תרגיל בית שני: ייצור קוד ביניים. סמסטר א 2019

ניתן להגיש בזוגות

את הפתרון יש להגיש ב- moodle.
במקרה שהתרגיל הוכן ע"י שני סטודנטים – רק אחד או אחת מהשניים יגישו את התרגיל (ויציינו את שמות שני המגישים בקובץ README).
יש להגיש קובץ zip הכולל את הקבצים של התוכנית עם התוספות שהכנסתם.
אין להגיש את הקבצים שיצרו `flex & bison`.

יש לצרף גם קובץ README שבו רשום איזה חלקים מהתרגיל עשיתם והוא עובד. הכוונה לפרוט קצר של שורה או שניים. למשל אפשר לכתוב: "הוספנו תמיכה במשפטים מסוג switch ו- break". (כמובן שרצוי לרשום פשוט "הכל").

כתובת המייל שלי: gadips@gmail.com

מצורפים קובצי המקור של "מיני קומפיילר" – תוכנית שקוראת קלט בשפת תכנות פשוטה ומתרגמת אותו לקוד ביניים מסוג `three address code`. התוכנית נכתבה בעזרת `flex` ו- `bison`. היא כתובה בשפת C ובשפת C++. (מאחר ששפת C היא subset של C++ ניתן להתיחס לתוכנית כאל תוכנית בשפת C++). נעשה כאן שימוש בסיסי בלבד בשפת C++ ומי שמכיר את Java ימצא כאן דברים מוכרים.

התרגיל הוא להכניס שינויים במיני קומפיילר.

הקומפיילר כרגע כולל בערך אלף שורות קוד (חלקן שורות ריקות ושורות עם הערות ובלי לקחת בחשבון קוד הנוצר אוטומטית ע"י `flex` ו- `bison`).
תצטרכו לכתוב כמאה – מאתיים שורות קוד.

ראשי פרקים של מסמך זה

תאור התרגיל: מה צריך לעשות ? (כולל הנחיות). כנראה שההנחיות יהיו ברורות יותר אחרי קריאת ההסברים על התוכנית שמופיעים בהמשך.

הקלט והפלט של הקומפיילר בקיצור: הקלט הוא שפת תכנות פשוטה. הפלט הוא קוד ביניים מסוג `three address code` דומה למה שראינו בכיתה.
(במקרה זה "קוד הביניים" הוא קוד המטרה כלומר הפלט הסופי של הקומפיילר).

בניית הקומפיילר (בקיצור: מריצים את flex & bison ומקמפלים עם קומפיילר של C++)
תאור המימוש של הקומפיילר -- יש כאן הסבר על ה- `AST (Abstract Syntax Tree)`, על ה- classes וה- methods העיקריים ועוד.

קבצים. פרוט של כל קובצי המקור של התוכנית. מקבצים אלו ניתן לבנות

את הגרסה הנוכחית של הקומפיילר (בלי התוספות שעליכם לכתוב). הגרסה הנוכחית של הקומפיילר היא תוכנית עובדת. בנוסף מצורפת תיקיה examples עם דוגמאות קלט פלט של הקומפיילר המורחב (כלומר עם חלק מהשינויים הכלולים בתרגיל). מצורף גם קובץ הרצה של הקומפיילר בגרסתו הנוכחית (myprog.exe -- יש להוריד את הסיומת .txt). קובצי ההרצה קומפלו ב- Windows 7 על מעבד 64 ביט עם הקומפיילר g++ (של GNU). (סביר שיעבדו גם על גרסאות 8 ו- 10 של Windows).

<u>מה צריך לעשות ?</u>

יש לממש את הדברים הבאים:

1. להוסיף תמיכה במשפטי for
2. להוסיף תמיכה במשפטי switch
3. להוסיף תמיכה במשפטי break.
4. להוסיף תמיכה באופרטור הלוגי fand
5. לאפשר ערבוב של אופרנדים מסוג int ו- float בביטויים אריתמטיים. בנוסף יש לתמוך במשפטי השמה בהם הטיפוס של הביטוי בצד ימין שונה מהטיפוס של המשתנה בצד שמאל.
6. להוסיף תמיכה באופרטור השארית (%)
7. להוסיף תמיכה בהכרזות של משתנים בהן מופיע ערך התחלתי

<u>מה צריך להגיש ?</u> קובץ zip הכולל את הקבצים של התוכנית עם התוספות שהכנסתם. אין להגיש את הקבצים שיצרו `flex & bison`.

הנחיות למימוש השינויים הנ"ל

<u>תמיכה במשפטי for</u>.

הכוונה למשפטים כמו למשל

```
for (i = 0; i < 10; i = i+1;)
    a = a + 3;
```

המשמעות דומה למשמעות של משפטי for בשפת C .

הדקדוק כבר כולל משפטים מהסוג הזה.
בתיקיה examples יש דוגמא לתרגום של משפטי for לקוד ביניים בקבצים for.txt, ו- for_out.txt. יש לתרגם כמו בדוגמא.

הטיפול במשפטי for מאוד דומה לטיפול במשפטי while (שכבר קיים).
הטיפול כולל:
-- עדכון המנתח הלקסיקלי כך שיכיר את האסימון (או אסימונים) הרלוונטיים.
-- הגדרה של subclass חדש ל- Stmt כדי לייצג משפטי for ב- AST.
בפרט יש להגדיר את genStmt עבור ה- class החדש.

עדכון ast.y כדי שה- parser ידע לבנות צמתים ב- AST שמייצגים משפטי for.

תמיכה במשפטי switch.

