

# Parsing

בדיקה האם המשפט נכון תחבירית  
(☹️ חלק לאט כדאי, לנסוע הכביש אז)  
(☺️ הכביש חלק, אז כדאי לנסוע לאט)

# חימום

שגיאות?	קוד
	<pre>void main(void) {     (((8))); }</pre>

# חימום

שגיאות?	קוד
Build: 1 succeeded	<pre>void main(void) {     (((8))); }</pre>

# חימום

שגיאות?	קוד
Build: 1 succeeded	<pre>void main(void) {     (((8))); }</pre>
	<pre>void main(void) {     (((8)); }</pre>

# חימום

שגיאות?	קוד
Build: 1 succeeded	<pre>void main(void) {     (((8))); }</pre>
<pre>syntax error: missing ')' before ';' syntax error: missing ')' before ';' syntax error: missing ')' before ';' </pre>	<pre>void main(void) {     (((8)); }</pre>

# חימום

שגיאות?	קוד
Build: 1 succeeded	<pre>void main(void) {     (((8))); }</pre>
<pre>syntax error: missing ')' before ';' syntax error: missing ')' before ';' syntax error: missing ')' before ';' </pre>	<pre>void main(void) {     (((8)); }</pre>
	<pre>void main(void) {     5;;;;; }</pre>

# חימום

שגיאות?	קוד
Build: 1 succeeded	<pre>void main(void) {     (((8))); }</pre>
<pre>syntax error: missing ')' before ';' syntax error: missing ')' before ';' syntax error: missing ')' before ';' </pre>	<pre>void main(void) {     (((8)); }</pre>
Build: 1 succeeded	<pre>void main(void) {     5;;;;; }</pre>

# חימום

שגיאות?	קוד
	<pre>void f(int a[]) {  }</pre>



# חימום

שגיאות?	קוד
<pre>Build: 1 succeeded</pre>	<pre>void f(int a[]) {  }</pre>

# חימום

שגיאות?	קוד
<pre>Build: 1 succeeded</pre>	<pre>void f(int a[]) {  }</pre>
	<pre>void f() {     int a[]; }</pre>

# חימום

שגיאות?	קוד
<pre>Build: 1 succeeded</pre>	<pre>void f(int a[]) {  }</pre>
<pre>error C2133: 'a' : unknown size</pre>	<pre>void f() {     int a[]; }</pre>

# חימום

שגיאות?	קוד
Build: 1 succeeded	<pre>void f(int a[]) {  }</pre>
error C2133: 'a' : unknown size	<pre>void f() {     int a[]; }</pre>
	<pre>void f() {     int a[10.0]; }</pre>

# חימום

שגיאות?	קוד
Build: 1 succeeded	<pre>void f(int a[]) {  }</pre>
error C2133: 'a' : unknown size	<pre>void f() {     int a[]; }</pre>
error C2058: constant expression is not integral	<pre>void f() {     int a[10.0]; }</pre>

# חימום

שגיאות?

קוד

```
void f()  
{  
    int i=8;  
    int j=3;  
  
    i---+---j;  
}
```

# חימום

שגיאות?

Build: 1 succeeded

קוד

```
void f()  
{  
    int i=8;  
    int j=3;  
  
    i---+---j;  
}
```

# חימום

שגיאות?

Build: 1 succeeded

קוד

```
void f()  
{  
    int i=8;  
    int j=3;  
  
    i--+--j;  
}
```

```
void f()  
{  
    int i=8;  
    int j=3;  
  
    i-----j;  
}
```



# חימום

שגיאות?

קוד

Build: 1 succeeded

```
void f()
{
    int i=8;
    int j=3;

    i--+--j;
}
```

error C2105: '---' needs l-value

```
void f()
{
    int i=8;
    int j=3;

    i-----j;
}
```

ולקינות, האם קיימת פונקציה g כך שהקוד יעבור קומפילציה?

האם יכול לעבור קומפילציה??	קוד
	<pre>void main(void) {     g()++; }</pre>

ולקינות, האם קיימת פונקציה g כך שהקוד יעבור קומפילציה?

