מאי 2022

<u>תרגיל בית 3 סמסטר 2022 ב -- יצור קוד ביניים.</u>

הגשה בזוגות דרך moodle. יש להגיש את הקבצים של התכנית כולל השינויים שהכנסתם בהם.

בתרגיל זה נתון קומפיילר קטן שמתרגם קוד בשפת תכנות פשוטה לקוד ביניים. הקומפיילר נכתב בעזרת flex & bison. הקומפיילר הנתון תומך בביטויים, משפטי השמה, משפטי while ומשפטי if. הקומפיילר מקבל את שם קובץ הקלט כ- command line argument. בתור ברירת מחדל הוא קורא את הקלט מה- standard input. הפלט של הקומפיילר (קוד הביניים שהוא מייצר) נכתב ל- standard output.

התכנית הנתונה כוללת מספר קבצים:

gen.y (קובץ המיועד ל- bison)

gen.lex (קובץ המיועד ל- flex)

symboltable.c, symboltable.h (טבלת הסמלים)

utilities.c, utilities.h (מחסנית של ints)

makefile

מצורפת גם תיקיה examples הכוללת דוגמאות לקלט ופלט של הקומפיילר.

יש להרחיב את הקומפיילר כפי שיתואר כאן.

<u>ההרחבות הנדרשות הן:</u>

(1) תמיכה בשני טיפוסים: int ו- double

(2) תמיכה במשפטי for.

(3) תמיכה במשפטי break

(4) תמיכה במשפטי switch. סעיף זה הוא רשות. (בונוס 20 נקודות למי שעושה את זה.

אבל בכל מקרה הציון על תרגילי הבית לא יותר מ- 15 אחוז מהציון הסופי בהתאם לסיליבוס).

(5) תמיכה במשפטי קלט ופלט

(6) הוספת הודעות שגיאה לקומפיילר.

הנחיות:

<u>תמיכה בשני טיפוסים: int ו- double</u>

הקומפיילר הנתון לא מבחין בין טיפוסים. יש לעדכן אותו כך שידע לטפל בשני טיפוסים: int ו- double.

ההכרזה של משתנה (ב- source code שהוא הקלט של הקומפיילר) תציין מה הטיפוס שלו. לדוגמא

```
int i;
double a, b;
```

מספר ללא נקודה עשרונית (למשל 56) הוא מטיפוס int. מספר עם נקודה עשרונית (למשל 10.26) הוא מטיפוס double.

הקומפיילר ינהל טבלת סמלים (symbol table) בה הוא ישמור מידע על המשתנים המופיעים בתכנית. בקומפיילר הפשוט שלנו המידע היחיד שישמר על כל משתנה (בנוסף לשמו) יהיה הטיפוס שלו.

כשהקומפיילר יראה הכרזה של משתנה הוא יוסיף את המשתנה ביחד עם הטיפוס שלו לטבלת הסמלים. כשהקומפיילר ירצה לברר מה הטיפוס של משתנה – הוא יחפש אותו בטבלת הסמלים. טבלת הסמלים כבר ממומשת בקוד הנתון כ- hash table פשוט (כדי לאפשר חיפושים מהירים).

הממשק לטבלת הסמלים כולל 2 פונקציות -- האחת מוסיפה משתנה לטבלה והשניה מחזירה מידע על המשתנה (הטיפוס שלו). הממשק לטבלת הסמלים מופיע בקובץ symboltable.h. האימפלמנטציה של טבלת הסמלים נמצאת בקובץ symboltable.c (אין צורך להכניס שינויים בקבצים אלו).

לכל ביטוי או תת ביטוי המופיע בתכנית הקלט לקומפיילר יש טיפוס -- int או double.

כשמפעילים אופרטור בינארי (למשל +) על שני אופרנדים מאותו טיפוס (כלומר שני האופרנדים הם מטיפוס int או שניהם מטיפוס double) אז התוצאה של הפעלת האופרטור היא מאותו טיפוס.

אם אחד האופרנדים הוא מטיפוס int והשני מטיפוס double אז יש להמיר

את האופרנד מטיפוס int לערך מסוג double לפני הפעלת האופרטור. התוצאה של הפעלת האופרטור תהיה אף היא מטיפוס double.