הדקדוק כבר כולל משפטים מהסוג הזה. דוגמא לתרגום לקוד ביניים מופיעה בקבצים switch_out.txt, switch.txt בתיקיה examples.
(ראו גם את הקבצים switch_no_break.txt, switch_no_break_out.txt)

בקבצים הנ"ל ניתן לראות פקודות קוד ביניים מהצורה case t V label
כאשר t הוא שם של משתנה, V הוא ביטוי פשוט (משתנה או מספר) ו- label זו תווית סימבולית. זה פשוט קיצור של הפקודה הבאה בקוד ביניים:

if t == V goto label

למשל הפקודה case t8 19 label5

זה קיצור של if t8 == 19 goto label5

למה להשתמש בקיצור? כי זה מקל על שלב מאוחר יותר בקומפילציה להבחין שיש כאן קוד שהוא מועמד לטיפול מיוחד ואז ניתן לייצר קוד (לא קוד ביניים, קוד מטרה למשל בשפת אסמבלי) שידע לטפל ביעילות בסדרת ההשוואות הזאת. זה לא רלוונטי לקומפיילר שלנו כי בשבילו קוד הביניים זה כבר הקוד הסופי אבל בכל זאת אתם מתבקשים לייצר קוד עבור משפטי switch שמשתמש בפקודות מהצורה case t v label כמו בקבצי הדוגמא.

המנתח הלקסיקלי כבר מכיר את האסימונים הרלוונטיים למשפטי switch וה- classes הרלוונטיים הוגדרו ב- ast.h (SwitchStmt ו- Case). השינויים הנחוצים ב- ast.y כבר נעשו זאת אומרת שה- parser כבר יודע לבנות צמתים ב- AST לתאור משפטי switch. כל שעליכם לעשות הוא לכתוב את הפונקציה SwitchStmt::genStmt() (כרגע היא מוגדרת כפונקציה שלא עושה כלום בקובץ gen.cpp). יתכן שתרצו גם להוסיף שדה label_ ל- class Case.

בנוסף יש להוציא הודעת שגיאה במקרה שהטיפוס של הביטוי המופיע במשפט switch אינו int. את הודעת השגיאה יש להדפיס ע"י קריאה לפונקציה errorMsg() (המוזכרת בהמשך).

תמיכה במשפטי break

משפטי break עשויים להופיע בתוך לולאות (while, for) ובתוך משפטי switch. ניתן לממש אותם בקוד ביניים ע"י קפיצה לתווית המשויכת לפקודה שמופיעה מיד אחרי הקוד עבור הלולאה (או משפט ה- switch).

דוגמא לתרגום לקוד ביניים מופיעה בקבצים
nestedWhile_with_break.txt ו-
nestedWhile_with_break_out.txt בתיקיה examples.

דרך פשוטה לממש משפטי break היא ע"י שימוש במחסנית של "exit labels". בראש המחסנית מופיעה תווית המשמשת ליציאה מהלולאה הנוכחית הפנימית ביותר. מתחתיה במחסנית מופיעה התווית עבור הלולאה המקיפה את הלולאה הפנימית ביותר וכך הלאה. (אם כרגע הקומפיילר לא מייצר קוד עבור לולאה – המחסנית תהיה ריקה). מחסנית כזאת מוגדרת בקובץ gen.cpp:

```
std::stack<int> exitlabels;
```

(זו מחסנית של int כי הקומפיילר מייצג תוויות ע"י מספרים שלמים למשל 17 מייצג את label17).

פעולות שניתן להפעיל על המחסנית:

```
exitlabels.push (int);
exitlabels.pop ();
exitlabels.empty(); // is stack empty ?
exitlabels.top ();
```

class BreakStmt כבר מוגדר בקובץ ast.h. המנתח הלקסיקלי כבר מכיר את האסימון BREAK. ה- parser כבר יודע לבנות צמתים ב- AST המייצגים משפטי break (ראו ast.y). עליכם לכתוב את הפונקציה BreakStmt::genStmt() (שמופיעה כרגע בקובץ gen.cpp כפונקציה שלא עושה כלום) ובנוסף לכך להוסיף קוד שיעשה push ו- pop למחסנית של ה- exitlabels (זה נחוץ במספר מקומות).

במקרה ש- break מופיע לא בתוך לולאה (או במשפט switch) יש להוציא הודעת שגיאה ע"י קריאה ל- errorMsg () (מוגדרת ב- ast.y).

הערה: break יכול להופיע בכל case (במשפט switch) מיד אחרי המשפט המופיע ב- case (כדי למנוע נפילה ל- case הבא) והוא גם יכול להופיע בתוך המשפט עצמו. במקרה השני ניתן לטפל ב- break בעזרת המנגנון הרגיל (עם המחסנית הנ"ל). במקרה הראשון -- השדה _hasBreak (ב- class Case) יהיה true והטיפול ב- break יכול להיעשות ב- SwitchStmt::genStmt().

תמיכה באופרטור הלוגי fand.

האופרטור מופיע בקלט כ- "$$". האסימון נקרא FAND בדקדוק.

הנה טבלת האמת של האופרטור fand (p ו- q הם ביטויים בוליאניים).

p	q	q $$ p
true	true	false
true	false	false
false	true	false
false	false	true

שימו לב שאם האופרנד השמאלי הוא true אז אין צורך לחשב את האופרנד הימני כי התוצאה במקרה זה תהיה false ללא תלות באופרנד הימני.

(השם fand הוא שם מומצא. זה קיצור של false and. fand מחזיר ערך true רק כאשר שני האופרנדים שלו הם false. זה ההיפך מ- and שמחזיר true רק כאשר שני האופרנדים הם true.)

