

# Bottom Up Parsing

אותו דבר כמו Top Down Parsing רק הפוך!  
בונים את עץ הגזירה מלמטה למעלה

# 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
```

```
;
```

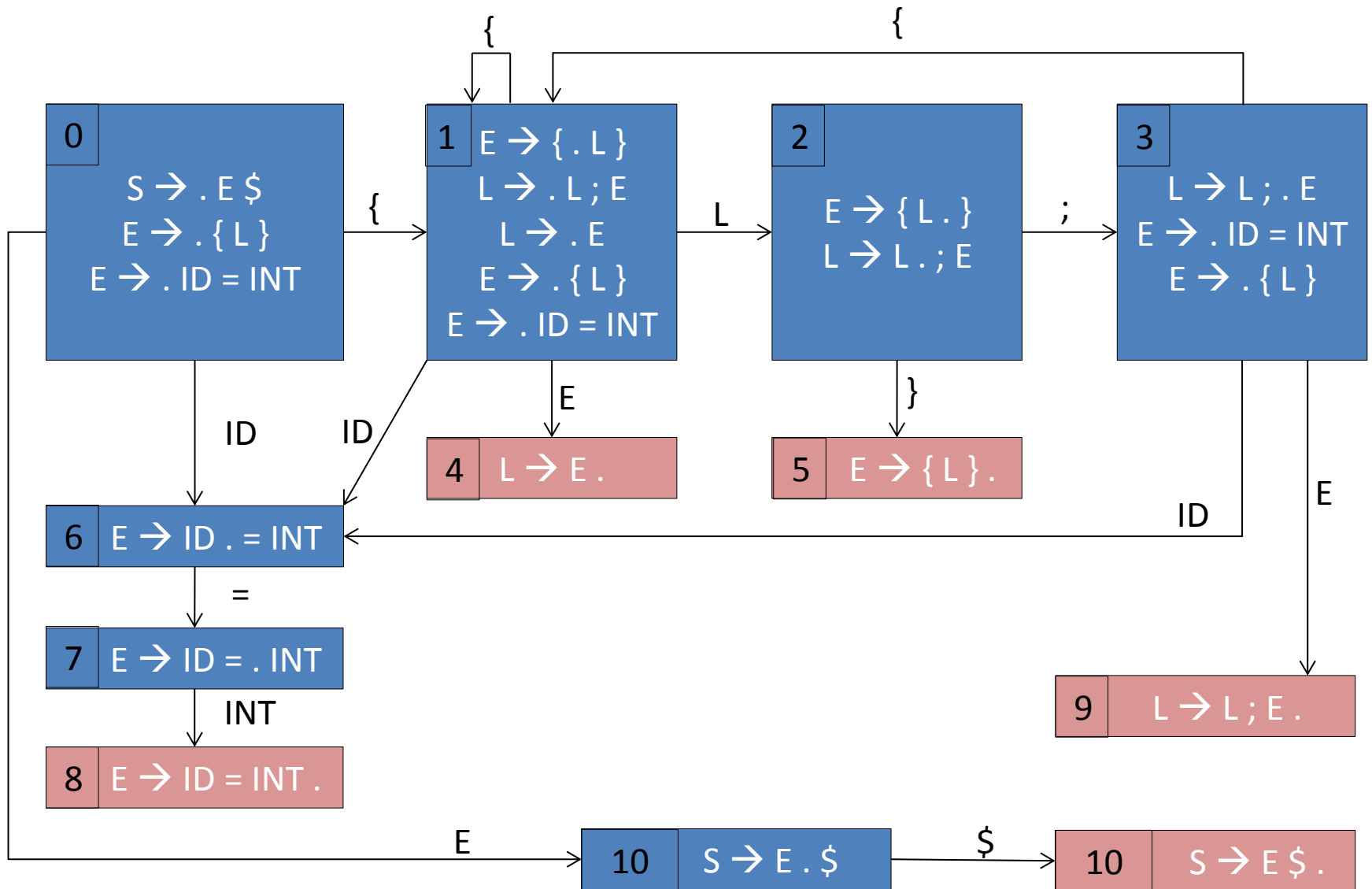
# Bottom Up Parsing

- ראינו שדקדוקים שמכילים רקורסיה שמאלית, אי אפשר לבנות עבורם predictive parser. אבל בדקדוק של שפת C שזה עתה ראינו יש רקורסיה שמאלית.
- האם אפשר לבנות parser שידע להתגבר על רקורסיה שמאלית? כן.
- נבנה את עץ הגזירה מלמטה למעלה. אם בשלב מסויים לא יודעים איזה כלל הופעל, נדחה את ההחלטה ונמשיך לקרוא את הקלט.

# Bottom Up Parsing – a simple example

- נסתכל על הדקדוק הבא שמכיל רקורסיה שמאלית:
  - $E \rightarrow ID = INT$        $E \rightarrow \{ L \}$        $S \rightarrow E\$$
  - $L \rightarrow L ; E$        $L \rightarrow E$
- ונסתכל על המילה הבאה בשפה:  $\{ x=8 ; y=7 \}$
- בקריאת התו הראשון, סוגריים מסולסלים שמאליים, אני יודע איזה כלל הופעל. אבל אחר כך, כשאני קורא  $x$  כלומר  $ID$ , אני לא יודע איזה כלל גזירה הופעל:
- אפשרות אחת:  $L \rightarrow E$  ואז  $E \rightarrow ID = INT$
- אפשרות שניה:  $L \rightarrow L ; E$  ואז  $L \rightarrow E$  ואז  $E \rightarrow ID = INT$