בקוד הביניים עשויים להופיע שני סוגים של אופרטורים בינאריים:

הסוג הראשון של אופרטור פועל על שני אופרנדים מטיפוס int (ומחזיר תוצאה מסוג int). אופרטורים מסוג זה יסומנו בקוד הביניים כמקובל: +, -, *, / .

הסוג השני של אופרטורים בינאריים פועל על שני אופרנדים מטיפוס double (ומחזיר תוצאה מסוג double). אופרטורים אלו יסומנו בקוד הביניים: <+>, <->,<*>,</> .

דוגמאות: אם בקלט מופיע

```
int i, j;
double a, b;
```

אז התרגום לקוד ביניים של הביטוי i + j * 5 עשוי להראות כך:

```
t1 = j * 5
t2 = i + t1
```

התרגום של a * 7.8 + b יראה כך:

```
t3 = a <*> 7.8
t4 = t3 <+> b
```

התרגום של j * (a+i)

יראה כך:

```
t1 = (double) i
t2 = a <+> t1
t3 = (double) j
t4 = t3 <*> t2
```

כאן הפקודה t1 = (double) i ממירה את הערך של i לערך מטיפוס double

("עושה casting ל- double") ואת הערך שהתקבל כותבת ל- t1.

אם במשפט השמה הטיפוס של הביטוי בצד ימין (של ההשמה) שונה מהטיפוס של המשתנה אליו כותבים אז יש להמיר את הטיפוס של הביטוי לפני ביצוע ההשמה.

לדוגמא (בהנחה שהמשתנים הם בעלי טיפוסים כמו מקודם), את ההשמה

```
b = i + j;
```

ניתן לתרגם כך:

```
t1 = i + j
b = (double) t1
```

הקומפיילר צריך יהיה לדעת מה הטיפוס של כל ביטוי (או תת ביטוי) המופיע בתכנית.

לצורך כך נגדיר שלמשתנה הדקדוק expression יש ערך סמנטי האומר מה הטיפוס של הביטוי. הערך הסמנטי של expression ישמר ב- struct exp (מוגדר בקובץ gen.y). struct זה כולל שדה type האומר מה טיפוס של הביטוי ושדה result האומר באיזה משתנה נשמרת תוצאת הביטוי.

הקומפיילר הנתון עושה שימוש רק בשדה result של ה- struct הזה.

בקוד שתכתבו יהיה צורך להשתמש גם בשדה type.

תמיכה במשפטי for

כלל הגזירה של משפטי for הוא:

```
stmt :   FOR '(' assign_stmt   expression ';'  ID  INC ')' stmt
```

כאן INC אסימון המייצג את אחד משני האופרטורים ++, --. המשמעות של האופרטורים האלו ושל משפט ה- FOR כמו בשפת C.

בתיקיה examples המצורפת לתרגיל ניתן למצוא דוגמא לקוד עם משפט for (בקובץ for.in.txt).

התרגום המתאים לקוד ביניים נמצא בקובץ for.out.txt.

תמיכה במשפטי break

כלל הגזירה הוא:

```
stmt: BREAK ';'
```

משפט break יכול להופיע רק בלולאות (משפטי for ו- while) ובמשפטי switch.

המשמעות של משפט break היא כמו בשפת C. שימו לב שלולאות (ומשפטי switch) עשויים להיות מקוננים זה בתוך זה. משפט break יגרום ליציאה מהלולאה הפנימית ביותר שמקיפה את ה- break (או ליציאה ממשפט ה- switch הפנימי ביותר).

משפטי break ימומשו בקוד הביניים כפקודות goto. דוגמא נמצאת בתיקיה examples בקובץ break.in.txt. קוד הביניים המתאים נמצא בקובץ break.out.txt

דרך פשוטה לממש משפטי break היא בעזרת מחסנית של תוויות סימבוליות שישמשו כיעדי הקפיצות שמממשות breaks. כשמתחילים לעבוד על משפט while (או for או switch) מייצרים תווית כזאת ודוחפים אותה לראש המחסנית. כשמסיימים לעבוד על המשפט עושים pop למחסנית.

