

## קומפילציה

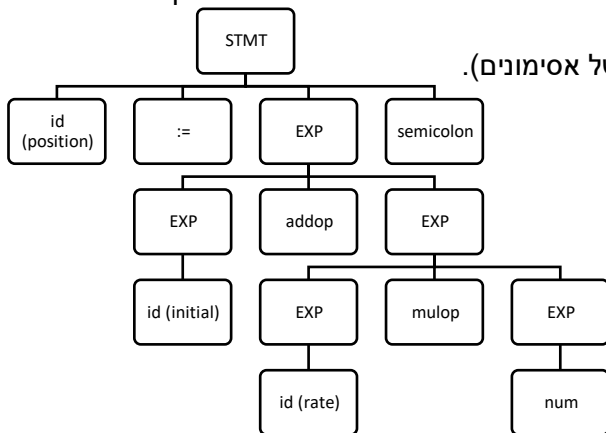
### שלבי קומפילציה

- **מנתח לקסיקלי (lexer – lexical analyzer)** – מפריד את הקלט ליחידות בסיסיות הנקראות אסימונים – tokens.

לדוגמא: **position := initial + rate \* 60;**

- position הוא האסימון, הערך הסמנטי שלו (לקסיקלי) הוא השם שלו – position וסוג האסימון (המזהה) הוא ID.
- האסימון הבא יהיה ':=', לאסימון זה אין ערך סמנטי.
- initial הוא אסימון מסוג ID.
- '+' הוא אסימון מסוג ADDOP.
- rate הוא אסימון מסוג ID.
- '\*' הוא אסימון מסוג MULOP.
- '60' הוא אסימון מסוג NUM.
- ';' הוא אסימון מסוג SEMICOLON.

מכך נובע שהביטוי **position := initial + rate \* 60** מפורק ע"י ה-lexer ל – **ID := ID ADDOP ID MULOP NUM SEMICOLON**. האסימונים הם בעצם ה-terminals של שפת התכנות, כמו כן מילים שמורה של אותה השפה נכנסים גם תחת הקטגוריה הזו.



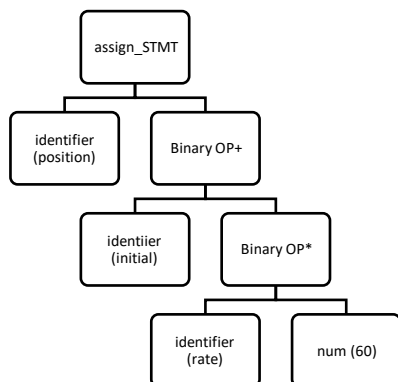
- **Parser:** בונה עץ גזירה (parse tree), עבור קלט נתון (הקלט הוא סדרה של אסימונים).

דוגמאות לחוקי גזירה בשפת תכנות:

- $STMT \rightarrow while (EXP) STMT$
- $STMT \rightarrow EXP semicolon$
- $STMT \rightarrow if (EXP) STMT else STMT$
- $STMT \rightarrow id := EXP semicolon$
- $EXP \rightarrow EXP addop$
- $EXP \rightarrow num$
- $EXP \rightarrow id$

- מתייחסים לאופרטור החלש יותר (במקרה הזה הוא '+'), הוא האופרטור הראשי של הביטוי. כלומר הוא מחלק את הביטוי שמיימין להשמה ל-2 חלקים: (EXP mulop EXP) **addop** (EXP).

התוצר של ה-parsing הוא לעיתים קרובות AST (abstract syntax tree), לא כל קומפיילר מייצר את המבנה נתונים הזה. במקרה הזה ה-AST הוא עץ השמה. דוגמא:



- **טבלת סמלים (symbol table)** – הקומפיילר שומר את המשתנים של התוכנית בטבלת משתנים. כאשר בטבלה הזו שמור מידע נוסף על כל משתנה (טיפוס, static, final ועוד..). את הערך ל המשתנה לא ומרים בטבלה, היות והוא יכול להשתנות. לעומת זאת ה- interpreter כן שומר את ערכי המשתנים בטבלה.

- **מנתח סמנטי** – בודק את תקינות המשמעויות של הביטויים לפי כללי שפת התכנות.

למשל: "האם כל משתנה שמשתמשים בו הוגדר בדיוק פעם אחת" – זוהי בדיקה סמנטית היות ואם אותו משתנה יוגדר יותר מפעם אחת זאת טעות סמנטית (במשמעות הביטוי) ולא טעות דקדוקית. ככלל כל טעות שמפרה את כללי שפת התכנות, שהיא לא טעות

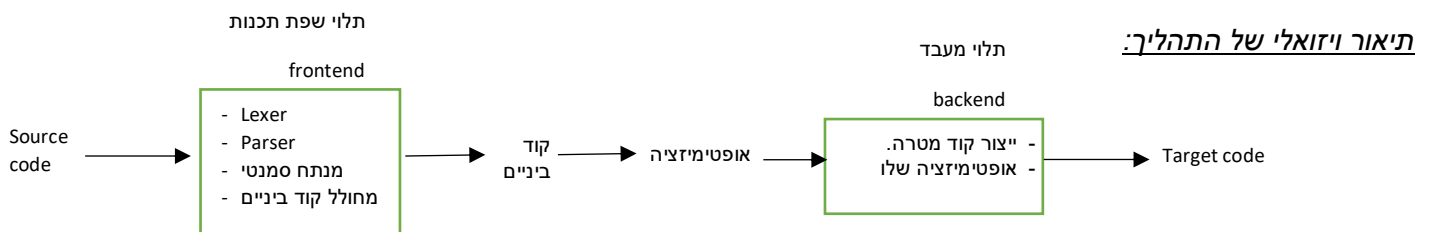
## קומפילציה

דקדוקית, זאת טעות סמנטית. ( $\text{int } l = \text{"hello"}$  – טעות סמנטית, כל כללי הדקדוק כאן תקינים אך המשמעות לא). חלק מהבדיקות שנעשות בשלב זה, הן בדיקות typechecker – בדיקות טיפוסים המשתנים בהתאם לביטוי בו הם נמצאים. **ייצור קוד ביניים** – intermediate code IR (intermediate representation). ייצור קוד פשוט יותר מקוד המקור (הקוד בשפת התכנות). קוד ביניים צריך להיות בעל תכונה – three address code, כלומר שיש לכל היותר אופרטור אחד בנוסף לאופרטור ההשמה (ביטוי פשוט). כמו  $a = b + c$ .

למה זה טוב?

- מודולריות – חלוקה ל – frontend / backend.
- מאפשר הרצה של אלגוריתמי אופטימיזציה על הקוד.
- מקל על התרגום לשפת אסמבלי.

לאחר השלב הזה מתבצעת אופטימיזציה של קוד הביניים (יכולות להיות שורות קוד שהמיקום שלהם ביחס לקוד מקורה יהיה שונה, על מנת לייעל את הריצה), כדי שהתוכנית תרוץ בצורה יעילה יותר. לאחר מכן מייצר את קוד המטרה (בשפת כונה שאפשר להריץ על המעבד) ומתבצעת אופטימיזציה שלו.



- נניח שיש ברשותינו קומפיילר :  $c \rightarrow \text{mips}$  ואנחנו מעוניינים בקומפיילר :  $\text{java} \rightarrow \text{mips}$ . לשם כך יש לשכתב רק את ה-frontend.

## Flex – כלי לייצור מנתח לקסיקלי

אנחנו רוצים להגדיר ש-175 יהיה בעל מזהה NUM, bar יהיה בעל מזהה ID, "hello" יהיה בעל מזהה STRING. נכין קובץ mylex.lex שיכיל את הפונקציה שתדפיס את נתונים בהתאם למזהה שלהם (לצורך העניין פונקציית ה-main). נכתוב את הפקודות הבאות ב-cmd:

1. `Flex.exe mylex.lex`  
נוצר קובץ `lexyy.c` הכולל את הפונ' `yylex`.
2. נקמפל את הקובץ `lexyy.c` בעזרת קומפיילר של שפת C – `gcc`, ויתקבל קובץ הרצה – `myprog.exe`.
3. `Myprog.exe mytest.txt` – הקובץ `mytest.txt` הוא קובץ הקלט אשר מכיל את השורה: `175 bar "hello"`.  
לן יודפס למסך:  
NUM: 175  
ID: bar  
String: "hello"

## ממשק ל-yylex

Yylex מחזירה מספר המייצר את סוג האסימון הנוכחי בקלט. את הערך הסמנטי של האסימון היא כותבת למשתנה הגלובלי `yylval` (משתנה של `bison`).

- Yylex – תמיד מחפשת את הרישא הארוכה ביותר של יתרת הקלט המתאימה לאחד הביטויים הרגולריים. במקרה שהרישא מתאימה ליותר מביטוי רגולרי אחד, היא תבחר את הביטוי שרשום קודם. במקרה ששני ביטויים רגולריים מתאימים לאותה הרישא

## קומפילציה

- הארוכה ביותר, אז יתבצע הביטוי הרגולרי שרשום קודם בפונ' yylex, וה-action שלו יתבצע. מהסיבה הזו הביטוי הרגולרי בשורה 39 בקוד הוא האחרון, כי נקודה מתאימה לכל תו חוץ מירידת שורה.
- %% - תוחם קטע קוד עבור ה-flex, שמגדיר את הדקדוק (rules). החוקים בדקדוק נמצאים בפונקציה yylex. במקרה הזה הפרדה בין סוגי החוקים מתבצעת ע"י רווח, כלומר bar17 יחשב כ-ID, למרות שיש בו מספר.
  - קטע קוד ב-C שכתוב אחרי הבלוק של החוקים, יוצר בקובץ lex.yy.c אחרי הפונ' yylex. כמו כן קטע קוד שכתוב לפני יוצב לפני.
  - Yywrap – פונ' שעל המתכנת לכתוב כי להורות ל-flex שיש להמשיך לקרוא קלט ממקום אחר. הקיראה לפונקציה זו מתבצעם בסוף פונקציית yylex.
  - Yytext – משתנה של yylex אשר מכיל את הקלט הנוכחי, ולעומתו בודקים את הביטויים הרגולריים.
  - Yyin – קובץ הקלט של yylex, וקובץ הפלט הוא yyout בתור ברירת מחדל.

קטע קוד:

```

1  %{
2
3  #define NUM 300
4  #define ID 301
5  #define STRING 302
6
7  union {
8      int ival;
9      char name[30];
10     char str[30];
11 } yyval;
12
13 #include <string.h>
14
15 extern int atoi (const char *);
16 %}
17
18 %option noyywrap
19
20 /* exclusive start condition -- deals with C++ style comments */
21 %x COMMENT
22
23 %%
24 [0-9]+ { yyval.ival = atoi (yytext); return NUM; }
25
26 [a-zA-Z][a-zA-Z0-9]* { strcpy (yyval.name, yytext); return ID; }
27
28 \"([^\"]|\\\"|\\n)*\" { strcpy (yyval.str, yytext); return STRING; }
29
30 [\\n\\t ]+ /* skip white space */
31
32 \"/\" { BEGIN (COMMENT); }
33
34 <COMMENT>.+ /* skip comment */
35 <COMMENT>\\n { /* end of comment --> resume normal processing */
36     BEGIN (); }
37
38 . { fprintf (stderr, "unrecognized token %c\\n", yytext[0]); }
39
40 %%
41
42
43 main (int argc, char **argv)
44 {
45     int token;
46
47     if (argc != 2) {
48         fprintf(stderr, "Usage: mylex <input file name>\\n", argv [0]);
49         exit (1);
50     }
51
52     yyin = fopen (argv[1], "r");
53
54     while ((token = yylex ()) != 0)
55     switch (token) {
56     case NUM: printf("NUMBER : %d\\n", yyval.ival);
57                 break;
58     case ID:  printf ("ID : %s\\n", yyval.name);
59                 break;
60     case STRING: printf ("STRING: %s\\n", yyval.str);
61                 break;
62     default:  fprintf (stderr, "error ... \\n"); exit (1);
63     }
64     fclose (yyin);
65     exit (0);
66 }
67

```

*Handwritten notes in Hebrew:*

- Line 18: *קטע קוד ב-C*
- Line 19: *קטע קוד לפני*
- Line 21: *הוא אחרי הפונ' של yylex, הממשיך לקרוא את הקלט*
- Line 23: *הוא אחרי הפונ' של yylex, הממשיך לקרוא את הקלט*
- Line 24: *מספר*
- Line 25: *מספר*
- Line 26: *מספר*
- Line 27: *מספר*
- Line 28: *מספר*
- Line 29: *מספר*
- Line 30: *מספר*
- Line 31: *מספר*
- Line 32: *מספר*
- Line 33: *מספר*
- Line 34: *מספר*
- Line 35: *מספר*
- Line 36: *מספר*
- Line 37: *מספר*
- Line 38: *מספר*
- Line 39: *מספר*
- Line 40: *מספר*
- Line 41: *מספר*
- Line 42: *מספר*
- Line 43: *מספר*
- Line 44: *מספר*
- Line 45: *מספר*
- Line 46: *מספר*
- Line 47: *מספר*
- Line 48: *מספר*
- Line 49: *מספר*
- Line 50: *מספר*
- Line 51: *מספר*
- Line 52: *מספר*
- Line 53: *מספר*
- Line 54: *מספר*
- Line 55: *מספר*
- Line 56: *מספר*
- Line 57: *מספר*
- Line 58: *מספר*
- Line 59: *מספר*
- Line 60: *מספר*
- Line 61: *מספר*
- Line 62: *מספר*
- Line 63: *מספר*
- Line 64: *מספר*
- Line 65: *מספר*
- Line 66: *מספר*
- Line 67: *מספר*

## קומפילציה

- כלל ברירת מחדל לביטויים הרגולריים:  $\{ \backslash n \}$ . כלל זה לא צריך לכתוב, הוא מתקיים לבד במקרה ואף ביטוי רגולרי לא מתאים לקלט.

