סיכום קומפילציה דרכי פתרון-נועה אביאל 2022

שאלות מסוג bison ו flex-

מתחלק ל2 סוגי שאלות-

1. הפלקס נתון וצריך לכתוב את קטעי הקוד של הביזון.
2. לא נתון פלקס וצריך לכתוב גם אותו וגם את הביזון: לא יהיה במבחן.

שאלות מסוג 1-

נשים לב לסיפור שבא עם התרגיל ונבין מה הפלט שצריך להיות בסוף, עבורו ניצור מבנה (בעת הצורך) שמחזיק את הערכים הסופיים של הפלט (נגדיר בתוך בלוק (%code requires{}) נוסיף מבנה רק כאשר נרצה שעבור אסימון מסוים יהיו כמה ערכים סמנטיים בתוכו. לא חייב בתוך הקוד ריקוויקס אפשר גם בתוך union.

לאחר מכן עלינו לבנות Union שיכיל את הגדרות התרגיל למבנה הקלט ונוסיף לתוכו גם שדה של המבנה שהגדרנו, אם הגדרנו (נגדיר בתוך בלוק (%union{}).

כעת עלינו להגדיר את האסימונים של הדקדוק **כפי שהם מופיעים בFLEX!** מלבד לאסימונים המופיעים בהפלקס עלינו להגדיר גם את האסימונים בהם אנחנו משתמשים בunion (ניתן לזהותם לפי מה שכותבים בפלקס לyylval) אלו יוגדרו בצורה מעט שונה מהאסימונים של הפלקס –

למה שיש ערך סמנטי בFLEX כלומר רושמים את ערכו לyylval נרשום אותו בunion ונגדיר את הטוקן שלו %tokan <name_as_in_union_which_is_like_in_flex> TOKEN_AS_IN_FLEX

כעת נגדיר את הtype% ים הנחוצים כדי להשלים את התרגיל (למשל כמו רשימה של מבנה מסוים).

השלב הבא כבר נוגע לחוקי הדקדוק ופה נגדיר את התנהגות התוכנית בהתאם לקלט.דוגמה-

שאלה מס' 1

יש לכתוב מפרט ל- bison כך שתתקבל תוכנית כמתואר כאן.
(המפרט ל- flex נתון והוא מופיע בהמשך).

התוכנית תקרא את הקלט המתאר רשימה של קורסים המוניים ותדפיס את שם הקורס שניתן באתר edx אותו סיימו המספר הגדול ביותר של נרשמים.

עבור כל קורס מופיעים בקלט הנתונים הבאים: שם הקורס, האתר שבו ניתן, מספר הנרשמים ואחוז המסיימים. האתר חייב להיות אחד מהאתרים הבאים:
coursera, udacity, edx.

מספר הנרשמים יכול להופיע בקלט כמספר פשוט (למשל 3000) ויכול גם להופיע כמספר שלאחריו המילה thousand (למשל 52 thousand שזה כמו 52,000)

בתחילת הקלט מופיעה הכותרת MOOC.
בקלט עשויים להופיע גם שורות כאלו: ignore this line. לשורות כאלו לא אמורה להיות השפעה על הפלט של התוכנית אבל הדקדוק צריך לאפשר אותן.

מספר כלשהו של סימני white space יכולים להפריד בין האסימונים השונים.

דוגמא לקלט (כל קשר בין הנתונים כאן לבין המציאות הוא מקרי):

```
<course> "functional programming" <site> edx
  <enrolled> 80 thousand <completion rate> 25%

<course> "python"  <site> coursera
   <enrolled> 3500 <completion rate> 10%

ignore this line
ignore this line

<course> "compilers" <site> coursera <enrolled> 330000
   <completion rate> 100%

<course> "Haskell" <site> udacity <enrolled> 4 thousand
   <completion rate> 50%

ignore this line

<course> "AI" <site> edx <enrolled> 100000
   <completion rate> 5%
```

בדוגמא זו, על התוכנית להדפיס

```
Highest number of students completed the course:
"functional programming"
```