האם יכול לעבור קומפילציה??	קוד
	<pre>void main(void) {     g()++; }</pre>
	<pre>int &amp;g() {     int x;      return x; }  void main(void) {     g()++; }</pre>

ולקינות, האם קיימת פונקציה g כך שהקוד יעבור קומפילציה?

קוד	האם יכול לעבור קומפילציה??
<pre>void main(void) {     g()++; }</pre>	
<pre>int &amp;g() {     int x;      return x; }  void main(void) {     g()++; }</pre>	<p>Build: 1 succeeded</p>

# שפת הסוגריים המאוזנים

- נתבונן בשפה הפרימיטיבית הבאה, שפת הסוגריים המאוזנים: בשפה יש אותיות מסוג סוגריים: (, ) וכן את כל הספרות. מילה חוקית בשפה היא מהצורה  $((... (8) ...))$ , כאשר המספר תחום משני צדדיו על ידי מספר שווה של סוגריים.
- האם השפה רגולרית? כלומר, האם קיים אוטומט דטרמיניסטי שמקבל אותה? האם קיים אוטומט לא דטרמיניסטי שמקבל אותה? האם אפשר למצוא ביטוי רגולרי שמייצר את כל המילים בשפה?

# שפת הסוגריים המאוזנים אינה רגולרית!

- הוכחה: בשלילה, קיים אוטומט דטרמיניסטי סופי שמקבל אותה, ונסמן את מספר המצבים שלו  $d$ . נתבונן בביטוי שמורכב מ  $d+1$  סוגריים שמאליים, המספר 77, ואז  $d+1$  סוגריים ימניים. בכל קריאה של ( נוספים, בוודאות נשנה מצב, כי אנו אמורים להיות בהיערכות שונה אם ראינו 4 סוגריים ( או 26 סוגריים ), למשל. קבלנו סתירה למספר מצבי האוטומט.

# שפת הסוגריים המאוזנים – את מי היא מעניינת בכלל?

- בהינתן ביטוי סוגריים כנ"ל (מילה), הקומפילר של שפת C מסוגל לענות בקלילות על שאלת השייכות שלה לשפת הסוגריים המאוזנים. איך? פשוט נרשום `int i=((...(8)...))`;
- הביטוי מאוזן אם ורק אם הקומפילציה תצליח
- כלומר, השלבים הראשונים של קומפילציה, שמוודאים שהתוכנית אכן חוקית בשפת התכנות, דורשים מנגנון זיהוי חזק יותר מ"סתם" אוטומטים סופיים. מהו?

# שפות חסרות הקשר

- שפות חסרות הקשר (מעל אלפבית כלשהו) הן שפות שנוצרות על ידי שימוש בכללי גזירה. יש בהן משתנה התחלה  $S$ , משתנים נוספים, טרמינלים, וכללי הגזירה.
- למשל, השפה  $S \rightarrow aSb$ ,  $S \rightarrow c$  מכילה את המילים:  
 $\{acb, aacbb, aaacbbb, aaaacbbbbb, \dots\}$
- האם שפת הסוגריים המאוזנים היא שפה חסרת הקשר? כן! נרשום את הדקדוק שלה על הלוח



# שפות חסרות הקשר – המשך

- כללי הגזירה בתוספת המשתנים והטרמינלים  
נקראים הדקדוק (grammar)
- נמצא את הדקדוק של השפה  
 $\{ w\#w^{reverse}\# \mid w \in \{0,1\}^* \}$
- האם יש שפות שאינן חסרות הקשר? כן!
- האם נוכיח שיש שפות שאינן ח"ה בקורס? לא!
- האם יכולים להיות כמה דקדוקים שמתארים אותה  
שפה? בהחלט!

# Predictive Parser – הגדרה

- עבור חלק מהדקדוקים חסרי ההקשר קיים predictive parser, שזה אומר 2 דברים:
  - היכולת לקחת מילה בשפה, לסרוק אותה פעם אחת משמאל לימין, ובכל פעם שנראה אותה חדשה, נדע מייד איזה כלל גזירה הופעל, כדי לתת את האות הזאת.
- אם המילה אינה בשפה, נעצור במקום הראשון שמראה זאת.