### Start conditions

הרעיון פה הוא להפריד את הביטויים הרגולריים לסטים, כדי שנוכל לשלוט באילו ביטויים רגולריים להשתמש במקרים שונים. ניתן לראות בקוד שהחל משורה 33, רק הביטויים הרגולריים של SC comment פעילים, כלומר שאר הביטויים הרגולריים לא ישפיעו על העבודה.

לדוגמא:

```
a?b
<foo>[0-9]+
<foo>abc*
<bar>aaaa
<bar>.*
```

בכל רגע נתון yylex נמצאת ב- start condition מסוים, כל מה שנמצא בתגיות מציין את ה- start condition, ומה שמימין לתגיות אלו הם הביטויים הרגולריים.

כאשר לא מצוין start condition מסוים, yylex נמצאת ב- <initial> : start condition, בדוגמא הנ"ל a?b זה ה- initial. כדי לעבור לביטויים הרגולריים של initial ניתן לכתוב BEGIN(initial) או BEGIN(0).

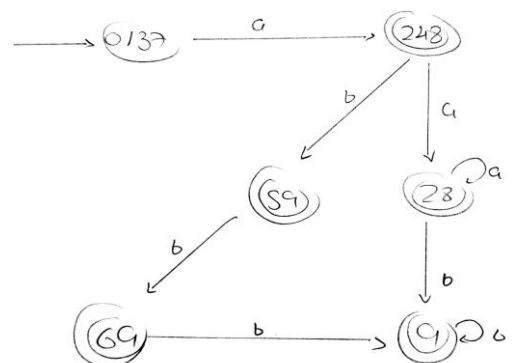
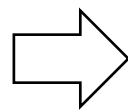
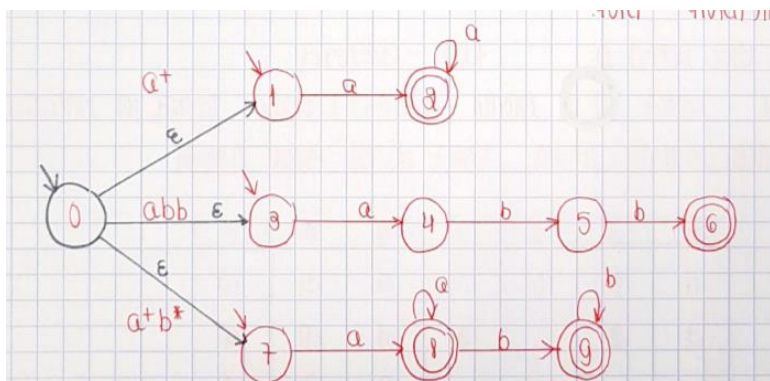
כדי לעבור ל- start condition כלשהו כותבים begin(foo).

- כשנמצאים ב- exclusive start condition (מוכרז עם %), אז רק הכללים שלו פעילים. בקוד בשורה 21 אפשר לראות שמוגדר SC עבור הערות – comment. כשנמצאים ב- non-exclusive start condition, אז פעילים הכללים שלו וגם הכללים של initial. את ה- SC האלו ניתן להגדיר בעזרת %S. ניתן לראות בקוד שבשורה 33 עוברים ל- BEGIN(COMMENT) ובשורות 35,36 משתמשים ב- SC הזה. בקובץ הקלט מה שיגרום לקוד לעבור ל- SC של comment זה \\\ שמוגדר כביטוי רגולרי של \".

### דוגמא להרצת אלגוריתם של כלי דמוי flex

- המר את הביטויים הרגולריים לאוטומט סופי לא דטרמיניסטי.
- מזל את האוטומטים לאוטומט אחד גדול
- המר את האוטומט הגדול לאוטומט סופי דטרמיניסטי. (yylex מסלמעת את האוטומט).

דוגמא: אוטומט לא דטרמיניסטי  $\rightarrow$  אוטומט דטרמיניסטי



## קומפילציה

עבור המחרוזת  $abbbbaa$ , ההתאמה הכי ארוכה תהיה עבור האוטומט התחתון של  $a+b^*$ , למרות שתהיה התאמה גם לשאר האוטומטים אבל ההתאמות יהיו קצרות יותר. בעצם כשהאוטומט יגיע למצב מת הוא ידע שהוא מצא את התאמה הכי ארוכה במצב שממנו הוא הגיע למצב המת. המחרוזת הנ"ל אחרי ה-b האחרון האוטומט יגיע למצב מת ממצב 9 (של DFSM) וכך הוא יודע שנמצאה ההתאמה הכי ארוכה.

### פוקנציית First

- 1.  $S \rightarrow ABCd$
- 2.  $A \rightarrow a1b2$
- 3.  $A \rightarrow a2H$
- 4.  $A \rightarrow \epsilon$
- 5.  $B \rightarrow b1h$
- 6.  $B \rightarrow b2DH$
- 7.  $B \rightarrow GH$
- 8.  $C \rightarrow c$
- 9.  $C \rightarrow \epsilon$
- 10.  $D \rightarrow d$
- 11.  $G \rightarrow g$
- 12.  $G \rightarrow \epsilon$
- 13.  $H \rightarrow h$
- 14.  $H \rightarrow \epsilon$

### מוסכמות:

- מחרוזות של סימני דקדוק (משתנים/טרמינלים)  $\alpha, \beta, \gamma$
- משתנים (nonterminals)  $B, A$
- טרמינלים  $a, b, c$
- מחרוזות של טרמינלים  $x, y, z$
- המילה הריקה זו מחרוזת עם 0 סימנים, זה לא טרמינל.

1.  $FIRST(\alpha)$  – זו קבוצה של טרמינלים\מילה ריקה. קבוצה זו מכילה את כל הטרמינלים שהם נמצאים במקום הראשון של הצד הימני בחוקים.  
טרמינל  $a$  שייך ל-  $FIRST(\alpha)$ , אם קיימת גזירה (קיימת מחרוזת  $\beta$ ) כך ש:  $\alpha \Rightarrow^* a\beta$
2. אם  $\epsilon \Rightarrow^* \alpha$  (אפיס  $\alpha$ ) אז  $\epsilon$  שייכת ל-  $FIRST(\alpha)$
3. אם  $a$  טרמינל אז נגזור:  $FIRST(a) = \{a\}$
4.  $First$  של טרמינל הוא אותו הטרמינל –  $FIRST(a) = a$

דוגמא:

$$FIRST(ABCd) = \{a1, a2, b1\}$$

$$ABCd \rightarrow \underline{a1}b2BCd$$

$$ABCd \rightarrow \underline{a2}HBCd$$

$$ABCD \rightarrow BCd \rightarrow \underline{b1}hCd$$

## קומפילציה

### הרצת אלגוריתם לחשוב FIRST של כל משתני הדקדוק

נעבור על חוקי הגזירה ונבדוק מהם ערכי ה-FIRST של ה-nonterminals.

S	A	B	C	D	G	H
a1	a1	b1	c	d	g	h
a2	a2	b2	ε		ε	ε
b1	ε	g				
b2		h				
g		ε				
h						
c						
d						

### פונקציית FOLLOW

FOLLOW(A) – קבוצה של טרמינלים \ \$ - סוף. הוא לעולם לא כולל את ε, כיוון שאם היה מכיל זה אומר שכל nonterminal גוזר אפסילון.

1. טרמינל a שייך ל- FOLLOW(A) אם קיימת גזירה :  $S \Rightarrow^* \alpha A a \beta$ , הנקודה החשובה היא ש-a מגיע ישר אחרי A
2. \$ שייך ל- FOLLOW(A) אם קיימת גזירה :  $S \Rightarrow^* \alpha A$ , כלומר אין כלום אחרי A. בנוסף \$ הוא תמיד ב- FOLLOW(S) של המשתנה ההתחלתי, כיוון שאפשר לגזור ב-0 צעדים מ-S ל-S ואז ה-\$ הוא אחרי ה-S.

דוגמא:

1.  $S \rightarrow DzDaG$       2.  $D \rightarrow ABG$       3.  $A \rightarrow Dh$       4.  $A \rightarrow a$       5.  $B \rightarrow b1h$       6.  $B \rightarrow b2$
7.  $B \rightarrow \epsilon$       8.  $D \rightarrow d$       9.  $G \rightarrow g$       10.  $G \rightarrow B$

$$\text{FOLLOW}(D) = \{z, a\}$$

$$\text{FOLLOW}(G) = \{\$, z\}$$

$$S \rightarrow DzDaG \rightarrow ABGzDaG$$

### אלגוריתם לחישוב FOLLOW של משתני הדקדוק

S	A	B	D	G
\$	b1	b1	z	z
	b2	b2	g	a
	g	g	h	\$
	z	z		h
	a	a		
	h	h		
		\$		

$$\text{FIRST}(S) = \{a, d\}$$

$$\text{FIRST}(A) = \{a, d\}$$

$$\text{FIRST}(B) = \{b1, b2, \text{epsilon}\}$$

$$\text{FIRST}(D) = \{a, d\}$$

$$\text{FIRST}(G) = \{b1, b2, \text{gepsilon}\}$$

כדי לחשב את ה-FOLLOW, נעבוד לפי חישובי ה-FIRST של ה-nonterminals.

- מכלל גזירה מהצורה :  $A \rightarrow \alpha B \beta$ , נסיק שכל הטרמינלים מ-  $\text{FIRST}(\beta)$  שייכים ל-  $\text{FOLLOW}(B)$ .
- מכלל גזירה מהצורה :  $A \rightarrow \alpha B$  או  $A \rightarrow \alpha_1 B \alpha_2$ , כאשר  $\alpha_2$  אפס, נסיק שכל איברי  $\text{FOLLOW}(A)$  שייכים ל-  $\text{FOLLOW}(B)$ .

מכלל 2, כל הטרמינלים של  $\text{FIRST}(G)$ ,  $\text{FIRST}(B)$  נמצאים ב-  $\text{FOLLOW}(A)$  כיוון ש-B אפס, אז צריך להתייחס גם לטרמינלים של G. בנוסף מכלל זה, כדי למצוא את ה-  $\text{FOLLOW}(A)$  נחפש את  $\text{FIRST}(B)$  וגם  $\text{FIRST}(G)$ .

## קומפילציה

### Top down parsing

לאחר המנתח הלקסיקלי מגיע שלב ה-parsing, שהתוצר שלו הוא עץ גזירה לקלט נתון.

- **Statement** – משפט שמבצעים אותו, אך הוא לא מחזיר ערך. למשפט יש side effects, השמה, IF, לולאות.
- **Expression** – ביטוי שמחזיר ערך (כמו חישוב פעולות מתמטיות).
- **רקורסיה ימנית** – כאשר ה-nonterminal של הכלל נמצא בצד הימני ביותר של כלל הגזירה:  $A \rightarrow \alpha A, A \rightarrow^* \alpha A$ . אותו דבר גם לגבי רקורסיה שמאלית.

**בניית טבלת עבור predictive parsing לא רקורסיבי**

**קלט: דקדוק G**

**פלט: טבלת ניתוח** תחבירי M שבה יש שורה עבור כל משתנה של G ועמודה עבור כל טרמינל של G ועבור הסימן \$ המיצג את סוף הקלט. כל כניסה בטבלה יכולה להיות ריקה או לכלי נזירה (רצוי לא יותר מאחד – ראו בהמשך) או ציון של שניאה (error).

**סימון:** M[A, a] היא הכניסה שמופיעה בשורה של המשתנה A ועמודה a.

**האלגוריתם:**

1. עבור כל כלל נזירה (של הדקדוק)  $A \rightarrow \alpha$  בצע את צעדים 2 ו-3
2. עבור כל טרמינל a שנמצא ב- FIRST( $\alpha$ ), הוסף את  $A \rightarrow \alpha$  לכניסה M[A, a]
3. אם  $\epsilon$  נמצא ב- FIRST( $\alpha$ ) או עבור כל טרמינל b שנמצא ב- FOLLOW(A), הוסף את  $A \rightarrow \alpha$  לכניסה M[A, b]. אם  $\epsilon$  נמצא ב- FIRST( $\alpha$ ) ו-S נמצא ב- FOLLOW(A), הוסף את  $A \rightarrow \alpha$  לכניסה M[A, \$].
4. כל כניסה של M שנותנה ריקה נחשבת לכניסה error.
5. אם יש כניסה אחת (או יותר) ב-M שמכילה יותר מכלל נזירה אחד או האלגוריתם נכשל אחרת הוא מסתיים בהצלחה.