אם במהלך המעבר על משפט ה- while (או for או switch) נתקלים בפקודת break אז מייצרים פקודת goto שקופצת לתווית שבראש המחסנית. דרושה כאן מחסנית בגלל שלולאות ומשפטי switch עשויים להיות מקוננים זה בתוך זה.

בקוד שמצורף לתרגיל יש לנוחיותכם מימוש של מחסנית של int. (ראו קבצים utilities,c, utilities.h). מאחר והקומפיילר

מייצג תוויות סימבוליות באופן פנימי כמספרים שלמים (למשל 3 מייצג את label3)

ניתן להשתמש במחסנית של int כדי לשמור תוויות.

המחסנית של התוויות הסימבוליות כבר מוגדרת בקוד הנתון (קובץ gen.y) והיא נקראית exitLabelsStack. בקוד הנתון יש גם אתחול של המחסנית בעזרת קריאה לפונקציה initStack (המוגדרת בקובץ utilities.c).

<u>מימוש של משפטי switch</u>

כללי הגזירה המתארים משפטי switch נמצאים בקובץ עם הדקדוק (gen.y).

הכללים האלו קצת לא טבעיים: caselist גוזר לא רק רשימה של cases

אלא גם את הביטוי עליו עושים את ה- switch ואת הסוגר המסולסל שאחרי הביטוי.

טבעי יותר היה להשתמש בכללים האלו:

```
switch_stmt:
   SWITCH '(' expression ')' '{'  caselist  DEFAULT ':' stmtlist  '}'

caselist:  caselist CASE INT_NUM ':' stmtlist
```

caselist: %empty

השימוש בכללים כפי שהם רשומים בקובץ עם הדקדוק הוא קצת נוח יותר כי הוא מאפשר להעתיק בקלות את הערך הסמנטי של expression לערך הסמנטי של caselist.

(ליתר דיוק: הערכים הסמנטיים של expression ו- caselist יהיו שניהם structs (מסוגים שונים). את אחד השדות ב- struct של expression כדאי יהיה להעתיק לאחד השדות של ה- struct של caselist).

אפשר כמובן להשתמש בכללי הגזירה הטבעיים יותר אם רוצים.

המשמעות של משפטי switch היא כמו בשפת C. זה אומר בפרט שאם מבוצע קוד עבור case מסוים אז בהיעדר break ממשיכים ומבצעים גם את הקוד של ה- case הבא. (זה מכונה fall through). ראו דוגמא למשפט switch ולתרגום שלו לקוד ביניים בקבצים switch.in.txt ו- switch.out.txt בתיקיה examples.

כפי שניתן לראות בדוגמא, עבור כל case יש לייצר שתי תוויות סימבוליות: אחת משמשת לקפיצה לפקודת ifFalse שבודקת האם יש לבצע את ה- case והשניה משמשת לקפיצה מהקוד של ה- case הקודם ישר לקוד של ה- case הנוכחי (בשביל ה- fall through).

שימו לב שלא צריך לייצר קוד זהה לגמרי לקוד שמופיע בדוגמא. למשל המספור של התוויות הסימבוליות הוא במידה מסוימת שרירותי כי הוא נקבע ע"י הסדר שבו הן נוצרו (ע"י קריאות לפונקציה newlabel() בתכנית שייצרה את הקוד).

כמובן שאין משמעות לשורות ריקות בקובץ שמכיל את קוד הביניים אם כי הן עשויות להקל על קריאת הקוד (שזה חשוב).

בדוגמא לקוד שנוצר עבור משפט ה- switch רואים שהוצמדה תווית סימבולית גם לקוד של ה- case הראשון. אפשר לוותר על התווית הזאת כי אין אף קפיצה אליה.

כדי לתמוך במשפטי switch יש צורך להגדיר ערך סמנטי למשתנה הדקדוק caselist. הטיפוס של הערך הסמנטי הזה יהיה struct caselist המוגדר (חלקית)

בקובץ gen.y. כנראה שתצטרכו להוסיף שדה נוסף ל- struct הזה.

<u>משפטי קלט פלט.</u>

שפת הקלט של הקומפיילר כוללת משפטים פשוטים שמבצעים קלט פלט.