# Predictive Parser – דוגמא

- לשפת הסוגריים המאוזנים משקף 2 קיים predictive parser.
- כך למשל, המילה ((800)) נבנתה על ידי שימוש בכלל  $S \rightarrow (S)$  ולסיום בכלל  $S \rightarrow \text{INT}$
- בהינתן מילה שאינה בשפה, כמו (((800))) נוכל לדעת כבר בסוגר הימני השני שהיא אינה שייכת

# מיון – Predictive Parser

```
void G()
{
    switch (tok) {
        case (LPAREN):
            // G ---> (G)
            Eat(LPAREN);
            G();
            Eat(RPAREN);
            break;

        case (INT):
            // G ---> int
            Eat(INT);
            break;

        default:
            printf("Error in position %d\n\n", EM_tokPos);
            printf("unexpected %s\n\n", tokname(tok));
            exit(0);
    }
}

void Eat(int expectedToken)
{
    if (tok == expectedToken)
    {
        tok=yylex();
    }
    else
    {
        status = 0;

        printf("Error in position %d\n\n", EM_tokPos);
        printf("Should be %s instead of %s\n\n",
            tokname(expectedToken),
            tokname(tok));
        exit(0);
    }
}
```

# Predictive Parser – תוצאות הרצה

- כשנריץ על הקלט ((8)) נקבל תוצאה



```
C:\Windows\system32\cmd.exe  
BALANCED  
Press any key to continue . . . █
```

- וכשנריץ על הקלט ((8)) נקבל תוצאה



```
C:\Windows\system32\cmd.exe  
Error in position 4  
Should be END_OF_FILE instead of RPAREN  
Press any key to continue . . . █
```

## שפת הסוגריים המאוזנים 2

- מצאו דקדוק חסר הקשר שיוצר שפה של סוגריים מאוזנים מעל א"ב עם 6 אותיות: { }, [ ], ( ), ומקבל מילים כמו  $(([[[]\{\}\}])$  אבל לא  $((\{\}))$ . האם קיים עבורו predictive parser?

# Predictive Parser (2)

```
void S()
{
    switch (tok) {

        case (LPAREN):

            // S ----> (S)S
            Eat(LPAREN);
            S();
            Eat(RPAREN);
            S();
            break;

        case (LBRACK):

            // S ----> [S]S
            Eat(LBRACK);
            S();
            Eat(RBRACK);
            S();
            break;

        case (LBRACE):

            // S ----> {S}S
            Eat(LBRACE);
            S();
            Eat(RBRACE);
            S();
            break;

    }
}
```

```
void Eat(int expectedToken)
{
    if (tok == expectedToken)
    {
        tok=yylex();
    }
    else
    {
        status = 0;

        printf("Error in position %d\n\n",EM_tokPos);
        printf("Should be %s instead of %s\n\n",
            tokname(expectedToken),
            tokname(tok));
        exit(0);
    }
}
```

# שפת המחשבון

- שפה שמקבלת ביטויים חשבוניים עם פעולות כפל, חיבור, חיסור וחילוק. מותר לבצע פעולות עם סוגריים. כמה אותיות יש בשפה?
- האותיות בשפה:  $(, ), +, -, *, /, 0, \dots, 9$  ולא לשכוח את האות 'רווח'
- ביטוי לדוגמא:  $(3+5)*(7/2)$
- האם שפת התכנות C מסוגלת לקבל ביטוי חשבוני ולהגיד האם הוא חוקי?



# דקדוק חסר הקשר של שפת המחשבון

$$E \rightarrow \text{INT}$$

$$E \rightarrow (E)$$

$$E \rightarrow E + E \bullet$$

$$E \rightarrow E * E \bullet$$

$$E \rightarrow E / E \bullet$$

$$E \rightarrow E - E \bullet$$

- האם השיטה שבה עבדנו בשפת הסוגריים

מתאימה גם כאן?

- כלומר, האם קיים predictive parser שרואה בכל

פעם תו אחד של הקלט ויודע איזה כלל נגזר?