**הערות:** זה לא טוב כאשר כניסה בטבלה מכילה יותר מכלל נזירה אחד נותן קרא קונפליקט כי כאשר המנתח נוקט לכניסה זו, אין הוא יודע באיזה מהכללים הרשומים בה להשתמש.

**דוגמה**

הרצת האלגוריתם לבניית טבלה על הדקדוק שהובא לעיל תתן את הטבלה הבאה. המספרים בטבלה מציינים כללי נזירה. הכניסות הריקות מציינות error.

- כדי לסמן את סוף הקלט במחסנית, תמיד שמים \$ לפני שדוחפים לתוכה את הקלט. כל פעם שבמחסנית מגיעים לאסימון (כלומר האסימון – טרמינל, הוא בראש המחסנית), משווים את האסימון למחרוזת/ תו הקלט שמסומן ע"י ה-lookahead.
- אם בטבלה אין גזירה למחרוזת כלשהי מכלל כלשהו, זה יתן syntax error אם הקומפילר יתקל במקרה כזה. כלומר אם בראש המחסנית יעמוד הכלל הנ"ל והמחרוזת הנוכחית תהיה המחרוזת הנ"ל תתקבל שגיאה.

**מתי נוצר קונפליקט בין כללי גזירה,  $A \rightarrow^* \alpha, A \rightarrow^* \beta$**

1.  $FIRST(\alpha) = FIRST(\beta)$  – קיים טרמינל שגוזר מאלפא ומביטא מחרוזות המתחילות מאותו טרמינל:  $FIRST(\alpha) \cap FIRST(\beta)$ .
2. המילה הריקה – אפסילון, שייכת ל-  $FIRST(\alpha) \cap FIRST(\beta)$ , כלומר גם אלפא וגם ביטא אפסיות.
3. ביטא אפסיה וקיים טרמינל  $FOLLOW(A) \cap FIRST(\alpha)$ .

לדוגמא:

$S \rightarrow Az$

$A \rightarrow \epsilon$

$A \rightarrow z$

**דקדוק עם רקורסיה שמאלית אינו LL(1)**

דקדוק כזה יוצר קונפליקט בטבלה. היות וזאת רקורסיה מיידית, ובעץ גזירה, כדי להגיע לטרמינל הרצוי, תמיד תהיה האלטרנטיבה לעבור דרך אותו ה-nonterminal שוב.

לדוגמא:

$S \rightarrow Az$

$A \rightarrow Aa$

$A \rightarrow a$

## קומפילציה

### בניית טבלה לדקדוק LL(2)

בדקדוק  $LL(k)$  ה- $parser$  רואה  $k$  תווים קדימה ממיקום ה- $lookahead$ , לכן בטבלה יהיו עמודות שבהגדרתם יהיו  $k$  תווים, ובהתאם נצטרך לחשב  $FIRST_k$  וגם  $FOLLOW_k$ .

בדקדוק  $LL(2)$  מסתכלים 2 תווים קדימה, לכן כל עמודה בטבלה תכיל הגדרה של 2 תווים, כלומר אם רואים את 2 התווים שבראש העמודה, באיזה כלל נשתמש לגזירה כדי להגיע למחרוזת בת 2 התווים הזאת. כמו כן בגלל זה נדרש לחשב את  $FIRST_2$  של צד ימין של חוקי הגזירה, נרצה לדעת איזה 2 תווים יהיו הראשונים בכל צד ימין של כלל. דקדוק  $LL(k+1)$  מכיל את דקדוק  $LL(k)$  ולכן יכול לגזור גם מחרוזות באורך עד  $k+1$  (כמובן שאם קיימים כללי גזירה מתאימים בדקדוק). במצב של non-terminal אפיס, נצטרך לחשב את  $FOLLOW_2$ .

### Bottom up parsing

#### LR parsing

עדיף להשתמש ברקורסיה שמאלת.

- Shift: דחוף את ה- $lookahead$  (התו הנוכחי בקלט) למחסנית והתקדם בקלט. (קרא למנתח הלקסיקלי ועדכן את ה- $lookahead$ ).
- Reduce: החלף את מה שבראש המחסנית (יכול להיות גם רצף מחרוזות שתואם את הכלל גזירה) ב- $non-terminal$  של הכלל.

לדוגמא קיים הכלל:  $B \rightarrow id\ relop\ num$ .

אז כל מחרוזת נדחפת למחסנית בתורה (פעולת shift), כשבמחסנית יהיו שלושת המחרוזות, הן תוצאנה ויכנס במקומן  $B$  (פעולת reduce). וכך בעצם נבנה הצד הימני של הכלל של  $S$ , עד שניתן לרוקן את המחסנית ולהכניס את  $S$ , מה שיציין שהקלט תקין והגענו לסופו.

פעולת ה- $reduce$  מתבצעת לפי ה- $FOLLOW$  של ה- $non-terminal$ , כלומר כשה- $lookahead$  מצא מחרוזת שהיא ב- $FOLLOW$  של אותו ה- $non-terminal$ , תתבצע פעולת ה- $reduce$  לפי אורך כלל הגזירה המתאים.

בפועל במחסנית של העץ גזירה יהיו רק המצבים של טבלת ה- $SLR(1)$ , וצמצום המחרוזות נעשה ע"י כך שבטבלה יש כמות מצבים לפי כמות המחרוזות שנקראו כביכול למחסנית, רק שבמקומם יש מספרים. כניסה ריקה בטבלה מעידה על  $syntax\ error$ .

### בניית טבלת $SLR(1)$

1. מרחיבים את הדקדוק ע"י הוספת כלל גזירה  $S' \rightarrow S$  (משתנה ההתחלתי החדש), כך שעכשיו  $S'$  יהיה השורש של עץ הגזירה של הדקדוק, אשר יסיים את סוף ה- $parsing$ . עושים זאת כיוון שהמשתנה ההתחלתי  $S$  יכול להופיע כמה פעמים בעץ ולא ניתן להתבסס על כך שהגענו ל- $S$  שזהו סוף הקלט.

2. בונים אוטומט פריטי  $LR(0)$ . פריטים אלו הם כללי גזירה בהם יש נקודה בצד ימין של הכלל. הנקודה מציינת את התקדמות ה- $parser$ , מה שלפני הנקודה נקרא כבר ונמצא במחסנית. אפשר ליצור קבוצות של כללים לפי מיקום הנקודה, כלומר ה- $parser$  יעבוד במקביל על הכללי גזירה באותה הקבוצה, וזה אומר שיש כמה דרכים להמשיך מהמקום בו נמצא ה- $lookahead$ . האוטומט מקבל את כל המחרוזות שעשויות להיות על המחסנית במהלך הניתוח התחבירי של קלט תקין. המצב ההתחלתי של האוטומט יהיה הסגור של קבוצת הפריטים, כלומר כל כללי הגזירה של  $S$  ואם בצד הימני של כלל גזירה כלשהו יש עוד  $non-terminal$  אז גם כללי הגזירה שלו וכך הלאה.

חישוב המעברים של האוטומט מתבצע כך שמזיזים את הנקודה בעצם ימינה לאחר כל מחרוזת בצד הימני של כלל הגזירה, ובמצב הבא מחשבים את הסגור בהתאם לחוקים. והמעבר הוא בעצם אותה המחרוזת שהייתה מימין לנקודה במצב הקודם.

כשהנקודה נמצאת בסוף המחרוזת במצב מסוים זה אומר שהגענו לסוף צד ימין של כלל כלשהו וצריך לעשות  $reduce$ .  $A \rightarrow \alpha$ ,  $A \rightarrow \circ$  אומר שיש לעשות  $reduce$  לפי הכלל  $A \rightarrow \alpha$  בכל העמודות שמתאימות לאיברי  $FOLLOW(A)$ . פריט שבו יש  $non-terminal$  מימין לנקודה מתפרש בעמודות של  $goto$  בטבלה.

3. בונים את הטבלה לפי האוטומט.



### סגור של קבוצת פריטים

מתחילים מהכלל הראשון לאחר הוספת  $S'$ , ומוסיפים נקודה משמאל ל- $S \leftarrow S' \rightarrow$ . לאחר מכן כותבים את כל כללי הגזירה של  $S$  ושמים את הנקודה בצד השמאלי ביותר של צד ימין של הכלל. כל non-terminal שצמוד לנקודה, צריך להוסיף את כללי הגזירה שלו לקבוצה של סגור הקבוצות, וכך הלאה.

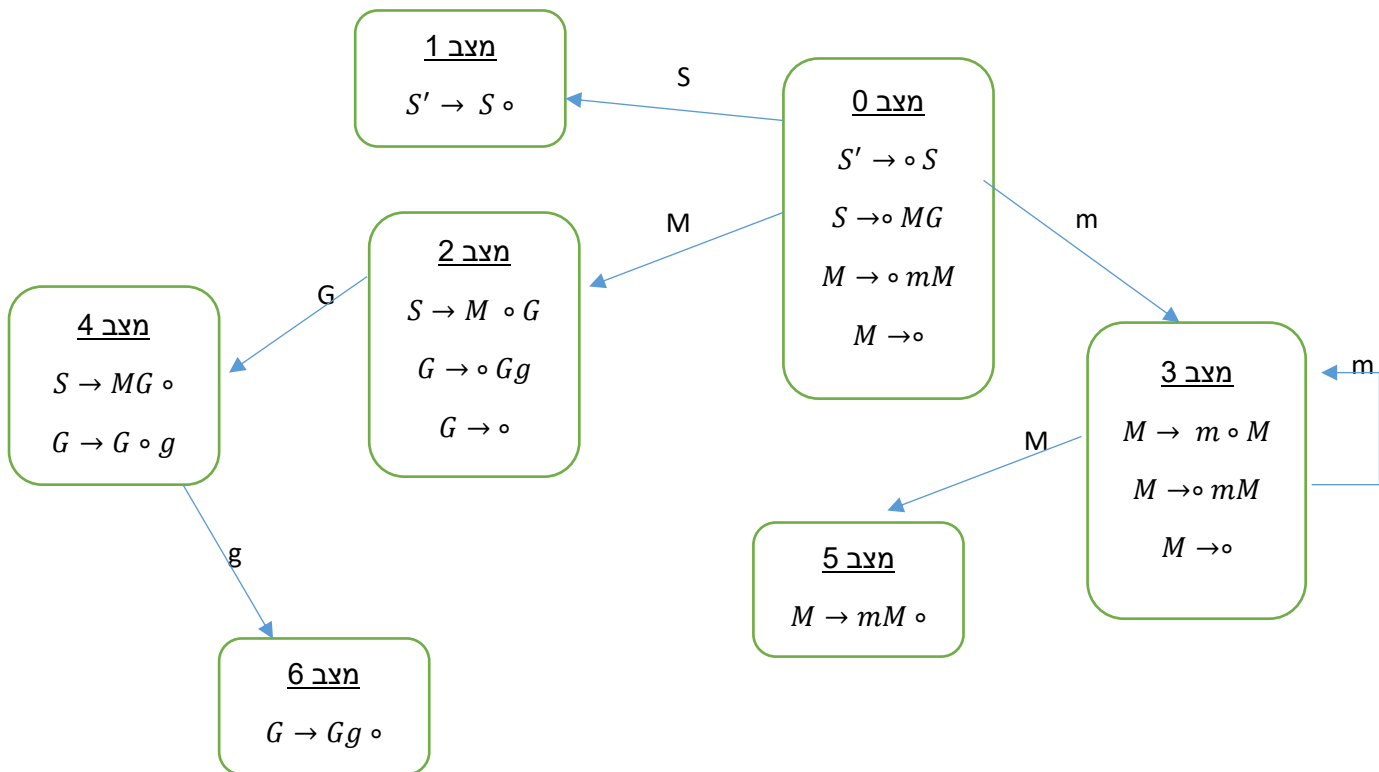
### סוגי קונפליקטים

- קונפליקט – 2 או יותר actions (shift / reduce) באותה כניסה בטבלת ה-SLR. עבור כל דקדוק קיימת טבלת SLR אחת. אם הטבלה בלי קונפליקטים, אז הדקדוק הוא SLR(1) – simple | left read input | find right most derivation | lookahead size is 1.
- קונפליקט מסוג shift/reduce: זה קורה כאשר במצב מסויים יש 2 פריטים מהצורה  $A \rightarrow \alpha_1 \circ, B \rightarrow \alpha_2 \circ, \alpha \alpha_3$ , ומתקיים התנאי ש- $a$  שייך ל- $FOLLOW(A)$ . כלומר בטבלה, באותה הכניסה עבור אותו הטרמינל יש גם reduce וגם shift.
- קונפליקט מסוג reduce/reduce: זה קורה כאשר במצב מסויים יש 2 פריטים מהצורה  $A \rightarrow \alpha \circ, B \rightarrow \beta \circ$ , ומתקיים התנאי שקיים איבר ששייך גם ל- $FOLLOW(A)$  וגם ל- $FOLLOW(B)$ .

