פרויקט – חלק 2

נמשיך במלאכת בניית הקומפיילר והפעם נשתמש באסימונים של שפת --C על מנת לבנות עץ גזירה.
בתרגיל זה עליכם לממש מנתח תחבירי הכולל פעולות סמנטיות בסיסיות לצורך בניית העץ. המנתח התחבירי יקבל קלט של אסימונים מהמנתח הלקסיקלי שמימשתם בחלק 1 וידפיס את העץ כפלט.

דקדוק השפה --C:

יש לשים לב להבדל בין (Definition) DEF ל-(Declaration) DEC בכללי הגזירה!

```
PROGRAM      → FDEFS
FDEFS        → FDEFS FUNC_DEF_API BLK | FDEFS FUNC_DEC_API | ε

FUNC_DEC_API → TYPE id ( ) ; |
               TYPE id ( FUNC_ARGLIST ) ; |
               TYPE id ( FUNC_ARGLIST , ... ) ;

FUNC_DEF_API → TYPE id ( ) |
               TYPE id ( FUNC_ARGLIST ) |
               TYPE id ( FUNC_ARGLIST , ... )

FUNC_ARGLIST → FUNC_ARGLIST , DCL | DCL

BLK          → { STLIST }
DCL          → id : TYPE | id , DCL
TYPE         → int | float | void
STLIST       → STLIST STMT | ε
STMT         → DCL ; | ASSN | EXP ; | CNTRL | READ | WRITE | RETURN |
               BLK
RETURN       → return EXP ; | return;
WRITE        → write ( EXP ) ; | write ( str ) ;
READ         → read ( LVAL ) ;
ASSN         → LVAL assign EXP ;
LVAL         → id
CNTRL        → if BEXP then STMT else STMT | if BEXP then STMT |
               while BEXP do STMT

BEXP         → BEXP or BEXP  |
               BEXP and BEXP |
               not BEXP |
               EXP relop EXP |
               ( BEXP )
```

המשך הדקדוק בעמוד הבא

```
EXP          → EXP addop EXP  |
               EXP mulop EXP  |
```

```
                ( EXP ) |
                ( TYPE ) EXP |
                id |
                NUM |
                CALL |
                VA_MATERIALISE

NUM               → integernum | realnum
CALL              → id ( CALL_ARGS )
VA_MATERIALISE    → va_arg ( TYPE , EXP  )

CALL_ARGS         → CALL_ARGLIST |  ε
CALL_ARGLIST      → CALL_ARGLIST , EXP | EXP
```

כבכל תחביר, משתני השפה הינם המשתנים המופיעים בחלק השמאלי של החוקים (סומנו באותיות גדולות). הטרמינלים הינם האסימונים שהוגדרו בחלק 1 של הפרויקט (סומנו באותיות קטנות ותווים בודדים). המשתנה התחילי הינו המשתנה של חוק הגזירה הראשון ברשימה, כלומר, במקרה שלנו: PROGRAM .

המשימה:

1. כתבו מנתח תחבירי אשר מקבל תוכנית בשפת --C, גוזר את התוכנית ומדפיס את עץ הגזירה שלה. על מנת לעשות זאת עליכם לכתוב קובץ y עבור הכלי Bison ולעדכן את קובץ ה- Flex אשר כתבתם בתרגיל הקודם, על מנת להעביר את האסימונים למנתח התחבירי, כפי שלמדנו.
2. שלב הניתוח התחבירי יבצע את בניית עץ הגזירה באמצעות כללים סמנטיים שיוצמדו לחוקי הגזירה, תוך שימוש במבנה נתונים של עץ אשר כל צומת בו מוגדר באופן הבא:

```
typedef struct node {
  char *type;  // Syntax variable or token type for tokens
  char *value; // Token value. NULL for syntax variables
  struct node *sibling;
  struct node *child;
} Node, *NodePtr;
```

השדה `type` מכיל את סוג האסימון, כמוגדר בחלק 1 של הפרויקט. השדה `value` מכיל את ערך העזר של האסימון (בדר"כ הלקסמה), עבור האסימונים הרלוונטיים, בהתאם להגדרות בחלק 1 של הפרויקט. עבור אסימונים ללא ערך עזר שדה `value` יהיה `NULL`.

ראו ציור בעמוד הבא:

באמצעות צומת זה נוכל לייצג עץ עם שורש שהוא הורה לשלושה צאצאים כך:

```
parent, root
+---------+
| type    |
+---------+
| value   |
+---------+
| sibling |
+---------+
| child   |===+
+---------+   |          +==========+          +==========+
              |          |          |          |          |
              V          |          V          |          V
         +---------+     |     +---------+     |     +---------+
         | type    |     |     | type    |     |     | type    |
         +---------+     |     +---------+     |     +---------+
         | value   |     |     | value   |     |     | value   |
         +---------+     |     +---------+     |     +---------+
         | sibling |====+     | sibling |====+     | sibling |====+
         +---------+          +---------+          +---------+    |
         | child   |          | child   |          | child   |    V
         +---------+          +---------+          +---------+   NULL
```

שימו לב כי הצאצאים של צומת מסוים מוגדרים כרשימת צמתי הצאצאים החל מהצאצא השמאלי ביותר.