שימו לב שיש לקחת בחשבון רק קורסים ב- edx.
את הקורס "functional programming" סיימו
80,000 * 25% = 20,000 סטודנטים. (את AI סיימו רק 5000 סטודנטים)

```
%code requires {
  // semantic value of courselist: name of course with
  // maximum completions
  // also semantic value of course: name of course and
  // number of completions

   struct maxi {int max; char name[SIZE]; }
}
%union {

  char cname[SIZE];
  enum site_type site;
  int num;
  struct maxi m;
}

%token COURSE SITE ENROLLED RATE IGNORE_THIS_LINE MOOC
%token THOUSAND
%token <cname> CNAME
%token <num> NUM number
%token <site> SITE_NAME

%type <m> courselist course

start: MOOC courselist {
     if ($2.max > 0)
       print("Highest number of students completed the
course: %s\n", $2.name) }

courselist: courselist course { if ($1.max > $2.max)
                                  $$ = $1;
                                else
                                  $$ = $2; }
courselist: /* empty */ { $$.max = -1; }

courselist: courselist IGNORE_THIS_LINE { $$ = $1; }

course:
 COURSE CNAME SITE SITE_NAME ENROLLED number RATE NUM '%'
  { strcpy($$.name, $2; $$.max = $6*$8/100; }
```

הניתו שמוגדר enumeration type כזה:

```
enum site_type { EDX, COURSERA, UDACITY };
```

הנה המפרט ל- flex:

(הערה: הפונקציה atoi ממירה מחרוזת למספר מטיפוס int. לדוגמא atoi ("12") מחזירה 12 כמספר מטיפוס int)

```
%%

"<course>"     { return COURSE; }
"<site>"       { return SITE; }
"<enrolled>"   { return ENROLLED; }
"<completion rate>" { return RATE; }
"ignore this line"  { return IGNORE_THIS_LINE; }
"MOOC"              { return MOOC; }

\"[A-Z][a-z]*(" "[A-Z][a-z]*)*\"     {
        strcpy (yylval.cname, yytext+1);
        yylval.cname[strlen(yylval)-1] = '\0';
        return CNAME; }

"edx"       { yylval.site = EDX; return SITE_NAME; }
"coursera"  { yylval.site = COURSERA; return SITE_NAME; }
"udacity"   { yylval.site = UDACITY; return SITE_NAME; }

"thousand"  { return THOUSAND; }


[0-9]+ { yylval.num = atoi(yytext);
        return NUM; }

  "%"         { return '%'; }

  [ \t\n]+                  /* skip white space */

  .       { fprintf (stderr, "illegal token: %c",
                    yytext [0]); exit (1); }
```

<u>הערות-</u>

בכללי גזירה ה-$$ מציין את הערך הסמנטי שמוחזר למשתנה.

בכללי גזירה כדי לעשות reduce ממשתנה למשתנה, צריך לכתוב :

Expression : term($$=$1), ואז בעצם ניתן לעבור מ-term ל-expression. וכשרוצים לעשות reduce לביטוי, צריך לבדוק את התנאים של הביטוי בכלל גזירה של במשתנה. כל זה לפי מצב המחסנית.

Union- הביזון יודע ישירות שמשתנה זה הוא בעצם yylval.

תרגילים מסוג טבלת LL1 והרצת הפארסר על קלט מסוים-

בניית הטבלה –

עמודות – הטרמינליים האפשריים ודולר (אותיות קטנות, לא צריך עמודה לאפסילון) שורות- נונ-טרמינליים.

איך ממלאים? עבור כל שורה נבדוק האם ניתן להגיע לטרמינל של העמודה, במידה וכן נכתוב את הכלל הראשון בסדרת הגזירה שמביא אותנו לטרמינל הזה.

איך מריצים את הפארסר על קלט?

עמודות- כלל גזירה, יתרת הקלט, מחסנית. עבור המחסנית ויתרת הקלט נוסיף $ בסוף (סימון לסוף קלט). מתחילים מ S$ בעמודת המחסנית ובקלט שמים את הקלט משורשר עם $ בסוף.

איך ממלאים? עבור על שלב נבצע את הגזירה לפי הכלל המתאים כך שנתקרב לתוצאת הקלט הרצויה. אם יש לנו טרמינל במחסנית ננסה להתאים אותו אם יש לנו נונ-טרמינל במחסנית נגזור אותו.

Top down

סעיף א (10 נקודות) נתון הדקדוק הבא

(1) S -> H A
(2) S -> c a
(3) A -> a b
(4) A -> G
(5) G -> g
(6) H -> epsilon

בנו טבלת LL(1) של הדקדוק הנתון.
תשובה:

	a	b	c	g	$
S	S -> HA		S -> ca	S -> HA	
A	A -> ab			A -> G	
G				G -> g	
H	H -> epsilon			H -> epsilon	

FIRST(HA) = {a, g} FOLLOW(H) = {a, g}

סעיף ב (5 נקודות)
הראו את ריצת ה- parser המשתמש בטבלת ה- LL(1) הנ"ל על הקלט g.

השלימו את הטבלה הבאה (אין צורך למלא את כל השורות).

בעמודה הימנית יש לרשום את כלל הגזירה בו משתמשים (או match).
בעמודה של תוכן המחסנית -- הסימן השמאלי ביותר הוא שנמצא בראש המחסנית.