### דגומא לבניית טבלת SLR(1)

1.  $S \rightarrow MG$  , 2.  $G \rightarrow Gg$  , 3.  $G \rightarrow \epsilon$  , 4.  $M \rightarrow mM$  , 5.  $M \rightarrow \epsilon$

- א. הרחבת הדקדוק – מוסיפים כלל:  $S' \rightarrow S$   
ב. בניית אוטומט פריטי LR(0):



## קומפילציה

ג. בניית הטבלה:

נחשב את פונקציית FIRSTS על ה-nonTerminals:

$$\text{FIRST}(S) = \{m, g, \epsilon\}$$

$$\text{FIRST}(G) = \{g, \epsilon\}$$

$$\text{FIRST}(M) = \{m, \epsilon\}$$

נחשב את פונקציית FOLLOW-ה:

$$\text{FOLLOW}(S) = \{\$ \}$$

$$\text{FOLLOW}(M) = \{g, \$ \}$$

$$\text{FOLLOW}(G) = \{\$, g\}$$

	action			goto		
	m	g	\$	S	G	M
0	s3	$r(M \rightarrow \epsilon)$	$r(M \rightarrow \epsilon)$	1		2
1			Accept			
2		$r(G \rightarrow \epsilon)$	$r(G \rightarrow \epsilon)$		4	
3	s3	$r(M \rightarrow \epsilon)$	$r(M \rightarrow \epsilon)$			5
4		s6	$r(S \rightarrow MG)$			
5		$r(M \rightarrow mM)$	$r(M \rightarrow mM)$			
6		$r(G \rightarrow Gg)$	$r(G \rightarrow Gg)$			

נרץ את ה-parser על הקלט mgg:

תוכן המחסנית	יתרת הקלט	action
0	mgg\$	
0m3	gg\$	Shift
0m3M5	gg\$	Reduce by $M \rightarrow \epsilon$
0M2	gg\$	Reduce by $M \rightarrow mM$
0M2G4	gg\$	Reduce by $G \rightarrow \epsilon$
0M2G4g6	g\$	Shift
0M2G4	g\$	Reduce by $G \rightarrow Gg$
0M2G4g6	g\$	Shift
0M2G4	\$	Reduce by $G \rightarrow Gg$
0S1	\$	Reduce by $S \rightarrow MG$
	accept	

תרגיל:

דקדוק לא ידוע:

קלט abcd

כמה shifts יהיו? – 4

כמה פעולות reduce? – תלוי בכללי גזירה, כלומר כמה non-terminals יש בחוקי השפה, כמספר ה-nonterminals יהיה מספר reduce.

**הגדרה מונחת תחביר – (SDD – sdd.doc) syntax directed definition**

במקרים מסויימים נרצה לחשב ערכים בצמתים של עץ גזירה. לדוגמא בביטוי אריתמטי.

הגדרה מונחת תחביר מתארת בצורה פורמלים כיצד ניתן לחשב ערכים אלו. כלומר נוצר עץ גזירה.

יש תכונות לסימני דקדוק – attributes

**תכונות נבנות – synthesized** – צומת N בעץ הגזירה מחושבת בעזרת תכונות בילדים של N. (ואולי תכונות אחרות ב-N עצמו). אם למשתנה A יש תכונת נבנית A.s אז לכל כלל גזירה של A יש לשייך כלל סמנטי שמחשב את A.s (בעזרת תכונות של סימני דקדוק המופיעים בכלל הגזירה). תכונות של אסימונים (טרמינלים) נחשבות לתכונות נבנות.

**תכונות מורשות – inherited** – צומת N בעץ הגזירה מחושבת בעזרת תכונות באחים ובאבא של N. (ואולי תכונות אחרות ב-N עצמו). אם למשתנה A יש תכונה מורשת A.i אז לכל כלל גזירה שבו A מופיע בצד ימין. יש לשייך כלל סמנטי שמחשב את A.i (בעזרת תכונות של סימני דקדוק המופיעים בכלל הגזירה).

נשתמש בדקדוק הבא:

$E \rightarrow E+T$

$E \rightarrow T$

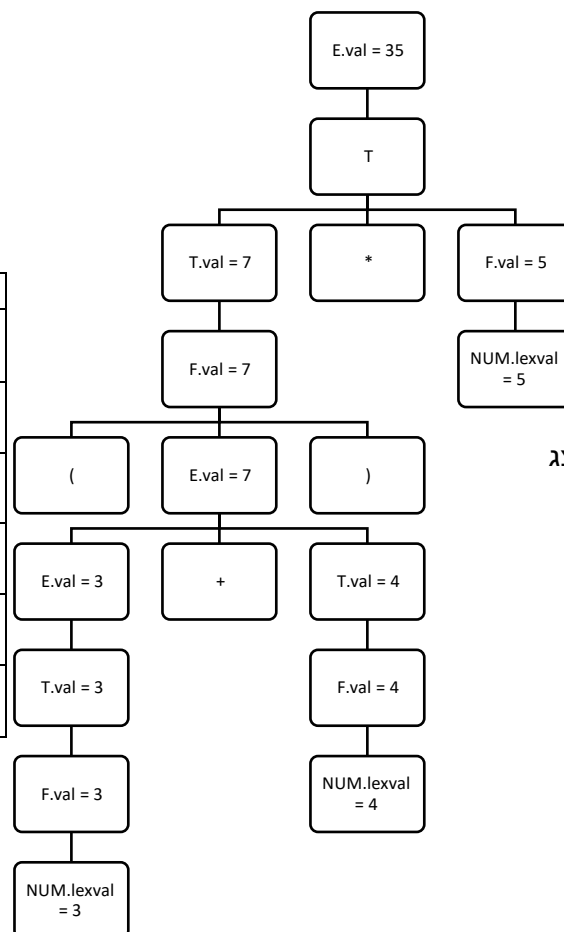
$T \rightarrow T * F$

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow \text{NUM}$

כלל גזירה	כלל סמנטי
$E \rightarrow E1+T$	$E.val = E1.val + T.val$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow T1 * F$	$T.val = T1.val * F.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow (E)$	$F.val = E.val$
$F \rightarrow \text{NUM}$	$F.val = \text{NUM.lexval}$



ניקח לדוגמא את הקלט הבא:  $(3+4) * 5 = 23$

בדוגמא זו נעשה שימוש בתכונות נבנות.

בדוגמא שלנו התכונות הם:

- E.val – הערך של הביטוי ש-E מייצג (גוזר)
- T.val – הערך (term) של המחובר ש-T מייצג
- F.val – אותו הדבר
- NUM.lexval – הערך של האסימון

דוגמא למציאת מקסימום מבין סדרת מספרים

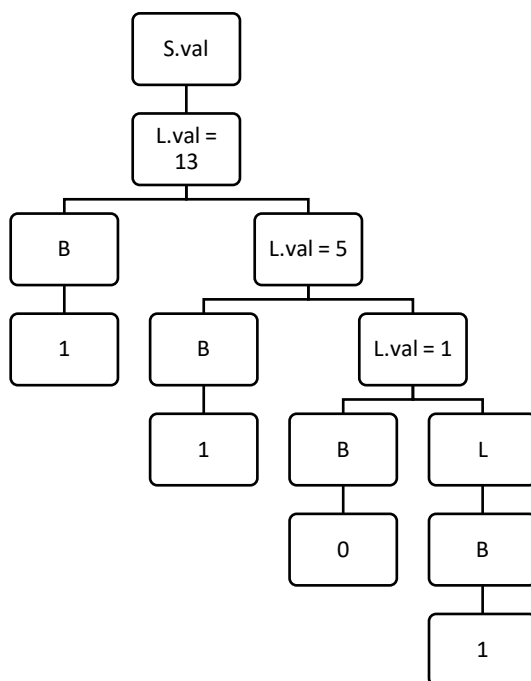
ניקח את סדרת המספרים הבאה: 3,1,5,4

כלל גזירה	כלל סמנטי
$S \rightarrow L$	$S.max = L.max$
$L \rightarrow L1, NUM$	$L.max = \text{maximum}(L1.max, NUM.val)$
$L \rightarrow NUM$	$L.max = NUM.val$

חישוב ערך של מספר בינארי

ניקח את המחזורת:  $1101 = 13$

כלל גזירה	כלל סמנטי
$S \rightarrow L$	
$L \rightarrow BL1$	$L.val = B.bit * 2^{(L1.len)} + L1.val$ $L.len = 1 + L1.len$
$L \rightarrow B$	$L.val = B.bit$ $L.len = 1$
$B \rightarrow 0$	$B.bit = 0$
$B \rightarrow 1$	$B.bit = 1$



Bison parse generator (גרסת קוד פתוח של yacc, של GNU)

מה עושים כדי להכין את התוכנית? (עבודה עם CMD)

1. Flex exp.lex – נוצר קובץ lex.yy.c שכולל את הפונ' yylex.
2. מריצים bison -d exp.y – נוצרים קבצים exp.tab.h ; exp.tab.c - בקובץ C נמצאת הפונקציה yyparse() שהיא אחראית על כל העבודה.
3. מקמפלים בעזרת קומפיילר לשפת C את הקבצים שנוצרו בצעדים קודמים, למשל : gcc exp.tab.c lex.yy.c -o exp.exe
4. מריצים את התוכנית שבנינו על הקלט (למשל  $3+4*5$ ) : exp.exe mytext.txt ויודפס 23 value is.

הפונקציה yylex תחזיר מס' המציין את סוג האסימון שהיא מצאה. אם לאסימון יש ערך סמנטי היא תכתוב אותו למשתנה הגלובלי yyval שהוא משתנה של bison.

בקובץ exp.y של BISON מגדירים את ה-tokens וה-type .  
NUM <ival> token %ומר אומר שהקבוע – NUM יהיה ב-ival של yyval.

factor expression term <ival> type %, מגדיר את חוקי הגזירה עבור ival, שאותם מגדירים בהמשך הקובץ. כלל הגזירה line הוא שורש העץ, אי אפשר להגיע אליו שוב במהלך הניתוח של הקלט ומשתמשים בו רק בתחילת הקלט, כמו כן אין לו ערך סמנטי (בהתאם לקלט, אפשר לחשוב על זה כשהשמה למשתנה - nonterminal) ולכן אין צורך להגדיר אותו בשורה של type %. זה גם אומר שהערך הסמנטי של כללי הגזירה האלו (\$\$) ישבו בשדה ival.

ככל אצבע אסימונים – טרמינלים נכתבים באותיות גדולות ומשתנים nonterminals נכתבים באותיות קטנות (אלה שמצד שמאל לכלל גזירה).

כשנעשה push למחסנית נגיד כשראינו ADDOP, אז נדחף למחסנית המשתנה yyval, שבו אגור הערך הסמנטי (+/-) של ADDOP.

בכללי גזירה ה-\$\$ מציין את הערך הסמנטי בעצם 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.

לדוגמא:

עבור הקלט  $3+4*5$ . 3 נכנס בהתחלה למחסנית, זהו אסימון NUM לפי ה-flex, לכן הוא יכנס תחילה ל-factor (זהו הערך הסמנטי שלו), ו-3=\$\$. לאחר מכן יש במחסנית factor(\$\$=\$1) כי יש רק ביטוי אחד, ומזה ניתן לעשות צמצום ל-term וכך הלאה. כשמגיעים למצב שיש ביטוי במחסנית כמו 5 = factor, \* = MULOP, 4 = term, אפשר לצמצם את זה ע"י factor MULOP term כשנעשים הבדיקות הנדרשות וזה יצמצם ל-expression.

הפעולות של  $4*5$ : רואים את 4 במחסנית, מחליפים אותו ב-NUM ועושים reduce ל-factor לפי הכלל גזירה שלו, ומצמום נוסף ל-term. לאחר מכן רואים '\*' ומחליפים אותו ב-MULOP, לאחר מכן רואים 5, מחליפים אותו ב-NUM ואז ל-FACTOR, ואז מתקבל במחסנית הביטוי factor MULOP term. לפי החוקי גזירה ובניית האוטומט הוא יודע לא לצמצם את ה-term ל-expression.

המשתנה yyvsp מייצג את המחסנית, שנשמרת במערך. yyvsp[(2)-(3)] = yyvsp[-1] כלומר ניגש לראש המחסנית. האינדקסים עובדים הפוך – בסדר שלילי. 2- זה האיבר מתחת לראש המחסנית וכו'.

דוגמא ל-struct:

נגדיר את ה-struct:

```
%code requires {  
    Struct mydate {  
        Int day, month, year;  
    }  
    %union {  
        Struct mydate d;  
        Int ival;  
    }  
    %token <ival> NUM  
    %type <d> date  
}
```

```
date: NUM '/' NUM '/' NUM  
      $1 $2 $3 $4 $5  
{ $$ .day = $1;  
  $$ .month = $3;  
  $$ .year = $5; }
```

```
Line : date { printf("day = %d, month = %d, year = %d", $1.day, $1.month, $1.year)}
```

\$1 מייצג את ה-date בכלל גזירה של line, אשר הוא שמור ב-d שהוא מטיפוס mydate ולא יש את השדות day, month, year.