# רקורסיה שמאלית

- הדקדוק של שפת המחשבון לא ניתן ל predictive parsing
- נסתכל על 2 כללי הגזירה  $E \rightarrow INT$ ,  $E \rightarrow E + E$
- אם התו הראשון שראינו בקלט הוא המספר 5, אי אפשר לדעת האם לפנינו  $5+8$  או אולי רק 5. אי אפשר לנבא!
- המצב הזה נגרם בגלל רקורסיה שמאלית.
- המשתנה  $E$  גוזר כלל בו **הוא עצמו** נמצא מייד משמאל לחץ:  
 $E \rightarrow E + E$ , ובנוסף כלל אחר  $E \rightarrow INT$ . כשנראה בביטוי הסופי  $INT$ , לא נוכל לדעת מאיזה כלל הגיע ה  $INT$  הזה.
- רוצים predictive parser? תהיו חייבים לבטל ראשית רקורסיה שמאלית!

# ביטול רקורסיה שמאלית

- נניח שיש שני כללים מהצורה:  $X \rightarrow \alpha$ ,  $X \rightarrow X\gamma$   
כאשר  $\alpha$  לא מתחיל עם  $X$ , אז מה שיכול להיווצר  
הוא  $\alpha\gamma^*$  כלומר  $\alpha$  ואז מספר כלשהו של הופעות  
של  $\gamma$  ברצף. במקרה כזה פשוט נגדיר כללים  
אחרים שיגזרו את אותו סט של ביטויים:

$$\begin{pmatrix} X \rightarrow X\gamma \\ X \rightarrow \alpha \end{pmatrix} \Rightarrow \begin{pmatrix} X \rightarrow \alpha X' \\ X' \rightarrow \gamma X' \\ X' \rightarrow \epsilon \end{pmatrix}$$

# ביטול רקורסיה שמאלית – המשך

- מה עושים אם יש כמה כללים מהצורה  $X \rightarrow X\gamma$ ?
- ובנוסף כמה כללים מהצורה  $X \rightarrow \alpha$ ?
- כלומר נניח שיש  $X \rightarrow X\gamma_1$ ,  $X \rightarrow X\gamma_2$ ,  $X \rightarrow X\gamma_3$
- ונניח שיש  $X \rightarrow \alpha_1$ ,  $X \rightarrow \alpha_2$
- באופן מאוד דומה לקודם, המחרוזות האפשריות כאן מתחילות או ב  $\alpha_1$  או ב  $\alpha_2$  ואחר כך בא מספר כלשהו של גאמות.
- למשל  $\alpha_2\gamma_1\gamma_3\gamma_2\gamma_2$  או  $\alpha_1\gamma_3\gamma_3\gamma_3\gamma_1$  או  $\alpha_2$  וכו'
- הנה הדקדוק שלנו: (כמה אלפות וכמה גאמות יש כאן?)
- $E \rightarrow \text{INT}$
- $E \rightarrow E + E$
- $E \rightarrow (E)$
- $E \rightarrow E * E$
- $E \rightarrow E / E$
- $E \rightarrow E - E$

# ביטול רקורסיה שמאלית – המשך

$$\begin{pmatrix} X \rightarrow X\gamma \\ X \rightarrow \alpha \end{pmatrix} \Rightarrow \begin{pmatrix} X \rightarrow \alpha X' \\ X' \rightarrow \gamma X' \\ X' \rightarrow \epsilon \end{pmatrix} \text{ במקום להשתמש ב}$$

$$\begin{pmatrix} X \rightarrow X\gamma_1 \\ X \rightarrow X\gamma_2 \\ X \rightarrow X\gamma_3 \\ X \rightarrow X\gamma_4 \\ X \rightarrow \alpha_1 \\ X \rightarrow \alpha_2 \end{pmatrix} \Rightarrow \begin{pmatrix} X \rightarrow \alpha_1 X' \\ X \rightarrow \alpha_2 X' \\ X' \rightarrow \gamma_1 X' \\ X' \rightarrow \gamma_2 X' \\ X' \rightarrow \gamma_3 X' \\ X' \rightarrow \gamma_4 X' \\ X' \rightarrow \epsilon \end{pmatrix} \text{ ואם יש כמה:}$$