הנה כללי הגזירה שלהם:

input_stmt: INPUT '(' ID ') ';'

output_stmt: OUTPUT '(' expression ') ';'

משפט הקלט קורא מהקלט מספר וכותב אותו לתוך המשתנה ID.

הטיפוס של המספר שנקרא מהקלט אמור להתאים לטיפוס של המשתנה (נניח שכך יקרה מבלי שנטפל בחריגות מהכלל הזה).

משפט הפלט מחשב את הביטוי וכותב את התוצאה לפלט.

בקוד הביניים יש ארבע פקודות שעוזרות לממש את המשפטים האלו.

read i

קורא מהקלט מספר שלם וכותב אותו לתוך המשתנה i. (שאמור להיות מטיפוס int).

<read> a

קורא מהקלט מספר מטיפוס double (עם נקודה עשרונית) וכותב אותו לתוך המשתנה a. (שאמור להיות מטיפוס double).

write t1

כותב את הערך של המשתנה t1 לפלט. t1 יכול להיות משתנה זמני או משתנה שמופיע בתכנית הקלט לקומפיילר. משתנה זה הוא מטיפוס int. (במקום שם של משתנה יכול להופיע מספר שלם לדוגמא write 17).

<write> t1

כנ"ל עבור משתנים (או מספרים) מטיפוס double.

<u>הדפסת הודעות שגיאה</u>

הקומפיילר צריך להוציא הודעות שגיאה במקרים הבאים:

* שימוש במשתנה שלא הוגדר

* הגדרה של משתנה יותר מפעם אחת

* הביטוי עליו עושים switch חייב להיות מטיפוס int. אם הוא מטיפוס double יש להוציא הודעת שגיאה.

  למשל, זה לא חוקי: switch (a + b) { ... } כאשר הטיפוס של a+b הוא double.

* break יכול להופיע רק בתוך לולאה (while או for) או משפט switch.

  יש להוציא הודעת שגיאה במקרה ש- break מופיע בהקשר אחר.

* המשתנה שעושים לו increment בלולאת for (ראו כלל גזירה של משפטי for) צריך להיות מטיפוס int. אם הוא מטיפוס double יש להוציא הודעת שגיאה.

  לדוגמא כאן i חייב להיות מטיפוס int:

```
for (a = 8.0; i < 10; i++) ...
```

(שימו לב שאין חובה ש- a יהיה int. גם אין חובה שהמשתנה שמאותחל בתחילת לולאת ה- for (a בדוגמא זו) יהיה אותו משתנה שעושים לו increment (i בדוגמא) אם כי בדרך כלל הם כן יהיו אותו משתנה).

יש לקרוא לפונקציה errorMsg() כדי לכתוב את הודעות השגיאה. הפונקציה מוגדרת בקובץ gen.y. הארגומנטים שלה הם כמו של printf().

<u>תאור השפות של הקומפיילר</u>

יש להבחין בין שתי שפות: שפת התכנות הפשוטה בה נכתב הקלט לקומפיילר וקוד הביניים אותו כותב הקומפיילר לפלט.

<u>שפת הקלט לקומפיילר</u>

הדקדוק של השפה מופיע בקובץ gen.y. המשמעות של המשפטים והביטויים אמורה להיות ברורה. דוגמאות לתכניות בשפה נמצאות בתיקיה examples (בקבצים ששמם מסתיים ב- in.txt.)

<u>קוד הביניים</u>

הפקודות של קוד הביניים (שהוא מסוג Three Address Code) הן פשוטות.

דוגמאות לקוד ביניים ניתן למצוא בתיקיה examples בקבצים ששמם מסתיים ב-.out.txt

<u>טיפוסים ואופרטורים בקוד הביניים</u>

יש רק שני סוגים של טיפוסים: int ו- double. (הערה: כאן מדובר על טיפוסים שמופיעים בקוד הביניים. למעלה דובר על טיפוסים בשפת התכנות הפשוטה שבה נכתבות תכניות הקלט לקומפיילר. במקרה (או לא) -- שתי השפות תומכות באותן הטיפוסים. אבל יש הבדלים קטנים בשימוש בטיפוסים האלו. כפי שמיד נראה, בעוד שבשפת התכנות הפשוטה מותר להפעיל אופרטור בינארי על אופרנדים מטיפוסים שונים בקוד הביניים זה אסור).