## קומפילציה

דוגמא להגדרה מונת תחביר של סדרת פיבונאצ'י: 011235

תכונות :

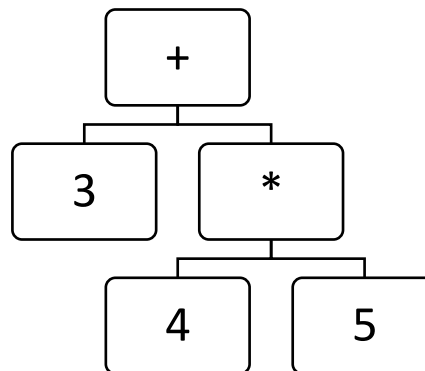
- NUM.val – המספר
- S.fib – האם זו סדרת פיבונאצ'י ( בעל ערך true/false).
- L.fib – האם סדרת המספרים ש-L מייצג היא סדרת פיבונאצ'י (בעל ערך true/false).
- L.sum2 – סכום 2 המספרים האחרונים בסדרה L

$S \rightarrow L$	$S.fib = L.fib$
$L \rightarrow L \text{ NUM}$	$L.fib = L1.fib \ \&\& \ L1.sum2 == \text{NUM.val}$ $L.sum2 = L1.last + \text{NUM.val}$ $L.last = \text{NUM.val}$
$L \rightarrow \text{NUM1 NUM2}$	If ( $\text{NUM1.val} == 0 \ \&\& \ \text{NUM2.val} == 1$ ) $L.fib = \text{true};$ else $L.fib = \text{false};$ $L.sum2 = \text{NUM1.val} + \text{NUM2.val}$ $L.last = \text{NUM2.val}$

### SDD המתאר בניית AST (abstract syntax tree) עבור ביטויים אריתמטיים

כלל גזירה	כלל סמנטי
$E \rightarrow E1 + T$	$E.tree = \text{mknode}(+, E1.tree, T.tree)$
$E \rightarrow T$	$E.tree = T.tree$
$T \rightarrow T1 * F$	$T.tree = \text{mknode}(*, T1.tree, F.tree)$
$T \rightarrow F$	$T.tree = F.tree$
$F \rightarrow (E)$	$F.tree = E.tree$
$F \rightarrow \text{NUM}$	$F.tree = \text{mkleaf}(\text{NUM.val})$

המטרה כאן היא לבנות syntax tree של ביטוי אריתמטי ולא לחשב אותו. כלומר נרצה לקבל את עץ שמכיל ערכים סמנטיים לפי כללי הגזירה. (עץ שהקומפילר יכול לבנות לעצמו):



כל עלה בעץ הזה הוא מספר (למרות שיכול להיות גם משתנה, אבל לא מטפלים בזה כרגע). מטרת הקומפילר לבניית העץ הזה הוא לבדוק תקינות סינטקס של ביטויים אריתמטיים. דוגמא לבניית העץ:

```

P1 = mkleaf(3);
p2 = mkleaf(4);
p3 = mkleaf(5);
p4 = mknode(*, p2, p3);
p5 = mknode(+, p1, p4);
    
```

P הוא פוינטר לצומת בעץ (כולל עלים).

תכונות:

- NUM.val – המספר בביטוי האריתמטי
- E.tree – syntax tree עבור הביטוי E
- T.tree
- F.tree

## קומפילציה

כלל סמנטי	כלל גזירה
	$S \rightarrow L$
$C.pos = L'.len$ $L.len = L'.len + 1$	$L \rightarrow L'C$
$C.pos = 0$ $L.len = 1$	$L \rightarrow C$
	$C \rightarrow a$
	$C \rightarrow b$

### דוגמא פשוטה לשימוש בתכונה מורשת

נרצה לחשב את התכונה המורשת  $C.pos$  (המיקום של  $C$  – תו, בקלט).

$L.len$  – אורך המחרוזת ש- $L$  מייצג.

### דוגמא נוספת

חישוב ערך של מספר בינארי :  $1101 = 13$

כלל סמנטי	כלל גזירה
$S.val = L.val$	$S \rightarrow L$
$B.w = 2^{(L1.len)}$ $L.len = L1.len + 1$ $L.val = L1.val + B.bit * B.w$	$L \rightarrow BL1$
$B.w = 1$ $L.len = 1$ $L.val = B.bit$	$L \rightarrow B$
$B.bit = 1$	$B \rightarrow 1$
$B.bit = 0$	$B \rightarrow 0$

$B.w$  – המשקל של הסיבית, תכונה מורשת. מכיוון שמשתמשים ב"אח" של  $B$  כדי לחשב אותה ( $L.len$ )

$B.bit$  – תכונה נבנית, ערך הסיבית.

ניתן לקשט את העץ, כלומר לחשב את ערכי התכונות בכל הצמתים של עץ הגזירה, בסדר הבא:

- נחשב את  $B.bit$  בכל הצמתים המסומנים ב- $B$ .
- נחשב את  $L.len$  בכל הצמתים המסומנים ב- $L$ . (מלמטה למעלה).
- נחשב את  $B.w$  בכל הצמתים המסומנים ב- $B$ . (הסדר לא חשוב).
- נחשב את  $L.val$  בכל הצמתים המסומנים ב- $L$ .
- נחשב את  $S.val$ .

**הערה: ההגדרה אינה מסוג  $L$ .**

### אלגוריתם למציאת סדר חישוב תכונות בצמתים של עץ גזירה

הבעיה היא כשיש תכונות מורשות (כי בתכונות נבנות זה מלמטה למעלה תמיד), כיוון שאפשר לחשב חלק מהתכונות ללא כל סדר, ואילו תכונות אחרות תלויות בתכונות מסויימות ולהן נדרש סדר חישוב. ניתן להסתכל על סדר חישוב התכונות כעל מטלות בעלות סדר היררכי. לכן ניתן לעשות עליהן מיון טופולוגי.

בונים גרף תלויות שבו כל צומת מייצג "מטלה" – חישוב תכונה מסויימת בצומת מסויים בעץ הגזירה. בגרף תהיה קשה מ"מטלה"  $X$  ל"מטלה"  $Y$  אם צריך לבצע את  $X$  לפני  $Y$ . ועל גרף זה עושים מיון טופולוגי.



## קומפילציה

### דוגמא נוספת

כלל גזירה	כלל סמנטי
$S \rightarrow L$	$L.in = 0$
$L \rightarrow L1B$	$B.w = 2^{L.in}$ $L1.in = L.in + 1$ $L.val = L1.val + B.bit * B.w$
$L \rightarrow B$	$B.w = 2^{L.in}$ $L.val = B.bit * B.w$
$B \rightarrow 1$	$B.bit = 1$
$B \rightarrow 0$	$B.bit = 0$

$L.val$  – סכום התרומות של הסיביות שמרכיבות את  $L$  לתוצאה הסופית.  
 $L.in$  – כמה סיביות יש מימין ל $L$ . (תכונה מורשת – כיוון שמשתמשים בתכונה של האבא).  
 $B.bit$   
 $B.w$  – המשקל של  $B$  (תכונה מורשת).

### $L$ – attributed definition

הגדרה מונחית תחביר היא  $L$ -attributed אם כל החישובים של התכונות המורשות תלויות רק בתכונות של האבא ובתכונות של האחים השמאליים. כיוון שחישוב ערכי התכונות בעץ גזירה בהגדרה מסוג  $L$  מתבצע ברקורסיה משמאל לימין בעץ, וכל לכל צומת קיימות התכונות רק של האבא ושל האחים השמאליים. בעצם מתבצע DFS משמאל לימין. התהליך הוא לרדת לעלה השמאלי ביותר, לחשב את התכונות שלו, לעבור לאח הימני שלו ולחשב את התכונות שלו. בדרך לעלה השמאלי מחשבים את התכונות המורשות של כל צומת (בעצם של כל אבא עד לעלה). אחרי שמחשבים את כל הילדים של התת עץ השמאלי הקטן ביותר (כלומר את הילדים שלו) עולים חזרה לאבא (כמו שיטת post-order), ומחשבים את התכונות הנבנות של האבא.

### קוד ביניים – intermediate code

[מחולל קוד מטרה, backend]  $\rightarrow$  קוד ביניים  $\rightarrow$  תלוי שפת תכנות [frontend, lexer, parser, semantical analyzer]  $\rightarrow$  Source program  
 $\rightarrow$  target code (assembly ...)  $\rightarrow$  assembler  $\rightarrow$  machine code

### למה צריך קוד ביניים

- מודלריות – חלוקה ל-frontend ו-backend. זה מאפשר להחליף שפת תכנות ולהשפיע רק על ה-frontend או לחליפין להחליף מעבד ולהשפיע רק על ה-backend.
  - כדי שנוכל להפעיל אלגוריתמים לאופטימיזציה שאינה תלויה במעבד ספציפי.
  - קל יותר לייצר קוד מטרה מקוד ביניים מאשר משפת המקור – source language.
- בקוד ביניים המטרה היא לפשט את הקוד כמה שניתן, לכן מתשמשים רק באופרטור יחיד (בנוסף להשמה) כמו:  $a = b + c$ ,  $a = b$ , if  $a > 0$  goto

דוגמא:

תכונות : כל התכונות במקרה הזה הן נבנות

- ID.name – שם האסימון (הייצוג שלו בתור מחרוזת).
- Stmt.code – התרגום של המשפט לקוד ביניים.
- Exp.code – התרגום של הביטוי לקוד ביניים. הקוד מחשב את הביטוי וכותב את התוצאה למשתנה exp.result.

כלל גזירה	כלל סמנטי
Stmt $\rightarrow$ ID = exp;	Stmt.code = exp.code    gen(ID.name '=' exp.result)
Exp $\rightarrow$ exp1 + exp2	Exp.result = newtemp(); Exp.code = exp1.code    exp2.code    gen(exp.result '=' exp1.result '+' exp2.result)
Exp $\rightarrow$ exp1 * exp2	Exp.result = newtemp(); Exp.code = exp1.code    exp2.code    gen(exp.result '=' exp1.result '*' exp2.result)
Exp $\rightarrow$ (exp1)	Exp.code = exp1.code Exp.result = exp1.result
Exp $\rightarrow$ ID	Exp.code = "" Exp.result = ID.name

נשתמש בקלט :  $a = (b+c)*(d+f)$

קוד ביניים:

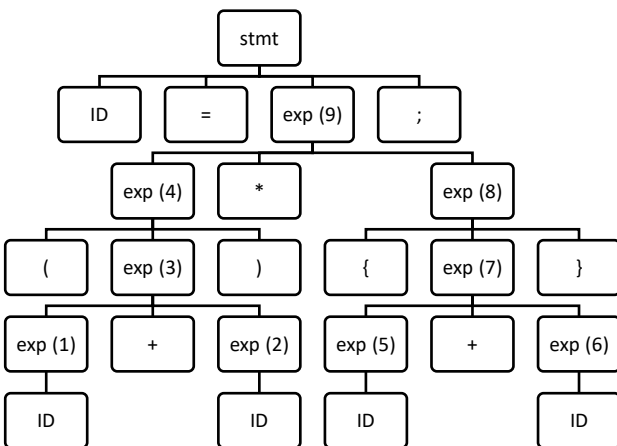
$T1 = b+c$

$t2 = d+f$

$t3 = t1*t2$

$a = t3$

כדי לייצר את המשתנים הזמניים  $t1, t2, t3$  נשתמש בפונקציה newtemp –



דוגמת הרצה:

Exp1.code = "", exp1.result = ID.name = b

Exp2.code = "", exp2.result = ID.name = c

Exp3.result = newtemp()  $\rightarrow$  t1, exp3.code = "" || "" || t1 = b+c

Exp4.result = exp3.result = t1, exp4.code = t1=b+c

Exp5.code = "", exp5.result = ID.name = d

Exp6.code = "", exp6.result = ID.name = f

Exp7.result = newtemp()  $\rightarrow$  t2, exp7.code = "" || "" || t2 = d+f

Exp8.result = exp7.result = t2, exp8.code = t2= d+f

Exp9.result = newtemp()  $\rightarrow$  t3, exp9.code = t3 = (b+c)\*(d+f)

## קומפילציה

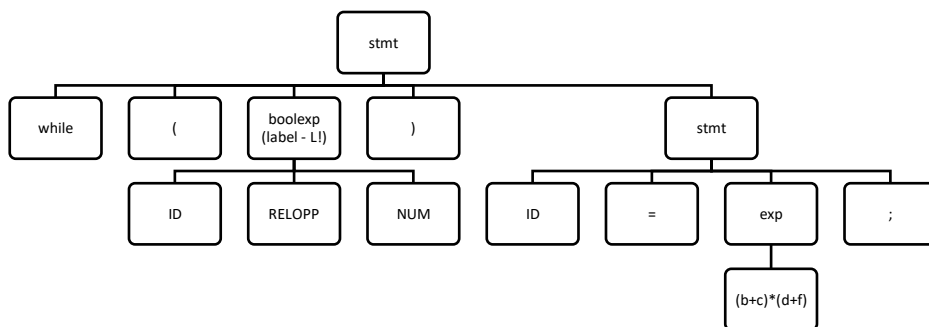
### קוד ביניים עבור משפטי while

נראה כיצד נראה תרגום של משפטים while לקוד ביניים. לצורך כך נשתמש בהגדרה מונחת תחביר.