3. בסיום בניית העץ על המנתח להדפיס את עץ הגזירה. על מנת להבטיח אחידות בפלט העץ, מסופקת לכם הפונקציה `dumpParseTree` המדפיסה את העץ. הפונקציה מניחה כי קיים מצביע גלובלי בשם `parseTree` מסוג `ParserNode*`, המצביע בסיום הגזירה לצומת השורש של עץ הגזירה הנ"ל. בהתאם, על המנתח לבנות את העץ כך ששורשו מוצבע על ידי משתנה גלובלי בשם `parseTree`.
4. עליכם לספק, כמו בחלק 1 של הפרויקט, קובץ makefile לבניית קובץ ההרצה של המנתח התחבירי. שם קובץ הריצה שייצר חייב להיות **`part2`** . קובץ ריצה זה יופעל כמו בחלק 1, כלומר, יקבל את הקלט מהקלט הסטנדרטי, או מקובץ באמצעות redirection.
5. בבניית מנתח תחבירי עם bison לא מלנקג'ים עם הספריה fl, כפי שעשינו בחלק 1 של הפרויקט, ולכן יש צורך לממש פונקציית main. עם חומרי העזר לחלק 2, מסופקת לכם פונקציית main למנתח התחבירי. פונקציה זו מפעילה את פונקציית המנתח ()yyparse ולאחר מכן קוראת ל-`dumpParseTree` להדפסת עץ הניתוח, במידה והצליח.

התמודדות עם שגיאות והודעות שגיאה:

אין הנחת קלט תקין בפרויקט, כלומר יש להתמודד עם שגיאות. במקרה של גילוי שגיאה בזמן הניתוח, יש לעצור את המנתח, להוציא לפלט הסטנדרטי הודעת שגיאה, ולצאת מהמנתח 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.

במקרה של שגיאה אין להוציא כל פלט אחר מלבד הודעת השגיאה. כלומר, רק במקרה של סיום מוצלח של הניתוח יודפס עץ הגזירה.

אין צורך לטפל בשגיאות סמנטיות אלא רק בשגיאות הנובעות מהניתוח התחבירי.

* עיינו בתיעוד של Bison לגבי טיפול בשגיאות תחביריות, ובפרט בפרק בשם:

The Error Reporting Function yyerror.

הנחיות נוספות:

- הדקדוק הוא רב משמעי ביחס לחלק מהמשתנים. אין לשנות את הדקדוק כדי לפתור את הקונפליקטים. יש להגדיר עדיפויות ואסוציאטיביות (של Bison) עבור האסימונים כדי לפתור את הקונפליקטים. יש ליישם קדימויות ואסוציאטיביות כמקובל ב-C/C++ . ניתן להיעזר בסיכום הקדימויות בקישור הבא: http://en.cppreference.com/w/cpp/language/operator_precedence
- פונקציות העזר לבניית מבנה הנתונים של עץ הניתוח, הדפסתו וה-main כנ"ל מסופקות לכם בקבצים part2_helpers.c/h . יש לכלול קבצים אלו בהגשה כך שניתן לבנות את המנתח ללא צורך בהוספת קבצים נוספים באמצעות פקודת make.
- לחומרי התרגיל מצורפות דוגמאות לתוכניות קלט והפלט המצופה עבורן.
- בנוסף לדוגמאות המסופקות, מומלץ ליצור עוד קלטים לבדיקת המנתח שלכם – גם קלט שבשפה (good cases) וגם קלט שגוי (bad cases).
- פרטים נוספים על שימוש בכלי ה- Bison תמצאו במצגת התרגול ובקישורים באתר הקורס.

הוראות ההגשה

- מועד אחרון להגשה: יום א 5/1/2025 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 העליתם 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-part2-<student1_id>-<student2_id>.tar.bz2`
  לדוגמה: `proj-part2-012345678-345678901.tar.bz2`
- בארכיב יש לכלול את הקבצים הבאים:
  - את כל קבצי הקוד בהם השתמשתם (Bison, Flex, headers, וכל קובץ קוד מקור הנדרש לבניית המנתח, כולל קבצי קוד מקור שסופקו על ידי צוות הקורס) .
  - `makefile` הבונה את המנתח התחבירי - שם קובץ הריצה של המנתח התחבירי הנוצר צריך להיות **`part2`**.
  - מסמך תיעוד קצר בפורמט PDF המכיל הסבר על התוכנית שלכם, מבני נתונים בהם השתמשתם והנחות שעשיתם.
- קובץ הארכיב צריך להיות "שטוח" (כלומר, שלא ייצור ספריות משנה בעת הפתיחה אלא הקבצים ייווצרו בספריה הנוכחית).
- **בחלק זה בפרויקט יינתן דגש רב על סדר ותיעוד בקוד. לקוד מבולגן ולא מתועד <u>ירדו נקודות!</u>**
- כמו בחלק 1, סביבת הבדיקה הרשמית הינה המכונה הווירטואלית של לינוקס המסופקת לכם. ניתן לפתח במחשב אחר, אולם חובה עליכם לוודא שהתרגיל המוגש נבנה ורץ היטב במכונה הווירטואלית הרשמית. לא יאושרו דחיות בתרגיל לצורך התאמות למכונת היעד הרשמית.
- <u>תרגיל שלא יצליח להתקמפל יקבל ציון 0.</u>

בהצלחה!