מבוא

Yacc הוא מנתח סינטקטי. כלומר הוא מקבל קלט שמזוהה כיחידות לקסיקליות (או כתווים בודדים) ש-Lex זיהה, ומנסה לבדוק האם הקלט תקין מבחינה תחבירית.
למעשה אנו מגדירים ב-Yacc רשימת כללי דקדוק, עם קטע תוכנית לביצוע לכל כלל. קטע התוכנית מבוצע כאשר אנו משתמשים בכלל הדקדוק שלו.

כדוגמה לכלל דקדוק ב-Yacc נוכל לכתוב:

```
date :          month_name    day    ','        year ;
```

כפי שלמדתם בהרצאות, כלל זה נכתב בהגדרת דקדוק תיאורטי כ:

Date → month_name day ',' year

ברור כי month_name, day ו-year חייבים להיות מוגדרים באיזושהי צורה. הסימן פסיק תחום תחת גרש בודד משני הצדדים אומר כי אנו מעוניינים בסימן פסיק כתו ממש ולא כמשתנה דקדוקי. הסימנים נקודותיים ונקודה פסיק מחויבים בכתיבת כלל ב-Yacc.

מבנה תוכנית ב-Yacc

בדומה למבנה תוכנית ב-Lex, מבנה תוכנית ב-Yacc הוא:

```
{definitions}
%%
{rules}
%%
{user subroutines}
```

דוגמה לצמד תוכניות ב-Lex ו-Yacc

נבנה צמד תוכניות המזהה תאריך (כשהשנה החל משנת 2000 ועד 2099) בפורמט של חודש שנה (לדוגמה June 2012).
נתחיל מתוכנית ה-Lex (בקובץ test2.l) :

```
%%
January          return JANUARY;
February         return FEBRUARY;
March            return MARCH;
April            return APRIL;
May              return MAY;
June             return JUNE;
July             return JULY;
August           return AUGUST;
September        return SEPTEMBER;
October          return OCTOBER;
November         return NOVEMBER;
December         return DECEMBER;
20[0-9][0-9]     return YEAR;
.                return yytext[0];
%%
```

1) lex test2.l נבצע קומפילציה של Lex ע"י

קיבלנו את הקובץ lex.yy.c .

כעת נבנה את קובץ ה-Yacc בשם test2.y:

```
%token JANUARY
%token FEBRUARY
%token MARCH
%token APRIL
%token MAY
%token JUNE
%token JULY
%token AUGUST
%token SEPTEMBER
%token OCTOBER
%token NOVEMBER
%token DECEMBER
%token YEAR
%%
s:      month ' ' YEAR
        {
                printf("Date Ok\n");
        };
month: JANUARY |  FEBRUARY | MARCH | APRIL | MAY | JUNE | JULY | AUGUST | SEPTEMBER |
OCTOBER | NOVEMBER | DECEMBER;
%%
#include "lex.yy.c"
main()
{
        return yyparse();
}

int yyerror()
{
        printf("Error in date\n");
        return 0;
}
```

בחלק האחרון **חובה לכתוב את קטע הקוד שכתבנו לצורך פעילות ותקשורת בין Lex ל-Yacc**. אם יש צורך, נוכל להגדיר גם פונקציות נוספות משלנו.

כעת נקמפל את קובץ ה-Yacc (ניצור את הקובץ y.tab.c):

2) yacc test2.y

וכעת נקמפל את הקובץ y.tab.c :

3) cc –o test2 y.tab.c -ll -Ly

כעת נוכל להריץ את הקובץ test2 ע"י

4) ./test2<test2.t

Yacc מקבל קלט ומנסה בעצם לחזור לאחור מתוך הקלט כולו אל משתנה הגזירה ההתחלתי. במלים אחרות, ניתן לומר כי Yacc בודק האם הקלט הוא מלה חוקית בדקדוק שהגדרנו, דהיינו האם ניתן להגיע מהמשתנה ההתחלתי ע"י כללי גזירה אל הקלט שלנו.

הפונקציה yyerror מופעלת כאשר לא הצלחנו לחזור מהקלט אל המשתנה ההתחלתי, כלומר כאשר הקלט הוא מלה שאינה בשפה. במקרה זה ביקשנו להדפיס שהקלט אינו תאריך. יש לזכור כי **חובה לממש את הפונקציה yyerror (לפחות שתחזיר 0)**.