# ביטול רקורסיה שמאלית בשפת המחשבון

- לפני ביטול רקורסיה שמאלית:

$$\begin{aligned} E &\rightarrow \boxed{\text{INT}} \\ E &\rightarrow \boxed{(E)} \end{aligned}$$

$$\begin{aligned} E &\rightarrow E \boxed{*} E \cdot \\ E &\rightarrow E \boxed{/} E \cdot \\ E &\rightarrow E \boxed{+} E \cdot \\ E &\rightarrow E \boxed{-} E \cdot \end{aligned}$$

- אחרי ביטול רקורסיה שמאלית:

$$\begin{aligned} E &\rightarrow \boxed{\text{INT}} E' \\ E &\rightarrow \boxed{(E)} E' \end{aligned}$$

$$\begin{aligned} E' &\rightarrow \boxed{*} E E' \cdot \\ E' &\rightarrow \boxed{/} E E' \cdot \\ E' &\rightarrow \boxed{+} E E' \cdot \\ E' &\rightarrow \boxed{-} E E' \cdot \\ E' &\rightarrow \epsilon \cdot \end{aligned}$$

# Predictive Parser for calculator language – correct?

```
void E()
{
    switch (tok) {

        case (LPAREN):

            // E ----> (E)E'
            Eat(LPAREN);
            E();
            Eat(RPAREN);
            E_tag();
            break;

        case (INT):

            // E ----> INT
            Eat(INT);
            E_tag();
            break;

    }
}
```

```
void E_tag()
{
    switch (tok) {
        case (TIMES):
            // E' ----> * E E'
            Eat(TIMES);
            E();
            E_tag();
            break;

        case (DIVIDE):
            // E' ----> / E E'
            Eat(DIVIDE);
            E();
            E_tag();
            break;

        case (PLUS):
            // E' ----> + E E'
            Eat(PLUS);
            E();
            E_tag();
            break;

        case (MINUS):
            // E' ----> - E E'
            Eat(MINUS);
            E();
            E_tag();
            break;

    }
}
```

# Predictive Parser for calculator language – correct!

```
void E()
{
    switch (tok) {

        case (LPAREN):

            // E ----> (E)E'
            Eat(LPAREN);
            E();
            Eat(RPAREN);
            E_tag();
            break;

        case (INT):

            // E ----> INT
            Eat(INT);
            E_tag();
            break;

        default:

            printf("Error in position %d\n\n", EM_tokPos);
            printf("unexpected %s\n\n", tokname(tok));
            exit(0);

    }
}
```

```
void E_tag()
{
    switch (tok) {
        case (TIMES):
            // E' ----> * E E'
            Eat(TIMES);
            E();
            E_tag();
            break;

        case (DIVIDE):
            // E' ----> / E E'
            Eat(DIVIDE);
            E();
            E_tag();
            break;

        case (PLUS):
            // E' ----> + E E'
            Eat(PLUS);
            E();
            E_tag();
            break;

        case (MINUS):
            // E' ----> - E E'
            Eat(MINUS);
            E();
            E_tag();
            break;

    }
}
```



# דקדוק מול שפה

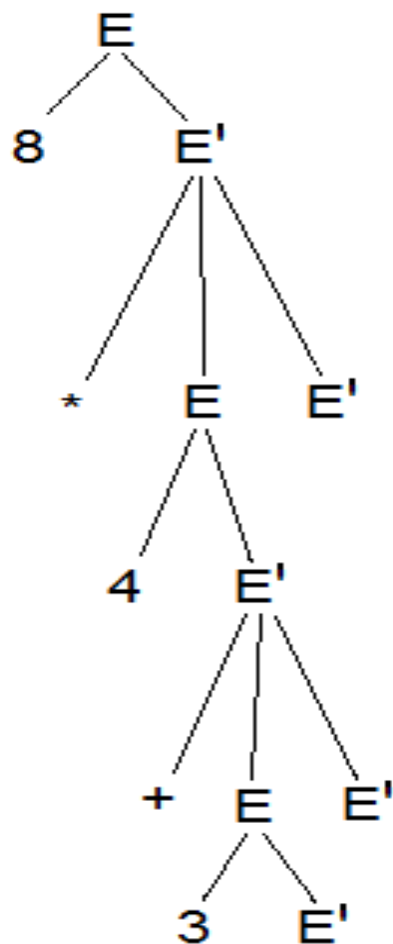
- לשפה מסוימת יכולים להיות המון דקדוקים שונים שמתארים אותה!
- ראינו עכשיו שהיכולת לבנות predictive parser תלויה בדקדוק! כלומר, יכול להיות שלשפה מסוימת יש שני דקדוקים שעבור אחד מהם אפשר לבנות predictive parser ועבור השני לא!
- דקדוק שעבורו ניתן לבנות predictive parser נקרא  $LL(1)$
- שפה שקיים דקדוק  $LL(1)$  שמתאר אותה נקראת  $LL(1)$