- כלומר, אי אפשר לנבא מראש איזה כלל הופעל. צריך לדחות את ההחלטה למועד מאוחר יותר. מה המועד? קריאת התו מהקלט.
- נדחוף את התו למחסנית ונמשיך לקרוא

# Bottom Up Parsing – a simple example



# Building a Bottom Up, LR(0) Parser

- יש ארבע אפשרויות, והן תלויות בלבד במצב בו אני נמצא
- Shift: קרא את ה token הבא בקלט, דחוף אותו למחסנית, ועבור למצב הבא לפי הדיאגרמה
- Reduce: בראש המחסנית נמצא צד ימין של כלל גזירה. החלף אותו בצד שמאל של כלל הגזירה. עבור למצב המתאים
- Accept: סיים את הריצה ודווח: הכל תקין.
- Error: סיים את הריצה ודווח: המילה אינה בשפה

# Bottom Up Parsing – another example

- ניזכר בדקדוק של שפת המחשבון:

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

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

$$E \rightarrow (E)$$

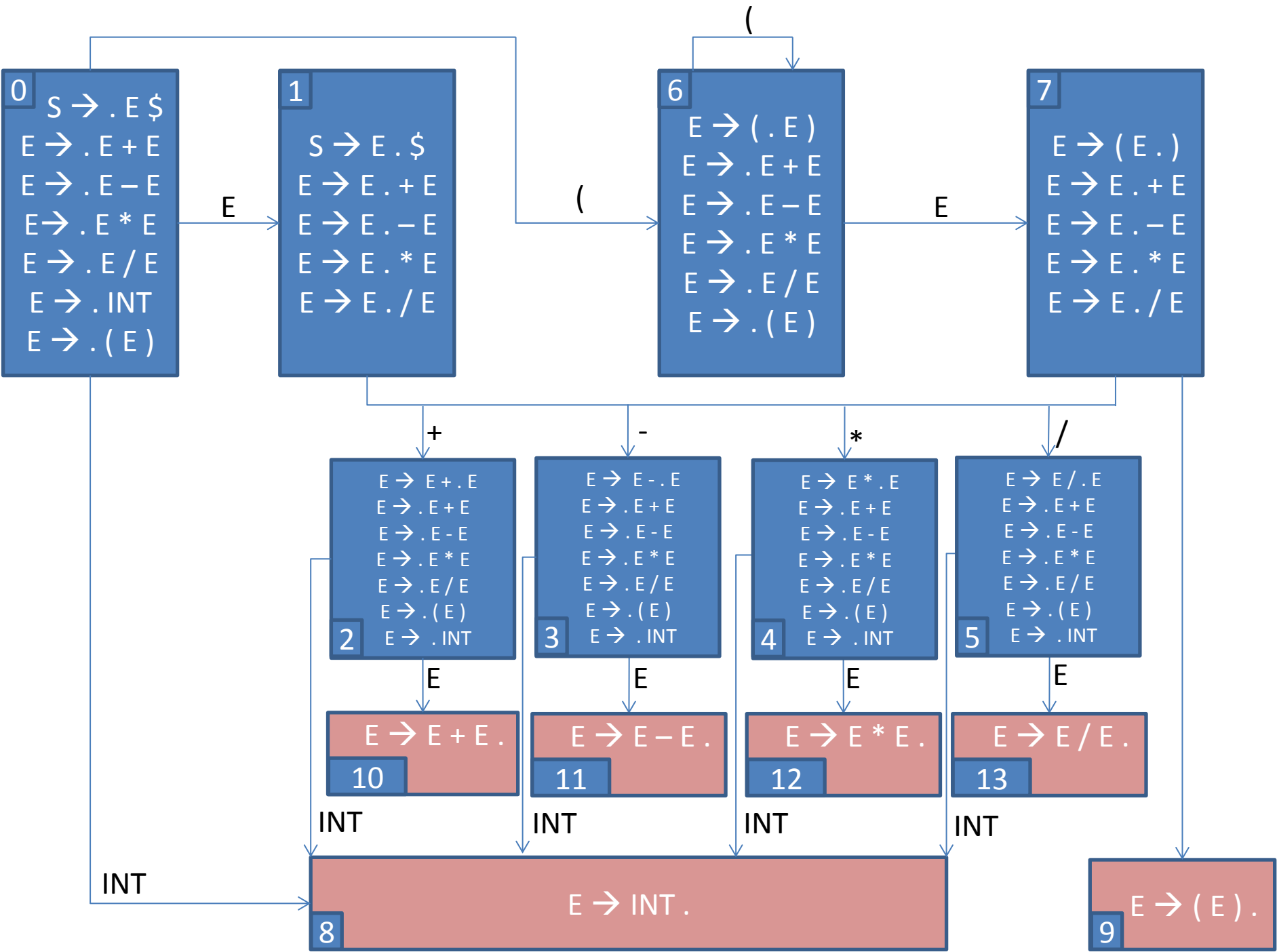
$$E \rightarrow E * E \quad \bullet$$

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

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

- האם ניתן לבנות עבורו parser כמו לדקדוק הקודם?

- מהו עץ הגזירה של הביטוי  $3+8*4$ ?





הרצה על הקלט  $1 + 2 * (500-3)$

C:\Windows\system32\cmd.exe