While( $y < 3$ )  
 $a = (b+c) * (d + f)$

- Boolexp.code – התרגום של המשפט לקוד ביניים. - Boolexp. - Newlabel() – פונ' שיוצרת תווית חדשה, L1, L2	L2 : if false $y < 3$ goto L1 $t1 = b+c$ $t2 = d+f$ $t3 = t1*t2$ $a = t3$ goto L2 L1 :
--	--

כלל גזירה	כלל סמנטי
$\text{Stmt} \rightarrow \text{ID} = \text{exp};$	$\text{Stmt.code} = \text{exp.code} \parallel \text{gen}(\text{ID.name} = \text{exp.result})$
$\text{Exp} \rightarrow \text{exp1} + \text{exp2}$	$\text{Exp.result} = \text{newtemp}();$ $\text{Exp.code} = \text{exp1.code} \parallel \text{exp2.code} \parallel \text{gen}(\text{exp.result} = \text{exp1.result} + \text{exp2.result})$
$\text{Exp} \rightarrow \text{exp1} * \text{exp2}$	$\text{Exp.result} = \text{newtemp}();$ $\text{Exp.code} = \text{exp1.code} \parallel \text{exp2.code} \parallel \text{gen}(\text{exp.result} = \text{exp1.result} * \text{exp2.result})$
$\text{Exp} \rightarrow (\text{exp1})$	$\text{Exp.code} = \text{exp1.code}$ $\text{Exp.result} = \text{exp1.result}$
$\text{Exp} \rightarrow \text{ID}$	$\text{Exp.code} = ""$ $\text{Exp.result} = \text{ID.name}$
$\text{Boolexp} \rightarrow \text{ID RELOP NUM}$	$\text{Boolexp.label} = \text{newlabel}()$ – (creates L1 label, in an attribute of boolexp, it's needed in more places) $\text{Boolexp.code} = \text{gen}(\text{"if False" ID.name RELOP.op NUM.val "goto" boolexp.label})$
$\text{Stmt} \rightarrow \text{WHILE}(\text{boolexp}) \text{ stmt1}$	$\text{Startlabel} = \text{newlabel}()$ – (create L2 label, assign it to a local variable) $\text{Stmt.code} = \text{gen}(\text{startlabel} : \text{""} \parallel \text{boolexp.code} \parallel \text{stmt1.code} \parallel \text{gen}(\text{"goto" startlabel}) \parallel \text{gen}(\text{boolexp.label} : \text{""}))$



עץ הגזירה של הדוגמא הזו (חלקי, החלק החסר נמצא בדוגמא הקודמת של exp).

לרוב ניצור if False בלולאת while, כיוון ש-while זאת לא פקודה בקוד ביניים, לעומת זאת if זאת פקודה קיימת בקוד ביניים.

## קוד ביניים עבור משפטי if

```
If(y<3)
a = (b+c)*(d+f);
else
a = h+g
```

```
If False y<3 goto L1
    t1 = b+c
    t2 = d+f
    t3 = t1*t2
    a = t3
    goto L2
L1: t4 = h+g
    a = t4
L2:
```

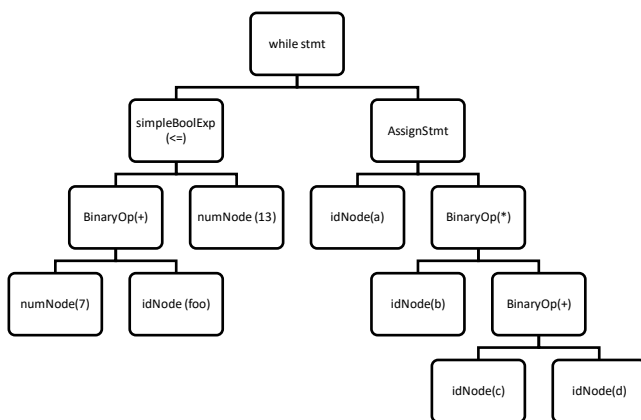
טבלת כללי גזירה חלקית (הדברים החסרים בטבלאות הקודמות).

כלל גזירה	כלל סמנטי
Boolexp $\rightarrow$ ID RELOP NUM	יוצר תווית 'אל' – $\text{Boolexp.label} = \text{newlabel}()$ $\text{Boolexp.code} = \text{gen}(\text{"if False" ID.name RELOP.op NUM.val "goto" boolexp.label})$
Stmt $\rightarrow$ IF(boolexp) stmt1 ELSE stmt2	$\text{Exit\_label} = \text{newlabel}()$ – (creates L2 label, exit_label is a local variable) $\text{Stmt.code} = \text{boolexp.code} \parallel \text{stmt1.code} \parallel \text{gen}(\text{"goto" exit\_label})$ $\parallel \text{gen}(\text{boolexp.label} \text{":"}) \parallel \text{stmt2.code} \parallel \text{gen}(\text{exit\_label} \text{":"})$

## תרגיל 2:

אופציה א' - ניתן לייצר קוד ביניים במהלך ה-parsing

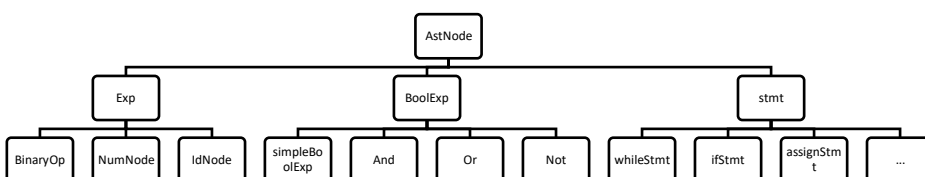
אופציה ב' - ה-parser מייצר AST (abstract syntax tree) עושים מעבר על העץ ומייצרים קוד ביניים.



דוגמא ל-AST עבור הקוד:  
 $\text{while}(7+\text{foo} \leq 13)$   
 $a = b * (c + d)$

משתנה FALL\_THROUGH - אומר שאם התנאי הוא אמת, אז תמשיך לגוף הביטוי, אם שקר תעבור לתווית אחרת. ב-bison יש סימון נוסף - @ שהוא מסמן location של האסימון בקלט. Yylex כותבת את המיקום של האסימון לתוך המשתנה הגלובלי - yylloc - lexical location. Bison מנהל אוטומטית גם את מיקום המשתנים - נוגרמינלים.

## היררכיה של המחלקות של ה-AST

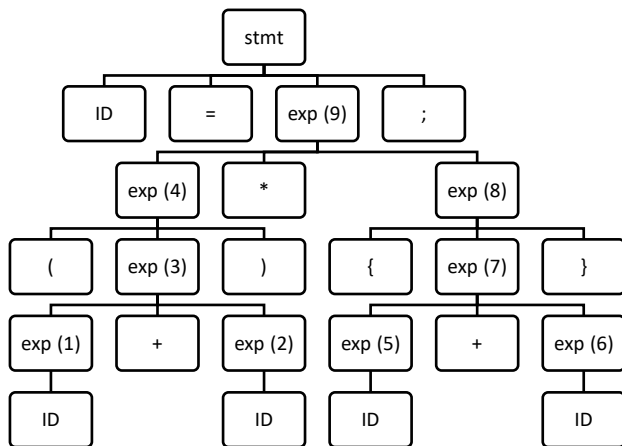


דוגמא לייצור קוד ביניים במהלך ה-parsing עם bison

כלל גזירה	ייצור קוד ביניים (קוד שנכתב בתוך ה-scope של כלל הגזירה)
Stmt $\rightarrow$ ID = exp;	Emit (\$1 '=' \$3) - \$3 has the t3 var, which representing the result
Exp $\rightarrow$ exp1 + exp2	\$\$ = newtemp() – will create the temp vars t1 and t2 emit (\$\$ '=' \$1 '*' \$3) – write the medCode to the file
Exp $\rightarrow$ exp1 * exp2	\$\$ = newtemp() – will create the temp var t3 emit (\$\$ '=' \$1 '*' \$3)
Exp $\rightarrow$ (exp1)	\$\$ = \$2
Exp $\rightarrow$ ID	\$\$ = \$1

קוד ביניים מייצג רק שינויים

בערכים של המשתנים, כלומר בפעולה שבה אין שינוי בערך המשתנה אין צורך לקוד ביניים. כמו בכלל גזירה: exp $\rightarrow$ (exp1)



ערך סמנטי עבור כל צומת – exp יהיה אותו המשתנה שיחזיק את תוצאת החישוב של אותו הצומת.  
לדוגמא: עבור הקלט  $a = (b+c)*(d+f);$  והקוד הביניים:

$T1 = b+c$

$t2 = d+f$

$t3 = t1*t2$

$a = t3$

הצומת של exp(9) יחזיק את המשתנה t3, כיוון שכרגע אין מספרים לכל המשתנים, וכמו כן התוצאה יכולה להשתנות בכל קלט או בריצה בלולאה (אנחנו בונים קוד ביניים כללי עבור כל קלט אפשרי תקין)

דוגמא ללולאת while:

עבור הקלט :  
while(y<3)  
a = (b+c)\*(d+f)

כלל גזירה	קוד ביניים
Boolexp → ID RELOP NUM	\$\$ = newLabel() emit("if False" \$1 \$2 \$3 "goto" \$\$)
Stmt → WHILE(boolexp) stmt1	<p>After the WHILE we want to create ne label for the medCode  So we can write:  {\$\$ = newlabel()  emit(\$"\$:")} – in this case bison will create a var for the rule (the \$\$ is for the rule and not representing the semantic value of the 'stmt')</p> <p>Or we can place another rule as 'beginLoop' (bison will do it in the background for the previous case).  beginLoop: \$empty  { \$\$ = newLabel()  emit(\$"\$:") }</p> <p>The type for the midRuleAction \$\$ will be defined as: \$&lt;---&gt;\$, in the diamond braced will be the type of the var from the union</p> <p>When the parser will work on the last stmt it will create the medCode for the while scope, and in the end we need to add the "goto" :  {emit "goto" \$2} – the \$2 is the new label in the beginLoop  Emit( \$4":") - \$4 is the label in the boolexp</p>

Backpatching

בבוקד ביניים, כשנרצה לעבור בעזרת goto לפקודה אחרת, במקום תווית כמו שראינו עד עכשיו, "נקפוץ" למספר כלשהו, אשר מייצג את מיקום הפקודה (בשורות).

100 : if false y<3 goto __106	לדוגמא:
101:t1 = b+c	While(y<3)
102:t2 = d+f	a = (b+c)*(d+f)
103:t3 = t1*t2	
104:a = t3	כשניצור את הקוד ביניים, אנחנו לא נדע מה יהיה אורך הגוף של הלולאה
105:goto 100	ולכן נשלים את ה-goto הראשון רק אחרי שנסיים להפוך את גוף הלולאה לקוד ביניים.

Emit – מוסיפה פקודה בבוקד ביניים למערך של פקודות לכניסה הפנוייה הבאה עם אינקס next\_instr, ומקדמת את next\_instr. זהו משתנה גלובלי בתוך הפקודה emit.

בדוגמא הנ"ל נכתוב backpatch(100,106), פקודה שאומרת מאיזו שורה לקפוץ לאיזו שורה. Backpatch זאת פקודה שאנחנו צריכים לכתוב, וכל מה שהיא עושה זה אומרת לבison, לך לשורה 100 (במקרה שלנו) ותכתוב שם ב-goto את 106.

כלל הגזירה עבור הדוגמא:

Boolexp: ID RELOP NUM

```
{
    $$ = next_instr
    emit("if False" $1 $2 $3 "goto _"); }
```

אנחנו נרצה לחזור לפקודה הזו ולהגדיר את ה-`next_instr` ולכן נתייחס אליו כערך הסמנטי של `boolexp`.

Stmt: WHILE '(' boolexp ')'

```
stmt
{ emit("goto"$3)
  backpatch($3, next_instr); }
```

\$3 זה הערך הסמנטי של `boolexp` בכלל הזה, ואנחנו רוצים לכתוב בסוף גוף ה-`while` את פקודת הקפיצה, `goto 100`,  
`$3 = (boolexp)next_instr = 100`. כמו כן נרצה להשלים את ה-`goto` הראשון (106) ולכן נעדכן עם פקודת `backpatch` את הערך  
 הסמנטי של `boolexp` לשורה הנוכחית.

