תרגיל בית : ייצור קוד ביניים סמסטר קיץ 2021

ניתן להגיש בזוגות או בשלשות (שלשות יצטרכו לעשות קצת יותר כפי שמתואר בהמשך).

קובץ ה- zip שאתם מגישים צריך לכלול את הקבצים של התוכנית עם התוספות שהכנסתם. אין צורך להגיש את הקבצים שיצרו flex & bison. (יש להגיש גם את קובץ ההרצה או לפחות makefile).

הגישו קובץ README עם פרטי המגישים . יש לרשום גם איזה חלקים מהתרגיל עשיתם והוא עובד. הכוונה לפרוט קצר של שורה או שניים. למשל אפשר לכתוב: "הוספנו תמיכה במשפטים מסוג switch ו- break".

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 כ- אלף שורות קוד (סדר גודל. לא כולל קוד הנוצר אוטומטית ע"י flex ו- bison).
תצטרכו לכתוב (אני מעריך) מספר דו ספרתי של 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).

.

מה צריך לעשות ?

יש לממש ארבעה מהסעיפים הבאים (פרוט מופיע בהמשך).

בכל מקרה יש לממש "תמיכה במשפטי switch" ו- constant folding (ושני סעיפים נוספים לפי בחירתכם).

מי שמגיש בשלשות צריך לממש שישה מהסעיפים הבאים (כולל "תמיכה במשפטי switch", constant folding).

1. להוסיף תמיכה במשפטי repeat
2. להוסיף תמיכה במשפטי switch
3. להוסיף תמיכה במשפטי break.
4. כרגע הקומפיילר מאפשר להפעיל אופרטור בינארי אריתמטי (חיבור, כפל וכ"ו) על שני אופרנדים בעלי טיפוס זהה. אם לשני האופרנדים יש טיפוסים שונים אז הקומפיילר מוציא הודעת שגיאה. יש להוסיף תמיכה בהפעלה של אופרטורים בינאריים על אופרנדים מטיפוסים שונים.
   בנוסף יש לתמוך במשפטי השמה בהם הטיפוס של הביטוי בצד ימין שונה מהטיפוס של המשתנה בצד שמאל.
5. להוסיף תמיכה בהכרזות של משתנים עם אתחול ל- iota.
6. constant folding עבור ביטויים אריתמטיים (מוסבר בהמשך).
7. תמיכה באופרטור ^
8. תמיכה בקשר הלוגי nand.

מה צריך להגיש ? קובץ zip הכולל את הקבצים של התוכנית עם התוספות שהכנסתם. אין צורך להגיש את הקבצים שיצרו flex & bison.
(יש להגיש גם את קובץ ההרצה או לפחות makefile).

הנחיות למימוש השינויים הנ"ל

תמיכה במשפטי repeat

הכוונה למשפטים כמו למשל

```
repeat (a + b * 5)
  z = z + 3;
```

כלל הגזירה הוא

```
stmt -> REPEAT '(' expression ')'  stmt
```

משמעות המשפט: הביטוי בתוך הסוגריים מחושב פעם אחת. נסמן את התוצאה ב- n. אז גוף הלולאה יבוצע n פעמים. (אם n אינו מספר חיובי אז גוף הלולאה לא יתבצע אפילו פעם אחת).

דוגמא לקוד הנוצר ניתן לראות בתיקיה examples בקבצים repeat.txt

(זה קובץ קלט לקומפיילר) ו- repeat.3.txt (זה הקוד שהקומפיילר צריך לייצר). יש לייצר קוד דומה לדוגמא.

הביטוי בתוך הסוגריים אמור להיות מטיפוס int. אם אינו כזה אז יש להוציא הודעת שגיאה. את הודעת השגיאה יש להדפיס ע"י קריאה לפונקציה errorMsg() (המוזכרת בהמשך).

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

עדכון ast.y כדי שה- parser ידע לבנות צמתים ב- AST שמייצגים משפטי repeat. יש להוסיף גם את כלל הגזירה של משפטי repeat לדקדוק (קובץ ast.y).

