות. אנא כתבו את הקוד שלכם בעצמכם.

- **הערה** – את קבצי הטקסט הנמצאים באתר מומלץ להוריד, ולא לפתוח בדפדפן ולעשות copy-paste, דבר זה יכול לשבש ירידות שורה ולגרום לכך שהם לא יעבדו כראוי.

בהצלחה!