### דוגמא עבור if

```
If(y<3)
a = (b+c)*(d+f);
else
a = h+g
```

Boolexp: ID RELOP NUM

```
{
    $$ = next_instr
    emit("if False" $1 $2 $3 "goto _"); }
```

Stmt: IF '(' boolexp ')'

```
stmt
ELSE
{ $$ = next_instr;
  emit("goto" __); }
stmt
{ backpatch ($3, $7+1)
  backpatch($7, next_instr); }
```

Another way to remember the instruction  
 line:  
 Else marker { emit ("goto" \_\_); }  
 ...  
 marker : /\* empty\*/ { \$\$ = next\_instr ;}

```
100:if False y<3 goto 106
101: t1 = b+c
102: t2 = d+f
103: t3 = t1*t2
104: a = t3
105: goto 108
106: L1: t4 = h+g
107: a = t4
108:....
```

## איך זה עובד?

כשמריצים קוד, יש מחסנית קריאות שבה נשמרים פעולות וקריאות לפונקציות. עבור קריאות לפונקציות, במחסנית הקריאות יש מחסנית פריימים נוספת ששומרת את סדר הקריאות של הפונקציות (מתוך פונקציות), כיוון שצריכים סדר LIFO ביציאה מהפונקציות.

Access link (static link) - מצביע לרשומת הפעלה של הפונקציה שמקיפה את הפונקציה הנוכחית. יש שפות שמאפשרות כתיבה של פונקציה בתוך פונקציה (כמו פייתון). **לפני הריצה**, במצב סטטי, ניתן לראות את היררכיית הקריאות של הפונקציות המקוננות, לפי הקריאות באותן הפונקציות. לכן כשרוצים לקרוא לפונקציה מקיפה מהפונקציה המקוננת הנוכחית, צריכים את ה-access link וכך ניתן לגשת גם לפרמטרים של הפונקציה המקיפה. ה-ARP של ה-access link תמיד מצביע על תחילת הפונק' העוטפת הקודמת ברמת הקינון וכדי לגשת למשתנה בפונקציה העוטפת נגיע ל-ARP של הפונק' העוטפת, ונוסיף לכתובת הזו את ה-offset של המשתנה (ה-offset בהנחה שהוא ידוע). לכן אם נרצה לעבור מרמת קינון 3 לרמת קינון 0, נצטרך לגשת 3 פעמים ל-ARP, כל פעם של פונק' ברמה אחת מעלה, כלומר נעבור ל-ARP של רמה 2, ואז ל-ARP של רמה 1 ולבסוף ל-ARP של רמה 0.

Dynamic link - מצביע לרשומת ההפעלה של הפונקציה שקראה לפונקציה הנוכחית (לא פונקציות מקוננות). אם לדוגמה פונקציה פנימית, נקרא לה r2, קוראת לפונקציה שעוטפת אותה, נקרא לה p1, החיבור הזה הוא dynamic link.

אם פונ' r2 נמצאת ברמת קינון 3, והיא צריכה להגיע למשתנה שנמצא בפונקציה העוטפת הראשונה, ברמה 0, הקומפילר יודע איפה הוא נמצא כרגע והוא יודע באיזו רמת קינון נמצא המשתנה, אז בזכות ה-access link הוא יחסיר את רמת הקינון הרצויה (0) מרמת הקינון הנוכחית (3) וככה הוא ידע כמה מצביעים הוא יצטרך לחזור.

## איך מוקם access link בזמן קריאה לפונקציה

הפונ' הקוראת אחראית להקמת ה-access link.

### מקרה א':

פונ' f' קוראת ל-g (יתכן ש-g, f, זאת אותה הפונקציה), בתיהן באותה רמת קינון. בגלל ששתיהן באותה רמת קינון, יש להן את אותו ה-access link, לכן ה-ARP של g יהיה זהה ל-ARP של f. F תעתיק את ה-access link של עצמה לרשומת של g.

### מקרה ב':

F קוראת ל-g.

$n_g = n_f + 1$ , (n זאת רמת הקינון של פונקציה g). כלומר רמת הקינון של g שווה לרמת הקינון של f+1. לכן בהכרח g מקוננת ישירות בתוך f, ולכן ה-access link של g יהיה זהה ל-ARP של f.

### מקרה ג':

F קוראת ל-G.

$n_g < n_f$ , כלומר רמת הקינון של g קטנה מזו של f. עוקבים אחרי  $n_f - n_g$  מצביעים ברשימת ה-access link המתחילה ב-access link של f ושם מוצאת אץ ה-access link של g. זה אומר בהכרח ששני הפונקציות מקוננות תחת אותה פונ' עוטפת כלשהי.



## קומפילציה

### Linker

Source code → compiler → assembly code → assembler → relocatable object code → linker

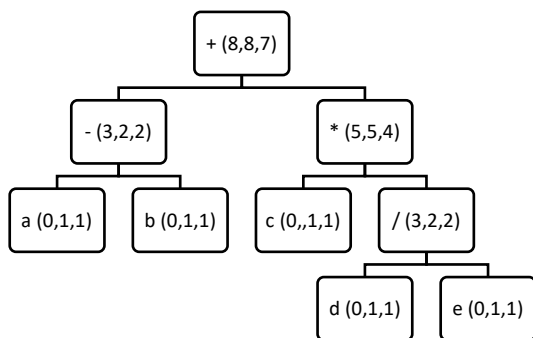
Assembler מייצר קובץ הכולל: data segment, code segment, symbol table for external symbols, relocation info

External symbols הם אותם הסמלים (משתנים גלובליים ופונקציות) אשר עושים בהם שימוש בקובץ אחד וההגדרה שלהם נמצאת בקובץ אחר. הצורך ב-symbol table נובע מכך שכשהקוד עובר קומפילציה, הקומפיילר לא יודע איפה ההגדרה (מימוש) של אותה הפונקציה, כמו כן גם האסמבלר לא יודע איפה הפונ' ממומשת, כיוון שהקומפיילרים עובדים קובץ קובץ ולא מקשרים בין קבצים. האסמבלר מייצר קוד בשפת מכונה, והלינקר כבר אחראי על הקישור בין הקבצים וככה גם בין קריאת הפונ' לבין ההגדרה שלה. האסמבלר מייצר את ה-symbol table שנמצא ב-object file והלינקר משתמש בו.

- כשכותבים בקוד C: `#include <stdio.h>`, הקומפיילר יחליף את השורה הזו בקוד, בתוך של כל הקובץ. לרוב אין שם את המימוש של הפונקציות אלא רק את החתימות, המימוש של הקוד נמצא בקבצים מקומפילים בספריות. הלינקר ישתיל בתוך הקוד שלנו בסוף את הפונקציות שהשתמשו בהם בקוד, ולא את כל המימושים של כל הפונקציות.

עבור כל קובץ, האסמבלר יוצר code segments, ככה שכל אחד מהם שמור במקום אחר בזכרון. הלינקר לוקח את אותם הסגמנטים ויוצר ביניהם הקשר, ככה שכל הכתובות שהיו בכל סגמנט (שהיו יחסיות לאותו הסגמנט) יהיו עכשיו כתובות שהלינקר מכיר וידע לגשת אליהם, כשהוא מקבץ את הסגמנטים לבלוק אחד גדול (לאו דווקא שמור ברצף בזכרון), אבל הכתובות היחסיות בכל סגמנט הופכות להיות כתובות מקושרות בין כל הסגמנטים ויחסיות עכשיו לכל הסגמנטים.

### אלגוריתם תכנות דינאמי לייצור קוד אופטימלי (בשפת אסמבלי) עבור ביטויים אריתמטיים



ה-syntax tree מייצג את הביטוי הבא:  $(a-b) + (c*(d/e))$

כדי להקל על החישובים, נניח כי למעבד יש רק 2 רגיסטרים –  $r0, r1$ , ועלויות של כל הפקודות זהות (נגיד 1).

1. מחשבים וקטורי מחירים בכל הצמתים. לכל פעולה, הקומפיילר מחזיק "מחיר" עבור אותה הפעולה, כדי שיוכל לתעדף פעולות זולות יותר.

המחירים בעץ ליד האופרטורים מייצגים:

a. הערך הימני בוקטור המחירים מייצג את מחיר חישוב הביטוי (לתוך הרגיסטר) כאשר לרשותינו עומדים 2 רגיסטרים.

b. הערך האמצעי בוקטור מייצג את המחיר כמו הימני אם לרשותינו עומד רק רגיסטר אחד.

c. הערך השמאלי מייצג את מחיר החישוב של הביטוי כאשר התוצאה נשמרת בזכרון.

2. מייצרים את הקוד.

**הסבר על עלה a:** לחשב את a זה אומר שצריך לטעון הערך שלו מהזכרון לרגיסטר: `load r0 a` (טעינה של הערך של a מהזכרון לרגיסטר  $r0$ ) ולכן העלות של פעולה זו 1 (גם כשיש לרשותינו 2 רגיסטרים וגם כשיש אחד, כי צריך רק רגיסטר אחד), והעלות לשמור את הערך הזה בזכרון היא 0 כי הוא כבר בזכרון, ולכן וקטור המחירים שלו הוא  $(0,1,1)$ .

### חישוב וקטור המחירים בצומת עם המינוס:

א. לרשותינו רגיסטר בודד – חייבים להתשמש בפקודה מהסוג: `sub r r m`

חישוב אופרנד ימני לתוך הזכרון – 0 (המשתנה m)

חישוב אופרנד שמאלי לתוך הרגיסטר – 1 (המשתנה r האמצעי) – כאשר לרשותינו רגיסטר 1 (רואים ב-a שהעלות היא 1)

מחיר ה-sub-חישוב של החיסור בין  $r-m$  כלומר  $a-b$ , המחיר הוא 1.

עכשיו נצטרך לסכום:  $0+1+1=2$  ולכן בצומת של המינוס בעץ, בוקטור הערך האמצעי הוא 2 (הערך עבור רגיסטר אחד).

## קומפילציה

ב. לרשותינו 2 רגיסטרים –

a. אופציה ראשונה היא לפתור את החיסור כמו ב-א' ואז התוצאה תצא גם 2.

b. אופציה שנייה להשתמש בפקודה:  $sub\ r\ r\ r$ , כלומר ערכי 2 האופרנדים נמצאים ברגיסטר. במצב כזה אנחנו יכולים לחשב בסדר שונה כל אופרנד (כלומר אם לכל אופרנד היה עוד תת עץ שנדרש היה לחשב את הערך שלו). אם נחשב קודם את האופרנד הימני (הערך שלו כבר יהיה שמור באחד הרגיסטרים, והיות ויש לנו רק 2 רגיסטרים, אז ישאר רק רגיסטר אחד לחישוב האופרנד השמאלי. לכן נבחן את העלויות של כל אופרנד ביחס לכמה רגיסטרים יש לו, בעץ שלנו רואים שהערכים 2 עברו 2 האופרנדים בשני המצבים של הרגיסטרים, ולכן זה לא משנה איזה אופרנד נחשב קודם). נגיד ונחשב את האופרנד השמאלי קודם, הערך שלו ישמר ב- $r$  האמצעי (רגיסטר  $r_0$  או  $r_1$ ) בפקודת ה- $sub$ , והעלות של החישוב הזה הוא 1 – לפי הוקטור מחירים של  $a$  עם 2 רגיסטרים. לאחר מכן נשאר לנו רגיסטר אחד פנוי לחישוב האופרנד הימני, העלות של פעולה זו היא 1 – לפי הוקטור המחירים של  $b$  עם רגיסטר בודד, והערך של חישוב זה יכנס לרגיסטר  $r$  הימני בפקודת  $sub$ . וכעת נותר לנו לחשב את הפקודה עצמה שהעלות של חישוב זה היא 1. ולכן אם נסכום את העלויות, התוצאה היא 3 והיא גבוהה יותר מאופציה א'. ולכן נעדיף להשתמש באופציה א'.

ג. רוצים לאסחן את התוצאה בזכרון – אין פקודה שמחשבת חיסור וגם שומרת לזכרון, לכן בשלב הראשון יש לנו 2 רגיסטרים וכבר ראינו שחישוב פעולת החיסור האופטימלי עולה 2, ועכשיו נרצה לאסחן את התוצאה לזכרון, פקודת  $store$  תעלה עוד 1 ולכן סה"כ יצא שהעלות של חישוב החיסור ואסחון התוצאה בזכרון היא 3. (ככלל אצבע, נתבונן על העלות עם 2 רגיסטרים ונסוץ לו 1 עבר פקודת  $store$ ).

### חישוב וקטור המחירים בשורש

א. לרשותינו רגיסטר בודד – חייבים להשתמש בפקודה  $add\ r\ r\ m$ . לפי הצומת של הכפל רואים שכדי לחשב את כל תת העץ הימני ולאסחן אותו בזכרון יעלה 5. לכן חישוב האופרנד הימני לזכרון בפקודת  $add$  יעלה 5. העלות של הכנסת האופרנד השמאלי לרגיסטר בודד (לפי תת העץ השמאלי) יעלה 2 ולבסוף פקודת  $add$  עולה 1. אם נסכום העלות הכוללת תהיה  $5+2+1=8$ .