הטיפול באופרטור fand מאוד דומה לטיפול באופרטורים or ו- and.
המנתח הלקסיקלי כבר מכיר את האסימון FAND וכלל הגזירה המתאים כבר מופיע בדקדוק. את כל השאר יש לממש: הגדרה של class fand לייצוג ביטויים בוליאניים עם אופרטור זה ב- AST. בנית צמתים כאלו ע"י ה- parser. וכתיבת הפונקציה `Fand::genBoolExp()`.

בתיקיה examples מופיע קובץ לדוגמא שבו יש שימוש באופרטור fand.
הקובץ נקרא fand.txt. (אין שם קובץ פלט מתאים.)

תמיכה בהפעלת אופרטור אריתמטי בינארי על אופרנדים מטיפוסים שונים

(מחייב הוספה של בערך 10 שורות ומחיקה של מספר קטן של שורות (שמוציאות הודעות שגיאה במקרה שאופרטור בינארי מופעל על אופרנדים מטיפוסים שונים או שבמשפט השמה הטיפוסים של צד שמאל וצד ימין הם שונים)

נרצה לאפשר (בתוכנית הקלט לקומפיילר) להפעיל אופרטור אריתמטי על אופרנדים מסוגים שונים כלומר אחד האופרנדים מטיפוס int והשני מטיפוס float. במקרה כזה, הקומפיילר צריך לייצר קוד שממיר את האופרטור מסוג int ל- float ולאחר מכן מפעיל את האופרטור (על שני ערכים מטיפוס float). למשל אם בתוכנית המקורית הוגדר

```
int k; float a;
```

אז את הביטוי k + a יש לתרגם כך:

```
_t1 = static_cast<float> k
_t2 = a
_t3 = _t1 @+ _t2
```

כאן "+@" הוא אופרטור החיבור של ערכים מסוג float (ראו בהמשך תיאור של קוד הביניים).

ההמרה מ- float ל- int בקוד הביניים נעשית ע"י הפעלת static_cast<int>.
כדי להמיר מ- int ל- float מפעילים static_cast<float>

בדומה לכך, במשפטי השמה שבהם הטיפוס של הביטוי בצד ימין שונה מהטיפוס של המשתנה בצד שמאל -- בקוד הביניים צריכה להופיע המרה מפורשת מטיפוס אחד לטיפוס האחר.
במקרה שצד ימין של משפט השמה הוא מטיפוס float והמשתנה בצד שמאל הוא מטיפוס int אז צריך לייצר קוד עם static_cast<int> אבל הקומפיילר

צריך גם להדפיס warning. ה- warning נחוץ כי השמה כזאת כרוכה באובדן מידע.
למשל אם נניח שהמשתנה i הוא מטיפוס int אז הפקודה i = 3.14 תגרום להשמה של הערך 3 ל- i (ולא הערך 3.14) כך שמידע הלך לאיבוד.

בהמשך לדוגמא הקודמת (תרגום של k+a), אם הוגדר גם int m; אז התרגום של `m = k + a;` לקוד ביניים יכיל (בנוסף לפקודות הנ"ל שמחשבות את k + a) גם את הפקודה:

```
m = static_cast<int> _t3
```

והקומפיילר צריך להוציא warning.
שימו לב ש- _t3 הוא מטיפוס float כי הטיפוס של הערך המוחזר ע"י אופרטור אריתמטי הוא int אם כל האופרנדים שלו מטיפוס int ואחרת הוא float.
דוגמאות נוספות נמצאות בקבצים `cast.txt` ו- `cast_out.txt` בתיקיה `examples`.

הערות:
בדוגמא הנ"ל a מועתק למשתנה זמני (_t2). זה בעצם מיותר אבל כך פועל הקומפיילר (לא קשה לתקן את זה).

בגרסה הנוכחית של הקומפיילר -- הפעלת אופרטור אריתמטי על אופרנדים מטיפוסים שונים נחשבת שגיאה. גם משפט השמה בו הטיפוס של הביטוי בצד ימין אינו זהה לטיפוס של המשתנה בצד שמאל נחשב לשגוי.

האמור כאן מתייחס רק לאופרטורים האריתמטיים (חיבור, חיסור, כפל וחילוק). למען הפשטות נחליט שאת האופרטורים המשמשים להשוואה (< , > וכן הלאה) ניתן להפעיל על כל סוגי האופרנדים. מותר גם שאחד האופרנדים יהיה `int` והשני יהיה `float`. במילים אחרות - אין צורך לשנות דבר הקשור ליצור קוד עבור אופרטורים המשמשים להשוואה.

תמיכה באופרטור הבינארי %

נרצה להוסיף תמיכה באופרטור השארית %
למשל 3 % 17 פרושו השארית של 17 בחלוקה ל- 3 (כלומר 2)
שני האופרנדים צריכים להיות מטיפוס int. במקרה שאחד האופרנדים או שניהם מטיפוס float אז הקומפיילר צריך להוציא הודעת שגיאה.

האופרטור מסומן ב- % גם בקלט לקומפיילר וגם בקוד הביניים.
למשל את הביטוי a % 3 ניתן לתרגם לקוד ביניים כך (בהנחה ש- a מטיפוס int):

```
_t1 = a
_t2 = 3
_t3 = _t1  % _t2
```

כדי שהקומפיילר יתמוך באופרטור החדש יש לדאוג לכך שהמנתח הלקסיקלי יזהה את האופרטור % כאסימון מסוג MULOP. לאופרטור % יש אותה עדיפות כמו לכפל וחילוק (כמו בשפת C) ולכן הוא מיוצג בדקדוק ע"י אותו סוג של אסימון כמו האופרטורים של כפל וחילוק.