# עצי גזירה

- בהינתן ביטוי חשבוני, אנו יודעים כעת כיצד לבדוק אם הוא שייך לשפת המחשבון. כלומר, האם מבחינה תחבירית (syntax) הוא משפט חוקי.
- נקודה שעד עתה לא נגענו בה היא הבנת המשמעות של הביטוי, הסמנטיקה שלו.
- במהלך ה parsing, ניתן לבנות עץ גזירה, ובאמצעותו ניתן מאוחר יותר להבין את משמעות הביטוי (בשפת המחשבון למשל, ראשית לוודא שהביטוי חוקי, ואז גם לחשב אותו)

# עצי גזירה – המשך

- הביטוי  $8 * 4 + 3$  הוא כמובן ביטוי חשבוני חוקי.



- E
- $\text{INT}(8)E'$
- $\text{INT}(8) + EE'$
- $\text{INT}(8) * \text{INT}(4)E' E'$
- $\text{INT}(8) * \text{INT}(4) * EE' E'$
- $\text{INT}(8) * \text{INT}(4) + \text{INT}(3)E' E' E'$
- מה הבעיה כאן?

# הבנת המשמעות (סמנטיקה) של עצי גזירה

## בשפת המחשבון

- הביטוי  $3 + 4 * 8$  ועץ הגזירה שלו, מדגימים את הבעיה בדקדוק שלנו. אמנם אפשר לפענח את משמעות העץ כך שקדימות האופרטורים הרגילה תשאר – אבל זהו מידע נוסף שאינו נמצא בעץ ועל המתכנת לשמור אותו בנפרד ולהשתמש בו. סביר בדקדוק קטן כמו שלנו אבל בלתי סביר בדקדוק אמיתי של שפת תכנות שלמה, שהוא עצום.
- נחפש דקדוק חדש לשפת המחשבון שיכיל בעצמו את המידע של קדימויות האופרטורים. כלומר, מה שדרוש לפענוח המידע בעץ הגזירה יימצא כבר בעץ עצמו.

# דקדוק נוסף לשפת המחשבון שיכיל בתוכו את המידע על קדימות האופרטורים

$$F \rightarrow \text{INT}$$

$$T \rightarrow T * F$$

$$E \rightarrow E + T \bullet$$

$$F \rightarrow (E)$$

$$T \rightarrow T / F$$

$$E \rightarrow E - T \bullet$$

$$T \rightarrow F$$

$$E \rightarrow T \bullet$$

- נשאל ראשית, האם קיים עבור הדקדוק הזה  
predictive parser?

- גם כאן יש לנו רקורסיה שמאלית. אם אתם רוצים  
predictive parser, תהיו חייבים לבטל רקורסיה  
שמאלית.