למספרים בלי נקודה עשרונית יש טיפוס int. למספרים עם נקודה עשרונית יש טיפוס double.

לכל משתנה שמופיע בקוד הביניים יש טיפוס: int או double. אין הכרזות של משתנים. במקום זה הטיפוס של משתנה נקבע כשכותבים לתוכו. הטיפוס של משתנה הוא קבוע: זה לא חוקי לכתוב ערך מסוג int למשתנה ובהמשך לכתוב לתוכו ערך מטיפוס double.

למשל הפקודה a = b + c אומרת שהטיפוס של a הוא int. (כי האופרטור + מחזיר ערך מטיפוס int). הפקודה b = 7.5 אומרת שהטיפוס של b הוא double כי הטיפוס של 7.5 הוא double.

לכל ביטוי ותת ביטוי המופיע בקוד הביניים יש טיפוס.

האופרטורים +, -, *, / (המציינים חיבור, חיסור כפל וחילוק) פועלים על שני אופרנדים ששניהם מטיפוס int ומחזירים ערך מטיפוס int.

יש אופרטורים דומים שפועלים על שני אופרנדים מטיפוס double ומחזירים ערך מטיפוס double. האופרטורים האלו מסומנים <+>, <->, <*>, </>

ניתן לעשות casting כדי להמיר ערך מטיפוס int לערך מטיפוס double או להיפך.

למשל (int) a ממיר את הערך של a (שצריך להיות מטיפוס double) לערך מטיפוס int. (double) i ממיר את הערך של i (שאמור להיות מטיפוס int) לערך מטיפוס double. זה לא חוקי לעשות casting ל- int לערך שהוא מראש מטיפוס int.

(וגם לא חוקי לעשות casting ל- double לערך שהוא double).

בקוד הביניים יש גם אופרטורים המשמשים להשוואה: <, >, <=, >=, ==, !=

אופרטורים אלו יכולים להופיע רק בפקודות if ו- ifFalse (ראו בהמשך).

לאופרטור השוואה יש שני אופרנדים. למען הפשטות נרשה לאופרנדים אלו להיות מטיפוסים שונים (כלומר אחד מהם עשוי להיות מטיפוס int והשני מטיפוס double). כמובן ששני האופרנדים עשויים גם להיות מאותו טיפוס.

<u>פקודות בקוד הביניים</u>

נשתמש רק בשלושה סוגים של פקודות: פקודות השמה, פקודות קפיצה ופקודות קלט ופלט.

(בקוד ביניים "אמיתי" יש צורך בסוגים נוספים של פקודות: פקודות המשמשות לקריאה לפונקציות, פקודות המשמשות להקצאת זיכרון ...)

<u>פקודות השמה</u>

בפקודת השמה יכול להופיע לכל היותר אופרטור אחד. (גם casting זה אופרטור).

לשני הצדדים של ההשמה (הביטוי מימין לסימן ההשמה "=" והמשתנה משמאל להשמה) חייב להיות טיפוס זהה.

דוגמאות:

```
a = b // a,b are both ints or both doubles
i = j * t3    // i, j, t3 are ints
a = b </> c // a, b, c are doubles
x = 3.5 <*> z // x, z are doubles
t1 = (int) x  // x is a double; t1 is an int
t = (double) s // s is an int; t is a double
t5 = (double) 2 // t5 is a double
```

<u>פקודות קפיצה</u>

לכל פקודה בקוד הביניים ניתן להצמיד "תווית סימבולית"

למשל

```
label1:
    a = b + c
```

ניתן להצמיד לפקודה גם יותר מתווית אחת (אם כי ספק אם נזדקק לאפשרות הזאת).

למשל גם זה חוקי:

```
label1:
label2:
label3:
    a = b + c
```

היעד של פקודת קפיצה היא תווית סימבולית. יש שני סוגים של פקודות קפיצה:

קפיצות ללא תנאי וקפיצות עם תנאי.