בנוסף לכך יהי צורך להוסיף עוד מספר קטן של שורות קוד.

<u>הכרזות של משתנים עם ערך התחלתי.</u>
לדוגמא

```
auto foo = 3 * 4;
```

הכרזה זו מגדירה משתנה בשם foo ונותנת לו את הערך ההתחלתי 3 * 4 (כלומר 12).
הקומפיילר מסיק מה הטיפוס של המשתנה לפי הטיפוס של הביטוי שמופיע בצד ימין של הסימן "=". בדוגמא זו הטיפוס הוא int. auto היא מילה שמורה.

כלל הגזירה המתאים כבר מופיע בדקדוק.

<u>הקלט והפלט של המיני קומפיילר</u>
מצורפים קבצים עם דוגמאות לקלט ולפלט של הקומפיילר. הקבצים נמצאים בתיקיה examples. <u>קוד הביניים בדוגמאות נוצר ע"י קומפיילר שכבר תומך בתוספות הנדרשות בתרגיל.</u>
המוסכמה היא שאם קובץ הקלט נקרא foo.txt אז קובץ הפלט המתאים (המכיל את התרגום לקוד ביניים) נקרא foo_out.txt. למשל הקובץ while.txt כולל דוגמא למשפט while והקובץ while_out.txt כולל את התרגום לקוד ביניים.

<u>הקלט (שפת תיכנות פשוטה)</u>

הקלט היא תוכנית בשפת תכנות מאוד פשוטה שקל להבינה.
השפה כוללת סוגים שונים של משפטים: משפטי if, משפטי השמה, משפטי while ועוד. יש להכריז על כל משתנה שבו משתמשים.

הדקדוק של שפה זו מופיע בקובץ ast.y. יש שני סוגים של ערכים: ערכים מטיפוס int וערכים מטיפוס float. הטיפוס של משתנה נקבע לפי ההכרזה שלו. מספר שלם (למשל 3) הוא מטיפוס int. מספר עם נקודה עשרונית (למשל 3.14) הוא מטיפוס float. לכל ביטוי אריתמטי (למשל a + b / 3) יש טיפוס.
הערך המוחזר ע"י אופרטור אריתמטי בינארי (למשל +) הוא int במקרה ששני האופרנדים הם מטיפוס int ואחרת הוא float. (כלומר אם אחד מהאופרנדים או שניהם מטיפוס float אז האופרטור מחזיר ערך מטיפוס float).

<u>הפלט (קוד הביניים)</u>

הפלט הוא התרגום של הקלט לקוד ביניים מסוג three address code.
הפלט נכתב ל- standard output.

הנה דוגמאות לפקודות של שפת three address code. דברים דומים ראינו בשעורים.

```
a = b + c

if a >  3 goto  label7

ifFalse b < g goto  label2

goto label4
```

)
בכל פקודה יכול הופיע לכל היותר אופרטור אחד.

הפקודה halt מסיימת את התוכנית.

לפקודה ניתן לשייך תווית סימבולית (שלאחריה נקודותים) למשל:

```
label9:  foo = bar / stam
```

נרשה גם לשייך יותר מתווית סימבולית אחת לאותה פקודה (זה מקל על יצור הקוד במקרים מסוימים) למשל:

```
label5:
label7:
    a = b * c
```

משתנים ב- three address code יכולים להיות מטיפוס int או מטיפוס float.
אין הכרזות בשפה זו והטיפוס של משתנה נקבע לפי השימוש בו.
למשל אם מופיעה הפקודה a = 3 אז a הוא int. אם מופיעה פקודה a = 3.17 אז a הוא float. דוגמא נוספת: אם a, b הם מטיפוס float ומופיעה הפקודה c = a @+ b אז גם c מטיפוס float.

בקוד הביניים יש שני סוגים של אופרטורים אריתמטיים: אופרטורים הפועלים על ערכים מסוג int ואופרטורים הפועלים על ערכים מסוג float.
האופרטורים הפועלים על int הם: +, -, * , / (חיבור, חיסור, כפל וחילוק).
האופרטורים הפועלים על float מתקבלים ע"י הוספת התו @ ("שטרודל"):
@+, @-, @*, @/

למשל אם a, b, c הם משתנים מטיפוס int אז התרגום של
a = b + c; לקוד ביניים יהיה a = b + c
אבל אם הם משתנים מסוג float אז קוד הביניים יהיה:
a = b @+ c

כדי להפעיל אופרטור אריתמטי על שני אופרנדים שאחד מהם int והשני float - יש להמיר קודם את הערך של האופרנד מסוג int -- ל- float.
לצורך כך מותרים casts (כמו בשפת C אבל הסימון קצת אחר):
static_cast<int> a ממיר את הערך של a מ- float ל- int. static_cast<float> b ממיר את הערך של b מ- int ל- float.

במשפט השמה הטיפוס של הערך בצד ימין חייב להיות זהה לטיפוס של המשתנה בצד שמאל. אם למשל k הוא משתנה מסוג int ו- a משתנה מסוג float אז זה לא חוקי: a = k

זה כן חוקי: a = static_cast<float> k

יש גם אופרטורים להשוואה (!=, ==, <=, >=, <, >). האופרטורים האלו מופיעים תמיד בפקודות קפיצה עם תנאי למשל

```
if a > 3 goto label9
ifFalse stam == bar goto label13
```

אופרטורים של השוואה יכולים לפעול על אופרנדים מאותו טיפוס או על אופרנדים מטיפוסים שונים (מבלי שיהיה צורך בהמרה של האופרנד שהוא int ל- float).