המנתח הסינטקטי מקבל יחידות טרמינליות מ-Lex, ומנסה לחזור מהקלט אל המשתנה ההתחלתי. אם הצליח הרי שזה קלט חוקי בדקדוק. המנתח הסינטקטי ממומש בעזרת אוטומט מחסנית והוא מסוגל לבצע שתי פעולות עיקריות:

- **Shift** – כלומר לקבל מ-Lex עוד יחידה טרמינלית של הקלט.
- **Reduce** – כלומר לבצע צמצום בעזרת כלל דקדוק. **הצמצום מתבצע בצד ימין של הקלט**.

נדגים זאת בעזרת רצף הכללים הבא:

r1: E → E + E
r2: E → E * E
r3: E → id

נניח כי הקלט הוא id+id*id. לכן זהו רצף הפעולות ש-Yacc יבצע:

1) .id+id*id
2) id.+id*id (shift)
3) id+.id*id (shift)
4) id+id.*id (shift)
5) id+id*.id (shift)
6) id+id*id. (shift)
7) id+id*E. (reduce r3)
8) id+E*E. (reduce r3)
9) id+E. (reduce r2)
10) E+E. (reduce r3)
11) E. (reduce r1)

Yacc מטפל בקונפליקטים בצורה הבאה:

- בקונפליקט Shift/Reduce הוא מבצע Shift. קונפליקט זה קיים בשורות 2,3,4,5,6
- בקונפליקט Reduce/Reduce הוא מבצע את פעולת ה-Reduce שתשתמש בכלל שהופיע מוקדם יותר מבחינת סדר הופעת הכללים.

Yacc מדווח על מספר הקונפליקטים משני הסוגים שפגש בדקדוק, על אף שהוגדרו לו החוקים הנ"ל לפתירתם.
תפקידנו כמתכנתים לפתור את הקונפליקטים משני הסוגים.
ניתן לפתור אותם על ידי שכתוב של כללי הדקדוק.

פעולות בתוך כלל דקדוק והעברת ערכים

נשכלל את התוכנית שבנינו. נבנה את הקובץ test3.l :

```
%%
January         {       yylval=1;
                        return JANUARY;
                }
February        {       yylval=2;
                        return FEBRUARY;
                }
March           {       yylval=3;
                        return MARCH;
                }
December        {       yylval=12;
                        return DECEMBER;
                }
20[0-9][0-9]    {       yylval=atoi(yytext);
                        return YEAR;
                }
. return yytext[0];
%%
```

כעת נבנה את הקובץ test3.y

```
%token JANUARY
%token FEBRUARY
%token MARCH
%token DECEMBER
%token YEAR
%%
s:      month   ' ' YEAR
        {
                printf("We can write the date as %d/%d\n",$1,$3);
                printf("Date Ok\n");
        };
month: JANUARY
        {
                $$=$1;
        }                       |
        FEBRUARY
        {
                $$=$1;
        }                       |
        MARCH
        {
                $$=$1;
        }                       |
        DECEMBER
        {
                $$=$1;
        };

%%
```

```
#include "lex.yy.c"
main()
{
	return yyparse();
}

int yyerror()
{
	printf("Error in date\n");
	return 0;
}
```

השינוי הוא השימוש במשתנה - **yylval** זהו משתנה מיוחד המקשר בין Lex ל-Yacc. בתוכנית ה-Lex אנו מבקשים להעביר ל-Yacc את מספר החודש, בנוסף לסימון שמצאנו טרמינל שהוא חודש מסוים. אנלוגית, אנו מבקשים להעביר ל-Yacc גם את השנה עצמה, בנוסף לסימון שמצאנו טרמינל שהוא שנה. יש לזכור כי **ברירת המחדל של yylval הוא int, כלומר הוא מעביר ל-Yacc רק ערכי int.**

בכל כלל דקדוק, **$1** הוא הערך שחזר עם הטרמינל/הלא טרמינל הראשון מצד ימין (**מתוך הנחה שאכן חזר ערך כזה**), **$2** עם זה השני מצד ימין, וכך הלאה. **$$** הוא הערך אותו אנו מחזירים (**מתוך הנחה שאנו אכן רוצים להחזיר**) לרמה אחת למעלה בעץ הגזירה. יש לזכור כי **ברירת המחדל של משתני הדולר הוא int**.

באופן ספציפי בדוגמה שלנו, כלל הגזירה בו המשתנה בצד שמאל הוא month נחשב כ-4 כללים שונים. בגוף כל אחד מהכללים מבוצעת השורה **$$=$1**, דהיינו הערך $1 של הטרמינל שהגיע מ-Lex מועבר רמה אחת למעלה בעץ הגזירה, כלומר מספר החודש מועבר רמה אחת למעלה בעץ הגזירה.