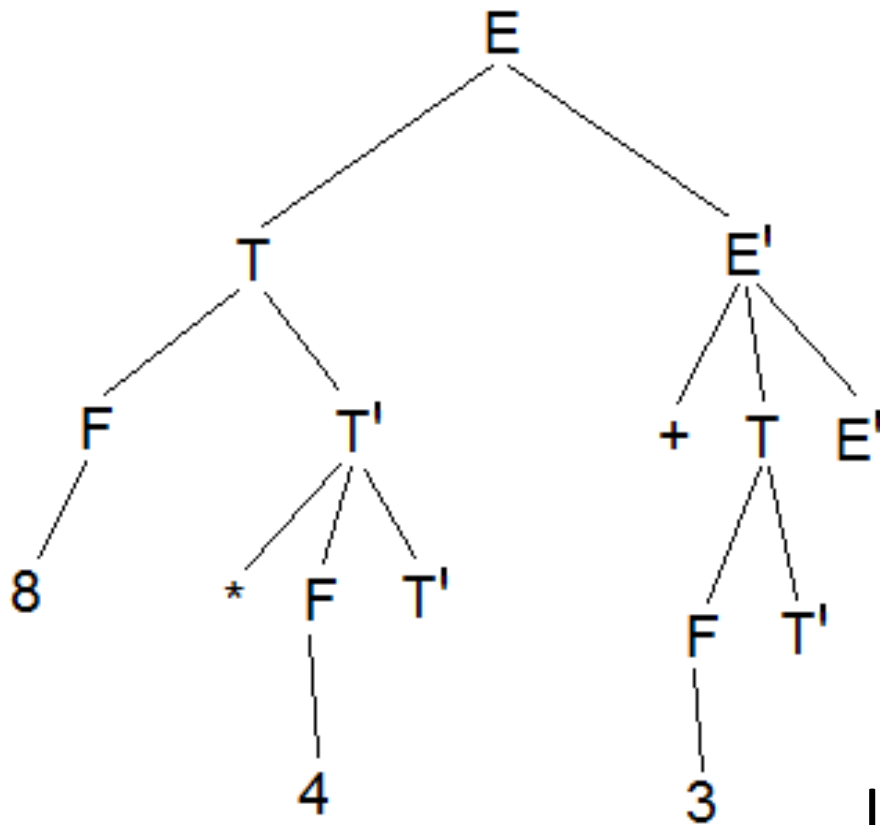
לפני ביטול רקורסיה שמאלית:

$$\begin{array}{lll}
 F \rightarrow \text{INT} & T \rightarrow T * F & E \rightarrow E + T \cdot \\
 F \rightarrow (E) & T \rightarrow T / F & E \rightarrow E - T \cdot \\
 & T \rightarrow F & E \rightarrow T \cdot
 \end{array}$$

אחרי ביטול רקורסיה שמאלית:

$$\begin{array}{lll}
 F \rightarrow \text{INT} & T \rightarrow FT' & E \rightarrow TE' \cdot \\
 F \rightarrow (E) & T' \rightarrow * FT' & E' \rightarrow + TE' \cdot \\
 & T' \rightarrow / FT' & E' \rightarrow - TE' \cdot \\
 & T' \rightarrow \epsilon & E' \rightarrow \epsilon \cdot
 \end{array}$$

# עץ גזירה לביטוי $8 * 4 + 3$ בדקדוק החדש של שפת המחשבון



- $E$  •
- $E \rightarrow TE'$  •
- $FT' E'$  •
- $INT(8) T' E'$  •
- $INT(8) * FT' E'$  •
- $INT(8) * 4 T' E'$  •
- $INT(8) * 4 E'$  •
- $INT(8) * INT(4) + TE'$  •
- $INT(8) * INT(4) + FT' E'$  •
- $INT(8) * INT(4) + INT(3) T' E'$  •

# Left Factoring

- ראינו שרקורסיה שמאלית מפריעה לבניית predictive parser וראינו כיצד להתגבר עליה. האם יש עוד בעיות?
- מה תאמרו למשל על הדקדוק הבא:
- $E \rightarrow \text{if } (Y) \text{ then } E$
- $E \rightarrow \text{if } (Y) \text{ then } E \text{ else } E$
- $E \rightarrow \text{int}$
- אותו משתנה גוזר שני כללים שיש להם התחלה זהה



# Left Factoring – solution

- נהפוך את הדקדוק:

- $E \rightarrow \text{if } (E) \text{ then } E$

- $E \rightarrow \text{if } (E) \text{ then } E \text{ else } E$

- $E \rightarrow \text{int}$

- לדקדוק:

- $E \rightarrow \text{if } (E) \text{ then } E \text{ } X$

- $X \rightarrow \epsilon$

- $X \rightarrow \text{else } E$

- $E \rightarrow \text{int}$

# Nullable Rules

- מה תאמרו על הדקדוק הבא:
- $S \rightarrow A a b$
- $A \rightarrow a$
- $A \rightarrow \epsilon$
- רקורסיה שמאלית – אין!
- Left factoring – אין!
- אז אפשר לבנות predictive parser?
- לא לא לא!