קוד הביניים כולל גם פקודות פשוטות לביצוע קלט פלט. לכל פקודה כזאת יש שתי גרסאות: אחת עבור ערכים מסוג int והשניה עבור ערכים מסוג float.

הפקודות iread ו- fread קוראות מהקלט (ה- standard input). יש להן אופרנד אחד: שם המשתנה בו נשמר הערך שנקרא מהקלט.

iread i קוראת מהקלט ערך מסוג int וכותבת אותו למשתנה i (i משתנה מסוג int). הפקודה fread a קוראת ערך מסוג float וכותבת אותו למשתנה a (שהוא משתנה מסוג float).

הפקודות iwrite ו- fwrite כותבות לפלט (ה- standard output). יש להן אופרנד אחד: משתנה שאת ערכו יש לכתוב לפלט.

iwrite i כותבת לפלט את הערך של i (שחייב להיות משתנה מסוג int). הפקודה fwrite a כותבת לפלט את הערך של a (שחייב להיות משתנה מסוג float).

<u>בניית הקומפיילר</u>

על Windows נריץ את הפקודות הבאות בחלון המריץ את cmd.exe (או בחלון המריץ shell של MinGW או משהו דומה לכך).
במערכות הפעלה אחרות יש לעשות דברים דומים.

1. מריצים את flex:

```
flex  ast.lex
```

נוצר קובץ lex.yy.c

2. מריצים את bison עם האופציה d–

```
bison –d ast.y
```

bison יצור שני קבצים ast.tab.c ו- ast.tab.h. (את השני הוא יצור בגלל האופציה d-).

בקובץ ast.y שולבו actions הכתובים בשפת C++ (להבדיל משפת C). bison לא יודע על כך והוא מייצר (כרגיל) קוד בשפת C. בקוד זה משולבים ה- actions הכתובים בשפת C++ (אותם bison מעתיק באופן עיוור לקובץ שהוא יוצר). מאחר ושפת C היא subset של C++ נתיחס בהמשך לקבצים שיצר bison כאל קבצי C++ כלומר נקמפל אותם עם קומפיילר של C++ (ולא של C). כך נעשה גם עם הקובץ שיצר flex (שהוא קובץ C).
הערה: flex ו- bison יודעים גם לייצר קוד בשפת C++ אבל יכולת זו לא נוצלה כאן.

הערה נוספת: אין חשיבות לסדר שבו מבצעים את שני הצעדים הראשונים כלומר ניתן להריץ קודם את bison לפני שמריצים את flex)

3. עתה יש לקמפל את הקבצים שיצרו flex & bison ואת הקבצים הנוספים שכוללת התוכנית בעזרת קומפיילר לשפת C++. אם משתמשים בקומפיילר של GNU לשפת C++ (הנקרא g++) הפקודה היא (את הפקודה יש לרשום בשורה אחת):

```
g++ -o myprog.exe  ast.tab.c
                lex.yy.c gen.cpp symtab.cpp ast.cpp
```

כאן האופציה o- מציינת את שם הקובץ שהוא התוצר של הקומפילציה.
במקרה זה שם הקובץ הוא myprog.exe.

4. נכין קובץ טקסט שנקרא לו while.txt ובו נכתוב קלט לדוגמא למשל

```
int a;
int z;

while (a > 3)  z = z + 1;
```

נריץ את הפקודה

```
myprog while.txt
```

והפלט יופיע ב- standard output:

```
label1:
    _t1 = a
    _t2 = 3

    ifFalse _t1 > _t3 goto label2
    _t3 = z
    _t4 = 1
    _t5 = _t3 + _t4
    z = _t5
    goto label1
    label2:
```

מצורף לתרגיל גם קובץ Makefile למי שמעונין בכך.
קובץ זה נועד לתוכנית make שמאפשרת בנית קובץ הרצה בצורה אוטומטית. כאשר אתם מכניסים שינויים בחלק מהקבצים של התוכנית -- make תדאג לעשות את המינימום הנדרש כדי לבנות את קובץ ההרצה מחדש. למשל אם לא הכנסתם שינויים בקובץ C++ מסוים אז היא לא תקמפל אותו מחדש. אם לא הכנסתם שינויים בקובץ הקלט ל- bison אז היא לא תפעיל את bison שוב.
כמובן שלצורך כך התוכנית make צריכה להיות מותקנת על המחשב שלכם.
יתכן שתצטרכו להכניס שינויים ב- Makefile:
כרגע Makefile מניח שהקומפיילר הוא g++, קובץ ההרצה של bison נקרא win_bison (ליתר דיוק: win_bison.exe) וקובץ ההרצה של flex נקרא win_flex.

תאור המימוש של הקומפיילר

בשלב ראשון ה- parser קורא את הקלט ובונה AST (Abstract Syntax Tree).

לאחר מכן עוברים על ה- AST ומיצרים קוד ביניים.

הקלט לקומפיילר נמצא בקובץ שניתן כ- command line argument.
הפלט (Three Address Code) נכתב ל- standard output.

נוח שה- AST יהיה Object Oriented ולכן התוכנית כתובה ב- C++.

יש שלושה סוגים עיקריים של צמתים ב- AST (ראו קובץ ast.h)

הסוגים השונים של הצמתים נועדו לייצג ביטויים (expressions), ביטויים בוליאניים (boolean expressions) ומשפטים (statements).