תמיכה במשפטי switch.

הדקדוק כבר כולל משפטים מהסוג הזה. דוגמא לתרגום לקוד ביניים מופיעה בקבצים switch.3.txt, switch.txt בתיקיה examples.

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).

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).

הערה: יש להתייחס לכל case המופיע בקלט כאילו יש משפט break בסופו.

תמיכה במשפטי break

משפטי break עשויים להופיע בתוך לולאות while. המשמעות שלהם כמו בשפת C: צא מהלולאה. ניתן לממש אותם בקוד ביניים ע"י קפיצה לקוד שבא אחרי הקוד של משפט ה- while.

דוגמא לתרגום לקוד ביניים מופיעה בקבצים
nestedWhile_with_break.txt ו-
nestedWhile_with_break.3.txt בתיקיה examples.
(שימו לב שבדוגמא רואים משפט while המקונן במשפט while אחר.

משפט break מסיים את משפט ה- while
הפנימי ביותר המקיף את ה- break).

דרך פשוטה לממש משפטי break היא ע״י שימוש במחסנית של תוויות (labels) . יש צורך במחסנית כי משפטי while עשויים להיות מקוננים. בראש המחסנית מופיעה תווית שתשמש ליציאה ממשפט ה- while הפנימי ביותר הנוכחי (זה שהקומפיילר מייצר עבורו קוד כרגע). מתחתיה במחסנית מופיעה התווית עבור משפט ה- while המקיף את משפט ה- while הפנימי ביותר וכך הלאה. אם כרגע הקומפיילר לא מייצר קוד עבור משפט while אז המחסנית תהיה ריקה. מחסנית כזאת מוגדרת בקובץ gen.cpp:

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

(זו מחסנית של int כי הקומפיילר מייצג תוויות ע״י מספרים שלמים למשל 17 מייצג את label17).
פעולות שניתן להפעיל על המחסנית:

```
breaklabels.push (int);
breaklabels.pop ();
breaklabels.empty(); // is stack empty ? returns 1/0
breaklabels.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- breaklabels.

במקרה ש- break מופיע לא בתוך משפט while אז יש להוציא הודעת שגיאה ע״י קריאה ל- errorMsg () (מוגדרת ב- ast.y).

תמיכה בהפעלת אופרטור אריתמטי בינארי על אופרנדים מטיפוסים שונים. בנוסף תמיכה בהשמות בהן צד ימין בעל טיפוס שונה מצד שמאל.

נרצה לאפשר להפעיל אופרטור
אריתמטי על אופרנדים מסוגים שונים כלומר אחד האופרנדים מטיפוס int והשני מטיפוס float. במקרה כזה, הטיפוס של האופרנד הימני יקבע את אופן הפעלת האופרטור: אם האופרנד הימני הוא מטיפוס int אז האופרנד השמאלי (שהוא float) יומר ל- int ואז הפעולה תופעל על שני ערכים מטיפוס int (והטיפוס של התוצאה יהיה int).

אם האופרנד הימני הוא float אז האופרנד השמאלי (שהוא int) יומר ל- float

ואז הפעולה תופעל על שני ערכים מטיפוס float (והטיפוס של התוצאה יהיה float).

הערה: זו הגדרה טיפשית שנועדה רק לתרגיל. הגדרה טבעית יותר אומרת שהאופרנד שהוא מטיפוס int (שיכול להיות האופרנד השמאלי או הימני) יומר ל- float והפעולה תבוצע על שני ערכים מטיפוס float).
הערה נוספת: כמובן שניתן גם להפעיל אופרטור בינארי על שני אופרנדים מאותו הטיפוס. (את זה הקומפיילר הנתון כבר מאפשר). במקרה זה הטיפוס של התוצאה זהה לטיפוס של האופרנדים.

דוגמא: אם בתוכנית המקורית הוגדר

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

אז את הביטוי k + a יש לתרגם כך:

```
_t1 = (float) k
_t2 = _t1 <+> a
```

אבל אם הוגדר:

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

אז את הביטוי k + a יש לתרגם כך:

```
_t1 = (int) k
_t2 = _t1 + a
```

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

ההמרה מ- float ל- int בקוד הביניים נעשית ע"י הפעלת cast: (int). כדי להמיר מ- int ל- float מפעילים (float).

בנוסף לכך, יש לטפל גם במשפטי השמה בהם הביטוי בצד ימין בעל טיפוס שונה מהטיפוס של המשתנה בצד שמאל.
במקרה כזה יש לייצר קוד שממיר את הערך של הביטוי בצד ימין לטיפוס של המשתנה בצד שמאל (ע"י cast) ואז לעשות את ההשמה. במקרה של המרה מטיפוס float לטיפוס int הקומפיילר צריך גם להוציא warning כי המרה 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.

דוגמא:
אם בתוכנית מופיע:

```
float a;
float b;
int i;
...
```

```
i = a + b;
```

אז התרגום לקוד ביניים יהיה

```
_t1 = a <+> b
i = (int) _t1
```

והקומפיילר צריך להוציא warning בגלל שיש כאן השמה של ערך מטיפוס float למשתנה מסוג int. (_t1 כאן הוא מטיפוס float).

דוגמאות נוספות נמצאות בקבצים cast.txt ו- cast.3.txt בתיקיה examples. הקומפיילר (בגרסתו הנוכחית) מוציא הודעות שגיאה כשהוא רץ על הקלט cast.txt כי הוא מחשיב הפעלה של אופרטור אריתמטי על אופרנדים מטיפוסים שונים כשגיאה (וכך גם השמה בה צד ימין וצד שמאל בעלי טיפוסים שונים).

הערה:

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 ב- iota (רעיון דומה קיים בשפת Go)

כשמוגדר משתנה ניתן לאתחל אותו בערך iota (זו מילה שמורה). כלל הגזירה המתאים הוא

```
declarations:  declarations ID '=' IOTA ';'
```

המשתנה הראשון שמאותחל עם iota יאותחל עם הערך 0. המשתנה השני יאותחל עם הערך 1, השלישי יאותחל ב- 2 וכן הלאה. כל משתנה שמוגדר בצורה כזאת יהיה מטיפוס int.

לדוגמא, אם בקלט לקומפיילר מופיע

```
a = iota;
int  foo;
float bar;
b = iota;
c = iota;
```

אז הקומפיילר יצור כניסות עבור כל המשתנים האלו בטבלת הסמלים כאשר bar ירשם כבעל טיפוס float וכל השאר מטיפוס int.
בנוסף לכך, הקומפיילר ייצר את הקוד הבא כדי לאתחל את המשתנים:

```
a = 0
b = 1
```

c = 2

constant folding

זה סוג של אופטימיזציה (שיפור) של הקוד שיוצר הקומפיילר. הכוונה שהקומפיילר מחשב בעצמו ביטויים קבועים במקום לדחות את החישוב לזמן ריצה. למשל אם בקלט מופיע (נניח ש- i מטיפוס int):

i = 4 + 5;

אז במקום לייצר את הקוד הבא

```
_t1 = 4 + 5
i = _t1
```

אז הקומפיילר ייצר את הקוד

```
i = 9
```

דבר דומה יעשה גם עבור ביטויים מורכבים יותר למשל את המשפט

i = 3+4 * 5;

ניתן לתרגם כך:

```
i = 23
```

דוגמא נוספת (נניח ש- a מטיפוס float):

a = 4.0 + 5.0

יתורגם ל-

a = 9.00

(הקומפיילר הנוכחי כותב מספרים ממשיים עם 2 ספרות אחרי הנקודה).

דבר דומה יש לעשות עבור האופרטורים האריתמטיים הבינאריים:
פלוס, מינוס, כפל וחילוק.

הבהרה: מה שרשום בסעיף שעוסק בהפעלת אופרטור אריתמטי על אופרנדים מטיפוסים שונים תקף (כמובן) גם במקרים שבהם שני האופרנדים הם מספרים קבועים. במילים אחרות, אם מופעל אופרטור על שני מספרים בעלי טיפוסים שונים, יש להמיר את המספר השמאלי לטיפוס של המספר הימני ואז להפעיל את האופרטור. הטיפוס של התוצאה יהיה כמו של המספר הימני.

לדוגמא נניח שבקלט מופיע ביטוי 3 + 5.7

אז הקומפיילר ימיר את 3 לערך מטיפוס float ואז יפעיל את האופרטור +.

התוצאה היא 8.7 והטיפוס של התוצאה הוא float.

אם בקלט מופיע a = 3 + 5.7 כאשר a מטיפוס int אז הקוד הנוצר יהיה
a = (int) 8.7 (זה מוסבר בסעיף הנ"ל: אם עושים השמה של ביטוי למשתנה שהטיפוס שלו שונה מהטיפוס של הביטוי אז בקוד הנוצר יופיע cast של הביטוי לטיפוס של המשתנה).

אם מופיע בקלט b = 3+5.7 כאשר b מטיפוס float אז הקוד הנוצר יהיה
פשוט b = 8.7

אם בקלט מופיע b = 5.7 + 3 אז בקוד הנוצר יופיע b = (float) 8

במקרה זה, כאשר הקומפיילר ימיר (בעזרת cast) את 5.7 לערך מטיפוס int יתקבל הערך 5 (ולא 6) כי בשפת C המרה כזאת מעגלת כלפי מטה. שימו לב שהקומפיילר הוא שעושה את ההמרה ובקוד הנוצר לא מופיע משהו כמו (int) 5.7.

תמיכה באופרטור הבינארי ^

נרצה להוסיף תמיכה באופרטור ^ שהמשמעות שלו היא xor.
התוצאה של הפעלת אופרטור זה היא 1 אם בדיוק אחד משני האופרנדים הוא 0 ואחרת התוצאה היא 0. (זו לא בדיוק המשמעות שיש לאופרטור הזה בשפת C. בכל אופן אין חשיבות כאן למשמעות המדויקת של האופרטור).

למשל 3 ^ 17 יניב את התוצאה 0. 0 ^ 12 יתן תוצאה 1.

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: a ^ 3.

כדי שהקומפיילר יתמוך באופרטור החדש יש לדאוג לכך שהמנתח הלקסיקלי יזהה את האופרטור ^ כאסימון מסוג XOR.
(כך הוא מופיע כרגע בקובץ ast.y בהכרזה `%left XOR`
השאירו את ההכרזה הזאת ללא שינוי. היא נועדה לתת ל- XOR עדיפות ואסוציאטיביות. אין צורך לגעת בזה).

יש צורך להוסיף גם כלל גזירה לדקדוק עם האסימון XOR (דומה לכללים עבור ADDOP ו- MULOP).

בנוסף לכך יהיה צורך להוסיף עוד מספר קטן של שורות קוד.
שימו לב שהטיפוס enum op (מוגדר בקובץ gen.h) כבר כולל שם לאופרטור xor.

תמיכה בקשר הלוגי NAND

האופרטור מופיע בקלט כ- nand. האסימון נקרא NAND בדקדוק.

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

p	q	p nand q
true	true	false
true	false	true
false	true	true

false	false	true

(not (p and q) שקול ל- p nand q)

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

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

(לחילופין ניתן לוותר על המחלקה Nand ולייצג nand ב- AST כאילו היה כתוב not and בקלט).

בתיקיה examples מופיע קובץ לדוגמא שבו יש שימוש באופרטור nand.
הקובץ נקרא nand.txt. (קובץ הפלט המתאים הוא nand.3.txt).

<u>הקלט והפלט של המיני קומפיילר</u>
מצורפים קבצים עם דוגמאות לקלט ולפלט של הקומפיילר. הקבצים נמצאים בתיקיה examples.
המוסכמה היא שאם קובץ הקלט נקרא foo.txt אז קובץ הפלט המתאים (המכיל את התרגום לקוד ביניים) נקרא foo.3.txt. למשל הקובץ while.txt כולל דוגמא למשפט while והקובץ while.3.txt כולל את התרגום לקוד ביניים.
("3" כאן זה קיצור של "Three Address Code" -- הסוג של קוד הביניים בו אנו משתמשים).