כלל גזירה בו משתמשים	יתרת הקלט	תוכן המחסנית
	g$	S$
S -> HA	g$	HA$
H -> epsilon	g$	A$
A -> G	g$	G$
G -> g	g$	g$
match(g)	$	$
accept		

שאלות על לינקר סטטי –

יש לשים לב לentry point של פונקציה – האם מוגדר יותר מפעם אחת.

יש לשים לב לכתובות יחסיות בקוד סגמנט – האם בטווח הגיוני.

שאלות על איזה מחלקי הקומפיילר מזהה איזה שגיאות בקוד-

Parser- כאשר הקלט לא מתאים לדקדוק כלומר יש syntax error לדוגמה חסר אופרנד בתנאי, חסר סוגריים, חסר נקודה פסיק אחרי שורה, הופעת else בלי if, חוסר התאמה לתנאי הדקדוק.

מנתח סמנטי- קשור לכללים של שפת התכנות, עושה את בדיקות על טיפוסים (type check), שימוש במשתנה מבלי שהוגדר קודם, הגדרה של משתנה פעמיים באותו scope, קריאה לפונקציה עם מס' שונה של ארגומנטים, קריאה לפונקציה עם ארגומנט מטיפוס לא נכון, הפעלה של אופרטור על אופרנדים מטיפוס לא מתאים.

מנתח לקסיקלי- תפקידו לזהות אסימונים בקלט ולהוציא הודעות שגיאה על אסימונים לא חוקיים, למשל אם נפתחה הערה ולא נסגרה, אם יופיע אסימון בשפת C כך- a=b+c$ ה$ לא תקין.

שגיאות שמתגלות בזמן לינק- חסרה הגדרה של פונקציה שמשתמשים בה. אם בקובץ C מסוים יש קריאה לפונקציה)שהחתימה שלה הוכרזה קודם כנדרש לפי כללי C) אבל הפונקציה לא מוגדרת באף מקום אז לא הקומפיילר אלא ה- linker הוא שיגלה את השגיאה.

שגיאות שמתגלות בזמן קומפילציה- מאלוק נכשל.

למשל, בשפת C אם עושים include "foo.h" ולא קיים קובץ כזה אז ה **preprocessor** -הוא שיגלה את השגיאה.

כתיבת recursive descent parser לדקדוק-

צריך להסתכל על First (ואם יש כלל של גזירת אפסילון אז גם על follow) של המשתנה שמבקשים לכתוב עבורו פונקציה. כל טרמינל שניתן להגיע אליו כך שיהיה הראשון משמאל אחרי גזירה יקבל case. עבור כל case נבדוק לאיזה כלל צריך לקרוא כדי להשלים את הפעולה (כך שהוא משמאל) ובמידת הצורך לעשות match לטרמינלים שאחריו (ולעצמו במידה והגיע דרך גזירת אפסילון על נונטרמינל שהיה לפניו). אם הנונטרמינל עבורו כותבים את הפונקציה מופיע בסוף כלל גזירה של S יהיה case של $ בו לא נבצע כלום אם יש כלל גזירה מהנונטרמינל לאפסילון.

שאלה 3 (15 נקודות)

סעיף א (10 נקודות) נתון הדקדוק הבא

(1) S -> Ac
(2) S -> z A
(3) A -> Bac
(4) A -> epsilon
(5) B -> G
(6) B -> epsilon
(7) G -> gz

recursive descent parser עבור הדקדוק הזה כולל פונקציה עבור כל אחד ממשתני הדקדוק. יש לכתוב (רק) את הפונקציה עבור המשתנה A. השתמשו לצורך כך במשתנה הגלובלי lookahead ובפונקציות match () ו- error () (אותן אין לכתוב).

הפונקציה match () מקבלת אסימון כארגומנט. היא משווה אותו ל- lookahead.
אם הם שווים היא מתקדמת בקלט אחרת היא קוראת לפונקציה error ().

הניחו גם שלכל סוג של lookahead ישנה הגדרה כמו לדוגמא: #define a 300

פתרון

השורה הרלוונטית בטבלת LL(1) של הדקדוק הנתון היא:

$	z	g	c	a	
A -> epsilon		A -> Bac	A -> epsilon	A ->Bac	A

FIRST(Bac) = { a, g }

FOLLOW(A) = { c, $}

הנה הפונקציה עבור המשתנה A :

```
void A() {
    switch(lookahead) {

    case a: case g:   B(); ,match(a); match(c); break; // A -> Bac

    case c:  case $:  break;   // A -> epsilon
    default: error();
}
```