צמתים המייצגים ביטויים אריתמטיים
אלו הם אובייקטים מ- classes הנגזרים מ- Exp (כלומר הם subclasses של Exp). אובייקטים מטיפוס BinaryOP מייצגים ביטויים המורכבים מאופרטור המופעל על שני תתי ביטויים כמו למשל (a + b) * z. (כמובן שגם תתי הביטויים עשויים להכיל אופרטורים כפי שרואים בדוגמא). אובייקטים מטיפוס NumNode מייצגים מספרים (המהווים ביטויים פשוטים). אובייקטים מטיפוס IdNode מייצגים ביטויים כמו למשל bar הכוללים רק שם של משתנה (בלי אופרטורים).
בכל צומת המייצג ביטוי נשמר הטיפוס של הביטוי בשדה _type. שדה זה מוגדר ב- class Exp כדי שכל ה- subclasses ירשו אותו.

הטיפוס של כל ביטוי (ותת ביטוי) מחושב כבר בזמן בנית העץ. ראו לדוגמא את ה- constructor של BinaryOp (בקובץ ast.cpp).

בנוסף לכך נשמר בצומת מידע נוסף בהתאם לסוג הצומת. למשל בצומת מסוג BinaryOp נשמרים גם האופרטור ומצביעים לשני האופרנדים.
(כל אחד מהמצביעים האלו מצביע לצומת ב- AST).
ה-classes היורשים מ- Exp עושים override ל- genExp method.
הגרסאות השונות של genExp משמשות ליצור קוד ביניים עבור הסוגים השונים של ביטויים.
genExp() מחזירה את המשתנה שבו תאוחסן התוצאה של חישוב הביטוי.
למשל אם היא מחזירה t17_ פרוש הדבר שהקוד שהיא יצרה עבור הביטוי יאחסן את תוצאת הביטוי במשתנה t17_.
(טכנית genExp() מחזירה int ובדוגמא זו היא תחזיר את הערך 17 שמייצג את המשתנה t17_).

שימו לב שהקומפיילר לא יודע מה תוצאת הביטוי: הוא רק מייצר קוד שיחשב "בזמן ריצה" את התוצאה הזאת.

<u>צמתים המייצגים ביטויים בוליאניים</u>
אלו הם אוביקטים מ- classes שהם subclasses של BoolExp. ה-classes ה- הם: SimpleBoolExp, Or, And ו- Not. אוביקטים מסוג SimpleBoolExp מייצגים ביטויים בוליאניים המורכבים מאופרטור השוואה המופעל על שני ביטויים אריתמטיים (לא בוליאניים).
למשל (a + b) < 17
בצמתים אלו נשמרים האופרטור ומצביעים לשני האופרנדים.

אוביקטים מסוג Or מייצגים ביטויים בוליאניים המורכבים מהאופרטור or המופעל על שני אופרנדים (שכל אחד מהם גם הוא ביטוי בוליאני).

אוביקטים מסוג And ו- Not דומים ל- Or (ל- Not יש רק אופרנד אחד).

כל class שיורש מ- BoolExp צריך לעשות override ל- genBoolExp.
הגרסאות השונות של זו מייצרות קוד ביניים עבור הסוגים השונים של ביטויים בוליאניים. קוד זה הוא "קוד עם קפיצות" כלומר הוא אמור לקפוץ לתווית מסוימת אם התנאי הבוליאני מתקיים ולתווית מסוימת (אחרת מן הסתם) כאשר התנאי אינו מתקיים. שתי התוויות האלו מועברות כארגומנטים ל- genBoolExp.
הארגומנטים נקראים truelabel ו- falselabel. כל אחד מהארגומנטים האלו יכול להיות תווית רגילה (המיוצגת ע"י מספר חיובי לדוגמא 17 מייצג את התווית label17) או FALL_THROUGH. אם למשל truelabel הוא FALL_THROUGH
פרוש הדבר שבמקרה שהתנאי מתקיים יש "ליפול" לפקודה הבאה אחרי הקוד עבור הביטוי הבוליאני. זו אפשרות שבמקרים מסוימים מאפשרת לחסוך בפקודות. למשל נרצה שהקוד שמחשב את התנאי של לולאת while "יפול" לתוך גוף הלולאה כאשר התנאי מתקיים. לצורך כך נקרא ל- genBoolExp עם הארגומנט FALL_THROUGH בתור ה- truelabel. (לעומת זאת נרצה שהוא יקפוץ לתווית המשויכת לפקודה שאחרי משפט ה- while במקרה

שהתנאי לא מתקיים). ראו את ה- WhileStmt::genStmt() method בקובץ gen.cpp וראו גם סעיף בהמשך על יצור קוד עבור ביטויים בוליאניים.

<u>צמתים המייצגים משפטים</u>

אלו הם אוביקטים מ- classes שהם subclasses של Stmt.
כל subclass כזה נועד לייצוג משפטים מסוג מסוים.
רשימה חלקית של ה- subclasses האלו:
ifStmt, WhileStmt, AssignStmt, Block, SwitchStmt

Block כאן מייצג סדרה של משפטים המוקפת בסוגריים מסולסלות.

כל צומת המייצג משפט מכיל מצביעים למרכיבי המשפט. למשל צומת המייצג משפט if יכיל מצביעים לתנאי של המשפט, למשפט שיתבצע כאשר התנאי מתקיים ולמשפט שיתבצע כאשר התנאי אינו מתקיים.

דוגמא נוספת: צומת המייצג משפטי switch מכיל מצביעים לביטוי של ה- switch, לרשימת ה- cases שלו ול- default statement שלו.