```
[0]
[0] INT [8]
[0] E [1]
[0] E [1] + [2]
[0] E [1] + [2] INT [8]
[0] E [1] + [2] E [10]
[0] E [1]
[0] E [1] * [4]
[0] E [1] * [4] < [6]
[0] E [1] * [4] < [6] INT [8]
[0] E [1] * [4] < [6] E [7]
[0] E [1] * [4] < [6] E [7] - [3]
[0] E [1] * [4] < [6] E [7] - [3] INT [8]
[0] E [1] * [4] < [6] E [7] - [3] E [11]
[0] E [1] * [4] < [6] E [7]
[0] E [1] * [4] < [6] E [7] > [9]
[0] E [1] * [4] E [12]
[0] E [1]
[0] E [1] $ [14]
```

Legal Expression

Press any key to continue . . . ■

# Building a Bottom Up, LR(1) Parser!

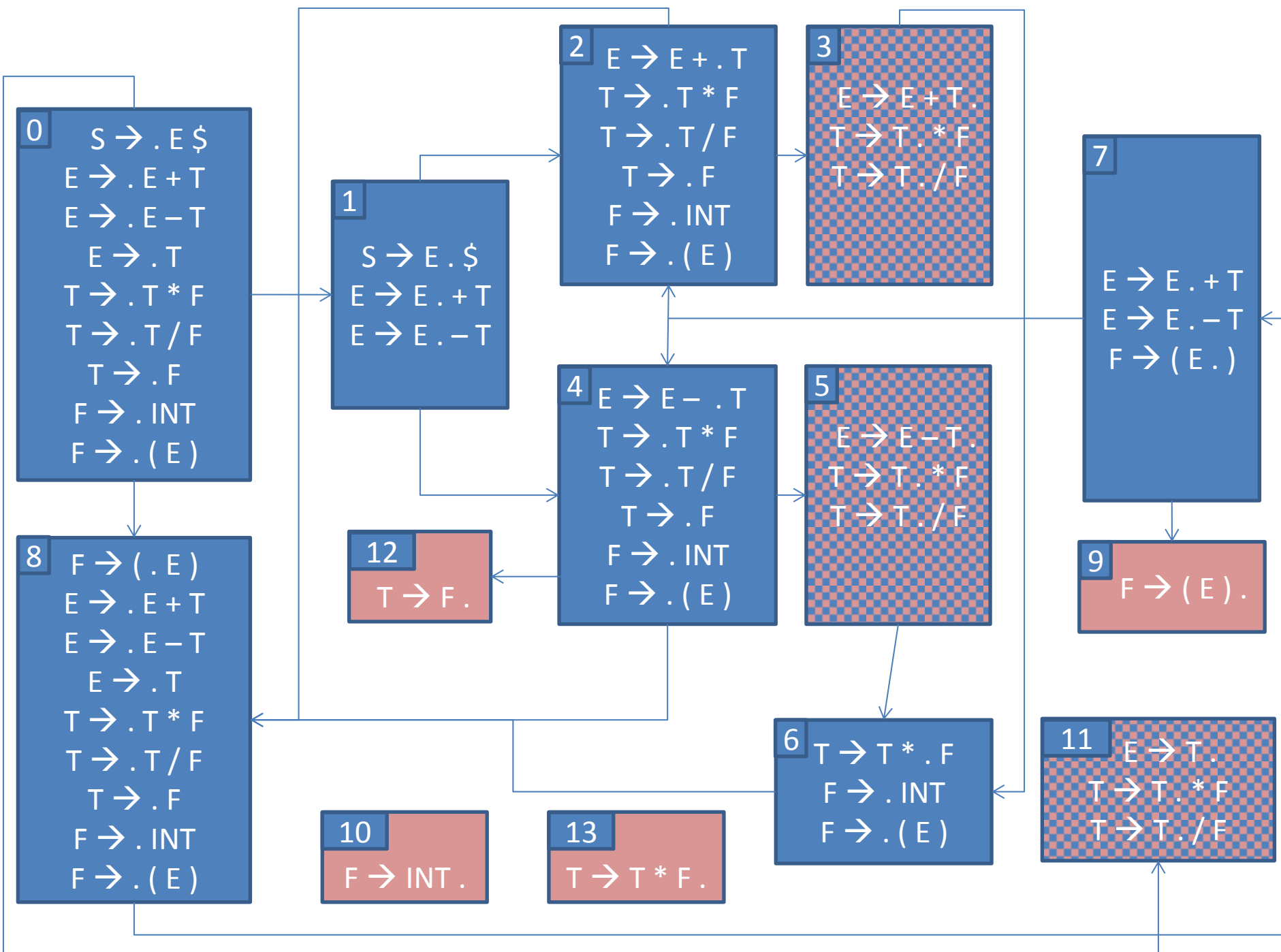
- נסתכל על בניה דומה עבור הדקדוק של שפת המחשבון עם קדימויות אופרטורים:

$F \rightarrow \text{INT}$                        $T \rightarrow T * F$                        $E \rightarrow E + T$  •

$F \rightarrow (E)$                        $T \rightarrow T / F$                        $E \rightarrow E - T$  •

$T \rightarrow F$                        $E \rightarrow T$  •

- מה ניתן לומר על מצב 5 בדיאגרמת המצבים בעמוד הבא?



# Building a Bottom Up LR(1) Parser!

- ההבדל בין LR(0) לבין LR(1) הוא בכך שב LR(1) מותר לקרוא את התו הבא בקלט, והוא יכול להשפיע אם עושים shift או reduce. ב LR(0) ההחלטה אם לעשות shift או reduce מושפעת אך ורק מהמצב הנוכחי (זה שבקצה המחסנית)
- שימו לב במימוש של LR(1) איך ממצב אחד יכולות לקרות שתי אפשרויות

**שימו לב איך קדימות האופרטורים באה לידי ביטוי:**

אם ראיתי E + T ואחריו כפל, אני לא עושה reduce אלא shift