<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. אם שני האופרנדים הם מטיפוס float אז התוצאה תהיה float. אם שני האופרנדים הם מטיפוסים שונים אז הטיפוס של האופרנד <u>הימני</u> יהיה גם הטיפוס של התוצאה. (הכלל האחרון הוא קצת מטופש

ונועד לצרכי התרגיל בלבד).

ניתן לקצר ולומר שבכל מקרה הטיפוס של התוצאה זהה לטיפוס של האופרנד הימני.

<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 נקודותיים) למשל:

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

נרשה גם לשייך יותר מתווית סימבולית אחת לאותה פקודה (זה עשוי להקל 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 הם (פשוט מקיפים את האופרטור המקורי ב- <>):

<+>, <->, <*>, </>

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`

ניתן להשתמש ב- casts (כמו בשפת C) כדי להמיר ערך מטיפוס אחד לטיפוס שני. למשל אם a משתנה מסוג float אז (int) a ממיר את הערך של a מ- float ל- int. (float) b ממיר את הערך של b מ- int ל- float (בהנחה ש- b מסוג int).

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 = (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 מפורשת של אופרנד מטיפוס אחד לטיפוס שני).

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:
   ifFalse a > 3 goto label2
    _t1 = z + 1
    z = _t1
    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.exe .

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 אריתמטיים (expressions), ביטויים בוליאניים (boolean expressions) ומשפטים (statements).

צמתים המייצגים ביטויים אריתמטיים

אלו הם אובייקטים מ- classes הנגזרים מ- Exp (כלומר הם subclasses של Exp). אוביקטים 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). אוביקטים מטיפוס NumNode מייצגים מספרים (המהווים ביטויים פשוטים). אוביקטים 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_.
genExp עשויה להחזיר גם מספר במקום משתנה במקרה שהיא יודעת מה תוצאת הביטוי -- כרגע היא יודעת מה התוצאה רק במקרה שהביטוי הוא מספר (אין אופרטורים בביטוי). טכנית, genExp מחזירה אוביקט מהמחלקה Object (מוגדרת בקובץ gen.h). כל אוביקט כזה יכול לייצג מספר (שלם או ממשי) או משתנה.

הערות: Object היא מחלקה רגילה. אין בשפת C++ מחלקה Object שהיא בשורש של היררכית המחלקות כפי שיש בשפת Java.

שימו לב שבדרך כלל הקומפיילר 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- ה- classes 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 -- לא רלוונטי לתרגיל של קיץ 2021).

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:
    ifFalse a > b goto label2
    ifFalse y < z goto label2
   _t1 = y + 3
   y = _t1
   goto label1
label2:
```

<u>דוגמא נוספת</u>: התרגום של

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

יכול להראות כך:

```
label1:
    if a > b goto label3
    ifFalse y < z goto label2
label3:
   _t1 = y + 3
   y = _t1
   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 תוויות סימבוליות כמספרים שלמים:

המספר 17 למשל מייצג את התווית label17).

זו צורת ייצוג פנימית של הקומפיילר. כמובן שבפלט של הקומפיילר מופיעים תוויות 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 הרלוונטית 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).

<u>הנה הסבר קצר על מנגנון המיקומים (Locations) של bison</u>

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

<u>קבצים</u>

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.3.txt.

בהצלחה!

גרסה זו נכתבה ב- 21 לדצמבר 2018. עדכונים נוספים נעשו בתאריכים הבאים:
12 ספטמבר 2019. 18 לאפריל 2020. 28 מאי 2020. 3 ספטמבר 2020. 2 ספטמבר 2021