רשימת ה- cases היא רשימה מקושרת של אוביקטים מסוג Case שכל אחד מהם מכיל את המספר הקבוע של ה- case ומצביע למשפט של ה- case.
(בנוסף לכך נשמר חיווי האם יש break אחרי ה- case).

כל subclass של Stmt צריך לעשות override ל- genStmt.
הגרסאות השונות של method זה מייצרים קוד ביניים עבור סוגי המשפטים השונים.
.

<u>קוד עבור ביטויים בוליאניים (קוד עם קפיצות)</u>

הקוד שמייצר הקומפיילר עבור ביטויים בוליאניים <u>אינו</u> כותב את התוצאה (שהיאtrue או false) לתוך משתנה (כפי שעושים עבור ביטויים אריתמטיים) אלא זה "קוד עם קפיצות": הקוד קופץ למקום אחד כשהתוצאה היא true ולמקום אחר כשהתוצאה היא false.

בנוסף לכך הקוד הוא short circuit code כלומר האופרנד השני של and ו- or מחושב רק אם זה נחוץ (כמו בשפת C). למשל אם האופרנד הראשון של or הוא true אז אין צורך לחשב את האופרנד השני כי ברור שהתוצאה הסופית תהיה true. במילים אחרות, רק אם האופרנד הראשון של or הוא false יש צורך לחשב את האופרנד השני.

<u>דוגמא</u>: התרגום של

```
while  ( a > b  and y < z)
     y = y + 3;
```

יכול להראות כך:

```
label1:
   _t1 = a
   _t2 = b
```

```
   ifFalse _t1 > _t2 goto label2
   _t3 = y
   _t4 = z
   ifFalse _t3 < _t4 goto label2
   _t5 = y
   _t6 = 3
   _t7 = _t5 + _t6
   y = _t7
   goto label1
label2:
```

<u>דוגמא נוספת</u>: התרגום של

```
while  ( a > b  or y < z)
    y = y + 3;
```

יכול להראות כך:

```
label1:
   _t1 = a
   _t2 = b
   if _t1 > _t2 goto label3
   _t3 = y
   _t4 = z
   ifFalse _t3 < _t4 goto label2
label3:
   _t5 = y
   _t6 = 3
   _t7 = _t5 + _t6
   y = _t7
   goto label1
label2:
```

שימו לב שהקומפיילר מייצר את התוויות label1 ו- label2 כחלק מהטיפול במשפט ה- while. (ראו את WhileStmt::genStmt() בקובץ gen.cpp). את התווית label3 מייצרים כחלק מהטיפול ב- or. (ראו את Or::GenBoolExp() בקובץ gen.cpp. המשתנה next_label בפונקציה הזו יחזיק את label3 בדוגמא זו).
(לחילופין אפשר היה להחליט שעבור כל משפט while מייצרים תווית המשויכת לתחילת הקוד של גוף הלולאה ואז גם label3 היה נוצר כחלק מהטיפול במשפט ה- while).

<u>טבלת הסמלים (symbol table)</u>
כאן שומר הקומפיילר מידע על כל המשתנים המופיעים בתוכנית.
בפועל בקומפיילר הפשוט שלנו נשמרים עבור כל משתנה רק השם שלו והטיפוס שלו. הקומפיילר מוסיף את המשתנה לטבלת הסמלים כשהוא רואה את ההכרזה שלו. הממשק לטבלת הסמלים כולל שתי פונקציות: getSymbol() מחפשת משתנה בטבלת הסמלים ומחזירה את הטיפוס שלו. putSymbol() יוצרת

כניסה חדשה בטבלת הסמלים. הפונקציות מוגדרות בקובץ symtab.cpp ומוכרזות בקובץ symtab.h (לא תצטרכו להכניס שינויים בקבצים אלו).

<u>הפונקציה emit.</u>

קוד הביניים מודפס לפלט (ל- standard output) בעזרת קריאות לפונקציה emit() המוגדרת בקובץ gen.cpp. זו פונקציה שמקבלת מספר משתנה של ארגומנטים כלומר ניתן לקרוא לה עם ארגומנט אחד או יותר (זו המשמעות של שלוש הנקודות בהגדרה שלה). באופן מעשי, הפונקציה הזו מקבלת ארגומנטים בדיוק כמו הפונקציה printf.

הפונקציה <u>emitlabel()</u> מדפיסה לפלט תווית ואחריה נקודותיים.

<u>משתנים זמניים ותוויות סימבוליות</u>

הקומפיילר מייצר משתנים זמניים (_t1, _t2, _t3 ...) בעזרת קריאות לפונקציה newTemp() (המוגדרת בקובץ gen.cpp) . הקומפיילר מייצר תוויות סימבוליות (label1, label2, label3 ...) בעזרת קריאות לפונקציה newlabel() (שגם היא מוגדרת בקובץ gen.cpp).

הקומפיילר מייצג משתנים זמניים ותוויות סימבוליות כמספרים שלמים:

המספר 17 למשל מייצג את המשתנה _t17 (או את התווית label17).

זו צורת ייצוג פנימית של הקומפיילר. כמובן שבפלט של הקומפיילר מופיעים משתנים ותוויות סימבוליות בצורה הרגילה.