```

C:\Windows\system32\cmd.exe

[ 0]
[ 0] INT [10]
[ 0] F [12]
[ 0] T [11]
[ 0] E [ 1]
[ 0] E [ 1] - [ 4]
[ 0] E [ 1] - [ 4] INT [10]
[ 0] E [ 1] - [ 4] F [12]
[ 0] E [ 1] - [ 4] T [ 5]
[ 0] E [ 1]
[ 0] E [ 1] + [ 2]
[ 0] E [ 1] + [ 2] INT [10]
[ 0] E [ 1] + [ 2] F [12]
[ 0] E [ 1] + [ 2] T [ 3]
[ 0] E [ 1] + [ 2] T [ 3] * [ 6]
[ 0] E [ 1] + [ 2] T [ 3] * [ 6] < [ 8]
[ 0] E [ 1] + [ 2] T [ 3] * [ 6] < [ 8] INT [10]
[ 0] E [ 1] + [ 2] T [ 3] * [ 6] < [ 8] F [12]
[ 0] E [ 1] + [ 2] T [ 3] * [ 6] < [ 8] T [11]
[ 0] E [ 1] + [ 2] T [ 3] * [ 6] < [ 8] E [ 7]
[ 0] E [ 1] + [ 2] T [ 3] * [ 6] < [ 8] E [ 7] - [ 4]
[ 0] E [ 1] + [ 2] T [ 3] * [ 6] < [ 8] E [ 7] - [ 4] INT [10]
[ 0] E [ 1] + [ 2] T [ 3] * [ 6] < [ 8] E [ 7] - [ 4] F [12]
[ 0] E [ 1] + [ 2] T [ 3] * [ 6] < [ 8] E [ 7] - [ 4] T [ 5]
[ 0] E [ 1] + [ 2] T [ 3] * [ 6] < [ 8] E [ 7]
[ 0] E [ 1] + [ 2] T [ 3] * [ 6] < [ 8] E [ 7] > [ 9]
[ 0] E [ 1] + [ 2] T [ 3] * [ 6] F [13]
[ 0] E [ 1]
[ 0] E [ 1]

Legal Expression
Press any key to continue . . . █

```