# תיקון דקדוק עם nullable rules על ידי substitution

- בדקדוק:
- $S \rightarrow A a b$
- $A \rightarrow a$
- $A \rightarrow \epsilon$
- נחליף את  $A$  בנגזרים האפשריים שלו, ונקבל:
- $S \rightarrow a a b$
- $S \rightarrow ab$
- האם עבור הדקדוק החדש קיים predictive parser?
- כן! נבטל left factoring ונקבל
- $S \rightarrow aX$
- $X \rightarrow ab$
- $X \rightarrow b$

# סיכום הבעיות הנפוצות שמונעות בניה של predictive parser

1. רקורסיה שמאלית

2. Left Factoring

3. Nullable rules

- האם ישנן עוד בעיות שיכולות למנוע predictive parsing? כן!

- תיאור האלגוריתם הכללי FIRST, FOLLOW

# Predictive parsing is NOT always possible!

- הנה דקדוק שאי אפשר לתקן אותו כך שיהיה ניתן לבנות לו predictive parser:

$$S \rightarrow A \bullet$$

$$S \rightarrow B \bullet$$

$$A \rightarrow a A b \bullet$$

$$A \rightarrow \epsilon \bullet$$

$$B \rightarrow a B b b \bullet$$

$$B \rightarrow \epsilon \bullet$$

# Predictive Parsing – Is it always desirable???

- האם עבור דקדוקים מורכבים יותר, כמו דקדוקים של שפות תכנות אמתיות זה בכלל רצוי להתחיל לתקן את הדקדוק כך שניתן יהיה לבנות עבורו predictive parser?
- דקדוק של שפת תכנות סבירה בדרך כלל עמוס ברקורסיה שמאלית, left factoring, ו nullable rules
- בנוסף, גם אם אפשר זה לא תמיד רצוי – הדקדוק ייהפך להיות מסובך ומסורבל.
- ועוד: קדימויות אופרטורים רבות (Visual Studio)
- יש דרך טובה יותר לתכנן parser עבור דקדוקים אלה

# הדקדוק חסר ההקשר של שפת C

- שימו לב לגודלו העצום של הדקדוק חסר ההקשר של שפת C:

- <http://www.lysator.liu.se/c/ANSI-C-grammar-y.html>, שימו לב לרקורסיה שמאלית (כמעט בכל כלל..)

```
multiplicative_expression
```

```
: cast_expression
```

```
| multiplicative_expression '*' cast_expression
```

```
| multiplicative_expression '/' cast_expression
```

```
| multiplicative_expression '%' cast_expression
```

```
;
```

```
additive_expression
```

```
: multiplicative_expression
```

```
| additive_expression '+' multiplicative_expression
```

```
| additive_expression '-' multiplicative_expression
```

```
;
```

# Predictive Parser for JavaScript

- מעבר לקלות המימוש, predictive parsers הם מהירים מאוד. בלינק הבא נמצא המימוש של chrome ל predictive parser של JavaScript  
<http://src.chromium.org/svn/trunk/src/tools/gn/parser.cc>
- יש הרבה הנהלת חשבונות, אבל הרעיון הכללי הוא בדיוק מה שראינו. על סמך התו הראשון מתבצעת ההחלטה איזה כלל גזירה הופעל, ולאילו פונקציות לקרוא.



# דוגמא לטיפול בכלל גזירה של if

```
scoped_ptr<ParseNode> Parser::ParseCondition() {
    scoped_ptr<ConditionNode> condition(new ConditionNode);
    Consume(Token::IF, "Expected 'if'");
    Consume(Token::LEFT_PAREN, "Expected '(' after 'if'.");
    condition->set_condition(ParseExpression());
    if (IsAssignment(condition->condition()))
        *err_ = Err(condition->condition(), "Assignment not allowed in 'if'.");
    Consume(Token::RIGHT_PAREN, "Expected ')' after condition of 'if'.");
    condition->set_if_true(ParseBlock().Pass());
    if (Match(Token::ELSE))
        condition->set_if_false(ParseStatement().Pass());
    if (has_error())
        return scoped_ptr<ParseNode>();
    return condition.PassAs<ParseNode>();
}
```