<u>פקודת קפיצה ללא תנאי</u>

דוגמא: goto label3

<u>פקודת קפיצה עם תנאי</u>

דוגמא:

```
ifFalse a < b  goto  label7
```

פקודה זו בודקת אם התנאי a < b מתקיים. אם <u>אינו</u> מתקיים אז קופצים לתווית הסימבולית label7. אם התנאי כן מתקיים אז ממשיכים לפקודה הבאה

התנאי חייב להיות פשוט כלומר ללא אופרטורים (חוץ מאופרטור ההשוואה). אלו תנאים חוקיים:

a == b a > 7

זה לא חוקי: a+b == c

קוד הביניים כולל גם פקודה דומה שבה מופיע if במקום ifFalse. לא נזדקק לפקודה כזאת כאן. דוגמא:

```
if a < b  goto  label7
```

כאן מבצעים את הקפיצה אם התנאי a < b כן מתקיים.

<u>פקודות קלט פלט</u>

לפקודה write יש אופרנד בודד שהוא משתנה (או מספר) מטיפוס int. הפקודה כותבת את ערך המשתנה לפלט.

לפקודה <write> יש אופרנד בודד שהוא משתנה (או מספר) מטיפוס double.

הפקודה כותבת את ערך המשתנה לפלט.

לפקודה read יש אופרנד בודד שהוא משתנה מטיפוס int. הפקודה קוראת מספר (מטיפוס int) מהקלט וכותבת אותו למשתנה.

לפקודה <read> יש אופרנד בודד שהוא משתנה מטיפוס double. הפקודה קוראת מספר (מטיפוס double) מהקלט וכותבת אותו למשתנה.

דוגמאות:

```
read i  //  i has type int
<read> a // a has type double
write 8
<write> 8.5
write t1 // t1 has type int
<write> b // b has type double
```

<u>הערות נוספות</u>

(1) הרעיון הבסיסי הוא שמייצרים את קוד הביניים כבר במהלך ה- parsing.

בזמן שעושים parsing למשפט -- מייצרים את הקוד שלו. בזמן שעושים parsing לביטוי -- מייצרים את הקוד שלו. זה אפשרי כי הסדר בו מופיעים קטעי קוד הביניים עבור משפטים וביטויים (ותתי ביטויים) תואם את סדר הופעתם בקלט המקורי.

לדוגמא למשפט if יש שלושה מרכיבים שמופיעים בקלט בסדר זה: התנאי (ביטוי בוליאני), "משפט ה- then" ו- "משפט ה- else". התרגום בקוד הביניים של משפט if כולל קוד ביניים עבור שלושת המרכיבים האלו <u>באותו סדר</u>: קודם הקוד עבור הביטוי הבוליאני, לאחריו הקוד עבור משפט ה- then ובסוף הקוד עבור משפט ה- else.

בין קטעי הקוד האלו מופיעים קטעי קוד נוספים שמבטיחים שהמרכיבים ישולבו בצורה תקינה. למשל אחרי הקוד של משפט ה- then מוסיפים פקודת goto שמדלגת על הקוד של משפט ה- else.

המקרה החריג הוא במשפט for שבו הקוד עבור ה- increment (ראו כלל גזירה של משפטי for) מופיע בקלט <u>לפני</u> הקוד של גוף הלולאה אבל בקוד הביניים, הקוד עבור ה- increment מופיע <u>אחרי</u> הקוד של גוף הלולאה. כך קוד הביניים יהיה פשוט יותר.

(2) יש לכתוב את קוד הביניים ע"י קריאות לפונקציה emit (מוגדרת בקובץ gen.y).

הארגומנטים של הפונקציה הם כמו של printf. אם נחליף כל קריאה ל- emit בקריאה ל- printf עם אותם ארגומנטים אז הפלט ישאר ללא שינוי (מלבד האינדנטציה).

יש להשתמש בפונקציה emitlabel (מוגדרת בקובץ gen.y) כדי לכתוב לפלט תוויות סימבוליות (עם נקודותיים). emitlabel() לא מוסיפה אינדנטציה ולכן תוויות סימבוליות יופיעו בפלט כאשר הם צמודים לתחילת שורה בעוד שרוב הקוד מוזז קצת פנימה (emit() דואגת לאינדנטציה). כך הקוד קריא יותר.