<u>הודעות שגיאה</u>

הקומפיילר עושה מספר קטן של בדיקות סמנטיות (למשל האם משתנה הוגדר לפני השימוש בו) ובמקרה הצורך מוציא הודעת שגיאה ע"י קריאה לפונקציה errorMsg () המוגדרת בקובץ ast.y. זו פונקציה שמקבלת מספר משתנה של ארגומנטים כלומר ניתן לקרוא לה עם ארגומנט אחד או יותר (זו המשמעות של שלוש הנקודות בהגדרה שלה). באופן מעשי,הפונקציה הזו מקבלת ארגומנטים בדיוק כמו הפונקציה printf. למשל ניתן לקרוא לה כך:

```
errorMsg ("line %d: %s is undefined\n",
          line, name);
```

חשוב שכל הודעת שגיאה תכיל גם את מספר השורה בה נפלה השגיאה.

לצורך כך כל אחד מה- classes הבאים (זו רשימה חלקית) כולל שדה _line שבו מאוחסן מספר השורה הרלוונטי בקובץ הקלט לקומפיילר:

```
BinaryOp, IdNode, AssignStmt, BreakStmt, SwitchStmt
```

ב- BinaryOp נשמר בשדה _line מספר השורה בקלט בה הופיע האופרטור. ביטוי כזה יכול להתפרש על פני מספר שורות בקלט. אם מעוניינים לשמור רק שורה אחת ולא טווח של שורות אז טבעי להשתמש בשורה בה הופיע האופרטור (הראשי) של הביטוי.

באופן דומה, ב- AssignStmt נשמר מספר השורה בה הופיע אופרטור ההשמה. ב- Idnode נשמר המיקום של המזהה. ב- SwitchStmt נשמר המיקום של האסימון SWITCH. ב- BreakStmt נשמר המיקום של האסימון BREAK.

(כרגע לא כל צמתי ה- AST מכילים שדה line_ . לא קשה לתקן את זה אבל זה לא נדרש בתרגיל הבית).

הנה הסבר קצר על מנגנון המיקומים (Locations) של bison

לצורך הטיפול במספרי השורות נעשה שימוש במנגנון של bison המאפשר לעקוב אחר מיקומים (מספרי שורות ומספרי עמודות) של אסימונים (וסימני דקדוק באופן כללי) בקלט.
הסימון @1 ב- action מתיחס למיקום (location) של הסימן הראשון המופיע בצד ימין של כלל הגזירה (כמו שהסימון $1 מתיחס לערך הסמנטי שלו).
הסימון @2 מתיחס למיקום של הסימן השני וכן הלאה.
למשל @2.first_line ב- action המשויך לכלל הגזירה של assign_stmt (בקובץ ast.y) מתיחס למיקום של הסימן '=' בקובץ הקלט. המנתח הלקסיקלי יכול לדווח ל- parser על המיקומים של האסימונים שהוא מזהה בקלט. זה נעשה ע"י כתיבה למשתנה הגלובלי yylloc (כפי שדיווח על הערך הסמנטי נעשה ע"י כתיבה למשתנה הגלובלי yylval).
בתוכנית שלנו זה נעשה בשורה שבה מוגדר YY_USER_ACTION בקובץ הקלט ל- flex (ast.lex).
(באופן כללי YY_USER_ACTION מבוצע בכל פעם שנמצאת התאמה לביטוי רגולרי -- לפני שמבוצע ה- action ששויך לביטוי הרגולרי).

הסבר מפורט יותר ניתן למצוא ב- manual של bison. ראו
www.gnu.org/software/bison/manual/html_node/Tracking-Locations.html#Tracking-Locations

קבצים

מצורף גם קובץ הרצה של הגרסה הנוכחית של הקומפיילר (שלא כוללת את התוספות שאתם תכתבו). הקובץ נקרא myprog.exe והוא הוכן בסביבה הנ"ל. (יש להוריד את הסיומת txt משם הקובץ לפני השימוש)

קובצי המקור של הקומפיילר:
הקובץ ast.h מכיל את ההיררכיה של ה- classes עבור ה- AST.

הקובץ ast.cpp מכיל מספר constructors של צמתים ב- AST.
חלק מה- constructors נמצאים ב- ast.h.
בדרך כלל constructors שכוללים דברים מעבר לאתחול טריוויאלי של שדות נמצאים ב- ast.cpp אבל השאלה באיזה משני הקבצים ממוקם ה- constructor לא חשובה.
הקובץ gen.cpp מכיל את המימוש של ה- methods שמייצרים את קוד הביניים. למשל BinaryOp::GenExp מייצרת קוד ביניים עבור ביטוי המורכב מאופרטור בינארי המופעל על שני אופרנדים.
דוגמא נוספת: IfStmt::genStmt מייצר קוד ביניים עבור משפטי if.

ast.lex הוא קובץ הקלט ל- flex.

`ast.y` הוא קובץ הקלט ל- `bison`.

הקובץ `symtab.cpp` כולל את המימוש של טבלת הסמלים.
(לא תצטרכו לשנות קובץ זה).
הקובץ `symtab.h` כולל את הממשק לטבלת הסמלים. יש כאן הכרזה של שתי פונקציות. `getSymbol()` מחפשת משתנה בטבלת הסמלים ומחזירה את הטיפוס שלו. ו- `putSymbol()` יוצרת כניסה חדשה בטבלת הסמלים. גם את הקובץ הזה לא תצטרכו לשנות.

הקובץ `gen.h` מכיל מספר הכרזות נוספות (הקובץ `ast.h` כולל את השורה `#include "gen.h"`).

בנוסף מצורפים קובץ Makefile ותיקיה examples עם מספר דוגמאות לקובצי קלט ופלט. מוסכמה: אם קובץ הקלט נקרא foo.txt אז קובץ הפלט (התרגום לקוד ביניים) נקרא foo_out.txt.

בהצלחה!

גרסה זו נכתבה ב- 21 לדצמבר 2018