ב. לרשותינו 2 רגיסטרים –

a. אופציה א', שימוש בפקודה  $add\ r\ r\ m$ . אז התוצאה תצא כמו ב-א'. (כיוון שגם פה שומרים את האופרנד בימני לזכרון)

b. אופציה ב', שימוש בפקודה  $add\ r\ r\ r$ . במצב שבו נרצה לחשב קודם את האופרנד השמאלי שיעלה 2 (מהצומת של המינוס) ולאחר מכן ישאר לנו רגיסטר בודד עבור האופרנד הימני והעלות של החישוב הזה היא 5 ועוד העלות של  $add$ , לכן סה"כ יצא 8.

c. אופציה ג', שימוש בפקודה  $add\ r\ r\ r$  כאשר נחשב קודם את האופרנד הימני, לפי התת עץ הימני העלות של זה היא 4 וחישוב של האופרנד השמאלי עם רגיסטר 1 היא 2. בסוף חישוב פקודת ה- $add$  היא 1, לכן התוצאה הסופית היא 7. והיא עדיפה.

### ייצור הקוד:

המטרה היא לייצר את הקוד, לפי וקטור המחירים בשורש נראה כי מספר שורות הקוד המינימלי הוא 7 (לפי העלות המינימלית):

Load  $r_0\ C$  – get the value of  $C$  from the memory, now we have only one available register –  $r_1$

load  $r_1\ D$

div  $r_1\ r_1\ E$  – get the value of  $E$  from the memory (costs 1), divide  $r_1 (=D) / E$  and put it in  $r_1$

mul  $r_0\ r_0\ r_1$  – now we have the right operand of the + operator. (for the left operand we have only one available register –  $r_1$ ).

load  $r_1\ a$

sub  $r_1\ r_1\ b$

add  $r_1$ (can be also  $r_0$ )  $r_1\ r_0$  – the result of the addition in register  $r_1$  (or  $r_0$ )

Store  $r_1$  – if we want to store the result in the memory we need one more command (and it generates 8 command – as we can see in the price vector of the root).

חומר למבחן: אין בחירה!!!!

- כתיבת קוד ב-*flex & bison* – כנראה רק *bison* וגם דקדוק
- טבלת  $SLR(1)$  – צריך לדעת איך לבנות ולהשתמש בה.
- טבלת  $LL(1)$  – סיכוי גבוה שיהיה (כולל *recursive descent parser*).
- ייצור קוד ביניים בעזרת *bison* (תוויות סימבוליות, *recursive descent parser*, *backpatching*) – יכול להיות בעזרת *bison* לפי כללי גזירה נתונים. אולי עם *recursive descent parser*.
- ייצור קוד מ-AST
- נושאים קטנים יותר:
- GC
- אלגוריתם תכנות דינאמי
- מספרי *ershor*
- *linker* – אם תהיה שאלה, יהיה סיכום.
- *activation record (access link)*

Garbage collection (GC)

בעיות עם שחרור זכרון ידני :

- כששוכחים לשחרר זכרון – *memory leak*
- ממשיכים להשתמש במצביע לזכרון שכבר שוחרר – *dangling pointer*.
- **Reference counter** – לכל אובייקט יש מונה הסופר את מס' ה-*references* לאובייקט (*c++,python...*). בכל פעולת השמה של המצביעים לאובייקט המונה יתעדכן בהתאם. כל פעולת השמה תגרור איתה הפחתת המונה של אובייקט אחד והגדלת המונה של אובייקט אחר, פעולה שלוקחת זמן (אומנם מאוד קטן, אבל בהיבטים מסויימים זה יכול להיות קריטי) ובנוסף המונה תופס עוד תא בזכרון לאותו האובייקט. אם יש אובייקט שיש לו תחתיו עץ מוצבעים (אובייקטים שהוא מצביע עליהם), ברגע שאותו אובייקט משוחרר, נגיד כשאין לו יותר מצביעים עליו (המונה שלו התאפס), אז צריך לעבור על כל התת עץ שלו ולהקטין בהתאם את המונה שלהם, במקרים מסויימים זה יכול לקחת יחסית הרבה זמן.
- בעיה נוספת היא מעגל של הצבעות, אובייקט 1 מצביע על אובייקט 2 שמצביע על אובייקט 3 שמצביע חזרה לאובייקט 1, ועל אובייקט 1 מצביע עוד אובייקט מבחוץ (הדרך היחידה להגיע למעגל הזה) אז כשננתק את ההצבעה החיצונית, המונה לא התאפס, ולכן יש זליגת זכרון כי נותר מעגל שאין דרך להגיע אליו, זהו אחד החסרונות הגדולים של השימוש במונה.

Mark and sweep

- שיטת עבודה אחת של *garbage collector*. לכל אובייקט יש שדה (סיבית אחת) אשר מסמנת האם ניתן להגיע לאובייקט מהתוכנית או לא, אם הסיבית היא 1 אז ניתן להגיע אליו וה-*garbage collector* לא ישחרר אותו, אם היא 0 אז הוא יעשה לו *sweep*.
- ה-*GC* סורק את האובייקט ומוציא את כל ה-*references* המאוחסנים בו.

*Root set* ב-*java*, האובייקט שממנו מתחילים את החיפוש של "ההצבעות", שכולל משתנים מקומיים של המתודות שפעילות עדיין ומשתנים סטטים – היות ולמשתנים סטטיים תמיד (כמעט) ניתן לגשת. ה-*root set* מיוצג כעץ וה-*GC* עובר על העץ הזה בעזרת *BFS* ומסמן את האובייקטים שניתן להגיע אליהם – שלב ה-*mark*. לאחר מכן שהוא סיים את הסריקה הוא עושה סריקה לכל ה-*heap* שמחזיק את המשתנים של התוכנית ועבור אותם האובייקטים שהסיבית שלהם היא 0 (כלומר אי אפשר לגשת אליהם) הוא מפנה אותם ועבור אלו שהסיבית שלהם 1, הוא הופך אותה ל0 עבור הסריקה הבאה – זה השלב של ה-*sweep*. בסריקה הבאה הוא שוב יסרוק את העץ ואם ניתן עדיין להגיע לאותו האובייקט שסומן בסיבית 0 בסבב הקודם, הסיבית שלו תשתנה ל1 שוב, וכך חוזר חלילה. הסריקה של ה-*heap* היא החסרון הגדול של השיטה.

## קומפילציה

### Baker – 4 רשימות של אובייקטים

- **Unscanned** – אובייקטים שגילינו שניתן להגיע אליהם אך טרם נסרקו.
- **Unreached** – לא ניתן להגיע אליהם ככל הידוע (בהתחלה כל ה-allocated objects ברשימה *or*).
- **Free list**
- **Scanned**

היתרון של baker הוא שאין צורך לסרוק את כל ה-heap כדי לעשות sweep.

בסוף הסריקה של ה-root set כל מי שנשאר ברשימה של ה-unreached הוא אובייקט שלא ניתן להגיע אליו ולכן הוא עובר ל-free list.

Free = free U unreached

לאחר מכן לסריקה הבאה מחליפים את הכתובת של הרשומה של ה-unreached לכתובת של ה-scanned וככה בעצם במקום לשנות את הסיבית של כל אובייקט, ה-GC ידע לסרוק את הרשימה של ה-scanned מהסבב הקודם וזאת ניתן לעשות ב- $O(1)$  כיוון שזה רק שינוי כתובת.

### mark and compact

לאחר הסריקה של ה-root set, את כל האובייקטים שהם לא זבל, הוא מקבץ אותם למקום כלשהו בזכרון (compact). כלומר קובעים לכל אובייקט כתובת חדשה ורושמים אותה בטבלה הממפה כתובות ישנות לכתובות חדשות, ולאחר מכן מזיזים את האובייקטים למקומם החדש, ההזזה נעשית תוך כדי סריקה של הזכרון מהכתובות הנמוכות לגבוהות.

### Cheney copy collector

בשיטה הזו מנהלים טבלה שמחזיקה 2 מצביעים בעמודת ה-TO, האובייקטים שכתובים לפני המצביע unscanned הם אותם אובייקטים שנסרקו. מה שנמצא בין unscanned ובין free הם אובייקטים שלא נסרקו וגם אובייקטים שצריך אולי לשחרר אותם בסבב הבא. כשמוסיפים אובייקט שצריך לשחרר המצביע של free מקודם וכך בעצם אוטומטית אותו אובייקט מתווסף לסריקה של הסבב הבא. כשמסיימים לסרוק אובייקט מזיזים את המצביע unscanned ככה שאותו אובייקט מתווסף לרשימה של ה-scanned. כשהמצביעים unscanned ו-free שווים זה אומר שהסריקה הסתיימה.

### Basic block graph

קומפילטורים בונים מהקוד ביניים, גרף בלוקים, כאשר כל בלוק מכיל קטע קוד ביניים אשר אין בתוכו – באמצע, קפיצות לתוויות חיצוניות, כלומר כל בלוק מתבצע בשלמותו, וקפיצה לתוויות אחרת, או תנאי עם פיצול יהיה בסוף הבלוק. הקומפילטור בונה את הגרף הזה עבור אופטימיזציה – שיפור של קוד הביניים. את הגרף בונים בנפרד לכל מתודה.

ישנם כמה סוגי אופטימיזציה:

- אופטימיזציה לוקאלית – עובדת על בלוק יחיד בגרף.
- אופטימיזציה גלובאלית – עובדת על כל גרף הבלוקים.

## קומפילציה

Copy propogation - אופטימיזציה שבה הקומפיילר משנה משהו בקוד הביניים, וזה גורר שיפור נוסף שניתן לעשות.

available variables – משתנים זמינים, כלומר משתנה שמחזיק ערך כלשהו כך שבמקום לעשות חישוב חוזר של אותו הערך, ניתן להחליף את החישוב במשתנה הזמין.  $A=8+5, b = 8+5 \Rightarrow b = a$ .

Live variables – משתנה חי בנקודה מסוימת בתוכנית אם קיימת אפשרות שיהיה שימוש בערכו הנוכחי בהמשך התוכנית. אנליזה של משתנים חיים מתבצעת על בלוק יחיד (בד"כ מסוף הבלוק להתחלה). עוברים על הבלוק מהסוף ורואים באילו משתנים משתמשים לצורך חישוב או השמה לתוך משתנה אחר, כלומר עד לנקודה הנוכחית אנחנו צריכים את המשתנים (החיים). אם דורסים משתנה (עושים לתוכו השמה) אז עד לאותה השורה בקוד אותו המשתנה שלתוכו עושים השמה לא חי, כי הוא גם ככה נדרס בשורה הזאת, אבל לאחר השורה הזאת הוא הופך להיות שוב רלוונטי (לא בכל המקרים זה ככה). לאחר ביצוע האנליזה, עוברים מתחילת הבלוק לסופו, ורואים האם המשתנה שלתוכו עושים השמה, חי לאחר ההשמה, אם לא אפשר למחוק את השורה של ההשמה, כלומר מוחקים קוד מת.

ניתן להסיק מזה שכמה משתנים חיים באותה נק' זמן, שאסור להקצות למשתנים האלה את אותו הרגיסטר, כלומר כל אחד אמור להיות ברגיסטר משלו, היות וניתן לעשות מיחזור רגיסטרים עבור משתנים חיים.

## גישה של קומפיילרים להקצאות רגיסטרים עבור שפת אסמבלי – chaitin's algorithm

לאחזר האנליזה של המשתנים החיים, עוברים על הקוד מהתחלה, ועבור כל נק' זמן שבה יש כמה משתנים חיים ביחד, מציירים גרף מלא עבורם. לאחר שציירנו את הגרף עבור כל המצבים, צובעים את הצמתים כך שלא יהיה אותו הצבע בקודקודים סמוכים. כמות הצבעים היא לפי כמות הרגיסטרים הזמינים. נגיד שיש 4 רגיסטרים, אז יש 4 צבעים שונים. צביעת הגרף תתבצע ברקורסיה, נוציא בכל צעד קודקוד אשר יש לו לכל היותר 3 שכנים, זאת כיוון שכשנוציא את כל הקודקודים, נצבע ברקורסיה, ולכן אם נחזיר קודקוד שיש לו 4 שכנים, והם כבר צבועים, לא נוכל לצבוע את הקודקוד הנוכחי שהחזרנו. נמשיך להוציא קודקודים ולצבוע אותם (נגיד לפי סדר הצבעים). במצב שבו יש קודקוד עם 4 שכנים ואין ברירה אלא להוציא קודקוד כזה, כשנחזיר אותו הערך שלו ישמר בזכרון – spilled. ישנם מצבים שבהם בהחזרת הקודקודים למרות שלקודקוד מסויים יש הרבה שכנים (כמספר הרגיסטרים או יותר), הצביעה תתבצע כך שלא יהיו 2 שכנים בעלי אותו צבע.

## JVM

הקומפיילר של ג'אווה יוצר מקבצי java. קבצים עם סיומת class שהם כתובים ב-byte-code. הקבצים האלו נכנסים ל-JVM. בנוסף קיים קומפיילר – JIT – just in time Compiler, שהוא חלק מה-JVM ומייצר את הקוד בשפת מכונה, הוא קומפיילר חכם שעוקב אחרי הנעשה בקוד.