(3) יש לייצר משתנים זמניים בעזרת קריאות לפונקציה newtemp(). יש לייצר תוויות סימבוליות בעזרת קריאות לפונקציה newlabel(). שתי הפונקציות מוגדרות בקובץ gen.y. שימו לב ששתי הפונקציות מחזירות מספר שלם. כש- newtemp() מחזירה 7 למשל אז הכוונה היא שהיא מחזירה את המשתנה הזמני t7. כמובן שכאשר משתנה זה יכתב לפלט (קוד הביניים) הוא יופיע עם השם "t7" ולא "7".

היצוג של משתנים זמניים כמספרים שלמים בתוך הקומפיילר נעשה מטעמי נוחות. זה לא ענין עקרוני. לצורך כך הוגדר (בקובץ gen.y):

```
typedef int TEMP;
```

דברים דומים נכונים גם עבור תוויות סימבוליות: כשהפונקציה newlabel() מחזירה

למשל 7 אז הכוונה היא שהיא מחזירה את התווית הסימבולית label7.

שימו לב להגדרה (בקובץ gen.y):

```
typedef int LABEL
```

(4) מותר להכניס שינויים בדקדוק כל עוד הדקדוק החדש שקול לדקדוק

הנתון. בפרט ניתן להגדיר משתני דקדוק חדשים שגוזרים את המילה הריקה. ה- action שמצרפים לכלל גזירה כזה יכול להביא תועלת.

(ראו למשל את המשתנה exit_label בקוד הנתון.

(5) הערך הסמנטי של האסימונים ADDOP, MULOP הוא מטיפוס enumeration type. לעומת זאת הערך הסמנטי של האסימון RELOP הוא מחרוזת.

לערכים הסמנטיים האלו תפקיד דומה: לציין מה סוג האופרטור. שתי צורות היצוג

(enum type ומחרוזת) הן סבירות אם כי יש כאן חוסר עקביות. (אפשר לא לגעת בזה).

(6) בקובץ gen.y יש שתי פקודות %left. הם נועדו לפתור קונפליקטים שיש בדקדוק. אין צורך לגעת בהם.

(7) הפקודה %define parse.trace

ופקודות ה- %printer נועדו לצרכי דיבוג. אם נותנים למשתנה yydebug את הערך 1

(כרגע הוא מקבל ערך 0 ב- main (קובץ gen.y)) אז כשהתוכנית רצה מקבלים "trace" של פעולת ה- parser: פרוט של כל צעדי ה- shift וה- reduce שלו

ופרוט של הערכים הסמנטיים של סימני הדקדוק. פקודת ה- %printer מאפשרת להגדיר כיצד יודפס ערך סמנטי של סימן דקדוק. יש דוגמאות בקובץ gen.y.

(8) קוד שמוקף ב- %code requires יופיע לפני ההגדרה של ה- union

בקוד ש- bison מייצר. הקוד גם יופיע ב- header file ש- bison מייצר (שכולל

גם הגדרה של ה- union). קובץ זה נקרא gen.tab.h במקרה שלנו. מאחר שבקובץ ה- flex עושים #include "gen.tab.h"

נוח להקיף ב- code requires הגדרות שמשותפות ל- lexer ול- parser (בנוסף

להגדרות של טיפוסים שצריכות להופיע לפני ה- union כי משתמשים בהן בהגדרתו).

זו הסיבה שההכרזה של הפונקציה errorMsg מופיעה בקוד המוקף ב-

%code requires

(גם ה- lexer וגם ה- parser קוראים ל- errorMsg()).

(8) בדקדוק של bison יש הבדל בין ; לבין ';'. נקודה פסיק עם גרש בכל צד זה סוג של אסימון (זה האסימון שהקוד שלו הוא קוד ה-ascii של התו נקודה פסיק). לעומת זאת נקודה פסיק בלי גרשים מסמן סוף של כלל גזירה (או סוף של מספר אלטרנטיבות של אותו משתנה).

למשל A : B | C | D;

או A: B; או A: B {...};

11 מאי 2022